CoHLA: Rapid Co-simulation Construction

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college van decanen
in het openbaar te verdedigen

op
vrijdag 31 januari 2020
om
11:00 uur precies

door

Thomas Christian Nägele

geboren op 8 april 1992
te Nijmegen

Promotor:

Prof. dr. J.J.M. Hooman

Manuscriptcommissie:

Prof. dr. M.I.A. Stoelinga
Prof. dr. ir. J.P.M. Voeten (Technische Universiteit Eindhoven)
Dr. ir. B.D. Theelen (Océ Technologies)

Het werk in dit proefschrift is uitgevoerd onder auspiciën van de onderzoeksschool IPA (Instituut voor Programmatuurkunde en Algoritmiek).

IPA Dissertation Series 2020-02

Dit onderzoek werd mede mogelijk gemaakt door het NWO project 12701: Robust Motion Control.

Omslag: Thomas Nägele
Druk: GVO drukkers & vormgevers

ISBN: 978-94-6332-598-1

CoHLA: Rapid Co-simulation Construction

DOCTORAL THESIS

to obtain the degree of doctor
from Radboud University Nijmegen
on the authority of the Rector Magnificus prof. dr. J.H.J.M. van Krieken,
according to the decision of the Council of Deans
to be defended in public

on
Friday, January 31, 2020
at
11:00 a.m.

by

Thomas Christian Nägele

born on April 8, 1992
in Nijmegen, the Netherlands

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA Dissertation Series 2020-02

This research was supported by the NWO, project 12701: Robust Motion Control.

Cover: Thomas Nägele
Print: GVO drukkers & vormgevers

ISBN: 978-94-6332-598-1

Contents

CHAPTER 1

Introduction

In this chapter, Section 1.1 describes the problem that is addressed in this dissertation. Section 1.2 introduces some of the terminology and Section 1.3 explains the goal of this work. The challenges for reaching this goal are described in Section 1.4. Section 1.5 briefly introduces the industrial context of the work described in this dissertation. Section 1.6 highlights related works and Section 1.7 explains the contribution of this work. The structure of the remainder of the dissertation is described in Section 1.8.

1.1 Problem statement

Modern vehicles, production robots and smart energy grids are just a couple of examples of cyber-physical systems (CPSs) [46]. A CPS is a system that involves both the physical and cyber domain. Since most systems that were developed in the last few decades are controlled by software, CPSs are becoming more and more important in both industry and everyday life.

The development of a CPS, however, is a complex multidisciplinary process. Specific subsystems are developed by separate development groups, all having their own field of expertise. These disciplines have their own development methods, tools and workflows. Nevertheless, the subsystems that are designed by these development groups should work together as one system in the end.

In a model-based design approach, different disciplines create models during the development of a system. Since, for example, mechanical engineers use different modelling techniques than thermal engineers, these models are developed using formal languages and modelling tools that meet the needs of the discipline. Model-based collaboration of the different disciplines during the development is hard due to the use of these different modelling techniques.

While most of the models can be simulated to determine and verify their behaviour, it is complex to simulate the models of different components and disciplines together. This complexity is caused by the different domains that are being modelled, their different notions of time and their differences in their computational model. Some models – such as state machines – are event-based models that change their state when specific events occur. Other models are continuous-time models, where the state of the model should be calculated for a requested point in time by a mathematical solver. There are also discrete-time models that only change their state at specific time intervals.

Although it is hard to simulate these models together, the creation of a virtual prototype of the system by co-simulating the different models could provide great insight in the functioning of the system. Such a prototype would also allow for design space exploration to compare different designs relatively quickly. Having this, it could help in making design decisions based on simulation results and early validation of system requirements. It also allows for testing the system virtually in an early stage of the development, which could reduce the costs of making changes and shorten the time to market [9, 85, 72]. These tests can either focus on the *happy flow* behaviour of the system [3] or on the robustness of the system by intentionally providing faulty inputs. *Happy flow* focuses on the behaviour of the system when provided with correct inputs. By injecting faults into the system, it can be tested whether it properly handles faulty input. A virtual prototype of the system is very useful for both types of behavioural tests, as it minimises the need for actual hardware and no physical components can be broken when testing.

1.2 Terminology

This section outlines some of the terminology that is used throughout the dissertation. The following term definitions will be used in this work.

Component A component is an element that has a specific function. This is often an elementary function that cannot be separated into multiple components. Components provide interfaces to interact with other components. A component can be replaced by another component that has the same functionality with a different implementation [38]. Compound components are considered to be a set of components that tightly collaborate to perform a specific function.

System A system is a set of at least two components that are connected to each other [2]. The components collaborate with each other to provide specific functionality. For example, a heating system is designed to maintain the temperature in a specific area or building and consists of heaters and thermostats.

Cyber-physical system A cyber-physical system (CPS) is a system that integrates computation with physical processes [46]. Such systems typically consist of (embedded) control systems – software – and hardware components, such as mechanical components. The hardware components are controlled by the software components. Examples of cyber-physical systems are robots, cars and drones.

Component model A component model is a formal representation of a (compound) component, following the definition of functional models in [25]. A model is an abstract representation of reality, which may be for the sake of simplicity. Since models can be used to validate and verify system or component properties, they are often used during the development of a system. Some models can be simulated to estimate the behaviour of the component. For example, a model can be used to verify whether a motor design is capable to pull certain loads or not. Simulation can be used to also calculate the trajectory of the movement of the load.

Simulation Simulation is a technique in which the computer imitates a real-world process to evaluate one or more properties of this process [70]. In this dissertation, these processes are components of a system, for which mathematical models have been made. The computer calculates the state of each of these components repeatedly to imitate the behaviour of the component as it would be in the real world. The approach to calculate the model's state step by step is called simulation. The results from the simulation can be used for further analysis.

Co-simulation Co-simulation is a technique to synchronise and simulate models of different representations in their individual runtime environments – simulators [62]. These simulators simulate different types of models, such as mechanical models or thermal models. These model simulations are connected to each other to enable the exchange of data. In a co-simulation, the local simulation time of each of the simulations should be synchronised with the other simulations in order to ensure proper collaboration of the component models. By simulating the models in a synchronised manner, a co-simulation can be used to estimate the behaviour of a system.

Design space exploration Design space exploration (DSE) is a method to systematically analyse different designs of the system under design by changing the system's parameters [57]. DSE is used to compare different design alternatives to find and select a design that best fits the desired characteristics or requirements. The approach is also useful during the design of the system to eliminate the least promising design paths and to proceed in the most promising design path to improve design speed.

Hardware-in-the-loop simulation Hardware-in-the-loop (HIL) simulation is a technique where one or more hardware components of the envisioned system under design are added to a system simulation [37]. This is typically done for control systems to test new control sequences. For the simulation or implementation of the control loop it is required that it is capable to run in real-time.

Software-in-the-loop simulation Software-in-the-loop (SIL) simulation is a technique where an implementation of (control) software is executed together with a simulation of the other components of the system [86]. SIL enables the developer to test software before hardware becomes available in the development process.

SIL is also useful for testing the software without risking to damage the physical system.

Domain-specific language A domain-specific language (DSL) is a language that is designed for a specific application [71, 80, 41, 28]. A DSL allows the developer to use syntax and semantics that are specific for its domain, which should simplify the development compared to using general-purpose languages. Frameworks such as Xtext[1] [22] and Jetbrains MPS[2] [83] have been developed for the development of DSLs.

Abbreviations A number of abbreviations is used throughout the dissertation. The most commonly used abbreviations and their meaning are listed in Appendix A.

1.3 Goal

As mentioned in Section 1.1, it is very useful to be able to simulate models of different components together during the development phase. Our goal is to support the multidisciplinary development of CPSs by enabling the rapid construction of co-simulations in an early stage of the development process. This co-simulation functions as a virtual prototype of the system that can be used to verify system requirements and proper integration of the components. It provides early insight and supports concurrent development of different components and disciplines. The importance of using co-simulation for system design has been addressed in [65].

The approach being presented in this dissertation should meet a number of requirements. The requirements are divided into three different types of requirements. First, a number of technical requirements regarding the approach itself are explained. Secondly, there are three functional requirements for the approach. Finally, three usability requirements are formulated. All requirements are grouped by type and briefly explained in the following sections.

Technical requirements

1. **A co-simulation of simulation models of different disciplines can be constructed fast (20 models within 1 day).**
   To support the multidisciplinary development, the approach should minimise the overhead of the construction of a co-simulation of a set of simulation models. The approach should therefor enable the user to construct a co-simulation of 20 already existing models in less than a day's work.

2. **Changes in either the models or the interfaces connecting these models can be adapted quickly using the approach (5 models within 1 hour).**
   Once the co-simulation has been constructed, changes of the models and their interfaces should have minimal impact on the co-simulation construction. To

[1]`https://www.eclipse.org/Xtext/`
[2]`https://www.jetbrains.com/mps/`

be able to quickly incorporate such changes in the co-simulation, changeability of the co-simulation specification is important. The approach should therefor enable the user to change the interfaces of five different models in the co-simulation within an hour.

3. **Simulation models from multiple tools are supported: at least 10 modelling tools.**
   Since modellers should be allowed to use different tooling to create their models with, models created with different tools should be supported. The approach should therefor support models from at least ten different modelling tools.

4. **The approach is easily extendable to support new tools.**
   The approach should be able to be rapidly extended to support new modelling tools or other external applications. It should be able to add support for new tools within a week of development.

5. **The co-simulation can be executed in a distributed manner to provide scalability.**
   To be able to deal with very large co-simulations or compute-intensive simulations, being able to distribute the co-simulation over a number of computation nodes would be helpful to improve the simulation speed. The approach should therefor support the individual simulations to be distributed over different nodes to make use of more computational power.

6. **The simulation results are trustworthy.**
   The co-simulations should be trustworthy to be able to base design decisions solely on simulation results. The approach must therefor yield trustworthy results of the co-simulation.

Functional requirements

7. **The approach has logging capabilities for analysis afterwards.**
   Performing analyses during the simulation or after its execution are key of the co-simulation. The approach should therefor produce a log from a co-simulation execution.

8. **The approach has support for automated design space exploration.**
   To compare different design parameters with each other, the approach should support the automated execution of a series of co-simulations having slightly different parameter configurations. Support for DSE allows the user for example to setup nightly runs to compare design alternatives.

9. **The approach has support for fault injection.**
   Fault injection should be supported by the approach to analyse the robustness of the system under design. Since faults could be injected in different locations within a co-simulation, the intended faults for this requirement focus on the communication layer between the components.

Usability requirements

10. **The framework is easy to maintain and extendable.**
    Since the approach is likely to interface with many different applications, it might become difficult to maintain its source code. Extending requirement 4, the framework to be designed should also keep in mind that the supported tools might change over time, requiring maintenance of the framework. This maintenance should not be too time consuming.

11. **The framework is documented properly.**
    In order to be able to work with the approach, it should be sufficiently documented.

12. **The framework runs on Windows, Linux and Mac.**
    Since our aim is to support many different modelling tools, the framework should also work on multiple different platforms to allow the users to use their system of choice.

1.4 Challenges

It is challenging to create a co-simulation of models from different disciplines. Models in different domains are developed in different tools and may have different timing behaviour. Models that abstract from real-world artefacts or processes are usually modelled in a continuous-time fashion, while control models and software models are typically discrete-time models. Some models mainly communicate using channels to transmit their attribute values to other models, while others use events to communicate with each other. Creating one coherent co-simulation from those models is not trivial.

Modelling tools sometimes provide support for importing models created in different tools. These models can then be connected and simulated together within one of the tools that created the model. This approach, however, is limited, as it depends on the support of one tool vendor. When one of the tools changes its modelling language or simulation method, importing a model might not be possible anymore. Also, the set of supported modelling tools is usually rather small. For a large system with many different disciplines collaborating, chances are small that all models will work together. The process of integrating the models into one tool can also be very time consuming, especially if the models or their interfaces change frequently.

More generic co-simulation frameworks have also been developed. Most of these frameworks require the user to write connectors to connect with the simulator for the models. Even though this approach potentially supports many different simulators to be co-simulated, a connector needs to be developed for each simulator. To properly connect all different types of models to each other, a wrapper needs to be developed for every model. This wrapper is capable of connecting to the model's simulator and interacting with the co-simulation framework for synchronisation. As stated before, the models could change frequently during the design process, making this approach very time consuming. The challenge is to make this fast and easy.

1.5 Industrial context

The work described in this dissertation was carried out in both an academic and an industrial context. A number of industrial partners were involved in the project and two case studies have been conducted in collaboration with these partners. An X-ray diffractometer that is designed by Malvern Panalytical[3] is one of the drivers during the development of the methodology and therefore a major case study throughout the entire project. This case study is described in Section 6.1.

In previous projects, ESI (TNO)[4] has conducted research to a smart indoor lighting system. The smart lighting system consists of a large number of relatively simple components, which makes the system suitable for analysing the scalability of the approach. For this purpose, a similar system was designed using our approach under development. Section 7.1 describes this case study.

Other industrial partners that participated in the project were Océ Technologies[5] and Controllab[6]. Even though no case studies were conducted in collaboration with these partners, they also provided input to the project.

1.6 Related work

The related works are structured in three subsections. Section 1.6.1 provides a brief historic overview of model-based development. Techniques for co-simulation are discussed in Section 1.6.2 and Section 1.6.3 describes a number of frameworks using the High Level Architecture. Section 1.6.4 introduces the INTO-CPS project. Section 1.6.5 provides a brief comparison of existing frameworks with respect to the requirements described in Section 1.3.

1.6.1 Model-based development

The concept of computer-aided software engineering (CASE) to support the design and implementation of software was already proposed in the 1980s [15]. Software supporting the CASE approach provides tools for all stages in the software development life-cycle, such as analysis tools, development tools and validation tools. Different meta-level languages were developed for each of these tools. These languages can be considered as meta-models describing different aspects of the software being developed, such as a structured notation for the requirements. CASE evolved into model-driven engineering (MDE) [63]. The model-driven architecture (MDA) [42] initiative that was launched by the Object Management Group (OMG)[7] is one of the best known collections of standards regarding MDE. The MDA includes widely used standards such as CORBA [68] and the Unified Modeling Language (UML) [61, 38].

In [67], Sha et al. emphasise the importance of a type of system that becomes ever more important: cyber-physical systems. While MDE focused on the devel-

[3]`https://www.malvernpanalytical.com/`
[4]`https://www.esi.nl/`
[5]`https://www.oce.com/`
[6]`https://www.controllab.nl/`
[7]`https://www.omg.org/`

opment of software at first, the growing importance of CPSs causes the need to combine MDE for software with model-based approaches for other disciplines such as mechanical engineering. The use of both physical models and software models for the development of CPSs was proposed in [40].

1.6.2 Co-simulation

The Functional Mock-up Interface (FMI) standard[8] was developed as part of the MODELISAR project[9] for the exchange and co-simulation of models in different domains that are created in different modelling tools [7]. Modelling tools supporting the FMI standard allow models to be exported to Functional Mock-up Units (FMUs) or to import FMUs from other tools. This enables the integration of different domains and modelling tools. The FMI standard is supported by over 100 tools. Section 3.1 provides more details on the FMI standard.

In the early 90s, the Ptolemy project was developed to connect different heterogeneous models to each other and to simulate them together [13]. The approach was primarily focused on discrete-event simulations and finite-state machine models, which made the framework less suitable for the co-simulation of CPSs, as these also include continuous-time models of physical components. Support for such models was added by Ptomely II [20, 47, 58]. The new version of the framework also supports the integration of discrete-event models into the continuous-time domain. Even though the Ptolemy II project provides tools to create models and connect them to each other, the use of models created in other tools is more difficult, as it requires a wrapper to be developed to simulate the model or to connect to the model simulator. The Ptomely II project has developed a library to incorporate models adhering the FMI standard, but this imposes a number of restrictions and requirements to the FMU that is used [12].

Another early attempt to simulate two models from different tools and disciplines together is found in [36]. Here, models created in the UML based CASE tool Rose RealTime and Simulink[10] are co-simulated. Due to the lack of a proper notion of time in Rose RealTime, the UML models needed to be extended to be simulated together with another model. The approach shows how the two models are simulated together using the simulation time of Simulink, but only provides a proof of concept that requires some extensions to the UML model.

OpenMETA[11] [73] is a tool chain for model- and component-based design of CPSs. A number of widely used modelling tools is supported by the tool chain, including Simulink and OpenModelica[12]. The goal of OpenMETA is to represent multi-domain models as one unified design model. OpenMETA strongly focuses on design space exploration [53], but does not support standards such as the FMI standard.

DESTECS[13] (Design Support and Tooling for Embedded Control Software) was a project working on collaborative modelling and co-simulation for the de-

[8]https://fmi-standard.org/

[9]https://itea3.org/project/modelisar.html

[10]https://www.mathworks.com/products/simulink.html

[11]https://openmeta.metamorphsoftware.com/

[12]https://openmodelica.org/

[13]http://www.destecs.org/

velopment of real-time embedded control systems [55, 54]. Using the DESTECS framework, a co-model was created from models from different disciplines. The co-model can be simulated to obtain results from the modelled system. DESTECS is focused on building fault-tolerant systems. The project was followed up by the INTO-CPS project, which focuses more on the co-simulation of separate models. This project is described in more detail in Section 1.6.4.

1.6.3 High Level Architecture frameworks

The High Level Architecture (HLA) was developed by the US Department of Defense as common architecture specification for all of its classes of simulations [16]. The HLA definition includes rules for the simulations, an interface specification and an object model. HLA covers, among others, data distribution management and time management [32] and is used to co-simulate multiple models of different domains. A Run-Time Infrastructure (RTI) provides the necessary services to all simulation models for time and data synchronisation. In the year 2000, HLA was standardised by IEEE as IEEE 1516-2000, which evolved to IEEE 1516-2010 in the year 2010 [1]. More details on the standard are provided in Section 3.2.

SimGE[14] (Simulation Generation) [78] has an object model editor for the HLA standard. The tool can be used to generate the required configuration files as well as base implementations for the specified simulation models according to the HLA standard. The simulation models need to be specified as object models, for which skeleton code is generated to be used with a supported RTI. The skeleton code needs to be completed manually to communicate with a simulator that executes the model.

The use of FMU component models in an HLA co-simulation was demonstrated in [52]. Here, the C2WT (Command and Control Wind Tunnel) environment [35] is used to automatically wrap FMUs to simulate them together using an RTI that implements the HLA standard. The approach requires rather extensive meta-modelling in order to run a co-simulation of a set of FMUs. Even though the C2WT does not run on all platforms, support for distributed simulation execution is implemented.

To simplify the development of HLA federates, the HLA Development Kit Framework (DKF)[15] [23, 24] was developed. Several RTIs are supported by the framework, but it requires the user to develop the models in Java or to write a Java connector for each model. The incorporation of FMI in the DKF was discussed in [33]. The MONADS (Model-driven architecture for distributed simulation) tool chain [8] uses the DKF to construct an HLA-based co-simulation from a set of SysML models [34]. This method abstracts from the co-simulation tool – or HLA implementation in this case – that has been used.

The US National Institute of Standards and Technology (NIST) aims for an open set of connectors for creating an HLA co-simulation from a set of heterogeneous models. The UCEF (Universal CPS Environment for Federation) toolkit[16] [14] is a joint development of NIST and the Institute for Software In-

[14]`https://sites.google.com/site/okantopcu/simge`
[15]`https://smash-lab.github.io/HLA-Development-Kit/`
[16]`https://pages.nist.gov/ucef/`

tegrated Systems at Vanderbilt University [60] that aims for HLA co-simulation of CPSs using the HLA standard. The toolkit allows the user to specify a system and generate code to run the models. It is being actively developed since 2018.

1.6.4 INTO-CPS

The Integrated Tool Chain for Model-based Design of Cyber-Physical Systems (INTO-CPS) [44, 45] is a framework to support model-based design of CPSs. The tool chain also aims for improving the multidisciplinary design process of CPSs from the specification of the requirements to the realisation. Since the INTO-CPS framework enables traceability during the developed process, it plays a role in all stages of the development.

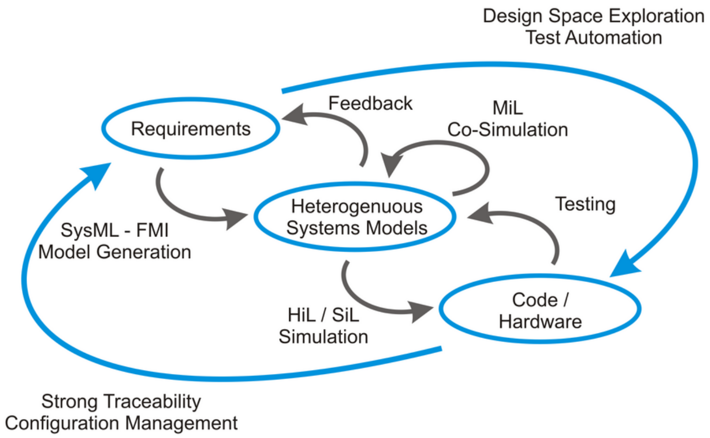


Figure 1.1: *INTO-CPS framework support throughout the development of a CPS. Figure from [44].*

The INTO-CPS framework consists of a set of tools that can be used to collaboratively develop a CPS. The modelling tools used by INTO-CPS all support the FMI standard [7], which is described in Section 3.1. Figure 1.1 displays how the framework supports the development of a CPS during the three main development phases: requirements definition, system design (using models) and realisation. SysML [29] is used to formalise the requirements, after which model interfaces are generated that act as stub models for the initial model design. These models can be developed further to come to a virtual prototype that can be simulated. Then, the models can be realised into hardware and software, which can be run using HIL and SIL simulations. During all stages of the development, the initial requirements in SysML form the base upon which the system is designed.

Similar to our goal, INTO-CPS also allows models to be co-simulated together to gain insight in the behaviour of the system and to support design decisions [26]. Figure 1.2 shows some of the tools that are part of the INTO-CPS tool chain and how they are related to each other. Modelio[17] is used to define system requirements using SysML. From SysML, a configuration for the INTO-CPS Application is

[17] `https://www.modelio.org/`

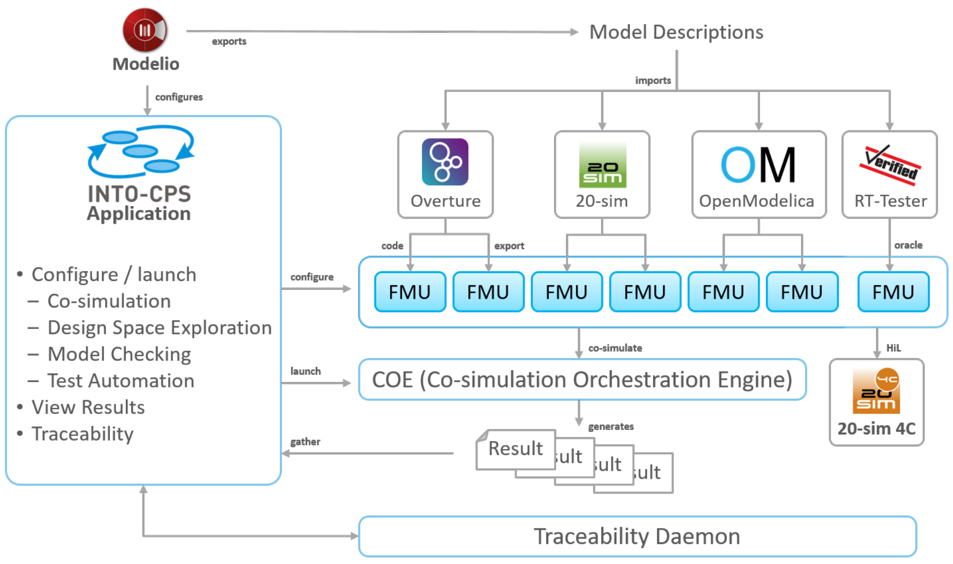


Figure 1.2: *INTO-CPS tool support and their relations. Downloaded on May 27th 2019 from https://cordis.europa.eu/project/rcn/194142/reporting/en.*

created and model descriptions in terms of interfaces are exported. These models must be created according to their interfaces by any of the tools, after which these can be exported to FMUs. These FMUs can be co-simulated using the Co-simulation Orchestration Engine (COE) [77, 76]. The results from the co-simulation can be viewed in the INTO-CPS Application and used as input during the design process.

Whereas the focus of the INTO-CPS framework lies on traceability and the development of a CPS from requirements to realisation, our focus is more on the rapid construction of a co-simulation from a set of simulations. Our co-simulation approach aims for supporting existing model-based system development by increasing the usability of the component models being developed, allowing the user to create a co-simulation from them while keeping the overhead of creating this co-simulation as low as possible.

1.6.5 Framework comparison

Although a number of existing tools and frameworks have been described in this section, none of these meets all of our requirements as stated in Section 1.3. Table 1.1 displays whether the requirements are met for a number of frameworks. A requirement that is met is marked as a +, while an unmet requirement is marked as –. Requirements that are partly met are marked ±. Note that the table is based on available documentation and web pages of the frameworks, not on our own experiments and findings.

Requirement	**Ptolemy II**	**OpenMETA**	**SimGE**	**HLA-DKF**	**INTO-CPS**
1 (fast construction)	±	+	−	±	+
2 (quickly adapting)	±	+	−	±	+
3 (tool support)	−	±	−	−	+
4 (extendability)	−	+	+	±	±
5 (distribution)	−	−	+	+	−
6 (trustworthiness)	+	+	±	±	+
7 (logging)	+	+	−	±	+
8 (DSE)	−	+	−	−	+
9 (fault injection)	−	−	−	−	−
10 (maintainability)	±	+	±	±	±
11 (documentation)	+	+	±	−	+
12 (multi-platform)	+	−	−	+	+

Table 1.1: *Requirement comparison between existing frameworks and tools that support the construction of co-simulations of CPSs. Requirements can either be met (+), partly met (±) or not met (–).*

The table shows that HLA-based tools (SimGE and HLA-DKF) support distributed execution while the other frameworks provide better tool support. Most frameworks appear to be trustworthy and support logging and allow for maintenance. By developing an HLA-based co-simulation framework that uses the FMI standard to support a wide range of modelling tools, our aim is to combine the best of both types of frameworks.

1.7 Contribution

This dissertation provides a research design of tooling for rapidly constructing and running co-simulations of component models to support the design of CPSs. In particular, we have developed a framework called CoHLA. CoHLA uses the HLA and FMI standards for the co-simulation execution. The main contributions of this dissertation are related to these techniques. The most notable contributions are listed below.

- An approach to use the HLA standard as running algorithm for models compliant to the FMI standard. The approach is embedded in CoHLA. The contributions of CoHLA itself are highlighted in more detail in the next section.

- In addition to a number of experimental systems, case studies were conducted at industrial partners to validate the methodology.

- Experiments were conducted to analyse the trustworthiness of HLA-based co-simulations. The timing behaviour of HLA-based co-simulation is analysed and the co-simulation results are compared to those resulting from a single simulator and from another co-simulation framework.
- Experiments were conducted to analyse the scalability of HLA-based co-simulations. An approach to improve the co-simulation performance is given by means of distributing the co-simulation execution over a set of computation nodes.
- Different approaches to incorporate POOSL models in an HLA-based co-simulation are discussed and experimented with.

CoHLA

To rapidly construct a co-simulation from a set of simulation models, a domain-specific language (DSL) called CoHLA was developed. The language allows the user to specify the interfaces of the simulation models and how these are connected to each other. Source code for the co-simulation of the models is automatically generated from such a specification to reduce the effort of implementing the co-simulation. The CoHLA framework includes the following functionality.

- Support for the construction of a co-simulation consisting of POOSL models and models adhering to the Functional Mock-up Interface (FMI) standard. These types of models are described in more detail in Sections 2.2.2 and 3.1 respectively.
- Specification and application of reusable model configurations to quickly add variations to the models being simulated.
- Logging capabilities to record certain attribute values during co-simulation and the automatic calculation of a number of basic performance metrics.
- Specification and replay during co-simulation of scenarios and fault scenarios to mimic user input and the occurrence of communication faults.
- A collision detector that uses the 3D drawings of the physical components of the system to detect possible collisions between the components.
- A basic form of automatic design space exploration for the system under design. The implementation runs the specified parameter sets automatically to minimise the user interaction required for comparing design alternatives.

An analysis was conducted on the timing behaviour of HLA-based co-simulations using the CoHLA framework. This analysis resulted in a number of changes to the simulation execution of the individual models. By comparing co-simulation results of the HLA-based co-simulation with an integrated modelling tool simulator and a similar co-simulation framework, the co-simulation results of our simulation were shown to be trustworthy.

The scalability of HLA-based co-simulations was tested by a series of experiments with large numbers of simulations representing a smart lighting system. The experiments focused on the execution aspect as well as the specification aspect of dealing with large numbers of simulators in a co-simulation. The execution time of the large co-simulations scaled well when distributed over a number of computation nodes. An approach to use a DSL – called the Lighting DSL – for the specification of a lighting system for the generation of a CoHLA system specification is described. The co-simulation is then generated from this system specification.

Finally, co-simulation approaches for POOSL are discussed and a separate DSL – called the Connector DSL – for the generation of connector applications for SIL simulation was developed.

List of publications

The content of this dissertation is mainly based on the following peer reviewed publications.

P.1 T. Nägele and J. Hooman. Co-simulation of Cyber-Physical Systems using HLA. In *Proceedings 7th IEEE Annual Computing and Communication Workshop and Conference (CCWC)*, pages 267–272, 2017.

P.2 T. Nägele and J. Hooman. Rapid Construction of Co-simulations of Cyber-Physical Systems in HLA using a DSL. In *Proceedings 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 247–251, 2017.

P.3 T. Nägele, J. Hooman, T. Broenink and J. Broenink. CoHLA: Design Space Exploration and Co-simulation Made Easy. In *Proceedings IEEE Industrial Cyber Physical Systems (ICPS)*, pages 225–231, 2018.

P.4 T. Nägele, J. Hooman and J. Sleuters. Building Distributed Co-simulations using CoHLA. In *Proceedings 21st Euromicro Conference on Digital System Design (DSD)*, pages 342–346, 2018.

P.5 T. Nägele and J. Hooman. Scalability Analysis of Cloud-based Distributed Simulations of IoT Systems using HLA. In *Proceedings 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1075–1080, 2018.

P.6 T. Nägele, T. Broenink, J. Hooman and J. Broenink. Early Analysis of Cyber-Physical Systems using Co-simulation and Multi-level Modelling. In *Proceedings IEEE Industrial Cyber Physical Systems (ICPS)*, pages 133–138, 2019.

Resources

Resources for CoHLA can be found on its website[18]. This includes a user manual, an installation manual and a VirtualBox[19] image for testing the CoHLA framework without having to install all of its dependencies.

[18]https://cohla.nl/

[19]https://www.virtualbox.org/

Sources

CoHLA, the Lighting DSL and Connector DSL are all available as open source projects. Also, the DSL definitions, models and results of all experiments described in this dissertation can be found online. The resources can be found in the following repositories.

CoHLA	`https://github.com/phpnerd/CoHLA`
Lighting DSL	`https://github.com/phpnerd/Lighting-DSL`
Connector DSL	`https://github.com/phpnerd/Connector-DSL`
Experiments	`https://github.com/phpnerd/CoHLA-projects`

1.8 Dissertation structure

The remainder of this dissertation is structured as follows.

Chapter 2: Multidisciplinary Modelling This chapter describes a sample thermostat system called RoomThermostat. This system is used throughout the dissertation as a toy case study for illustration purposes. The component models for the RoomThermostat system are described, as well as the modelling tools that were used to create them, being 20-sim, POOSL, VDM-RT and Modelica.

Chapter 3: Co-simulation This chapter explains the FMI and HLA standards that are used by CoHLA to construct a co-simulation. Additionally, details on the co-simulation of POOSL models in an HLA environment are given. The models for the RoomThermostat system are used to manually construct a proof of concept for running an HLA-based co-simulation of FMI models and POOSL models. The contents of this chapter are based on paper P.1.

Chapter 4: CoHLA This chapter introduces the CoHLA framework and elaborates on its features. A brief introduction to the language and the code that is generated is provided. The framework is used to create a co-simulation of the RoomThermostat system. A number of experiments are conducted to highlight different aspects of the CoHLA framework. The contents of this chapter are based on papers P.2 and P.3.

Chapter 5: Trustworthiness of Co-simulation Results This chapter highlights a number of aspects regarding the trustworthiness of the results of an HLA-based co-simulation. A number of experiments regarding the timing within an HLA co-simulation is conducted, which resulted in small changes to the running algorithm for the individual simulations in the co-simulation. Furthermore, the co-simulation results from a CoHLA co-simulation of the RoomThermostat system are compared to the results when running identical models in a single integrated simulator. Finally, a different case study is introduced to compare the CoHLA co-simulation results with the results of an identical co-simulation when using another co-simulation framework. The results of these experiments are used to draw a conclusion on the trustworthiness of the results of the generated CoHLA co-simulations.

Chapter 6: System Design using CoHLA This chapter starts by describing the industrial context in which a case study was conducted. Due to the confidential nature of this system, a system called the Slider Setup was designed that reflects many similar challenges compared to the industrial case. This case study serves as an example of how the intended CoHLA design flow could be used and highlights features such as metric collection and design space exploration. The contents of this chapter are based on paper P.6.

Chapter 7: Scalability This chapter describes a number of experiments regarding the scalability of CoHLA. A smart lighting system serves as a type of sample system that consists of many individual simulations that should work as one system. Challenges regarding both the specification of such a large co-simulation in CoHLA and the execution of the co-simulation are addressed. For the specification of the system, a new DSL is developed to allow the user to specify a lighting system more intuitively, from which a CoHLA specification is generated. To analyse the scalability in terms of execution time, the large co-simulations are executed in a distributed manner in the cloud. The contents of this chapter are based on paper P.4 and P.5.

Chapter 8: Conclusion This chapter concludes the dissertation by providing a brief overview of the CoHLA framework and reflecting on the requirements. Limitations of the approach as well as future work are summarised.

CHAPTER 2

Multidisciplinary Modelling

Throughout the dissertation, small examples are given to illustrate certain functionality or to conduct sample experiments. While larger case studies are explained in Chapters 6 and 7, a smaller case study is often used as an example. This chapter describes the sample system that is used for this: a thermostat system called the *RoomThermostat*. The system and its components are described in Section 2.1. Section 2.2 highlights the modelling tools that have been used to create the models for all case studies that were conducted. These modelling tools are 20-sim, POOSL, VDM-RT and Modelica. For every modelling tool, one or more example models are described that have been developed using the tool. Concluding remarks can be found in Section 2.3.

2.1 Room thermostat case study

The RoomThermostat system is introduced in Section 2.1.1. The component models used for the system are described in Section 2.1.2.

2.1.1 RoomThermostat

The RoomThermostat, which will be the name of the case, is a small heating system. It is designed to control the temperature in a building. Similar systems can be found in basically every building.

Buildings consist of rooms, each room having different characteristics. The size, height, purpose and contents of each room may be different. Domestic houses usually have a heater located in every room, while only having one thermostat to control them all. This thermostat is located in the room where most human activity is expected, which is often the living room. Based on the temperature in the living room, the heaters in all rooms are switched on and off. Some rooms, however, have windows or doors, which cause the energy loss in the room to be different from other rooms. Additionally, the heating capacity of the heaters may be different from one room to another. All these factors affect the temperatures in the different rooms and therefore affect the level of comfort as experienced by the users.

Hence, the RoomThermostat is a system that allows for many variations and still is simple enough to understand quickly. Due to these variations in the system, the heating behaviour of the system is hard to predict. To estimate this behaviour beforehand, a co-simulation of the system could be used.

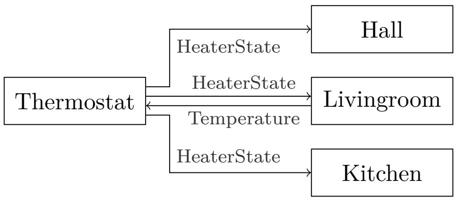


Figure 2.1: *Architecture of the default RoomThermostat configuration.*

The RoomThermostat system consists of one or more thermostats and at least as many rooms. When the number of rooms is lower than the number of thermostats in the system, at least one thermostat is not connected to a room, which causes these to have no function in the system. Since most domestic houses have one thermostat and multiple rooms, this will be the default configuration for the RoomThermostat system. However, to reduce the size of the co-simulation, only three rooms will be simulated: the living room, hall and kitchen. Of these rooms, only the output temperature of the living room will be used as input for the thermostat. The heaters in all rooms are toggled by the thermostat. Figure 2.1 shows this default system configuration.

2.1.2 Models

To construct such a co-simulation, the components of the RoomThermostat should be modelled. Together, the models create the RoomThermostat system. The system requires a target temperature input to be specified for the thermostat and each of the room models has a temperature attribute as output that can be inspected. An actor that is not part of the RoomThermostat system specifies the input target temperature(s). The system consists of one or more models of a thermostat and one or more models of rooms. These thermostats and rooms are connected to each other. The goal of the co-simulation is to determine the heating behaviour for specific configurations for the thermostats, heaters and rooms.

Room

Instead of creating separate models for the heaters and the rooms in which these are located, one compound model of a room is created. The model includes parameters to change the surface area and height of the room, the size of the window and the heating capacity of the heater that is located in the room. These parameters are displayed in Table 2.1.

Room – Parameters		
Name	**Type**	**Description**
RadiatorSize	`real`	The size of radiator (m^2).
Surface	`real`	The surface area of the room (m^2).
Height	`real`	The height of the room (m).
WindowSize	`real`	The size of the window (m^2).
InitTemp	`real`	The initial temperature in the room (℃).

Table 2.1: *Parameters of the room model.*

The room model calculates both the energy losses through the window and the energy gain from the heater and uses these values to compute the inside temperature of the room. Table 2.2 displays the input and output attributes for the model and their types.

Room – Attributes			
In/Out	**Name**	**Type**	**Description**
Input	HeaterState	`boolean`	The state of the heater: on is represented by `true`, off by `false`.
Output	Temperature	`real`	The current temperature in the room (℃).

Table 2.2: *Input and output attributes of the room model.*

The room abstracts from reality by simplifying both the room and the heater. For example, radiator heaters can usually be tuned by the user by opening or closing the valve that allows hot water getting in. This allows the heater to be turned on for only 30 or 50 percent instead of only being turned on or off. The same holds for modern boilers, which are also capable of heating water at different power levels. For the sake of simplicity, we abstracted from these features.

Thermostat

A model of the thermostat is created to control the heater state in the rooms. The thermostat reads the current temperature of the connected room and compares this temperature to the specified target temperature. If the temperature drops below the target temperature minus some threshold, the heater state is turned on, if it rises above the target temperature plus some threshold, it is turned off. Table 2.3 displays the input and output attributes for the model. Note that that *TargetTemperature* input may also be used as a model parameter to specify the initial target temperature.

Similar to the room model, the model of the thermostat is also an abstraction of a real-life thermostat for the sake of simplicity. The modelled thermostat is a rather inefficient one that only turns on and off instead of modulating, as modern

Thermostat			
In/Out	**Name**	**Type**	**Description**
Input	TargetTemperature	`real`	The specified target temperature (℃). This attribute may also be used as model parameter to specify the initial target temperature.
Input	Temperature	`real`	The current temperature in the connected room (measured) (℃).
Output	HeaterState	`boolean`	The output state of the heater(s) based on the inputs. On is represented by `true`, off by `false`.

Table 2.3: *Input and output attributes of the thermostat model.*

thermostats do. The developer of the model decides the complexity of the model, which means that such features could be added if desired.

The sample thermostat model that is used throughout this dissertation switches on the heater when the room temperature drops below the target temperature minus 1.5%. When the temperature rises above the target temperature plus 1.5%, the heater state is switched off. The polling interval is 30 seconds, which means that the temperature is checked once every 30 seconds, after which the heater state is determined. Algorithm 2.1 shows the control algorithm of our thermostat.

Algorithm 2.1 Thermostat control logic.

```
loop                                                      ▷ every 30 seconds
    if Temperature $<$ $0.985 \cdot$ TargetTemperature and not HeaterState then
        HeaterState = true
    else if Temperature $>$ $1.015 \cdot$ TargetTemperature and HeaterState then
        HeaterState = false
    end if
end loop
```

2.2 Modelling

This section describes a number of modelling tools that have been used throughout this dissertation as these tools are supported by our co-simulation approach. These tools are 20-sim, POOSL, VDM-RT and Modelica, which are explained in Sections 2.2.1 to 2.2.4. The 20-sim modelling tool and POOSL language were selected because we already had experience with these. VDM and Modelica are both widely used modelling languages for which open source tooling is available.

2.2.1 20-sim

20-sim[1] is a modelling and simulation tool for mechatronic systems [79]. It allows the user to graphically develop a system by adding components and connecting them to each other. A set of standard building blocks for various domains is provided, but it also allows the user to develop custom building blocks. 20-sim also provides support for bond graph modelling [56, 11]. Inclusion of 3D mechanics provides visual means of verifying and inspecting the designed system. 20-sim also supports code generation in C. This allows for rapid prototyping by generating code directly for an embedded controller or for HIL simulation. 20-sim includes a simulator that can be used to simulate the created models. The results of the simulation can be plotted or animated for analysis.

Example: Room

The model of the room of the RoomThermostat system is a continuous-time model that can be modelled in 20-sim. The model calculates the energy flows through the room, including the energy gain from the heater and energy loss through the window. Figure 2.2 shows the bond graph model of the room in 20-sim.

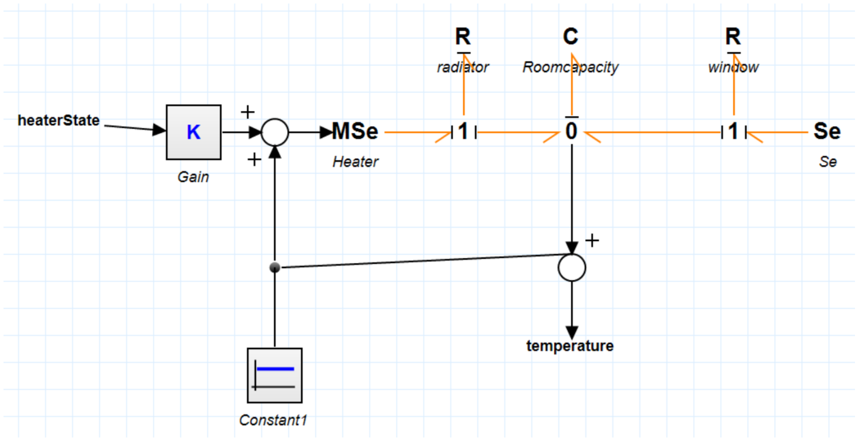


Figure 2.2: *The model of the room, as modelled in 20-sim.*

Example: Thermostat

The 20-sim model of the thermostat is a discrete-time model that determines the heater state based on a current temperature and a target temperature. Figure 2.3 shows the 20-sim thermostat model.

2.2.2 POOSL

The Parallel Object-Oriented Specification Language (POOSL) is a language with formal semantics [74, 10] that is suitable for modelling software architectures and

[1]https://www.20sim.com/

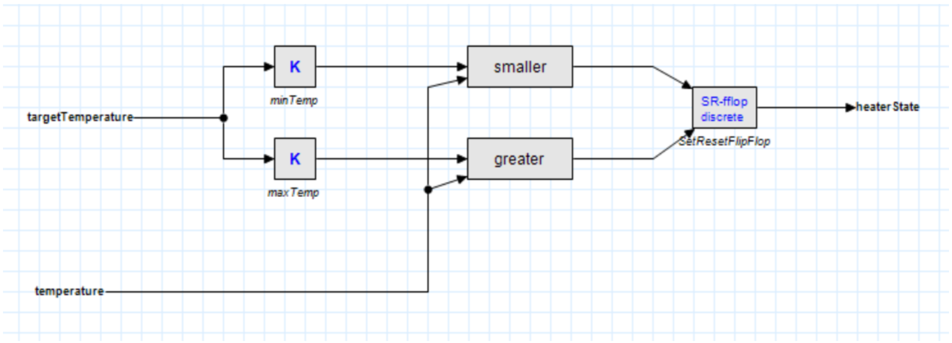


Figure 2.3: *The model of the thermostat, as modelled in 20-sim.*

software behaviour [64]. POOSL allows the developer to incorporate timing behaviour in the model, which adds support for time-aware simulation of software models and performance analysis [75, 84].

POOSL supports traditional statements from programming languages, such as defining variables, while loops and conditional branching. Additionally, POOSL allows the developer to easily execute tasks in parallel and introduce guarded execution paths. A POOSL model contains one or more processes. These processes may run in parallel and may have a number of ports to communicate with each other. Messages can be send to other processes over these ports.

Statements in a POOSL model consume 0 time, except for the `delay`-statement. This statement can be used to control the simulation time of the model. The `delay`-statement is only executed when no other statement can be executed.

In contrast with continuous-time models, it is not possible to move the simulation time of a POOSL simulation to any arbitrary time. When a time step has been taken, all statements until the next `delay`-statement are executed. The POOSL method displayed in Listing 2.1 consists of a loop with a cycle time of 5. When the simulator has just executed the `delay`-statement on line 2, a time step executes the statements on lines 3 and 4 as well as the `delay`-statement on line 2, which increases the simulation time by 5. Only points in time that are multiples of 5 are therefore valid time points in this POOSL simulation.

```
1 | cycle()()
2 |   delay 5;
3 |   i := i + 1;
4 |   cycle()()
```

Listing 2.1: *Sample POOSL process method.*

In general, a POOSL model may contain multiple `delay`-statements in parallel processes and the delay value might depend on the values of variables. When executing a time step, first all possible non-`delay`-statements are executed. Next, the lowest possible delay is determined, after which this `delay`-statement is executed, increasing the simulation time with this value. When multiple `delay`-statements have equal delays, these are all executed.

POOSL models can be simulated with the Rotalumis[2] simulator. To develop POOSL models, a plugin for the Eclipse IDE[3] is available. This IDE provides syntax highlighting and tools for direct simulation and debugging of the POOSL models.

Example: Thermostat

The thermostat model of the RoomThermostat system is a discrete-time control model that is modelled in POOSL. The model contains variables for the input and output attributes. An initialisation method first sets default values for all variables, after which the method `cycle` is started. This method represents the control loop of the model and is displayed in Listing 2.2.

```
1 cycle()()
2   delay 30
3   if (temperature < (targetTemperature * 0.985)) & !heaterState then
4     heaterState := true
5   else if (temperature > (targetTemperature * 1.015)) & heaterState then
6     heaterState := false
7   fi fi;
8   cycle()()
```

Listing 2.2: *Model of the thermostat, as modelled in POOSL.*

The `delay`-statement on line 2 ensures a cycle time of 30. Lines 3 and 5 show the conditions for switching the heater state. The method is called recursively, as displayed on line 8. Note that all statements in a POOSL model – except for the `delay`-statement – are timeless. Consequently, the execution of one cycle takes precisely 30 time units of simulation time.

2.2.3 VDM

The Vienna Development Method (VDM) [6, 5] is a formal method for developing computer-based systems. The VDM Specification Language (VDM-SL) is a specification language that allows the user to create a model of a computing system. Early in de development phase, VDM-SL could be used to create an abstract model of the software without having to develop the actual software yet. The models could then be used for validation purposes.

VDM-SL was later extended to VDM++ [19], which added object-oriented and concurrency patterns to the language. In 2006, VDM-RT was introduced to add support for real-time embedded systems to VDM [81]. VDM-RT allows the modeller to specify timing behaviour of a component either in nanoseconds or clock cycles while still providing the same validation tools that were provided by VDM-SL. VDM is therefore very useful for creating models of software components. Many analyses can be performed on VDM models, because of its mathematical semantics.

There are a number of different tools available for developing VDM models, ranging from tools with only syntax highlighting to tools that are able to perform model analyses. VDMTools [27] is a set of tools for the development and analysis

[2]http://www.es.ele.tue.nl/poosl/Tools/rotalumis/

[3]https://www.eclipse.org/

of models in VDM. The Overture Tool[4] [43] is an IDE for both development and analysis of VDM models. Both tools provide support for code generation and simulation of the created models.

Example: Thermostat

Since the thermostat is a discrete-time component of the RoomThermostat system, it can be modelled using VDM-RT. The VDM thermostat model contains some initialisation procedures that set the variables to their default values. The model contains objects for the input and output attributes that are used for retrieving and setting its values: `temperatureSensor`, `targetTemperature` and `heaterActuator` for the Temperature, TargetTemperature and HeaterState respectively. The method that implements the control loop in the model is displayed in Listing 2.3.

```
 1  Cycle() == (
 2    let
 3      temperature : real = temperatureSensor.getTemperature(),
 4      targetTemperature : real = targetTemperature.getTemperature()
 5    in (
 6      if temperature < (targetTemperature * 0.985) and not heaterState then (
 7        heaterActuator.setHeaterState(true);
 8        heaterState := true;
 9      ) else if (temperature > (targetTemperature * 1.015)) and heaterState
            then (
10        heaterActuator.setHeaterState(false);
11        heaterState := false;
12      )
13    );
14  );
15
16  thread
17    periodic (3e10, 0, 0, 0) (Cycle)
```

Listing 2.3: *Thermostat model in VDM-RT.*

The input attributes are captured into local variables on line 3 and 4, after which line 6 and 9 check the conditions for switching the heater state. Then, the heater state is set both in the `heaterActuator` and the local variable for caching purposes. A periodic thread with an interval of 30 seconds is created for the method `Cycle` on line 17.

