TERRA support for architecture modeling

K.J. (Karim) Kok

MSc Report

Committee:
Dr.ir. J.F. Broenink
Z. Lu, MSc
Prof.dr.ir. A. Rensink

August 2016

040RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

This report present the design, implementation and test results for the co-modeling and co-simulation support of physical systems in TERRA.

TERRA is a model-driven development suite for designing embedded control software for cyber-physical systems. The way of working for designing cyber-physical systems describes the steps which are necessary for the design of a cyber-physical system. The first step is to design an architecture model of the cyber-physical system. TERRA has a quit basic architecture editor which can draw an architecture model. The second step is to give in implementation to all the different components of the architecture model. At the moment the architecture editor of TERRA only supports the implementation by using the CSP editor of TERRA. The third step is to test the controller in combination with a model of the physical system. This step is not supported by TERRA yet.

The goal of this research is to design and implement the support for co-modeling and co-simulation of physical systems in TERRA. This makes it possible to use a model of a physical system in combination with the controller software for simulation purposes. Because there exists enough modeling software suites for physical systems, it is not necessary to create a new tool for it. For the modeling and simulation of physical system the modeling software suite *20-sim* is used. So for the co-modeling and co-simulation the modeling suites TERRA and 20-sim are used.

First the design for the support of co-modeling for physical systems in TERRA is provided. This support is given by first design a model of the physical system in the architecture editor of TERRA. This model contains only the interface for the model of the physical system. The detail design of the model itself is done in 20-sim. From the architecture editor in TERRA an 20-sim file is generated which contains the same interface for the physical system model as in the architecture editor. The implementation of this design is not implemented in TERRA yet due time constraints.

Second the design for the support of co-simulation for physical systems in TERRA is provided. To support the co-simulation between TERRA and 20-sim, the FMI standard is used. The FMI standard describes a standard interface for the simulation of models. The tool 20-sim generates an FMU from a model of a physical system which contains the dynamics of the model and the standard interface to perform simulations on the model. TERRA generates automatically an interface which is connected to the interface of the FMU. This interface in TERRA is automatically coupled to the controller software by the architecture editor of TERRA.

The implementation of the support for co-simulation for physical systems is added to TERRA by creating some plugins. These plugins contains model descriptions which are used to automatically generate the interface to the FMU. Some test are performed with simple physical system models to show the correct working of the co-simulation between TERRA and 20-sim.

At the moment the 20-sim tool itself can not be used as co-engine for the simulation of the physical system model but in the future this will be possible by using the same interface. It is advised that TERRA also get some support for compiling and linking of .c and library files because these files are mainly used as source files for the FMU. The interface to the FMU needs also a logger function to give information about the state of the FMU. This logger function is also created during the research. This logger function needs to be implemented in the LUNA framework because at the moment it needs to be added by hand.

Contents

1 Introduction

1.1 Context

Nowadays, cyber-physical systems (CPS) can be seen everywhere in our daily life, such as industrial robots, auto-mobiles, medical health systems and in people's homes. CPS are systems which combine *cyber* parts and *physical* parts. In Figure 1.1 a top level view of a generic cyber-physical system is shown. The three basic parts of CPS are the controller, input/output (I/O) and the plant. The controller belongs to the *cyber* domain and the I/O and plant belong to the *physical* domain. As can be seen in the figure the *physical* domain is divided into a *electrical* domain and a *mechanical* domain. Actuators and sensors are used to convert signals between the *electrical* domain and the *mechanical* domain. Digital-analog converters (DACs) and analog-digital converters (ADCs) are used to convert signals between the *cyber* domain and the *electrical* domain.

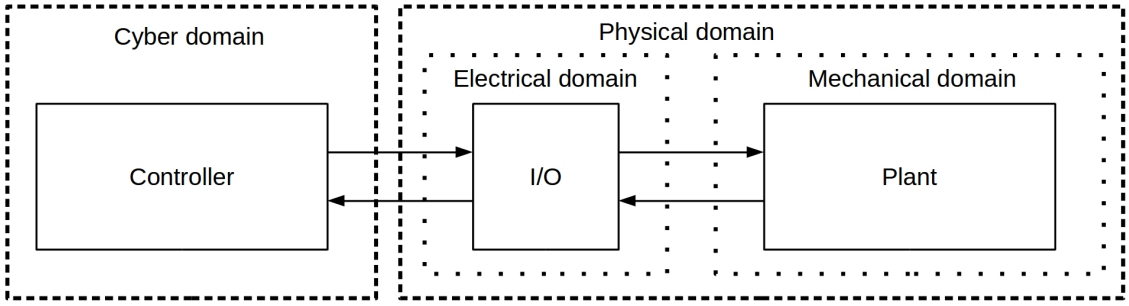

Figure 1.1: Top level view of a cyber-physical system

Designing control software for modern CPS become more and more difficult because of the increasing amount and complexity of their requirements (Kranenburg-de Lange et al., 2012). Due the increasing interaction with humans and other CPS, the need of safety of CPS arises. CPS also tend to become more mobile, so they also need to be as energy efficient as possible. All these requirements increase the development time and development costs for CPS. Because of the increase complexity of the systems, the possibility to make errors in the design phase become higher. These errors can have serious consequences for the system itself or its environment.

Model-Driven Development (MDD) is an engineering technique that uses model construction and model-based transformations to reach a desired end-result (Bezemer, 2013). In the case of CPS it means that models are used to create control software. Because models are used, it is possible to perform all kinds of complex tasks like model transformation and model simulation.

Because CPS belong to more than one domain, different type of models are used. Each model type has its advantages and disadvantages and is usable for a specific task or domain. Model simulation can be used to check the behaviour of a model. Model transformation can be used to transform a model to a formal language. With formal verification tools it is possible to check the model quality and consistency issues. This leads to less errors in real CPS.

Model transformation can also be used to transform different model types into a form that is compatible with all model types in order to combine them. The model-to-code transformation is an example of this kind of model transformations. Model-to-code transformation can be used to generate code from the different models to form the actual control software. This control software can directly run on an embedded processor in the cyber-physical system. With

model-to-code transformation it is not necessary to add code manually. This also leads to less errors in real CPS.

When using MDD techniques it is important to have a complete toolchain. This toolchain contains different tools for model construction and model transformations. Also the integration between these different tools in the toolchain is important. The main goals of the toolchain is to streamline the development process, preventing unnecessary human-based errors and reducing the development time of the control software.

1.1.1 TERRA

The *Twente Embedded Real-time Robotic Application (TERRA)*[1] is a model-driven design tool suite for designing (control) software for CPS. The ultimate goal of TERRA is to create (embedded) control software for CPS in a structural manner. TERRA should decrease the development time of the (embedded) control software design and the software should be 'first-time right'.

The current version of TERRA consists of a graphical Communicating Sequential Processes (CSP) editor and a graphical architecture editor. The CSP editor is used to create CSP models. These CSP models describes patterns of interaction in concurrent systems. From these CSP models, controller software for CPS can be generated in the form of LUNA-based C++ code.

The architecture editor is used to create architecture models. An architecture model describes how a system can be divided into different sub-systems an how these sub-systems interacts with each other. An example of an architecture model of a cyber-physical system is given in Figure 1.2.

Figure 1.2: Example of an architecture model in the architecture editor of TERRA

More information about CSP, the graphical CSP editor, the architecture editor and LUNA is given in Section 2.3

1.2 Problem Description

The current version of the architecture editor in TERRA is quite basic. At the moment it is possible to include CSP models from the graphical CSP editor into the architecture model. These CSP models are mainly focused on the design of the controller of the cyber-physical system. Looking at the top level view of a cyber-physical system as shown in Figure 1.1 it means that the controller part can be implemented in the architecture editor. The architecture editor gives also some support for the input/output interface. This interface is used for the realization of the software on embedded hardware. This interface is very basic and only consist of an PWM port, encoder port and a digital I/O port which is only specified for the Mesa Anything I/O FPGA board[2]. There is no support to add a model of the physical system in the architecture editor of TERRA which is also a part of a cyber-physical system.

Before the control software is implemented on a real physical system, it is important to test the behavior of the controller software. These tests can be performed on a real physical system

[1] https://www.ram.ewi.utwente.nl/ECSSoftware/terra.php, accessed on 8 April 2016.
[2] http://www.mesanet.com/fpgacardinfo.html, accessed on 23 July 2016.

but when the controller software does not work like expected, it can give dangerous situations. Therefore it is important to first test the control software in a simulation with a model of the physical system. Currently because it is not possible to add a model of the physical system in the architecture model, there is also no support to simulate the controller software created in TERRA in combination with a model of the physical system.

1.3 Goals

This assignment focus on the support of using models of physical systems in TERRA and to use these models in combination with the control software to perform simulations. These simulations are used to verify the behaviour of the controller software which is based on CSP models. To support the use of physical-system models in TERRA and to use these models in simulations two goals are defined:

- The first goal is to make it possible to add a model of a physical system in the architecture editor of TERRA. This means that the architecture model in TERRA does not only contains a controller block but must also contain a plant block which represents the model of the physical system. The detail design of the controller block can be created by the CSP editor of TERRA. Support must be added to TERRA to also give the plant block an detail design.
- The second goal is to make it possible to simulate the controller software which is based on CSP models in combination with the model of the physical system which is represented by the plant block in the architecture editor.

1.4 Approach

There exist enough modeling software suites for physical systems so it is not necessary to create a new tool in TERRA which can be used for the design and simulation of physical system models. An existing modeling software suite can be used for the design and simulation of physical systems. A connection between TERRA and one of the existing modeling software suites for physical systems needs to be created to archive the two goals.

For the first goal, a plant block is placed in the architecture editor of TERRA as shown in Figure 1.2. This plant block contains some ports which describes the interface for the physical system. This interface is also used for the model of the physical system. The interface of the plant block in the architecture editor of TERRA needs to be transfered to the editor of an existing modeling suite for physical systems. From this point it is possible to design an implementation for the model of the physical system using the interface which is defined in the architecture model. This approach is also known as co-modeling. The design of the controller is done in TERRA and the design of the model of the physical system is done in another modeling software suite.

For the second goal, a simulation must be performed between the controller in TERRA and the model of the physical system in another modeling software suite. The controller in TERRA is translated to code which can be executed. The inputs and outputs of the controller needs to be coupled to the model of the physical system. So during code generation of the controller, these inputs and outputs are coupled to the simulator of the model of the physical system. This approach is also known as co-simulation. The simulation of the controller and the simulation of the model of the physical system is done in their own environment.

Model-driven design techniques are used to translate the interface of the model of the physical system from the architecture editor in TERRA to an existing modeling software suite. These techniques are also used to generate a connection between the controller software and the simulator of the model of the physical system.

1.5 Outline

Some background information which is important for this assignment is provided in Chapter 2. The way of working for CPS and the basics of TERRA are explained in this chapter. The analysis of the architecture in TERRA is described in Chapter 3. It mainly focus on how to implement the support for co-modeling and co-simulation of physical systems in TERRA. A list of requirements for this assignment is also described in the chapter. The meta-models which are used in TERRA to support the co-modeling and co-simulation of physical systems in TERRA are provided in Chapter 4. The implementation of the co-simulation interface between the controller software and the simulator for physical systems which is automatically generated by TERRA is described in Chapter 5. To verify the implementation of the co-simulation interface, some test are performed which are discussed in Chapter 6. Finally, the report is concluded with a conclusion and recommendations.

2 Background

This chapter gives some background information which is used during this assignment. First the workflow for designing cyber-physical systems is explained. The following section gives some information about the CPC meta-model. The CPC meta-model is the basic of all other models which are used in TERRA. The third section gives some more information about the tool suite TERRA. A more detailed description about the graphical CSP editor is given. Also some information about the LUNA framework is given which is used in combination with TERRA. The fourth section gives some information about the tool Eclipse. TERRA is developed in Eclipse and uses some plugins which are available in Eclipse. The last section gives some information about the Functional Mock-up interface (FMI). The FMI can be used for model exchange and for co-simulations.

2.1 Way of Working

In general, designing a cyber-physical system contains the following phases as indicated in Figure 2.1 (Broenink et al., 2010a). As can be seen in the figure, an idea leads to multiple realizations. At the start of the design phase, an abstract top-level model is used. Via *Stepwise Refinement* more detail is added to this model to finally get the real working cyber-physical system. Each choice which is made during the design phase can result in a different realization of the cyber-physical system. All the different realizations together forms the *design space*. Trying out several alternatives to get the final realization is called *Design Space Exploration*.

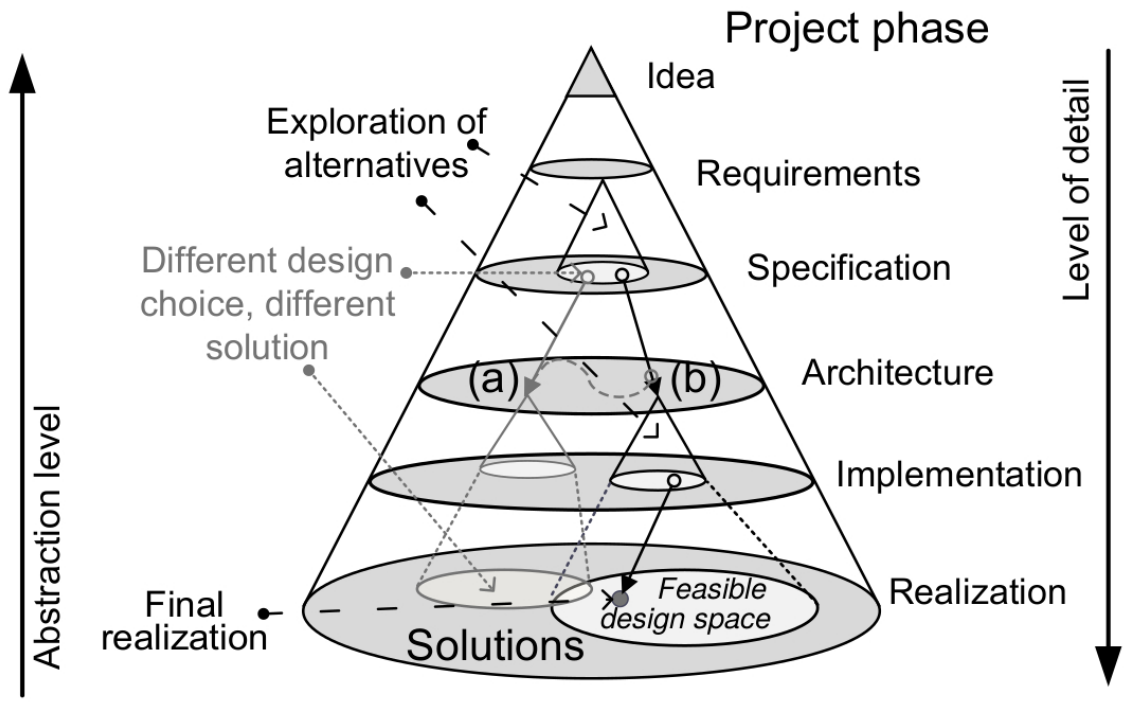

Figure 2.1: Design Pyramid with different abstraction levels, (Broenink et al., 2010a)

As explained before a cyber-physical system consists of multiple parts and these parts belong not to one domain. So after determining the requirements and specifications of the whole system it is possible to parallelize the design of the cyber-physical system. So the steps of the way of working can be described in more detail as can be seen in Figure 2.2. As can be seen from the figure, there are five starting points to start the design of a cyber-physical system.

Each starting point handles a particular domain of the design. For each starting point there are some steps that can be taken. The figure shows the four general steps:

- Step 1: *Top-level Architecture*, this step follows after determining the requirements and specifications of the whole system. In this step an overall architecture model of the system will be created. This model indicates how the system can be divided into multiple sub-systems. It also indicates how the different sub-systems communicate with each other.
- Step 2: *Detail Design*, in this step the different sub-systems which are defined in the previous step will be implemented by using the stepwise refinement manner.
- Step 3: *Implementation*, in this step the implementations of the different sub-systems will be connected to each other to get the complete model of the system. The requirements and specifications which are determined at the beginning of the design space will be verified by performing simulations on the model.
  - Step 3a: In this sub-step the software is tested in combination with the dynamic plant model to check whether the software behaves as intended. Depending on the results, the model can be fine tuned to get a better behaviour.
  - Step 3b: In this sub-step the real-time constraints will be included in the simulations. This is done by executing the software on the target computing platform. In this step only the dynamic plant model is used in the simulations.
- Step 4: *Realization*, in this step the real setup of the system will be realized. The real dynamic plant is connected to the target computing platform. By performing tests it can be verified if the real systems also satisfied the requirements en specifications of the system.