2.2.4 Modelica

Modelica[5] is an object-oriented, multi-domain modelling language [31, 48, 30]. In Modelica, every component consists of sets of equations that should be solved by the simulator in order to determine the model's state. Since Modelica is a free modelling language, there are multiple different simulation and development environments available, either commercial, free or open source. A couple of examples of environments are Dymola[6], JModelica[7] and OpenModelica[8]. In this disserta-

[4]`http://overturetool.org/`
[5]`https://www.modelica.org/`
[6]`https://www.3ds.com/products-services/catia/products/dymola/`
[7]`https://jmodelica.org/`
[8]`https://openmodelica.org/`

tion, the latter environment is the one of our choice, since it is an open source implementation that has an interface that is rather similar to 20-sim.

Example: Room

A model of the room was created using OpenModelica. Similar to the 20-sim model, this model calculates losses through the window and the energy gain from the heater, for which the state is provided by the heater state input attribute. This room model is similar to the one that is modelled using 20-sim in terms of behaviour and interfaces, but its implementation is entirely different. Figure 2.4 displays the model of the room as it is modelled in OpenModelica.

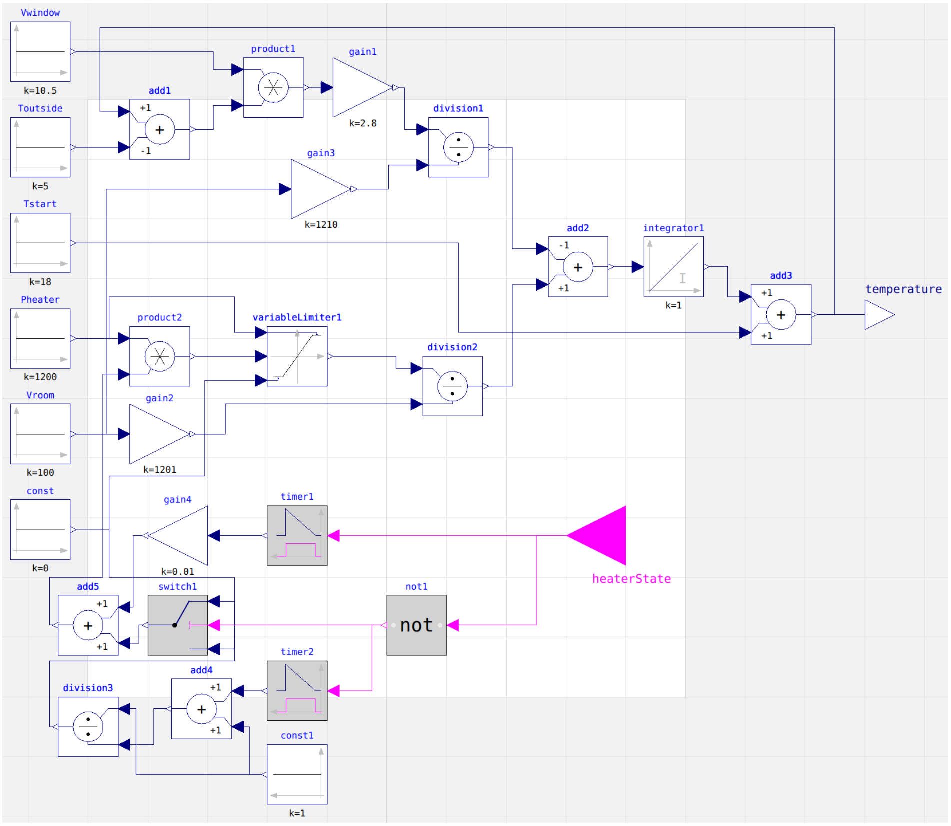


Figure 2.4: *Model of the room, as modelled in OpenModelica.*

Example: Thermostat

Since Modelica supports many different domains, it is also possible to create a model of the thermostat using it. Figure 2.5 shows an illustrative implementation of the algorithm in OpenModelica. The threshold values are calculated from the TargetTemperature. A sampler with an interval of 30 seconds ensures that the input temperature is updated at the right frequency. Numerical comparisons set or reset the state of a flipflop, which outputs the heater state.

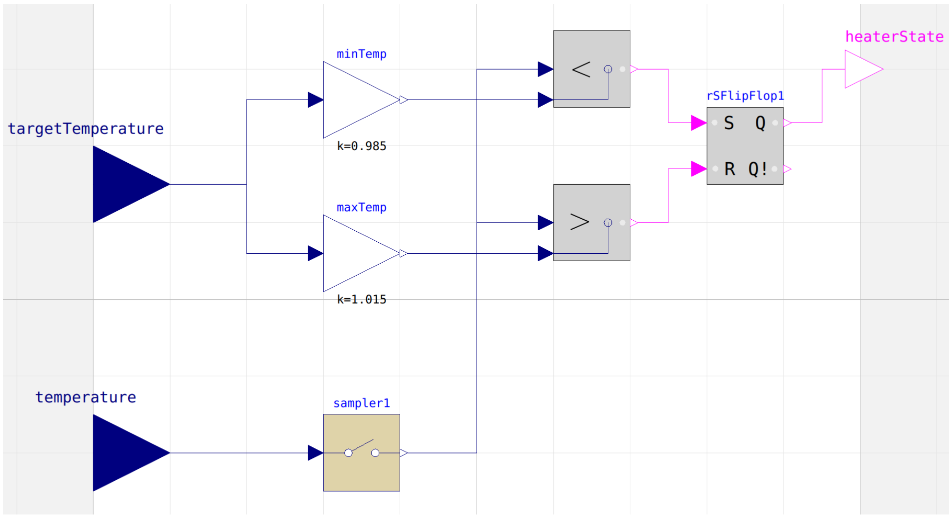


Figure 2.5: *Model of the thermostat, as modelled in OpenModelica.*

2.3 Conclusion

This chapter has introduced the RoomThermostat system, a small and easy to understand case study that will serve as illustrative sample system throughout this dissertation. The components and their models are described as well as a number of modelling tools that were used to create the models. The modelling tools and languages used are 20-sim, POOSL, VDM and Modelica.

The RoomThermostat system was selected as it is an intuitive system that is easy to comprehend. It is clear what to expect from the system, which benefits the understanding of simulation results. The system also allows for the customisation of many different parameters, such as the radiator power and room surface area. This allows for design space exploration to be applied on the system.

CHAPTER 3

Co-simulation

Chapter 2 introduced a number of modelling tools that can be used for the development of simulatable component models. A sample system called the RoomThermostat is also explained. To analyse the system's behaviour with the designed components, these can be simulated together. The concept of simulating these individual component models together as one coherent simulation is called co-simulation. In a co-simulation, the simulation time of the separate component simulations should be synchronised and shared data between the components should be transferred properly. This synchronisation is the responsibility of the co-simulation framework that is used.

This chapter introduces two widely used standards for the co-simulation of different models: the Functional Mock-up Interface (FMI) and the High Level Architecture (HLA). These standards are described in Sections 3.1 and 3.2 respectively. Methods for using POOSL model simulations in a co-simulation are addressed in Section 3.3. Section 3.4 provides a manually developed proof of concept for using the FMI and HLA standards together with a POOSL model simulation for the co-simulation of the RoomThermostat system. Section 3.5 concludes the chapter.

3.1 Functional Mock-up Interface

The Functional Mock-up Interface (FMI) [7] standard aims for model exchange and co-simulation. It is a widely accepted standard for co-simulation purposes [65]. Modelling tools adhering the standard allow the user to import and export Functional Mock-up Units (FMUs). An FMU is a compressed container that contains the following components.

- An XML description of the model with the name *modelDescription.xml* in the root of the container. The file specifies the model's attributes and their properties. A sample of such a file is displayed in Listing 3.1, which displays the contents for the thermostat model of the RoomThermostat system.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<fmiModelDescription fmiVersion="2.0" modelName="Thermostat">
<ModelVariables>
  <ScalarVariable name="Temperature" valueReference="1" causality="
     input" variability="continuous" initial="exact">
    <Real start="18.0" />
  </ScalarVariable>
  <ScalarVariable name="TargetTemperature" valueReference="2" causality
       ="input" variability="continuous" initial="exact">
    <Real start="18.0" />
  </ScalarVariable>
  <ScalarVariable name="HeaterState" valueReference="3" causality="
       output" variability="continuous" initial="exact">
    <Boolean start="false" />
  </ScalarVariable>
</ModelVariables>
</fmiModelDescription>
```

Listing 3.1: *ModelDescription of the thermostat model for the RoomThermostat. Note that the continuous variability of the HeaterState is caused by the fact that this attribute can change at any point in time, not only upon events.*

- The sources of the model may be included in the FMU, so that they can be imported in another tool. Including the sources is optional.
- The executable model may also be included. Inclusion of the compiled model allows other tools to simulate the model without giving the (confidential) sources. Including the compiled model is optional.
- Some tools or models require specific additional configuration or even simulators to be executed. For these types of models, an FMU may also contain a folder with additional resources.

The FMI standard also specifies an interface that must be implemented for every FMU. This interface describes methods to read and write attributes from and to the simulation model, to perform initialisation and for controlling the time of the simulation. Models exported into an FMU can be included in other modelling tools. Since the model description contains all input and output attributes of the model, it can be used as a black box model.

These external interfaces of an FMU can also be used to create a co-simulation from multiple FMUs. Since the FMI standard only provides an interface for controlling the model's simulation without a control algorithm to co-simulate multiple FMUs, such a control algorithm should be provided by the user. The co-simulation therefor relies on an external controller implementation to call the appropriate methods to coordinate the co-simulation.

Modelling tools that support the FMI standard and allow importing of externally created FMUs could be used for building and running such a co-simulation. A list of tools supporting specific features of the FMI standard can be found online[1].

[1] https://fmi-standard.org/tools/

3.2 High Level Architecture

The High Level Architecture (HLA) [1] is a widely accepted standard (IEEE 1516-2010) for co-simulation and distributed simulation [65]. HLA allows multiple models from different modelling tools to be simulated together, creating one coherent simulation of a system. The HLA standard consists of a set of rules that every component in the co-simulation must comply to and an interface specification that specifies how these components could interact with each other.

In an HLA co-simulation, every simulation is called a *federate*, while the entire co-simulation is called a *federation*. A federation consists of roughly three types of components, which are listed below as well as displayed in Figure 3.1.

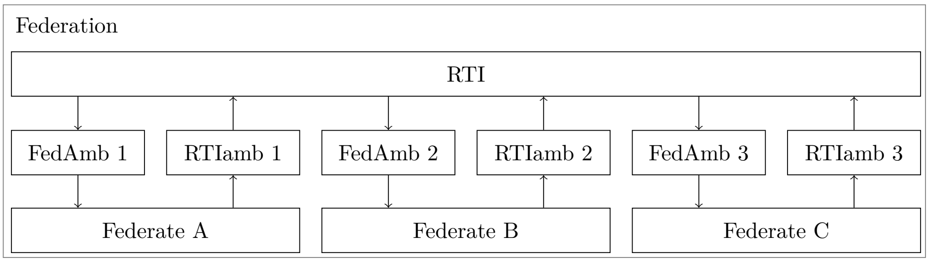


Figure 3.1: *The structure of an HLA federation. The FederateAmbassador and RTI-Ambassador are abbreviated as 'FedAmb' and 'RTIamb' respectively.*

- A Run-Time Infrastructure (RTI) that coordinates the co-simulation. It is responsible for correctly handling attribute and message synchronisation between all simulations as well as controlling the simulation time of the federates. The RTI ensures that the rules of an HLA co-simulation are adhered.

- One or more federates, which are the model simulations that are being executed. These federates communicate with each other through the RTI. The RTI ensures that all messages and updates are transferred in accordance with the simulation time of the federates.

- For each federate, two ambassadors that together form the communication layer between the RTI and the federate. The FederateAmbassador allows the RTI to communicate to the federate by using callbacks while the RTIAmbassador exposes the services of the RTI to the federate. While the RTI mainly ensures that all constraints and rules are met, the ambassadors ensure proper communication according to the standard interface.

Every federation requires a Federation Object Model (FOM) that specifies all simulation models and interactions between them. This includes the specification of attributes and parameters and their properties. The FOM is defined in one or more XML files. The FOM does not describe the number of federates that participates in the federation; it only specifies the federates on a class level. The number of federate instances that joins the federation is determined during run-time. Listing 3.2 displays a large part of the FOM for the RoomThermostat system. Some HLA-specific configuration has been removed to improve readability.

```
<?xml version='1.0' encoding='utf-8'?>
<objectModel
 xmlns='http://standards.ieee.org/IEEE1516-2010'
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xsi:schemaLocation='http://standards.ieee.org/IEEE1516-2010 http://standards
     .ieee.org/downloads/1516/1516.2-2010/IEEE1516-DIF-2010.xsd'>
  <modelIdentification>
      <name>RoomThermostat</name>
      <type>FOM</type>
      <useLimitation>NA</useLimitation>
  </modelIdentification>
  <objects>
    <objectClass>
      <name>HLAobjectRoot</name>
      <sharing>Neither</sharing>
      <attribute>
        <name>HLAprivilegeToDeleteObject</name>
        <dataType>HLAtoken</dataType>
        <updateType>Static</updateType>
        <updateCondition>NA</updateCondition>
        <ownership>DivestAcquire</ownership>
        <sharing>PublishSubscribe</sharing>
        <transportation>HLAreliable</transportation>
        <order>TimeStamp</order>
      </attribute>
      <objectClass>
        <name>Room</name>
        <sharing>PublishSubscribe</sharing>
        <attribute>
          <name>HeaterState</name>
          <dataType>HLAboolean</dataType>
          <updateType>Conditional</updateType>
          <updateCondition>On change</updateCondition>
          <ownership>NoTransfer</ownership>
          <sharing>Subscribe</sharing>
          <transportation>HLAreliable</transportation>
          <order>TimeStamp</order>
          <semantics>N/A</semantics>
        </attribute>
        <attribute>
          <name>Temperature</name>
          <dataType>HLAfloat64BE</dataType>
          <updateType>Conditional</updateType>
          <updateCondition>On change</updateCondition>
          <ownership>NoTransfer</ownership>
          <sharing>PublishSubscribe</sharing>
          <transportation>HLAreliable</transportation>
          <order>TimeStamp</order>
          <semantics>N/A</semantics>
        </attribute>
      </objectClass>
      <objectClass>
        <name>Thermostat</name>
        <sharing>PublishSubscribe</sharing>
        <attribute>
          <name>TargetTemperature</name>
          <dataType>HLAfloat64BE</dataType>
          <updateType>Conditional</updateType>
          <updateCondition>On change</updateCondition>
          <ownership>NoTransfer</ownership>
          <sharing>PublishSubscribe</sharing>
          <transportation>HLAreliable</transportation>
          <order>TimeStamp</order>
          <semantics>N/A</semantics>
        </attribute>
        <attribute>
```

```
66            <name>Temperature</name>
67            <dataType>HLAfloat64BE</dataType>
68            <updateType>Conditional</updateType>
69            <updateCondition>On change</updateCondition>
70            <ownership>NoTransfer</ownership>
71            <sharing>PublishSubscribe</sharing>
72            <transportation>HLAreliable</transportation>
73            <order>TimeStamp</order>
74            <semantics>N/A</semantics>
75          </attribute>
76          <attribute>
77            <name>HeaterState</name>
78            <dataType>HLAboolean</dataType>
79            <updateType>Conditional</updateType>
80            <updateCondition>On change</updateCondition>
81            <ownership>NoTransfer</ownership>
82            <sharing>Publish</sharing>
83            <transportation>HLAreliable</transportation>
84            <order>TimeStamp</order>
85            <semantics>N/A</semantics>
86          </attribute>
87        </objectClass>
88      </objectClass>
89    </objects>
90    <interactions>
91      <interactionClass>
92        <name>HLAinteractionRoot</name>
93        <sharing>Neither</sharing>
94        <transportation>HLAreliable</transportation>
95        <order>TimeStamp</order>
96      </interactionClass>
97    </interactions>
98    <!-- HLA-specific configuration removed for readability -->
99  </objectModel>
```

Listing 3.2: *Federation Object Model (FOM) for the RoomThermostat system. Some HLA-specific configuration has been removed to improve readability.*

In Listing 3.2, line 7 shows the name of the federation that is being described by the configuration. Line 25 starts the definition of the Room federate. The federate contains two attributes (lines 28 and 39) that each have a number of properties, such as sharing type and order of updates, which are specific for HLA. Similar to the Room federate, the Thermostat federate definition starts on line 51 and also contains a number of attributes. Note that this federation does not have any interactions (line 90) specified.

Since both the FederateAmbassador and the RTIAmbassador only transfer data between the RTI and each federate and they provide no additional functionality, we will abstract from these components throughout this dissertation.

3.2.1 Attribute synchronisation

Attribute synchronisation is achieved by a publish-subscribe mechanism [21]. Federates subscribe to attributes from other federates that they are interested in and publish attributes that they would like to share with others. The thermostat in the RoomThermostat publishes its HeaterState attribute and subscribes to the Temperature attribute from the room federate class. Each of the rooms subscribes to the HeaterState attribute that is published by the thermostat. All rooms may publish their temperature – regardless whether the thermostat reads it or not. The thermostat receives temperature attributes from all of the participating room instances that publish the Temperature attribute and applies a filter to find the values coming from the intended room (Livingroom). Figure 3.2 shows an example of how the publish-subscribe mechanism works for the RoomThermostat system.

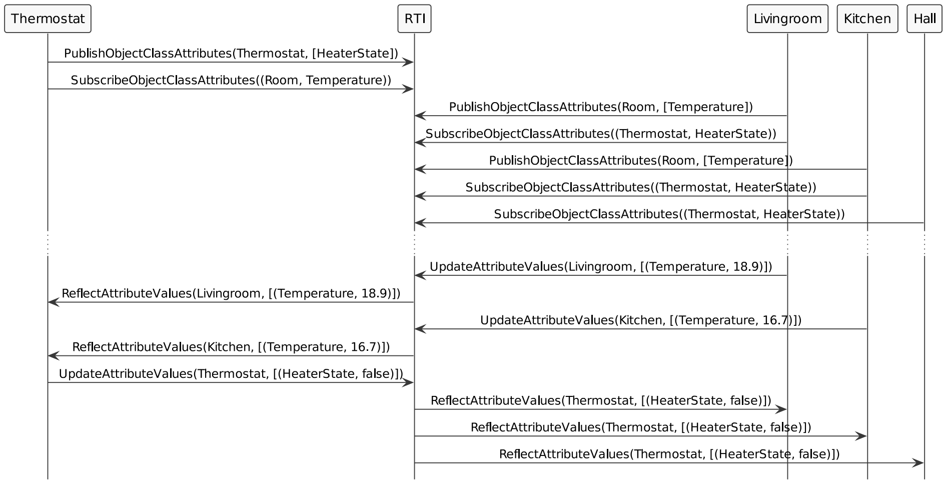

Thermostat
RTI
Livingroom
Kitchen
Hall
PublishObjectClassAttributes(Thermostat, [HeaterState])
SubscribeObjectClassAttributes((Room, Temperature))
PublishObjectClassAttributes(Room, [Temperature])
SubscribeObjectClassAttributes((Thermostat, HeaterState))
PublishObjectClassAttributes(Room, [Temperature])
SubscribeObjectClassAttributes((Thermostat, HeaterState))
SubscribeObjectClassAttributes((Thermostat, HeaterState))
UpdateAttributeValues(Livingroom, [(Temperature, 18.9)])
ReflectAttributeValues(Livingroom, [(Temperature, 18.9)])
UpdateAttributeValues(Kitchen, [(Temperature, 16.7)])
ReflectAttributeValues(Kitchen, [(Temperature, 16.7)])
UpdateAttributeValues(Thermostat, [(HeaterState, false)])
ReflectAttributeValues(Thermostat, [(HeaterState, false)])
ReflectAttributeValues(Thermostat, [(HeaterState, false)])
ReflectAttributeValues(Thermostat, [(HeaterState, false)])

Figure 3.2: *Attribute synchronisation between three federates.*

All federates first notify the RTI about their published and subscribed attributes using the *PublishObjectClassAttributes* and *SubscribeObjectClassAttributes* services as provided by the RTI. The thermostat publishes its HeaterState and subscribes to the Temperature attributes from federates of class Room. Only the Livingroom and Kitchen publish their Temperature attribute, while all three rooms subscribe to the HeaterState attribute from Thermostat classes. Note that the hall does not publish its Temperature attribute. Then, both the Livingroom and Kitchen actually publish new values for their Temperature attribute by using the *UpdateAttributeValues* service. The RTI transmits these values to the thermostat using the *ReflectAttributeValues* callback. The thermostat then updates its HeaterState attribute, which is transmitted to all three rooms, since these are all subscribed to this attribute. Because the *ReflectAttributeValues* callback also specifies from which federate instance the values come, the thermostat is capable of internally filtering which value to react upon, since it should only use the Temperature value from the Livingroom.

3.2.2 Time control

Every federate in an HLA federation has its own local simulation time – called logical time – as well as its own timing behaviour. Federates can be time regulating, time constrained, both or neither one of these. Federates that are time regulating communicate with the RTI in a timestamped fashion: a timestamp is included with every message and attribute update that is sent to the RTI. This allows the federates to share updates while preserving timestamped order (TSO). Time constrained federates are able to receive timestamped messages and attribute updates.

Federates that must synchronise their simulation time with each other should be both time regulating and constrained, since this allows them to both send and receive updates in chronological order according to the simulation times of the federates. When a federate is not bound by time and does not need updates to be received in TSO, this federate can be neither time regulating nor constrained. An example for such a federate is one that only has to determine whether a specific message occurs during the simulation; the time when it occurs is not important.

Every time regulating federate has a lookahead value l, where $l \geq 0$. The lookahead value is communicated to the RTI upon becoming time regulating. The lookahead sets the lower bound of each update sent by the federate, since the federate is not allowed to send updates with a timestamp smaller than its logical time t plus its lookahead l. According to the HLA standard, the following rules apply the each HLA federation.

1. Every time regulating federate where $l > 0$ will never share updates with a timestamp ts smaller than its logical time plus its lookahead: $ts \geq t + l$.
2. Every time regulating federate where $l = 0$ will never share updates with a timestamp ts smaller than or equal to its logical time: $ts > t$.
3. The RTI delivers all TSO messages to time constrained federates in the order of non-decreasing timestamps.

As a result of 3, the RTI can only send an update with timestamp t when it is confident that no other federate will send an update with a smaller timestamp t' ($t' < t$). To be confident, the RTI must know the lower bound of each of the time regulating federates in the federation. The lower bound can be calculated per federate by adding the lookahead to the logical time of the federate. Having specified a lookahead greater than 0 increases the lower bound (1), thus increases the number of updates that can be delivered by the RTI. As described in [32], a federate with zero lookahead (2) limits the number of updates that can be sent by the RTI. Consequently, specifying a larger lookahead to federates improves the simulation speed, since it allows the RTI to send more updates in parallel.

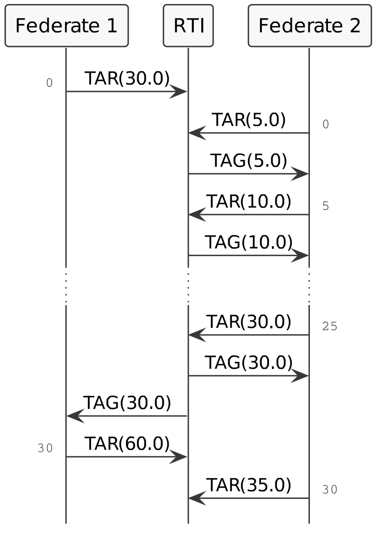


Figure 3.3: *Sample timing behaviour of two federates.*

Advancing time

When a federate wishes to advance in time, it may request permission to do so from the RTI. Permission is requested by sending a Time Advance Request (TAR) to the RTI. The requested time t is provided as attribute with the TAR. Upon sending a TAR to the RTI, the federate promises not to send any more timestamped updates with a timestamp below time t. The RTI sends a Time Advance Grant (TAG) to the federate to notify the federate that it may advance its logical time to time t. If the federate is time constrained, the RTI must send all TSO updates with a timestamp below or equal to the requested time to the federate before granting the time advancement.

Figure 3.3 illustrates the timing mechanism of HLA with a federation consisting of two federates. The logical time of each federate is displayed along the timeline. Both federates are time regulating and time constrained and should therefor be synchronised accordingly. *Federate 1* has a step size of 30 seconds, while *Federate 2* has a step size of 5 seconds. Since this example focuses on the timing mechanism, other communication and updates are ignored. Both federates request a time advancement for their step size to the RTI, but only *Federate 2* receives a grant at first. *Federate 2* continues to simulate solo, until the requested time of *Federate 1* is reached. At that point, both federates receive a TAG from the RTI, after which the process repeats itself.

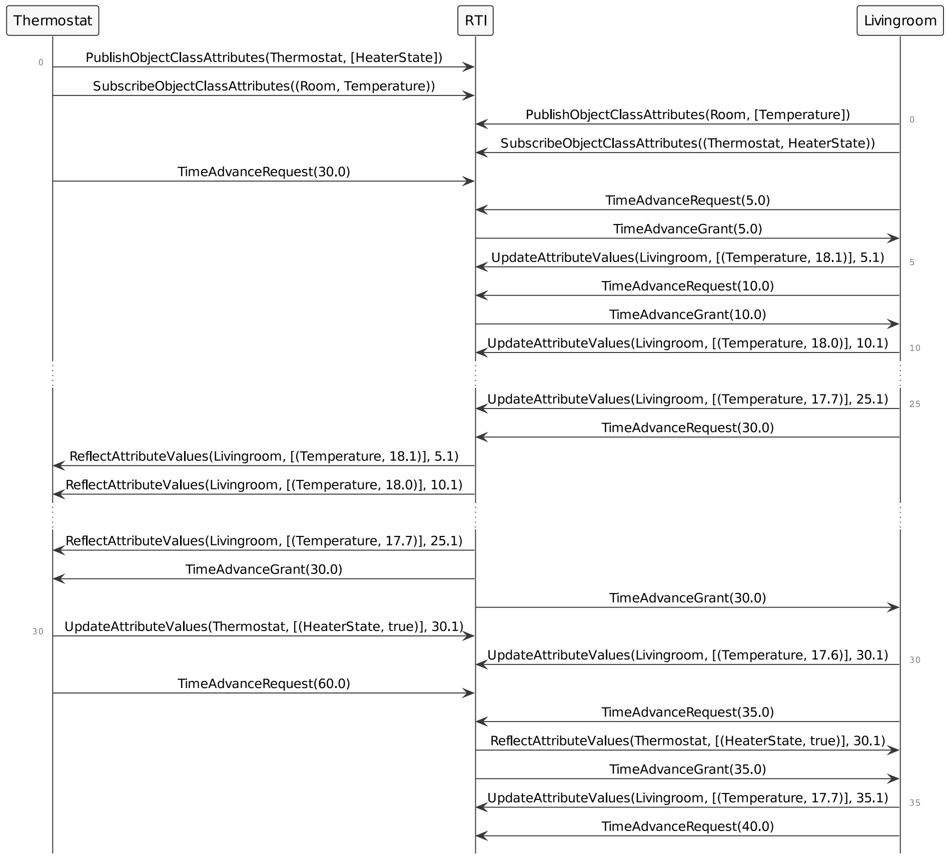

Thermostat
RTI
Livingroom
0
PublishObjectClassAttributes(Thermostat, [HeaterState])
SubscribeObjectClassAttributes((Room, Temperature))
PublishObjectClassAttributes(Room, [Temperature])
0
SubscribeObjectClassAttributes((Thermostat, HeaterState))
TimeAdvanceRequest(30.0)
TimeAdvanceRequest(5.0)
TimeAdvanceGrant(5.0)
UpdateAttributeValues(Livingroom, [(Temperature, 18.1)], 5.1)
5
TimeAdvanceRequest(10.0)
TimeAdvanceGrant(10.0)
UpdateAttributeValues(Livingroom, [(Temperature, 18.0)], 10.1)
10
UpdateAttributeValues(Livingroom, [(Temperature, 17.7)], 25.1)
25
TimeAdvanceRequest(30.0)
ReflectAttributeValues(Livingroom, [(Temperature, 18.1)], 5.1)
ReflectAttributeValues(Livingroom, [(Temperature, 18.0)], 10.1)
ReflectAttributeValues(Livingroom, [(Temperature, 17.7)], 25.1)
TimeAdvanceGrant(30.0)
TimeAdvanceGrant(30.0)
30
UpdateAttributeValues(Thermostat, [(HeaterState, true)], 30.1)
UpdateAttributeValues(Livingroom, [(Temperature, 17.6)], 30.1)
30
TimeAdvanceRequest(60.0)
TimeAdvanceRequest(35.0)
ReflectAttributeValues(Thermostat, [(HeaterState, true)], 30.1)
TimeAdvanceGrant(35.0)
UpdateAttributeValues(Livingroom, [(Temperature, 17.7)], 35.1)
35
TimeAdvanceRequest(40.0)

Figure 3.4: *Attribute and timing synchronisation of an HLA federation.*

Figure 3.4 shows the synchronisation mechanism of both attributes and time for a federation consisting of one thermostat and a living room. Here, the thermostat and living room have step sizes of 30 and 5 seconds respectively. Both federates are time regulating and time constrained and have a lookahead value of 0.1. The federates first publish and subscribe attributes, after which time advancements are requested. The living room federate first proceeds up to time point 25 seconds, after which both federates receive a TAG for advancing to time 30 seconds. Just before this TAG is sent to the thermostat, all updates that were sent from the living room are transmitted. After having granted both time advancements, both federates update their attributes, share them and request a new advancement in time. It can be seen that the updated attributes from the thermostat are again transmitted before granting the time advancement to the living room. Note that the timestamps that are sent to the RTI already include the lookahead value.

In addition to regular TARs, federates may also send a Next Message Request (NMR) to the RTI to request a time advancement. An NMR is similar to a TAR and also includes a time t to which the federate wishes to advance. The RTI, however, is allowed to grant a time advancement to a time t' that is smaller than the requested time: $t' < t$. An NMR allows the RTI to interrupt the time advancement if it receives an update from a federate to which the federate that requested the NMR is subscribed to. When such an update is received, the update is transmitted to the federate, followed by a TAG for time t'. This time synchronisation mechanism is particularly useful for event-stepped federates.

The HLA standard supports many different synchronisation mechanisms and configurations. Instead of synchronising messages in TSO, HLA also supports messages to be synchronised in the order in which they are received, called Receive Order (RO). Properties regarding update conditions and reliability can also be specified. Since all federations in this dissertation use TSO communication, this type of communication was primarily discussed in this section. Moreover, there are many features in HLA that are not described in this work, since it mainly focuses on basic federate specification and attribute synchronisation.

HLA also allows event-based communication by means of interactions between federates. Classes for interactions are specified in the Federation Object Model and may contain attributes. The way interactions are distributed is very similar to the way this is done with attribute updates. Federates may subscribe to interactions and publish interactions. Interactions support intuitive event-based co-simulation and can be used as interrupts when using NMR to advance in time.

3.2.3 HLA implementations

There are multiple different implementations of the HLA standard available, both commercial, free and open source. The MathWorks HLA Toolbox[2] is a commercial implementation for MATLAB and Simulink models. Pitch pRTI[3] is another commercial HLA RTI implementation that provides a set of APIs for use with generic models. CERTI[4], Open HLA[5], Portico[6] and OpenRTI[7] are all open source

[2]https://www.mathworks.com/products/connections/product_detail/forwardsim-hla-toolbox.html

[3]http://pitchtechnologies.com/products/prti/

[4]https://savannah.nongnu.org/projects/certi

[5]https://sourceforge.net/projects/ohla/

[6]http://www.porticoproject.org/

[7]https://sourceforge.net/projects/openrti/

implementations of HLA that provide bindings for various type of generic models using C++ or Java interfaces. Most commercial tools provide graphical user interfaces (GUIs) in which the developer can graphically create federates and a federation. To simulate a model using one of these implementations, a specific interface for each model should be implemented.

Feature		**Pitch pRTI**	**CERTI**	**Open HLA**	**Portico**	**Open-RTI**
Commercial		Yes	No	No	No	No
Open source		No	Yes	Yes	Yes	Yes
Development		Active	Active	Inactive	Active	Active
GUI		Yes	No	No	No	No
HLA version	1516-2010	Yes	WIP[8]	Yes	Yes	Yes
	1516-2000	Yes	Partly	Yes	Yes	Yes
	HLA 1.3	Yes	Yes	Yes	Yes	Yes
Platforms	Windows	Yes	Yes	Yes	Yes	Yes
	Linux	Yes	Yes	Yes	Yes	Yes
	Mac OS	Yes	Yes	Yes	Yes	Yes
Bindings	C++	Yes	Yes	No	Partly	Yes
	Java	Yes	HLA 1.3	Yes	Yes	No
Documentation		Yes	Yes	No	No	Yes

Table 3.1: *Feature comparison of different RTI implementations for HLA.*

Table 3.1 compares the features of a number of available implementations for HLA with each other. In addition to features such as whether the framework is actively maintained and its license, also the supported version of the HLA standard – IEEE 1516-2010, IEEE 1516-2000 or HLA 1.3 – and simulator bindings are important factors to take into account when selecting a framework. The table shows that the commercial product Pitch pRTI supports most features and open source and freely available implementations tend to lack features or active maintenance. Since CERTI does not support the latest standard (IEEE 1516-2010) and Open HLA is not actively maintained, these implementations are less suitable for our purpose. Due to the nature of this research, an open source implementation is preferred over a commercial one, thus Portico and OpenRTI are the best fitting HLA implementations for the research described in this dissertation.

3.3 POOSL

POOSL models currently cannot be exported to a format such as an FMU to be co-simulated with other models; they can only be executed by the Rotalumis simulator. A POOSL model supports three methods to communicate with the simulation. First, Rotalumis provides a debugging interface that can be used to control the simulation execution. Secondly, the POOSL model allows for using

[8]Work in progress

socket within the model. Such a socket can be exploited to specify a small protocol to control the simulation. Finally, the latest version of POOSL allows for message-based communication by means of external ports. These external ports follow the same paradigm as internal ports in the POOSL model. These three methods are described in Sections 3.3.1 to 3.3.3. Section 3.3.4 briefly addresses the possibility to implement a method to export POOSL models to FMUs for co-simulation.

3.3.1 Rotalumis debugging socket

The Rotalumis simulator for POOSL models can open a debugging socket to which another application can connect. The socket is used to communicate according to an XML-based protocol. To connect the POOSL model to a co-simulation, a wrapper is required to interface between the protocols used to communicate with the co-simulation on one end and the Rotalumis debugging interface protocol on the other. The protocol provides commands to perform steps to simulate the POOSL model. It is also possible to read and write attribute values via this socket connection. Even though POOSL primarily focuses on message-based communication, the debugging interface does not provide means to exchange messages with the process being simulated. A subset of the commands that can be send to the debugging socket is listed below.

- A **Timestep** command can be given, which proceeds the simulation time with the minimum of the values of the `delay`-statements that can be executed. All statements until the next `delay`-statement are executed. The command returns the execution state, which represents the state of the simulation execution, including the current simulation time.
- A **Simulationstep** command can be given, which allows the simulator to execute a step of the model. The command also returns the execution state.
- By using **GetProcess**, the state of a specific process in the POOSL simulation can be requested. The command returns the process's state, which includes the attributes and their values of the requested process.
- By using the **SetAttribute** command, a specific attribute in a process can be assigned a value.

With a POOSL simulation participating in an HLA-based co-simulation, the wrapper for POOSL is responsible for proper time management of the federate and sends TARs to the RTI. When a TAG is received, the wrapper requests the simulator to simulate the next time step. Since POOSL models do not need to have cyclic periods for which time steps need to be made, the step size may differ from time to time. However, the debugging interface in Rotalumis does not support to specify the size of the next time step. The wrapper can only ask for a time step to be simulated, after which the wrapper receives the current simulation time. The implementation of the wrapper therefor keeps track of the time by asking Rotalumis to make a time step and then calculating the time step that was taken from the execution state that was returned. A TAR for this time step is then sent to the RTI, even though the step has already been taken in the model simulation.

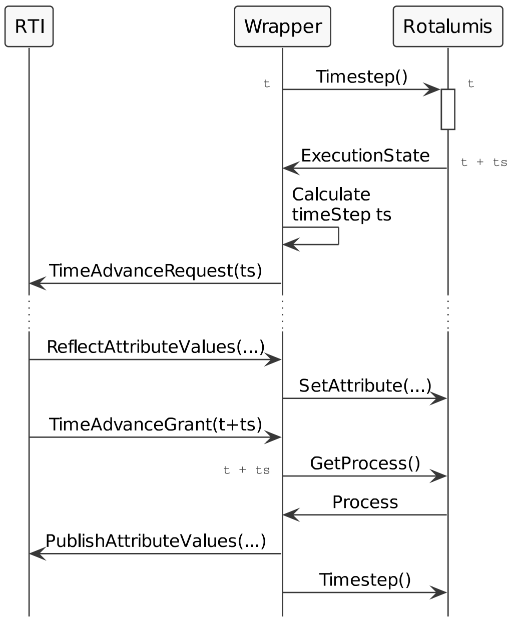


Figure 3.5: *Sequence diagram for the communication steps for running the POOSL model simulation via the Rotalumis debugging socket in an HLA-based co-simulation.*

The mismatch between the time in the POOSL model simulation and the time according to the federate is illustrated in Figure 3.5, which shows that the simulation time of the Rotalumis simulator has already been advanced at the time the TAG is received from the RTI. More importantly, the updates received before this time are not processed at the appropriate time, as these are processed upon simulating the next step. Since many of the processes being simulated are cyclic processes, this is usually not a big problem: the POOSL model is one cycle ahead of the rest of the simulation, but the behaviour remains the same. When processes are simulated in the POOSL model that have a-cyclic timing behaviour, this could be an issue.

3.3.2 POOSL sockets

POOSL models support the use of a socket to communicate with other application. The POOSL model can be extended to use such a socket to allow an external application to control the simulation. For this, a POOSL model should be extended with an additional process – henceforth called *SimulationConnector* – that opens a socket for a wrapper to connect to. Communication according to a simple, custom-made protocol could then be used by the wrapper to control the model simulation. Internally, the processes of the POOSL model should request time advancements to the *SimulationConnector*, which translates these requests to and from the wrappersuppor according to the protocol.

A drawback of this approach is that it requires changes to the POOSL model itself. The added process could be rather generic and modular, but the mappings to and from the attributes in the POOSL model should be adapted for every single POOSL model. Small experiments have shown that this approach does perform better compared to using the Rotalumis debugging interface.

3.3.3 External ports

The developers of POOSL extended the specification language with external ports to enable native socket communication using messages. The aim of this addition is to allow POOSL models being co-simulated without using the debugging socket. Using this approach there is no native support for reading and writing attributes, which is in line with the concepts of POOSL. This differs from the FMI standard as this standard primarily focuses on attribute synchronisation. Also in contrast with FMI, there is no method for time management.

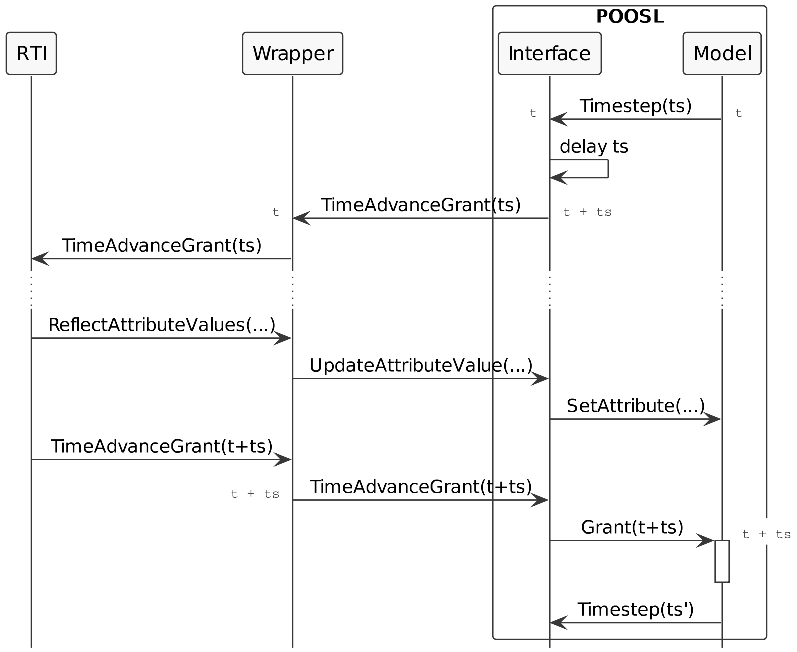


Figure 3.6: *Sequence diagram for the communication steps for running the POOSL model simulation via the external port using the new wrapper.*

We developed a wrapper that uses this external port to communicate with our POOSL wrapper using a custom JSON protocol. A dedicated interface process is added to the POOSL model that communicates over the external port with the federate implementation. In the original POOSL model every `delay`-statement is replaced by a request to the new interface process to perform a time step. This interface process uses a `delay`-statement to transmit a TAR to the wrapper, which then sends this TAR to the RTI. This sequence is displayed in Figure 3.6. Attribute management is also added to this interface process.

Even though the local simulation time of the POOSL model is still advanced before the grant is obtained from the RTI, the addition of the separate process prevents the actual model from proceeding before the attributes are updated. Once the added process receives a grant from the wrapper, this grant is passed on to the model, after which it proceeds the simulation until the next time step. Note that this approach also includes the updating of newly received attribute values at the proper simulation time. Similar to using the POOSL sockets, a drawback of the approach is that it requires the POOSL model to be changed to incorporate the interface process.

Unfortunately, this update for POOSL came rather late during the research project. Consequently, we were not able to finish the implementation of this new communication protocol in time for the experiments described in this dissertation.

3.3.4 FMI standard for POOSL

The aforementioned approaches to allow co-simulation of POOSL models are all sub-optimal. Neither one of these approaches natively allows for time control and message-based communication according to the semantics of POOSL. The debugging socket does support time control and setting and getting of attributes, but it does not allow for message-based communication and the external ports only provide means for this type of communication. Using POOSL sockets allows the user to specify a dedicated protocol, but requires quite some manual development. Combining two or more of these approaches potentially introduces race conditions.

A suggestion is to allow POOSL models to be wrapped in an FMU container for co-simulation purposes. The FMI standard provides methods to control time, read and modify attribute values, which is similar to the functionality that is currently available through the debugging socket. In order to speed up the simulation, a version of Rotalumis specifically developed for co-simulation could be included in the FMU. Since FMUs are flexible enough to implement wrapping mechanisms in different forms, this should not be too difficult to implement. Unfortunately, this approach would not allow the external ports to be used through the FMI, since this kind of message-based communication is not supported by the FMI standard.

3.4 RoomThermostat in HLA

As a proof of concept for the co-simulation of a system using the HLA and FMI standards, an HLA-based co-simulation of the RoomThermostat system is developed. For this, the Portico HLA implementation is used, which provides support for both Java and C++ bindings for connecting federates. The Portico HLA implementation was chosen because it is an open source implementation that provides all HLA functionality that is required for our co-simulation and because it is rather actively maintained. Here, the RoomThermostat will only consist of a single thermostat and a single room: the living room. For each of the models, their connection to the RTI is briefly described below.

Living room

The OpenModelica model of the room that was described in Section 2.2.4 is used for the living room and exported to an FMU. A simulation wrapper is manually developed to connect the simulation model to the RTI. The wrapper uses JFMI[9], which is a Java wrapper for FMI. JFMI provides an API to read and write attributes from and to the an FMU simulation and to control its simulation time.

Thermostat

The POOSL mode of the thermostat that was described in Section 2.2.2 is used for the thermostat. A simulation wrapper for the POOSL model of the thermostat was manually developed using a regular POOSL socket as described in Section 3.3.2. This approach was chosen over the use of the debugging socket because of its simulation speed. The POOSL model of the thermostat is extended with a new process that communicates with the simulation wrapper to allow its simulation time to be controlled and attributes being read and written. For this, a simple protocol based on JSON[10] was designed.

Results

With the two wrappers implemented in Java and the Portico RTI implementation, the RoomThermostat co-simulation could be executed. To be able to execute the same co-simulation multiple times, a scenario was added to the thermostat model. This scenario modifies the target temperature at predefined time points, influencing the co-simulation. The scenario describes a period of four hours and starts with an initial target temperature of 18 °C. After 45 minutes, the target temperature is increased to 20 °C. Two hours later, the target temperatures is lowered to 18.5 °C. After one hour it is lowered again to 18 °C.

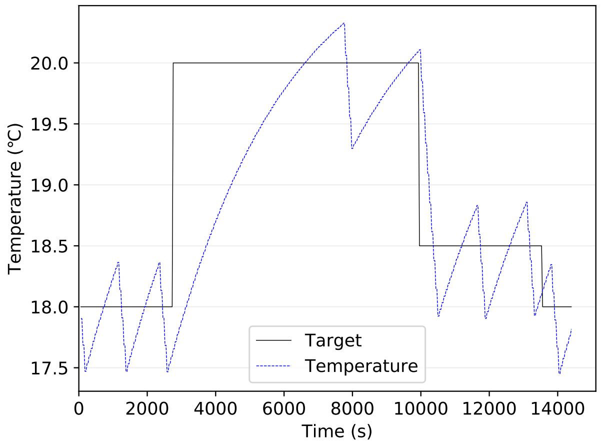


Figure 3.7: *RoomThermostat co-simulation results in Portico.*

The results of the co-simulation of the described system and scenario are displayed in Figure 3.7. The results are similar to the expected intuitive results:

[9]`https://ptolemy.berkeley.edu/java/jfmi/`
[10]`http://www.json.org/`

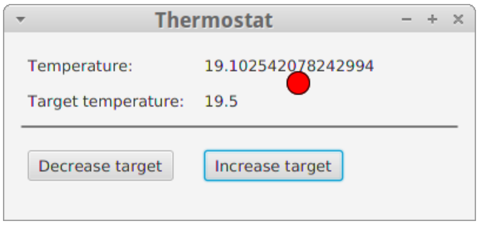
the actual temperature in the living room fluctuates around the specified target temperature.

User interaction

Although predefined scenarios provide possibilities for automated testing and verification, actual user interaction can still be useful for testing purposes. For real heating systems such as the RoomThermostat, the user provides input to the thermostat by specifying the target temperature. To allow user interaction with the co-simulation, a graphical user interface (GUI) was developed for the thermostat.

The GUI joins the co-simulation just like any other federate. It subscribes to the *Temperature* of the living room and to the *HeaterState* of the thermostat. It publishes a *TargetTemperature*, to which the thermostat subscribes to.

Figure 3.8 shows the GUI, which displays the current temperature, target temperature and the heater state and provides buttons to increase or decrease the target temperature by half a degree. The federate for the GUI is both time regulating and constrained and only sends an update when the user has pressed a button. This timing policy is required to keep the federate synchronised with the other simulations.


Thermostat

Temperature: 19.102542078242994

Target temperature: 19.5

Figure 3.8: *The thermostat GUI for user interaction.*

3.5 Conclusion

In this chapter, two existing standards – HLA and FMI – were described. These standards can be used to construct a co-simulation of different types of models. Also, approaches for simulating POOSL models in the context of a co-simulation are described.

Since FMI is supported by many modelling tools, it is a very useful standard to support. The standard, however, does not provide a master algorithm for the co-simulation of multiple FMUs. The HLA standard provides such a master algorithm to orchestrate a co-simulation of different simulations and several commercial and open source implementations are available. We therefor use the HLA standard to run a co-simulation of FMUs.

To illustrate the use of these techniques together in a co-simulation, a co-simulation for the RoomThermostat system is developed. Here, the co-simulation construction serves as a proof of concept, as all wrapper are developed manually. The process of developing these wrappers is simplified by CoHLA, which is introduced in Chapter 4.

Reflection on requirements

3. *Simulation models from multiple tools are supported: at least 10 modelling tools.* A wide range of modelling tools is supported by using the FMI standard for creating simulatable models for the co-simulation. Approaches to support POOSL models being co-simulated are discussed.

4. *The approach is easily extendable to support new tools.* The use of open source implementations of the HLA standard allows for building extensions to support different types of models or interactions with other tools. Depending on the chosen implementation, the development of such extensions may be more or less timing consuming than specified by the requirement.

12. *The framework runs on Windows, Linux and Mac.* Open source implementations for both the HLA and FMI standards are available. The implementations used for the proof of concept as described in Section 3.4 claim to run on all major platforms.

CHAPTER 4

CoHLA

Although HLA is a very suitable co-simulation framework for CPSs, it requires the FOM and wrappers to be changed when the models or their interfaces are changed. Since this happens frequently during the system design, the construction and maintenance of the co-simulation become very time consuming. To overcome this issue, Configuring HLA (CoHLA) was developed. CoHLA is a domain-specific language (DSL) that allows the system architect to easily specify the simulation models in terms of their interface and to connect them with each other to form a co-simulation. From this specification, the wrapper implementations for the simulations as well as the FOM are generated. Consequently, the construction of a co-simulation only requires a CoHLA specification and does not require any manually developed code, speeding up the process of constructing a co-simulation.

This chapter starts with a description of the intended design flow that CoHLA supports in Section 4.1. Details on the implementation of CoHLA are given in Section 4.2. Section 4.3 describes the language itself. A selection of the features of CoHLA is described in Section 4.4. Section 4.5 lists the components that are generated by the framework. An example of using CoHLA by specifying the RoomThermostat system is given in Section 4.6. The chapter is concluded in Section 4.7.

4.1 Design flow

CoHLA supports a rather straight-forward model-based approach for the design of a cyber-physical system. A simplified design flow is displayed in Figure 4.1. After specifying the basic requirements of the system (1), the components are identified (2). To enable proper communication between these components, their interfaces have to be specified (3). Depending on the level of abstraction of the

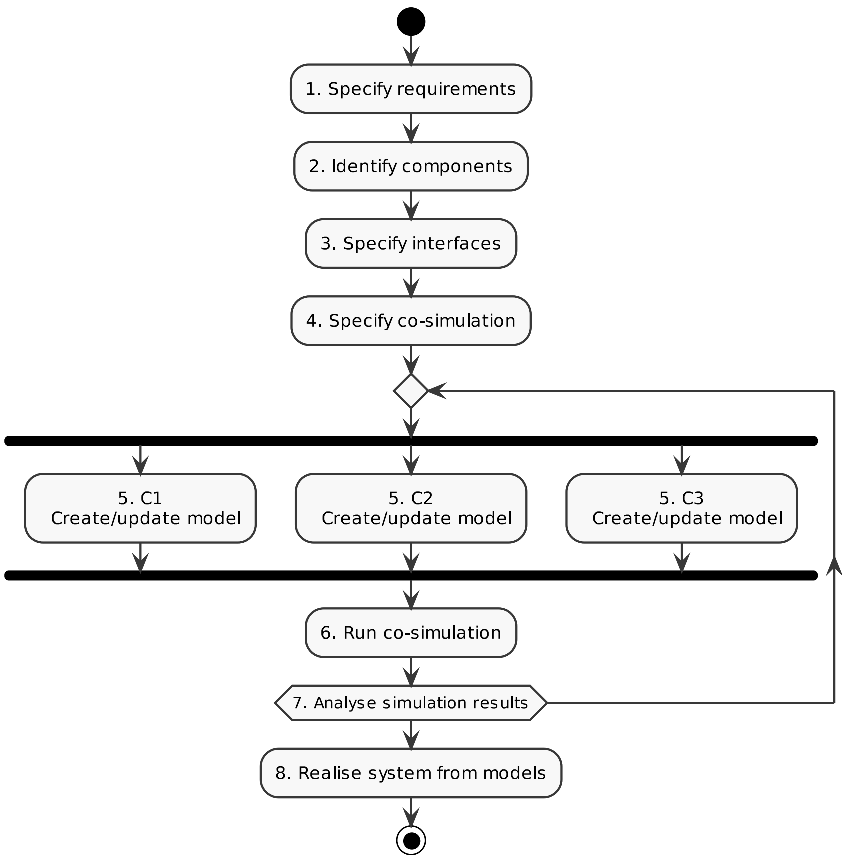

1. Specify requirements
2. Identify components
3. Specify interfaces
4. Specify co-simulation
5. C1 Create/update model
5. C2 Create/update model
5. C3 Create/update model
6. Run co-simulation
7. Analyse simulation results
8. Realise system from models

Figure 4.1: *Activity diagram of the intended design flow of a CPS. C1, C2 and C3 represent three different components of the system. This is just exemplary, as the design flow also applies for more systems consisting of more components. CoHLA plays a prominent role in steps 4 and 6 of the design flow.*

models, these interfaces can be used for both the implementation of the system itself as well as the construction of the co-simulation. Next, a CoHLA co-simulation definition can be created (4). Then, the development of the individual components starts (5). Components are developed by teams specialised in the field of the component using the tools that fits the development best. The co-simulation of the models of the separate component is run (6) and the results are analysed (7). Steps 5, 6 and 7 are repeated until the models are sufficiently detailed and mature to be implemented into real hardware and software. During this iterative process, interfaces between the components may change, which also requires adjustments to the co-simulation specification. When the process is finished, the components are implemented and assembled (8) to complete the system development.

Since every system is different, their development processes are also different. Consequently, the design flow described above is only a schematic representation of the development process. Although (5) suggests that the development of every component is perfectly synchronised with the other components, this is usually not the case. The development of the component models itself is also an iterative process that differs from one component to another. Some components are even (partly) realised during this phase. Depending on the system, actual realisations could also be executed together with simulations of other components, which is also less linear to what is suggested in the picture. Moreover, when the requirements change, one may return to either one of the first four steps.

4.2 Implementation

There are many frameworks to develop domain-specific languages (DSLs) [80, 41, 28]. The Xtext [22] framework is used to implement CoHLA, because it provides the tools to easily distribute CoHLA by means of an Eclipse[1] plugin. The framework is relatively easy to learn and allows the developer to rapidly create, modify and distribute the newly created language. Eclipse serves as the IDE for the language. Xtext allows the developer to create a DSL by defining a grammar and implementing generators [4]. From a grammar specification in Xtext, the Xtext plugin generates the required back-end for parsing and validation. Additionally, an empty generator, validator and scope provider are generated. These sources are all generated in Xtend, which is an extension of Java. From an Xtext project, it is possible to export an Eclipse plugin for convenient distribution of the IDE for the DSL. CoHLA is open source and its grammar is included in Appendix B.

CoHLA supports code generation for OpenRTI[2]. This is different from the Portico implementation that was used for the proof of concept as described in Section 3.4, because it was found that Portico lacked a proper implementation of the NMR. In order to support interrupts being triggered in the co-simulation, the NMR call must be supported by the HLA implementation. Consequently, OpenRTI was selected because it is also an open source implementation of the HLA standard that is actively maintained and it supports all major features of HLA, as was displayed in Table 3.1. OpenRTI is developed using C++, which allows bindings with many different types of models.

4.2.1 Libraries

Since simulation models are uncapable of connecting directly to the RTI, a number of libraries were developed to simplify the implementation of such a connection. Such connection libraries were developed for both FMUs and POOSL models. The libraries implement basic handles for controlling time and synchronising attributes between the RTI and the model. Not all functionality, however, is implemented in the libraries, as some of the functionality depends on the model. This functionality is generated by CoHLA in the form of a wrapper for the model. This wrapper inherits the base implementation of the library and is used to control a simulation instance of one specific model.

Figure 4.2 shows the federation architecture for a federation consisting of two federates. The RTI and the interface implementation of HLA are provided by

[1]`https://www.eclipse.org/`

[2]`https://sourceforge.net/projects/openrti/`

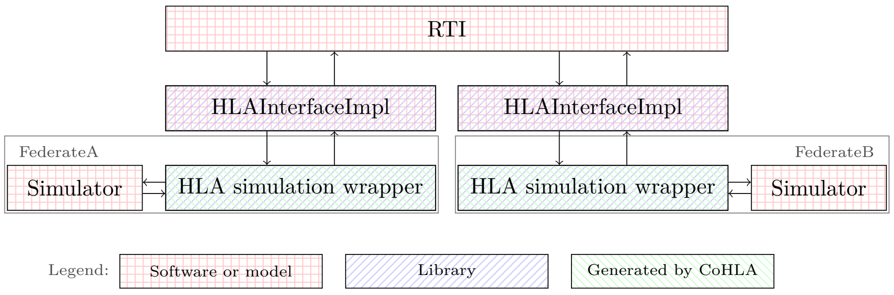


Figure 4.2: *Federation architecture as generated by CoHLA.*

OpenRTI. The CoHLA libraries extend the interface implementation and add functionality that the simulation wrappers could use. The main difference between the HLA interface implementation and the wrapper is that the wrapper is generated specifically for the model while the interface implementation includes only generic methods. A simulator can be an FMU container or a simulator running some model, e.g. Rotalumis running a POOSL model. The HLA simulation wrapper is generated by CoHLA and inherits functionality from the libraries that were developed. It can be seen that the libraries together with the generated code by CoHLA form the glue between the RTI and simulation models.

In contrast with the connector that was developed for the co-simulation of a POOSL model in Section 3.4, the CoHLA library uses the debugging interface to connect to POOSL models. The former method required an additional process in the model to facilitate time control and attribute synchronisation to the library. Consequently, a model should be changed to support co-simulation. To avoid that every POOSL model has to be created with co-simulation in mind, the debugging interface is used by the CoHLA library. This approach slows down the execution, but it decreases the overhead of having to change the models.

4.2.2 Wrappers

The CoHLA-generated wrappers partly inherit the functionality implemented by the library they extend and partly consist of an implementation specifically generated for one kind of model, as displayed in Figure 4.2. The goal of the wrapper is to connect a simulation model to the HLA implementation. For this, the wrapper mainly consists of methods that translate HLA method calls to the corresponding calls for the simulation model, which is generally an FMU or a POOSL model.

For POOSL models, the main functionality during the execution of the simulation is displayed in Figure 3.5. HLA calls are translated to the appropriate calls to the Rotalumis debugging socket and the wrapper retrieves information from the simulation before transmitting an update to the RTI. The communication with an FMU simulation is displayed in Figure 4.3.

This figure shows that the synchronisation mechanism is very similar to the one for POOSL models. Since the time step s must be specified for FMUs, the wrapper first requests this time advancement to the RTI. Before the RTI grants this time advancement, the updates from other federates are transmitted to the

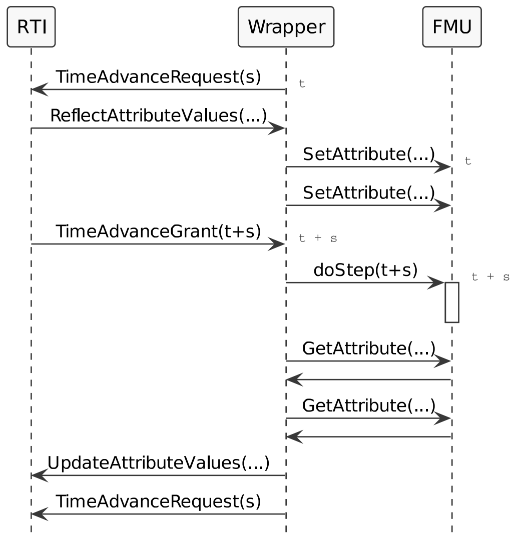


Figure 4.3: *Sequence diagram for the communication steps between the RTI and FMU simulation through the CoHLA-generated simulation wrapper.*

wrapper. The wrapper uses the appropriate FMI methods to set these attribute values in the simulation model. When a time advance grant is received from the RTI, the *doStep* method is called to trigger the FMU simulation to compute its new state. Hereafter, the wrapper requests the attribute values from the FMU and transmits these updates to the RTI. Finally, a new time advancement is requested to the RTI.

The displayed sequence diagram shows the basic functionality of all CoHLA wrappers. Models are treated as black-box models from which information can be retrieved, to which information can be set and of which the time can be controlled. The exact implementation of the model is therefor not relevant for the co-simulation itself. A number of CoHLA libraries and wrappers, such as the logger and collision simulator in Sections 4.4.3 and 4.4.11 respectively, do not connect to an external simulator, as these already contain all the required functionality for their purpose.

4.2.3 Extending CoHLA

The Xtext and Xtend frameworks were selected for developing CoHLA due to their flexibility. Therefore, it should be relatively easy to add functionality to CoHLA or to modify existing functions. To add new functionality to CoHLA, the following components should be added or modified.

- A base library for the functionality should be developed. The library could contain generic methods for time control and or attribute handling.
- The CoHLA grammar should be extended to support the new concepts.

- Code generators for the new functionality should be implemented and included in the existing code generation process. For new wrappers for specific types of models, the code generator typically consists of a generator for the wrapper and one for connecting to the simulator. Concepts might also require new configuration files to be created or the run script to be changed. In this case, generators for these configuration files should be developed or the generator for the run script should be modified.

- Possible scope providers for the Eclipse plugin to support autocompletion and avoid errors being made by the users.

- Code validators to detect possible errors or undesirable behaviour while the user is specifying the co-simulation.

- The CoHLA documentation should be extended to explain the added or modified functionality.

Not every change requires all these components to be changed as well. For example, some functionality may be implemented by only adding a generator for a wrapper without the need of a base library to be developed. The addition of the first version of the collision detector to CoHLA, as introduced in Section 4.4.11, involved the development of a base library, an extension of the CoHLA grammar and wrapper code generators. This process was finished within a day. Hereafter, the extension was updated several times before being finalised. During this process, the time consumed by applying these changes to the framework was limited.

4.3 Language

This section describes the CoHLA language and provides a number of small examples of how the language can be used to specify a co-simulation. More details on the language can be found in the documentation[3]. Every co-simulation specification in CoHLA requires an environment specification. The environment is used to configure the federation that is being specified. A sample environment is displayed in Listing 4.1.

```
Environment {
  RTI {
    OpenRTI
    Libraries "/opt/OpenRTI-libs"
  }
  PrintLevel State
  PublishOnlyChanges
}
```

Listing 4.1: *Example of a CoHLA environment specification.*

The environment starts with an `Environment` keyword, after which the body contains the configuration properties. The body contains the following elements.

[3]`https://cohla.nl/docs/`

- The RTI implementation is specified using the `RTI` keyword. With each supported RTI implementation, a set of libraries to connect the models is included. The location of these libraries is specified by using the `Libraries` keyword.
- During the simulation execution, every federate prints its state to the console by default. This behaviour can be changed by specifying the `PrintLevel`. Printing can be turned off (`None`), only print the time (`Time`) or print the time and state, which outputs all attribute values as well (`State`).
- The `PublishOnlyChanges` keyword can be used to limit the amount of attribute updates being sent by the federates. When this keyword is set, federates only publish their attributes if the attributes are different from the last time they were published. Attribute values are cached by the federate library.

After the environment definition, the federates that participate in the co-simulation need to be specified. Every simulation model requires its own federate class specification. Listing 4.2 shows the specification for the Room class in the RoomThermostat system.

```
FederateClass Room {
  Type FMU
  Attributes {
    Input Boolean HeaterState
    Output Real Temperature
  }
  Parameters {
    Real RadiatorSize "radiator.A"
    Real Surface "Roomcapacity.surface"
    Real Height "Roomcapacity.height"
    Real InitTemp "Roomcapacity.initialtemp"
    Real WindowSize "window.A"
  }
  TimePolicy RegulatedAndConstrained
  DefaultModel "../../Room.fmu"
  AdvanceType NextMessageRequest
  DefaultStepSize 5.0
}
```

Listing 4.2: *CoHLA federate class specification for the room of the RoomThermostat system.*

The specification of the federate class contains the following properties.

- The federate class definition starts with a specification of the type: FMU or POOSL.
- The model attributes are specified. Each attribute is either an `Input`, `Output` or `InOutput` (both input and output) attribute of a specific type and with a name.
- Model parameters can be specified. The model parameters also have a name and a type. Optionally, an alias for the parameter in the model can be defined: the parameter RadiatorSize is used in the CoHLA definition and maps to *radiator.A* in the model. If no alias is specified, the name of the parameter is used.

- The time policy – which states whether the federate is time regulating, time constrained, both or neither one of these – is specified as well.
- The path to the model to load by default is specified on line 15.
- By default, all time advancement requests are sent to the RTI as TARs (`TimeAdvanceRequest`). `AdvanceType` on line 16 allows the user to change this behaviour to send NMRs (`NextMessageRequest`) to the RTI.
- Line 17 specifies the step size to simulate the model with.
- Similar to specifying the default step size of a model, the default lookahead value of the federate may also be specified. The example above does not include this. When no default lookahead value is specified, the default value (1.0) will be used.

For the RoomThermostat system a specification for the thermostat model should also be included. Listing 4.3 shows the specification for this model.

```
 1  FederateClass Thermostat {
 2    Type POOSL {
 3      Processes {
 4        Thermostat in "Thermostat"
 5      }
 6    }
 7    Attributes {
 8      InOutput Real TargetTemperature in Thermostat as "targetTemperature"
 9      Input Real Temperature in Thermostat as "temperature"
10      Output Boolean HeaterState in Thermostat as "heaterState"
11    }
12    Parameters {
13      Real TargetTemperature "targetTemperature" in Thermostat
14    }
15    TimePolicy RegulatedAndConstrained
16    DefaultModel "models/StandaloneThermostat.poosl"
17    AdvanceType TimeAdvanceRequest
18  }
```

Listing 4.3: *CoHLA federate class specification for the thermostat of the RoomThermostat system.*

The thermostat federate class is a POOSL model with one internal POOSL process and with a default port for the Rotalumis debugging socket (lines 3 to 6). The attributes are defined, including a mapping to the process in which these can be found in the model (lines 8 to 12). Note that the *TargetTemperature* is also an output attribute to enable the attribute to be logged by our logger. This attribute is also a parameter to set during initialisation. The model is time regulating and time constrained, a default model is specified and regular time advance requests are sent to the RTI. No default time step should be specified for POOSL models, since POOSL models have a variable step size.

When all federate classes for the co-simulation are specified, the co-simulation can be constructed in CoHLA. This co-simulation definition specifies simulation instances and their connections. Listing 4.4 displays the most basic federation definition of the RoomThermostat system in CoHLA.

```
Federation House {
  Instances {
```

```
 3 |    Livingroom : Room
 4 |    Kitchen : Room
 5 |    Hall : Room
 6 |    Thermostat : Thermostat
 7 |    Logger : Logger
 8 |  }
 9 |  Connections {
10 |    { Livingroom.HeaterState <- Thermostat.HeaterState }
11 |    { Kitchen.HeaterState <- Thermostat.HeaterState }
12 |    { Hall.HeaterState <- Thermostat.HeaterState }
13 |    { Thermostat.Temperature <- Livingroom.Temperature }
14 |    { Logger <- Livingroom.Temperature, Kitchen.Temperature, Hall.Temperature
   |        , Thermostat.TargetTemperature }
15 |  }
16 |}
```

Listing 4.4: *CoHLA federation definition of the RoomThermostat system.*

Lines 2 to 8 specify the instances of the model simulations participating in the co-simulation, which are three rooms, a thermostat and a logger. Then, lines 9 to 15 specify the connections between the attributes of the federates. Only the temperature of the *Livingroom* is used by the thermostat and all rooms receive the *HeaterState* as input. The logger receives the temperatures of all rooms and the target temperature from the thermostat.

CoHLA specifies connections of attributes on an instance level, while HLA provides synchronisation as a publish-subscribe mechanism on a class level. However, every attribute update that is sent to a federate in HLA includes the source of the attribute, which represents the instance. Even though the RTI transmits the temperature updates from all instances of the room class to every thermostat, the implementation of the thermostat could still apply a filter on these incoming updates. These filters are implemented in the libraries, which allows CoHLA to specify attribute sharing on instance level.

4.4 Features

This section briefly explains the functionality of the CoHLA framework. This includes a number of language constructs that could be used and the library support that is implemented.

4.4.1 Functional Mock-up Interface

A base library is developed to co-simulate models in FMU format in a CoHLA-generated co-simulation. From a CoHLA co-simulation specification, a wrapper is automatically generated to wrap the specific model with all relevant attributes and parameters. For now, the CoHLA framework is limited to fixed-step FMUs. These FMUs contain simulations that can be simulated with a specified interval. A mechanism to request the desired time point to simulate to the FMU is missing in CoHLA, even though this is supported by the FMI standard.

While the base library implements most generic methods for communicating with FMI simulations, the wrapper provides implementations for the model-specific methods. Together, these form the wrapper of the FMU being simulated and translate HLA-specific methods to the appropriate FMI methods and vice versa. Additionally, a number of caching mechanisms is implemented to improve the simulation speed.

4.4.2 POOSL

Similar to the FMI support in CoHLA, both a base library and wrapper generation for POOSL models is supported by CoHLA. Since the proof of concept as described in Section 3.4 showed that modification of the POOSL model to enable it to be co-simulated was rather time consuming, the wrapper connects to the Rotalumis debugging socket as described in Section 3.3.1. The wrapper is responsible for the translation of HLA-specific methods to calls to the Rotalumis simulator and vice versa. Since CoHLA currently only supports attribute synchronisation and no message-based interaction between federates, the POOSL wrapper also does not support this.

4.4.3 Logging

In order to be able to extract useful data from a co-simulation, a logging mechanism should be available. A logging federate is therefor implemented in the libraries for inclusion in a federation. The logger is a time constrained federate that participates in the co-simulation as an ordinary federate. The user specifies the attributes of other federates that need to be logged, after which the logger federate subscribes to these attributes. Consequently, the logger receives all attribute updates that it is interested in during the simulation. An end time must be specified by the user to configure the time span that should be recorded. When either the end time is reached or the federation execution is stopped, the logger stores all logged attribute values into a single CSV file.

4.4.4 Parameter configurations

Since many models have parameters that can be configured upon starting the simulation execution, the FMU and POOSL libraries also include some methods to specify their initialisation values. Examples for the RoomThermostat include the size of the radiator, the window and the room itself. The wrappers that are generated for each of the models include arguments for specifying these parameter values for the model. When specified, these parameter values are set by the wrapper implementation upon initialisation of the model execution.

When the co-simulation itself has been defined in terms of instances and connections, values for the parameters of each of the instances can be specified. To do this, CoHLA allows the user to specify a *Configuration* for a specific federate class. Such a configuration specifies values for one or more parameters of the class and can be applied to federate instances in the federation. It is not required to provide value for all parameters in a parameter configuration. When no value is specified for a specific parameter, its default value as specified in the model is used. Listing 4.5 shows an example configuration – called *Large* – for a room federate. This configuration specifies parameter values for the surface of the room and the sizes of both the window and radiator in the room. Similar to this configuration, the configurations *Small* and *Medium* have also been created to represent different rooms in terms of surface area, window size and radiator size. A configuration was also created for the thermostat federate class to define the initial target temperature. This configuration is called *Comfortable*.

```
Configuration Large for Room {
  Surface = "45.0"
  WindowSize = "11"
  RadiatorSize = "1.0"
}
```

Listing 4.5: *Configuration Large for a Room federate.*

One or more of these configurations can be applied to different federate instances within a co-simulation. Such a set of applied configurations to instances is called a *Situation*. The use of such situations allows the user to quickly create multiple co-simulation configurations for one co-simulation definition. Listing 4.6 shows a situation as it can be specified in CoHLA. This situation applies different parameter configurations to different instances in the co-simulation. These parameter configurations have briefly been introduced in the previous paragraph.

```
Situation ComfyBase {
  Apply Comfortable to Thermostat
  Apply Large to Livingroom
  Apply Medium to Kitchen
  Apply Small to Hall
}
```

Listing 4.6: *A situation for the RoomThermostat system co-simulation.*

It is possible to let a situation extend another situation or to apply multiple configurations to a single federate instance. When a configuration overwrites a previous configuration, overlapping parameters values will be determined by the configuration that is applied last, while other parameters supplement each other. The same holds for situations extending each other. By being able to extend these parameter values, the user is able to create a highly flexible and reusable sets of predefined parameter values.

4.4.5 ConnectionSets

CoHLA supports the specification of sets of attribute connections between federates on a class level. In CoHLA these sets of connections are called *ConnectionSets*. *ConnectionSets* focus on the reuse of the same type of connections. Listing 4.7 displays a sample *ConnectionSet* for the RoomThermostat system.

```
ConnectionSet between Room and Thermostat {
  { Room.HeaterState <- Thermostat.HeaterState }
}
```

Listing 4.7: *Sample ConnectionSet for the RoomThermostat system.*

Since the Room and Thermostat classes should always connect the *HeaterState* attributes when connected to each other, this connection is included in the *ConnectionSet*. Another attribute that can be connected between these federates is the *Temperature* of the Room. However, since only one room typically provides this input for the thermostat, this attribute could better be left out of the *ConnectionSet*. Listing 4.8 shows how the connections between the different federates in the RoomThermostat as displayed in Listing 4.4 could be specified when using the *ConnectionSet*.

```
Connections {
  { Livingroom - Thermostat }
  { Kitchen - Thermostat }
  { Hall - Thermostat }
  { Thermostat.Temperature <- Livingroom.Temperature }
}
```

Listing 4.8: *Example use of ConnectionSets for the RoomThermostat system.*

When a connection is specified using the '-' instead of the '<-', the corresponding *ConnectionSet* will be used automatically. While this notation is just a little more comprehensive in this example, it becomes more useful when the system is larger or when more attributes are shared between many components of the same class. Due to the small size of the co-simulation and the small number of attributes, this example only illustrates the use of a *ConnectionSet*. Case studies in Chapters 6 and 7 make more efficient use of the feature.

4.4.6 Input operators

Sometimes, an input attribute may receive input from multiple other federates. By default, the updates are all transmitted to the receiving federate, causing every new value to overwrite the previous one. Consequently, when a federate reports a specific value, this value might be overwritten by another federate reporting a different value, since these are both stored in the same attribute of the receiving federate. The receiving model will then ignore the first attribute value, as there is no simulation step processed after receiving this update. This is the result of HLA only providing a TAG when all updates have been sent.

CoHLA supports an optional operator for an input attribute. The wrapper that is generated by CoHLA combines multiple attribute values into one single input attribute value using the specified operator. The wrapper re-computes this attribute value on every TAG that is received. When one or more federates did not update their attribute values, their previous values are used for the computation. Supported operators are `and` (&&) and `or` (||) for boolean values and addition (+) and product (∗) for numeric values.

Listing 7.3 illustrates how these operators can be used on lines 4 and 5. The input operator for the attribute is specified directly after the type of the attribute. For this model, the input for the attribute *activity* is `true` when at least one of the connected sensors outputs `true`. This functionality is implemented in the wrapper that is generated by CoHLA. The implementation caches input attributes to determine the correct value for the input attribute.

4.4.7 Scenarios

To be able to make design decisions based on relevant use cases of the system, CoHLA supports the co-simulation of these cases by means of scenarios. Scenarios often describe the interaction the system has with a user or other systems. The possibility to automatically run a predefined scenario is particularly useful during the DSE phase, where many simulations are executed with slightly different configurations.

In CoHLA, a scenario may specify changes of variables or input to the system at specific points in time during the simulation. Every scenario is CoHLA is specifically designed for a federation and is identified by a unique name. A scenario contains a set of events and optionally an end time for the scenario, after which the co-simulation of the system is stopped. When multiple different end times are specified, such as in a scenario and a MetricSet, the simulation is stopped when the earliest end time is passed.

Scenario

A scenario consists of a number of events. A scenario may start with some settings to specify whether the co-simulation should stop at a specific time or to define the sockets that should be connected to. A socket specification requires a unique name and is tied to one of the federate instances in the federation. It requires a hostname and a port number that it should connect to. The targeted federate wrapper then connects to the specified socket upon starting and sends the messages at the specified points in time. Listing 4.9 displays a sample scenario for the RoomThermostat system. The types of events on lines 4 to 6 are elaborated in the next section.

```
1  Scenario testScenario {
2    AutoStop: 3600.0
3    Socket userInterface for thermostat on "localhost" : 9999
4    300.0: thermostat.targetTemperature = "20.5"
5    1560.0: userInterface <- 0x02 0x00 0x01
6    2700.0: thermostat.targetTemperature = "16.0"
7  }
```

Listing 4.9: *CoHLA scenario specification for the RoomThermostat system.*

The *thermostat* federate opens a socket to the listening socket on *localhost* on port 9999. During the simulation, the *thermostat* federates updates its target temperature to 20.5 ℃ after 300 seconds, after which it is changed to 16.0 ℃ at time 2700 seconds. At time 1560, the socket that mimics the user interface to the federate transmits three bytes. After 3600 seconds of simulation time, the co-simulation is stopped.

Scenarios like these can be very useful for DSE, since they allow the user to replay the exact same sequence of events for every configuration that is co-simulated. The addition of metrics to such a scenario definition makes different executions of the “interactive” co-simulation easily comparable.

Events

This section briefly describes the events that are supported by CoHLA and how these can be used.

AssignEvent An *AssignEvent* can be used to assign a specified value to an attribute. The event could mimic the user pushing a button or changing a value. The change occurs immediately after the first time step that is greater or equal to the moment at which the event should happen. Listing 4.10 shows a short sequence of assign events that adjust the target temperature of the thermostat in the RoomThermostat system.

```
300.0: thermostat.targetTemperature = "20.5"
1800.0: thermostat.targetTemperature = "16.0"
```

Listing 4.10: *CoHLA example of a sequence of assign events to set the thermostat target temperature.*

SocketEvent A *SocketEvent* requires a federation in which at least one socket is open for incoming connections. In general, this socket is used to provide an interface that allows communication with a management application that provides – for example – a graphical user interface to the user. When one of the federates in the co-simulation contains a socket in the model, a *SocketEvent* can be used to send specified messages to this socket to mimic user input. The need for a *SocketEvent* originates from the industrial case, which is described in Chapter 6. Here, two software components running on separate systems communicate with each other according to a specified protocol. To allow SIL co-simulation by replacing one component with real software, one of the models used the same protocol to communicate over a socket connection. For this, the *SocketEvent* was introduced to enable specific byte sequences being transmitted to the model at specific points in time during the simulation.

In order to specify a `SocketEvent`, the scenario should specify a socket in its settings. The socket is identified by a name, which is used in the definition of a *SocketEvent*. The *SocketEvent* specifies which message is sent to which socket at which time. The message must be provided in bytes in hexadecimal format.

```
150.0: userInterfaceA <- 0x01 0x23 0x45 0x67 0x89 0xab
270.0: userInterfaceA <- x3a xb2 x00
360.0: userInterfaceB <- 0x00 0x01 0x00
```

Listing 4.11: *CoHLA example of a sequence of socket events.*

Listing 4.11 shows a sequence of three socket events that are sent to two different sockets – named *userInterfaceA* and *userInterfaceB* – in the co-simulation. These two sockets have different names that should be specified in the settings of the scenario. A sample of these settings is displayed in Listing 4.9. Also note that the notation of the bytes can have either a `0x` or just an `x` as prefix.

Implementation

All federates that are generated by CoHLA are derived from the same base federate implementation. This implementation is capable of parsing a provided scenario configuration file upon starting. From the set of events, all events that directly affect the federate itself are stored. Upon every time step taken during the simulation, the federate applies all events that are specified for the current time.

When a scenario configuration specifies one or more sockets, the federate retrieves the sockets to which it should connect to and connects with them. Hereafter, every socket event in the scenario is transmitted to the socket at the specified time.

4.4.8 Fault scenarios

Some of the errors that might occur during the life time of a CPS are caused by faulty sensors, broken wires and unresponsive actuators. To test a design of such a CPS for robustness or fault recovery, a co-simulation of the design could be injected with faults to mimic the errors that might occur. To properly compare the different designs with each other during DSE, fault scenarios could be specified.

Fault scenarios in CoHLA are similar to regular scenarios. A fault scenario consists of one or more faults that are injected during the simulation. Since CoHLA is responsible for the the interaction between simulation models, all faults are represented as communication errors that may occur between different components, such as the changes of attributes communicated between federates. The introduction of faults that occur within a model cannot be introduced by CoHLA, as these should be part of the model that is simulated. To allow such faults to be simulated by CoHLA, these may be specified by the co-simulation framework using model parameters.

The fault scenarios supported by CoHLA focus on the reuse for DSE purposes. Hence, we do not support probabilistic scenarios that inject faults according to a certain probability distribution. The supported faults occur at given points in time, during fixed period in time or during the whole simulation.

Fault scenario specification

A fault scenario is identified by a name and consists of a set of faults as specified above. An example fault scenario is shown in Listing 4.12. The types of faults shown on lines 2 to 6 are elaborated in the next section.

```
1  FaultScenario testFaults {
2    Variance for livingroom.temperature = 0.3
3    Variance for kitchen.temperature = 0.3
4    Variance for hall.temperature = 0.3
5    From 1200.0 to 1500.0 disconnect thermostat.heaterState
6    From 1800.0 offset livingroom.temperature = -0.5
7  }
```

Listing 4.12: *An example fault scenario specified in CoHLA.*

This fault scenario introduces noise to each of the three temperature sensors in the rooms and disconnects the heater state from the rooms from time 1200 to 1500 seconds. Consequently, the heaters in all rooms will remain in the same state for that time, regardless of the actual heater state that is provided by the thermostat. Starting from time point 1800 seconds, the temperature sensor in the living room has an additional offset of -0.5 °C.

Faults

A number of faults that may occur in a co-simulation of a system are listed below.

1. Wrong attribute values are transmitted.
2. Updates are lost.
3. Updates are duplicated.
4. Timing of updates is disturbed – timestamp is changed.

CoHLA supports the first two faults in different forms. Due to the fact that HLA guarantees that all updates are transmitted before granting a time step, fault 3 only causes the CoHLA wrapper to overwrite an attribute value with the same value, thus not changing the model simulation. This type of fault is therefore not supported by CoHLA. Fault 4 is also limited by HLA, as it is not possible to transmit an update with a timestamp before its own logical time (plus its lookahead). Delaying an attribute update is possible in HLA, but currently not supported by CoHLA.

Based on the faults listed above, CoHLA supports four types of faults to be injected in the co-simulation. Each type of fault mimics faults that may occur in the real system. These types are briefly described in this section.

Value fault A value fault overrides the value of an attribute for a specified period in time. The fault mimics a broken sensor that transmits a faulty value. It requires a target attribute and a value to fix the attribute to. Also, a moment or period in time should be specified. Listing 4.13 shows three examples of value faults.

```
On 30.0 set thermostat.heaterState = "false"
From 45.0 set livingroom.temperature = "17.3"
From 63.0 to 78.0 set kitchen.temperature = "16.3"
```

Listing 4.13: *Examples of value faults in CoHLA.*

Offset fault An offset fault specifies an offset instead of a fixed value. It enables the user to add or subtract a specified number to or from the attribute for a given period during the simulation. An offset fault can be used to mimic a broken or poorly calibrated sensor in the co-simulation. Similar to a value fault, an offset fault also requires a target attribute, offset value and a moment or period in time in which the fault occurs. Listing 4.14 shows three examples of offset faults.

```
From 0.0 offset livingroom.temperature = +0.3
From 60.0 to 90.0 offset kitchen.temperature = -0.24
On 150.0 offset hall.temperature = +0.8
```

Listing 4.14: *Examples of offset faults in CoHLA.*

Connection fault A connection fault mimics a (temporary) lost connection between two attributes. While the fault is active, no new attribute values will be received by federates subscribed to the targeted attribute. A connection fault requires a target attribute and a moment or period in time for which the fault occurs. Listing 4.15 shows an example of a connection fault.

```
From 30.0 to 120.0 disconnect thermostat.heaterState
```

Listing 4.15: *Example of a connection fault in CoHLA.*

Variance A variance fault adds noise to the value of an attribute. Even though this fault incorporates a probabilistic component, it is supported to mimic the use of non-perfect wires and sensors. A variance fault is applied to the target attribute during the whole simulation and cannot be applied only for a specified period of time. Noise is added according to a normal distribution with mean 0.0 and a standard deviation that is specified. Listing 4.16 shows an example in which variance is applied to an attribute.

```
Variance for livingroom.temperature = 0.3
```

Listing 4.16: *Example of adding variance to an attribute in CoHLA.*

Implementation

For each fault scenario a configuration file is generated that can be provided to the run script upon starting a co-simulation. Every federate in the federation parses the configuration and stores the faults that impact the federate. During the simulation, faults are applied at the receiving side instead of applying these before distributing the attribute values. Reason for this is that it allows the correct values to be logged by the logger federate, since these values have not been corrupted by the fault. Upon receiving an attribute update from the RTI, the federate first checks whether a fault should be applied to the value. If a fault should be applied, it is applied before further processing. This mechanism is implemented in the libraries provided with CoHLA. Figure 4.4 shows the location of the mapper in the HLA federation architecture that applies faults to incoming attribute updates.

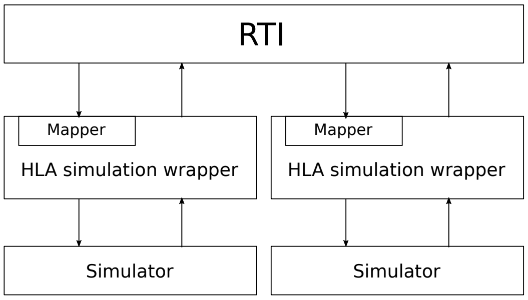


Figure 4.4: *The mapper applies faults to attribute values that are received by the wrapper before these are updates in the simulator.*

Following this implementation, a connection fault basically ignores the incoming attribute update while both the offset and value faults overwrite the value before further processing. Since every federate applies faults by itself on incoming attributes, the variance that is applied on the same attribute value is different for each receiving federate. Noise is introduced on the connection from one component to another instead of being applied before the attribute value is sent. Introduction of variance is therefore suitable mainly for modelling connection inaccuracies rather than sensor inaccuracies.

4.4.9 Performance metrics

Individual co-simulations produce large amounts of simulation data. During DSE, many different configurations of the same co-simulation are simulated. To extract relevant information from each of these co-simulations, all data should be processed. This is particularly time consuming when the design space is large.

The calculation of performance metrics is a method that allows easy comparison of these sets of data, since performance metrics are capable of summarising simulation data into just one value [66, 50]. CoHLA supports the calculation of four types of performance metrics during the co-simulation execution. When a DSE is executed using the run script and one or more metrics are being measured, the DSE configurations and their metrics are exported to one file.

MetricSet

For every CoHLA federation specification multiple sets of metrics may be defined. Each *MetricSet* has a unique name and a *MeasureTime* to specify for how long the co-simulation must be executed. A *MetricSet* can be provided to the run script that starts the co-simulation to retrieve the metrics in the set from the co-simulation. An example of a *MetricSet* for the RoomThermostat system is shown in Listing 4.17. The types of metrics are explained in the next section.

```
MetricSet sampleMetricSet {
  MeasureTime: 3600.0
  Metric evLivingroomTemp as EndValue livingroom.temperature
  Metric minLivingroomTemp as Minimum of livingroom.temperature
  Metric errLivingroomTemp as Error livingroom.temperature relative to
      thermostat.targetTemperature
  Metric overheatingTime as Timer for livingroom.temperature >= 21
      EndCondition
}
```

Listing 4.17: *CoHLA example of a MetricSet specification.*

The shown *MetricSet* calculates all metrics over a simulation period of 3600 seconds, except for when the end condition *overheatingTime* metric is met. When this metric is finished – i.e. the living room temperature exceeds 21 °C – the simulation is stopped. When the simulation stops, the mean error of the living room temperature compared to the target temperature is calculated and the minimum temperature in the living room is returned. The final temperature in the living room is also returned. These metric results are gathered together in one single file having the name of the *MetricSet*.

Metric types

This section briefly introduces the four basic metrics that are currently implemented in CoHLA. The first three metrics are implemented because these represent commonly used characteristics the could be retrieved from a co-simulation execution. While these could be calculated from the resulting log files, automating the calculation during the co-simulation execution simplifies the process of comparing different configurations of the system with each other. The *timer metric* was introduced to measure the initialisation time for the case study described in Chapter 6. While these four performance metrics are currently implemented in CoHLA, these primarily illustrate an approach to incorporate the automatic calculation of system characteristics from a co-simulation. This approach could be used to add more complex performance metrics to CoHLA in the future.

End value metric The *end value metric* returns the value of an attribute at the end of the co-simulation. Optionally, the absolute attribute value is returned. The value that is returned could also be relative to another attribute, which can be used to show the difference between two attributes at the end of the co-simulation. Listing 4.18 shows three sample definitions of the end value metric.

```
Metric evMetric1 as EndValue livingroom.temperature
Metric evMetric2 as Absolute EndValue livingroom.temperature
Metric evMetric3 as Absolute EndValue livingroom.temperature relative to
    thermostat.targetTemperature
```

Listing 4.18: *CoHLA examples of end value metric definitions.*

When these metrics are included in the *MetricSet* displayed in Listing 4.17, the metric *evMetric1* would return the value of the living room temperature at time 3600, since this is the specified measure time for the *MetricSet*. Metric *evMetric2* would return the absolute value of the same attribute: $|livingroom.temperature|$. Metric *evMetric3* would return the absolute end value difference between the temperature of the living room and the target temperature of the thermostat: $|livingroom.temperature - thermostat.targetTemperature|$.

Minimum or maximum value metric The *minimum value metric* returns the lowest value of an attribute that is reached during the simulation. Similarly, the *maximum value metric* returns the highest value of the attribute during the simulation. Listing 4.19 shows a sample for each of the two variants of this metric.

```
Metric mmMetric1 as Minimum of livingroom.temperature
Metric mmMetric2 as Maximum of kitchen.temperature
```

Listing 4.19: *CoHLA examples of minimum and maximum value metrics.*

The metric *mmMetric1* would return the minimum temperature of the living room that is reached during the execution of the co-simulation. The time to measure is specified in the *MetricSet* or is limited by the user when manually stopping the co-simulation. Metric *mmMetric2* returns the maximum value of the temperature in the kitchen reached during the simulation.

Error metric The *error metric* calculates the mean error value of one attribute relative to another attribute for the duration that is specified in the MetricSet, as explained in Section 4.4.9. Optionally, the mean squared error can be calculated.

Listing 4.20 shows two examples of how the error metric can be defined in a MetricSet.

```
Metric erMetric1 as Error livingroom.temperature relative to thermostat.targetTemperature
Metric erMetric2 as Squared Error kitchen.temperature relative to thermostat.targetTemperature
```

Listing 4.20: *CoHLA examples of error metrics.*

The error value E is calculated by the following formula: $E = \frac{1}{n}\sum_{t=0}^{n} x_t - y_t$. Here, n is the measured time, x is the first attribute and y is the second attribute. The point in time is represented by t. The squared error is calculated similarly by the following formula: $E = \frac{1}{n}\sum_{t=0}^{n}(x_t - y_t)^2$.

Timer metric The *timer metric* can be used to find the first moment for which a specified condition holds. The condition allows basic numerical comparison and comparing boolean values. This type of metric was introduced to measure the time taken by the initialisation procedure of the SliderSetup, which is described in Chapter 6. Here, the metric could be used to measure the first moment in time for which an *initialised* flag is `true`. A timer metric may be marked to be an end condition. When all timer metrics that are marked as end conditions are finished the simulation is stopped. This is the case when all timer metrics marked as end conditions have found a simulation time for which its condition was true. This can be used to minimise the simulation time required to measure metrics to speed up the DSE process. When the timer metric is set to be an end condition, a delay may be specified to allow the simulation to run for the specified amount of time after meeting the condition. Listing 4.21 shows a number of examples of timer metrics in CoHLA. Note that the metric on line 4 is an end condition with a specified delay.

```
1 Metric tiMetric1 as Timer for livingroom.temperature >= 20
2 Metric tiMetric2 as Timer for kitchen.temperature < 16.0
3 Metric tiMetric3 as Timer for thermostat.heaterState == true EndCondition
4 Metric tiMetric4 as Timer for thermostat.heaterState == false EndCondition
      after 1.5
```

Listing 4.21: *CoHLA examples of timer metrics.*

Implementation

Metrics are calculated by a metrics processor, which is a dedicated federate in the HLA co-simulation. The metrics processor implementation is similar to the logger implementation in CoHLA and is part of the libraries provided by CoHLA. Source code for the metrics processor is generated by the CoHLA code generator when a MetricSet has been specified in the federation specification. The processor is started by the run script upon starting the co-simulation when a MetricSet configuration has been provided.

The metrics processor subscribes to all relevant attributes from other federates and collects their values during the simulation. When the co-simulation is ended or when the measure time – as specified in the MetricSet – is exceeded, the metrics are calculated from the data that was collected. The metric results are then printed to an output text file.

4.4.10 Design space exploration

During the design of a CPS, many decisions have to be made. Component implementations, material choices and trade-offs have to be decided on during the design. The process of investigating design or implementation alternatives is called Design Space Exploration (DSE) [39, 57].

Early-stage co-simulation of component models could provide in-depth insight in the consequences of design decisions [51]. Information from the co-simulation of the system can be used to select the best fitting design. This approach helps the designers to select a design path that appears to be most promising after having compared a number of alternatives. Automatic DSE co-simulation is capable of running a large set of different configurations of a co-simulation without requiring human supervision and would speed up the DSE process even more.

During such an automated DSE execution a large amount of simulation data is produced. To compare these results with each other, manually comparing the resulting logs or applying analysis techniques on them would require quite some manual labour. The measurement of performance metrics could be used to perform these analyses during the co-simulation execution.

Some system simulations require user input, which cannot be provided manually during an automated DSE execution. Hence, such user scenarios could also be included in the DSE to ensure that each co-simulations all use the same input.

CoHLA supports the specification of a design space. The specified design space can automatically be co-simulated by the generated run script. This approach requires all parameters to be specified and does not allow for automatic parameter optimisation in any way. It is possible to extend CoHLA to use an external tool for this, but this is still future work.

The specification of scenarios and fault scenarios as described in Sections 4.4.7 and 4.4.8 allow the user to run a predefined sequence of events in a co-simulation. Adding metrics, as introduced in Section 4.4.9, to the co-simulation improves the usability of the results by comparing them with each other. CoHLA allows the user to create a DSE configuration that specifies the configurations of the system to simulate. Every DSE configuration has a unique name and optionally a scenario and fault scenario. To specify the system configurations, situations, parameter configurations and individual parameters can be used. These have already been described in Section 4.4.4.

Federate	**Attribute**	**DS**
Federate 1	Attribute A	x, y, z
	Attribute B	a, b, c
Federate 2	Attribute C	q
	Attribute D	u, v, w

Table 4.1: *Hypothetical design space (DS).*

Table 4.1 displays a hypothetical design space to illustrate the use of DSE configurations. Almost every attribute in this federation has three possible configurations, listed in column *DS*. CoHLA supports two DSE sweep modes: *independent* and *linked*. Independent DSE combines all possible parameter values, configurations or situations with all others. Following this approach, the hypothetical DSE shown in Table 4.1 results in 27 possible configurations: (x, a, q, u), (x, a, q, v), $\ldots$, (z, c, q, w). Linked DSE combines the n-th parameter value, configuration or situation with all other n-th values in the lists. It is required that all lists have equal lengths for this method. Consequently, *Attribute C* cannot be part of this DSE configuration. When this attribute is ignored, there are three design spaces using the linked method: (x, a, u), (y, b, v) and (z, c, w).

For the definition of the design space in CoHLA, these lists could be defined on different levels. These three levels are listed below.

- **Federation level**: A list of situations can be used to specify the values for the model parameters in the co-simulation. Such a situation consists of parameter configurations applied to federates, model parameter values or both to specify the configuration of the federation.
- **Federate level**: For each federate in the federation, a list of parameter configurations can be specified. These parameter configurations refer to already defined valuations of the model parameters that should be simulated.
- **Parameter level**: For every model parameter for every federate in the federation, a list of possible values could be provided.

Design spaces are often specified at a parameter level, which is also displayed in the table. Since CoHLA provides ways to reuse sets of parameters by means of situations and parameter configurations, these can also be used to specify a design space for DSE. Listing 4.22 shows what the DSE configuration for the hypothetical design space in Table 4.1 would look like in CoHLA specified on parameter level using independent sweep mode.

```
DSE HypotheticalDSE {
  SweepMode Independent
  Set Federate1.AttributeA: x, y, z
  Set Federate1.AttributeB: a, b, c
  Set Federate2.AttributeC: q
  Set Federate2.AttribureD: u, v, w
}
```

Listing 4.22: *CoHLA specification for the hypothetical design space displayed in Table 4.1.*

For every specified DSE configuration a configuration file is generated that describes all co-simulation configurations. This file can be provided to the run script to run these configurations sequentially. Optional scenarios and fault scenarios specified in the DSE configuration are applied automatically and log files are stored in a directory dedicated for the DSE execution. A metric set containing the metrics of interest may be provided to the run script as well. When a metric set is provided, a file containing the results of each configuration is stored in the appropriate directory. Additionally, the DSE execution automatically groups these results into one single file for the whole DSE to allow easy comparison of the configurations.

The support for DSE in CoHLA is limited to the automatic execution and (partial) collection of simulation results for a pre-specified finite design space. DSE using CoHLA is therefore not suitable for very large design spaces or when a certain optimal parameter configuration should be found automatically without specific boundaries.

4.4.11 Collision detection

For the industrial case study discussed in Chapter 6 a collision detection simulator was needed. CoHLA therefor supports a simulator type that allows the incorporation of 3D models into the co-simulation. This simulator consists of two parts: a renderer for the 3D models and a collision detector. The renderer can be used to visualise the system under design by means of the 3D models provided while the collision detector uses the same 3D models to check the system for collisions. The simulator receives its inputs, i.e. locations, from dynamic models being simulated by other simulators.

The CoHLA collision library uses the Bullet Real-Time Physics Simulation engine[4] for both rendering and collision detection. The library itself implements the bridge from the HLA simulation to the Bullet engine and CoHLA generates a federation-specific wrapper for every collision federate included in the simulation. Listing 4.23 displays a collision detection federate specification for the SliderSetup that is described in Chapter 6.

```
FederateClass CollisionDetector {
  Type BulletCollision
  Attributes {
    Input Real topSliderPosition as "axis1"
    Input Real bottomSliderPosition as "axis2"
    Output Integer [Collision] collisions
    Output Boolean [Collision] hasCollisions
  }
  DefaultModel "models/sliders.json"
  DefaultLookahead 0.00001
}
```

Listing 4.23: *Collision detector federate specification for the SliderSetup described in Chapter 6.*

The federate class is defined as a *BulletCollision* type of federate with a number of attributes. The first two input attributes represent the positions of the two axes, each being an alias for their names – *axis1* and *axis2* – in the collision model. Output attributes for the collision detector can be either `integer` or `boolean` attributes, which represent the number of collisions or whether there is a collision respectively. The keyword `[Collision]` is used to bind the output attribute to the appropriate output of the collision detector library. The collision model itself is defined in JSON format. A part of such a model is shown in Listing 4.24.