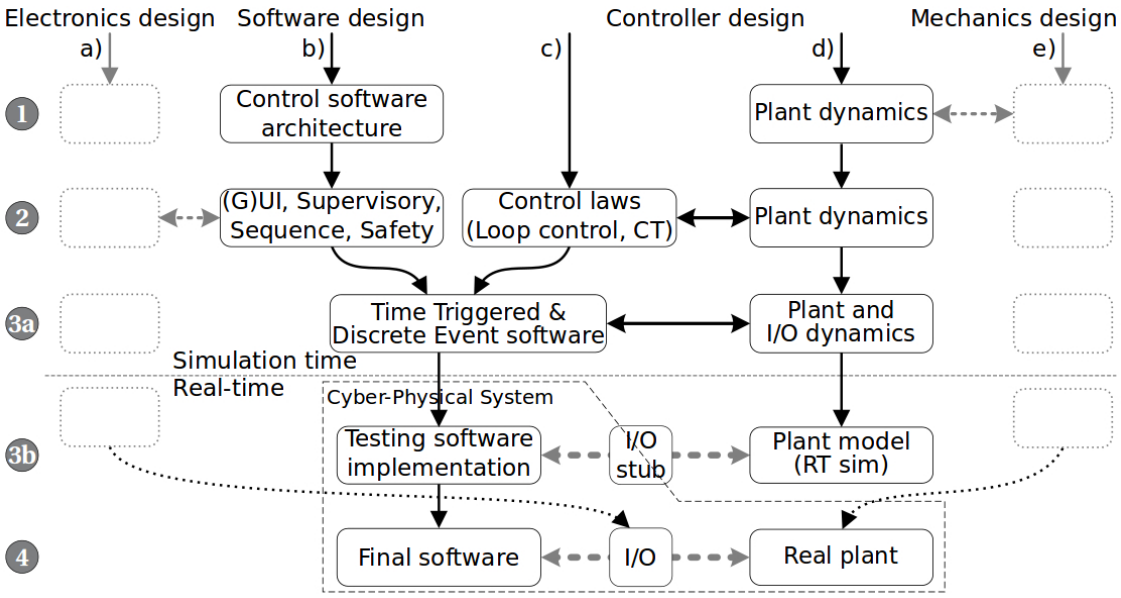

Figure 2.2: Steps of the way of working to design control software for cyber-physical systems, (Bezemer, 2013)

Ni describes in her thesis three different co-modeling approaches for cyber-physical systems (Ni, 2015). As explained in Chapter 1 a cyber-physical system can be divided into a *cyber* part

and a *physical* part. The *physical* part belongs most of the time to the continuous time (CT) domain. The *cyber* part belongs most of the time to the discrete time (DT) domain. The three different co-modeling approaches are related to the domain which is most important for the design of the cyber-physical system. The three different approaches are:

- *Dynamic-behaviour oriented:* In this approach the dynamic behaviour of the cyber-physical system is more of interest than the controller-logic behaviour. All the models are produced in the continuous time domain and later the controller part will be transformed into the discrete time domain to get a co-model.
- *Controller-logic oriented:* In this approach the controller part of the system is more of interest than the dynamic behaviour of the system. All the models are produced in the discrete time domain and later the models of the plant and analog interface are transformed to the continuous time domain to get a co-model.
- *Contract oriented:* In this approach, a contract has to be defined first as the start of the co-model development. The models are directly produced in the domain they belongs to.

These three approaches can be used during the development of a cyber-physical system. The dynamic-behaviour oriented approach starts at point *d* of Figure 2.2. The controller-logic oriented approach starts at point *b* and *c* of the same figure. This assignment focus on the contract oriented approach which starts at point *b* and *c* for the controller software which is created in TERRA. And the design of the physical system model starts add point *d* which is created in an existing modeling software suite for physical systems.

In step 3 of the way of working the different sub-systems are connected to each other. The whole systems will be simulated to test and verify the system. Co-simulation is used because every sub-system belongs to another domain and may have its own simulator. It is also impossible to directly connect a discrete-time model to a continuous time model. Broenink et al. (2010b) give an approach for co-simulation. In this approach a discrete-time simulator and continuous-time simulator are connected to each other using a co-simulation engine which synchronize the simulation time in both simulators. Both simulators simulates the models but on fixed times the inputs and outputs of both models are exchange with each other.

2.2 CPC Meta-model

All meta-models which are available in TERRA are in the end derived from the Component-Port-Connector (CPC) meta-model (Bruyninckx et al., 2013). Figure 2.3 shows an UML diagram representing a CPC meta-model. From these UML diagram the following properties holds for the CPC meta-model:

- A System is a collection of components and connectors.
- A Component provide the container for the functionality and behaviour.
- A Port gives access to the Component internals.
- A Component is configured by Properties.
- A Connection represent the interaction between to components.

Also the following constraints can be derived from the CPC meta-model:

- A System contains zero or more Components.

- A System contains zero or more Connectors.
- A Component contains zero or more Components.
- A Component can be contained by only one other Component
- A Component contains zero or more Properties.
- A Component contains zero or more Ports.
- A Port belongs to only one Component.
- A Connection is always between two Ports.

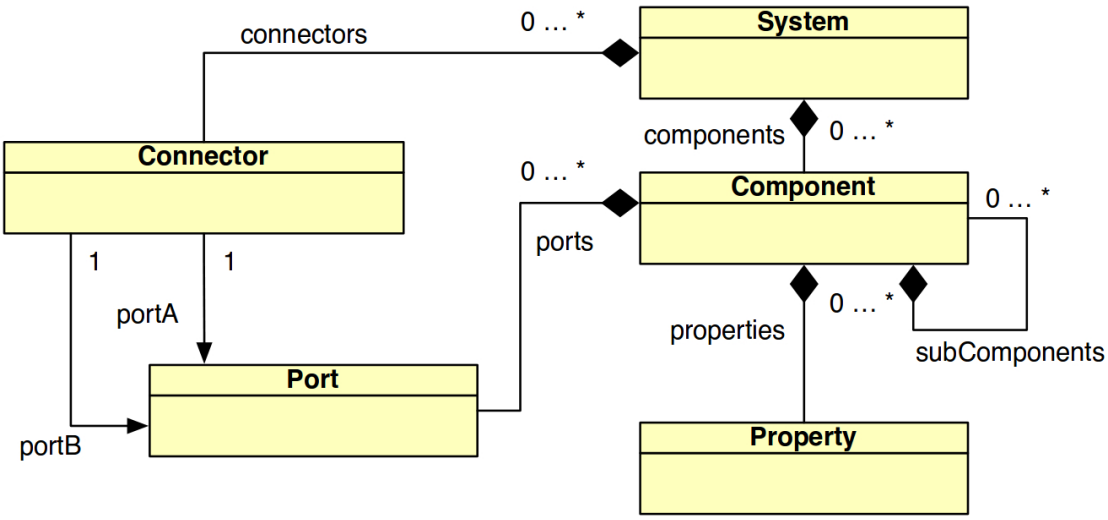

Figure 2.3: The UML diagram representing the CPC meta-model (Bruyninckx et al., 2013)

2.2.1 CPC Meta-model in TERRA

The CPC meta-model is implemented in TERRA because it is used to derive other meta-models from it. The implemented meta-model is more complex than the one shown in Figure 2.3. The explanation of the whole CPC meta-model implemented in TERRA is left out because it is too long and too complex. But parts of the meta-model which are important for this assignment will be explained.

The Port class of the CPC meta-model is called CPCPort in TERRA. This class has the following important attributes and references:

- **name:** A name for the port.
- **direction:** Indicates if the port is an input or an output.
- **link:** Reference to the link which is connected to the port.
- **unitType:** Reference to the unit type which is transported through the port.

The Port class contains a reference 'unitType' which refer to the class CPCUnitDescription. This class has the following important attributes:

- **name:** A name for the unit type.

- **type:** The data type of the unit, for example, real, boolean or integer.
- **unit:** Indicates the unit of the unit type, for example, m^2 or m*s.

The System class of the CPC meta-model is called CPCDiagram in TERRA. The Component class is called CPCModel in TERRA. There also exists a CPCExternalModel which is inheriting from CPCModel and only adds the possibility to add a file to the model. This file contains another model which can implement the CPCExternalModel class.

2.3 TERRA

As explained before TERRA is a model-driven design tool suite. TERRA is based on the way of working principles as explained in section 2.1. Currently TERRA contains a CSP editor and a basic architecture editor. These editors are used to design models which represents the cyber-physical system. When the model of the cyber-physical system is designed en simulated, TERRA can translate this model to actual execution code. TERRA uses the LUNA framework for this model-to-code translation.

2.3.1 CSP Editor

Communicating Sequential Processes (CSP) (Hoare, 1985) is a formal language for describing patterns of interaction in concurrent systems. CSP allows to divide a process in multiple sub-processes. The structure of these sub-processes can be mainly Sequential and Parallel. The Sequential structure indicates that one sub-processes must be finished before the next sub-process can be executed. The Parallel structure indicates that more than one sub-process can be executed at the same time.

Communication between the sub-processes is implemented using channel communication. This means that one sub-process can write on a channel by using a Writer and another sub-process which is connected to the same channel can read from the channel by using a Reader. The communication is based on rendezvous. This means that the Reader and Writer of one channel needs to be ready for execution to transfer the data from one sub-process to the other. This provides the possibility to synchronize sub-processes based on their communication flow.

The CSP editor in TERRA is used to create CSP models. These CSP models contains processes with channels between them. Each port of a process is connected to a Reader or a Writer to read from or write to a channel. It is also possible to include a C++ block into the CSP model. The implementation of the CPP file can be added by hand. Also a 20-sim block is available in the editor. This 20-sim block makes it possible to design a controller in 20-sim and include this controller into the CSP model.

TERRA can translate these CSP models into machine-readable CSP (CSPm) (Scattergood and Armstrong, 2011). A tool like FDR3[1] reads CSPm and can be used for formal verification of the CSP models. It checks for example, for dead-lock and live-lock. TERRA can also translate CSP models into executable code which can be executed on a real cyber-physical system.

2.3.2 Architecture Editor

As explained before the architecture editor is used for designing architecture models. An example of an architecture model of a generic cyber-physical system created in the architecture editor of TERRA is given in Figure 1.2. In this example the sub-systems are represented by rectangles and are called *components*. Each sub-systems has an interface for input and output. These inputs and outputs are visible as small squares which are visible in the components of the architecture model. These small squares are called *ports*. Different components can

[1] https://www.cs.ox.ac.uk/projects/fdr/, accessed on 8 April 2016.

communicate with each other by the channels which are between the different ports. These channels are the lines between the ports of different components.

TERRA can translate this architecture model into executable code. This model-to-code generation is based on the model-to-code generation for the CSP editor. This means that every component in the architecture model is seen as a CSP process which are connected to each other with channels. All the CSP processes which are generated by the architecture editor run in parallel structure.

2.3.3 LUNA

The LUNA Universal Network Architecture (LUNA) framework (Bezemer et al., 2011) is a component-based execution platform for cyber-physical systems. The core components take care of the platform related issues. The implementation of each core component depends on the used platform. High-level components take care of the platform-independent tasks, using the core components. One of these high-level components is the CSP execution engine. TERRA can transform CSP models into C++ code which make use of this CSP execution engine in LUNA.

2.4 Eclipse

As mentioned TERRA is a Model-Driven Development tool-suite which means it uses models to design cyber-physical systems. TERRA is developed in Eclipse[2]. Eclipse provides a modeling framework which can be used to create models. The Epsilon plugin is used in Eclipse to support model-to-model and model-to-text translations.

2.4.1 Eclipse Modeling Framework (EMF)

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data models (The Eclipse Foundation, 2016). EMF is a common standard for data models where many technologies and frameworks are based on. The core of EMF includes a meta-model called *Ecore* for describing models. The Ecore meta-model consists of classes which can contain attributes. It is possible to give dependencies between the different classes. EMF also supports the creation of editors for the models which are based on the Ecore meta-model.

2.4.2 Epsilon

Epsilon is a family of languages and tools for code generation, model-to-model transformation, migration and refactoring that work out of the box with EMF and other types of models (The Eclipse Foundation, 2014). Epsilon is supported by Eclipse. In this section only the languages which are used for the development of TERRA are mentioned.

- **Epsilon Object Language (EOL):** EOL is an imperative programming language for creating, querying and modifying EMF models. EOL forms the core of epsilon. EOL provides an extended feature set, which include the ability to update models, access to multiple models, conditional and loop statements, statement sequencing, and provision of standard output and error streams.
- **Epsilon Transformation Language (ETL):** ETL is a rule-based model-to-model transformation language built on top of EOL. This language can translate elements of a source model into elements in a target model. This language can be used if a model of one type needs to be translated into a model of another type.

[2] http://www.eclipse.org/, accessed on 24 July 2016.

- **Epsilon Validation Language (EVL):** EVL is a validation language built on top of EOL. This language can be used to specify constraints and check the model on these constraints.
- **Epsilon Generation Language (EGL):** EGL is template-based model-to-text language for generating code, documentation and other textual artefacts from models. EGL use a template file which specify what kind of code is generated and it can include values from models into the code.

2.5 Functional Mock-up Interface

The Functional Mock-up Interface (FMI) is a tool-independent standard to support both model exchange and co-simulation of dynamic models (FMI-standard, 2016). A model using this standard is distributed in one zip file called Functional Mock-up Unit (FMU). The FMU contains the following files:

- An XML file containing the definition of all exposed variables in the FMU and other static information about the model.
- Source files which contains a small set of easy to use C functions. These C function implements all needed model equation or the access to co-simulation tools.
- Further data can be included in the FMU zip file, especially a model icon, documentation files, maps and tables needed by the FMU, and/or all object libraries or dynamic-link libraries that are utilized.

The FMI makes it possible to exchange models between different tools. This means that a model which is created in tool A can be used in tool B for simulations. With the FMI it is also possible to connect different simulation tools together. This means that different models can be simulated in their own simulator and that the inputs an outputs of the different models are exchange between the different simulators.

FMI for Co-simulation can be used in two ways. The first way is that the simulation tool generates an FMU which includes the model dynamics and the solver for the model. This situation is shown in Figure 2.4. This means that for the simulation only the generated files are necessary and not the simulation tool itself. The second way is that the simulation tool generates an FMU which only contains a FMI Wrapper which communicates with the simulation tool. This situation is shown in Figure 2.5. This means that for the simulation of the model the corresponding simulation tool is used. For both ways the interface for the FMI is the same.

In both figures a *Master* process is visible which is connected to the FMU through the FMI interface. This process controls the FMUs and needs to be created by the user itself. This process takes care of the exchange of variables between different FMUs and also controls the simulation of the FMU.

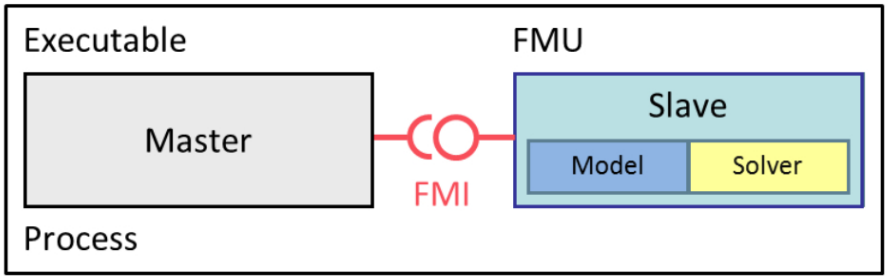

Figure 2.4: Co-simulation with generated code (FMI-standard, 2014)

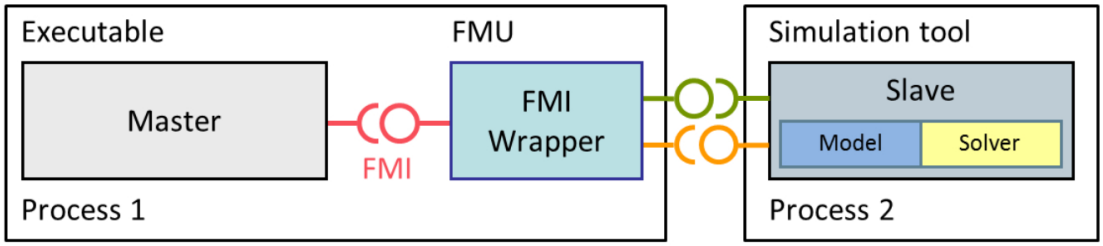

Figure 2.5: Co-simulation with tool coupling (FMI-standard, 2014)

3 Analysis

3.1 Architecture Modelling in TERRA

In Figure 3.1 the same figure is shown as in chapter 1. As explained before the goal of the architecture editor is to draw a model like the one which is shown in Figure 3.1. There is a controller block, a I/O block and a plant block. The controller block contains the models which will finally contains the implementation of the control software. For the implementation of this block the CSP editor in TERRA can be used. These CSP models are translated to C++ code which runs in combination with the LUNA framework. In general the CSP models contain controllers which directly control the actuators of the plant. These are hard real-time tasks. Currently also some research has been done to connect the ROS framework[1] to the LUNA framework (van der Werff, 2016). ROS contains lot of algorithms which are useful for controlling cyber-physical systems. Most of these algorithms contains soft real-time tasks. So it can be useful to split the controller block into two blocks. One block contains the soft real-time tasks and the other block contains the hard real-time tasks. Figure 3.2 shows the architecture model where the controller block is divided into the two blocks.