```
{
  "states":[ "axis1", "axis2" ],
  "viewpoint": {
    "position": [ 0, 10, 1000 ],
    "lookAt": [ 0, 0, 0 ]
  },
  "bodies": {
    "bottomSlider": {
      "meshes": {
        "carriage": {
          "type": "stl",
          "file": "models/Double_slider_simplified_bottom_carriage.STL",
          "collider": false,
          "render": true,
          "offset": [0,0,0]
        },
        "rod": {
```

[4]https://pybullet.org/wordpress/

```
18            "type": "stl",
19            "file": "models/Double_slider_bottom_rod.STL",
20            "collider": true,
21            "render": false,
22            "offset": [0,0,0]
23          }
24        },
25        "transforms": [
26            {
27              "origin": [1000,0,0],
28              "rotation": [0,0,0],
29              "state": "axis2"
30            }, {
31              "origin": [0,0,0],
32              "rotation": [0,0,0],
33              "state": ""
34            }
35          ]
36      },
```

Listing 4.24: *A part of the collision model for the SliderSetup described in Chapter 6.*

First, line 2 specifies the input *states* of the model. These input states correspond with the names to which the input aliases are mapped in Listing 4.23. The *viewpoint* specifies the initial position of the camera and its aim. The *bodies* element contains a map of 3D elements in the system. Every 3D element has a name – such as *bottomSlider* on line 8 – and contains one or more meshes, which are the actual 3D representations, and a list of transformations. The meshes – here *carriage* and *rod* on lines 10 and 17 – are either defined as an external STL file [59] or one of the basic 3D models such as a cube or ball. Every mesh has an offset relative to coordinate (0, 0, 0) that allows the designer to correctly position the component in the system. For every mesh it can be specified whether it is a collider and whether it should be rendered. Only meshes that have been marked to be a collider are checked for collisions with other colliders. The list of transformations is used to move or rotate the 3D element based on one or more state values.

One or more of these collision models may be provided to a collision detector federate. Appending more models overwrites already existing elements with identical names, allowing a modular approach to reuse small collision models. Optionally, a visualisation of the 3D models may be rendered for the user by providing the `render` flag to the run script upon starting the co-simulation. Figure 4.5 shows the rendered 3D models by the collision detector during co-simulation.

4.5 Code generation

From a co-simulation specification as described in Section 4.3, a number of files are generated. Wrapper code extending the CoHLA libraries is generated for each of the federate classes that is defined. This wrapper implementation compiles to an executable that connects the simulation model to the RTI. A FOM – such as Listing 3.2 – is generated that is required for creating an HLA federation. Configuration files that contain the parameter configurations and situations are also generated. Finally, a Python run script is generated that enables the user to build and run the co-simulation easily. The run script provides three basic methods to the user.

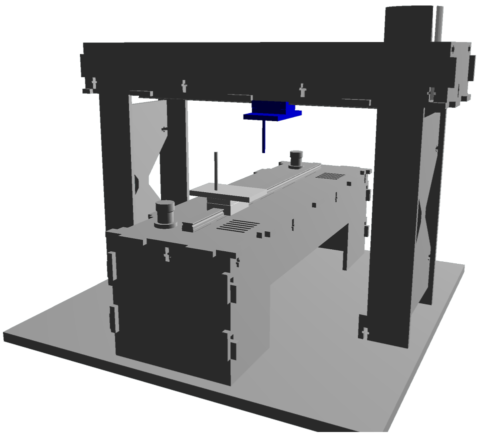


Figure 4.5: *SliderSetup as rendered by the collision detector during co-simulation.*

- **Build**: Compiles all federate class wrappers and stores the executables in a folder with the name “build”.
- **Run**: Starts the co-simulation execution with the parameters provided to the run script or the default parameters. This method allows for customisation by providing generated configurations to the run script to modify the co-simulation initialisation. For example, a scenario or metric set may be provided to the run script, after which these are executed or measured. When a DSE configuration is provided, the co-simulations contained in the design space are automatically started. Starting a co-simulation triggers the run script to start the RTI, all simulators and wrappers in separate threads.
- **Display**: Prints the current configuration with respect to the provided arguments of the federation. This method does not start a simulation, allowing the user to verify whether the correct configurations are used.

4.6 RoomThermostat system in CoHLA

To illustrate the use of CoHLA and a number of its features, it is used to construct a co-simulation of the RoomThermostat system. The model that is used for the three rooms is the 20-sim model exported to an FMU, using the libraries described in Section 4.2.1. The POOSL model of the thermostat is used for co-simulation. A logger is added and the CoHLA specifications of the co-simulation environment, room federate, thermostat federate and federation itself are already listed in Section 4.3.

```
FederateClass Logger {
  Type CSV-logger {
    DefaultMeasureTime 3600.0
  }
}
```

Listing 4.25: *Federate class definition for the logger in CoHLA. Information on this specification can be found in Section 4.3.2 of the CoHLA user manual.*

The base CoHLA co-simulation specification of the RoomThermostat system is the combination of Listings 4.1, 4.2, 4.3 and 4.4. Only the definition of the federate class for the logger is missing. This specification is displayed in Listing 4.25.

Three different co-simulation executions were run, where each illustrates different features of the CoHLA framework. In Section 4.6.1 a basic co-simulation of the RoomThermostat is executed to show the use of parameter configurations. A sample of using a fault scenario is shown in Section 4.6.2 and design space exploration is applied to the co-simulation in Section 4.6.3.

4.6.1 Basic RoomThermostat

The basic RoomThermostat co-simulation consists of the three rooms, a thermostat and a logger and is specified by Listings 4.1 to 4.4 and 4.25. From these CoHLA specifications the framework generates wrappers, configuration files and a run script. The wrappers consist of little over 700 lines of C++ code and a CMake project file. The FOM for the federation contains 150 lines and a topology file of 16 lines was generated. Since the default file paths to the models have already been specified, the wrappers only have to be compiled before the co-simulation can be started. To compile the sources, only one call to the run script needs to be done, after which the run script can be used to start the co-simulation. Without any more configuring, the co-simulation simulates one hour (3600 seconds), as this is specified in the federate class of the logger (Listing 4.25).

Since not all rooms are equal, different sets of parameter values are specified. This mimics the behaviour of three different rooms within a house instead of having three identical rooms. For the parameters *RadiatorSize*, *WindowSize* and *Surface* different values were specified. Table 4.2 displays the parameter values that were used in this co-simulation. Note that the chosen parameter values are for illustrative purposes only.

Parameter	**Livingroom**	**Kitchen**	**Hall**
Surface	45.0	10.0	4.5
Window size	11.0	2.0	2.5
Heater size	1.0	0.25	0.1

Table 4.2: *Parameter values for the three rooms.*

The target temperature of the thermostat is set at 20.5 °C. These parameter values are all specified in configurations, which are then grouped into a situation. The configuration file for the situation is then passed to the run script to start the co-simulation with the correct parameter values. Figure 4.6 shows the simulation results for the co-simulation.

From this figure, it is clear that the thermostat receives its input from the living room, as this is the room for which the temperature is maintained around the target temperature. Since both the surface and the window size of the kitchen are smaller compared to the living room, while the heater is relatively large, the temperature in the kitchen rises a little faster than the temperature in the living room. Consequently, the temperature in this room is higher than the target temperature. Although the hall is even smaller, its heater is clearly not large enough

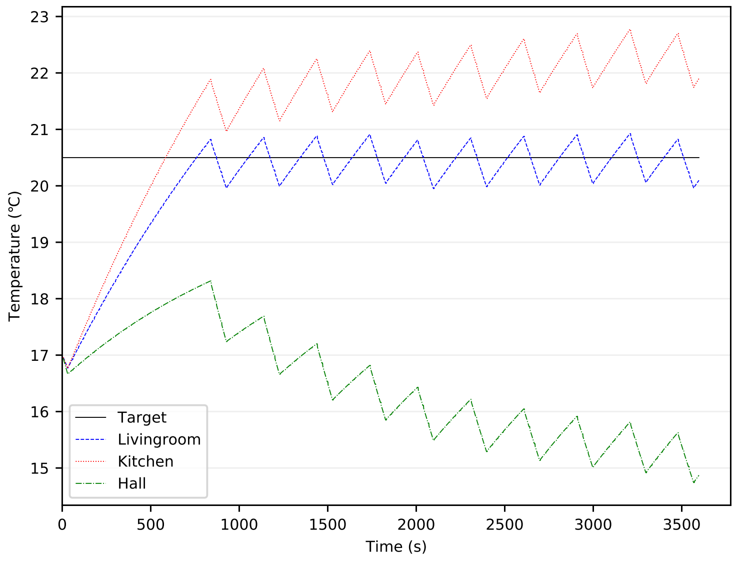


Figure 4.6: *Co-simulation results for basic RoomThermostat system.*

to compensate for the losses that are caused by the size of the window (which models the door) in this room. These results can be used to balance the sizes of the heaters to fit the other characteristics of each room to maintain a comfortable temperature throughout the house before having to build it.

4.6.2 RoomThermostat fault scenario

To illustrate the use of a fault scenario in a CoHLA co-simulation, a number of faults are injected into the co-simulation of the RoomThermostat. The system being co-simulated is identical to the one described in Section 4.6.1. A logger stores the attribute values during the co-simulation, which simulates a period of 3600 seconds – one hour. The characteristics of the rooms are identical to those displayed in Table 4.2. Listing 4.26 shows the faults that are simulated in the co-simulation.

```
FaultScenario BrokenCables {
  Variance for Livingroom.Temperature = 0.03
  From 300.0 to 480.0 set Thermostat.HeaterState = "false"
  From 1500.0 to 2100.0 disconnect Livingroom.Temperature
  From 2700.0 disconnect Thermostat.HeaterState
}
```

Listing 4.26: *RoomThermostat fault scenario with four faults.*

The faults are briefly elaborated below.

- During the whole simulation, variance with a standard deviation is applied to the temperature being transmitted from the living room. This variance may cause the thermostat to switch the heater state too late or too early compared to when no variance would be applied.

- From 5 to 8 minutes (simulation time, 300 to 480 seconds), the heater state as outputted from the thermostat is set to `false` (off). The expected consequence is that the temperature in all three rooms should decrease during this time.
- From 25 to 35 minutes (simulation time, 1500 to 2100 seconds), the thermostat will not receive any more updates from the living room, thus base its target heater state on an old value. Only the thermostat is affected by this, since it is the only federate that receives updates from the living room. The expected consequence of this fault is that the heater state as outputted by thermostat will not change during this period.
- Starting from 45 minutes (simulation time, 2700 seconds), the heater state of the thermostat is not transmitted at all to the rooms. The expected consequence is that the heater state of the rooms remains on its old value, as it was previous to 2700 seconds.

Figure 4.7 displays the results of the co-simulation of the RoomThermostat system with injected faults. The points in time when the faults are injected and when the effects end are marked by vertical lines. The effects of the faults can be recognised at the provided time points. To compare the base co-simulation with the co-simulation where faults are injected, Figure 4.8 displays the temperature values of the living room in both co-simulations.

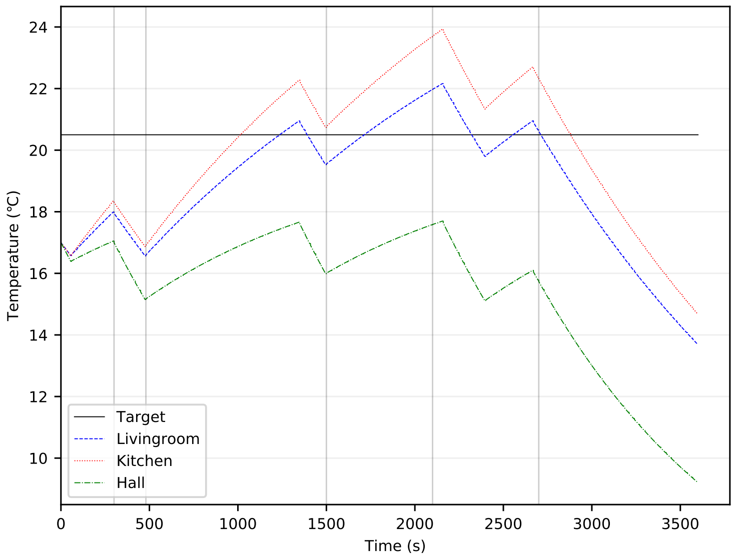


Figure 4.7: *Co-simulation results for RoomThermostat system with injected faults. The points in time when the faults are injected and when they end are marked by vertical lines.*

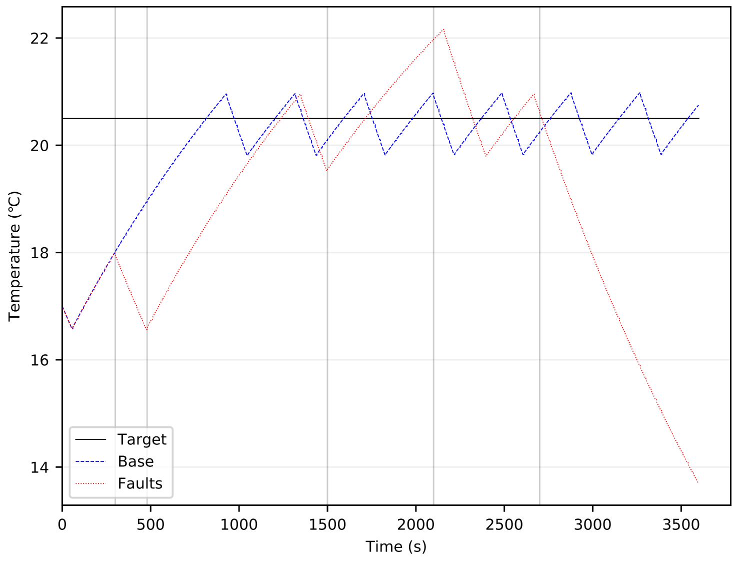


Figure 4.8: *Comparison of the living room temperature for the base co-simulation and the co-simulation with fault injected. The points in time when the faults are injected and when they end are marked by vertical lines.*

4.6.3 RoomThermostat design space exploration

DSE will be applied to the RoomThermostat system to illustrate the use of DSE in CoHLA. The RoomThermostat consists of three rooms, all having different characteristics, which have already been described in Section 4.6.1. Since the surface area and window size of each of the rooms are different, these rooms would require different sizes of radiators to maintain the temperature at a comfortable level. We will use DSE to find acceptable sizes for the radiators in the rooms, so that these are capable of maintaining the temperature in the rooms. For this co-simulation, the 20-sim models of the components will be used, as described in Section 2.2.1.

Scenario

In order to gain insight in the temperatures in the rooms through a whole day, a scenario is created to mimic the presence (and absence) of a user in the house. The scenario covers an ordinary working day that starts at midnight and ends 24 hours later. The thermostat is turned up in the morning to wake up comfortably and lowered to get to work. When arriving home, the thermostat is turned up again, after which

Time	Target temperature
12:00 A.M.	14.0 °C
06:00 A.M.	18.5 °C
07:00 A.M.	16.0 °C
04:00 P.M.	19.5 °C
06:00 P.M.	20.0 °C
08:00 P.M.	20.5 °C
10:30 P.M.	14.0 °C

Table 4.3: *Working day scenario.*

the target temperature is incremented twice during the evening. When going to bed, the thermostat is lowered again. This scenario is displayed in Table 4.3. Listing 4.27 displays the scenario as it is specified in CoHLA.

```
Scenario RegularDay {
  AutoStop: 86400.0
  0.0: Thermostat.targetTemperature = "14.0"      // 12:00 AM
  21600.0: Thermostat.targetTemperature = "18.5"  // 06:00 AM
  25200.0: Thermostat.targetTemperature = "16.0"  // 07:00 AM
  57600.0: Thermostat.targetTemperature = "19.5"  // 04:00 PM
  64800.0: Thermostat.targetTemperature = "20.0"  // 06:00 PM
  81000.0: Thermostat.targetTemperature = "14.0"  // 10:30 PM
}
```

Listing 4.27: *Working day scenario as specified using CoHLA.*

Metrics

When running a set of different configurations of the RoomThermostat system, each with different sizes of the radiators, it is convenient to specify a set of metrics to easily compare the co-simulation results. Since it is hard to measure the comfortableness of a room, a suitable metric to measure is the error of the temperature of each of the rooms relative to the target temperature as specified by the user. The results should show whether the radiator in the room is capable of maintaining the target temperature, even though the thermostat is located in the living room and therefor only receives temperature updates from this room. Zero error means that a room perfectly follows the target temperature. Any other values reflect the average temperature deviation from the target temperature during the simulation.

```
MetricSet Errors {
  MeasureTime: 86400.0
  Metric errLivingroom as Error Livingroom.temperature relative to Thermostat
      .targetTemperature
  Metric errKitchen as Error Kitchen.temperature relative to Thermostat.
      targetTemperature
  Metric errHall as Error Hall.temperature relative to Thermostat.
      targetTemperature
}
```

Listing 4.28: *MetricSet specification for the RoomThermostat system.*

Listing 4.28 shows the metrics as they have been specified in CoHLA for the system. The time during which the metrics should be measured is specified as 86400 seconds, which matches the duration of the scenario.

Design space

To find a size for each radiator that fits the room in which it is located, the design space only alternates the sizes of these radiators. The surface area and window size of each of the rooms remains fixed. Table 4.4 displays the design space of the RoomThermostat system that was selected.

To provide a base configuration of the system, a situation configuration is used that applies different parameter configurations to the different rooms in the federation. These configurations are displayed in Listing 4.29. The configurations

	Surface area	**Window size**	**Radiator sizes**		
Living room	45.0	11.0	0.5	0.75	1.0
Kitchen	10.0	2.0	0.1	0.25	0.4
Hall	4.5	2.5	0.1	0.125	0.15

Table 4.4: *Design space of the RoomThermostat system targeting the radiator size.*

specify the sizes of the rooms as well as their windows and the situation *Default-Base* applies them all end sets the initial target temperature of the thermostat.

```
Configuration Large for Room {
  Surface = "45.0"
  WindowSize = "11.0"
}

Configuration Medium for Room {
  Surface = "10.0"
  WindowSize = "2.0"
}

Configuration Small for Room {
  Surface = "4.5"
  WindowSize = "2.5"
}
Situation DefaultBase {
  Apply Large to Livingroom
  Apply Medium to Kitchen
  Apply Small to Hall
  Init Thermostat.targetTemperature as "14.0"
}
```

Listing 4.29: *Parameter and situation configurations for the RoomThermostat system.*

Since the situation "DefaultBase" provides a basic set of model parameter values for each of the federates, it should be included in each of the configurations in the DSE. This situation is therefor added to the DSE configuration. To combine every possible radiator size with the other sizes, the sweep mode is set to independent. Also, the scenario as specified before is included. Listing 4.30 shows the resulting DSE specification in CoHLA. The radiator sizes displayed in Table 4.4 are provided as parameter values.

```
DSE HeaterSizes {
  SweepMode Independent
  Scenario RegularDay
  Situations: DefaultBase
  Set Livingroom.RadiatorSize:  0.5, 0.75, 1.0
  Set Kitchen.RadiatorSize:     0.1, 0.25, 0.4
  Set Hall.RadiatorSize:        0.1, 0.125, 0.15
}
```

Listing 4.30: *DSE configuration for the RoomThermostat system. Information on this specification can be found in Section 4.6.6 of the CoHLA user manual.*

Results

From the aforementioned CoHLA configurations, configuration files are generated for each of parameter configuration definitions as well as one configuration file for the situation. Additionally, the scenario, metric set and DSE configuration generate one configuration file each. The DSE execution is started using the run script that is also generated with the federation. The 27 different configurations of the co-simulation are automatically executed after which a file is generated that presents each of the configurations together with the metric results. To easily compare the configurations with each other, the mean error of the individual absolute error values is calculated afterwards by post-processing the file that contains the metric results. The results are displayed in Table 4.5. For convenience, the configurations are ordered ascending by their mean absolute error.

ID	$\mathbf{R_L}$	$\mathbf{R_K}$	$\mathbf{R_H}$	$\mathbf{E_L}$	$\mathbf{E_K}$	$\mathbf{E_H}$	**Mean |E|**
1	0.500	0.100	0.125	-0.052	0.613	0.543	0.403
2	0.500	0.100	0.100	-0.052	0.613	-0.983	0.549
3	0.500	0.100	0.150	-0.052	0.613	1.839	0.835
4	0.750	0.100	0.150	0.004	-1.715	-0.791	0.837
5	0.750	0.100	0.125	0.004	-1.715	-1.827	1.182
6	1.000	0.250	0.150	0.013	1.554	-2.132	1.233
7	0.750	0.250	0.150	0.004	3.411	-0.791	1.402
8	1.000	0.250	0.125	0.013	1.554	-3.036	1.534
9	0.750	0.100	0.100	0.004	-1.715	-3.048	1.589
10	1.000	0.100	0.150	0.013	-2.903	-2.132	1.683
11	0.750	0.250	0.125	0.004	3.411	-1.827	1.747
12	1.000	0.250	0.100	0.013	1.554	-4.101	1.889
13	1.000	0.400	0.150	0.013	3.576	-2.132	1.907
14	1.000	0.100	0.125	0.013	-2.903	-3.036	1.984
15	0.750	0.250	0.100	0.004	3.411	-3.048	2.154
16	0.750	0.400	0.150	0.004	5.735	-0.791	2.177
17	1.000	0.400	0.125	0.013	3.576	-3.036	2.208
18	1.000	0.100	0.100	0.013	-2.903	-4.101	2.339
19	0.750	0.400	0.125	0.004	5.735	-1.827	2.522
20	0.500	0.250	0.125	-0.052	7.049	0.543	2.548
21	1.000	0.400	0.100	0.013	3.576	-4.101	2.563
22	0.500	0.250	0.100	-0.052	7.049	-0.983	2.695
23	0.750	0.400	0.100	0.004	5.735	-3.048	2.929
24	0.500	0.250	0.150	-0.052	7.049	1.839	2.980
25	0.500	0.400	0.125	-0.052	9.967	0.543	3.521
26	0.500	0.400	0.100	-0.052	9.967	-0.983	3.667
27	0.500	0.400	0.150	-0.052	9.967	1.839	3.953

Table 4.5: *DSE results for the RoomThermostat. R refers to the size of the radiator while E refers to the mean temperature error relative to the target temperature in the room. Rooms are identified by L (Livingroom), K (Kitchen) and H (Hall). The mean absolute value is calculated from the absolute error values.*

Since the thermostat is located in the living room and receives its temperature input from this room, the measured error is smallest in the living room. The other rooms show a wide range of possible mean error values, ranging from the room being 4 °C too cold to 9 °C too hot, on average. This wide variety is explained by the fact that the frequency at which the thermostat toggles the heater states is determined by the size of the radiator in the living room. The size of this radiator therefore indirectly impacts the temperature behaviour of the other rooms. From the possible configurations that have been simulated, ID 1 appears most promising: the kitchen and hall are off by slightly more than half a degree on average. Figure 4.9 shows the co-simulation results using this configuration.

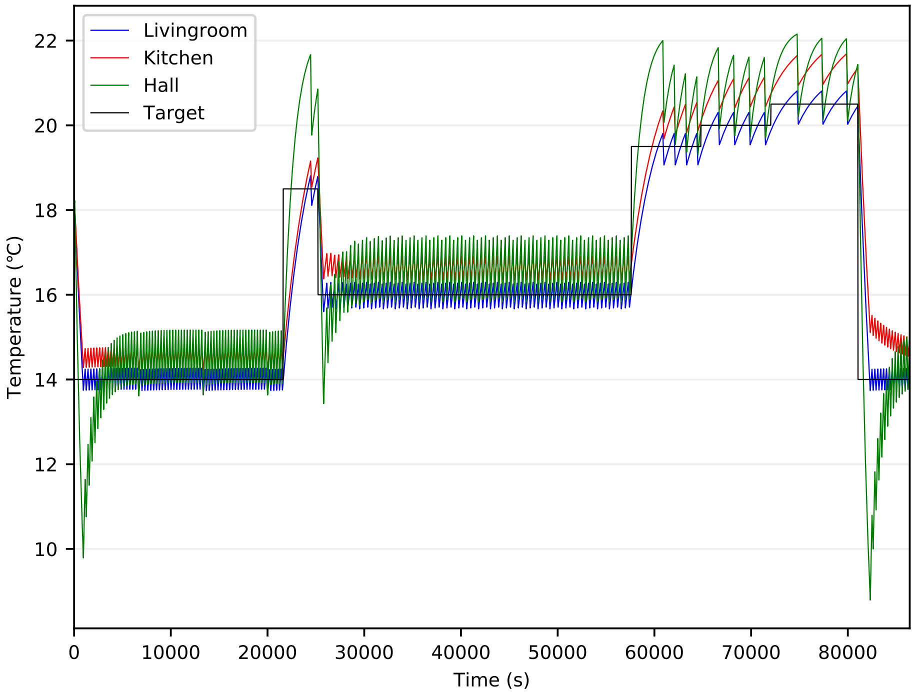


Figure 4.9: *The co-simulation results for the configuration with ID 1.*

Figure 4.10 displays the results for the configuration with ID 15. This configuration leads to temperature differences of roughly 3 °C higher (kitchen) or lower (hall) compared to the target temperature.

Depending on the application and selected parameter values, the DSE execution may also be used as a starting point for further investigation. Variations could be applied to the configuration that appeared to be the best configuration so far to find an optimum by using DSE again. This cycle may repeat itself until the moment the configuration of the system reaches a point at which for example all requirements are met.

Once the system has been specified as a federation in CoHLA, adding a design space to explore is relatively simple. The different configurations in the design space can be co-simulated automatically using the run script and basic metrics can be measured. Altogether, applying DSE to a small case such as the RoomThermostat proved to be rather simple and delivered useful results to select suitable

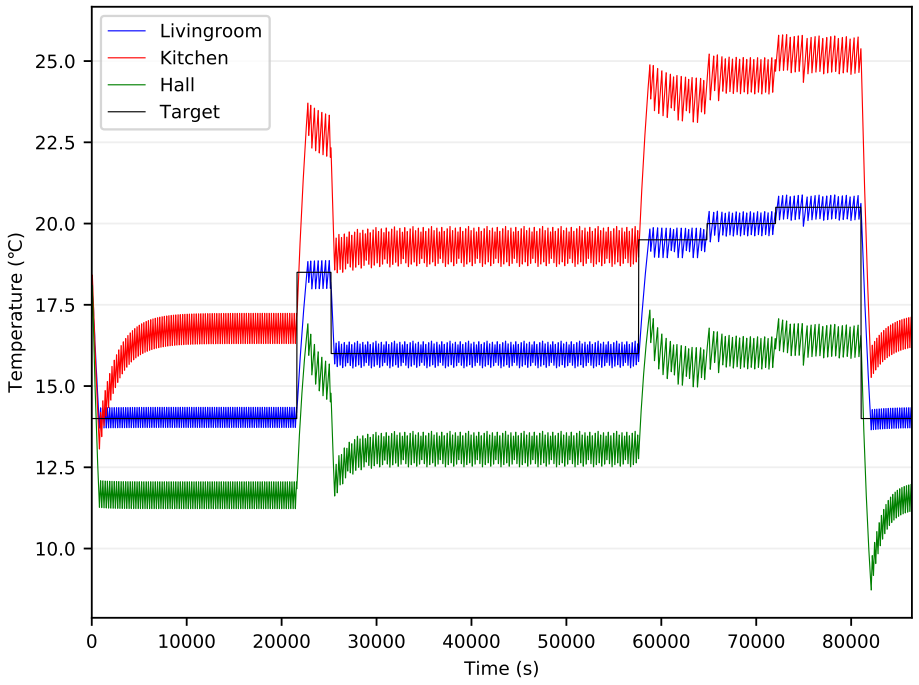


Figure 4.10: *The co-simulation results for the configuration with ID 15.*

radiators. However, note that this case serves only as an illustrative purpose for DSE and may also be solved otherwise.

4.7 Conclusion

In this chapter CoHLA has been described, a DSL that allows the rapid construction of an HLA-based co-simulation of models adhering the FMI standard and POOSL models. Each of these models is described in terms of attributes and parameters, together with information on how to run the simulation of the model. CoHLA is used to specify the co-simulation itself; simulation instances of the models should be specified as well as how the attributes of the instances are connected to each other. From these specifications, source code for a co-simulation of the models is generated.

Construction of a co-simulation using CoHLA is rather fast as it does not require the manual implementation of different wrappers for the models. The specification of a co-simulation of a small system, such as the RoomThermostat system, is done within 15 minutes, given that the models have already been created. Model parameters could be specified using CoHLA so that these can be changed rapidly as well. The language also allows the user to change the model's attributes and connections in only a matter of minutes.

By using the FMI standard, models developed in many different modelling tools are supported by the CoHLA framework. Because CoHLA is developed using Xtext and Xtend, the language itself as well as its code generator are extendable. Consequently, it is rather easy to extend CoHLA to support new modelling tools that do not support exporting their models to FMUs. For example, adding support for the collision detector took less than a day to complete.

A number of features are demonstrated on the RoomThermostat system. Starting with a base specification for this system and adding fault injection in a different experiment. An experiment with design space exploration for the RoomThermostat was also conducted. Chapters 6 and 7 describe other case studies that were conducted with CoHLA.

Reflection on requirements

1. *A co-simulation of simulation models of different disciplines can be constructed fast (20 models within 1 day).* The specification of a CoHLA co-simulation requires only basic federate class specifications that describe the interface of the model and a specification of how these simulations are connected to each other. For the RoomThermostat system, this specification can be created within an hour, which meets the requirement. Adding more models to come to a total of 20 simulations is expected to be completed within the specified time limit.

2. *Changes in either the models or the interfaces connecting these models can be adapted quickly using the approach (5 models within 1 hour).* Changes to the models could be adapted easily to the co-simulation specification. Unless the co-simulation is very large, which causes many connections to break upon changing a model's interface, changes can be applied very fast.

4. *The approach is easily extendable to support new tools.* The Xtext framework allows easy changes to the grammar, code generation, validators and scope providers of a language. As an example, adding the collision detector as library to the CoHLA framework took less than a day of work. Since the validation currently included with CoHLA is rather limited, extending the validators could be more time consuming. In the basics, however, these also are very flexible.

7. *The approach has logging capabilities for analysis afterwards.* The addition of the logger and basic metrics to CoHLA ensure that the framework meets this requirement.

8. *The approach has support for automated design space exploration.* CoHLA supports the specification of a design space in terms of model parameters. From this specification, all system configurations specified could be executed automatically. There is currently no support for automated parameter optimisation approaches by using an external tool for providing the parameter values and evaluating the co-simulation results in an automated manner. The run script could be extended to support this, but this exceeded the scope of this research.

9. *The approach has support for fault injection.* Fault injection for the communication layer is supported to mimic faulty connections between components. Model-specific faults or component degradation may be included in the models.

10. *The framework is easy to maintain and extendable.* Similar to requirement 4, the Xtext framework allows for relatively easy extension and plenty of documentation is available online to guide the maintenance of the framework.

CHAPTER 5

Trustworthiness of Co-simulation Results

This chapter discusses the relevant factors regarding the trustworthiness of the co-simulation results of a CoHLA-generated co-simulation. Even though the trustworthiness of the co-simulation depends on the quality of the simulation models used for the co-simulation, the framework that orchestrates the co-simulation could also have a negative impact on the trustworthiness. For example, when models are not properly synchronised, their behaviour may not be what is expected. Therefore, the timing and synchronisation behaviour of the co-simulation framework should be trustworthy in order to trust the co-simulation results. For this, this chapter analyses the timing behaviour of the co-simulation framework – CoHLA – and describes a number of experiments that should indicate whether the results coming from a co-simulation using this framework could be trusted.

Section 5.1 first introduces a number of definitions that are used in the following sections. The impact of the lookahead on the trustworthiness of the co-simulation results is discussed in Section 5.2. Sections 5.3 and 5.4 compare the results from a CoHLA co-simulation with a single simulator and the INTO-CPS co-simulation framework respectively. Section 5.5 describes an experiment in which the POOSL model of the RoomThermostat system is replaced by a VDM-RT model exported to an FMU. The goal of this experiment is to analyse whether the co-simulation behaviour changes when a model is replaced by a functionally equivalent model. The chapter is concluded in Section 5.6.

5.1 Definitions

Time is an important factor when looking at the confidence level in a simulation. In a CoHLA co-simulation, each federate has two notions of time.

- Logical time (LT) of a federate is the time as controlled by the RTI. This time is affected by sending TimeAdvanceRequests (TARs) to and receiving TimeAdvanceGrants (TAGs) from the RTI.

- Simulation time (ST) represents the time of the model that is being simulated by the federate. For instance, a federate that simulates an FMU controls the time of the FMU execution, which might be at a different point in time than the federate's LT.

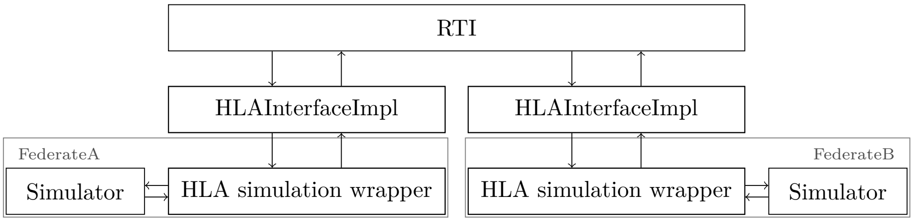


Figure 5.1: *Architecture of an HLA federation.*

Figure 5.1 shows the federation architecture in HLA. The federations consists of two federates, *FederateA* and *FederateB*. Following the aforementioned time definitions, *FederateA* has a logical time of LT_a and a simulation time ST_a, which are used in the simulation wrapper and simulator respectively. Similarly, *FederateB* has time LT_b and ST_b. Note that all these times may be different. Other important aspects regarding the timing of the federates are listed below.

- Resolution r represents the smallest possible time step that can be simulated by a simulator. Some FMUs have an internal solver that calculates the model's state using this specified time step which we shall refer to as resolution. Not all simulators have a pre-specified value for r.

- Step size s is a chosen value for stepping through simulation time that impacts simulation speed and precision. The step size is set in the simulation wrapper. When the simulator has a resolution specified, the step size should not be smaller than this resolution: $s \geq r$.

- Lookahead l has already been explained in Section 3.2.2 and is specified for each time regulating federate in the simulation wrapper. The lookahead value is chosen by the user and allows the RTI to concurrently process messages and updates between different federates. The lookahead is typically smaller than the step size of the federate: $l \leq s$.

The algorithms and examples discussed in this chapter mainly focus on fixed-step FMUs. In a co-simulation that is generated by CoHLA, these models are simulated with a predefined step size.

5.2 Lookahead

As described in Section 3.2.2, time regulating federates need to specify a lookahead value. In the naive algorithm used in the previous chapters, every attribute update sent by such federates includes the timestamp (TS). Following the rules of HLA, the TS is calculated by adding the lookahead value l to the LT of the federate.

Algorithm 5.1 Naive algorithm for connecting an FMU to HLA.

```
$t \leftarrow 0$
loop
    Send TimeAdvanceRequest for time $t + s$
    Await TimeAdvanceGrant for time $t'$                    $\triangleright\ t' = t + s$
    Proceed simulation of FMU to $t'$
    Read attributes from FMU
    Send attributes to RTI with timestamp $t' + l$
    $t \leftarrow t'$
end loop
```

Algorithm 5.1 shows the algorithm that is used to simulate an FMU in the HLA execution. A TAR for the current time t plus a step size s is sent to the RTI, after which a TAG is awaited that allows the federate to increase its logical time to t', where $t' = t + s$. The FMU is then allowed to proceed its simulation until time t'. Then, the attributes are retrieved from the FMU and sent to the RTI, having a timestamp of $t' + l$, where l is the lookahead of the federate. Finally, the federate's logical time is updated. Altogether, the attributes that were sent to the RTI now have a timestamp for $t' + l$ while the values are actually calculated for time t'. Table 5.1 shows the LT and ST of a sample federate together with the TS that is provided with the updates sent by the federate.

Cycle	0	1	2	3
Logical time (LT)	1.0	2.0	3.0	4.0
Simulation time (ST)	1.0	2.0	3.0	4.0
Timestamp (TS)	1.1	2.1	3.1	4.1

Table 5.1: *Time points in a federate ($s = 1.0$, $l = 0.1$) in the simulation wrapper, the simulator and the timestamp that is added to the updates sent to the RTI using Algorithm 5.1.*

As a consequence, the attribute values that are shared with the HLA federation are actually outdated values, since these were calculated at a time prior to their timestamp. This is particularly relevant for continuous-time model simulations, as these attribute values may also change continuously. Figure 5.2 shows the results of an experiment with the RoomThermostat system, in which the time shift caused by the lookahead is clearly visible. A particular temperature value obtains different timestamps for different lookaheads. In this case there is a difference of 0.9 seconds between the timestamps, which corresponds to the difference in their lookahead values. The figure only shows the first 30 seconds of the simulations.

To correct for this shift in timestamps, the CoHLA-generated wrapper code includes a mechanism to correct for the lookahead. This approach is implemented for FMUs only and does not work for POOSL models. Algorithm 5.2 shows the updated algorithm for FMUs.

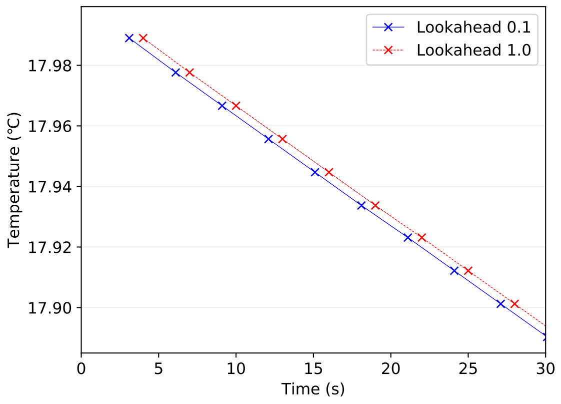


Figure 5.2: *Time shift caused by the lookahead demonstrated on the RoomThermostat system.*

For the first simulation step, the wrapper subtracts the lookahead l from the target time. From that point onward, the logical time of the federate has compensated for its lookahead, so simulation steps with the regular step size can be taken. By letting the model simulate until $t'' + l$, the timestamp that is included with the update that is sent to the RTI now also matches the time for which the values were computed by the model. We shall refer to the approach in which the simulator is allowed to calculate beyond the logical time of the federate as *forward calculation*. Table 5.2 shows the corrected logical time, simulation time and the timestamp that is provided with the attribute updates sent by the federate.

Cycle	0	1	2	3
Logical time(LT)	0.9	1.9	2.9	3.9
Simulation time (ST)	1.0	2.0	3.0	4.0
Timestamp (TS)	1.0	2.0	3.0	4.0

Table 5.2: *Time points in a federate ($s = 1.0$, $l = 0.1$) in the simulation wrapper, the simulation and the timestamp that is added to the updates sent to the RTI using Algorithm 5.2.*

From the table, it can be seen that the timestamp provided with the update now matches the time until which the simulator has executed its simulation. After conducting the same experiment on the RoomThermostat with different lookahead values, it was concluded that this approach successfully corrected for the shift in timestamps that was caused by the lookahead. The results of the experiment are shown in Figure 5.3. Both lookahead values for the co-simulation now provide a timestamp with their updates that matches the simulation time.

Forward calculation, however, brings the risk of missing updates that might be received between the federate's LT and ST. A federate with a step size s of 1.0 and a lookahead l of 0.1 calculates values for 1.0, 2.0, 3.0 etc. When using forward calculation, according to Table 5.2, the step from 0.9 to 1.9 is made in

Algorithm 5.2 Algorithm for connecting an FMU to HLA while correcting for the lookahead.

```
$t \leftarrow 0$
Send TimeAdvanceRequest for time $t + s - l$
Await TimeAdvanceGrant for $t'$                          ▷ $t' = t + s - l$
Proceed simulation of FMU to $t' + l$
Read attributes from FMU
Send attributes to RTI with timestamp $t' + l$
$t \leftarrow t'$
loop
    Send TimeAdvanceRequest for time $t + s$
    Await TimeAdvanceGrant for $t''$                      ▷ $t'' = t + s$
    Proceed simulation of FMU to $t'' + l$
    Read attributes from FMU
    Send attributes to RTI with timestamp $t'' + l$
    $t \leftarrow t''$
end loop
```

HLA and the FMU computes until time 2.0, which is also the timestamp that is provided with the update sent to the RTI. When another federate, however, sends an update with timestamp 1.95 in between, the RTI transmits this message only just before granting the federate a time advancement to logical time 2.9. This might potentially introduce an inaccuracy since the update with timestamp 1.95 might have changed the calculation of the simulator at time point 2.0. It is therefor important to consider such scenarios when determining suitable lookahead values for the federates in the co-simulation.

Initial measurement

In Figure 5.2 and Figure 5.3 it can be seen that attribute values at time 0 are missing. The cause for this is that the algorithm that controls the FMU simulation first awaits a TAG, then simulates the model, updates the attribute values and finally shares these values. When the first TAG has been received, however, its logical time has already been increased. To solve this, an initial simulation step is introduced to the federate. This initial step allows the simulation to simulate until the time that equals the lookahead value of the federate, after which the federate's attribute values at that time are sent to the RTI. Since the lookahead is usually smaller than the step size, this approach allows the federate to share its attribute values close to time 0.

Algorithm 5.3 shows the resulting algorithm to run an FMU in HLA. Before a TAR is sent to the RTI to advance its logical time, the federate first computes the attribute values of the FMU for the lowest possible timestamp that could be provided with the update sent to the RTI. The algorithm then compensates for the lookahead upon the first logical time step and then proceeds according to its step size. Note that this algorithm can only be used for fixed-step federates that allow forward computation. Other federates use the naive implementation as shown in Algorithm 5.1.

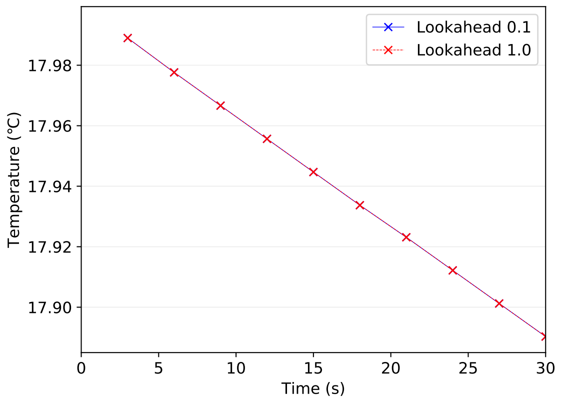


Figure 5.3: *No more time shift after correction.*

Table 5.3 shows the different notions of time using this algorithm. In contrast with the former algorithm, Algorithm 5.3 provides an initial measurement with the lowest possible timestamp – the lookahead – and synchronises the simulation time with the timestamp that is provided with the update.

Cycle	0	1	2	3
Logical time(LT)	0.0	0.9	1.9	2.9
Simulation time (ST)	0.1	1.0	2.0	3.0
Timestamp (TS)	0.1	1.0	2.0	3.0

Table 5.3: *Time points in a federate (s = 1.0, l = 0.1) in HLA, the FMU and the timestamp that is added to the updates sent to the RTI using Algorithm 5.3.*

5.3 Accurateness

To determine the accurateness of a co-simulation, the co-simulation results can be compared with results of the simulation when simulated in a single simulation tool. The RoomThermostat system with only one room – the living room – will serve as a demonstrator for this. For this, both the room model and thermostat model are created in 20-sim, as described in Section 2.2.1. The reference system simulation will be executed in 20-sim while the CoHLA co-simulation will run the separate models that are exported to FMUs by 20-sim. Consequently, the models that are being simulated are identical and should show similar behaviour in both the 20-sim simulation and the co-simulation as generated by CoHLA.

5.3.1 20-sim

The 20-sim model consists of two submodels, one submodel for the thermostat and one for the room. The simulation is executed using the Euler calculation method with a resolution of 0.1 seconds. The discrete-time submodel – the thermostat –

Algorithm 5.3 Algorithm for connecting an FMU to HLA with an initial measurement and correction for the lookahead.

```
$t \leftarrow 0$
Proceed simulation of FMU to $l$                              ▷ Initial step
Read attributes from FMU
Send attributes to RTI with timestamp $l$
Send TimeAdvanceRequest for time $t + s - l$                  ▷ First regular step
Await TimeAdvanceRequest for $t'$                             ▷ $t' = t + s - l$
Proceed simulation of FMU to $t' + l$
Read attributes from FMU
Send attributes to RTI with timestamp $t' + l$
$t \leftarrow t'$
loop
    Send TimeAdvanceRequest for time $t + s$
    Await TimeAdvanceGrant for $t''$                          ▷ $t'' = t + s$
    Proceed simulation of FMU to $t'' + l$
    Read attributes from FMU
    Send attributes to RTI with timestamp $t'' + l$
    $t \leftarrow t''$
end loop
```

is periodically updated with a step size of 30 seconds. A time span of one hour (3600 seconds) is simulated and the target temperature remains fixed at 18.0 ℃. For the resulting log file, a sampling size of 3 seconds was selected, so that the attributes are logged once every 3 seconds.

5.3.2 CoHLA

The 20-sim submodels are exported to FMUs, using the same Euler calculation method for simulating the models. The step size for the thermostat is set to 30 seconds, which corresponds with the step size that was chosen for the 20-sim simulation. To easily compare the logs from 20-sim and CoHLA with each other, the logging interval for the continuous-time model of the room should be 3 seconds as well. The step size for this model is therefor set to 3 seconds. The lookahead for both models is set to 0.1 seconds. The target temperature is specified by the parameter value using a situation specification. This situation is passed to the run script, which takes care of proper initialisation of the FMUs.

5.3.3 Results

Figure 5.4 shows the results of both the 20-sim simulation and the CoHLA co-simulation, which are very similar. Until 2520 seconds, both simulations output nearly identical values. Figure 5.5 shows the temperature difference between the simulations. The figure confirms that there is not much difference between the simulations until a simulation time of 2520 seconds. From time stamp 2520 onwards, the small difference between the simulations is just enough to let the thermostat in 20-sim turn off the heater, while the thermostat in the CoHLA simulation remains

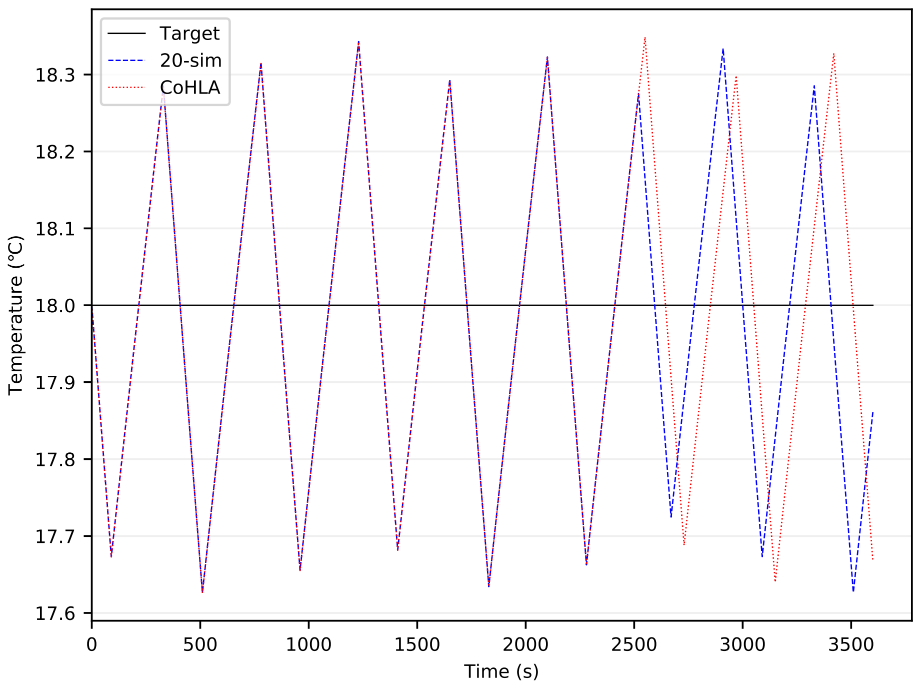


Figure 5.4: *Small RoomThermostat system simulated in 20-sim and CoHLA.*

on for another 30 seconds, causing a relatively large difference from that point onward.

The maximum difference between the two simulations that can be reached can be estimated as follows. At time t (in both simulations), the heater is on in one simulation while the heater is off in the other at time t. In both simulations, the room temperature is very close to the limits of the temperature bandwidth the thermostat has to maintain. Since both room temperatures do not exceed the limits, the thermostat does not toggle the heater state, after which the temperature continues to rise or fall for another cycle of the thermostat, which is 30 seconds in our system. During this period, the difference between the temperatures increases due to the fact that one is rising and one is falling. At time $t + 30$, the thermostat notices that the temperatures passed the limits of the bandwidth and toggles the heater state. At this point in time, the maximum difference between the two simulations is reached. The maximum difference therefore equals the temperature bandwidth around the target temperature, plus the amount of degrees the room could rise in the given interval of 30 seconds, plus the amount of degrees the room could rise in the same interval. This is confirmed by allowing the simulation to run for another 3600 seconds, which is displayed in Figure 5.6.

The temperature difference between the simulations for the first 2520 seconds is displayed in Figure 5.7. Note that the scale of the vertical axis has changed by more than a factor of 100. The figure shows that during the first 2520 seconds

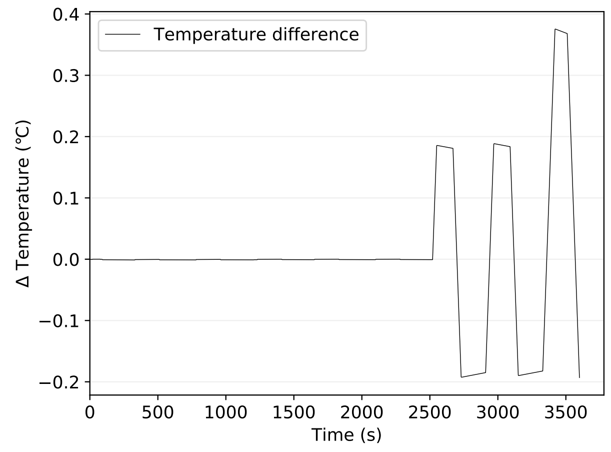


Figure 5.5: *Temperature difference between the simulations*

there are also temperatures differences between the two simulations. Since these differences appear to be too large to be caused solely by floating point inaccuracy, the source of these differences must be identified to be able to draw a conclusion on the accurateness of CoHLA.

Investigating the differences

In the comparison there are three main components, namely the 20-sim model simulation, the FMUs and the CoHLA co-simulation framework. The models that CoHLA simulates are FMUs as exported by 20-sim. These FMUs are connected to the RTI through a stack of libraries and wrappers provided by CoHLA. Figure 5.8 displays the components and layers that take part in the comparison.

To investigate what causes the measured differences, the influence of the layers between the components should be measured. For this, the following components – displayed as dotted blue boxed in Figure 5.8 – were logged.

- The **20-sim** model simulation. The resulting logs were already used in the previous comparisons with CoHLA.
- The **FMU** that was generated by 20-sim and executed by the CoHLA framework. To enable logging of the attribute values inside the FMU, its source code was modified to also generate a log file.
- The **Logger** that is generated by the CoHLA framework. The logs created by the Logger reflect the attribute values of the federates in the federation. These logs were previously used to compare the CoHLA co-simulation results with those of the 20-sim simulation.

By logging the attribute values of these three components during a simulation, these can be compared to each other, resulting in three comparisons. The comparison of the 20-sim results with the CoHLA results (from the Logger) are the

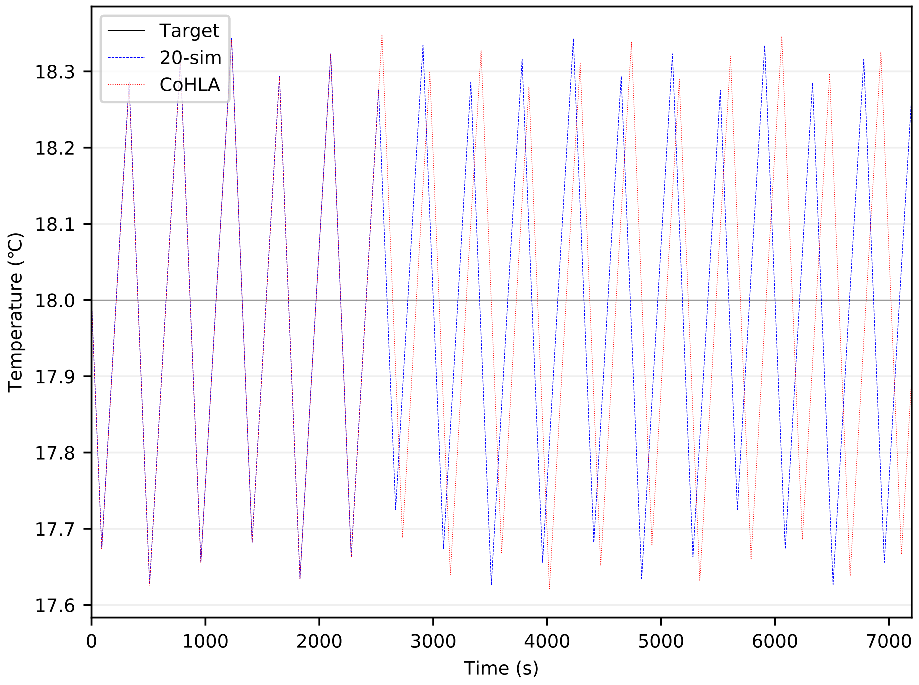


Figure 5.6: *Small RoomThermostat system simulated in 20-sim and CoHLA for a period of 7200 seconds.*

differences that we attempt to explain. These differences are displayed in Figure 5.7. The other comparisons are listed below, together with possible causes of differences.

1. **FMU vs Logger**: A series of libraries and wrappers is used to connect an FMU to the HLA RTI. From the RTI, these values are transferred to the logger, passing a similar set of layers in between. One or more of these intermediate layers could be responsible for the measured differences.

2. **20-sim vs FMU**: The exported FMU could be different from the model that is simulated by 20-sim.

Figure 5.9 shows the results of the comparisons as absolute temperature differences. Since the difference between the attribute values as read by the FMU and the HLA Logger are very small, the libraries and wrappers appear to have little impact on the measured difference. These differences are most likely caused by floating point variables being transported and converted, resulting in slightly different values.

The differences between the 20-sim model and the FMUs is the main cause for the differences between the 20-sim simulation and the CoHLA co-simulation. Since the FMUs form the base of the co-simulation, these differences can inevitably be

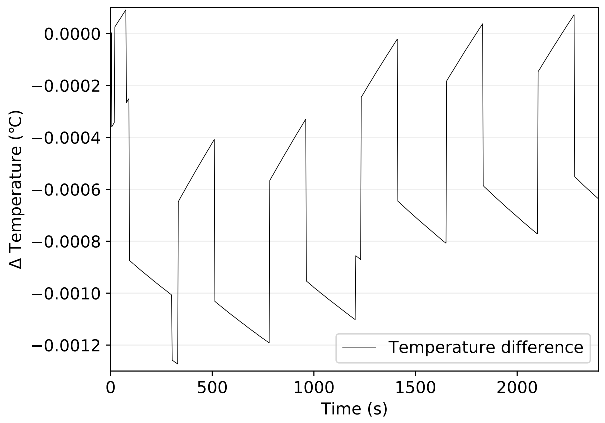


Figure 5.7: *Temperature difference between the simulations during the first 2520 seconds.*

seen in the CoHLA co-simulation as well. A number of reasons for these differences is listed below.

- 20-sim exports CSV files with a very high precision of floating-point values while the FMU uses a default print command – and rounding – to output its values.
- Differences in floating-point implementation between 20-sim and the FMU (C code).
- The exported model in the FMU is different from the 20-sim model that is being simulated.

Since the first two potential causes would only lead to very small differences between the simulations, it appears that the submodels being simulated in 20-sim differ from how these models are exported to FMUs. Hence, more logging was added to the FMUs. From these logs, it appears that the time points for which an attribute value is read from the FMU is sometimes different than expected. Table 5.4 shows the first ten time points at which the attributes are read from the FMU.

LT	0.0	0.1	3.0	6.0	9.0	12.0	15.0	18.0	21.0	24.0
ST	0.0	0.1	3.0	6.1	9.1	12.1	15.1	18.1	21.0	24.0

Table 5.4: *Time differences between logical time (LT) of the simulation wrapper and the simulation time (ST) of the FMU.*

It appears that the step size in the FMU is not that periodic as is expected by HLA and as simulated by 20-sim. Even though a step size of 3 seconds is used, the step that was actually taken in the FMU ranges from 2.9 to 3.1, except for the initial step. This is probably caused by the implementation of the *doStep*

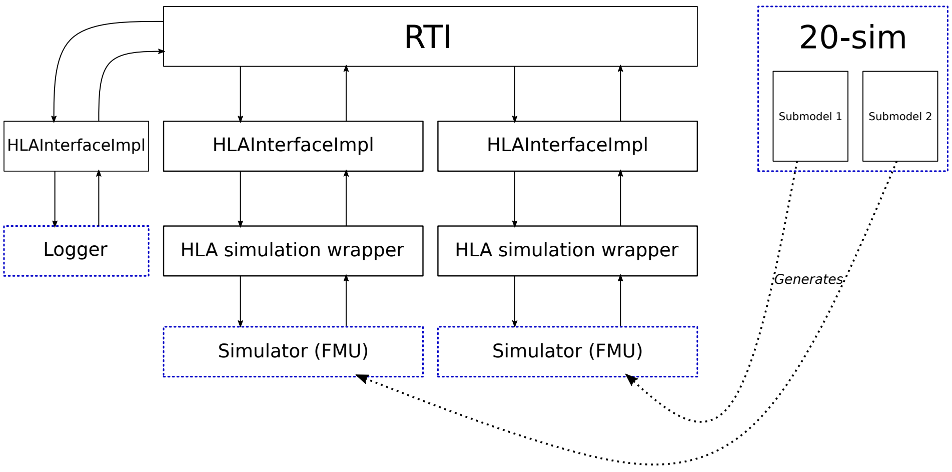


Figure 5.8: *An overview of the different components being simulated. The logged components are displayed as dotted blue boxed.*

method in the FMU that is used for stepping through the simulation. A schematic overview of this method is shown in Algorithm 5.4.

Algorithm 5.4 Schematic representation of the *doStep* method in the FMU as exported by 20-sim.

```
real time                                          ▷ The simulation time
real resolution                                    ▷ The FMU resolution
function DoStep(from_time, step_size)
    while time < from_time + step_size do
        time := time + resolution
        ComputeModel()
    end while
end function
```

Both FMUs have a resolution of 0.1 seconds. Due to the imprecise representation of floating point values, the condition for the *while* loop might sometimes be true even though it already exceeded the target time as intended by the wrapper. Consequently, the ST of the FMU is sometimes slightly different from the time to which the simulation wrapper intended to simulate (LT + l). The overshoot caused by these two components being slightly out of sync is a plausible explanation for the differences that could be measured between the simulations of the same system in both 20-sim and a CoHLA co-simulation.

5.4 Comparison with INTO-CPS

The INTO-CPS co-simulation framework was introduced in Section 1.6.4. A difference between the CoHLA framework and INTO-CPS is that the latter has developed its own engine – the Co-simulation Orchestration Engine – while CoHLA

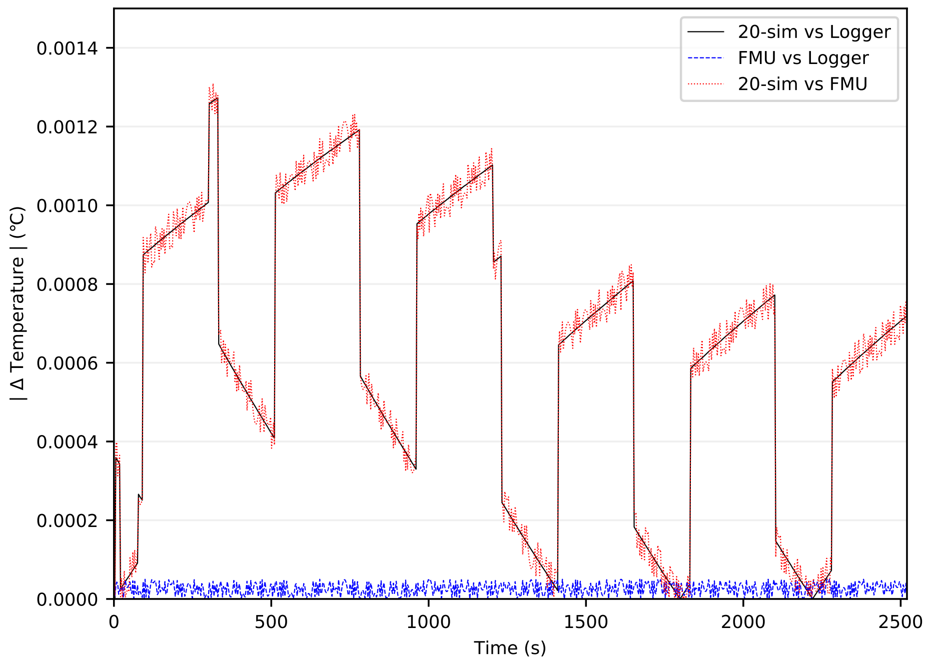


Figure 5.9: *Absolute temperature differences between 20-sim, the FMU and the logger of HLA. The differences between 20-sim and the Logger have already been displayed in Figure 5.7.*

uses an implementation of the HLA standard (OpenRTI). Consequently, there might be differences in the rules regarding time and attribute management. Both frameworks use the FMI standard for running co-simulations of models, which enables us to compare the co-simulation results. For this, a relatively easy to comprehend case study was selected from the set of open source samples provided by the INTO-CPS Association: the Single-tank Water Tank example[1], henceforth called SingleWatertank.

5.4.1 Single watertank

The SingleWatertank system consists of two simulation models. There is a water tank that fills with water at a constant rate, which increases the water level. The water tank has a valve that can be opened or closed to drain the water to avoid the tank from being too full. A valve controller controls the state of the valve to maintain the water level between specified limits.

[1]`https://github.com/INTO-CPS-Association/example-single_watertank`

Water tank

The water tank model is a continuous-time model that is created in 20-sim. There is a constant inflow of water that can be modified using a model parameter. Depending on the state – open or closed – of a valve that is located at the bottom of the tank, there is also an outflow of water. The outflow rate of the water depends on the level of water, the size of the tank and gravity as a pulling force. The size of the tank and the gravity constant can be modified using a parameter. The valve's state is an input and the water level is an output. The model is exported as an FMU for co-simulation.

Valve controller

The discrete-time model of the controller is created using VDM-RT. The controller receives the current water level of the tank as input and outputs whether the valve of the water tank should be open or closed. Parameters for the minimum level and maximum level can be specified, between which the controller should maintain the water level. This model is also exported as an FMU for co-simulation.

5.4.2 Co-simulation

Constructing a co-simulation of the two components is rather straightforward in both co-simulation frameworks. The valve state output of the controller is connected to the valve state input of the water tank and the water level output of the water tank is connected to the water level input of the controller. Initially, the water tank is empty and the controller is configured to maintain the water level between 1 and 2. Both models are simulated with a step size of 0.1 seconds from time 0 to 30 seconds. Since the CoHLA co-simulation requires a lookahead to be specified, this lookahead is 10^{-6} for both models. The full specification of the SingleWatertank system in CoHLA is displayed in Appendix F.

5.4.3 Results

Figure 5.10 shows the results as simulated by both frameworks. Although both co-simulations fail to strictly maintain the water level in the specified range, their results are very similar. Figure 5.11a shows the absolute difference in water level between the co-simulation according to their logs. This figure shows two spikes for which there is a relatively large difference in the water levels of the co-simulations. The first spike (at time 0) can be explained by the fact that the implementation of CoHLA triggers one initial computation step in the FMU, which then reports a water level that has been changed slightly since time point 0. Additional experiments show that the second spike appears to be caused by the FMU that simulates a little more than expected. This effect has already been shown in Table 5.4. The FMU as executed by CoHLA simulated until time 17.81 instead of the requested 17.8. Since the valve is open, the water level lowers at roughly 0.01 unit per second, which corresponds to the measured difference. Although the FMUs are exactly the same, the target time that is provided to the FMU by the co-simulation framework is might be slightly different because of floating point inaccuracy. Consequently, the algorithm that was already displayed in Algorithm 5.4 calculates one step further in one framework compared to the other.

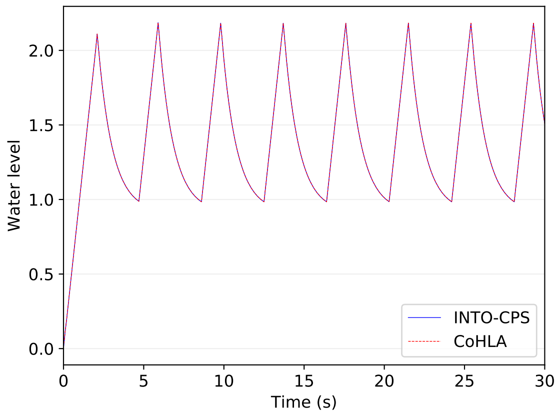
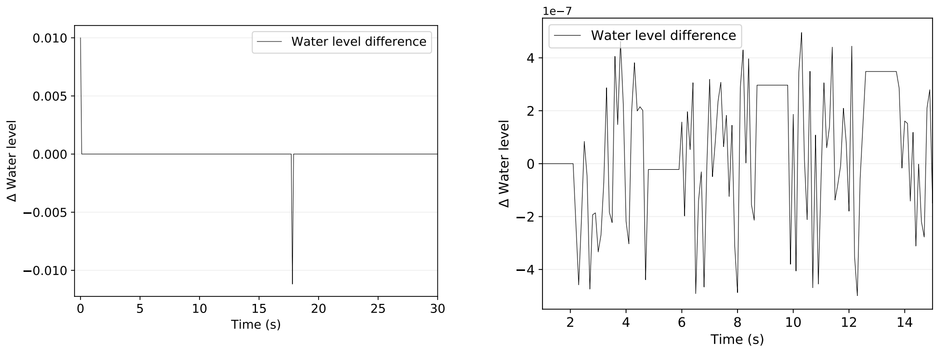


Figure 5.10: *Results of the SingleWatertank system for the INTO-CPS and CoHLA co-simulations.*

Figure 5.11b shows the water level differences between time 1 and 15. Note that the step size for the vertical axis has changed from $5 \cdot 10^{-2}$ to $2 \cdot 10^{-7}$. The figure shows that the differences vary, but remain small enough to be caused by different encodings of floating point values or rounding differences. From this experiment we conclude that both co-simulation frameworks yield nearly identical results.



(a) *Full simulation of 30 seconds.*

(b) *Simulation from 1 to 15 seconds.*

Figure 5.11: *Water level differences of the Watertank between the INTO-CPS and CoHLA co-simulations.*

5.5 Model replacement

CoHLA allows the user to replace one component model with another model of the same component without much effort. For instance, a component model created in OpenModelica could be replaced easily by a model that is created in 20-sim, especially since these modelling tools both support exporting the model to an FMU. When the models have identical interfaces, i.e. identical input and output attributes, this involves very little changes to the CoHLA specification, as will be shown below. More changes, or even the specification of a new federate class, are required when the interfaces are different. The control loop of the thermostat in the RoomThermostat system can be expressed in a POOSL model or in a VDM-RT model, as described in Sections 2.2.2 and 2.2.3 respectively.

To analyse the effort required and the impact on the co-simulation results of replacing one model with another model with the same interface, three different co-simulations were executed. These co-simulations are all co-simulation instances of the RoomThermostat system. The only difference between the co-simulations is the thermostat model that is being simulated. Three different models of the same thermostat are used for the co-simulations. All models implement the same control algorithm, which is displayed in Algorithm 2.1. One model is created in POOSL and is used for the co-simulations described in Sections 3.4 and 4.6. The other models are created in VDM-RT and 20-sim and exported to FMUs. These models are described in Sections 2.2.1 and 2.2.3 respectively.

For this experiment, only the federate class specification of the thermostat must be changed. The thermostat class as displayed in Listing 4.3 specifies the POOSL thermostat in CoHLA. Listing 5.1 displays the CoHLA specification of the thermostat federate class as an FMU type of simulation model.

```
 1  FederateClass Thermostat {
 2    Type FMU
 3    Attributes {
 4      Output Boolean HeaterState
 5      InOutput Real TargetTemperature
 6      Input Real Temperature
 7    }
 8    Parameters {
 9      Real TargetTemperature "TargetTemperature"
10    }
11    TimePolicy RegulatedAndConstrained
12    DefaultModel "../../models/Thermostat_20sim.fmu"
13    AdvanceType TimeAdvanceRequest
14    DefaultStepSize 30.0
15    DefaultLookahead 0.1
16  }
```

Listing 5.1: *FMU-type thermostat federate class specification for CoHLA.*

The listings show that changing the type of the simulation model using the same interface only requires changing the type (as shown on line 2 of Listing 5.1) and – in this case – removing the *process* that is required for POOSL models to map the attributes on. Since POOSL models cannot have a specified default step size, this is also added to the FMU-type specification in Listing 5.1 on line 14. Because the attribute names have not changed, no further changes to the co-simulation specification are required and new code is generated to wrap the FMU automatically. As can be seen on line 12 of Listing 5.1, the model *Thermostat_20sim.fmu* is simulated by default. Since the VDM-RT FMU model uses the exact same input and output attribute names, this model to be simulated can be changed by adding it as a parameter to the run script upon starting the co-simulation. Alternatively, the default model can be changed in the specification.

The co-simulation that is run is identical to the one described in Section 4.6.1. The rooms all have different characteristics and one hour (3600 seconds) is being simulated. Figure 5.12 displays the simulation results. To improve readability, only the temperature of the living room is displayed. Since the results for 20-sim and VDM are nearly identical, these lines overlap. Consequently, only the plot for 20-sim is visible in the chart.

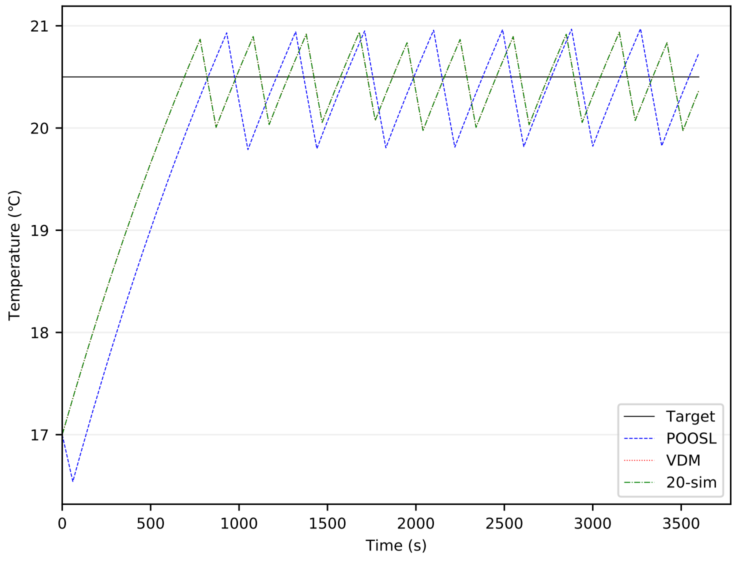


Figure 5.12: *Living room temperatures when using three different simulation models for the co-simulation. Due to the overlapping results of VDM and 20-sim, only the latter plot is visible.*

The results show that the simulated behaviour of the FMUs is identical. This seems reasonable, since both the wrapper and the modelled behaviour are identical. However, the co-simulation results with the POOSL model of the thermostat are different. Two causes for these differences have been explained in previous sections. The lack if an initial measurement in the POOSL simulation, as described for FMUs in Section 5.2, causes a delay after starting the simulation. Also, the mechanism used to control the POOSL simulation through the debugging socket causes the POOSL model to respond to changes one cycle late, as was described in Section 3.3.1. These causes can be read in the logging information displayed in Table 5.5.

From the table, it can be seen that the thermostat does not perform an initial measurement, thus the heater state remains in its default state. After the first cycle, at time 30, the thermostat has used an old value – thus default value in this case – to determine its heater state value. This is caused by the introduced delay,

Simulation time (s)	0.1	*30.0*	30.1	*60.0*	60.1
Room.Temperature (°C)	17.00	16.77	16.77	16.54	16.54
Thermostat.Temperature (°C)			16.77	16.77	16.54
Thermostat.HeaterState			Off	Off	On

Table 5.5: *The temperature of the living room according to the living room simulation (FMU) and the thermostat simulation (POOSL) and the heater state attribute of the thermostat at specific time points during the simulation. Time points marked italic represent the time points at which the thermostat calculates its heater state value by comparing its received temperature attribute with the target temperature. The updated attributes are published with a lookahead value of 0.1. The thermostat computes its state using the temperature value of the living room from the previous cycle. The first computation after the start of the simulation uses the defaults values.*

as explained in Section 3.3.1. Consequently, the heater state is only updated to a state that one should expect at time 60.1.

Because of the delay introduced in the POOSL simulation, these types of models are not suitable for simulating low-level control algorithms in a co-simulation. POOSL models may still be used for modelling high-level control of different components and to provide interaction with applications outside of the co-simulation. By implementing another mechanism for simulating POOSL models in a co-simulation that does not use the debugging socket, the delay could be removed. The addition of a request to ask the Rotalumis simulator for the next desired time step would also eliminate the delay. In the experiments, exchanging one FMU for another FMU, where both FMUs model identical components, did not show any issues.

5.6 Conclusion

In this chapter the impact of the lookahead value on the co-simulation results was analysed and a method to compensate for the time shift that was found is implemented for FMUs in a CoHLA co-simulation. The co-simulation results of the CoHLA co-simulation of the RoomThermostat system have been compared to the results when the models are all simulated by the simulator of the 20-sim modelling tool that was used to create the models. The CoHLA co-simulation results of a Watertank system have been compared to the same co-simulation executed by the INTO-CPS framework.

The HLA standard uses a lookahead value to improve the simulation speed of the co-simulation while still guaranteeing correct timing behaviour. This lookahead value caused a time shift between the time for which an attribute value was calculated and the timestamp that was included when sharing the value with other federates. The simulation algorithm in CoHLA was updated to compensate for the lookahead, so that the aforementioned time points match.

By comparing the results of the CoHLA co-simulation with those of the 20-sim model simulation we found that the simulations are very similar. However, there are small differences. More detailed comparisons show that the differences are mainly caused by the difference between 20-sim and the FMUs exported by 20-

sim. The FMUs are simulated for slightly difference time points than intended due to floating point inaccuracies, resulting in different attribute values. The results show that the CoHLA wrappers and libraries do not influence the co-simulation results other than introducing a very small floating point inaccuracy. Even though this difference is very small, it shows that the co-simulation results only provide an indication of the system's behaviour as this could be slightly different in the real system.

A comparison of the results from the CoHLA co-simulation with those of the INTO-CPS framework shows that these are very close to each other for the Single-Watertank case. The results from the two cases show that the results from co-simulations generated by CoHLA are trustworthy enough to use and to continue the development of CoHLA.

Reflection on requirements

6. *The simulation results are trustworthy.* Although the experiments in this chapter do not provide any proof of the results from a CoHLA co-simulation being correct, they show that the co-simulation framework appears to synchronise both time and attributes of FMUs in a correct fashion. Because of the implementation of the library that connects the Rotalumis simulator to the RTI, POOSL models introduce a delay in their timing.

CHAPTER 6

System Design using CoHLA

This chapter describes the application of the system development as illustrated in Section 4.1 in an industrial context. Since this research was conducted in partnership with an industrial company, a case study to one of their systems was conducted. Due to confidentiality of this case, a smaller system – the SliderSetup – was designed to reflect the challenges that were encountered for the industrial system. The use of CoHLA for the design of the SliderSetup is explained in this chapter.

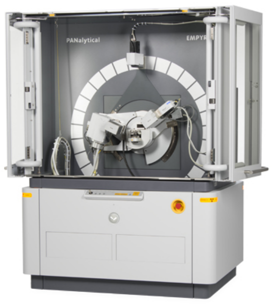
Section 6.1 first introduces the industrial system, after which the SliderSetup is described in Section 6.2. Section 6.3 explains the components and the models of the SliderSetup. The process of designing the system and the use of CoHLA for this is highlighted in Section 6.4. An application of design space exploration on the co-simulation of the system is explained in Section 6.5. The realisation of the system is briefly described in Section 6.6. Section 6.7 introduces a separate DSL that was developed to simplify the process of supporting the same protocol in the models as in the implemented system. Section 6.8 concludes the chapter.

6.1 Industrial context

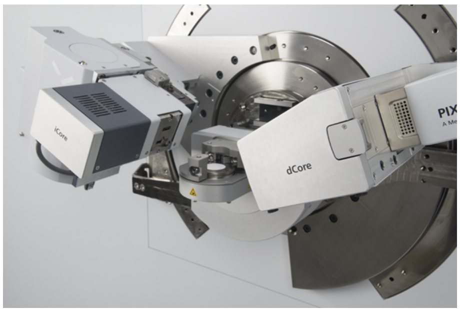
The research presented in this dissertation was conducted in partnership with Malvern Panalytical[1]. Malvern Panalytical designs and produces systems for the chemical, physical and structural analysis of materials. These systems are used by industry and academia all over the world for, for example, quality assurance and research. During the development of CoHLA, a system of Malvern Panalytical served as a larger case study. Subject of the case study was an X-ray diffracto-

[1]https://www.malvernpanalytical.com/

meter[2] (XRD) that was already designed prior to the presented research. Figure 6.1 displays an example Malvern Panalytical XRD system and a more detailed picture of a number of components.



(a) *The whole system.*



(b) *A detailed image of the components.*

Figure 6.1: *The Malvern Panalytical Empyrean X-ray diffractometer. Images were downloaded from https://www.malvernpanalytical.com/en/products/product-range/empyrean-range/empyrean on September 3rd 2019*

An XRD system uses X-rays to determine a number of characteristics of a sample material. This is done by directing an X-ray beam towards the material and analysing the scatter pattern of the beam as reflected by the material. For this, both a detector and the X-ray source change their position to change the angle at which the X-ray beam hits the material's surface. These movements are achieved by a goniometer and is controlled by software. Since most of these components are highly complex and expensive, a virtual prototype by means of a co-simulation of the component models may support the design process. The virtual prototype can be used to analyse the system's behaviour and performance characteristics before building a real-life prototype.

A co-simulation of this system was developed from a number of models of its components. Only a subset of component models was included in the co-simulation, since these components were sufficient to answer a number of research questions. A number of CoHLA features, such as the *SocketEvent* (Section 4.4.7) and collision detection (Section 4.4.11), were developed as proof of concepts for this particular case. The case also served as an example co-simulation to test other features on.

Due to the confidentiality of the industrial case, an experimental system was developed in collaboration with the University of Twente[3]. This case is called the slider setup (referred to as SliderSetup) and reflects most of the challenges that were encountered in the industrial case. This system is introduced in Section 6.2.

SliderSetup versus industrial case

For both the SliderSetup and the industrial system, the models of the supervisory controller were created using POOSL. This model for the industrial case contains

[2]https://www.malvernpanalytical.com/en/products/category/x-ray-diffractometers

[3]https://www.utwente.nl/

4046 lines of code, while the supervisory control model of the SliderSetup only consists of 679 lines. Both numbers exclude the library sources needed for the projects, as these largely overlap. Similar to the SliderSetup system, the physical components were modelled in 20-sim. To visualise the system as well as to analyse possible collisions of components, 3D drawings have been used for both systems. Multiple features for the CoHLA framework have been developed in accordance with the needs for the industrial system design.

For both systems, we experimented with multiple models for the same component. To change the model for one component with a model created in a different modelling tool, the CoHLA specification barely needed any change. The experiments have shown that swapping out a POOSL model for an FMU (or vice versa) requires only several minutes to change the CoHLA specification. Changes in the attribute or parameter names might take a little more time, but it is still rather fast. This allows modellers to change their modelling tools without causing a large amount of work for changing the co-simulation.

6.2 The SliderSetup system

The SliderSetup consists of two independent, intersecting axes. Every axis has a slider that can be moved along the axis. One axis is located at the bottom of the system while the other is located at the top. Figure 6.2a schematically displays both axes and their sliders as seen from above. The arrows in the figure indicate the possible movements by the sliders. A detailed view of the realised sliders is shown in Figure 6.2b, where the sliders are encircled in blue.

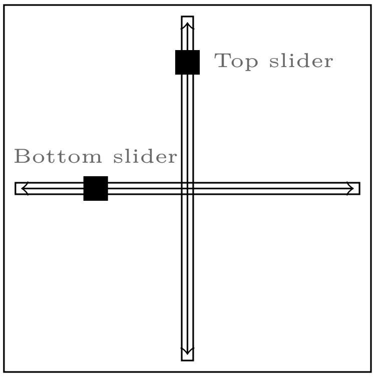
The goal of the SliderSetup is to unwind a thread coil from one slider to the other by letting the sliders orbit around each other. To achieve this, the top slider's coil is directed downwards while the bottom slider's coil is directed upwards. It is important that the mounted thread coils do not collide during the (un)winding routine. The system should be able to let the two sliders orbit each other with a minimum speed of two rotations per second to ensure a minimum (un)winding speed. Following step 1 of the design flow as described in Section 4.1, the requirements are listed below.

1. The system is capable of (un)winding the thread coil at a minimum speed of 2 rotations per second.

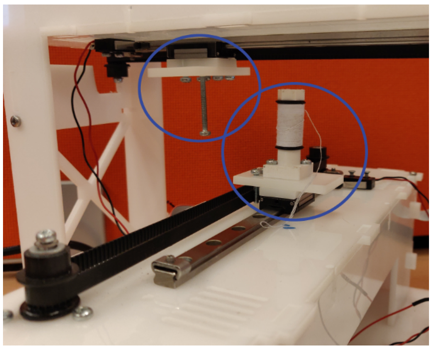
2. The components of the system may not collide.

6.3 Models

The SliderSetup consists of 6 components, which are displayed in the component architecture in Figure 6.3. Each slider is moved along its axis by a motor. A motion controller is responsible for the control of each motor. Embedded control software (ECS) fulfills the role of supervisory controller that controls the two controllers to coordinate their movements on a higher level. These software components all run on the embedded part of the system. To let a user control the SliderSetup system,



(a) *Top-level view of the two axes and their sliders of the SliderSetup.*



(b) *Side-view of the two sliders (encircled in blue) of the realised SliderSetup.*

Figure 6.2: *Two visual representations of the SliderSetup.*

management software that runs on a PC was developed. Except for the interface specification towards the supervisory controller, the management software is not considered to be a part of the SliderSetup system. Note that the identification of the components and the specification of their interfaces corresponds to steps 2 and 3 of the design flow that is described in Section 4.1. This section describes the three embedded components and their models used for the development of the SliderSetup.

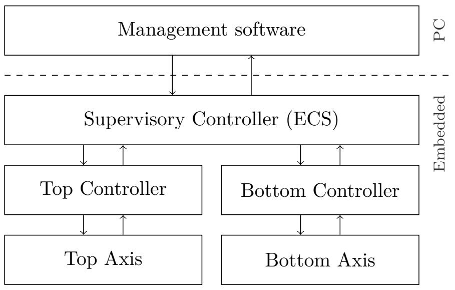


Figure 6.3: *Overview of the component architecture of the SliderSetup.*

6.3.1 Sliders

The slider models include all physical aspects of the SliderSetup such as the motor and the (position of the) slider. Based on the speed requirement (requirement 1), a belt was chosen to actuate the slider. The belt is driven by an electromotor. This motor is selected based on two factors: its torque and its ease to control these by using available motor drivers. The motor requires a certain amount of torque to meet the velocity requirement. The combination of the motor and driver should be driven by providing voltage to the motor. It should also be possible to enable or disable the motor. A method for determining the position of the slider on the axis is to be determined in a later stage during the design.

Axis			
In/Out	**Name**	**Type**	**Description**
Input	Voltage	`boolean`	The input voltage for the motor.
Input	Enable	`boolean`	The enabled state of the motor: enabled is represented by `true`, disabled by `false`.
Output	Encoder	`real`	The encoder position in rotations.
Output	Position	`real`	The real position of the slider.

Table 6.1: *Input and output attributes of the axis model.*

Table 6.1 shows the required input and output attributes of the axis model. It is a continuous-time model that is developed in 20-sim. The model describes the motor behaviour, the driving electronics, the transmission to linear motion and the necessary sensors for a single axis. The axis model represents the physical design of the SliderSetup.

6.3.2 Controllers

The controllers are responsible for the motion control of the axes. Each controller controls one axis and is implemented as a piece of embedded software in the system. The software runs at a high frequency and calculates the input voltage for the connected axis to properly move the slider along the axis. Since our intention is also to compare different control modes with each other, the controller should support the following control modes.

- A controlled move using a motion profile.
- A fast control mode using linear control.
- A constant velocity mode.

Each of the three modes requires different input attributes to the controller. These are converged into one set of attributes. The input and output attributes are displayed in Table 6.2. The controller model is a discrete-time model developed in 20-sim.

Controller			
In/Out	**Name**	**Type**	**Description**
Input	Mode	`integer`	The current control mode, either 0 (controlled, motion profile), 1 (fast linear) or 2 (constant velocity).
Input	Setpoint	`real`	The target location (mode 0 or 1) or the velocity (mode 2).

Controller			
In/Out	**Name**	**Type**	**Description**
Input	StrokeTime	`real`	The stroke time for the controlled movement (only for mode 0).
Input	Encoder	`real`	The encoder position of the axis.
Output	Voltage	`real`	The input voltage for the motor.
Output	EncoderPosition	`real`	The slider position calculated from the encoder rotations.

Table 6.2: *Input and output attributes of the controller model.*

6.3.3 Supervisory controller

The supervisory controller coordinates the movements of both sliders via their controllers. It is responsible for the overall behaviour of the system. It provides an interface to allow communication with desktop software or other systems. Sequences of actions for one or both sliders are stored and executed by the supervisory controller. Table 6.3 displays the input and output attributes for the model of the supervisory controller.

Supervisory controller			
In/Out	**Name**	**Type**	**Description**
Input	TopPosition	`real`	The current position of the top slider.
Input	BottomPosition	`real`	The current position of the bottom slider.
Output	TopEnable	`boolean`	Enabled state for the top axis, `true` when enabled, `false` when disabled.
Output	TopMode	`integer`	The control mode for the top axis.
Output	TopSetpoint	`real`	The setpoint or velocity for the top axis, depending on the mode.
Output	TopStrokeTime	`real`	The stroke time for a controlled move for the top axis, depending on the mode.
Output	BottomEnable	`boolean`	Enabled state for the bottom axis, `true` when enabled, `false` when disabled.
Output	BottomMode	`integer`	The control mode for the bottom axis.
Output	BottomSetpoint	`real`	The setpoint or velocity for the bottom axis, depending on the mode.

Supervisory controller			
In/Out	**Name**	**Type**	**Description**
Output	BottomStrokeTime	`real`	The stroke time for a controlled move for the bottom axis, depending on the mode.

Table 6.3: *Input and output attributes of the supervisory controller model.*

The model of the supervisory controller is a discrete-time model created using POOSL. The ability to incorporate sockets into the model for the development of the external interface lead to the choice to model this component in POOSL.

6.4 Design

The first versions of each of the component models might not be suitable for simulation purposes. When the models are sufficiently developed to be simulated and support all inputs and outputs specified in the previous section, a co-simulation can be constructed from these models, which is in line with step 4 of the design flow (Figure 4.1). This co-simulation forms the base of the development. From this point onward, the models can be iteratively improved, after which a co-simulation can be run again, as described as steps 5 to 7 in Section 4.1. This section introduces the co-simulation that was developed for the SliderSetup and the refinement steps during multiple iterations.

6.4.1 Co-simulation

In the co-simulation of the system, both the axis model and controller model are included twice: one for each axis and slider. Each of these 20-sim models is exported to an FMU that can be included in CoHLA. The POOSL model of the supervisory controller can be simulated by the CoHLA co-simulation and does not need to be exported. For each of the three models, a federate class specification in CoHLA is created. Refer to the CoHLA user manual or Section 4.4 for more information on the specifications.

```
FederateClass Axis {
  Type FMU
  Attributes {
    Input Boolean enable
    Input Real motor
    Output Real encoder
    Output Real position
  }
  DefaultModel "models/SliderAxis.fmu"
  DefaultStepSize 0.0005
  DefaultLookahead 0.0001
}
```

Listing 6.1: *Specification of the federate class for the axis model.*

```
FederateClass Controller {
  Type FMU
  Attributes {
    Input Real encoder
    Input Integer mode
    Input Real setpoint
    Input Real stroketime
    Output Real voltage
    Output Real encoder_position
  }
  DefaultModel "models/SliderController.fmu"
  DefaultStepSize 0.005
  DefaultLookahead 0.001
}
```

Listing 6.2: *Specification of the federate class for the controller model.*

Listing 6.1 and Listing 6.2 show the federate class specifications for the axis model and controller model respectively. The available parameters for each of the federate classes are hidden for the sake of readability. The federate class for the supervisory controller is displayed in Listing 6.3.

```
FederateClass SupervisoryController {
  Type POOSL {
    Processes {
      supController in "SupervisoryController"
    }
  }
  Attributes {
    Input Real topPosition in supController as "topPosition"
    Output Integer topMode in supController as "topMode"
    Output Real topSetpoint in supController as "topSetpoint"
    Output Real topStrokeTime in supController as "topStrokeTime"
    Output Boolean topEnable in supController as "topEnable"

    Input Real bottomPosition in supController as "bottomPosition"
    Output Integer bottomMode in supController as "bottomMode"
    Output Real bottomSetpoint in supController as "bottomSetpoint"
    Output Real bottomStrokeTime in supController as "bottomStrokeTime"
    Output Boolean bottomEnable in supController as "bottomEnable"
  }
  DefaultModel "models/sliders.poosl"
  DefaultLookahead 0.001
}
```

Listing 6.3: *Specification of the federate class for the supervisory controller model.*

The control loop in the controller model is executed at a frequency of 200 Hz. To ensure that the continuous-time axis model is sampled at a higher frequency, the step size for this model is a factor of 10 higher: 2 kHz. The lookahead values for the federates are kept small to minimise the delay of messages being transferred from one federate to another. Listing 6.4 shows the instances and connections as specified in the federation specification to connect the simulation models.

```
  Instances {
    bottomAxis        : Axis
    topAxis           : Axis
    bottomController  : Controller
    topController     : Controller
    supController     : SupervisoryController
    collisionDetector : CollisionDetector

  Connections {
    { bottomController.encoder      <- bottomAxis.encoder }
    { bottomAxis.motor              <- bottomController.voltage }
    { supController.bottomPosition  <- bottomController.encoder_position }
    { bottomAxis.enable             <- supController.bottomEnable }
```

```
14      { bottomController.mode         <- supController.bottomMode }
15      { bottomController.setpoint     <- supController.bottomSetpoint }
16      { bottomController.stroketime   <- supController.bottomStrokeTime }
17
18      { topController.encoder         <- topAxis.encoder }
19      { topAxis.motor                 <- topController.voltage }
20      { supController.topPosition     <- topController.encoder_position }
21      { topAxis.enable                <- supController.topEnable }
22      { topController.mode            <- supController.topMode }
23      { topController.setpoint        <- supController.topSetpoint }
24      { topController.stroketime      <- supController.topStrokeTime }
```

Listing 6.4: *Federation specification of the SliderSetup.*

Unlike what was displayed in Figure 6.3, the supervisory controller is connected directly to each of the axes to update the *enable* attribute on lines 13 and 21. The reason for this is that the abstract version of the controller model does not yet allow this attribute to pass through the model. The model should be updated to support this. In order to construct a co-simulation with the current abstract models, the attribute is passed directly from the supervisory controller model to the axis models.

From this federation specification, co-simulation code and its configuration is generated to run a co-simulation of the abstract models. A logger was added to the federation to retrieve useful information from the co-simulation execution. The co-simulation results form a starting point for further improvement of the models.

6.4.2 Refinement

The co-simulation of the system allows analysis whether the system design would meet the requirements. For the SliderSetup the orbiting speed of the sliders could be checked using the co-simulation and the collision detector can be used to check for possible collisions of the sliders. The co-simulation also allows for system-level simulation and impact analysis of design decisions, providing guidance during the design process. Such a system-level analysis allows for the development of features that affect the design of different components of the system. Since the initial models are rather abstract, these should be refined first to support the most basic functionality of each component. More functionality can later on be added iteratively.

System-level analysis

During an early co-simulation execution with abstract models, unexpected behaviour of the system was observed when switching the control mode of the controller. Simulation showed that the mode change was not processed correctly when the controller changed the mode and setpoint simultaneously or when the setpoint was changed before the mode. The observed behaviour was that the slider moved very slowly or not at all after this error occurred. Due to the early detection of this error, the supervisory controller was changed to always update the mode before the setpoint. If this error was only detected later in the design process, we expect it to take more effort to find the cause and change the control structure to guarantee correct timing.

Feature co-design

Adding detail or new features to the system design may require changes in one or more components. Following the co-modelling approach as posed in [54], these changes are adapted to the models, after which these models can be co-simulated to analyse the new feature or the added details. This approach allows for a shorter feedback loop on the consequences of design changes.

During the design of the SliderSetup, a new feature was developed following this approach. The initial design is unable to determine the absolute slider position on its axis. This is caused by the motor sensor, which is an absolute rotation sensor. This sensor can only be used to determine the displacement from the starting position of the slider. To calculate the absolute slider position after one or more movements, the starting position of the slider should be known.

To detect the absolute starting position, a limit switch was added to each axis. When the carriage hits the limit switch, a signal is sent to the controller, which then knows the absolute position of the slider. From that point onward, the rotational motor sensor can be used to calculate the displacement of the slider, thus calculating its absolute position on its axis. The limit switches are located at the far end of each axis. The axes are 30 cm long and the center – where the axes intersect – is defined as position 0 cm The limit switches are located on position -15 cm of each axis. Figure 6.4 shows the positions of the limit switches.

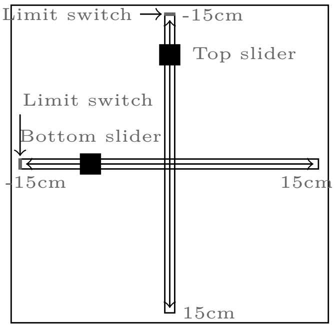


Figure 6.4: *Top-level view of the two axes, their sliders and the positions of the limit switches.*

To include these limit switches in the design, all models need to be changed. The switch itself is added to the axis model, as well as the required behaviour of the motor rotation sensor that is included in this model. Since the initial abstract axis model outputted the absolute position, this needs to be changed to rotations. The controller model is extended with calculations to handle these new sensor values and to handle the limit switch being triggered. To transmit the limit switch value, a new connection between the axis model and the controller model was specified. Finally, the supervisory controller needs to implement initialisation behaviour that moves both sliders towards their limit switches to be able to tell their positions. Section 6.5 describes the selection of an initialisation procedure in more detail.

The co-simulation was executed to test the initialisation procedure and the effect of the limit switches on the system. The results showed a flaw in the design of the controllers, as the correction of the position due to the limit switch was interpreted by the controller model as a large change in position, and thus in velocity. The controller attempted to correct this change, resulting in undesired behaviour of the system. While the erroneous movement that was caused by the correction was not a problem in the simulation, it could have been damaging when it would have happened on the real hardware. Co-simulation allowed us to detect this flaw early, after which it was fixed.

6.5 Design space exploration

Upon starting the SliderSetup, the positions of both sliders are unknown. A mechanism that could be used to determine the positions of the sliders was described in the previous section. Both sliders move slowly in the direction of the limit switch and stop when it is pressed. When the slider presses the limit switch, its position is known. After the limit switch, there is only 2 mm of space before the hard limit of the rail is hit. Hitting this hard limit should be avoided, as this may introduce errors in the calibration of the system. A possibility to avoid this, is to move really slow towards the limit switch so that there is enough time to prevent the slider from moving any further. However, this results in a rather slow initialisation procedure. To find a proper trade-off between initialisation speed and system safety, DSE is used to come to a suitable initialisation procedure. The goal is to find an initialisation procedure that quickly initialises the system and does not collide with the hard limit of the rail.

6.5.1 Design space

Section 6.3.2 introduced three different control modes that are supported by the controllers. The *FastMode* moves to a specified position as fast as possible while *StrokeMode* performs a controlled move based on a target position and movement duration (stroke time). Finally, the *FixedMode* specifies a movement speed instead of a target position.

Mode	**Attribute**		**Values**
StrokeMode	Starting position	(m)	0.15, 0.05, −0.05, −0.14
	Stroke time	(s)	1.0, 1.5, 2.0, 2.5, 3.0
FastMode	Starting position	(m)	0.15, 0.05, −0.05, −0.14
FixedMode	Starting position	(m)	0.15, 0.05, −0.05, −0.14
	Movement speed	(m/s)	0.02, 0.04, 0.06, 0.08, 0.10, 0.12, 0.14, 0.16, 0.18, 0.20

Table 6.4: *Design space for the initialisation procedure of the SliderSetup.*

For the initialisation method we can use these three modes. We use DSE to compare the modes with each other. For all modes, we iterate over a small set of initial slider positions, namely 15, 5, -5 and -14 cm. These positions were chosen as they divide the length of the slider by four, where the initial position of -14 cm is just one centimeter away from the position of the limit switch. The FastMode does not require any parameters to be set, while the StrokeMode will be used with different stroke times to modify the movement speed. The FixedMode will also iterate over different movement speeds. The resulting design space is displayed in Table 6.4.

Since both axes are identical, only the bottom slider is considered for DSE. Independent sweep mode is used for DSE to simulate all different starting and initialisation configurations. Since the three control modes require different parameters to be set, the DSE is split up into three separate DSE configurations, one for each control mode, having the appropriate parameters specified. Consequently, there are three DSE configurations, having 20 (StrokeMode), 4 (FastMode) and 40 (FixedMode) initialisation configurations for the SliderSetup. Listing 6.5 shows the DSE configurations for the system in CoHLA.

```
DSE strokeStartPositions {
  SweepMode Independent
  Set supController.initMode : "0"
  Set bottomAxis.Position_realInitial :  "0.15", "0.05", "-0.05", "-0.14"
  Set supController.initSpeed : "1.0", "1.5", "2.0", "2.5", "3.0"
}
DSE fastStartPositions {
  SweepMode Independent
  Set supController.initMode : "1"
  Set bottomAxis.Position_realInitial :  "0.15", "0.05", "-0.05", "-0.14"
}
DSE speedStartPositions {
  SweepMode Independent
  Set supController.initMode : "2"
  Set bottomAxis.Position_realInitial : "0.15", "0.05", "-0.05", "-0.14"
  Set supController.initSpeed : "0.02", "0.04", "0.06", "0.08", "0.1", "0.12"
      , "0.14", "0.16", "0.18", "0.2"
}
```

Listing 6.5: *DSE configurations for the initialisation procedure of the SliderSetup.*

All initialisation procedures consist of the slider moving towards the limit switch and moving back to position -10 cm when the limit switch is triggered. Once this position is reached, the initialisation is finished. Moving towards the limit switch is achieved by setting the setpoint to -30 cm for the FastMode and StrokeMode, because the limit switch must be found within this distance. FixedMode moves with a negative speed towards the limit switch.

6.5.2 Metrics

The goal is to balance out the initialisation speed of the SliderSetup with its accuracy. Here, accuracy can be measured as the overshoot from the limit switch upon impact. The overshoot must be less than 2 mm in order to avoid damage to the system. The speed of the initialisation procedure is the first moment during the simulation where the supervisory controller has finished the initialisation procedure. An attribute was added to the supervisory controller that is set to `true` upon finishing this procedure.

These properties can be automatically extracted from the simulation by using CoHLA performance metrics. The first moment for which the supervisory controller reports to be initialised can be tracked by using a timer metric. The overshoot basically is the minimum position that is reached by the slider during the simulation. This can be tracked by using a *minimum* metric. The resulting MetricSet for the DSE is displayed in Listing 6.6. A measure time of 300 seconds is specified, as it is expected that the initialisation finishes within this duration. When the initialisation procedure finishes, the co-simulation is stopped, as the *InitialisationTime* metric is set to be an end condition.