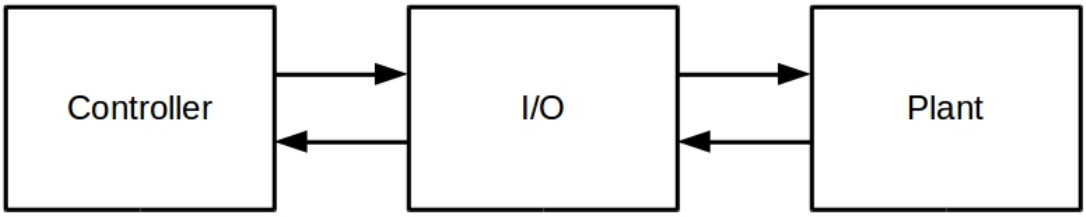

Figure 3.1: Global architecture model overview

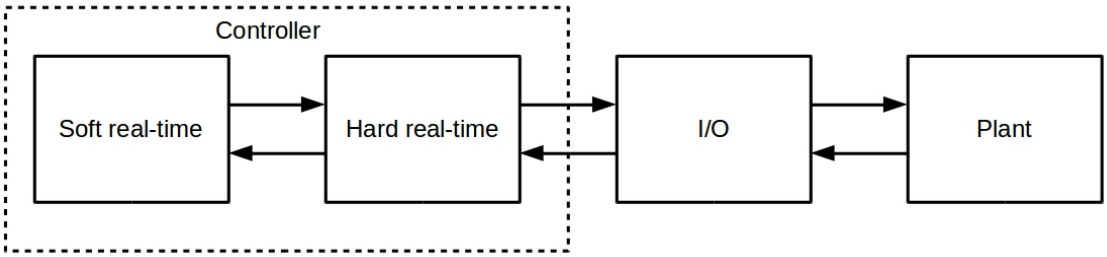

Figure 3.2: Global architecture model overview with divided controller

The I/O block contains the hardware interface from and to the controller block. For simulation purposes this block contains models of the hardware ports because the abstract functionality of these ports are important for the behaviour of the total cyber-physical system. It is not always possible to directly connect a signal from the software to an input or output of a physical system. For example when an motor needs to be driven in a physical system, this motor needs most of the time a pulse-width modulation (PWM) signal. The control software only calculates the direction, frequency, and duty cycle of this PWM signal. The PWM signal itself is generated by hardware. So for simulation purposes it is important to convert the signals between the controller and the physical system. Because the signals from and to the software are discrete time

[1] http://www.ros.org/, accessed on 25 July 2016.

or events and the signals from or to the plant are continuous time signals, the I/O block can be divided into two parts as is shown in Figure 3.3. The I/O block can be divided into a discrete time/events I/O block and a continuous time I/O block.

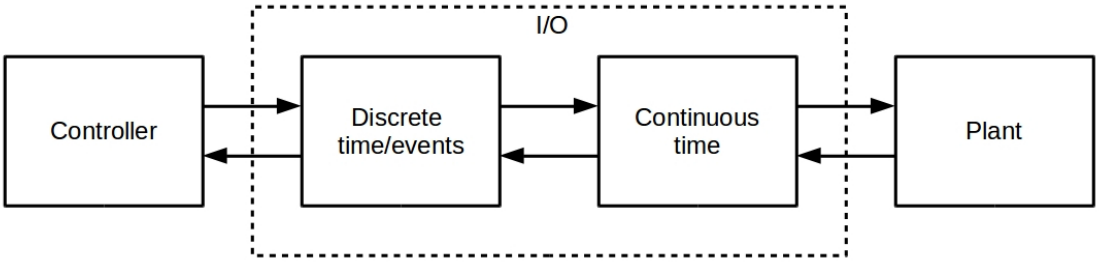

Figure 3.3: Global architecture model overview with divided I/O block

At the moment all models which are created in TERRA are translated to C++ code which runs in combination with the LUNA framework. This executable code runs most of the time on an embedded processor. Instead of translating the CSP models to software, it is also possible to translate CSP models to a hardware discription (Kuipers et al., 2016). The parallel nature of the FPGA make archiving hard real-time guarantees more easy. The functional language Cλash can be used to describe the mapping between CSP to hardware. This means that parts of the controller and I/O of the cyber-physical system model can be translated to Cλash instead of LUNA compatible code.

As explained in the way of working in Section 2.1 the controller of the cyber-physical system should first be tested on a plant model before the real plant setup is used. The plant block of the architecture model contains the model of the physical system which can be used for simulations without using the real physical system.

3.2 Using a Plant Model in the Architecture Editor

The focus of this assignment is on the use of a model of a physical system in the architecture editor of TERRA. It is not necessary to make a new implementation tool for physical systems in TERRA because there exist enough tools which can do that. These tools also contain simulators for the simulation of the physical systems, so it also not necessary to create a new simulator for physical systems in TERRA. This assignment is about using a tool-suite for physical systems in combination with TERRA. 20-sim[2] is a tool-suite for physical system which is most used at the chair of Robotics and Mechatronics and therefore this tool-suite will be used in combination with TERRA. The use of 20-sim in TERRA can be divided into different steps which are similar to step 1 to 3 of the way of working for cyber-physical systems. The use of 20-sim in TERRA for the different steps are explained in the next sub-sections.

3.2.1 Step 1 - Top-level Architecture

In this step the top-level architecture model of the cyber-physical system is designed. This means that the cyber-physical system is divided into different subsystems. The physical system is one of the sub-systems of a cyber-physical system. This system can also be divided into different subsystems. For every subsystem in the architecture model an interface is defined for the communication between the differen subsystems. This interface indicates what kind of signals are used for the communication between the different subsystems. Every subsystem can be implemented in its on way but the interface needs to be the same because then it is possible to directly connect the different subsystems together.

[2] http://www.20sim.com/

In this case the implementation of the controller is done in the CSP editor of TERRA while the implementation of the physical system is done in 20-sim. So it is important to know which signals are transfered between the controller and the physical system to connect these different systems together. It is possible to define the interface of both systems on paper and directly implement these interfaces into their own editors. This approach is very error prone because the implementation of the interface is done by hand.

A better approach is to define the interfaces between the different sub-systems in an editor and automatically transfer the interfaces for the different sub-systems into their own editors. At the moment the architecture editor of TERRA supports the design of an architecture model and to translate the interfaces for the different components to CSP models. This approach can also be used to transfer the interface of a component in the architecture editor of TERRA to the 20-sim editor. Figure 3.4 shows an example of an interface of a physical system model in 20-sim. From the architecture editor of TERRA this kind of interface needs to be created for 20-sim.

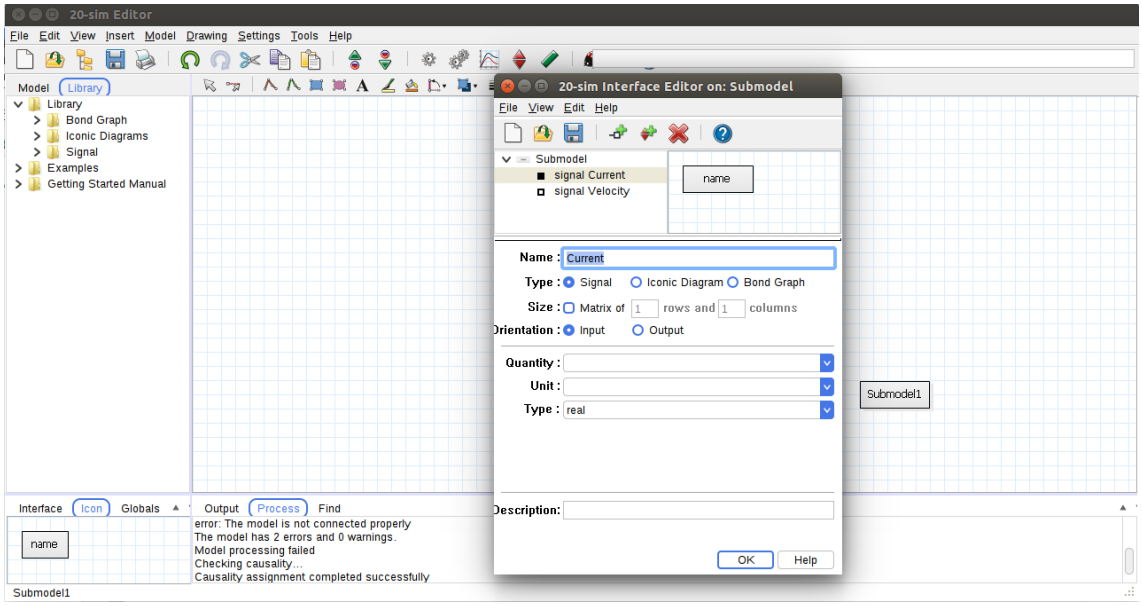

Figure 3.4: Example of a model with an interface in 20-sim

There are two approaches to copy the interface from TERRA to 20-sim. The first approache is to generated some template files with a model with different interfaces in 20-sim. 20-sim has an API which makes it possible to open the 20-sim tool and load a 20-sim file. So it should be possible to use this API in TERRA to select the right template file for the interface of the plant model which can be automatically opened in 20-sim. The problem with this approach is that a new template file needs to make in 20-sim when the interface is not available in the current selection of templates files. The interfaces of physical systems can variate a lot, so it means that for every kind of physical system a new template file is needed.

The second approach is to create an 20-sim file by TERRA itself. A 20-sim file describes the interface for a model, so TERRA can generated a 20-sim file which also describes an interface for a model. This interface is similar to the interface of the model for the physical system in the architecture editor. For this second approach it is necessary to define a meta-model for this interface which can be used by TERRA to translate the interface to a 20-sim file. This approach is very flexible but is more complex to implement in TERRA than the first approach. Because flexibility is more important for the end-user than the complex implementation by the developer, the second approach is performed.

3.2.2 Step 2 - Detail Design

In this step of the way of working all the different components of the architecture model gets an implementation. For the physical system model holds that the model with the generated interface by TERRA can be implemented by 20-sim. The implementation of the plant model can consists of bond graphs or/and iconic diagrams.

3.2.3 Step 3 - Implementation

In this step all the different implemented models are connected together for simulations. For the simulation the controller is designed in TERRA while the model of the physical system is designed by 20-sim. So an approach is needed to connected the model in TERRA with the model in 20-sim for co-simulation. Three approaches are possible for this connection.

The first approach is to use the API of 20-sim. The API of 20-sim also has some functionalities to preform some simulations on the model in 20-sim. It is also possible to exchange values from and to the model in 20-sim through the API. So it should be possible to make a connection to this API in TERRA to control the simulator of 20-sim and to exchange values like inputs and outputs between the controller model and the physical system model. A advantage of this approach is that the simulation of the physical system model runs in 20-sim itself and it can also generates plots from the simulation of the physical system model. A disadvantage of this approach is that the API works with python but TERRA only generates C++ code based on LUNA at the moment. So support for using python needs to be implemented in LUNA.

The second approach is to use the current implementation of model exchange in 20-sim which is created by the chair of Robotics and Mechatronics. This model exchange is currently used to transfer controller implementations from 20-sim to the CSP editor of TERRA. This method generated a XML file which contains information about the 20-sim model and reference value for getting and setting input and output values of the 20-sim model. Also some source codes are generated which contains the dynamic implementation of the controller models which are used to implement the controller in TERRA. A disadvantage of this approach is that the 20-sim simulator itself is not used for the simulations and so there is also no support to draw graphs from the simulation of the 20-sim model.

The third approach is to make use of the FMI for Co-simulation. It is possible to create a FMU of a model in 20-sim. At the moment only FMU's with the solver included is supported in 20-sim, see Figure 2.4. Later it should also be possible to use 20-sim as simulation tool for the FMU. Because the FMI is a standard it should also be possible to connect FMU's from other tools to TERRA. The FMU also consists of an XML file with important information about the model itself and reference values for getting and setting input and output values of the 20-sim model. When 20-sim can also be used as co-engine for the simulation of the FMU, it is also possible to draw graphs from the simulation of the 20-sim model. Because this approach has the most advantages in comparing with the other two approaches, this approach is used for this assignment.

For the implementation of the third approach it is important that the interface from the controller software to the FMU which contains the implementation of the physical system model is automatically generated to prevent errors. So also for this approach a meta-model for the FMU interface can be designed which can be used to automatically generate code for the FMU interface. The generated XML file from the FMU can be used as base for the implementation of the FMU interface because this file already contains information about the model, the inputs and the outputs which are necessary for the FMU interface.

Figure 3.5 gives an impression about the way of working for the co-modeling and co-simulation support for physical systems in TERRA. The way of working starts with the architecture model in the architecture editor of TERRA. Step 1 is to copy the interface of the physical system model

to 20-sim. Step 2 gives an implementation to the physical system model using the interface which is generated from TERRA. In step 3 a coupling is created between the controller software in TERRA and the simulator of the physical system in 20-sim. So the plant block of the architecture model in TERRA contains now an interface to the FMU of the physical system model.

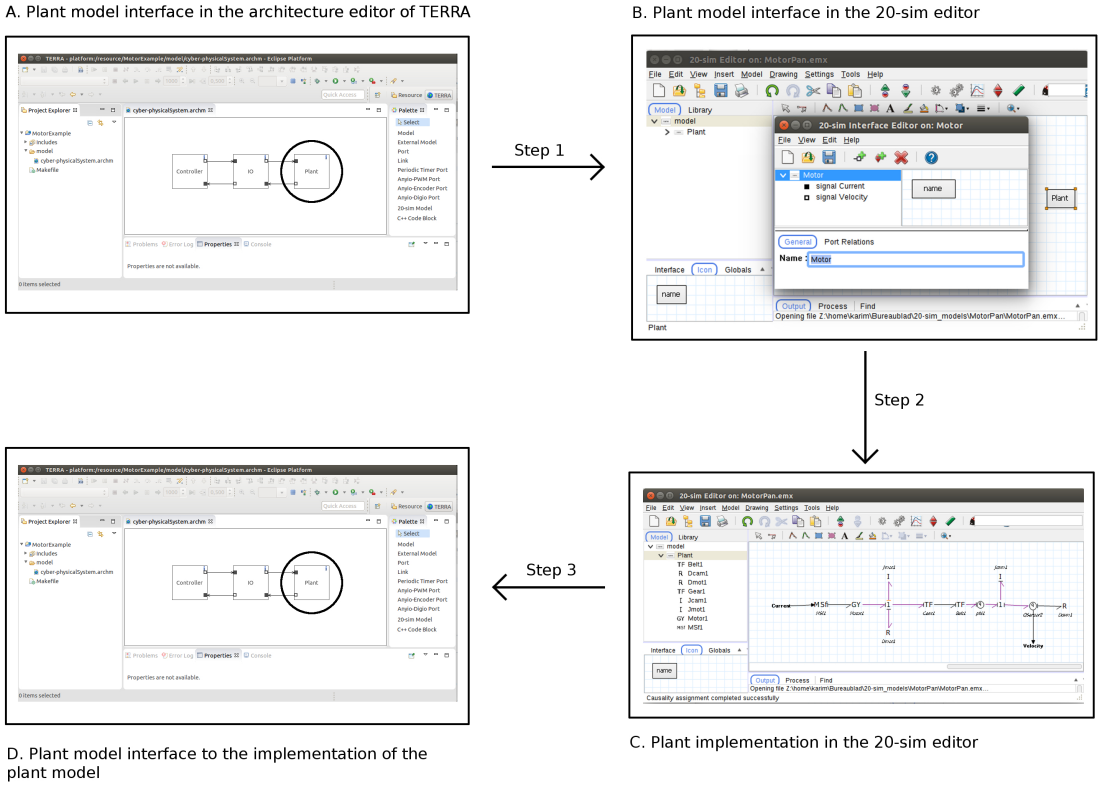

Figure 3.5: Co-modeling and co-simulation support impression for physical systems in TERRA

3.3 Requirements

This section describes the different requirements which are formulated for this assignment. The MoSCoW method is used to indicate the priority of each requirement.

A Must have

I All meta-models must be based on the CPC meta-model

All Terra models are in the end derived from the CPC meta-model. By using the CPC meta-model as base for the architecture models, it becomes possible to easily connect different architecture models to each other but also to connect other model types which are used in Terra.

II No code must be added by hand

The goal of model-driven development is to use models to create a working system. These models can be translated to other model types or code for simulations and deployment. The creation and translation of these models must be done without entering code by the user. This will prevent errors and finally in the realization of the system.

III The architecture editor must support the 20-sim tool

The 20-sim tool is used to use the plant model in combination with the architecture editor of TERRA. The design and simulation of the plant model is done in 20-sim itself while there is a connection with TERRA.

IV The architecture editor must support code generation for the LUNA framework
Currently it is possible to generate code for the LUNA framework in TERRA. So it must also be possible to generate code for the LUNA framework from architecture model in TERRA. All the models and interfaces which are used in the architecture model need to be translate to LUNA code, so it can be used for execution.

V Testing of the models must be supported by the architecture editor
To prevent errors in the design of the architecture model and finally in the realization of the system, it must be possible to detect errors in an early state. So the architecture editor needs to check the constraints of the models

B Should have

I The architecture editor should have the possibility to switch between the real plant and a model of the plant.
As explained in the way of working for designing cyber-physical systems first the software implementation is simulated with a model of the plant before it will be tested on a real plant. So the architecture editor must support the switch between the model of the plant which is created in 20-sim and the real plant which is connected through the interface of the computer or the embedded platform.