```
MetricSet Initialisation {
  MeasureTime: 300.0
  Metric InitialisationTime as Timer for supController.initialised == true (
      EndCondition)
```

```
    Metric MinBottomPosition as Minimum of bottomAxis.position
}
```

Listing 6.6: *MetricSet for finding a suitable initialisation procedure for the SliderSetup.*

6.5.3 Results

All initialisation procedure configurations in the design space were automatically simulated using the run script. Figure 6.5 shows the results. The configurations

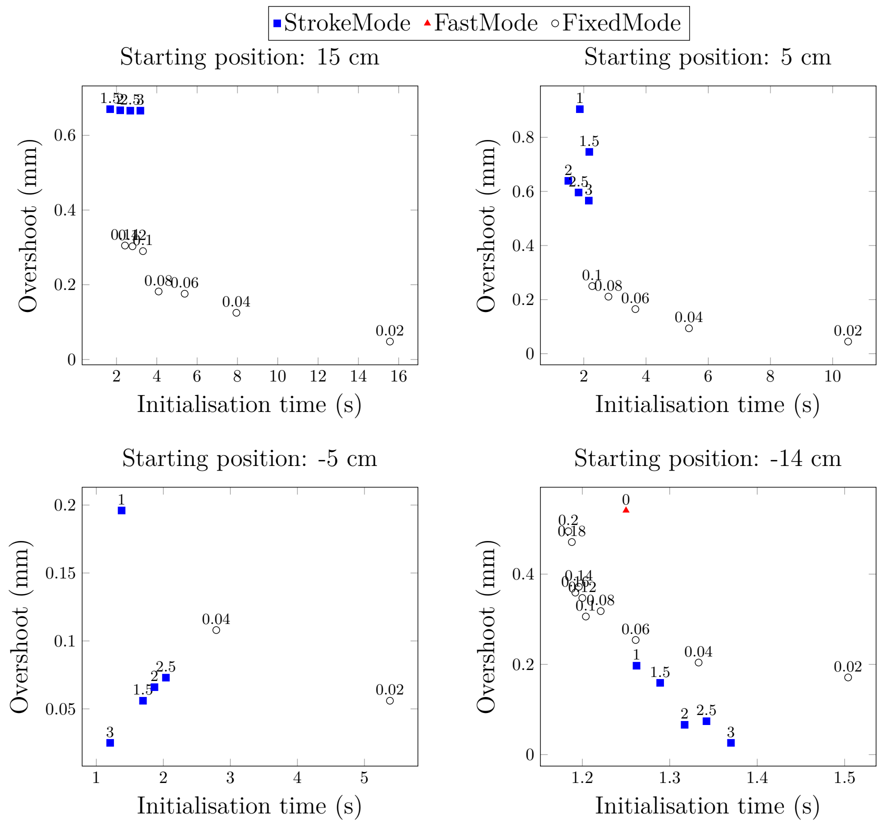


Figure 6.5: *DSE results for the different initialisation procedures of the SliderSetup. Every mark represents a single simulation configuration. Different marks and colours are used to represent the three possible initialisation modes. The coordinate is determined by its overshoot and time taken by the initialisation procedure. Movement speed is displayed above each data point. When no movement speed parameter was used, '0' is displayed.*

that exceeded the hard limit were ignored in the figure, as these do not meet the requirement of not having an overshoot of more than 2 mm. Initialisation procedures using the FastMode mostly caused a large overshoot, resulting in only one configuration to be displayed in the figure. A couple of observations from the DSE results are listed below.

- StrokeMode is typically faster in completing the initialisation, except when the initial position of the slider is very close to the limit switch, i.e. -14 cm.
- StrokeMode results in a bigger overshoot than FixedMode when the starting position of the slider is far away from the limit switch.
- Movement speeds exceeding 0.04 m/s are unsuitable in FixedMode, as they cause an unacceptable overshoot for starting position -5 cm and are therefor not displayed in Figure 6.5.

Based on other requirements or use cases, the most suitable initialisation procedure could be implemented. The use DSE here helps in separating suitable configurations from less suitable ones. The realised SliderSetup implements the initialisation procedure using the FixedMode with a movement speed of 0.04 m/s, since this method has a low overshoot in all cases as well as an acceptable initialisation speed.

6.6 Realisation

After a number of iterations, the models have become more detailed. The step from these models to realisation is small. According to step 8 of the design flow described in Section 4.1, the system can now be realised based on the models. The SliderSetup is physically constructed using rapid prototyping techniques based on the 3D models that have also been used for collision detection. For the motors and driver electronics a Raspberry Pi 3 is used in combination with a driver board that is capable of driving both motors and interfacing with the sensors. The Yocto Project[4] image for the Raspberry Pi 3 already implements the IO drivers for this board. The supervisory controller and slider controllers are implemented on this embedded board.

The embedded software consists of two parts: the control loops for the sliders and the supervisory control. Since 20-sim is capable of generating C++ code from a model, this will be used to generate code for the control loops. The remainder of the software consists of a JSON socket interface that interacts with the control loops together with the hardware interface. This implementation is developed manually as there is no direct model to code transformation available for POOSL.

However, transforming POOSL code to C++ is not difficult, as POOSL already shows the structure of the software. Additionally, the logic itself is basically a one-to-one translation from POOSL statements to C++ statements, which is rather straightforward. The overall development of the software can be split up into three steps.

1. Implementation of the interface to the management software. This can be tested without requiring the embedded board.
2. Implementation of the control loops. This requires the software to be tested on the Raspberry Pi instead of testing it on the development system (a regular PC).

[4]`https://www.yoctoproject.org/`

3. Implementation of the interface to the hardware components. This step requires all hardware to be in place to properly test all software functionality.

These steps show that a major part of the software development can be done without requiring any hardware components. The management software that runs on the PC and provides a graphical user interface (GUI) to the user to control the sliders was developed separately from the design flow that was described in this chapter. This software is developed in Python and uses the specified communication interface to communicate with either the model or real implementation of the supervisory controller. Since both the model and the implementation implement identical interfaces, the same piece of management software can be used to interact with either one of these.

After assembly of the system and installation of the software, the behaviour of the realised system closely resembles the behaviour as it was simulated. The resulting system is shown is Figure 6.6.

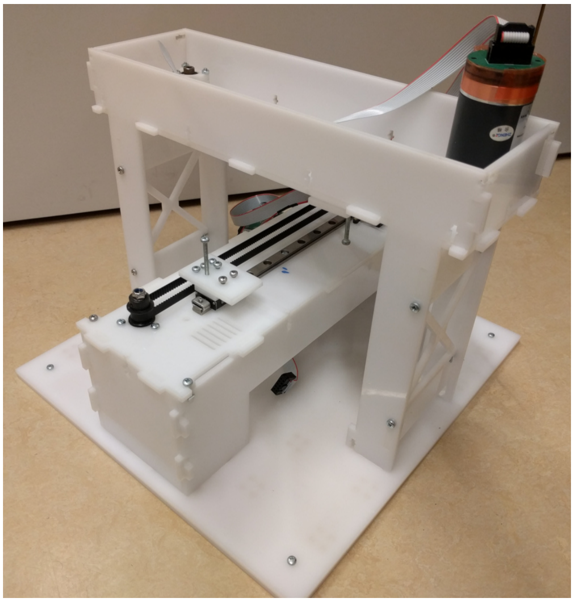


Figure 6.6: *The realised SliderSetup.*

6.7 Connector DSL

The supervisory controller of both the industrial case and the SliderSetup provide an interface that allows the user or another application to send commands to the system. These interfaces are added to the POOSL model using a socket for the communication. The protocol of the industrial case uses a well-structured and documented protocol using byte sequences to communicate. To allow the real software to connect to the POOSL simulation models, the POOSL model should be able to parse these byte sequences. By using a co-simulation of the physical components and the POOSL model as supervisory controller this approach would allow SIL testing. POOSL, however, handles socket streams as strings using UTF-8 encoding, causing only half of all possible bytes to be parsed properly. Consequently, all bytes above `0x7F` will become unreadable by the POOSL model.

To overcome this issue, a small bridging application was developed to convert byte sequences to JSON strings and vice versa. This application was developed in Python and could be used as man-in-the-middle to convert from one protocol to the other. As a result of this change, the POOSL model was adapted to parse the JSON strings.

This method was rather time consuming, as all conversions were to be implemented manually. A new DSL was therefor developed that allows the user to specify the protocol in terms of building blocks of the protocol. This DSL is called the Connector DSL and its grammar is included in Appendix E. It is developed using Xtext and Xtend and generates two pieces of code. First, a Python application is generated that is capable of transforming every building block of the protocol to and from bytes and JSON strings. The second file that is generated consists of the POOSL models for the protocol that are also able to parse the protocol to and from JSON, as well as providing getters and setters for the POOSL model. Listing 6.7 shows the protocol for a very simple messaging server.

```
Server {
  Name ChatServer
  Protocol TCP
  Type bytes
}
Client {
  Name ChatClient
  Protocol TCP
  Type json
}
Base Message

DataType Message {
  Components {
    string Sender
    string Receipient
    string Message
  }
}
```

Listing 6.7: *Sample protocol specification for a small messaging application using the Connector DSL.*

The specification starts with the definition of both ends of the connector: the server and the client. Each of these has a name and a networking protocol – TCP or UDP – and a type of communication. This type can either be `json` or `bytes` and specifies the communication method at each side of the connector. In the example, the server communicates using byte streams while the client uses JSON strings. The `Base` specifies the base datatype for all interaction. Every message transmitted from either of the two ends of the connector is parsed as this base datatype. After this basic configuration, one or more data types could be specified, where every datatype again consists of a number of components. Altogether, these data types specify the data structure that is used to communicate between the connector components.

The generated Python application is capable of transforming every message of the protocol to and from bytes and JSON strings. The application connects to a server socket and opens a socket for the client. Each of the sockets uses either JSON strings or bytes to communicate with the connected application while the application transforms the messages. Listing G.1 in Appendix G shows the Message class in Python that was generated from the protocol specification in Listing 6.7.

The generated POOSL classes only implement (de-)serialisation for JSON strings, as POOSL does not provide native support for bytes. The class implements getters and setters for each of the data types as well as `fromMap` and `toMap` methods to (de-)serialise from and to JSON. An example of a generated class is displayed in Listing G.2.

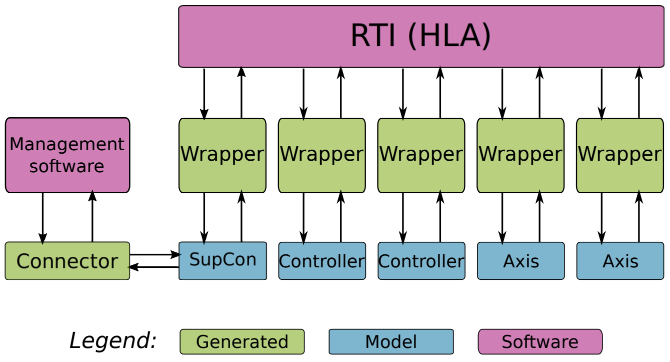


Figure 6.7: *SIL testing for the SliderSetup. The connector in the lower left corner is generated by the Connector DSL. The management software is an implemented piece of software.*

In the POOSL model, a process should be created that opens a socket for the connector to connect to. The latest version of POOSL can use an external port, which can be connected to the process. By using the `readLine` method on the socket, every line could be read as JSON string, resulting in a `Map` object. The `fromMap` method in the base class that was defined for the protocol can then be used to recursively parse the received message, after which this can be passed on to another process in the POOSL model via messages over the internal ports. Also, messages sent by the POOSL model can be sent in JSON format via the same translation in the opposite direction using the `toMap` method of the base class. This method allows both ends of the connector to communicate using their own format, following the same protocol. Figure 6.7 shows the co-simulation architecture of the SliderSetup with the generated connector in the lower left corner. Here, the management software communicates using byte streams and the supervisory controller *SupCon* uses JSON strings. The connector is generated by the Connector DSL.

As an alternative to this approach, a wrapper could have been developed for the management software to allow the implemented software to join the co-simulation as an ordinary federate. This approach would treat the management software as a simulation model and provide means to translate communication from and to the management software to HLA interactions. It does not allow the software synchronise its logical time with the simulated components, as the software itself does not support such time synchronisation mechanisms. Consequently, the management software communicates by means of a regular POOSL socket connection with the supervisory controller component simulation.

6.8 Conclusion

This chapter describes the industrial context in which the research in this dissertation was conducted. A CPS called the SliderSetup is introduced that reflects relevant characteristics of the industrial case. The design and development of this system is described. The design follows the design flow that was described in Section 4.1, from the specification of the requirements to the realisation of the system. Designs of the SliderSetup are analysed using CoHLA. This revealed a number of errors. Without the use of co-simulation, it is expected that these errors would have been found in a later stage or even when the system has already been realised. With the proposed approach, the errors were easily fixed, because they were found in an early stage of the development. It is therefor useful to be able to run a co-simulation early in the design of the system.

The co-simulation also provided a way to analyse whether a design of the system would meet its requirements. This supports the iterative development approach used to develop the models and shortens the feedback loop during the design. The use of automatic design space exploration was useful to find a suitable initialisation procedure and the collision simulator was used to find possible collisions of the sliders during the use of the system. After having realised the system, the co-simulated behaviour was found to be accurate enough to make design decisions.

The realised system meets the first requirement: it is capable of letting the sliders orbit each other with a speed of more than two rotations per second. It is very difficult to guarantee that the sliders will never collide with each other. Reason for this is that the control software does not know the positions of the two sliders when the system is started. The initialisation procedure was designed to determine their positions. Before this time, the sliders may still collide with each other, depending on their initial positions. To minimise the chances of the two sliders colliding, they are initialised one after another.

Finally, a new DSL called the Connector DSL is developed to quickly specify communication protocols. From such a specification, POOSL code is generated for use with a POOSL model to be able to use the protocol during co-simulation. This approach allows for SIL simulation by allowing an implemented software component to communicate with the model being co-simulated using the same protocol as the implemented system.

Reflection on requirements

1. *A co-simulation of simulation models of different disciplines can be constructed fast (20 models within 1 day).* Even though the co-simulation constructed for the described case studies consisted of less than 20 models, a co-simulation could be constructed within two hours once the models were ready to be simulated. Changes could also be applied really fast (requirement 2). Replacing a POOSL model for an FMU using identical interfaces required less than 5 minutes.

6. *The simulation results are trustworthy.* Comparing the co-simulation results with the implemented system shows that these behave very similar.

8. *The approach has support for automated design space exploration.* Automated co-simulation execution of a design space was used for finding a proper initialisation procedure for the SliderSetup. The feature showed to be useful for this.

12. *The framework runs on Windows, Linux and Mac.* Even though not mentioned in this chapter, notebooks running Linux and Mac were used for developing this system. This has shown that the framework is suitable for multi-platform usage.

CHAPTER 7

Scalability

Chapter 6 described the design of a small CPS – the SliderSetup – using a model-based approach and CoHLA for the generation of co-simulations. The use of CoHLA simplified the construction of the co-simulation and provided tools to perform analysis on the co-simulation to support the design process. The SliderSetup only consisted of 5 simulation models to be executed, resulting in 8 federates including the logger, metrics collector and collision detector. Many CPSs consist of much more components, resulting in very large co-simulations.

One type of such large CPSs are Internet of Things (IoT) systems. IoT systems typically consist of large numbers of interconnected sensors and actuators. With only a couple of sensors and actuators, such a system can easily be specified manually and the co-simulation of the system can simply be executed on a single computation node. However, for larger IoT systems that consist of many sensors and actuators, both the construction and execution of the co-simulation impose scalability challenges. A smart lighting system serves as a sample IoT system for experimenting with the scalability of HLA-based co-simulations and the construction of such co-simulations using CoHLA. This chapter proposes a method to execute a distributed co-simulation in the cloud and describes a new DSL to speed up the process of specifying a CoHLA co-simulation.

Section 7.1 first introduces the smart lighting system as well as the models used for its design. Then, Section 7.2 shows how the models could be co-simulated using CoHLA. A method to run distributed co-simulation with OpenRTI is described in Section 7.3. A DSL, called the Lighting DSL, for the rapid design of a smart lighting system and generation of CoHLA code is introduced in Section 7.4. Section 7.5 describes a number of experiments that use the Lighting DSL and distributed co-simulation execution for the specification and co-simulation of the smart lighting system. The chapter is concluded in Section 7.6.

7.1 Lighting system

An example of a large IoT system that consists of a large set of sensors and actuators is a smart lighting system. Such a smart indoor lighting system will serve as an illustrative case study regarding scalability challenges. The smart lighting system was inspired from ESI (TNO)[1]. Here, a simulation of the system as it is installed in the White Lady building in Eindhoven was used to perform robustness analysis on [18]. Co-simulations of similar systems have been created [69, 82] to serve as case studies for the development of a model-centric approach for designing such systems [17]. The goal in these studies is to build a virtual prototype of the system before actually building it. The virtual prototype can be used to analyse the behaviour and performance of the system and to find potential errors before building the system.

A lighting system consists of occupancy sensors, lights and controllers within a building. Occupancy sensors are used to detect human presence in the building. This information is used by controllers to set the state of the lights. A controller switches the lights on or off or sets their brightness level, based on information from the occupancy sensors.

Once a lighting system is installed on location it is usually very difficult to make changes to the system and to spot errors and fix them. It is therefor useful to analyse the system's behaviour for different scenarios, sensors and system configurations before the system is installed. Co-simulations can be executed to gain insight in the structure and working of the lighting system during the design of the system. Running a co-simulation of the system also provides a method to analyse performance characteristics such as the energy usage, which can also be used for optimisation purposes.

Different exemplary co-simulations of lighting systems have been created to illustrate the scalability challenges and to perform experiments with. For each experiment, the lighting system that was used for the experiment is described. The lighting systems have been designed following the same structures and models. The systems contain occupancy sensors that are able to detect human activity – usually motion – within their field of view. The sensors provide an *occupied* signal when activity is detected. The lights in the systems are all dimmable lights that can be turned off or turned on at a specific brightness level. The brightness level ranges from 0 to 100 and represents the power in percents. Lighting controllers receive activity information from the sensors and determine the target state of the lights.

Lighting systems usually cover a building or a part of a building. Buildings consist of multiple types of areas. Each of these types of areas could have a different lighting behaviour. The sample buildings used for the experiments only consist of two different types of areas: offices and corridors.

Lights in an office must be turned on when human activity is detected in the office. When no activity is detected for a while, the lights may be dimmed, after which they could be turned off. This behaviour provides the user some time to turn on the lights again when he or she is not being detected by the occupancy sensors, but still automatically turns off the lights after a while of inactivity.

[1]https://www.esi.nl/

Corridor lights behave differently compared to the office lights, as these should not be turned off when there are still people at work in the offices bordering the corridor. When there is no activity detected in the corridor itself, but there is still human activity in one of the connected rooms, the lights are dimmed instead of turned off. Only when there is no human activity in the corridor and none of the connected offices is occupied, the corridor lights may be turned off.

The lighting controllers are responsible for the behaviour of each area in the building, as they receive the occupancy information from the occupancy sensors and translate it into a target state for the connected lights. Therefore, every area has one lighting controller. While all instances of the lights and occupancy sensors are identical to each other, every type of area requires a different type of controller.

In a real building, human activity takes place, which is detected by the sensors of the lighting system. The lights are controlled based on this activity. To mimic user activity in the simulated system, an actor will be created. The actor performs a scenario that is described by a set of coordinates for specific periods of simulation time. The actor's behaviour is considered to be the input of the system.

Models

According to the design flow described in Section 4.1, the first step is to specify the requirements. These have been formulated in an informal manner in the previous paragraphs. For each of the three components, a basic type of model has been developed for the co-simulation of the lighting system. The co-simulation consists of many simulation instances of these models, e.g. multiple instances of a simulation of a light and multiple instances of a occupancy sensor simulation. The models and their interfaces are briefly described in this section, following steps 2 and 3 of the design flow.

Occupancy sensor

The model of an occupancy sensor is developed as a discrete-time model in 20-sim. The mode has one output representing the *occupied* signal. As input, the model receives the location of an actor in the building using coordinates. These coordinates are used in the model to determine whether the actor is within the field of view of the sensor. Table 7.1 shows the attributes of the occupancy sensor model.

Occupancy sensor – Attributes			
In/Out	**Name**	**Type**	**Description**
Input	actorX	`real`	The x-coordinate of an actor.
Input	actorY	`real`	The y-coordinate of an actor.
Output	occupied	`boolean`	Whether or not the sensor detected any activity, `true` if activity is detected, `false` if not.

Table 7.1: *Input and output attributes of the occupancy sensor model.*

To determine whether the actor's coordinates are within its field of view, the sensor model must be aware of its own location and line of sight. To prevent the model from detecting activity through walls, coordinates of the area in which the sensor is located should be provided. These coordinates describe the bounding box for presence detection. A timeout is included in the sensor model to specify the timeout during which the *occupied* signal is given. This information can be passed to the model by using model parameters. The list of parameters is shown in Table 7.2.

Occupancy sensor – Parameters		
Name	**Type**	**Description**
range	`real`	The line of sight of the sensor. Specified in cm.
timeout	`real`	The timeout for which the sensor remains in an *occupied* state after detecting presence. Specified in seconds.
positionX	`real`	Position specified in separate x and y coordinate components. Specified in cm.
positionY	`real`	
boxMinX	`real`	Top left (*min*) and bottom right (*max*) locations of the (rectangular) bounding box in separate x and y coordinates. Specified in cm.
boxMinY	`real`	
boxMaxX	`real`	
boxMaxY	`real`	

Table 7.2: *Model parameters of the occupancy sensor model.*

Light

The dimmable light is modelled in 20-sim as a continuous-time model. The model receives a target brightness level called setpoint as input. This setpoint is expressed as a real value from 0 to 100, representing the brightness level in percents. The light outputs its actual brightness level in percents. Table 7.3 displays the attributes of the model.

Light			
In/Out	**Name**	**Type**	**Description**
Input	setpoint	`real`	The target brightness level.
Output	power	`real`	The current brightness level.

Table 7.3: *Input and output attributes of the light model.*

Controller

For each of the area types, a separate controller model is created. Every controller has an input for the detection of activity by the connected occupancy sensors. Section 4.4.6 describes how the attributes of multiple sensor models are connected to one single input attribute in the controller model. Corridor lighting controllers also receive additional input from bordering areas. This model has one additional input for the related activity detected by these sensors. Every controller outputs a target brightness level in percents for the connected lights, thus they all have the same target brightness. Table 7.4 shows the attributes of the lighting controller model.

Lighting controller – Attributes			
In/Out	**Name**	**Type**	**Description**
Input	activity	`boolean`	Whether or not the connected occupancy sensors detect any activity.
Input	*relatedActivity*	`boolean`	Whether or not the connected occupancy sensors in related areas detect any activity.
Output	setpoint	`real`	The target brightness level of the connected lights.

Table 7.4: *Input and output attributes of the lighting controller model. Optional attributes are in italic font.*

All lighting controllers have at least three states: *active*, *absent* and *off*. When activity is detected by one of the connected occupancy sensors, the controller state is switched to *active*. Consequently, a timer is started and the connected lights are set to the setpoint specified for the *active* state. When the timer expires without any new activity being detected, the *absent* state is activated, after which a new timer is started and the lights are switched to a setpoint for the *absent* state. The controller state is set to *off* when this timer expires. Consequently, all timers are reset and the lights are turned off. When a timer is interrupted by any new activity being detected, the state is switched to the *active* state again, also restarting this timer.

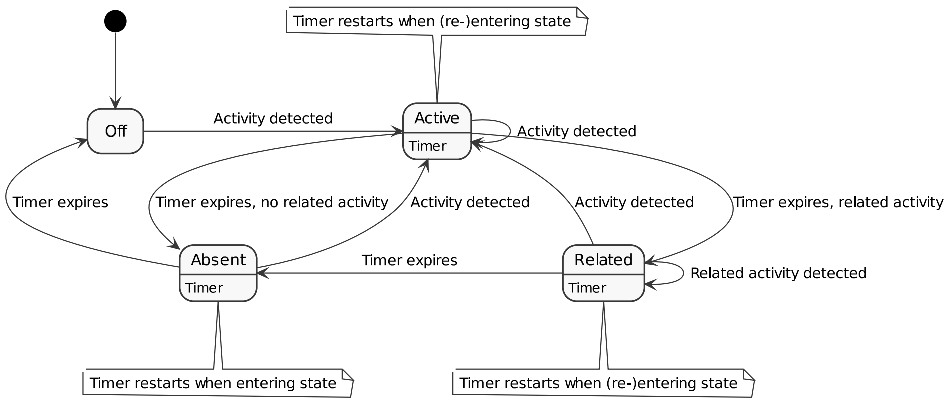

Timer restarts when (re-)entering state
Off
Activity detected
Active
Timer
Activity detected
Timer expires
Timer expires, no related activity
Activity detected
Activity detected
Timer expires, related activity
Absent
Timer
Timer expires
Related
Timer
Related activity detected
Timer restarts when entering state
Timer restarts when (re-)entering state

Figure 7.1: *State diagram of the corridor controller.*

Since the corridor controller has an additional input for related activity, this model also has an additional state called *related*. This state has its own setpoint and timer, which is started when no direct activity is being detected, but related areas do report activity. While this condition holds, the state will be *related*, unless direct activity is detected. A state diagram for the corridor controller is displayed in Figure 7.1. The parameters for the model are displayed in Table 7.5. The lighting controller models are all discrete-time models and have been developed in POOSL.

Lighting controller – Parameters		
Name	**Type**	**Description**
activeTimeout	`real`	The timeout and brightness level for the active state.
activeLevel	`real`	
absentTimeout	`real`	The timeout and brightness level for the absent state.
absentLevel	`real`	
relatedTimeout	`real`	The timeout and brightness level for the related state.
relatedLevel	`real`	

Table 7.5: *Model parameters of the lighting controller model. Optional parameters are in italic font.*

Figure 7.2 shows how the components of a small lighting system, consisting of only one office, are connected. Since this system only consists of one area, just one controller is required. An actor provides input coordinates for the occupancy sensors. The actor is a simple federate in the co-simulation that plays a pre-specified scenario. The controller receives its inputs from the sensors and outputs its setpoint to the connected lights.

For our experiments, two different lighting controllers were used: the corridor controller and a controller for basic rooms. The corridor controller has been explained in this section, while the *BasicRoomController* is a simple lighting controller that only receives input from a set of occupancy sensors located in the same room. The state diagram for this controller is therefore similar to the one displayed in Figure 7.1, except that the *related* state is removed.

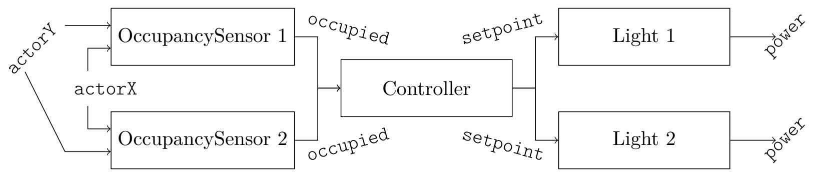


Figure 7.2: *Architecture of a small lighting system.*

7.2 Co-simulation

Step 4 of the design flow displayed in Figure 4.1 requires the specification of a co-simulation of the system. This process is explained in the current section and the following sections. Since the focus of this case study is to analyse the scalability of specifying and running the co-simulation, no further steps of the design flow have been taken.

To construct a co-simulation of a lighting system, the 20-sim models are exported to FMUs. For each of the models, a federate class specification is created in CoHLA, after which a federation could be specified. Listing 7.1 shows the federate class for the occupancy sensor. In addition to the input and output attributes, all model parameters are listed. Note that the parameters specified on lines 11 to 16 describe only three coordinates, each of them split up into separate x and y components. Refer to the CoHLA user manual or Section 4.4 for more information on the specifications.

```
 1  FederateClass OccupancySensor {
 2    Type FMU
 3    Attributes {
 4      Input Real actorXPosition as "x_position"
 5      Input Real actorYPosition as "y_position"
 6      Output Boolean occupied
 7    }
 8    Parameters {
 9      Real timeout    "threshold"
10      Real range      "bounded_range.range"
11      Real positionX  "bounded_range.origin[0]"
12      Real positionY  "bounded_range.origin[1]"
13      Real boxMinX    "bounded_range.bounds[0]"
14      Real boxMinY    "bounded_range.bounds[1]"
15      Real boxMaxX    "bounded_range.bounds[2]"
16      Real boxMaxY    "bounded_range.bounds[3]"
17    }
18    DefaultModel "models/OccupancySensor.fmu"
19    AdvanceType NextMessageRequest
20    DefaultStepSize 5.0
21    DefaultLookahead 0.1
22  }
```

Listing 7.1: *Federate class for the occupancy sensor model.*

All federate classes advance in time using a NextMessageRequest, so that they will receive updates as soon as they are available, accompanied by a TimeAdvanceGrant (TAG) for the corresponding simulation time. This method allows updates to be sent to every federate once they become available. Consequently, the step size for each of the models is chosen for logging purposes only. The lookahead value for each of the models is set to 0.1 seconds.

Listing 7.2 shows the federate class specification for the light. Apart from the input and output attributes, only one model parameter has been specified. The parameter allows the user to modify the speed at which the power level of the light changes.

```
FederateClass DimmableLight {
  Type FMU
  Attributes {
    Input Real setpoint
    Output Real power
  }
  Parameters {
    Real GainK "Gain.K"
  }
  DefaultModel "models/DimmableLight.fmu"
  AdvanceType NextMessageRequest
  DefaultStepSize 1.0
  DefaultLookahead 0.1
}
```

Listing 7.2: *Federate class for the light model.*

Listing 7.3 displays the specification for a corridor controller. The model has two inputs, which have an input operator specified to combine multiple input values. This operator is described in Section 4.4.6. All parameters that have been described previously are specified.

```
FederateClass CorridorController {
  Type FMU
  Attributes {
    Input Boolean [||] activity
    Input Boolean [||] relatedActivity
    Output Real setpoint
  }
  Parameters {
    Real activeTimeout
    Real activeLevel
    Real absentTimeout
    Real absentLevel
    Real relatedTimeout
    Real relatedLevel
  }
  DefaultModel "models/CorridorController.fmu"
  AdvanceType NextMessageRequest
  DefaultStepSize 10.0
  DefaultLookahead 0.1
}
```

Listing 7.3: *Federate class for the lighting controller model for a corridor.*

To mimic user activity in the co-simulated building in an automated manner, a federate class for an actor is specified. This class specification is displayed in Listing 7.4. The actor federate class only has two output parameters; one for each dimension in the two-dimensional space. When connected to an occupancy sensor, these outputs provide the sensor with input. Note that the simulator type is set to `None`, thus no simulator is used and no default model is given. Consequently, the generated wrapper code for this federate class has only limited functionality, as it is only capable of connecting to the RTI, requesting time steps and playing a scenario or fault scenario. The federate is basically an empty federate that holds two attributes – x and y – which together form the location of the actor in the building.

```
FederateClass Actor {
  Type None
  Attributes {
    Output Real xPosition
    Output Real yPosition
  }
  DefaultStepSize 5.0
  DefaultLookahead 0.1
}
```

Listing 7.4: *Federate class specification for the actor.*

By using a CoHLA scenario to change these attributes and publish them in the co-simulation, the actor moves according to the scenario. The federate class therefor does not require a model to be executed to perform this scenario. However, it is still possible to create a separate model that mimics a similar scenario.

A federation specification in CoHLA for a small sample building (*VerySmall-Building*) is displayed in Listing 7.5. The building consists of two basic rooms that are connected by a corridor. Every area contains two lights and one occupancy sensor. The occupancy sensors of the rooms are connected to the *related activity* input of the controller for the corridor (lines 27 and 28). One actor is added to provide input to the occupancy sensors (lines 29 to 34). Lines 36 to 45 also show a small part of a sample scenario for the actor.

```
1  Federation VerySmallBuilding {
2    Instances {
3      room1Controller : BasicRoomController
4      room1_l1 : DimmableLight
5      room1_l2 : DimmableLight
6      room1_s1 : OccupancySensor
7      room2Controller : BasicRoomController
8      room2_l1 : DimmableLight
9      room2_l2 : DimmableLight
10     room2_s1 : OccupancySensor
11     corridorController : CorridorController
12     corridor_l1 : DimmableLight
13     corridor_l2 : DimmableLight
14     corridor_s1 : OccupancySensor
15     actor : Actor
16   }
17   Connections {
18     { room1_l1.setpoint <- room1Controller.setpoint }
19     { room1_l2.setpoint <- room1Controller.setpoint }
20     { room1Controller.activity <- room1_s1.occupied }
21     { room2_l1.setpoint <- room2Controller.setpoint }
22     { room2_l2.setpoint <- room2Controller.setpoint }
23     { room2Controller.activity <- room2_s1.occupied }
24     { corridor_l1.setpoint <- corridorController.setpoint }
25     { corridor_l2.setpoint <- corridorController.setpoint }
26     { corridorController.activity <- corridor_s1.occupied }
27     { corridorController.relatedActivity <- room1_s1.occupied }
28     { corridorController.relatedActivity <- room2_s1.occupied }
29     { room1_s1.actorXPosition <- actor.xPosition }
30     { room1_s1.actorYPosition <- actor.yPosition }
31     { room2_s1.actorXPosition <- actor.xPosition }
32     { room2_s1.actorYPosition <- actor.yPosition }
33     { corridor_s1.actorXPosition <- actor.xPosition }
34     { corridor_s1.actorYPosition <- actor.yPosition }
35   }
36   Scenario JustWalk {
37     AutoStop: 3600.0
38     0.0: actor.xPosition = "0.0"
39     0.0: actor.yPosition = "375.0"
40     10.0: actor.xPosition = "450.0"
41     10.0: actor.yPosition = "375.0"
42     ...
43     1820.0: actor.xPosition = "0.0"
44     1820.0: actor.yPosition = "375.0"
45   }
46 }
```

Listing 7.5: *Federation specification in CoHLA for a small building called VerySmallBuilding.*

7.3 Distributed co-simulation

For larger buildings, the number of simulators that runs in parallel increases. Inevitably, this slows down the simulation rapidly. To reduce the simulation time required for co-simulating a large lighting system, the co-simulation execution is distributed over a set of computation nodes. The HLA standard allows for distributed co-simulation, which is also supported by OpenRTI, the RTI that is used by CoHLA. This section outlines the method by which CoHLA supports the distributed co-simulation of a system.

7.3.1 Distribution architecture

Federates are all connected to the RTI via regular socket connections, which allows federates to be started on a different computation node than on which the RTI is executed. Figure 7.3 shows the connection architecture of a co-simulation consisting of four federates that are distributed over two nodes. Only *Node 1* runs the RTI while the federates on the other federate connect directly to this RTI.

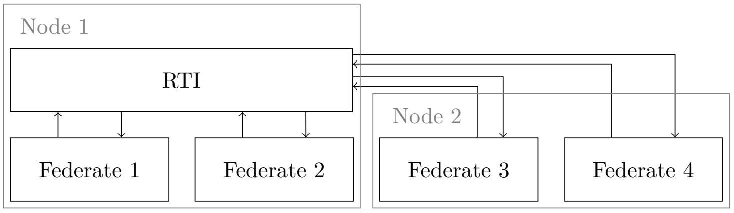


Figure 7.3: *Distributed co-simulation using a single RTI.*

OpenRTI also supports an architecture in which every node runs its own RTI. These RTIs all connect to one RTI that acts as a master. Figure 7.4 shows the connection architecture using this approach.

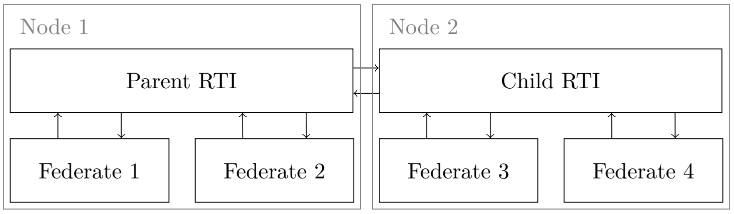


Figure 7.4: *Distributed co-simulation using multiple RTIs.*

Using the multi-RTI approach, the communication between the nodes can be bundled and compressed, which reduces network traffic. It may also provide a way to filter the updates to only share updates with other RTIs when they also affect the federates running on those nodes. Such a filter could reduce the amount of network traffic even more and reduces the number of irrelevant updates received by the federates, which might allow them to run even faster. The multi-RTI approach is therefore expected to have a better performance, which was confirmed by performing a number of small experiments. All distributed co-simulations that follow use this multi-RTI approach.

7.3.2 Distribution implementation

CoHLA already generates a run script for each federation that allows the user to easily start and configure the co-simulation. This generated run script was extended to provide an easy method to start the co-simulation in a distributed manner as well. Two different methods of distribution are implemented, both using the multi-RTI approach as described before.

The first method allows the user to assign a weight to each of the federate classes. This weight value represents the simulation complexity of the model and should be a non-zero positive integer. A higher number represents a more compute-intensive simulation model. Every federate class is assigned a weight of 1 by default, which implies every simulation to be equally heavy unless specified otherwise. The user is able to modify the weight in the federate class specification in CoHLA. To start the distributed co-simulation, the user must start the run script on every node, providing the total number of nodes and a unique node identification number (ID). Based on the weights of the federates, the script determines a distribution of the federates across all nodes and starts the federates that match its own ID. The fact that every execution of the run script on the different nodes determines the same distribution and has a different ID ensures that all federates are started. This does not require any manual distribution being specified, which makes it easy to use.

The second method allows the user to manually specify a distribution of the federate across a predefined number of nodes. Such a distribution method allows the user to group federates together on one node to potentially reduce network traffic. A distribution specification is identified by a name and has a fixed number of nodes. For each of the nodes, a list of federates that should be executed on it should be provided. An example of a co-simulation distribution – named *dist3* – over three nodes is displayed in Listing 7.6. The federation for which this distribution is made consists of 8 federates, having names *fed1* to *fed8*.

```
Distribution dist3 over 3 systems {
  System 0: fed1 fed3 fed4
  System 1: fed2 fed5 fed6
  System 2: fed7 fed8
}
```

Listing 7.6: *Sample of a manual co-simulation distribution specification.*

Both methods for starting the distributed co-simulation require the user to specify an ID for the node on which the run script is started. The node that was assigned ID 0 acts as the master node and starts the parent RTI. The other nodes are child nodes and start the RTI in child mode. These child RTIs connect to the parent RTI. For this, all child nodes must be provided with an IP address and optionally a port of the parent node's RTI. These properties can be provided to the run script upon starting. If none of the above parameters is provided to the run script upon starting a co-simulation, the node is assumed to be the parent node that has ID 0.

7.4 Lighting DSL

Even though a co-simulation of the lighting system could be constructed rather easily using CoHLA, the specification in Listing 7.5 is still lacking required information. For instance, the locations and bounding boxes of the sensors have not yet been specified using parameter configurations. In order for the co-simulation to simulate the intended building, every occupancy sensor needs 7 parameters to be specified: its line of sight, position and coordinates for the bounding box. Especially for larger lighting systems, the specification of these parameters forms a scalability challenge.

To reduce the effort required to specify a co-simulation for a lighting system, a separate DSL was developed, which is called the Lighting DSL. The grammar for the Lighting DSL is included in Appendix C. The DSL generates a CoHLA specification of the system, after which this specification generates all required sources and configuration files. The Lighting DSL was developed to decrease the amount of manual labour that it requires to specify a large lighting system in CoHLA.

The Lighting DSL allows the user to design a lighting system by specifying areas and their contained devices. Following this approach, the user describes a building with its lighting components in a more natural way than describing the system as a co-simulation by connecting attribute and specifying separate x and y coordinates for each component. From such a building specification, a CoHLA specification is generated, as well as an image of the designed system. From the CoHLA specification, the user can generate the co-simulation and required configuration files. The Lighting DSL is developed using the same technologies that were used to develop CoHLA: Xtext and Xtend. Figure 7.5 displays the two-step approach in generating the co-simulation.

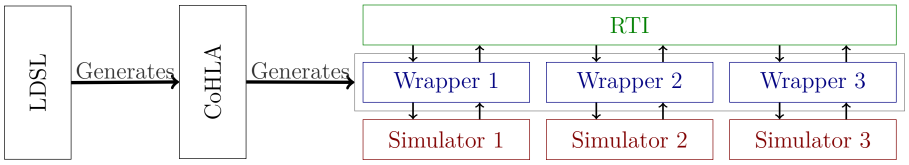


Figure 7.5: *The two-step approach of generating the co-simulation. The green box on top represents software that is used, red boxes in the bottom represent the models being simulated and blue boxes in the middle represent generated code from the DSLs.*

7.4.1 Language

In the Lighting DSL, a building has a unique name and contains a number of areas. These areas are either corridors or rooms. Every room may be specified to be of a certain type, such as a basic office or an office landscape. For each area, its corner coordinates should be specified to enable the generation of a figure of the building as well as providing a bounding box to the occupancy sensors in the room. Occupancy sensors and lights could be added to the devices list of a room. Each of these devices also requires coordinates, a name and optionally a specific type. Changing the type allows for the inclusion of different types of lights and sensors to the building. Every corridor also requires a list of the areas it is bordering to. Listing 7.7 shows a specification of the *VerySmallBuilding* sample using the Lighting DSL.

```
Building VerySmallBuilding {
  Room room1 {
    Area: (0,0) (600,0) (600,300) (0,300)
    Devices {
      Light l1 on (150,150)
      Light l2 on (450,150)
      Sensor s1 on (300,150)
    }
  }
  Room room2 {
    Area: (0,450) (600,450) (600,750) (0,750)
    Devices {
      Light l1 on (150,600)
      Light l2 on (450,600)
      Sensor s1 on (300,600)
    }
  }
  Corridor c1 {
    Area: (0,300) (600,300) (600,450) (0,450)
    Rooms: room1 room2
    Devices {
      Light l1 on (150,375)
      Light l2 on (450,375)
      Sensor s1 on (300,375)
    }
  }
}
```

Listing 7.7: *The specification of a sample Building (VerySmallBuilding) using the Lighting DSL.*

```
1 Configuration {
2   #Actors: 1
3   OccupancySensorRanges: 300
4   HasLogger
5   MeasureTime: 3600
6   Distributions: 2 7 14
7   ModelDir: "../../models"
8 }
```

Listing 7.8: *The configuration of the VerySmallBuilding specified in the Lighting DSL.*

In addition to the specification of the structure of the building itself, the DSL also allows for some basic configuration. For example, the line of sight of all occupancy sensors in the system can be modified. A logger can be added automatically and its default measure time may be set. Listing 7.8 shows a sample configuration for the *VerySmallBuilding*. Grouped distributions can be generated by the Lighting DSL by specifying one or more numbers representing the number of nodes that participate in the distribution (line 6). The directory in which the models are located could also be configured (line 7).

Line 2 specifies the number of actors to be included in the system. The actor federates have a scenario that updates its coordinates. These scenarios should be specified using the Lighting DSL for every actor that is included in the system. Such a scenario is displayed in Listing 7.9.

```
    Scenario JustWalk for Actor 0 {
      (0,375)
      [10] (450,375)
      [5] (450,150)
      [120] (450,150)
      [5] (275,375)
      [10] (250,550)
      [180] (250,550)
      [5] (275,375)
      [10] (580,375)
      [5] (450,375)
      [10] (450,150)
```

```
        [1440] (450,150)
        [10] (300,375)
        [10] (0,375)
    }
```

Listing 7.9: *A sample scenario for an actor in the lighting system.*

The scenario has a name (*JustWalk*) and an identification number for the actor, which starts at 0 and is incremented for each actor that is added. The coordinates of the starting point are specified first (line 2), after which a list follows that specifies a delay and the next coordinate. For each of these scenarios, a CoHLA scenario specification is generated. A sample of the CoHLA scenario that is generated was already displayed in Listing 7.5.

7.4.2 Code generation

From a building specification using the Lighting DSL, a CoHLA specification is generated. This CoHLA specification includes federate classes, instances, connections, scenarios and a set of situations that describe all model parameters of the different instances. Using the generated CoHLA specification, co-simulation code can be generated that allows the easy initialisation and execution of the co-simulation. CoHLA supports two distribution methods: automatic (weight-based) and manual as explained in Section 7.3.2. By specifying the number of computation nodes to create a distribution for, the Lighting DSL allows the user to generate a predefined distribution configuration. Multiple distribution sizes can also be provided to generate more grouped distributions. An example is displayed in Listing 7.8 on line 6. The distributions that are generated group all federates for one area on the same computation node: in the example, distributions are generated for 2, 7 and 14 computation nodes. In the *VerySmallBuilding* system, a distribution over three nodes will therefor have grouped all lights, sensors and the controller for each of the three areas on one single computation node. Consequently, every node runs all simulations for one area. When the *VerySmallBuilding* is distributed over two nodes, one of the nodes will run all federates for two rooms while the other node runs the federates of the remaining room. Since most of the communication between federates is isolated within one area and therefor one node, this method potentially minimises the network traffic between the nodes.

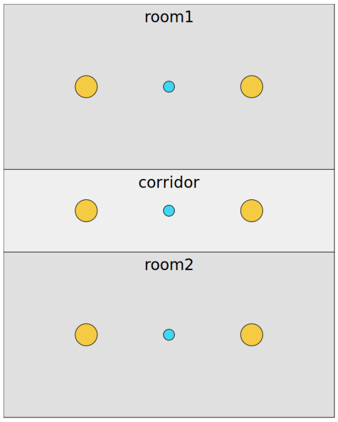


Figure 7.6: *Generated image of the VerySmallBuilding.*

Names of the models, such as *Occupancy-Sensor.fmu*, are hardcoded in the code generator of the Lighting DSL, thus cannot be changed without making a small change to the code generator. The folder that contains the simulation models can be specified using the Lighting DSL. The specified directory is passed on to the generated CoHLA specification as the default path for the models. From the CoHLA specification, code can be generated to be compiled and executed.

Additionally, a figure of the building is generated using the scalable vector graphics (SVG) format. The coordinates are used to position the different devices. For every occupancy sensor, a circle is drawn indicating the line of sight of the sensor. These lines can optionally be hidden using the building's configuration. Figure 7.6 shows the image that was generated from the building specification displayed in Listing 7.7. In the image, rooms and corridors are displayed as rectangles, occupancy sensors and small blue circles and lights as larger yellow circles.

From the resulting log file of a co-simulation execution, it is hard to comprehend the system's behaviour. To visualise this log file, the same figure that is mentioned above is also used in a web page that is generated for each building. This web page allows the resulting log file to be read and to replay the simulation. The state of each of the sensors and lights is indicated to the user by using colours. One actor can be included in the visualisation, which is moved according to its positions stored in the log file. The web interface allows for automatically replaying the co-simulation as well as manually stepping through it. Figure 7.7 shows the web interface for the *VerySmallBuilding*.

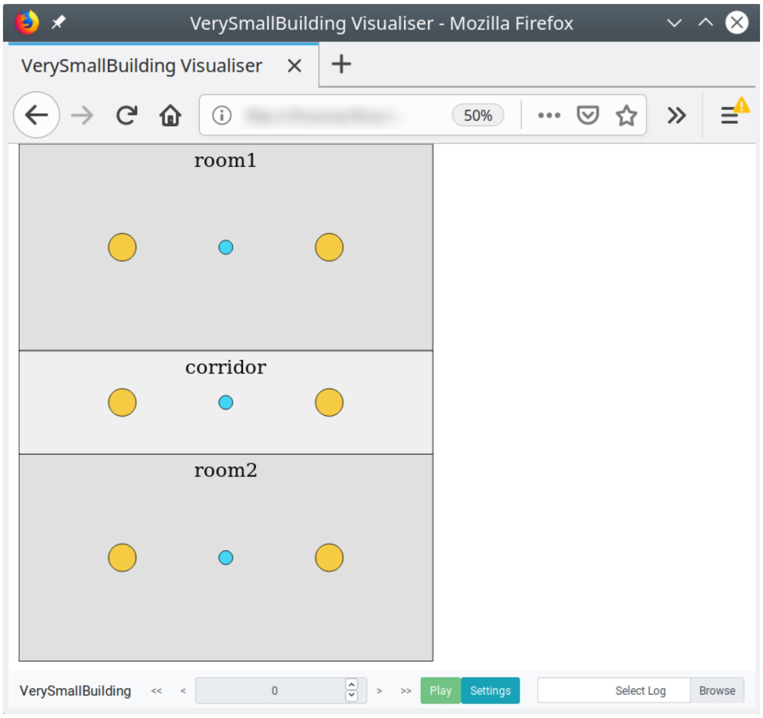

VerySmallBuilding Visualiser - Mozilla Firefox
VerySmallBuilding Visualiser
50%
room1
corridor
room2
VerySmallBuilding
<<
<
0
>
>>
Play
Settings
Select Log
Browse

Figure 7.7: *Web interface for the VerySmallBuilding to replay log files.*

7.5 Results

This section describes the experiments that were conducted to analyse the scalability of HLA and CoHLA. To construct co-simulations in CoHLA, the Lighting DSL is used.

7.5.1 Cloud nodes

All experiments are conducted on a set of cloud nodes. These nodes are virtual systems (Droplets) rented at Digital Ocean[2]. All nodes have identical specifications as far as we can control them. Due to the fact that we cannot control the physical hardware on which the nodes run, there may be small differences between the nodes. The nodes are optimised for compute-intensive tasks and connected with each other via an internal network. The high-level specifications of the nodes are displayed in Table 7.6.

CPU	8 virtual CPUs
Memory	16 GB
Disk	100 GB SSD
Network	40 GbE
Operating system	Ubuntu 18.04

Table 7.6: *High-level specifications of the computation nodes.*

7.5.2 Experiment execution

The experiments are executed in an automated manner. To do this, a python script was developed to automatically connect to all nodes using SSH. Every node is expected to run a clean installation of Ubuntu 18.04 server edition. The script first downloads and installs all dependencies on the system, after which it starts a specified set of benchmarks. Each of the benchmarks is an identical run of the co-simulation, which is distributed over a number of nodes. The number of runs and distributions to be executed is specified in a configuration file. The script handles the execution of each of the specified distributed co-simulations and aggregates the results into an output CSV file. Metrics such as the minimum, maximum and average execution time are computed and included in the output file.

The script requires two configuration files. The first configuration file specifies the nodes the script should connect to. Every node is specified by a name, external and internal IP address, port number of the SSH server and login details.