II The architecture editor should have support for hardware ports simulation
Hardware ports like A/D converters, D/A converters and PWM ports are used in cyber-physical systems for the connection between the embedded platform and the real plant. During simulations the embedded software is not directly tested on the dedicated platfrom with the real plant connected to it. First the embedded software is tested on a general computer with a model of the plant. Hardware ports like A/D converters and D/A converters are implemented in the hardware of the dedicated platform. So when simulations are done on a general computer these hardware ports need to be simulated to connect the model of the plant correctly to the embedded software.

III The architecture editor should have support for simulations on the general computer platform and on a dedicated embedded platform in combination with a model of the plant
As explained in the previous requirement the implementation of the software will first be tested on a general computer in combination with the model of the plant. After the tests on the general computer the embedded software will be tested on the dedicated embedded platform. Also for these tests first the model of the plant is used. Because the resources on the embedded platform are limited it is not recommend to also simulate the model of the plant on the platform. So this need to be done on a general computer.

C Could have

I The architecture editor could have support for code generation to Cλash
At the moment of writing, research is performed to translate CSP models to Cλash. So it will be possible to also run CSP models on FPGAs. So in the future TERRA could also support code generation to Cλash.

II The architecture model could have support for the ROS framework
At the moment of writing, research is performed to create a link between the LUNA framework and the ROS framework. Also the ROS framework can be used to create software for cyber-physical systems. So the architecture model could also support ROS models.

D Won't have

I The architecture editor won't have support for multiple plant models
The goal of this assignment is to first add one plant model in the architecture editor. Later on it should be possible to add functionalities to add multiple plant models into the architecture editor

4 Design of the Meta-models for Co-modeling and Co-simulation Support in TERRA

Because MDD is used for the design of cyber-physical systems, meta-models are used as base for the different models which can be created by TERRA. For the translation between the interface for the plant model, which is created in the architecture editor, to the 20-sim model, model-to-text transformation is needed. To perform this operation in TERRA, a meta-model of the plant interface in TERRA is needed. The design of this meta-model is discussed in Section 4.1.

When the plant model is designed in 20-sim, this implementation needs to be coupled to TERRA to perform co-simulations. FMI is used to setup a communication channel between TERRA and 20-sim. To generate an interface to the FMU of the plant model in TERRA, a meta-model needs to be defined for the FMU interface. This is explained in Section 4.2

4.1 Meta-model for Co-modeling Support in TERRA

To translate the plant model interface created in the architecture editor of TERRA to an interface in 20-sim, a 20-sim file with this interface needs to be created by TERRA. First a 20-sim file is analyzed to see which properties are important to generate a 20-sim file. These properties are used to generate a meta-model for the plant model interface which can be used to generated a 20-sim file. Finally an implementation approach in TERRA is proposed.

4.1.1 Analysis of the 20-sim File

Listing 4.1 shows the source of the 20-sim file which corresponds with the 20-sim model which is shown in figure 4.1. Line 3 to 15 describes the properties of the main model. This main model is the workspace which is visible in the 20-sim editor. Line 16 to 55 describes the implementation of this main model. The implementation of the main model consists of a submodel with the name 'plant' which is also visible in the 20-sim editor. Line 19 to 39 describes the properties of the submodel. Line 42 to 45 describes the ports which are included in the submodel. The port description has the following layout:

signal <dataType> <direction> <name>;

where <*dataType*> represents the datatype of the variable which can be send through the port, <*direction*> represents the direction of the port and <*name*> represents the name of the port. When <*dataType*> is replaced by nothing the datatype will be a real. Replacing <*dataType*> by *integer* the datatype will be an integer. The datatype will be a boolean when <*dataType*> is replaced by *boolean*. And finally the datatype of a port will be a string when <*dataType*> is replaced by *string*. To give a direction to a port, <*direction*> can be replaced by *in* when the port is an input and <*direction*> can be replaced by *out* when the port is an output.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Document>
3  <Model version="4.6" build="4.6.1.6898">
4   <Sidops><![CDATA[model 128 184
5    description '<Information>
6     <Description>
7      <Version>4.6</Version>
8      <IsMainModel>0</IsMainModel>
```

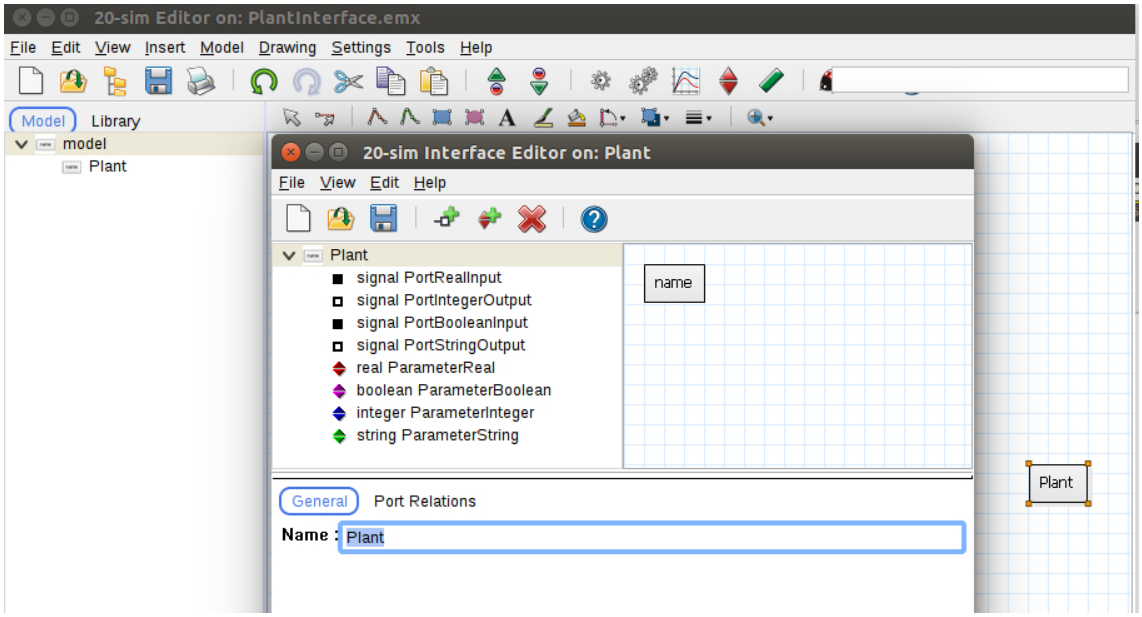

Figure 4.1: Example of a interface of a submodel in 20-sim

```
9       <KeepParameterValues>False</KeepParameterValues>
10      <LibraryPath>Z:\home\karim\Bureaublad\PlantInterface.emx</
            LibraryPath>
11      <TimeStamp>2016-7-27 13:08:50</TimeStamp>
12     </Description>
13    </Information>';
14    type Mainmodel
15    end;
16    implementation bg
17     submodels
18      Plant 632 280
19       description '<Information>
20        <Description>
21         <Version>4.0</Version>
22         <LibraryPath>C:\Program Files\20-sim 4.0\System\Submodel.
               emx</LibraryPath>
23         <TimeStamp>2007-10-31 11:32:54</TimeStamp>
24         <IsMainModel>1</IsMainModel>
25         <KeepParameterValues>False</KeepParameterValues>
26         <AllowLibraryUpdate>True</AllowLibraryUpdate>
27         <Configuration>
28          <struct>
29           <member>
30            <name>DocumentationMask</name>
31            <value>
32             <struct>
33             </struct>
34            </value>
35           </member>
36          </struct>
```

```
37          </Configuration>
38         </Description>
39        </Information>';
40        type Plant
41         ports
42          signal in PortRealInput;
43          signal integer out PortIntegerOutput;
44          signal boolean in PortBooleanInput;
45          signal string out PortStringOutput;
46         parameters
47          real ParameterReal = 1.0;
48          boolean ParameterBoolean = 1.0;
49          integer ParameterInteger = 1.0;
50          string ParameterString = 1.0;
51        end;
52      end;
53      connections
54      end;
55     implementation_end;
56    ]]>
57    </Sidops>
58  </Model>
59 </Document>
```

Listing 4.1: Textual representation of a 20-sim file

Line 47 to 50 describes the parameters which are included in the submodel. The parameter description has the following layout:

<dataType> <name> = <value>;

where <*dataType*> represents the datatype of the parameter, <*name*> represents the name of the parameter and <*value*> represent the value of the parameter. When <*dataType*> is replaced by *real* the datatype will be a real. Replacing <*dataType*> by *integer* the datatype will be an integer. The datatype will be a boolean when <*dataType*> is replaced by *boolean*. And finally the datatype of a parameter will be a string when <*dataType*> is replaced by *string*.

4.1.2 Plant Model Interface Meta-model

Figure 4.2 shows the meta-model for the plant model interface. The class *PlantModel* represents the plant model itself. This class has an *name* attribute which represents the name of the model. The value of this *name* attribute can be used during code generation to add the name of the model in line 18 and 40 of listing 4.1. The *PlantModel* class contains one or more ports. These ports contain a *name* attribute, a *direction* attribute and a *dataType* attribute. The values of these attributes can be used during model-to-text generation to generate lines 42 to 45 of listing 4.1 to add ports to the submodel in 20-sim.

The *PlantModel* class also contains four different types of parameters. The current version of the architecture editor in TERRA has no support for parameter exchange for the current supported models. But this feature can make it possible to use parameter values of the plant model as value or part of an equation in the controller model or the other way around. Keeping the use of this feature in mind it is useful to also add these parameters in the meta-model of the

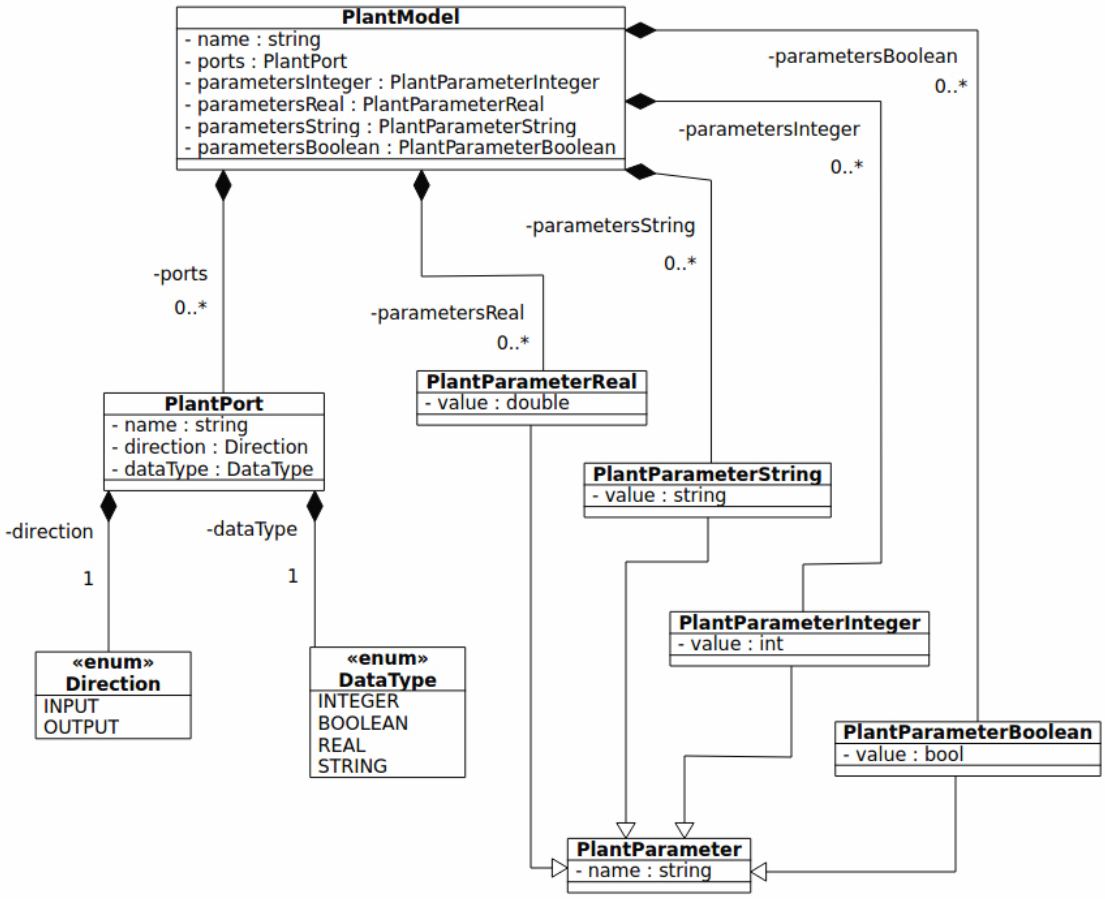

Figure 4.2: Meta-model of the plant model interface

plant model interface. Parameter values are from different data types like boolean or integer. So for every data type, a separate set of parameters is implemented in the *Plant* model.

Each parameter contains a *name* attribute and a *value* attribute. The values of these attributes can be used during model-to-text generation to generate lines 47 to 50 of listing 4.1 to add parameters to the submodel in 20-sim. Of course the data type of the parameter can be determined by looking where the parameter is located in the *Plant* class.

The rest of the 20-sim file which is visible in listing 4.1 is standard for every 20-sim file. So it is not necessary to include these values in the meta-model because they do not change for a different model interface.

4.1.3 Using the Meta-model in TERRA

To use the meta-model in the architecture it needs to be based on the CPC meta-model. The class *CPCModel* of the CPC meta-model is similar to the class *PlantModel*. So the class *PlantModel* can inheritance from the class *CPCModel*. The class *CPCPort* of the CPC meta-model is similar to the class *PlantPort*. So the class *PlantPort* can inheritance from the class *CPCPort*. This is shown in Figure 4.3.

There are two approaches to use the meta-model of the plant interface model in the architecture editor of TERRA. The first approach is to create an editor around the plant model interface meta-model. When an empty model is added in the architecture editor of TERRA, the CSP editor is automatically opened when double clicking on the empty model. When also some ports are added to the empty model, these ports are also automatically added to the CSP model when

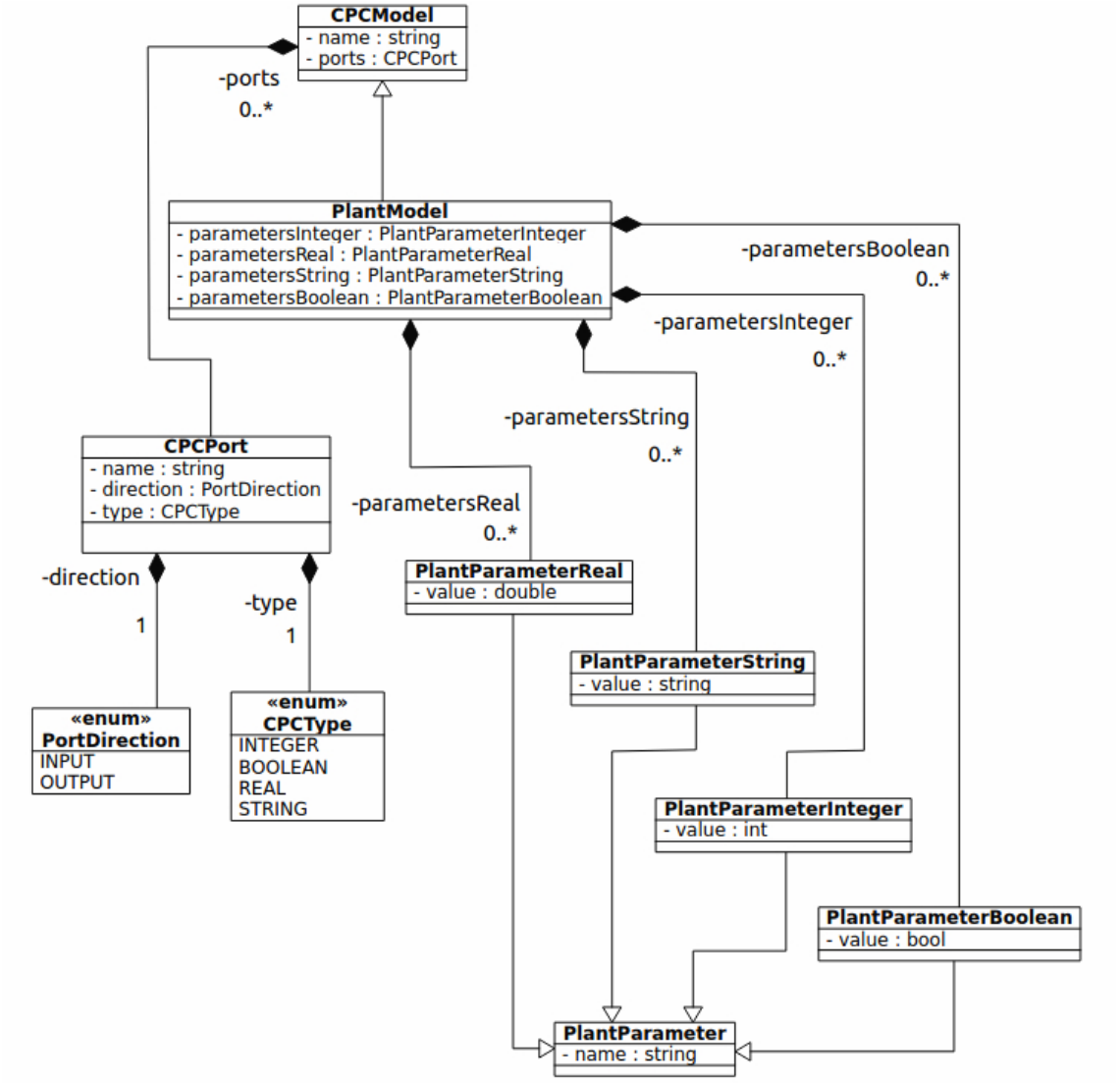

Figure 4.3: Meta-model of the plant model interface with inheritance from the CPC meta-model

double clicking on the model. It should be possible to make a choice between different editors when double clicking in an empty model in the architecture editor. If a empty model in the architecture model represents a plant model, the editor for the plant model interface can be opened when double clicking on the empty model. If already ports are added to the empty model, these ports can also automatically added to plant model interface. Then it is only necessary to add some parameters if necessary to the model before the 20-sim file is generated from the editor. Like the CSP editor it should be possible to design a plant model interface with the editor. This editor adds ports and parameter to the model and then use model-to-text generation to translate the model to a 20-sim file.

The second approach is to directly generated a 20-sim file from the architecture editor. The meta-model of an architecture component contains the same attributes as the *PlantModel* class of the plant model interface meta-model. The same holds for the architecture port and the *PlantPort* class. Only the parameters from the *PlantModel* class can not be created from the architecture editor. It is possible to add the parameter classes to the model class of the architecture meta-model. This should be the beginning to create a parameter sharing feature between the different components of the architecture model as mentioned in Section 4.1.2.

The preference is to implement the second approach into the architecture editor. The second approach gives less overhead than the first approach because in the first approach the architecture model file contains mostly the same content as the plant interface model file generated by the plant interface model editor. Only the parameters of the plant model are added. This is also a disadvantage of the second approaches because the parameters needed to be added in the architecture editor. But this can be a start of the development of a new feature of the architecture editor. It is also possible to remove the parameters from the meta-model when they are not shared with other models in the architecture model. With the second approach it is also not necessary to create a new editor in TERRA.

At this moment, due to time constraints, it was not possible to actually add this implementation in the architecture editor.

4.2 Meta-model for the Co-simulation Support in TERRA

To use a plant model in 20-sim in combination with TERRA for co-simulation, an interface needs to be created to the FMU generated by 20-sim according to the FMI standard. First FMI for co-simulation is analyzed to determine the important data which is needed to generated an interface to the FMU. A meta-model is created which contains all the necessary data for the interface. Finally an implementation approach in TERRA is proposed.

4.2.1 Analysis of FMI for Co-simulation

Following the FMI for Co-simulation standard, a FMU can be generated from a plant model which contains an XML file and some source files. The XML file contains information about the model itself and about how it can be simulated. The source files contains C files which implement the FMI functions with the plant dynamics or a interface to a simulation tool.

Listing 4.2 shows a master algorithm for a co-simulation between two FMU's. Figure 4.4 shows the connection graph of the two FMUs which are used in the master algorithm. Line 5 to 10 shows the creation of the struct *fmi2CallbackFunctions*. This struct provides callback functions to be used from the FMU functions to utilize resources from the environment. The important functions are: *logger*, *allocateMemory* and *freeMemory*. The *logger* function is called in the FMU, usually when the execution of a FMI function behaves not as desired. It is also used for information messages. The *allocateMemory* function is called in the FMU if memory needs to be allocated. The *freeMemory* function is called in the FMI if memory is freed that has been allocated with *allocateMemory*. The pointers to the *componentEnvironment* and *stepFinished* are optional and not used. It is possible to include these pointer at a later time when they are necessary.