```
node1;81.82.83.81;22;root;password;192.168.2.101
node2;81.82.83.82;22;root;password;192.168.2.102
node3;81.82.83.83;22;root;password;192.168.2.103
```

Listing 7.10: *Sample configuration specifying three computation nodes.*

The second configuration file specifies the benchmarks that must be executed. A sample configuration file is shown in Listing 7.11. The configuration file specifies the following.

```
source:https://example.org/dist.tar.gz
path:Lighting/src-gen/SampleBuilding
topology:conf/sampleBuilding.topo
situation:conf/SampleBuilding/base150.situation
scenario:conf/SampleBuilding/BasicEnterWorkLeave.scenario
measuretime:120.0
```

[2]https://www.digitalocean.com/

```
7  runs:3
8  nodist1;AUTO;1;node1
9  nodist2;AUTO;1;node2
10 nodist3;AUTO;1;node3
11 autodist2;AUTO;2
12 dist2;conf/SampleBuilding/dist2.distribution;2
13 autodist3;AUTO;3
14 dist3;conf/SampleBuilding/dist3.distribution;3
```

Listing 7.11: *Sample configuration file for the automated distribution benchmarking.*

- An URL where the distribution archive of the co-simulation can be downloaded (line 1). The package contains the CoHLA library sources and co-simulation sources, configurations and models.
- The relative path in the distribution archive where the co-simulation sources are located (line 2).
- The topology of the co-simulation to be executed (line 3).
- The situation containing all model parameter values for the co-simulation to use (line 4).
- The scenario to run in the co-simulation (line 5).
- The maximum period of (wall clock) time to let the distributed co-simulation run (line 6). This timeout is used to detect nodes that might have crashed.
- The number of times to execute each distribution of the co-simulation (line 7).
- A list of benchmarks to execute (from line 8 onward). Every benchmark starts with a name of the benchmark, followed by the path to the predefined distribution to use or `AUTO`. When `AUTO` is used, the distribution is created automatically using the weight-based method provided by the CoHLA run script. After this, the number of computation nodes for the distribution is given. Optionally, this is followed by a list of nodes – identified by their name – that should be part of the distribution. The length of the list should be equal the the number of nodes that runs the distributed co-simulation. When no nodes are specified for the benchmark, the nodes are automatically selected based on availability.

7.5.3 POOSL to FMU

Initially, the lighting controller models were developed in POOSL. Early experiments, however, showed that the simulation speed of POOSL models in HLA using our library was limited. A similar model was created using 20-sim, as this model could be exported to an FMU. FMUs appear to have a better simulation speed compared to POOSL, since these do not rely on socket connections to be controlled by the CoHLA libraries.

7.5.4 Distribution methods

To analyse the performance impact of running a distributed co-simulation, a lighting system was designed that is larger than the example that was previously given. The building consists of eleven offices and three corridors connecting them. The system is called *MediumFloor* and is displayed in Figure 7.8. The MediumFloor has 36 lights, 38 occupancy sensors and 14 light controllers. Even though the figure only shows 36 occupancy sensors, two positions contain two sensors. This is the consequence of the implementation of the bounding boxes for the line of sight in the sensor models. Intersections of corridors require each corridor to have a sensor to detect possible activity. Since the system has two corridor intersections, the sensors located on these intersections were added in each of the corridors, doubling the sensor at this specific positions. Consequently, with two intersections in the system, this results in two additional sensors. A logger federate and an actor that walks through the building according to a provided scenario are added to the system's specification. The final co-simulation consists of 90 federates. The specification of the *MediumFloor* system in the Lighting DSL is included in Listing D.1.

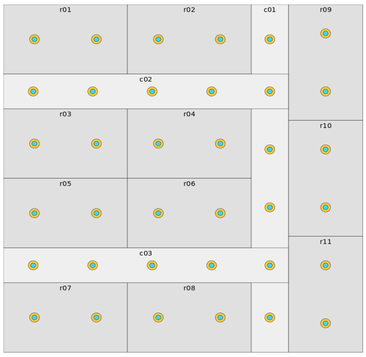
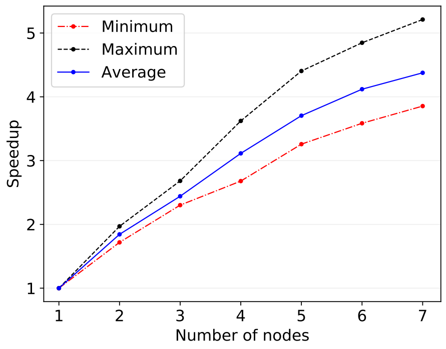


Figure 7.8: *The MediumFloor sample lighting system.*

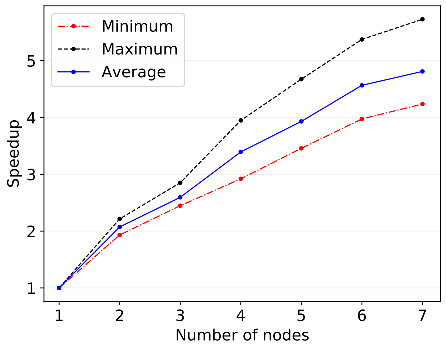
A scenario was created that mimics the behaviour of an employee entering the office in the morning. The employee enters the building, moves to his or her office, gets a cup of coffee, attends a meeting and moves back to the office. After that, he or she fetches some more coffee a couple of times before leaving the building again. The scenario describes the activity during three hours, after which the simulation is stopped.

The co-simulation is executed on nodes that were described in Section 7.5.1. It is run on one single node up to seven nodes in parallel. The initial speed is measured on a single node. For every number of nodes that execute the distributed co-simulation, the speedup compared to the speed of a single node is calculated. For this, the simulation time is compared to the slowest node (*maximum* speedup), the fastest node (*minimum* speedup) and the average simulation time of the participating nodes (*average* speedup). For this comparison, only participating nodes are considered.

The experiments were executed using the script that was described in Section 7.5.2. Every distribution is run three times, after which the averages of these runs are taken. This is done once for each of the two distribution methods: weight-based using the CoHLA run script and grouped using a generated distribution over a number of nodes by the Lighting DSL. The speedup of both distributed co-simulations is displayed in Figure 7.9. The simulation time when running the co-simulation on a single node ranged from 345 to 452 seconds, depending on the node. For every co-simulation execution, the log file was checked to verify that the simulation was correctly executed. The logs were checked by checking their hash sums with a log that was manually verified to be correct, e.g. show the expected



(a) *Automatic distribution.*



(b) *Manual distribution.*

Figure 7.9: *Speedup results of distributing the MediumFloor co-simulation using two different distribution methods.*

behaviour in the simulation.

Both the weight-based automatic distribution (Figure 7.9a) and the grouped distribution (Figure 7.9b) scale well, although the effect of adding nodes becomes smaller when 7 nodes are used. The scalability looks rather similar, which is shown in Figure 7.10. Although the grouped distribution as generated by the Lighting DSL scales better across all numbers of nodes, the difference is relatively small.

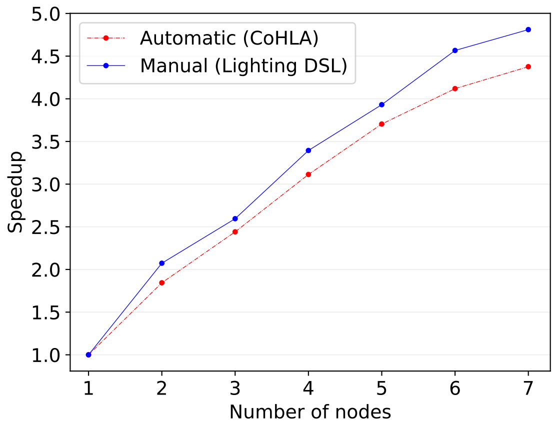


Figure 7.10: *Average speedup achieved per distribution method.*

Since the difference between the distribution methods is rather small, a third distribution is added to the co-simulation. The distribution aims for creating a worst-case distribution, in which all federates in a room are executed on a different node to maximise the communication required between the nodes. It is therefor expected that this distribution performs worse compared to the grouped and automatic distributions. This distribution is also generated by the Lighting DSL and shall be called the *separated* distribution. Additionally, this co-simulation is executed and distributed over 14 nodes. By distributing over 14 nodes, for the grouped distribution every node simulates all federates for one area. Since this distribution is considered to be optimal for load balancing the co-simulation, this grouped distribution is expected to outperform the other distributions. The speedup results of this experiment are shown in Figure 7.11. The separated distri-

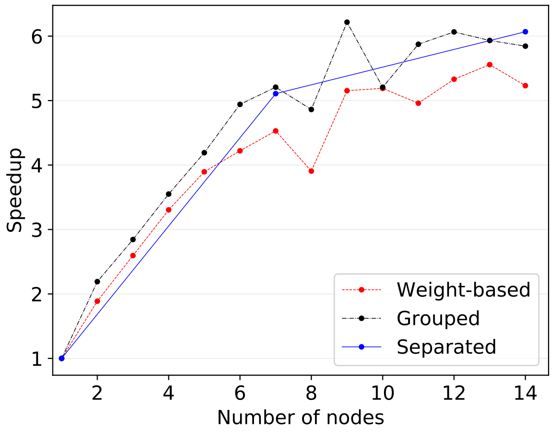


Figure 7.11: *Average speedup achieved per distribution method when distributing over up to 14 nodes.*

bution was only executed for distributions over 1, 7 and 14 nodes to speed up the benchmarking process and because these measurements provide sufficient results to compare the distribution method with the others. The results show that both the grouped and separated distributions achieve the highest speedup; the separated distribution even scales best for 14 nodes. Up to distributing over 7 nodes, the three methods have achieved very similar speedups. Manually specifying a distribution over the number of nodes does not improve the speedup by much, henceforth we will use the weight-based distribution method by default when running a distributed co-simulation. Section 7.5.6 gives a possible explanation for the results and proposes a method to improve the speedup when using the grouped distribution.

7.5.5 Scalability limit

The results displayed in Figure 7.11 show that the speedup per node that was added decreases from 7 nodes up. For this, there are two possible causes: either the network overhead becomes too large or the co-simulation is simply not large enough to make efficient use of more than 7 nodes. To find out which of these two causes is more likely, we create a larger co-simulation of a lighting system.

The larger lighting system – henceforth called *Floor* – is roughly twice as big as the *MediumFloor* and is displayed in Figure 7.12. The lighting system consists of 10 rooms, which are connected by 2 corridors. In total, there are 85 occupancy sensors and 85 lights, which are controlled by 12 lighting controllers. With a logger and an actor federate added to the co-simulation, the federation consists of 184

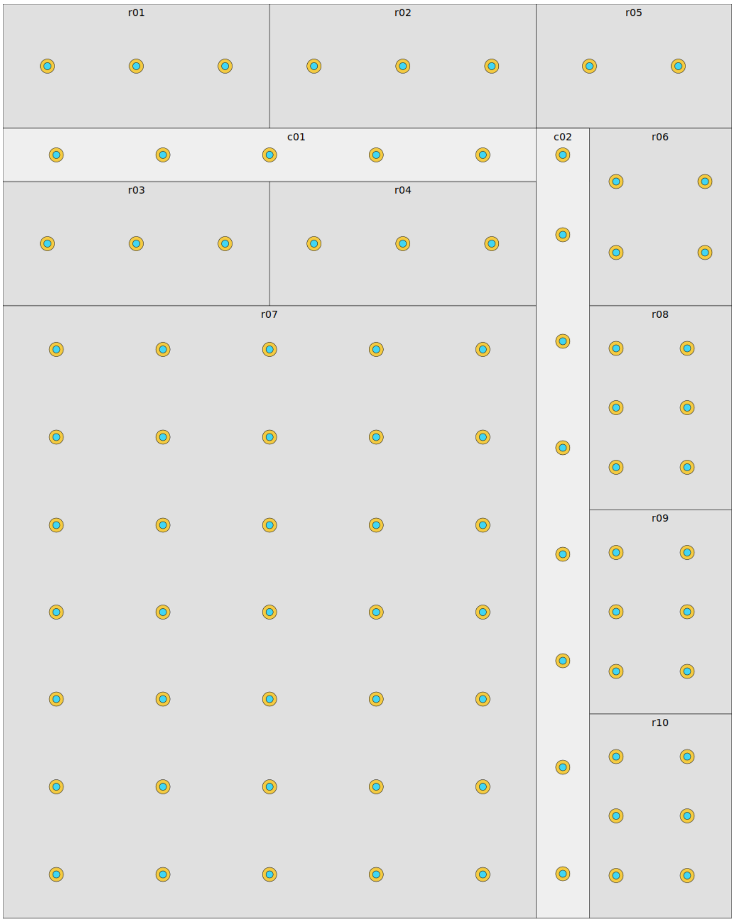


Figure 7.12: *Visual representation of a larger lighting system called Floor.*

participating federates. The Lighting DSL specification of the *Floor* system is available in the online experiments repository.

The co-simulation is distributed over up to 16 nodes using the automatic weight-based distribution method that is provided by CoHLA. Similar to earlier experiments, the actor mimics a path through the building by playing a scenario. Figure 7.13 displays the speedups achieved by distributing the co-simulation. The speedup is calculated by comparing the simulation time for a distribution with the simulation time of the slowest node (maximum speedup), fastest node (minimum speedup) and the average simulation time of the participating nodes (average speedup).

The results show that the distributed co-simulation using HLA and CoHLA scales quite well. The benefit of adding nodes decreases when the number of nodes increases. Considering a speedup of 8 to 10 when distributing the co-simulation of the system over 10 nodes, however, is rather good. Figure 7.14 shows the average speedups that were achieved for the MediumFloor system (90 federates) and Floor

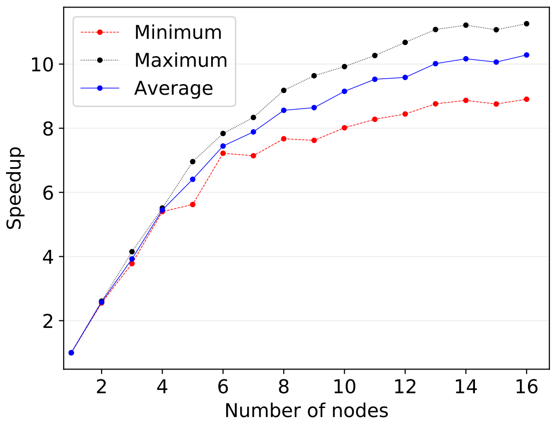


Figure 7.13: *Speedups achieved using distributed co-simulation for the Floor system compared to the fastest node (minimum), slowest node (maximum) or average simulation time of the nodes (average).*

system (184 federates). The figure shows that the speedup increases faster for a larger co-simulation, which means that the flattening during the first experiments was caused by the number of federates being too small. Consequently, not all nodes are fully under load during the co-simulation execution, reducing the speedup of the distribution. Larger co-simulations can therefor be efficiently distributed over a set of nodes.

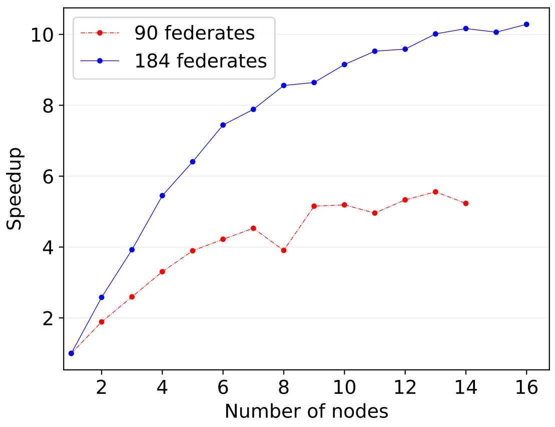


Figure 7.14: *Average speedups of the MediumFloor and Floor systems when executed in a distributed fashion.*

7.5.6 Optimising distribution performance

Even though the co-simulation scales quite well, the earlier experiments also show a possible optimisation. Grouping federates that interact with each other on the same node did not help as expected, which is caused by the publish-subscribe mechanism in HLA. In HLA, a federate instance subscribes to attributes from a specific type of federate class. Consequently, all updates of that attribute from all instances of that federate class are published to the subscribed federate instance. In the lighting system, the controller subscribes to the *occupied* attribute of the occupancy sensor class. Therefore, all updates of this attribute of all occupancy sensors in the system are shared with the controller, while the controller only intended on getting the updated values from the occupancy sensors that are within the same area. The controller then filters out the attribute values from the occupancy sensors that it is actually interested in. As a result of the publish-subscribe mechanism being specified on a class level, the RTIs are not capable of keeping attributes within a specific zone or node. Hence, quite some network traffic is generated to transmit updates that will not be used.

To make this attribute sharing more efficient, the HLA standard includes the interface specification of the HLA Data Distribution Management (HLA-DDM). HLA-DDM allows the creation of regions to limit the scope in which attributes are shared between federates by grouping the federates in regions. Although this seems to solve the problem, using HLA-DDM is rather difficult according to [49]. It is also mentioned that all major RTIs provide support for HLA-DDM, but we have found that support for this part of the standard is missing in some of the open source RTI implementations. OpenRTI, for instance, does not yet support HLA-DDM.

To enable similar behaviour for the federations running on RTIs not supporting HLA-DDM, each federate class may be split up for the number of regions in the co-simulation. For the lighting system, every room has its own federate class for a light instead of all rooms having instances of the same federate class. To illustrate this approach, Table 7.7 displays an example of the traditional instance and class configuration (*SingleClass*) versus the proposed one (*MultiClass*) for a small lighting system.

For the SingleClass setup of the small lighting system, there are 4 federate classes in total for 13 federate instances. Using the proposed MultiClass approach, the same system still has 13 federate instances, but now is based on 9 different federate classes. The MultiClass approach allows the light controllers to only subscribe for attributes from sensors that are also just found in this room. This is similar for the other classes. This approach is expected to reduce the load for the RTI as well as for the network by a significant amount. This is particularly the case when the RTI is also capable of keeping those updates locally, so that these do not have to be transmitted to and from the master RTI in the distribution. This is supported by OpenRTI. The Lighting DSL is extended to automatically generate all these separate classes for each room to minimise the effort required to define these all manually. Since the source code for an executable is generated by CoHLA for every federate class, this increases the number of executables that needs to be compiled, thus increasing its compilation time. This is not considered to be an issue as the sources only need to be compiled once. Note that these

Area	Instance	Class	
		SingleClass	**MultiClass**
Room 1	Controller	LightController	LightControllerRoom1
	Light 1	Light	LightRoom1
	Light 2		
	Sensor 1	Sensor	SensorRoom1
	Sensor 2		
Room 2	Controller	LightController	LightControllerRoom2
	Light 1	Light	LightRoom2
	Light 2		
	Sensor 1	Sensor	SensorRoom2
	Sensor 2		
Corridor	Controller	CorridorController	CorridorControllerCorridor
	Light 1	Light	LightCorridor
	Sensor 1	Sensor	SensorCorridor

Table 7.7: *Traditional instance and class configuration versus the proposed configuration for a small lighting system.*

classes still simulate the same models as before.

A new lighting system – called *ManyRooms* – is designed to potentially exploit the MultiClass approach. To limit the number of federates that need to connect to more than 1 area, the system does not contain any corridors. The system is displayed in Figure 7.15 and consists of 16 rooms that do not have interaction with each other, each having four occupancy sensors and four lights and one lighting controller. Similar to previous experiments, an actor is added that mimics a scenario in the building. The Lighting DSL specification of the *ManyRooms* system can be found in the online experiments repository.

Three different approaches are tested, each of them resulting in a different number of federate classes and sometimes even a different number of federates in the co-simulation. The approaches are briefly explained below. Table 7.8 shows the numbers of classes and instances for each of these approaches.

- The SingleClass approach does not generate separate classes for each room. All previously described experiments were conducted using this approach.
- The proposed MultiClass approach generates separate classes for each room. Only one actor class is used, which introduces one federate that publishes attributes that are subscribed to from multiple regions.
- The third approach is similar to the MultiClass approach as it generates separate classes for all areas in the system. The difference is that this approach also generates separate actor classes for the actors to eliminate all federates that might communicate with different regions than their own. Using this approach, all actors play the same scenario.

Figure 7.16 shows the average speedup results for the three federate class setups. With a low number of nodes, these approaches scale rather equally, but

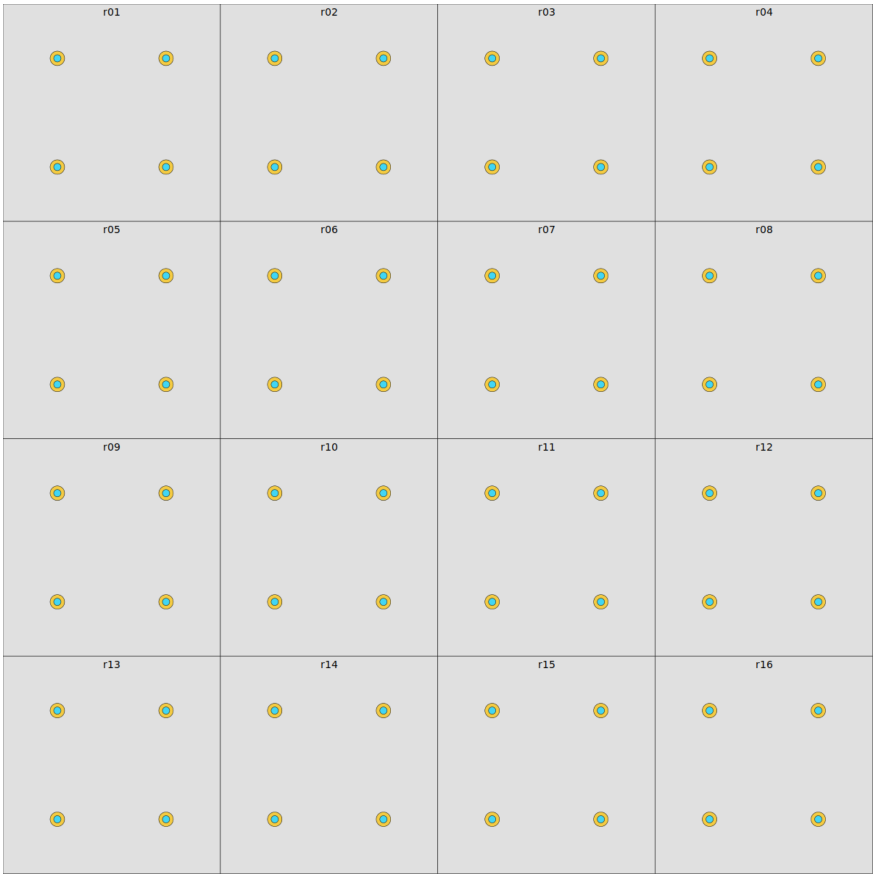


Figure 7.15: *Visual representation of the ManyRooms lighting system.*

from 8 nodes up they start to differ. The difference between the SingleClass and MultiClass approach with only one single actor is relatively small and constant. When separate actor classes are used too, the speedup boosts to 13 when running on 16 nodes. The reason for this is the fact that every node is running one single room without having interaction with federates on other nodes. Consequently, every node is basically running its own separate simulation of a room, while synchronising only time with each other. The results show that this method can be used to increase the speed of large distributed co-simulations when running on an RTI that does not support HLA-DDM.

However, not all systems are suitable for using this MultiClass approach. Suitable systems are those that can be separated into rather isolated subsystems where these subsystems are also rather similar. Systems where basically all instances interact with each other are less suitable. From the results it can be seen that the performance benefit is relatively small when there is at least one federate in the co-simulation that synchronises data to federates on other nodes. The approach also requires a rather statically connected system, while HLA also supports dy-

Approach	**Classes**				**Instances**			
	S	**L**	**C**	**A**	**S**	**L**	**C**	**A**
SingleClass	1	1	1	1	64	64	16	1
	4				145			
MultiClass – 1 actor	16	16	16	1	64	64	16	1
	49				145			
MultiClass – 16 actors	16	16	16	16	64	64	16	16
	64				160			

Table 7.8: *Number of federate classes and instances for each of the approaches. Numbers are categorised by federate: sensors (S), lights (L), controllers (C) and actors (A).*

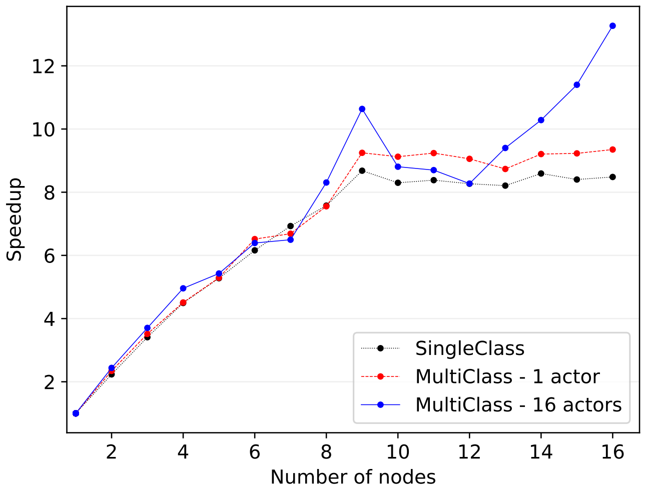


Figure 7.16: *Average speedup achieved for each of the federate class configurations for the ManyRooms co-simulation.*

namically changing systems during the simulation.

Additionally, using multiple classes also increases the absolute execution time for the co-simulation. The MultiClass approach using just one actor increased the average execution time on a single node by 17% compared to the SingleClass approach. When using multiple actors as well, the average execution time even increased by 48%. Consequently, the absolute execution time of the MultiClass approach with 16 actors was very similar to the SingleClass approach when distributed over 16 nodes, even though the speedup for the MultiClass approach with multiple actors was better. Since there are more different executables running concurrently using the MultiClass approach, processor caching may be less efficient, resulting in slower execution times. The addition of 15 actors to the federation is likely to impact the performance as well.

7.6 Conclusion

This chapter described a number of experiments to analyse the scalability of HLA and CoHLA. The co-simulation of a large smart lighting system was used to analyse the performance speedup of distributed co-simulation execution in the cloud. A new DSL was developed to quickly generate CoHLA specifications for the lighting system. Finally, a method is proposed to improve the speedup of the distributed co-simulation execution.

The experiments have shown that the distributed execution of a co-simulation can significantly shorten the execution time of the simulation. No limit on the number of nodes was observed, as the communication overhead between the nodes does not appear to be a blocking factor. Different distribution methods were compared to each other. Experiments show that grouped distribution of the federates over the computation nodes achieves similar speedups compared to a weight-based distribution.

An approach to improve the simulation speed for the grouped distribution is proposed. The approach uses regions to limit the scope of updates sent by the different federates. Results show that this approach scales better, but only for a very limited set of systems: only co-simulations that consist of isolated groups of federates benefit from the approach. Even though the approach increases the speedup, it may decrease the simulation speed itself by adding many different federates to the co-simulation.

For a particular class of systems – such as IoT systems, consisting of many similar instances – the use of a separate DSL that is specifically designed for this class of systems simplifies the construction of a co-simulation by generating its CoHLA definition. The development of such a DSL required a few days of development time using the technologies that were used for the development of CoHLA. It reduced the time required to specify the CoHLA co-simulation from a couple of hours to about 15 minutes for our lighting system examples. Whether or not the development of such a DSL is worth the investment greatly depends on the number of co-simulations to construct and their sizes. Using a separate DSL for the specification of a system could be more intuitive for the system architect and is less error prone. It also allows for easy extension, such as distribution generation and the generation of a visual representation.

Reflection on requirements

5. *The co-simulation can be executed in a distributed manner to provide scalability.* The experiments have shown that distribution of a co-simulation using CoHLA and HLA is a relatively easy method to speed up the simulation execution. This approach scaled well for our experiments. The use of the automated execution script for conducting the experiments proved that it is possible to abstract from whether the co-simulation is executed in a distributed environment or not. This might be implemented in the run script in future work.

CHAPTER 8

Conclusion

This chapter concludes the dissertation by first providing an overview of the CoHLA framework in Section 8.1. Section 8.2 provides a brief overview of the contributions of this dissertation and reflects on the requirements that were listed in Section 1.3. The section also summarises a number of limitations of the CoHLA framework. Potential future work is discussed in Section 8.3.

8.1 CoHLA

This dissertation presents CoHLA, a DSL that was developed to rapidly construct a co-simulation of a set of simulations. From a co-simulation specification, C++ code is generated for use with OpenRTI, an open source implementation of the HLA standard. The DSL was developed using Xtext and Xtend. Since the features of CoHLA are explained on different places in the dissertation, these are summarised below.

- **FMI**
  Models complying to the FMI standard and exported to an FMU can be simulated by the CoHLA co-simulation.

- **POOSL**
  POOSL models for discrete-time model simulation can be incorporated in the CoHLA co-simulation. The models are simulated by the Rotalumis simulator.

- **Collision simulator**
  3D drawings of components of the system can be used to visualise the system being simulated. The drawings can also be used for collision detection using

the collision simulator. For each of the 3D components, transformations can be provided that specify how the component moves according to one of the input states. These input states must be provided by another federate, i.e. another simulator in the co-simulation.

- **Logging**
  A logger federate can be added to the co-simulation to output a CSV file containing the attribute values it is subscribed to. The attributes that should be logged can be specified using a regular CoHLA connection to the logger.

- **Metric collection**
  Similar to the logging federate, a metric collector can be added to the federation. The metric collector is capable of measuring basic performance metrics of the simulated system and writing them to an output file when the simulation has ended.

- **Model parameters**
  Model parameters can be used to specify the value of an attribute or parameter upon initialisation. By changing these values, the same model can be reused to simulate different configurations of the same model.

- **Initialisation configuration**
  Initialisation configurations are sets of model parameters being assigned specific values. This allows the user to specify larger sets of model parameter values at once to easily reuse such configurations.

- **Situations**
  Using initialisation configurations and model parameters, different configurations of the same model class can be configured. To easily apply these configurations to a co-simulation, situations can be specified. A situation applies a set of configurations to specific instances of model simulations in the co-simulation. Situations can be combined to enable reuse.

- **Scenarios**
  A scenario allows the user to specify attribute values at specific points in time during the simulation. Such a scenario is capable of mimicking user interaction with the system or other external events. Another type of event that can be specified in a scenario is the transmission of a byte sequence through a socket.

- **Fault scenarios**
  Fault scenarios are very similar to regular scenarios as they influence the simulation based on time triggers. Events in fault scenarios allow the user to disconnect specific attributes or fix their value for a specified period. The goal of using fault scenarios is to test the fault tolerance of the system. Four basic types of faults are implemented by the CoHLA framework.

- **Design space exploration**
  Design space exploration (DSE) is used to run a co-simulation of a system for a number of times with alternating parameters. It allows the user to

specify a design space that should be co-simulated. A metric collector is used to retrieve useful information from the co-simulation executions. Every co-simulation execution produces its own logs and metric result files, and the metrics of all simulation executions are bundled into one single file. A DSE configuration may also include a scenario or fault scenario for the execution of the simulations.

- **Distributed simulation**
  To run the co-simulation in a distributed fashion, each federate may be assigned a weight. This allows the federates to be distributed over all participating computation nodes based on their weights. Optionally, a distribution may be specified using the CoHLA language that assigns specific federates to specific computation nodes.

For each co-simulation specification in CoHLA, source code for the model wrappers is generated together with configuration files that specify all aforementioned configurations, such as scenarios. A run script is also generated to easily build and start the co-simulation or execute a design space exploration. This allows the user to rapidly construct and run a co-simulation from a set of simulations, including all features listed above.

8.2 Reflection

This section starts with a brief overview of the contents of the dissertation in Section 8.2.1. Section 8.2.2 reflects on the requirements that were given in Section 1.3. Finally, Section 8.2.3 describes a number of limitations to the introduced approach.

8.2.1 Overview

The CoHLA co-simulation specification allows changes in the models or their interfaces to be adapted easily, which makes the co-simulation flexible during the system design. A wide range of modelling tools is compatible for use with CoHLA, as it allows the incorporation of models adhering the FMI standard.

To obtain confidence in the results of the generated co-simulation, its timing behaviour was analysed. The impact of the CoHLA framework on the co-simulation results was checked by comparing the co-simulations with an integrated system simulation by a single simulator and with the INTO-CPS co-simulation framework. The CoHLA framework and the code it generates do not have a noticeable impact on the results, so the quality of the results depends on the quality of the models being simulated. The CoHLA co-simulation results have therefore showed to be trustworthy.

The approach was applied on different case studies, which resulted in the CoHLA language being extended by adding features such as design space exploration, logging and fault injection. Since the language was developed using Xtext and Xtend, it is highly flexible and easy to extend. The addition of support for new tools or features was generally completed in less than a day. Due to the fact that the libraries are written in C++ and the dependencies are available for all platforms, CoHLA works on Windows, Linux and Mac.

The scalability of the approach was shown by applying it to a case study consisting of more than 100 simulation models. Distributed execution of the co-simulation proved to be possible and increased the simulation speed of a large co-simulation.

8.2.2 Requirements

Table 8.1 reflects on the requirements that were stated in Section 1.3. For every requirement, the chapters that describe the fulfillment of the requirement are listed in column 'C'. A brief description is given as well.

#	Requirement	C
1.	*A co-simulation of simulation models of different disciplines can be constructed fast (20 models within 1 day).*	4, 6
The case studies that were conducted for this research, except for the lighting systems, all consist of less than 20 models. Co-simulation for these systems could easily be specified and constructed within a day. A separate DSL was developed for the lighting system to be able to also construct a co-simulation for such systems within a day. However, the number of cases to which CoHLA was applied is rather small and insufficient to draw a solid conclusion on this requirement. To be able to draw a more solid conclusion, identical systems should be designed by different teams following different approaches. A number of teams will use CoHLA to support the design and a number of teams will follow a different approach or framework. By comparing the development process of the teams using the different frameworks, the experiment should show whether this requirement is met.		
2.	*Changes in either the models or the interfaces connecting these models can be adapted quickly using the approach (5 models within 1 hour).*	4, 6
Similar to the first requirement, this requirement was met for the case studies that were conducted during this research. The experiment described in Section 8.3 to further analyse the use of CoHLA could also be used to show whether this requirement is met.		
3.	*Simulation models from multiple tools are supported: at least 10 modelling tools.*	3
By supporting models adhering the FMI standard, all modelling tools that are capable of exporting their models to FMUs are supported. Currently, over 40 tools provide support for exporting their models into FMUs, so this requirement is met. In addition to the tools supporting the FMI standard, CoHLA also provides support for POOSL models.		

#	**Requirement**	**C**
4.	*The approach is easily extendable to support new tools.*	3, 4
	The use of Xtext and Xtend makes the implementation of CoHLA very flexible. For example, adding the collision simulator to CoHLA was finished within a day. The amount of development work to be done in order to add support for a new type of simulator is limited.	
5.	*The co-simulation can be executed in a distributed manner to provide scalability.*	7
	The HLA standard already supports distributed execution of a co-simulation. A number of experiments have been conducted on the co-simulation of more than 100 simulations using a CoHLA generated co-simulation. The co-simulations of these experiments scaled well when executed in a distributed manner. Since only one type of system was experimented with, more experiments with different types of systems should give more insight in scalability aspects.	
6.	*The simulation results are trustworthy.*	5, 6
	The trustworthiness of CoHLA co-simulations is analysed by conducting a number of experiments. One experiment analysed the timing behaviour of the co-simulation and result comparisons have been done with the INTO-CPS co-simulation framework and the 20-sim simulator. Although these experiments indicate that the co-simulation results of FMUs are trustworthy, more experiments are necessary to draw a conclusion on this requirement.	
7.	*The approach has logging capabilities for analysis afterwards.*	4
	Basic attribute value logging as well as logging based on a collection of basic metrics from a co-simulation execution are supported by CoHLA.	
8.	*The approach has support for automated design space exploration.*	4, 6
	A basic implementation of automated co-simulation execution to support DSE was added. The functionality is limited to the automated execution of a predefined design space and does not support automated parameter optimisation.	
9.	*The approach has support for fault injection.*	4
	A number of faults regarding the model's attribute transmission has been implemented in CoHLA. More types of faults could be added rather easily.	
10.	*The framework is easy to maintain and extendable.*	4
	The use of Xtext and Xtend make the implementation of CoHLA very flexible. Similar remarks as described in the reflection on requirement 4 apply here.	
11.	*The framework is documented properly.*	1
	An installation manual as well as a user manual were written for the CoHLA framework and are available online[1] and in the repositories.	

#	Requirement	C
12.	*The framework runs on Windows, Linux and Mac.*	3, 4, 6
Since the CoHLA framework is built with and depends on platform-independent technologies it can be used on all major platforms. The majority of case studies and development was done on Linux and a number of small experiment have been conducted on Windows and Mac.		

Table 8.1: *Reflection on the requirements stated in Section 1.3. Column **C** lists the chapters that discuss the fulfillment of the requirement. A short description is provided for each requirement.*

For the conducted case studies, the CoHLA framework has shown to be a flexible and fast method to create a co-simulation from a set of models. Built-in features allow the user to quickly analyse the system behaviour given certain conditions, which supports making design decisions in an early stage of the development. Changes are incorporated quickly, resulting in a low overhead for maintaining a co-simulation environment, allowing all disciplines to work in their own pace and use their own tools, while still being able to get insight in the working of all components together.

8.2.3 Limitations

Even though the HLA standard supports the use of both attribute synchronisation and message-based communication, the primary focus of CoHLA is the synchronisation of attributes. Especially for POOSL models, the addition of message-based communication to CoHLA would provide a more intuitive way of communicating, as this is also the internal communication method between processes in the POOSL models. The CoHLA framework could be extended to support such a communication between federates.

While the POOSL models are variable time step models, FMUs require a fixed step size to be specified in CoHLA. The CoHLA library should be extended to support a variable step size for FMUs as well. This would allow more complex discrete-time VDM-RT models to be simulated as well, without requiring a predefined step size that might result in incompatible steps being taken during simulation, where the model simulation might miss out on certain information from the model.

8.3 Future work

To analyse whether the approach meets the first requirements regarding the speed of constructing and changing a co-simulation additional experiments should be conducted. A suggestion is to formulate an assignment for students, where the students should design a system in groups. Half of the groups use CoHLA to support their design and half of the groups follow a different approach. The design

[1]`https://cohla.nl/docs/`

process could be analysed for all groups, after which these approaches could be compared with each other.

The Xtext framework provides tools to implement validation methods for a DSL that is developed. CoHLA has basic validation that verifies the types and direction of attribute connections as well as a number of checks for invalid properties for federate classes. For example, a logging federate does not need to have a default model specified. To avoid simple mistakes regarding the specification of a co-simulation or other configurations using CoHLA, more validation rules need to be implemented.

CoHLA provides support for basic fault scenarios. Since repeatability of co-simulation executions was our primary focus for DSE, these fault scenarios do not include more faults of a probabilistic nature. However, this type of faults might be useful for other types of experiments and to analyse long-term behaviour of a system. As such experiments were not conducted in this research, this type of fault was not implemented in CoHLA. To be able to perform such analyses, probabilistic faults may be added to CoHLA.

Basic support for DSE is implemented by CoHLA. This requires all parameters for the design space to be specified. The CoHLA run script is then able to automatically run all configurations and collect the results. This approach is less suitable for very large design spaces, as these cannot be exhaustively compared. It may be useful to extend CoHLA with searching strategies such as genetic algorithms to optimise the model's parameters. These strategies could be tuned by the user and allow DSE for very large design spaces.

As mentioned in Section 3.3, the POOSL library of CoHLA for the new POOSL version is not yet finished. Both the interface process for POOSL models and the C++ library for the CoHLA framework need some work to finish the transition from using the Rotalumis debugging socket to the use of external ports.

The run script that is generated from a CoHLA specification can be started and controlled primarily from the command line. A start has been made to also generate a very basic graphical user interface (GUI) with it, so that the user would be able to change model parameters using a more intuitive interface. Additionally, the user could start and stop the co-simulation execution from the GUI by simply pressing a button. The model parameters configured via the GUI would then be used for the co-simulation execution. This approach provides a better overview to the user on the processes being executed and their configurations. During the coarse of the research, maintaining the GUI lost its priority in favour of adding functionality to the framework. To improve the user experience for managing the co-simulation, the GUI should be updated or rewritten.

A script that starts a distributed co-simulation by connecting to all participating nodes is described in Section 7.5.2. The script proves that it is possible to do this automatically over SSH connections, but it was developed to run a number of specified benchmarks using different distributions. To improve the support for executing distributed co-simulations, the CoHLA-generated run script should also be able to start co-simulations in a distributed manner without manually starting the run script on every node. This would imply a number of changes to the code generated by CoHLA, as it would for instance require dependency checking on remote computation nodes to ensure that OpenRTI and the CoHLA libraries are properly installed. Implementing this would fade the difference between running a co-simulation locally and running it in a distributed fashion, thus making it more usable.

APPENDIX A

List of Abbreviations

Abbreviation	**Meaning**	**Introduced**
DSE	Design Space Exploration	Section 4.4.10
FMI	Functional Mock-up Interface	Section 3.1
FMU	Functional Mock-up Unit	Section 3.1
FOM	Federation Object Model	Section 3.2
HIL	Hardware-in-the-loop	Section 1.2
HLA	High Level Architecture	Section 3.2.2
LT	Logical Time	Section 5.1
NMR	NextMessageRequest	Section 3.2.2
RO	Receive Order	Section 3.2.2
RTI	Run-Time Infrastructure	Section 3.2
SIL	Software-in-the-loop	Section 1.2
ST	Simulation Time	Section 5.1
TAG	TimeAdvanceGrant	Section 3.2.2
TAR	TimeAdvanceRequest	Section 3.2.2
TS	Timestamp	Section 5.2
TSO	Timestamped Order	Section 3.2.2

Table A.1: *List of abbreviations.*

APPENDIX B

CoHLA grammar

```
/*
 * Copyright (c) Thomas Nägele and contributors. All rights reserved.
 * Licensed under the MIT license. See LICENSE file in the project root for
     details.
*/

grammar nl.ru.sws.cohla.CoHLA with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate coHLA "http://www.ru.nl/sws/cohla/CoHLA"

Model:
  (
    imports+=Import*
    environment=Environment?
    federateObjects+=FederateObject*
    interfaces+=Interface*
    configurations+=ModelConfiguration*
    federations+=Federation*
  );

Import:
  'import' importURI=STRING
;

// ===== HLA Environment =====

Environment:
  'Environment' '{'
    rti=HLAImplementation
    ('SourceDirectory' srcDir=STRING)?
    ('PrintLevel' printLevel=PrintLevel)?
    (publishOnlyChanges ?= 'PublishOnlyChanges')?
  '}'
;

HLAImplementation:
  'RTI' '{'
    implementation=HLAImp
    'Libraries' libRoot=STRING
    ('Dependencies' depRoot=STRING)?
  '}'
;

enum HLAImp:
    pitchRti="PitchRTI"
  | portico="Portico"
  | openRti="OpenRTI"
;

enum PrintLevel:
    state="State"
  | time="Time"
  | none="None"
;

// ===== FederateObject =====

FederateObject:
  'FederateClass' name=ID '{'
    'Type' type=FederateType
    ('{' config=SimulatorConfig '}')?
    ('Attributes' '{' attributes+=Attribute+ '}')?
    ('Parameters' '{' parameters+=Parameter+ '}')?
    ('TimePolicy' timePolicy=TimePolicy)?
    ('DefaultModel' defaultModel+=STRING+)?
    ('AdvanceType' advanceType=AdvanceType)?
    ('DefaultStepSize' defaultStepSize=P_FLOAT)?
    ('DefaultLookahead' defaultLookahead=P_FLOAT)?
    ('SimulationWeight' simulationWeight=INT)?
  '}'
;

enum TimePolicy:
    both="RegulatedAndConstrained"
  | regulated="Regulated"
  | constrained="Constrained"
  | none="None"
;

Process:
  name=ID 'in' path=STRING
;

Parameter:
  dataType=DataType name=ID alias=STRING ('in' process=[Process])?
;

SimulatorConfig:
  POOSLConfig | LoggerConfig
;

POOSLConfig:
  'Processes' '{'
    processes+=Process+
  '}'
;

LoggerConfig:
  'DefaultMeasureTime' defaultMeasureTime=P_FLOAT
;

enum FederateType:
    poosl="POOSL"
  | fmu="FMU"
  | csv="CSV-logger"
  | colsim="BulletCollision"
  | none="None"
;
```

```

enum AdvanceType:
    time="TimeAdvanceRequest"
  | message="NextMessageRequest"
;

// ===== Attribute =====

Attribute:
  sharingType=SharingType dataType=DataType (collision?='[Collision]')? ('['
       multiInputOperator=MultiInputOperator ']')? name=ID (alias=
       AliasProperty | processProperty=ProcessProperty)? ('{'
    updateTypeProperty=UpdateTypeProperty?
    updateConditionProperty=UpdateConditionProperty?
    transportationProperty=TransportationProperty?
    orderProperty=OrderProperty?
  '}')?
;

enum SharingType:
    neither="Void"
  | publish="Output"
  | subscribe="Input"
  | publishSubscribe="InOutput"
;

enum DataType:
    integer="Integer"
  | long="Long"
  | string="String"
  | real="Real"
  | bool="Boolean"
;

ProcessProperty:
  'in' process=[Process] 'as' attributeName=STRING
;

AliasProperty:
  'as' alias=STRING
;

UpdateTypeProperty:
  'UpdateTypeProperty' updateType=UpdateType
;

UpdateConditionProperty:
  'UpdateCondition' updateCondition=UpdateCondition
;

enum UpdateCondition:
  onChange="OnChange"
;

TransportationProperty:
  'Transportation' transportation=TransportationType
;

OrderProperty:
  'Order' order=OrderType
;

// ===== ConnectionSet =====

Interface:
  'ConnectionSet' 'between' class1=[FederateObject] 'and' class2=[
       FederateObject] '{'
    connections+=InterfaceConnection+
  '}'
;
```

```

InterfaceConnection:
  '{'
    in=ClassAttributeReference '<-' out=ClassAttributeReference
  '}'
;

ClassAttributeReference:
  objectClass=[FederateObject] '.' attribute=[Attribute]
;

// ===== Federation =====

Federation:
  'Federation' name=ID '{'
    ('Instances' '{'
      instances+=FederateInstance+
    '}')?
    ('Connections' '{'
      connections+=Connection+
    '}')?
    situations+=Situation*
    scenarios+=Scenario*
    faultScenarios+=FaultScenario*
    dses+=DSEConfig*
    metricSets+=MetricSet*
    distributions+=Distribution*
  '}'
;

FederateInstance:
  name=ID ':' federate=[FederateObject]
;

Connection:
    AttributeConnection
  | LoggerConnection
  | ObjectConnection
;

AttributeConnection:
  '{'
    inAttribute=AttributeReference '<-' outAttributes+=AttributeReference
  '}'
;

LoggerConnection:
  '{'
    logger=[FederateInstance] '<-' outAttributes+=AttributeReference (','
        outAttributes+=AttributeReference)*
  '}'
;

ObjectConnection:
  '{'
    instance1=[FederateInstance] '-' instance2=[FederateInstance]
  '}'
;

AttributeReference:
  instance=[FederateInstance] '.' attribute=[Attribute]
;

// ===== Situation =====

Situation:
  'Situation' name=ID ('extends' extends=[Situation])? '{'
    elements+=SituationElement+
  '}'
;
```

```

SituationElement:
  AttributeInitialiser | ApplyModelConfiguration
;

AttributeInitialiser:
  'Init' reference=InstanceParameterReference 'as' value=STRING
;

InstanceParameterReference:
  instance=[FederateInstance] '.' attribute=[Parameter]
;

ApplyModelConfiguration:
  'Apply' configuration=[ModelConfiguration] 'to' instance=[FederateInstance]
;

// ===== Configuration =====

ModelConfiguration:
  'Configuration' name=ID 'for' federateObject=[FederateObject] '{'
    initAttributes+=AttributeConfigurator+
   '}'
;

AttributeConfigurator:
  reference=ParameterReference '=' value=STRING
;

ParameterReference:
  attribute=[Parameter]
;

// ===== Scenarios =====

Scenario:
  'Scenario' name=ID '{'
    settings=ScenarioSettings?
    events+=Event+
  '}'
;

ScenarioSettings:
  'AutoStop:' (autoStopTime = P_FLOAT | noAutoStop ?= 'no')
  sockets+=ScenarioSocket*
;

ScenarioSocket:
  'Socket' name=ID 'for' instance=[FederateInstance] 'on' host=STRING ':'
       port=INT
;

Event:
  time=P_FLOAT ':' action=Action
;

Action:
  ActionAssign | ActionSocket
;

ActionAssign:
  attribute=AttributeReference '=' value=STRING
;

ActionSocket:
  socket=[ScenarioSocket] '<-' bytes+=HexByte (',' bytes+=HexByte)*
;

// ===== Faults =====

FaultScenario:
  'FaultScenario' name=ID '{'
    faults+=Fault+
  '}'
;

Fault:
  TimedAbsoluteFault | TimedConnectionFault | TimedOffsetFault |
      AccuracyFault
;

AccuracyFault:
  'Variance' 'for' attribute=AttributeReference '=' value=P_FLOAT
;

TimedAbsoluteFault:
  range=Range 'set' attribute=AttributeReference '=' value=STRING
;

TimedConnectionFault:
  range=Range 'disconnect' attribute=AttributeReference
;

TimedOffsetFault:
  range=Range 'offset' attribute=AttributeReference '=' value=C_FLOAT
;

Range:
    ('On' time=P_FLOAT)
  | ('From' startTime=P_FLOAT ('to' endTime=P_FLOAT)?)
;

// ===== Domain Space Exploration =====

DSEConfig:
  'DSE' name=ID '{'
    ('SweepMode' sweepMode=SweepMode)?
    ('Scenario' scenario=[Scenario])?
    ('Faults' faultScenario=[FaultScenario])?
    situations+=DSESituationConfig*
    configurations+=DSEFederateConfig*
    attributes+=DSEAttrInit*
  '}'
;

DSESituationConfig:
  'Situations' ':' situations+=[Situation] (',' situations+=[Situation])*
;

DSEFederateConfig:
  'Configurations' 'for' instance=[FederateInstance] ':' configurations+=[
      ModelConfiguration] (',' configurations+=[ModelConfiguration])*
;

DSEAttrInit:
  'Set' attr=InstanceParameterReference ':' value=MultiValue
;

enum SweepMode:
    independent="Independent"
  | linked="Linked"
;

MultiValue:
  value=(P_FLOAT | C_FLOAT | C_INT | STRING) | values+= (P_FLOAT | C_FLOAT |
      C_INT | STRING) (',' values+=(P_FLOAT | C_FLOAT | C_INT | STRING))+
;

// ===== Metrics =====

MetricSet:
  'MetricSet' name=ID '{'
    'MeasureTime:' measureTime=P_FLOAT
    metrics+=Metric+
  '}'
;

Metric:
  'Metric' name=ID 'as' (metricEV=MetricEndValue | metricErr=MetricError |
       metricTimer=MetricTimer | metricMinMax=MetricMinMax)
;

MetricEndValue:
  absolute?='Absolute'? 'EndValue' attribute=AttributeReference relativeTo=
       MetricRelativeTo?
;

MetricError:
  squared?='Squared'? 'Error' attribute=AttributeReference relativeTo=
       MetricRelativeTo
;

MetricTimer:
  'Timer' 'for' attribute=AttributeReference comparator=Comparator value=(
       P_FLOAT | C_FLOAT | C_INT | 'true' | 'false') ('(' isEndCondition?='
       EndCondition' ('after' delay=P_FLOAT)? ')')?
;

MetricMinMax:
  (min?='Minimum' | max?= 'Maximum') 'of' attribute=AttributeReference
;

MetricRelativeTo:
  'relative' 'to' ref=AttributeReference
;

// ===== Distributions =====

Distribution:
  'Distribution' name=ID 'over' nrOfSystems=INT 'systems' '{'
    systemSets+=SystemSet+
  '}'
;

SystemSet:
  systemId=INT ':' federates+=[FederateInstance]+
;

// ===== Types =====

HexByte:
  bytes+=HEXPAIR+
;

enum OrderType:
  receive="Receive"
  | timeStamp="TimeStamp"
;

enum TransportationType:
  reliable="Reliable"
  | bestEffort="BestEffort"
;

enum UpdateType:
  conditional="Conditional"
;

enum Comparator:
    lt="<"
  | le="<="
  | eq="=="
  | ge=">="
  | gt=">"
  | ne="!="
;

enum MultiInputOperator:
    none = "NONE"
  | and = "&&"
  | or = "||"
  | plus = "+"
  | product = "*"
;

terminal HEXPAIR returns ecore::EString:
  ('0x'|'x') HEXCHAR HEXCHAR
;

terminal HEXCHAR returns ecore::EChar:
  ('0'..'9'|'A'..'F'|'a'..'f')
;

terminal P_FLOAT returns ecore::EString:
  INT '.' ('0'..'9')+
;

terminal C_FLOAT returns ecore::EString:
  C_INT '.' ('0'..'9')+
;

terminal C_INT returns ecore::EString:
  ('-' | '+') INT
;
```

Listing B.1: *The CoHLA grammar.*

APPENDIX C

Lighting DSL grammar

```
grammar nl.ru.sws.dsl.lighting.Building with org.eclipse.xtext.common.
    Terminals

generate building "http://www.ru.nl/sws/dsl/lighting/Building"

Model:
  config = Configuration?
  buildings += Building+
;

Configuration:
  'Configuration' '{'
    '#Actors:' actors = INT
    'OccupancySensorRanges:' occupancySensorRanges += INT+
    (hasLogger ?= 'HasLogger')?
    ('MeasureTime:' measureTime = INT)?
    (hideSensorRange ?= 'HideSensorRange')?
    ('Distributions:' distributions+=INT+)?
    (separateClasses ?= 'SeparateClasses')?
    (separateActors ?= 'SeparateActors')?
    ('ModelDir:' modelDir=STRING)?
    (poosl ?= 'POOSL')?
  '}'
;

Building:
  'Building' name=ID '{'
    areas += Area+
    ('Scenarios' '{'
      scenarios += Scenario+
    '}')?
  '}'
;

Area:
  Room | Corridor
;

Room:
  'Room' (type=RoomType)? name=ID '{'
```

```
    ('Area:' draw = Draw)?
    ('Devices' '{'
      devices += Device+
    '}')?
  '}'
;

Corridor:
  'Corridor' name=ID '{'
    ('Area:' draw = Draw)?
    'Rooms:' connectedRooms += [Room]+
    ('Devices' '{'
      devices += Device+
    '}')?
  '}'
;

Device:
  Light | Sensor
;

Light:
  'Light' (type = LightType)? name=ID ('on' location=Coord)?
;

Sensor:
  'Sensor' (type = SensorType)? name=ID ('on' location=Coord)?
;

Draw:
  coords += Coord (coords += Coord)+
;

Coord:
  '(' x = INT ',' y = INT ')'
;

Scenario:
  'Scenario' name = ID 'for' 'Actor' actorId = INT '{'
    (
        ('ActorActivity:' actorActivity=ActorActivity)
      |
        (interpolate ?= 'Interpolate' interpolateStep = INT)
    )?
    startPosition = Coord steps += ScenarioStep+
  '}'
;

ScenarioStep:
  '[' delay = INT ']' position = Coord
;

ActorActivity:
  portion=INT '/' total=INT 's' 'from' startFrom=INT 's' 'to' endBefore=INT '
      s'
;

enum RoomType:
    basic = 'Basic'
  | office = 'Office'
  | officespace = 'OfficeSpace'
;

enum LightType:
    dimmable = 'Dimmable'
  | onoff = 'OnOff'
;

enum SensorType:
    occupancy = 'Occupancy'
```

```
;

enum OccupancySensorType:
    wideRange = 'WideRange'
  | medRange = 'MediumRange'
  | smallRange = 'SmallRange'
;
```

Listing C.1: *The Lighting DSL grammar.*

APPENDIX D

Lighting system: MediumFloor

```
Configuration {
  #Actors: 1
  OccupancySensorRanges: 150
  HasLogger
  MeasureTime: 1200
  Distributions: 2 3 4 5 6 7 8 9 10 11 12 13 14
  ModelDir: "../../models"
}

Building MediumFloor {
  Room r01 {
    Area: (0, 0) (500, 0) (500, 300) (0, 300)
    Devices {
      Light l1 on (125, 150)
      Sensor s1 on (125, 150)
      Light l2 on (375, 150)
      Sensor s2 on (375, 150)
    }
  }
  Room r02 {
    Area: (500, 0) (1000, 0) (1000, 300) (500, 300)
    Devices {
      Light l1 on (625, 150)
      Sensor s1 on (625, 150)
      Light l2 on (875, 150)
      Sensor s2 on (875, 150)
    }
  }
  Room r03 {
    Area: (0, 450) (500, 450) (500, 750) (0, 750)
    Devices {
      Light l1 on (125, 600)
      Sensor s1 on (125, 600)
      Light l2 on (375, 600)
      Sensor s2 on (375, 600)
    }
  }
  Room r04 {
    Area: (500, 450) (1000, 450) (1000, 750) (500, 750)
    Devices {
      Light l1 on (625, 600)
      Sensor s1 on (625, 600)
      Light l2 on (875, 600)
      Sensor s2 on (875, 600)
    }
  }
  Room r05 {
    Area: (0, 750) (500, 750) (500, 1050) (0, 1050)
    Devices {
      Light l1 on (125, 900)
      Sensor s1 on (125, 900)
      Light l2 on (375, 900)
      Sensor s2 on (375, 900)
    }
  }
  Room r06 {
    Area: (500, 750) (1000, 750) (1000, 1050) (500, 1050)
    Devices {
      Light l1 on (625, 900)
      Sensor s1 on (625, 900)
      Light l2 on (875, 900)
      Sensor s2 on (875, 900)
    }
  }
  Room r07 {
    Area: (0, 1200) (500, 1200) (500, 1500) (0, 1500)
    Devices {
      Light l1 on (125, 1350)
      Sensor s1 on (125, 1350)
      Light l2 on (375, 1350)
      Sensor s2 on (375, 1350)
    }
  }
  Room r08 {
    Area: (500, 1200) (1000, 1200) (1000, 1500) (500, 1500)
    Devices {
      Light l1 on (625, 1350)
      Sensor s1 on (625, 1350)
      Light l2 on (875, 1350)
      Sensor s2 on (875, 1350)
    }
  }
  Room r09 {
    Area: (1150, 0) (1450, 0) (1450, 500) (1150, 500)
    Devices {
      Light l1 on (1300, 125)
      Sensor s1 on (1300, 125)
      Light l2 on (1300, 375)
      Sensor s2 on (1300, 375)
    }
  }
  Room r10 {
    Area: (1150, 500) (1450, 500) (1450, 1000) (1150, 1000)
    Devices {
      Light l1 on (1300, 625)
      Sensor s1 on (1300, 625)
      Light l2 on (1300, 875)
      Sensor s2 on (1300, 875)
    }
  }
  Room r11 {
    Area: (1150, 1000) (1450, 1000) (1450, 1500) (1150, 1500)
    Devices {
      Light l1 on (1300, 1125)
      Sensor s1 on (1300, 1125)
      Light l2 on (1300, 1375)
      Sensor s2 on (1300, 1375)
    }
  }

```

```
  Corridor c01 {
    Area: (1000, 0) (1150, 0) (1150, 1500) (1000, 1500)
    Rooms: r02 r04 r06 r08 r09 r10 r11
    Devices {
      Light l1 on (1075, 150)
      Sensor s1 on (1075, 150)
      Sensor s2 on (1075, 375)
      Light l3 on (1075, 625)
      Sensor s3 on (1075, 625)
      Light l4 on (1075, 875)
      Sensor s4 on (1075, 875)
      Sensor s5 on (1075, 1125)
      Light l6 on (1075, 1350)
      Sensor s6 on (1075, 1350)
    }
  }
  Corridor c02 {
    Area: (0, 300) (1150, 300) (1150, 450) (0, 450)
    Rooms: r01 r02 r03 r04 r09
    Devices {
      Light l1 on (120, 375)
      Sensor s1 on (120, 375)
      Light l2 on (360, 375)
      Sensor s2 on (360, 375)
      Light l3 on (600, 375)
      Sensor s3 on (600, 375)
      Light l4 on (840, 375)
      Sensor s4 on (840, 375)
      Light l5 on (1075, 375)
      Sensor s5 on (1075, 375)
    }
  }
  Corridor c03 {
    Area: (0, 1050) (1150, 1050) (1150, 1200) (0, 1200)
    Rooms: r05 r06 r07 r08 r11
    Devices {
      Light l1 on (120, 1125)
      Sensor s1 on (120, 1125)
      Light l2 on (360, 1125)
      Sensor s2 on (360, 1125)
      Light l3 on (600, 1125)
      Sensor s3 on (600, 1125)
      Light l4 on (840, 1125)
      Sensor s4 on (840, 1125)
      Light l5 on (1075, 1125)
      Sensor s5 on (1075, 1125)
    }
  }

  Scenarios {
    Scenario ShortDay for Actor 0 {
              (0, 375)
        [20]    (1075, 375)
        [10]    (1075, 125)
        [10]    (1350, 350)
        [120]   (1350, 350) // @desk (r09)
        [10]    (1075, 125)
        [15]    (1050, 400)
        [45]    (1050, 400) // coffee
        [10]    (1075, 125)
        [5]     (1350, 350)
        [600]  (1350, 350) // @desk
        [10]    (1075, 125)
        [5]     (1075, 375)
        [10]    (625, 375)
        [5]     (625, 275)
        [20]    (625, 275) // fetching @r02
        [5]     (625, 375)
        [10]    (125, 375)
        [5]     (125, 275)
        [15]    (125, 275) // fetching @r01
        [5]     (125, 375)
        [10]    (1050, 400)
        [45]    (1050, 400) // coffee
        [10]    (1075, 625)
        [5]     (1175, 625)
        [10]    (1175, 625) // fetching @r10
        [5]     (1075, 625)
        [10]    (1075, 1125)
        [10]    (125, 1125)
        [5]     (125, 1025)
        [10]    (125, 1025) // fetching @r05
        [5]     (125, 1125)
        [15]    (1300, 1100)
        [600]  (1300, 1100) // stand-up @r11
        [15]    (625, 1125)
        [5]     (625, 1225)
        [80]    (625, 1225) // question @r08
        [5]     (625, 1125)
        [5]     (1025, 1075)
        [10]    (1050, 400)
        [30]    (1050, 400) //coffee
        [10]    (1075, 125)
        [10]    (1350, 350)
        [2400]  (1350, 350) // @desk
        [10]    (1075, 125)
        [5]     (1075, 375)
        [10]    (125, 375)
        [5]     (125, 525)
        [180]   (125, 525) // question @r03
        [5]     (125, 375)
        [15]    (1050, 400)
        [25]    (1050, 400) // coffee
        [5]     (1075, 125)
        [5]     (1350, 350)
        [1440]  (1350, 350) // @desk
        [5]     (1075, 125)
        [10]    (1075, 1125)
        [5]     (625, 1125)
        [5]     (625, 900)
        [90]    (625, 900) // question @r06
        [5]     (850, 950)
        [330]   (850, 950) // question @r06
        [10]    (625, 1125)
        [5]     (1075, 1125)
        [15]    (1075, 125)
        [5]     (1350, 350)
        [960]   (1350, 350) // @desk
        [5]     (1075, 125)
        [5]     (1050, 400)
        [30]    (1050, 400) // coffee
        [5]     (1075, 125)
        [5]     (1350, 350)
        [1020]  (1350, 350) // @desk
        [5]     (1075, 125)
        [10]    (1075, 375)
        [20]    (0, 375)
    }
  }
}
```

Listing D.1: *The MediumFloor specification using the Lighting DSL.*

APPENDIX E

Connector DSL grammar

```
grammar nl.ru.sws.connectordsl.ConnectorDSL with org.eclipse.xtext.common.
     Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate connectorDSL "http://www.ru.nl/sws/connectordsl/ConnectorDSL"

Model:
  config=Configuration
  containers+=Container+
  (handler=Handler)?
;

Configuration:
  'Server' '{'
    server=SocketConfig
  '}'
  'Client' '{'
    client=SocketConfig
  '}'
  'Base' base=[Container]
;

SocketConfig:
  'Name' name=ID
  'Protocol' protocol=Protocol
  'Type' type=ComType
;

Handler:
  'Handler' handler=STRING
;

Container:
  "DataType" name=ID ('responds' responds=[Container] '(' identifier=
       ComponentReference ')')? "{"
    "Components" "{"
      components += Component+
    "}"
    (fromSocket ?= "FromSocket")?
  "}"
;

ComponentReference:
  container=[Container] '.' component=[Component]
;

Component:
  type=ComponentType
  (
      ("[" count=[Component] "]")
    | ("(" byteCount=[Component] ")")
  )?
  name=ID
  (optional ?= "optional")?
;

ComponentType:
  ConditionalType | DataType
;

ConditionalType:
  "{" component=[Component] "|" conditions+=Condition ("," conditions+=
       Condition)* "}"
;

Condition:
  value=HEXPAIR "=>" type=ComponentType
;

DataType:
  dataType=BuiltinDataType | container=[Container]
;

enum BuiltinDataType:
    none = "None"
  | uint8 = "uint8"
  | ushort = "ushort"
  | uint = "uint"
  | ulong = "ulong"
  | int8 = "int8"
  | short = "short"
  | int = "int"
  | long = "long"
  | float = "float"
  | boolean = "bool"
  | bytes = "bytes"
  | string = "string"
;

enum Protocol:
      tcp = "TCP"
    | udp = "UDP"
;

enum ComType:
      bytes = "bytes"
    | json = "json"
;

terminal HEXPAIR returns ecore::EString:
  ('0x'|'x') (HEXCHAR HEXCHAR)+
;

terminal HEXCHAR returns ecore::EChar:
  ('0'..'9'|'A'..'F'|'a'..'f')
;
```

Listing E.1: *The Connector DSL grammar.*

APPENDIX F

SingleWatertank system (CoHLA)

```
Environment {
  RTI {
    OpenRTI
    Libraries "/opt/OpenRTI-libs"
  }
  PublishOnlyChanges
}

FederateClass SingleWatertank {
  Type FMU
  Attributes {
    Input Real valvecontrol
    Output Real level
  }
  DefaultModel "models/singlewatertank-20sim.fmu"
  AdvanceType TimeAdvanceRequest
  DefaultStepSize 0.1
  DefaultLookahead 0.000001
}

FederateClass WatertankController {
    Type FMU
    Attributes {
      Input Real level
      Output Boolean valve
    }
    Parameters {
      Real Maxlevel "maxlevel"
      Real Minlevel "minlevel"
    }
    DefaultModel "models/watertankController.fmu"
    AdvanceType TimeAdvanceRequest
    DefaultStepSize 0.1
    DefaultLookahead 0.000001
}

FederateClass Logger {
  Type CSV-logger {
    DefaultMeasureTime 30.0
  }
}

Configuration defaultWC for WatertankController {
  Maxlevel = "2.0"
  Minlevel = "1.0"
}

Federation SingleWatertankSystem {
  Instances {
    wt: SingleWatertank
    controller : WatertankController
    log : Logger
  }
  Connections {
    { controller.level <- wt.level }
    { wt.valvecontrol <- controller.valve }
    { log <- wt.level, controller.valve }
  }

  Situation default {
    Apply defaultWC to controller
  }
}
```

Listing F.1: *The SingleWatertank system as specified in CoHLA.*

APPENDIX G

Connector DSL examples

Generated Python class

```
class Message(Serializable):
  def __init__(self, sender, receipient, message):
    self.sender = sender
    self.receipient = receipient
    self.message = message

  @staticmethod
  def from_bytes(_buffer):
    sender, _buffer = bytes_to_string(_buffer)
    receipient, _buffer = bytes_to_string(_buffer)
    message, _buffer = bytes_to_string(_buffer)
    return Message(sender, receipient, message), _buffer

  def to_bytes(self):
    sender = b'' if self.sender is None else to_bytes('', self.sender)
    receipient = b'' if self.receipient is None else to_bytes('', self.
        receipient)
    message = b'' if self.message is None else to_bytes('', self.message)
    return b''.join([sender, receipient, message])

  @staticmethod
  def from_dict(_json):
    sender = str(_json['sender'])
    receipient = str(_json['receipient'])
    message = str(_json['message'])
    return Message(sender, receipient, message)

  def to_dict(self):
    return {
      'sender': self.sender,
      'receipient': self.receipient,
      'message': self.message
    }
```

Listing G.1: *Generated Python class for (de-)serialising the messages according to the protocol.*

Generated POOSL class

```
data class Message extends Object
variables
  Sender : String
  Receipient : String
  Message : String
methods
  getSender : String
    return Sender
  setSender(Sender_ : String) : Message
    Sender := Sender_;
    return self
  getReceipient : String
    return Receipient
  setReceipient(Receipient_ : String) : Message
    Receipient := Receipient_;
    return self
  getMessage : String
    return Message
  setMessage(Message_ : String) : Message
    Message := Message_;
    return self

  set(Sender_ : String, Receipient_ : String, Message_ : String) : Message
    Sender := Sender_;
    Receipient := Receipient_;
    Message := Message_;
    return self

  fromMap(json : Map) : Message
    return self set(json at("sender"), json at("receipient"), json at("
        message"))

  toMap : Map | json : Map, Sender_ : String, Receipient_ : String, Message_
      : String |
    json := new(Map);
    Sender_ := Sender;
    Receipient_ := Receipient;
    Message_ := Message;
    json putAt("sender", Sender_) putAt("receipient", Receipient_) putAt("
        message", Message_);
    return json

  printString : String
    return self toMap printString
```

Listing G.2: *Generated POOSL class for (de-)serialising the message according to the protocol.*

Bibliography

[1] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. *IEEE Std 1516-2010*, pages 1–38, Aug 2010. Cited on pages 9 and 29.

[2] A. Backlund. The definition of system. *Kybernetes*, 29(4):444–451, 2000. Cited on page 2.

[3] J. Beckers, G. Muller, W. Heemels, and B. Bukkems. Effective industrial modeling for high-tech systems: The example of Happy Flow: Seventeenth Annual International Symposium of the International Council On Systems Engineering (INCOSE) June 24-28, 2007. In *INCOSE International Symposium*, volume 17, pages 1758–1769. Wiley Online Library, 2007. Cited on page 2.

[4] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016. Cited on page 47.

[5] D. Bjørner. The Vienna Development Method (VDM). In *Mathematical Studies of Information Processing*, pages 326–359. Springer, 1979. Cited on page 23.

[6] D. Bjørner and C. B. Jones. The Vienna Development Method: The Meta-Language. *Language*, 61:3–5, 1978. Cited on page 23.

[7] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, J. Mauß, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, S. Wolf, and C. Clauß. The Functional Mockup Interface for tool independent exchange of simulation models. In *8th Modelica Conference*, pages 105–114, 2011. Cited on pages 8, 10, and 27.

[8] P. Bocciarelli, A. D'Ambrogio, A. Falcone, A. Garro, and A. Giglio. A model-driven approach to enable the simulation of complex systems on distributed architectures. *SIMULATION*, pages 1–27, 2019. Cited on page 9.

[9] B. Boehm and V. Basili. Software Defect Reduction Top 10 List. *IEEE Computer*, 34(1):135–137, 2001. Cited on page 2.

[10] L. v. Bokhoven. *Constructive tool design for formal languages: from semantics to executing models*. PhD thesis, Eindhoven University of Technology, 2002. Cited on page 21.

[11] J. F. Broenink. 20-sim software for hierarchical bond-graph/block-diagram models. *Simulation Practice and Theory*, 7(5-6):481–492, 1999. Cited on page 21.

[12] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate Composition of FMUs for Co-simulation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 2:1–2:12, Piscataway, NJ, USA, 2013. IEEE Press. Cited on page 8.

[13] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4:155–182, 1994. Cited on page 8.

[14] M. Burns, T. Roth, E. Griffor, P. Boynton, J. Sztipanovits, and H. Neema. Universal CPS Environment for Federation (UCEF). In *2018 Winter Simulation Innovation Workshop*, 2018. Cited on page 9.

[15] A. F. Case. Computer-aided software engineering (CASE): technology for improving software development productivity. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems*, 17(1):35–43, 1985. Cited on page 7.

[16] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly. The department of defense high level architecture. In *Winter Simulation Conference*, pages 142–149. Citeseer, 1997. Cited on page 9.

[17] R. Doornbos, B. Huijbrechts, J. Sleuters, J. Verriet, K. Ševo, and M. Verberkt. A domain model-centric approach for the development of large-scale office lighting systems. In *International Conference on Complex Systems Design & Management*, pages 109–120. Springer, 2018. Cited on page 122.

[18] R. Doornbos, J. Verriet, and M. Verberkt. Robustness analysis for indoor lighting systems. In *10th International Conference on Systems, Barcelona, Spain*, 2015. Cited on page 122.

[19] E. Durr and J. Van Katwijk. VDM++, a formal specification language for object-oriented designs. In *1992 Proceedings Computer Systems and Software Engineering*, pages 214–219. IEEE, 1992. Cited on page 23.

[20] J. Eker, J. W. Janneck, E. A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003. Cited on page 8.

[21] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. Cited on page 31.

[22] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010. Cited on pages 4 and 47.

[23] A. Falcone, A. Garro, A. Anagnostou, N. R. Chaudhry, O.-A. Salah, and S. J. Taylor. Easing the development of HLA Federates: the HLA Development Kit and its exploitation in the SEE Project. In *2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 50–57. IEEE, 2015. Cited on page 9.

[24] A. Falcone, A. Garro, S. J. Taylor, and A. Anagnostou. Simplifying the development of HLA-based distributed simulations with the HLA Development Kit software framework (DKF). In *Proceedings of the 21st International Symposium on Distributed Simulation and Real Time Applications*, pages 216–217. IEEE Press, 2017. Cited on page 9.

[25] P. A. Fishwick. Simulation model design. In *Proceedings of the 26th conference on Winter simulation*, pages 173–175. Society for Computer Simulation International, 1994. Cited on page 3.

[26] J. Fitzgerald, C. Gamble, R. Payne, and B. Lam. Exploring the Cyber-Physical Design Space. In *INCOSE International Symposium*, volume 27, pages 371–385. Wiley Online Library, 2017. Cited on page 10.

[27] J. Fitzgerald, P. G. Larsen, and S. Sahara. VDMTools: advances in support for formal modeling in VDM. *ACM Sigplan Notices*, 43(2):3, 2008. Cited on page 23.

[28] M. Fowler. *Domain-specific languages*. Pearson Education, 2010. Cited on pages 4 and 47.

[29] S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014. Cited on page 10.

[30] P. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010. Cited on page 24.

[31] P. Fritzson and V. Engelson. Modelica—a unified object-oriented language for system modeling and simulation. In *European Conference on Object-Oriented Programming*, pages 67–90. Springer, 1998. Cited on page 24.

[32] R. M. Fujimoto. Time management in the high level architecture. *Simulation*, 71(6):388–400, 1998. Cited on pages 9 and 33.

[33] A. Garro and A. Falcone. On the integration of HLA and FMI for supporting interoperability and reusability in distributed simulation. In *Proceedings of*

the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, DEVS '15, pages 9–16. Society for Computer Simulation International, 2015. Cited on page 9.

[34] M. Hause. The SysML modelling language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12. Citeseer, 2006. Cited on page 9.

[35] G. Hemingway, H. Neema, H. Nine, J. Sztipanovits, and G. Karsai. Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. *Simulation*, 88(2):217–232, 2012. Cited on page 9.

[36] J. Hooman, N. Mulyar, and L. Posta. Coupling Simulink and UML models. In *Proceedings of Symposium FORMS/FORMATS*, pages 304–311, 2004. Cited on page 8.

[37] R. Isermann, J. Schaffnit, and S. Sinsel. Hardware-in-the-loop simulation for the design and testing of engine-control systems. *Control Engineering Practice*, 7(5):643 – 653, 1999. Cited on page 3.

[38] ISO. IEC 19505-2: 2012 Information technology–Object Management Group Unified Modeling Language (OMG UML), superstructure, 2012. Cited on pages 2 and 7.

[39] E. Kang, E. Jackson, and W. Schulte. An Approach for Effective Design Space Exploration. In R. Calinescu and E. Jackson, editors, *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, pages 33–54, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. Cited on page 65.

[40] G. Karsai and J. Sztipanovits. Model-integrated development of cyber-physical systems. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 46–54. Springer, 2008. Cited on page 8.

[41] A. Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008. Cited on pages 4 and 47.

[42] A. G. Kleppe, J. Warmer, J. B. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003. Cited on page 7.

[43] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The Overture Initiative Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, Jan. 2010. Cited on page 24.

[44] P. G. Larsen, J. Fitzgerald, J. Woodcock, P. Fritzson, J. Brauer, C. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis, and A. Sadovykh. Integrated tool chain for model-based design of cyber-physical systems: the INTO-CPS project. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pages 1–6. IEEE, 2016. Cited on page 10.

[45] P. G. Larsen, J. Fitzgerald, J. Woodcock, C. Gamble, R. Payne, and K. Pierce. Features of Integrated Model-Based Co-modelling and Co-simulation Technology. In *Software Engineering and Formal Methods*, pages 377–390. Springer, 2018. Cited on page 10.

[46] E. A. Lee. Cyber-physical systems-are computing foundations adequate. In *Position paper for NSF workshop on cyber-physical systems: research motivation, techniques and roadmap*, volume 2, pages 1–9. Citeseer, 2006. Cited on pages 1 and 2.

[47] E. A. Lee and I. John. Overview of the Ptolemy project, 1999. Cited on page 8.

[48] S. E. Mattsson, H. Elmqvist, and M. Otter. Physical system modeling with Modelica. *Control Engineering Practice*, 6(4):501–510, 1998. Cited on page 24.

[49] B. Möller, F. Antelius, M. Johansson, and M. Karlsson. Building Scalable Distributed Simulations: Design Patterns for HLA DDM. In *Proc. of Fall Simulation Interoperability Workshop, 2016-SIW-003*. Simulation Interoperability Standards Organization, 2016. Cited on page 143.

[50] M. Monchiero, R. Canal, and A. González. Design Space Exploration for Multicore Architectures: A Power/Performance/Thermal View. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 177–186, New York, NY, USA, 2006. ACM. Cited on page 62.

[51] N. Mühleis, M. Glaß, L. Zhang, and J. Teich. A co-simulation approach for control performance analysis during design space exploration of cyber-physical systems. *SIGBED Rev.*, 8(2):23–26, June 2011. Cited on page 65.

[52] H. Neema, J. Gohl, Z. Lattmann, J. Sztipanovits, G. Karsai, S. Neema, T. Bapty, J. Batteh, H. Tummescheit, and C. Sureshkumar. Model-based integration platform for FMI co-simulation and heterogeneous simulations of cyber-physical systems. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, number 096, pages 235–245. Linköping University Electronic Press, 2014. Cited on page 9.

[53] H. Neema, Z. Lattmann, P. Meijer, J. Klingler, S. Neema, T. Bapty, J. Sztipanovits, and G. Karsai. Design space exploration and manipulation for cyber physical systems. In *IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems (IDEAL'2014), Springer-Verlag Berlin Heidelberg*, 2014. Cited on page 8.

[54] Y. Ni. *System Design Support of Cyber-Physical Systems, a co-simulation and co-modelling approach*. PhD thesis, University of Twente, Enschede, Netherlands, June 2015. Cited on pages 9 and 110.

[55] Y. Ni and J. Broenink. Hybrid systems modelling and simulation in DESTECS: a co-simulation approach. In M. Klumpp, editor, *26th European Simulation and Modelling Conference, ESM 2012*, pages 32–36. EUROSIS-ETI, 10 2012. Cited on page 9.

[56] H. M. Paynter. *Analysis and design of engineering systems*. MIT press, 1961. Cited on page 21.

[57] A. D. Pimentel. Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration. *IEEE Design Test*, 34(1):77–90, Feb 2017. Cited on pages 3 and 65.

[58] C. Ptolemaeus. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org Berkeley, 2014. Cited on page 8.

[59] L. Roscoe. Stereolithography interface specification. *America-3D Systems Inc*, 27, 1988. Cited on page 68.

[60] T. Roth, E. Song, M. Burns, H. Neema, W. Emfinger, and J. Sztipanovits. Cyber-physical system development environment for energy applications. In *ASME 2017 11th International Conference on Energy Sustainability*, 2017. Cited on page 10.

[61] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. Cited on page 7.

[62] F. Schloegl, S. Rohjans, S. Lehnhoff, J. Velasquez, C. Steinbrink, and P. Palensky. Towards a classification scheme for co-simulation approaches in energy systems. In *2015 International Symposium on Smart Electric Distribution Systems and Technologies (EDST)*, pages 516–521, Sep. 2015. Cited on page 3.

[63] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, Feb 2006. Cited on page 7.

[64] M. Schuts and J. Hooman. Formal Modelling in the Concept Phase of Product Development. In *WORLDCOMP'15-The 2015 World Congress in Computer Science, Computer Engineering, and Applied Computing*, pages 3–9. Herndon: CSREA Press, 2015. Cited on page 22.

[65] G. Schweiger, C. Gomes, G. Engel, J.-P. Schoeggl, A. Posch, I. Hafner, and T. Nouidu. An empirical survey on co-simulation: Promising standards, challenges and research needs. *arXiv preprint arXiv:1901.06262*, 2019. Cited on pages 4, 27, and 29.

[66] D. Sciuto, F. Salice, L. Pomante, and W. Fornaciari. Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 55–60. ACM, May 2002. Cited on page 62.

[67] L. Sha, S. Gopalakrishnan, X. Liu, and Q. Wang. Cyber-physical systems: A new frontier. In *2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (sutc 2008)*, pages 1–9. IEEE, 2008. Cited on page 7.

[68] J. Siegel and D. Frantz. *CORBA 3 fundamentals and programming*, volume 2. John Wiley & Sons New York, NY, USA:, 2000. Cited on page 7.

[69] J. Sleuters, Y. Li, J. Verriet, M. Velikova, and R. Doornbos. A Digital Twin Method for Automated Behavior Analysis of Large-Scale Distributed IoT Systems. In *2019 14th Annual Conference System of Systems Engineering (SoSE)*, pages 7–12, May 2019. Cited on page 122.

[70] J. A. Sokolowski and C. M. Banks. *Principles of modeling and simulation: a multidisciplinary approach*. John Wiley & Sons, 2011. Cited on page 3.

[71] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91 – 99, 2001. Cited on page 4.

[72] R. Stark, F.-L. Krause, C. Kind, U. Rothenburg, P. Müller, H. Hayka, and H. Stöckert. Competing in engineering design – The role of Virtual Product Creation. *CIRP Journal of Manufacturing Science and Technology*, 3(3):175 – 184, 2010. Cited on page 2.

[73] J. Sztipanovits, T. Bapty, S. Neema, L. Howard, and E. Jackson. *OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems*, pages 235–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. Cited on page 8.

[74] B. D. Theelen, O. Florescu, M. Geilen, J. Huang, P. Van Der Putten, and J. P. Voeten. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. In *5th Conference on Formal Methods and Models for Codesign*, MEMOCODE '07, pages 139–148. IEEE Computer Society, 2007. Cited on page 21.

[75] B. D. Theelen, J. Voeten, and R. Kramer. Performance modelling of a network processor using POOSL. *Computer Networks*, 41(5):667–684, 2003. Cited on page 22.

[76] C. Thule. Verifying the Co-Simulation Orchestration Engine for INTO-CPS. In *CEUR Workshop Proceedings*, volume 1744, 2016. Cited on page 11.

[77] C. Thule and P. G. Larsen. Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS. *Proceedings of the Institute for System Programming*, 28(2):139–156, 2016. Cited on page 11.

[78] O. Topçu, L. Yilmaz, H. Oguztüzün, and U. Durak. Distributed simulation. In *A Model Driven Engineering Approach*. Springer, 2016. Cited on page 9.

[79] J. van Amerongen. Modelling, simulation and controller design for mechatronic systems with 20-sim 3.0. *IFAC Proceedings Volumes*, 33(26):763–768, 2000. Cited on page 21.

[80] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000. Cited on pages 4 and 47.

[81] M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *International Symposium on Formal Methods*, pages 147–162. Springer, 2006. Cited on page 23.

[82] J. Verriet, L. Buit, R. Doornbos, B. Huijbrechts, K. Sevo, J. Sleuters, and M. Verberkt. Virtual prototyping of large-scale IoT control systems using domain-specific languages. In *7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 231–241, 2019. Cited on page 122.

[83] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering*, 16(3), 2010. Cited on page 4.

[84] J. Voeten, T. Hendriks, B. Theelen, J. Schuddemat, W. Suermondt, J. Gemei, K. Kotterink, and C. van Huët. Predicting Timing Performance of Advanced Mechatronics Control Systems. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 206–210, 2011. Cited on page 22.

[85] J. Westland. The cost of errors in software development: evidence from industry. *The Journal of Systems and Software*, 62:1–9, 2002. Cited on page 2.

[86] Wook Hyun Kwon and Seong-Gyu Choi. Real-time distributed software-in-the-loop simulation for distributed control systems. In *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pages 115–119. IEEE, 1999. Cited on page 3.

Summary

Cyber-physical systems (CPSs) are becoming ever more important in both industry and our everyday lives. A CPS integrates software components (cyber) with physical components. Typically, the software controls physical processes, such as motors or other actuators, based on sensor input. Examples of CPSs are airplanes, modern cars and industrial production line robots. These systems are highly complex systems that are constructed of many components working together. The system components have a multidisciplinary nature, as a system might contain mechanical components, electrical components and software components. Every discipline has its own development methods and tools and, in the end, all separately developed components should work together as one system.

Using a model-based development approach, all disciplines develop component models using their own tools and development cycles. While these individual component models may be simulated to verify their behaviour, it is hard to simulate them together to get a better understanding of the designed system as a whole. Co-simulation of the models provides means to analyse the system behaviour before building a prototype and to check the collaboration of the components. Several techniques exist that support the co-simulation of many simulations. Examples are the High Level Architecture (HLA) and Functional Mock-up Interface (FMI) standards. HLA provides an interface and a set of rules to orchestrate a co-simulation execution, while FMI provides an interface for communicating with the simulation models. Both open source and commercial implementations of HLA are available and FMI is widely supported by modelling tools. The construction of a co-simulation using these standards, however, introduces a significant additional workload for the system designers. Especially when the component models change frequently during the design process, adapting these changes in the co-simulation framework quickly becomes a blocking factor.

This dissertation introduces a domain-specific language (DSL) called CoHLA that supports existing model-based methodologies for the design of CPSs to rapidly construct a co-simulation of the system under design. CoHLA uses the HLA standard for the co-simulation execution and the FMI standard to support simulation models created in many different modelling tools. Its aim is to minimise the overhead for developing and maintaining a co-simulation during the development. CoHLA allows the system architects to quickly specify simulation models in terms of input and output attributes, after which the co-simulation itself can be specified by connecting these attributes. Source code for the co-simulation framework is generated from the co-simulation specification.

The CoHLA framework features basic logging as well as the specification of reusable parameter configurations for the co-simulation. Additionally, it supports the measurement of basic performance metrics, the specification and replay of scenarios, fault injection and basic design space exploration. A collision detection extension that uses 3D drawings of the system's components to detect potential collisions was also implemented. This extension can also be used to render the system during the co-simulation execution to provide visual feedback to the user.

A number of case studies were conducted with CoHLA to analyse different aspects of the approach. A co-simulation of a domestic heating system was used as a basic example of a CPS to design using CoHLA. In collaboration with the University of Twente a slider system was designed and built that reflects relevant design aspects for industry such as collision detection. This case study showed that CoHLA enabled the construction of a co-simulation in an early phase during the design of a system, which also revealed potential design flaws in an early stage. Consequently, these errors could be addressed early in the design process. Even though the models changed from time to time throughout the development, these changes could be adapted quickly – within an hour – using CoHLA.

To analyse the trustworthiness of the co-simulation results, the impact of the HLA implementation and the CoHLA framework on the simulation timing has been measured. For one sample system, the co-simulation results from a CoHLA co-simulation were compared to the results when executing the same models in one integrated simulator. Even though small differences were found, the results were very similar. The results from a CoHLA co-simulation were also compared to an established co-simulation project by running identical co-simulations. Since these results were also nearly identical, the co-simulation results from a co-simulation as generated by CoHLA appear to be trustworthy.

Systems such as Internet of Things (IoT) systems are a class of CPSs that consist of large numbers of sensors and actuators. A case study on a smart lighting system was used as an example of an IoT system that could be designed following a similar approach using CoHLA. The focus of this case study was to analyse the scalability of HLA and CoHLA. Experiments were conducted by running the co-simulations in a distributed manner using a commercial cloud provider. The impact of distributing the individual simulations over a number of computation nodes was analysed. It was found that HLA scales rather good when distributed over a number of nodes. To simplify the specification of a smart lighting system in CoHLA, a separate DSL was developed. The approach of introducing a new DSL to specify a specific type of system proved to be beneficial for a specific set of systems.

Using co-simulation during system design allows for early system analysis and the development of system-level features. With CoHLA, the construction of a co-simulation from simulation models of different disciplines becomes less time consuming compared to other approaches. Also, adapting changes of the models in the co-simulation requires less effort, which makes the approach suitable for maintaining the co-simulation throughout the system design process.

Samenvatting

Zowel in de industrie als in ons dagelijks leven spelen cyber-physical systemen (CPSen) een steeds belangrijkere rol. Een CPS integreert software componenten met fysieke componenten. Daarbij is de software doorgaans verantwoordelijk voor de aansturing van de fysieke processen, zoals motoren en actuatoren. De software gebruikt hiervoor als input metingen van sensoren. Voorbeelden van CPSen zijn vliegtuigen, moderne auto's and industriële robots voor productielijnen. Dit zijn allen zeer complexe systemen die bestaan uit veel samenwerkende componenten uit verschillende disciplines. Een systeem kan bijvoorbeeld bestaan uit mechanische, elektrische en software componenten. Elk discipline heeft haar eigen methodes voor de ontwikkeling van deze componenten.

Met een model-gebaseerde aanpak kunnen alle disciplines hun eigen componentmodellen ontwikkelen. Deze losse modellen kunnen vaak gesimuleerd worden om het gedrag te verifiëren, maar het is lastig om ze samen te simuleren om het gedrag van het ontworpen systeem als geheel te analyseren. Co-simulatie van de modellen biedt de mogelijkheid om het gedrag van het systeem vroegtijdig te analyseren en om de samenwerking van de componenten te controleren. Er bestaan verschillende technieken om co-simulaties van verschillende soorten modellen te ondersteunen. Voorbeelden zijn de High Level Architecture (HLA) en de Functional Mock-up Interface (FMI) standaarden. HLA biedt een interface en regels voor het coördineren van een co-simulatie. FMI biedt een interface om te communiceren met simulatiemodellen. Van HLA zijn zowel open source als commeciële implementaties beschikbaar en FMI wordt ondersteund door veel modelleerapplicaties. Het opzetten en onderhouden van een co-simulatie met deze standaarden creëert echter een significante extra taak voor de systeemontwerpers. Vooral wanneer de componentmodellen gedurende het ontwerpproces vaak veranderen kan dit een blokkerende werking hebben.

Dit proefschrift introduceert CoHLA, een domein-specifieke taal (DSL) die bestaande modelgebaseerde methodologieën voor de ontwikkeling van CPSen ondersteunt door snel co-simulaties te kunnen maken van het systeem in ontwikkeling. CoHLA gebruikt de HLA standaard voor het uitvoeren van de co-simulatie en de FMI standaard om brede ondersteuning voor modellen van verschillende modelleerapplicaties te bieden. Het doel is om de overhead van het ontwikkelen en onderhouden van de co-simulatie tijdens het design te minimaliseren. CoHLA stelt

de gebruiker in staat om simulatiemodellen snel te specificeren in termen van attributen, waarna de co-simulatie gespecificeerd kan worden door deze attributen met elkaar te verbinden. Uit deze specificatie wordt broncode gegenereerd voor de co-simulatie.

Het CoHLA framework ondersteunt de specificatie van herbruikbare parameter configuraties voor de co-simulatie. Verder is er ondersteuning voor enkele basismetrieken voor het meten van de prestatie van het systeem, de specificatie en het afspelen van scenario's, foutinjectie en de automatische uitvoering van verschillende configuraties van het systeem. Verder is er botsingsdetectie op basis van 3D tekeningen van de componenten van het systeem om potentiële botsingen te detecteren. Het kan ook gebruikt worden om de staat van het systeem te visualiseren voor de gebruiker.

Om verschillende aspecten van de aanpak te analyseren zijn er een aantal case studies uitgevoerd met CoHLA. Een co-simulatie van een kamerthermostaatsysteem is gebruikt als basisvoorbeeld van een te ontwerpen CPS met behulp van CoHLA. In samenwerking met de Universiteit van Twente is er een slider systeem ontworpen en gebouwd waarin relevante aspecten voor de industrie naar voren komen, zoals de detectie van botsingen. Deze case study laat zien dat het met CoHLA mogelijk is om vroeg in het ontwikkelproces een co-simulatie van het systeem te maken. Hierdoor werden er enkele fouten in het ontwerp gevonden, waarna deze vroeg in het proces konden worden hersteld. Veranderingen aan de modellen tijdens het ontwikkelproces kunnen snel – in minder dan een uur – doorgevoerd worden door middel van CoHLA.

Om de betrouwbaarheid van de co-simulatieresultaten te analyseren is de invloed van de HLA implementatie en het CoHLA framework op het tijdsgedrag van de simulatie gemeten. Voor een voorbeeldsysteem zijn de co-simulatieresultaten van CoHLA vergeleken met de resultaten van een simulatie van dezelfde modellen in een geïntegreerde simulator. Hoewel er kleine verschillen werden gevonden zijn de resultaten erg vergelijkbaar. De resultaten van een CoHLA co-simulatie zijn ook vergeleken met een ander co-simulatie project door identieke co-simulaties uit te voeren. Omdat ook deze resultaten nagenoeg identiek zijn lijken de resultaten komend uit een CoHLA co-simulatie betrouwbaar.

Internet of Things (IoT) systemen zijn een klasse van CPSen welke bestaan uit een groot aantal sensoren en actuatoren. Een case study naar een systeem voor intelligente verlichting is gebruikt als voorbeeld van een IoT systeem dat ontworpen kan worden met behulp van CoHLA. De focus van deze case study was om de schaalbaarheid van HLA en CoHLA te analyseren. Hiervoor zijn er experimenten uitgevoerd waarbij de individuele simulaties gedistribueerd zijn uitgevoerd bij een commerciële cloud aanbieder. De resultaten wijzen uit dat HLA vrij goed schaalt met deze methode. Een losse DSL is ontwikkeld om de specificatie van intelligente verlichtingssystemen in CoHLA te vereenvoudigen.

Door gebruik te maken van co-simulatie tijdens de ontwikkeling van een systeem is het mogelijk om het gedrag van het systeem vroeg in het ontwikkelproces te analyseren. Het bouwen van een co-simulatie op basis van simulatiemodellen kost met CoHLA minder tijd vergeleken met andere methodes. Ook het wijzigen de modellen van de co-simulatie is eenvoudiger, waardoor de aanpak ook geschikt is voor het onderhouden van de co-simulatie gedurende het ontwikkelproces.

Acknowledgements

This dissertation, its research and framework would not be without the support of a group of people whose names are not on the cover of this work. Nevertheless, I'd like to take the opportunity to express my gratitude to a number of people in this section.

First of all, I feel fortunate that Jozef offered this PhD position after being my supervisor for my master's thesis. Our weekly meetings on Wednesday not only facilitated fruitful discussions and gaining of new insights, but they also provided me with a weekly update on industrial projects he was working on. His knowledge and experience in both academia and industry were of great help. Thank you for the opportunity to work on this project and for your guidance throughout its execution.

I'd also like to thank Tim and Jan for their input in the project. In particular, the discussions with Tim on Mondays were very helpful and many of the discussed items found their way into the final result.

Despite their overfull agendas, Marcel and Gerco were always open for discussion and also provided lots of useful feedback throughout the years.

Petra and Ramon have shown to be great roommates in the office. Even though I wasn't much around, I'd like to thank you both for the discussions on research related as well as personal topics.

Special thanks to Matthijs, who, whenever there was something to discuss, complain or simply to talk about, always provided a friendly ear. Our discussions and chats have supported me greatly during the process of the research, both professionally and personally.

For our regular evening drinks in Nijmegen, I thank Matthijs, Mathijs, Koen, Joost, Ramon and Petra. These distractions from all seriousness during the week were very welcome.

Last, but definitely not least, I thank my parents – Helga and Reinoud –, my sister and her friend – Masha and Benjamin – and, of course, Laura for all the love and support along the way.

I thank all of you who supported me in the past four years.

Curriculum Vitae

Thomas Christian Nägele

April 8, 1992:
Born in Nijmegen, The Netherlands.

2004 – 2010:
Gymnasium, technical profile,
Olympus College, Arnhem, The Netherlands.

2010 – 2013:
Bachelor Computing Science,
Radboud University, Nijmegen, The Netherlands.

2013 – 2015:
Master Computing Science,
Radboud University, Nijmegen, The Netherlands.

2015:
Master's thesis,
Client-side performance profiling of JavaScript for web applications,
Topicus, Deventer, The Netherlands.

2015 – 2019:
PhD student,
Institute for Computing and Information Sciences,
Radboud University, Nijmegen, The Netherlands.

Titles in the IPA Dissertation Series since 2017