```
1 //////////////////////////
2 //Initialization sub-phase
3
4 //Set callback functions,
5 fmi2CallbackFunctions cbf;
6 cbf.logger = loggerFunction;       //logger function
7 cbf.allocateMemory = calloc;
8 cbf.freeMemory = free;
9 cbf.stepFinished = NULL;           //synchronous execution
10 cbf.componentEnvironment = NULL;
11
12 //Instantiate both slaves
13 fmi2Component s1 = s1_fmi2Instantiate("Tool1", fmi2CoSimulation,
       GUID1, "", fmi2False, fmi2False, &cbf, fmi2True);
```

```
14 fmi2Component s2 = s2_fmi2Instantiate("Tool2", fmi2CoSimulation,
       GUID2, "", fmi2False, fmi2False, &cbf, fmi2True);
15
16 if((s1 == NULL) || (s2 == NULL))
17     return FAILURE;
18
19 // Start and stop time
20 startTime = 0;
21 stopTime = 10;
22
23 //communication step size
24 h = 0.01;
25
26 // set all variable start values (of "ScalarVariable / <type> /
       start")
27 s1_fmi2SetReal/Integer/Boolean/String(s1, ...);
28 s2_fmi2SetReal/Integer/Boolean/String(s2, ...);
29
30 //Initialize slaves
31 s1_fmi2SetupExperiment(s1, fmi2False, 0.0, startTime, fmi2True,
       stopTime);
32 s2_fmi2SetupExperiment(s2, fmi2False, 0.0, startTime, fmi2True,
       stopTime);
33 s1_fmi2EnterInitializationMode(s1);
34 s2_fmi2EnterInitializationMode(s2);
35
36 // set the input values at time = startTime
37 s1_fmi2SetReal/Integer/Boolean/String(s1, ...);
38 s2_fmi2SetReal/Integer/Boolean/String(s2, ...);
39
40 s1_fmi2ExitInitializationMode(s1);
41 s2_fmi2ExitInitializationMode(s2);
42
43 ////////////////////////
44 //Simulation sub-phase
45
46 tc = startTime; //Current master time
47
48 while((tc < stopTime) && (status == fmi2OK))
49 {
50     //retrieve outputs
51     s1_fmi2GetReal(s1, ..., 1, &y1);
52     s2_fmi2GetReal(s2, ..., 1, &y2);
53
54     //set inputs
55     s1_fmi2SetReal(s1, ..., 1, &y2);
56     s2_fmi2SetReal(s2, ..., 1, &y1);
57
58     //call slave s1 and check status
59     status = s1_fmi2DoStep(s1, tc, h, fmi2True);
60     switch(status) {
```

```
61      case fmi2Discard:
62          fmi2GetBooleanStatus(s1,fmi2Terminated, &boolVal);
63          if(boolVal == fmi2True)
64              printf("Slave s1 wants to terminate simulation.");
65      case fmi2Error:
66      case fmi2Fatal:
67          terminateSimulation = true;
68          break;
69      }
70      if(terminateSimulation)
71          break;
72
73      //call slave s2 and check status as above
74      status = s2_fmi2DoStep(s2, tc, h, fmi2True);
75      ...
76
77      //increment master time
78      tc += h;
79  }
80
81  ////////////////////////
82  //Shutdown sub-phase
83  if((status != fmi2Error) && (status != fmi2Fatal))
84  {
85      s1_fmi2Terminate(s1);
86      s2_fmi2Terminate(s2);
87  }
88
89  if(status != fmi2Fatal)
90  {
91      s1_fmi2FreeInstance(s1);
92      s2_fmi2FreeInstance(s2);
93  }
```

Listing 4.2: A simple master algorithme for FMI

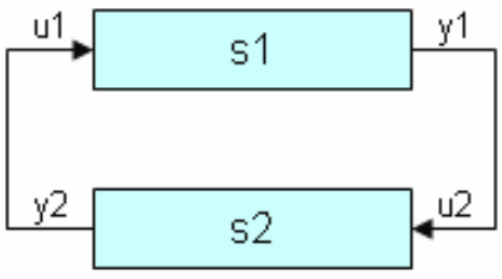

Figure 4.4: Connection graph of the slaves of the master algorithm in Listing 4.2

Line 13 and 14 instantiate the FMU's for co-simulation. The first argument of the function *fmi2Instantiate* is *instanceName*. This argument is an unique identifier for the FMU instance. The second argument *fmuType* is used if the FMU is used for model exchange or co-simulation. The third argument *fmuGUID* is a string which is used to check if the used XML file is compat-

ible with the source code which is provided by the FMU. So the value for this argument can be taken from the corresponding XML file. The fourth argument *fmuResourceLocation* can contain a path to the resource folder which is also located in the FMU directory. The fifth argument *functions* provides the callback functions which are discussed before. The sixth argument *visible* defines that the interaction with the user should be reduced to a minimum. The last argument *loggingOn* sets the debug logging.

Line 16 and 17 checks if the FMU's are correctly instantiated. Line 16 to 24 define the start time and stop time of the simulation. It also defines the step size for the simulation. The preferred step size can be found in the XML file of the FMU. Line 27 and 28 sets all the variable start values. It is not necessary to perform this step because the FMU which is generated by 20-sim also contains the standard values in the source code itself. But if not the standard values are used then these values can be set by using the following function:

fmi2Status fmi2SetXX(fmi2Component c, const fmi2ValueReference vr[], size_t nvr, fmi2Real value[]),

where *XX* needs to be replaced by *Real*, *Integer*, *Boolean* or *String* depending on the datatype of the variable which needs to be set. All variables with the same datatype can be set with one function call. The first argument *c* of the function is the reference to the instance of the FMU for which the variables need to be set. The second argument *vr[]* is a vector of reference values to the variables which needs to be set. The reference values of the variables can be found in the XML file. The third argument *nvr* is the amount of values which are set with the function. The last argument *value[]* is a vector with the actual values of the variables which are set by the function.

Line 31 and 32 initialize the instances of the FMUs. The first argument of the function *fmi2SetupExperiment* is *c*. This argument contains a reference to the instance of the FMU which need to be initialized. The second argument *toleranceDefind* is used to indicate if error estimation needs to control the communication interval. The third argument *tolerance* can be used to for the error estimation. The fourth argument *startTime* indicates the start time of the simulation. the fifth argument *stopTimeDefind* indicates if a stop time is defined. The last argument *stopTime* indicates the stop time of the simulation.

Line 33 and 35 set both instances of the FMUs in initialization mode. In this mode it is possible to read the output values of the FMU and set the input values of the FMU. This is done line 37 and 38 of the code. The functions *fmi2SetXX* and *fmi2GetXX* are used to set the inputs and outputs of the FMU. Both functions works in the same way as explained before for setting the start values. Only the argument *value[]* of the function *fmi2GetXX* contains now an empty vector which will be set with the values which are get from the output.

Line 40 and 41 close the initialization mode of both instances of the FMUs. The actual co-simulation is done in line 46 to 79. Line 51 to 56 are responsible for the exchange of values between the inputs and outputs of both instances of the FMUs. Line 59 to 71 performs a simulation step and checks the status of the simulation. The first argument *c* of the function *fmi2DoStep* contains a reference to the instance of the FMU which need to perform a simulation step. The second argument *currentCommunicationPoint* is the current time for which the simulation is done. The third argument *communicationStepSize* is the communication step size for which the simulation needs to be performed. The fourth argument *noSetFMUStatePriorToCurrentPoint* is for now not important.

Line 83 to 87 informs the instances of the FMUs that the simulation is terminated. And line 89 to 93 unloads the instances of the FMUs and frees all the allocated memory.

4.2.2 FMU Interface Meta-model

Figure 4.5 shows the meta-model for the FMU interface. The class *FMIModel* represents the FMU interface itself. This class has an *name* attribute which represents the name of the model. The *FMIModel* class contains one or more ports which represents the inputs and outputs of the FMU. These ports contain a *name* attribute, a *direction* attribute, a *dataType* attribute and a *reference* attribute. The *name* attribute is used to identify the ports in the model itself. The *direction* attribute is used to indicates if the port is an input or an output. This is important to know because because to set a input the function *fmi2SetXX* is used and to get an output the function *fmi2GetXX* is used. The *dataType* attribute is used to determine if the *XX* indication of the function *fmi2SetXX* or the function *fmi2GetXX* needs to be replace by *Real*, *Integer*, *Boolean* or *String*. The last attribute *reference* holds the reference value to the input or output variable as explained in Section 4.2.1. The values for these attributes can be taken from the corresponding XML file of the FMU. Listing 4.3 shows an example of a XML file of a FMU generated by 20-sim. Line 12 to 16 contains the description of the input and the output of the FMU. This description contains a name, datatype, reference and a direction.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <fmiModelDescription fmiVersion="2.0" modelName="Motor" guid="{99
      b42531-8f87-402d-9e5e-a47279e32fe6}" generationTool="20-sim"
3 numberOfEventIndicators="0" copyright="Controllab Products B.V."
      license="-">
4 <CoSimulation modelIdentifier="Motor" needsExecutionTool="false"
      canHandleVariableCommunicationStepSize="true"
      canInterpolateInputs="false"
5 maxOutputDerivativeOrder="0" canRunAsynchronuously="false"
      canBeInstantiatedOnlyOncePerProcess="true"
      canNotUseMemoryManagementFunctions="true"
6 canGetAndSetFMUstate="false" canSerializeFMUstate="false"
      providesDirectionalDerivative="false" />
7 <DefaultExperiment startTime="0.0" stopTime="10.0" stepSize="0.01"
      />
8 <ModelVariables>
9 <ScalarVariable name="Motor1.r" valueReference="6" variability="
      tunable" causality="parameter">
10 <Real start="0.0394" />
11 </ScalarVariable>
12 <ScalarVariable name="Current" valueReference="20" variability="
      continuous" causality="input">
13 <Real start="0.0" />
14 </ScalarVariable>
15 <ScalarVariable name="Velocity" valueReference="21" variability="
      continuous" causality="output">
16 <Real />
17 </ScalarVariable>
18 <ScalarVariable name="Jcam1.state_initial" valueReference="7"
      variability="fixed" causality="parameter">
19 <Real start="0.0" />
20 </ScalarVariable>
21 <ScalarVariable name="phi1.q_initial" valueReference="8"
      variability="fixed" causality="parameter">
22 <Real start="0.0" />
```

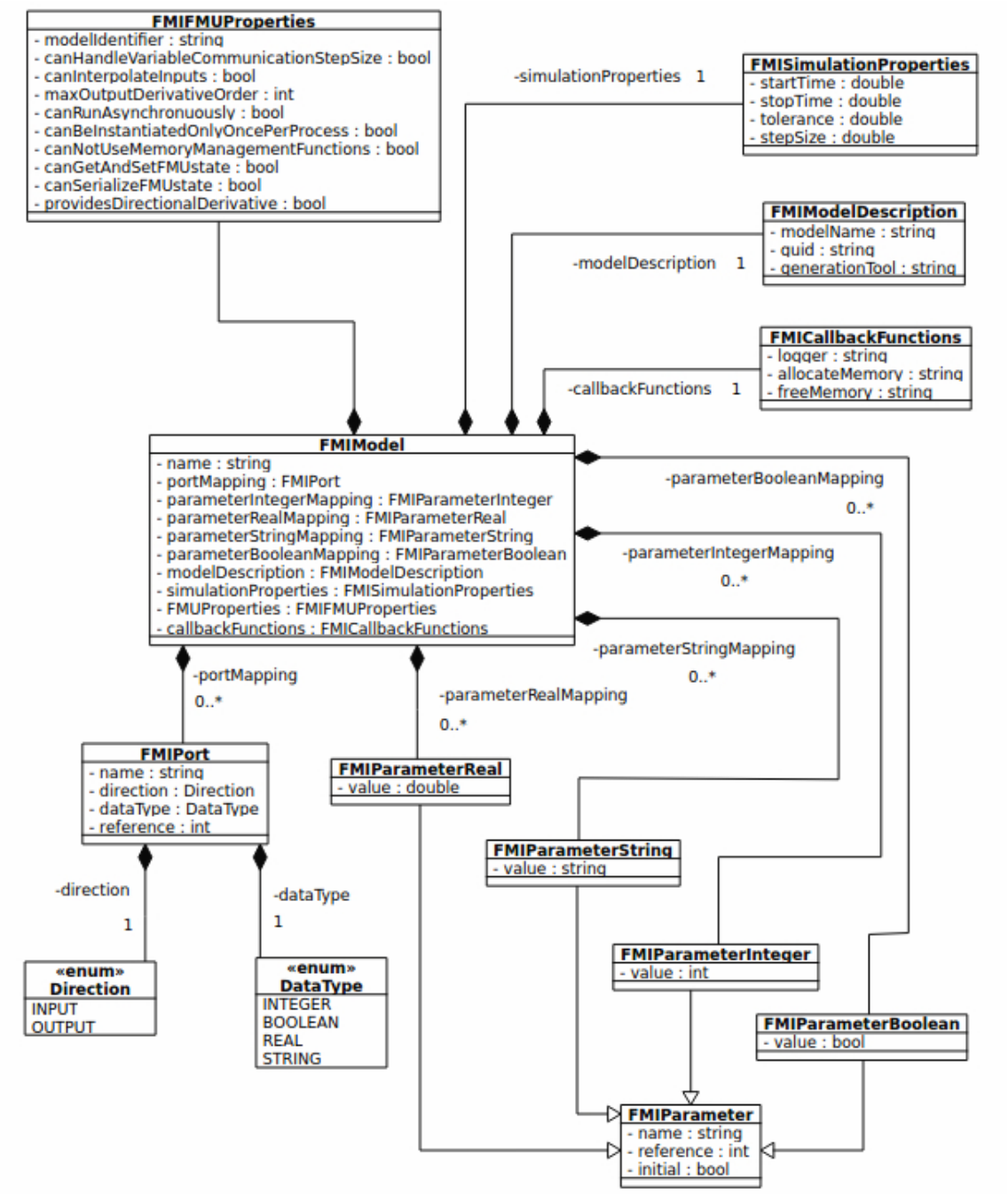

Figure 4.5: Meta-model of the FMU interface

```
23 </ScalarVariable>
24 <ScalarVariable name="QSensor2.q_initial" valueReference="9"
       variability="fixed" causality="parameter">
25 <Real start="0.0" />
26 </ScalarVariable>
27 <ScalarVariable name="Jmot1.state" valueReference="22" variability
       ="continuous" causality="local">
28 <Real />
29 </ScalarVariable>
```

```
30 <ScalarVariable name="Jmot1.p.e_in" valueReference="23" variability
       ="continuous" causality="local">
31 <Real />
32 </ScalarVariable>
33 </ModelVariables>
34 <ModelStructure>
35 <Outputs>
36 <Unknown index="53" />
37 </Outputs>
38 </ModelStructure>
39 </fmiModelDescription>
```

Listing 4.3: An example of the XML file of a FMU from 20-sim

The *FMIModel* class also contains four references to parameters. These parameters are split into four classes because every parameter contains a value. The datatype of this value dependence on the datatype of the parameter. So to hold the value of parameter of a specific datatype, for each datatype a separate class is necessary. Each parameter contains a *name* attribute, a *reference* attribute, a *initial* attribute and a *value* attribute. The *name* attribute is used to identify the parameters in the model itself. The *reference* attribute holds the reference value to the parameter variable as explained in Section 4.2.1. The *initial* attribute indicates if the parameter is fixed during simulation or can be changed during simulation. The last attribute *value* contains the value of the parameter. To include the parameters of the FMU into the FMU interface model, the XML file can be used. Line 9 to 11 of Listing 4.3 describes an parameter which can be changed during a simulation. This is indicated by the value *tunable* of the *variability* element. Line 18 to 26 describes the parameters which are fixed during a simulation. This is indicated by the value *fixed* of the *variability* element. The description of the parameters contains a name, datatype, reference and value which are needed for the FMU interface model. It is possible to change the value of a parameter by an other value than the value which is indicated by the XML file.

The XML file of Listing 4.3 contains also a description for local variables. These values represents the state variables of the model. It is possible to also include these values to the meta-model for the FMU inteface. But it is not necessary to set or change these values because they are only used during the simulation of the model and depends on the input, output and parameter values of the model. So these values are not added to the meta-model.

The *FMIModel* class contains a reference to a *CallbackFunctions* class. This class contains a *logger* attribute, a *allocateMemory* attribute and a *freeMemory* attribute. These attributes are used to indicates which functions can be used for logging, allocating memory and freeing memory on the platform where the FMU interface is used on. The values of these attributes needs to be set by the user because the names of these functions depends on the platform where the FMU interface is executed on.

The *FMIModelDescription* class contains a *modelName* attribute, a *guid* attribute and a *generationTool* attribute. The *modelName* attribute contains the name of model for which a FMU is generated. The *guid* contains a string which is used to check if the source files of the FMU belongs to the XML file of the FMU. The *generationTool* attribute contains the name of the tool which is used for the generation of the FMU. The values for these attributes can be taken from the XML file. Line 2 of Listing 4.3 gives the values for the three attributes.

The *FMISimulationProperties* class contains the attributes which are used to setup the time and communication step size for the simulation. Line 7 of Listing 4.3 describes some default values which can be used for the simulation properties. The *FMIFMUProperties* class contains some properties for the FMU which are indicated by the XML file. For this assignment these

properties are not used because only a simple simulation algorithm will be implemented. But when in the future some complex simulation algorithms are created for the co-simulation between TERRA and 20-sim, these properties can be used for code generation or validation of the FMU interface model.

4.2.3 Using the Meta-model in TERRA

The meta-model of the FMU interface is used to create a model for the FMU interface in TERRA. This model contains all the information which is necessary to create a C++ file which can handle the communication with the FMU. There are two approaches to use the FMU interface in TERRA.

The first approach is to create a new component in the architecture editor which implements the meta-model for the FMU interface. During code generation a C++ file can be generated which contains the communication to the FMU like the code in Listing 4.2. The epsilon plugin in eclipse can be used for this transformation. The problem for this approach is that also an interface needs to be created to the channels which are connected to the FMI interface component to exchange the input and output values with the other components in the architecture model.

The second approach is to use a CSP model to implement the FMU interface. A CSP model can contain a C++ code block which can be used to implement the FMU interface. Readers can be used to read values from the channels and make them available in the C++ block. Writers can be used to tranfer values from the C++ block to the channels. So it is not necessary to generate code for the interface to the channels because this code is automatically generated by the CSP editor when readers and writers are used. Also the code generation of the architecture models is based on the code generation for CSP models. So it is also not necessary to create code for the execution of the FMU interface. And finally the communication between CSP processes are based on rendezvous, so this can be used as a simple co-simulation engine. Every time when the inputs and outputs of the FMU are exchange with the rest of the model, the FMU can perform a simulation step.

The second approach is implemented in TERRA as described in Chapter 5.

5 Implementation of the FMU Interface in TERRA

As explained in chapter 4 the FMU interface is implemented as a CSP model. The XML file of the FMU is used as base for the implementation of the FMU interface. Section 5.1 describes how the XML file is used to generate a CSP model of the FMU interface. After the generation of the CSP model, this model needs to be translated to LUNA based code. Section 5.2 decribes how the CSP model of the FMU interface is translated to code.

The implementation of the FMU interface in TERRA is divided into different plugins. An overview with the important TERRA plugins for the implementation of the FMU interface is shown in Figure 5.1.

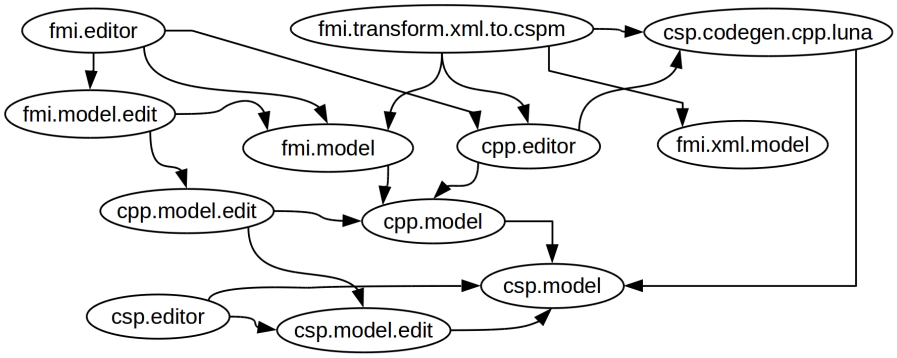

Figure 5.1: Overview with the important TERRA plugins for the implementation of the FMU interface

5.1 Translating the FMU's XML File to a CSP Model

This section describes how the XML file of the FMU is used to generate a CSP model of the FMU interface. First the implementation of the FMU interface meta-model in TERRA is discussed in Section 5.1.1. To read the XML file of a FMU, a plugin is created in TERRA to support it. This is explained in Section 5.1.2. Finally in Section 5.1.3 is discussed how a CSP model of a FMU interface is generated.

5.1.1 Implementation of the FMU Interface Meta-model

The plugin *nl.utwente.ce.terra.fmi.model* of TERRA contains the Ecore model of the meta-model of 4.2.2. In this Ecore model the class *FMIModel* inheritance from the class *CPPCodeBlockConfiguration*. The class *CPPCodeBlockConfiguration* is a class from the CPP meta-model which is also implemented in TERRA. The CPP meta-model implements the C++ block of the CSP editor in TERRA. The class contains the attributes *sourceFiles* and *HeaderFiles* which can be used in the FMU interface model to select the source files of the FMU which are also needed for the code generation, compiling and linking. The class inheritance form the class *ICPCExternalToolConfiguration* which is part of the CPC meta-model. This class is used to give a configuration to a component of a CPC model. So the meta-model for the FMU interface is in this way based on the CPC meta-model. And because it is also based on the CPP meta-model, code generation features of the CPP model can be used for the FMU interface model. In this way it is possible to easily create a C++ file with an interface to the FMU. Figure 5.2 shows an UML diagram of the inheritance of the *FMIModel* class. All the attributes and references of the *FMIModel* class itself is not included in this figure.

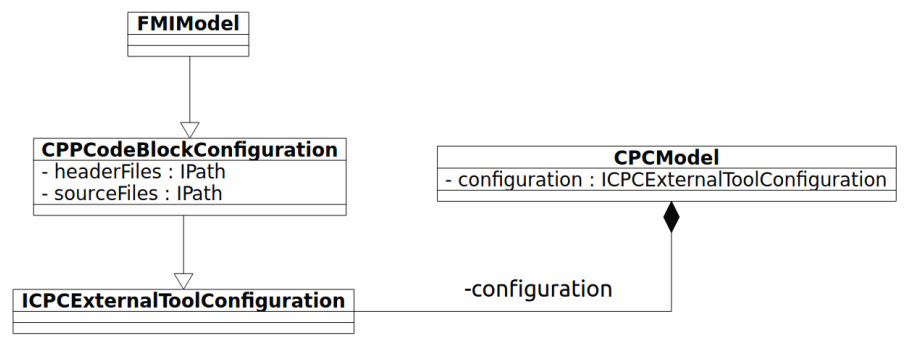

Figure 5.2: UML diagram of the inheritance of the *FMIModel* class

The plugin *nl.utwente.ce.terra.fmi.model.edit* of TERRA contains providers to display the properties of the FMU interface model in an UI. The plugin *nl.utwente.ce.terra.fmi.editor* of TERRA contains an simple editor for the FMU interface model. This editor makes it possible to show a C++ code block as FMU interface block. It makes it also possible to see and change the properties of the FMU interface model by clicking on the FMU interface block. Figure 5.3 shows the FMU interface model editor.

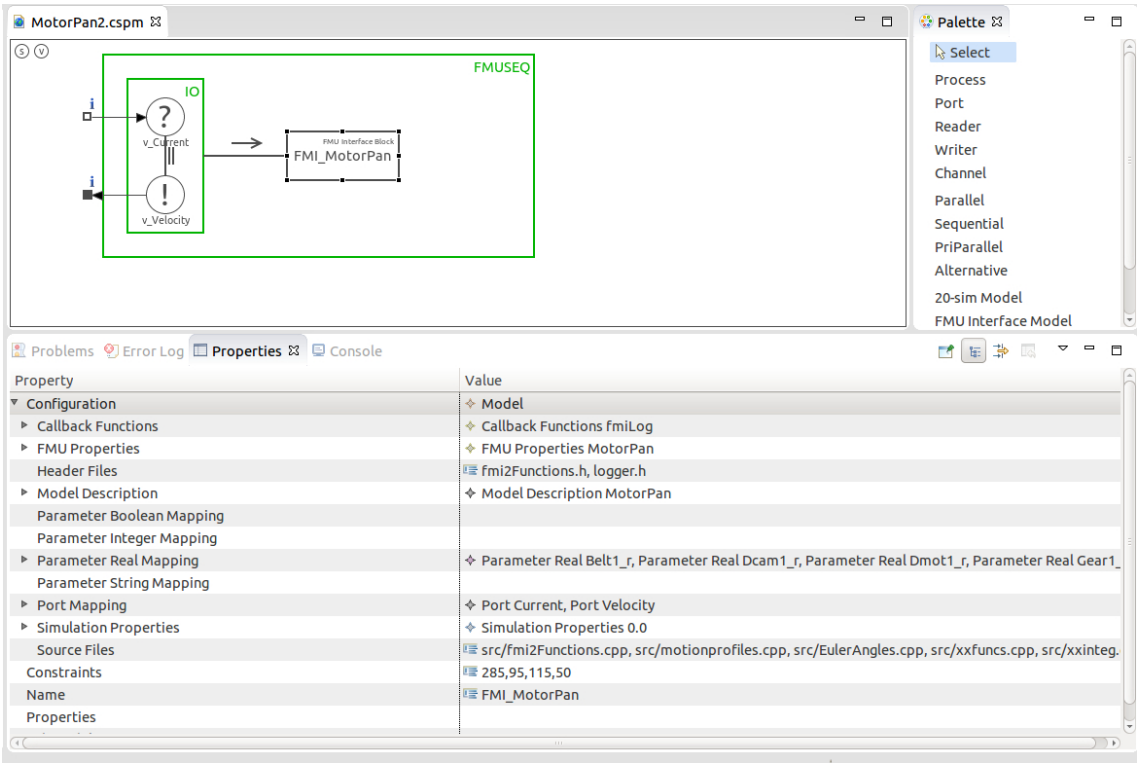

Figure 5.3: FMU interface model editor in TERRA

5.1.2 Support for Reading the XML File of a FMU

The plugin *nl.utwente.ce.terra.fmi.xml.model* of TERRA contains an Ecore meta-model which is generated from the XML schema definition of the XML files of the FMUs. This meta-model makes it possible to read a XML file of a FMU in TERRA. All the elements which are defined in the XML file are placed in the model which is generated from the meta-model. This model is finally used to define a CSP model for the FMU interface model. A simplified UML diagram of

the meta-model is shown in Figure 5.4. This UML diagram contains only classes and attributes which are used for the definition of the CSP model because otherwise the figure becomes very complex.

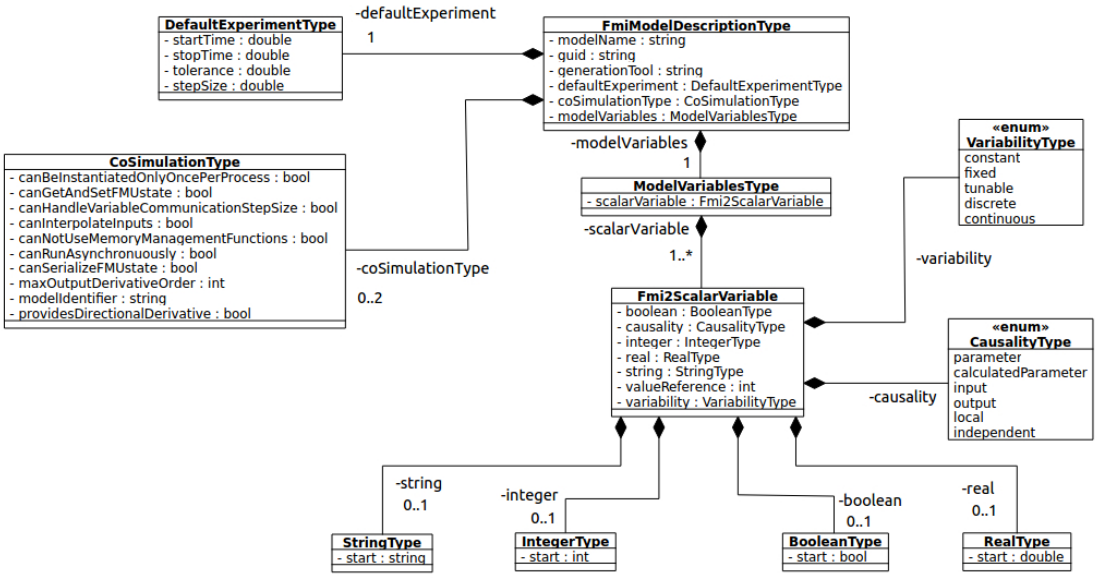

Figure 5.4: A simplified UML diagram of the meta-model for reading the XML files of the FMUs

The class *FmiModelDescriptionType* is the main class. It contains attributes for the name of the model, the name of the generation tool and the guid string which is necessary for the validation of the source files. The class contains a reference to the *ModelVariablesType* class. This reference contains all the information about the inputs, outputs, parameters and state variables of the model. The reference to the class *DefaultExperimentType* contains the default simulation properties which can be used for a simulation. The reference to the class *CoSimulationType* contains properties which hold for the FMU of the model.

5.1.3 Creating a CSP model of a FMU Interface

The plugin *nl.utwente.ce.terra.fmi.transform.xml.to.cspm* of TERRA contains an ETL file which is used to generate a CSP model from the XML file of the FMU. It use the meta-model of the XML file to read the XML file and to translate this XML file to a CSP model.

The generated CSP model contains a FMU interface block and some readers and writers. Figure 5.5 shows an example of a CSP model of the FMU interface. During the model translation the ETL file first generates an FMU interface block. This FMU interface block is based on the meta-model of the FMU interface model and is used to finally create a c++ file which contains the interface to the FMU. Information like the name, guid, generation tool, FMU properties and the default simulation properties are copied from the XML file to the FMU interface block. The FMU interface block also holds some source and header files which are used during the compilation and linking of the control software. These files are automatically added to the Makefile which is used to generate an application of the controller code. So the source files which are necessary for the FMU interface can also be added. This files are automatically added by the ETL file because for every model in 20-sim where an FMU is generated from use the same source files. A better option was to select the files by hand because as in that case it is also possible to use FMU's which are created by other programs. But in the current TERRA application the selection of files via a window does not work.

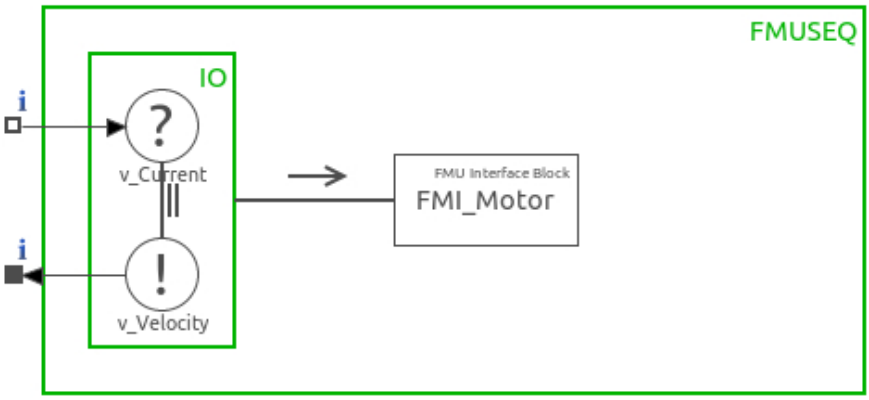

Figure 5.5: Example of a FMU interface model

The readers and writers in the CSP model of the FMU interface are used to send values to and receive values from the channels which are connected to the FMU interface model. These values are saved in variables. Every reader and writer has it own variable for holding a value. These variables can be shared with the FMU interface block. This make it possible to send values from the channels to the FMU inputs and values from the FMU outputs to the channels. It is necessary to define a unit type for each variable which is defined in a CSP model. This unit type contains a unit name, a quantity name and a data type. So it is possible for example to create a unit type for the current with data type real and a unit type for the velocity with data type real. In the end only the data type of the different variables are important. The quantity name and unit name of a unit type is not used during code generation of the FMU interface. So for the CSP model for the FMU interface only three unit types are defined which can be used as unit type for all the variables inside the FMU interface model. These unit types are:

- **fmi_real** with data type *real*
- **fmi_boolean** with data type *boolean*
- **fmi_integer** with data type *integer*

These unit type are also used in the FMU interface block to also create variables of the same unit type which makes it possible to share the variables with the variables of the readers and writers. For every variable in the XML file of the FMU, which represents an input or an output the following elements are created or added in the CSP model:

- A variable for the reader or writer with the name *v_XXX* where *XXX* is the name of the variable in the XML file. The unit type of this variable corresponds with the data type which is mentioned in the XML file.
- A port with the name *XXX* where *XXX* is the name of the variable in the XML file. For an input variable the direction of the port is *Outgoing* and for an output variable the direction of the port is *Incoming*.
- A reader with the name *r_XXX* or a writer with the name *w_XXX* where *XXX* is the name of the variable in the XML file. The created variable is added to the reader or writer.
- A link between the port and the reader or writer.

- Also an instance of the class *FMIPort* is added to the FMU interface block with the data of the input or output variable from the XML file. The data of these instance is used to finally implements a function in the C++ file which can change the input or ouput value of the FMU.

The readers and writer are placed group with a parallel structure as can be seen from Figure 5.5. This is done because for co-simulation it is necessary to exchange the values of the inputs and the outputs of the system at the same time.

Also instances of the classes *FMIparameterInteger*, *FMIParameterReal*, *FMIParameterBoolean* and *FMIParameterString* are added to the FMU interface block for every variable in the XML file which represents an parameter. The data of these instances are used to finally implements a function in the C++ file which can change the parameters of the FMU.

At the end of the conversion the group with readers and writers are placed in a group with the FMU interface block with a sequential structure. This sequential structure is used because first all the output values needs to be send to the channels and all the input values needs to be read from the channels before these values are send to the FMU or the values are replaced by other values from the FMU.

After the conversion a CSP model like the one shown in Figure 5.5 is generated. When clicking on the FMU interface block the properties of this block is shown in the properties window as shown in Figure 5.3. It is possible to change the names of the callback functions in this window to use the functions which are supported by the execution platform. Also the simulation properties can be changed to use a different step size or a different stop time.

5.2 Code Generation of the CSP Model to LUNA

The FMU interface needs a logging function which is used to send error or info messages about the state of the FMU. Without this logging function, it is not possible to use the FMU interface. For this assignment Linux is used as operating system for the execution of the FMU interface. Linux has not a logging function with the same structure as the one which is used for the FMU interface. Section 5.2.1 describes a logger library which can be used in combination with the FMU interface.

After a CSP model is generated from the XML file of a FMU, code needs to be generated from the model to actually use the FMU interface in combination with a controller. Code generation for the CSP model itself is already implemented in TERRA. Only the code generation for the FMU interface block needs to be implemented in TERRA. Section 5.2.2 described the code generation of the FMU interface block.

5.2.1 The Logger Library

The logging function can be implemented in two ways. The first way was to simple create a source file which implements the logging function. The second way was to create a logger class which contains the logging function. The advantage of creating a logger class is that multiple instances of the logger class can be created. But for the current implementation of the FMU interface, multiple instances of the logger class are not necessary. So the logging function is simply implemented as function in a source file.

The logger library contains the function *fmiLog()* which is the logging function which can be used in combination with the FMU interface. All messages which are generated by the FMU are displayed on the screen by this function. Also all the messages are saved into a file because sometimes a lot of output is printed on the screen during the execution of the control software. So to easily find a message which is generated by the FMU, the logging file can be used.

The logger library contains also the function *enumStatusToString()*. This function is used to translate the variable of the type *fmi2Status* which is an enum from a integer to a word. This variable is used in the *fmiLog()* function. So to make the messages more readable this translation is needed.

At the moment it is not possible to run the simulation of the physical system model in 20-sim itself. So at the moment it is not possible to automatically draw graphs in 20-sim from the inputs and outputs of the model. A possibility is to create the graphs at the FMU interface. But this means that a function needs to be created which can create graphs from the values which are send through the FMU interface. This function is not available in the LUNA framework so it needs to be implemented. Because this takes too much time for this project, the values of the inputs an outputs of the FMU interface will be saved in a .csv file. This .csv file can be used in Excel or Matlab to create a graph from these values.

The function to save the values of the inputs and outputs of the FMU interface is also implemented in the logger library. This function is called *saveMeasurementPoint()*. This function gets a string as arguments. This string contains the values which can be written to a file. It was also possible to place the values into vectors but then for every data type a different vector was necessary. It is also a lot easier to generate a string of values with the EGL language which is used to generate code from the FMU interface block.

Finally the logger library also contains the function *initLogger()*. This function gets the names for the logger file and measurement file as argument. These two files are generated and used to save the messages and the values of the inputs and the outputs of the FMU interface. It gets also a string of names as argument. This string can contain the names of the values which are saved in the measurement file. This names are placed at the top of the .csv file to indicate what the numbers mean.

5.2.2 Code Generation of the FMU Interface Block

During the code generation of a CSP model, the C++ block in a CSP model is transformed to a source file and a header file which contains an implementation of a class. The name of the class is equal to the name of C++ block in the CSP model. The attributes of the class are equal to the variables which are added to the c++ block in the CSP model. The class contains a constructor, a destructor and the function *execute()*. The constructor of the class is called when the program is started, the destructor is called when the program ends and the *execute()* function is called every time during the execution of the program when the C++ block is active following the structure of the CSP model. The user can manually add an implementation to the class.

For the FMU interface block the same files are generated as for the C++ block because the FMU interface block is based on the C++ block. The only difference is that the functions of the FMU interface block are automatically implemented using the properties of the FMU interface block. The plugin *nl.utwente.ce.terra.fmi.transform.xml.to.cspm* of TERRA contains some EGL files which are used to generate implementations for the functions based on the properties of the FMU interface block.

The first EGL file is *classdefinitions.egl*. Some extra attributes are added to the class, which is generated from the FMU interface class, by this file. A variable of the type *fmi2Component* is defined here. This variable is used to hold an instance of the FMU component. Because this instance must be available for all functions in the class it is defined as attribute. A instance of the *fmi2CallbackFunctions* is also defined here because the callback functions are used by the instance of the FMU component, so it also needs to be defined for all functions in the class. The instance of the *fmi2CallbackFunctions* use the functions which are defined by the user in the FMU interface block. The pointer to the function *stepFinished* and the pointer to *compo-*

nentEnvironment are not defined because they are optional and in this case not necessary for the co-simulation. It is possible to add this pointers later when they are necessary.

Also the variables *timeStep*, *currentTime*, *status* and *simulationFinished* are defined as attribute of the class because the variables are used in the function *execute()* and need to hold there values for the next run of the *execute()* function.

The second EGL file is *constructor.egl*. The implementation of the constructor of the class is provided by this file. First the function *initLogger()* is defined here with some standard names for the logging and measurement files. It is possible to also include the names of these files in the meta-model of the FMU interface model. This makes it possible that the user can use another name for the files. But to keep the meta-model simple this feature is not added. Also when it is possible to run the physical system model in 20-sim itself, it is not necessary to use the measurement file anymore.

The names of all inputs and outputs of the FMU interface are placed in a string which is also given as argument to the *initLogger()* function. This makes it possible that the measurement file contains the names of the ports. This make it easy to distinguished the values of the measurement file.

The *fmi2Instantiate()* function is defined in the constructor to create a instance of the FMU. This function needs a name, a guid and callback functions as argument. The name and guid are taken from the FMU interface block. The callback functions are defined as class attribute as explained before.

After a FMU instance is created, the attributes of the class are initialized. The values of the variables *timeStep* and *currentTime* are taken from the FMU interface block. The value of *currentTime* is equal to the value of *startTime* of the FMU interface block because this is the time which is used as start time for the simulation.

Then the function *fmi2SetupExperiment()* is defined to place the instance of the FMU in simulation mode. The values for the start time, tolerance, stop time and step size are taken from the FMU interface block. This function also needs to know if a tolerance and a stop time are defined. This is automatically determined by checking if the values for the tolerance and stop time are defined as zero. In this case there is no tolerance and stop time defined.

The function *fmi2EnterInitializationMode()* is defined to place the instance of the FMU in initialization mode. When the FMU is in this mode, all the parameters which are defined in the FMU interface block are set. This is done because the user can change the value of the parameter in the FMU interface block. This change needs also be transferred to the FMU. This can be done when the FMU is in initialization mode.

Also all the values of the outputs of the FMU are transferred to the output channels. The information about the output ports can be found in the port mapping reference of the FMU interface block. This makes it possible that for the first execution of the controller the output values of the physical system model are available at the inputs of the controller. Otherwise it first use the standard values of zero as input values.

Then the function *fmi2ExitInitializationMode* is defined to place the instance of the FMU out of the initialization mode. All the described functions are placed in the constructor of the class which is generated from the FMU interface class because these functions needs to be called only at the beginning of the execution of the application. The constructor is only called one time at the beginning of the execution of the application. Figure 5.6 shows an overview of the execution steps of the constructor.

The third EGL file is *execute.egl*. The implementation of the *execute()* function of the class is provided by this file. First a check is implemented in this function which checks if the stop

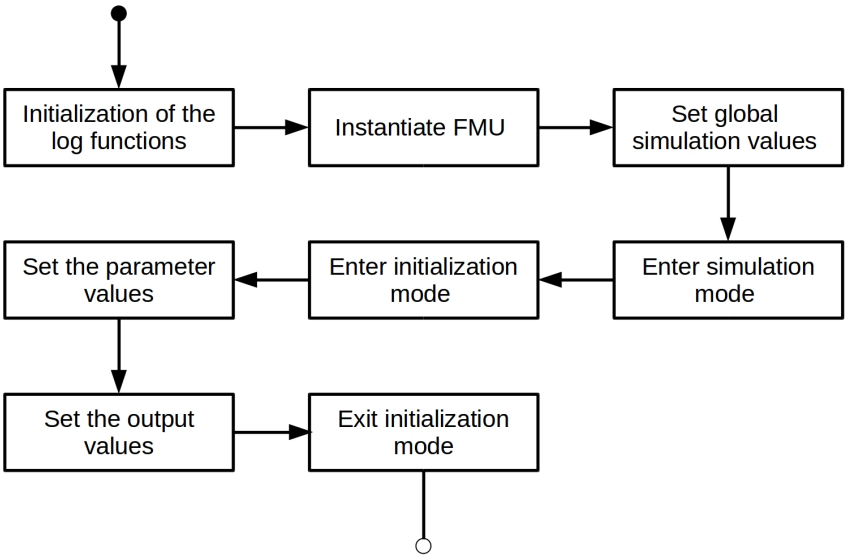

Figure 5.6: Overview diagram of the execution steps of the constructor

time of the simulation is reached or that the instance of the FMU does not work like expected. Because if this is the cause, it is not necessary to execute a simulation step.

Then all the inputs of the FMU interface which are defined in the FMU interface block receive there new input values from the channels because it is necessary that the simulation step is done with the actual input values. After the inputs of the FMU interface receive their new values, the time, input and output values are saved. Then a simulation step is done with the step size which is defined in the FMU interface block. After the simulation step the *currentTime* variable is increased with the time of the step size. All the outputs of the FMU interface which are defined in the FMU interface block send there new values to the channels. This values can be used by another component in the next execution round of the total system.

These function are added to the *execute()* function because they need to be called for every execution round of the total system. This makes it possible that the co-simulation is automatically performed by using the synchronization of the channels in the total system. Figure 5.7 shows an overview of the execution steps of the execute functions.

The fourth EGL file is *destructor.egl*. The implementation of the *destructor* of the class is provided by this file. Only the functions *fmi2Terminate* and *fmi2FreeInstance* are called to exit the instance of the FMU from the simulation mode and to delete the instance to make memory free. These functions needs to be called one time at the end of the execution of the application. Therefore they are placed in the destructor. Figure 5.8 shows an overview of the execution steps of the execute functions.

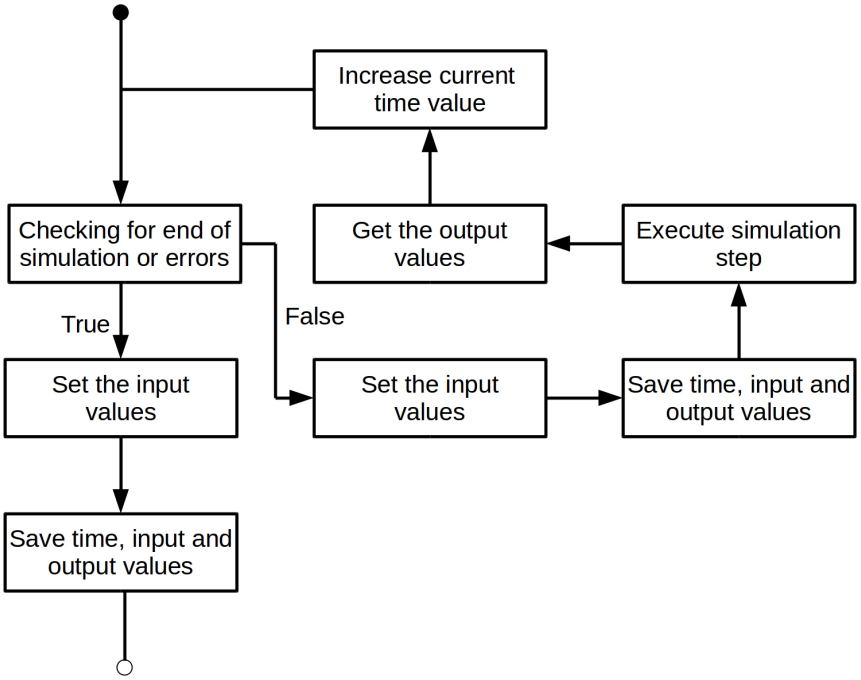

Figure 5.7: Overview diagram of the execution steps of the *execute* function

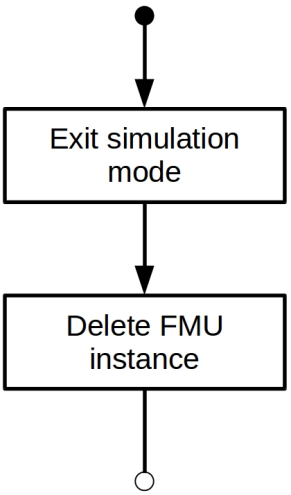

Figure 5.8: Overview diagram of the execution steps of the destructor

6 Testing the FMI Interface in TERRA

This chapter shows some experiments for testing the FMU interface in TERRA. Each experiment begins with a model of a plant which is designed in the 20-sim editor. From this plant model a FMU is generated. The files of the FMU are used by TERRA to generate a FMU interface model. Appendix A describes how a model in 20-sim can be translated to an FMU interface model in TERRA. So this translation is not discussed for every experiment. For every experiment a simple architecture model is used which only contains a controller block and a plant block like the one which is shown in Figure 6.1. The plant component contains the FMU interface model and the controller block contains a CSP model like the one shown in Figure 6.2. The controller model contains some readers and writers to read from a channel or write to a channel. All readers and writers are placed in a parallel structure because for co-simulation the data of the inputs and outputs of the models are interchanged at the same time. And of course the readers and writers of the FMU interface model are also placed in a parallel structure. A C++ block is placed in the controller to make it possible to define a specific input for the FMU interface. The results of all the experiments are automatically saved in a file by the *saveMeasurementPoint* function which is implemented in the FMU interface as explained in section 5.2.1.

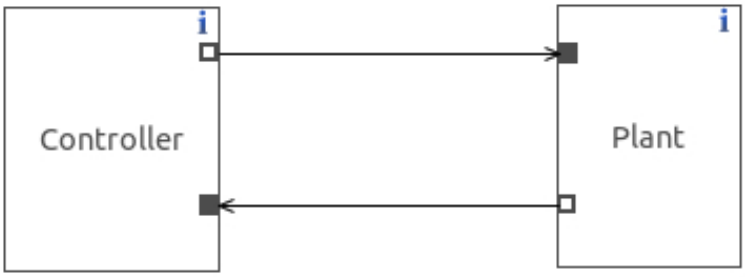

Figure 6.1: Example of an architecture model for the experiments

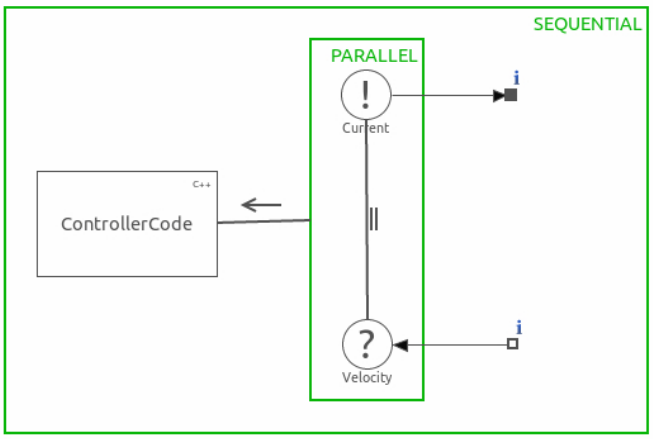

Figure 6.2: Example of a controller model for the experiments

6.1 Experiment 1: Check Code Generation for Ports and Parameters

This experiment is used to check if the communication from and to the FMU interface is done correctly. It checks if the communication to the inputs of the FMU interface and the communication from the output of the FMU interface works as expected. It also checks if the values of the parameters of the model can be changed through the FMU interface. This experiment shows that the code generation of the FMU interface by TERRA works.

6.1.1 Model Description

A test model is used to perform the check. The test model interface consists of twelve ports:

- An input port with the name *BooleanInput1* and the datatype *boolean*
- An output port with the name *BooleanOutput1* and the datatype *boolean*
- An input port with the name *BooleanInput2* and the datatype *boolean*
- An output port with the name *BooleanOutput2* and the datatype *boolean*
- An input port with the name *IntegerInput1* and the datatype *integer*
- An output port with the name *IntegerOutput1* and the datatype *integer*
- An input port with the name *IntegerInput2* and the datatype *integer*
- An output port with the name *IntegerOutput2* and the datatype *integer*
- An input port with the name *RealInput1* and the datatype *real*
- An output port with the name *RealOutput1* and the datatype *real*
- An input port with the name *RealInput2* and the datatype *real*
- An output port with the name *RealOutput2* and the datatype *real*

Figure 6.3 shows the implementation of the test model in the 20-sim editor. The following parameters are used:

- AndBool1.condition = true
- AndBool2.condition = true
- GainInteger1.K = 1
- GainInteger2.K = 1
- GainReal1.K = 1
- GainReal2.K = 1

The test model shows that the *BooleanInput1* and *BooleanInput2* signals are connected to an AND port model. These AND port models have one input port and one output port. The implementation of these AND port models is shown in Listing 6.1. This implementation contains a parameter *condition*. When the value of this parameter is equal to *true* it means that the input and output are connected. When the value is equal to *false* the connection between the input and output is broken.

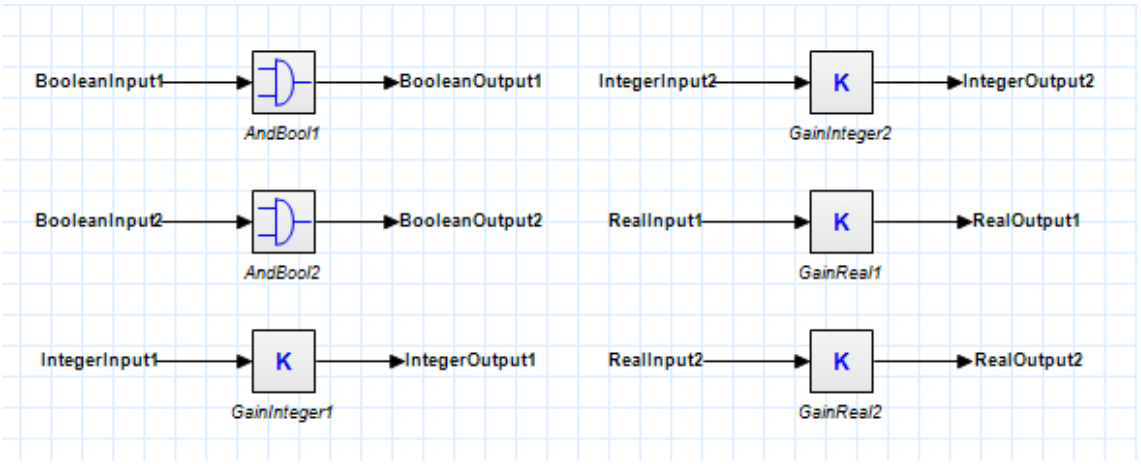

Figure 6.3: Implementation of the test model in the 20-sim editor

```
parameters
   boolean condition = true;
equations
   output = if input1 > 0.5 and condition then
      true
   else
      false
   end;
```

Listing 6.1: Implementation of the AND port

The inputs with datatype *integer* and *real* are connected to a Gain model. This model multiple the input value with the value of parameter *K*. When this values is equal to 1 the input value is the same as the output value.

For the FMU generation the Euler integration method is used with a start time of 0, a stop time of 10 seconds with a step size of 1 second. Figure 6.4 shows the implementation of the FMU interface model which is generated by TERRA using the XML file of the FMU. It can be seen that the input and output ports with the corresponding readers and writers are generated as expected. Also the port mapping and parameter mappings are visible when clicking on the FMU interface block of the CSP model. It is also possible to change the values for the simulation properties.

6.1.2 Experiments

Experiment A: Test Model with Standard Values for the Parameters

In this experiment the controller send values to the inputs of the FMU interface which variate over time. This is done by adding the following code in the *execute()* function of the source file of the code block of the controller:

```
if(CounterBool1 < 1)
   CounterBool1++;
else
{
   if(BooleanOutput1 == false)
      BooleanOutput1 = true;
   else
```

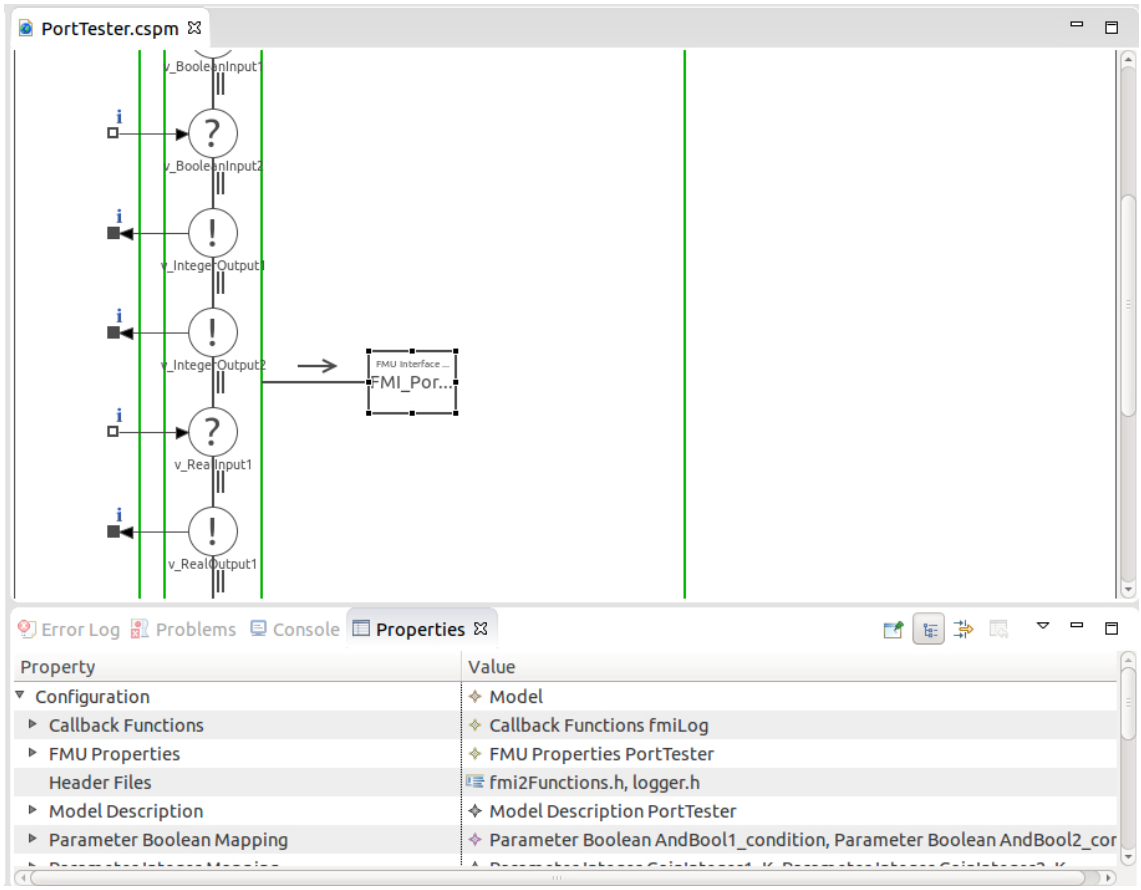

Figure 6.4: part of the CSP model of the FMU interface to the test model

```
        BooleanOutput1 = false;
    CounterBool1 = 0;
}

if(CounterBool2 < 2)
    CounterBool2++;
else
{
    if(BooleanOutput2 == false)
        BooleanOutput2 = true;
    else
        BooleanOutput2 = false;
    CounterBool2 = 0;
}

IntegerOutput1 = IntegerOutput1 + 1;
IntegerOutput2 = IntegerOutput2 + 5;
RealOutput1 = RealOutput1 + 0.01;
RealOutput2 = RealOutput2 + 0.05;
```

The co-simulation starts at time 0 and runs for 10 seconds with a time step of 1 second. The results are shown in tables 6.1, 6.2 and 6.3. These tables show that the values which are received by the inputs are one simulation cycle later available at the corresponding output ports.

Table 6.1: Results of the ports of datatype boolean with parameter values *AndBool1.condition = true* and *AndBool2.condition = true*

Time (s)	BooleanInput1	BooleanOutput1	BooleanInput2	BooleanOutput2
0	0	0	0	0
1	0	0	0	0
2	1	0	0	0
3	1	1	1	0
4	0	1	1	1
5	0	0	1	1
6	1	0	0	1
7	1	1	0	0
8	0	1	0	0
9	0	0	1	0
10	1	0	1	1

Table 6.2: Results of the ports of datatype integer with parameter values *GainInteger1.K = 1* and *GainInteger2.K = 1*

Time (s)	IntegerInput1	IntegerOutput1	IntegerInput2	IntegerOutput2
0	0	0	0	0
1	1	0	5	0
2	2	1	10	5
3	3	2	15	10
4	4	3	20	15
5	5	4	25	20
6	6	5	30	25
7	7	6	35	30
8	8	7	40	35
9	9	8	45	40
10	10	9	50	45

Table 6.3: Results of the ports of datatype real with parameter values *GainReal1.K = 1.0* and *GainReal2.K = 1.0*

Time (s)	RealInput1	RealOutput1	RealInput2	RealOutput2
0	0.00	0.00	0.00	0.00
1	0.01	0.00	0.05	0.00
2	0.02	0.01	0.10	0.05
3	0.03	0.02	0.15	0.10
4	0.04	0.03	0.20	0.15
5	0.05	0.04	0.25	0.20
6	0.06	0.05	0.30	0.25
7	0.07	0.06	0.35	0.30
8	0.08	0.07	0.40	0.35
9	0.09	0.08	0.45	0.40
10	0.10	0.09	0.50	0.45

Table 6.4: Results of the ports of datatype boolean with parameter values *AndBool1.condition = false* and *AndBool2.condition = false*

Time (s)	BooleanInput1	BooleanOutput1	BooleanInput2	BooleanOutput2
0	0	0	0	0
1	0	0	0	0
2	1	0	0	0
3	1	0	1	0
4	0	0	1	0
5	0	0	1	0
6	1	0	0	0
7	1	0	0	0
8	0	0	0	0
9	0	0	1	0
10	1	0	1	0

Table 6.5: Results of the ports of datatype integer with parameter values *GainInteger1.K = 2* and *GainInteger2.K = 5*

Time (s)	IntegerInput1	IntegerOutput1	IntegerInput2	IntegerOutput2
0	0	0	0	0
1	1	0	5	0
2	2	2	10	25
3	3	4	15	50
4	4	6	20	75
5	5	8	25	100
6	6	10	30	125
7	7	12	35	150
8	8	14	40	175
9	9	16	45	200
10	10	18	50	225

Experiment B: Test Model with not Standard Values for the Parameters

This experiment is similar to the previous experiment. The only difference is the change of the parameter values. These parameter values are changed in the FMU interface block of the FMU interface model in TERRA. The new values for the parameters are:

- AndBool1.condition = false
- AndBool2.condition = false
- GainInteger1.K = 2
- GainInteger2.K = 5
- GainReal1.K = 2.5
- GainReal2.K = 4.7

The results of this experiment are shown in tables 6.4, 6.5 and 6.6. The values of *BooleanOutput1* and *BooleanOutput2* are always zero as expected because changing the condition values of the AND port models to *false* blocks the input signals. The other signals are multiplied with the corresponding parameter value of the Gain models as expected.

Table 6.6: Results of the ports of datatype real with parameter values *GainReal1.K = 2.5* and *GainReal2.K = 4.7*

Time (s)	RealInput1	RealOutput1	RealInput2	RealOutput2
0	0.000	0.000	0.000	0.000
1	0.010	0.000	0.050	0.000
2	0.020	0.025	0.100	0.235
3	0.030	0.050	0.150	0.470
4	0.040	0.075	0.200	0.705
5	0.050	0.100	0.250	0.940
6	0.060	0.125	0.300	1.175
7	0.070	0.150	0.350	1.410
8	0.080	0.175	0.400	1.645
9	0.090	0.200	0.450	1.880
10	0.100	0.225	0.500	2.115

6.1.3 Discussion

This experminent shows that the translation from the XML file of the FMU to the FMU interface model works correctly. All the data from the XML file are copied to the FMU interface block. Also the inputs and outputs of the model are genereted correctly.

Also the communication between the FMU and TERRA works correctly. Input values are send to the correct inputs of the FMU interface. And the FMU interface also sends the values to the correct outputs. Also it is possible to change the values of the parameters of the test model. This values changes at the correct places through the FMU interface. So the code generation for the FMU interface by TERRA works correctly.

The shift of one simulation cycle between the input values and output values is caused by a delay between the input and output values. It takes a simulation cycle before the inputs of the FMU interface has influence on the outputs of the FMU interface.

6.2 Experiment 2: Using a Simple Plant Model for Co-simulation

This experiment shows that the FMU interface works in combination with a simple model which is modeled by a bond graph. In this case a model of a motor is used where a camera is attached on. In the first sub-experiment the standard values for the parameters of the motor model are used. Also the input of the motor model is kept constant. In the second sub-experiment a parameter of the motor model is changed in the editor of TERRA. In the third sub-experiment the input is variated. All these experiments are also performed in 20-sim to compare the simulation results.

6.2.1 Model Description

The motor model interface consists of two ports:

- An input port with the name *Current* and the datatype *real*
- An output port with the name *Velocity* and the datatype *real*

Figure 6.5 shows the bond-graph implementation of the motor model in the 20-sim editor. The following parameters are used:

- Belt.r = 5.0
- Dcam1.r = 1.35E-5

- Dmotq.r = 1.77E-6
- Gear1.r = 4.0
- Jcam1.i = 0.0045
- Jcam1.state.initial = 0.0
- Jmot1.i = 2.63E-6
- Motor1.r = 0.0394
- phi1.q.initial = 0.0
- QSensor2.q.initial = 0.0

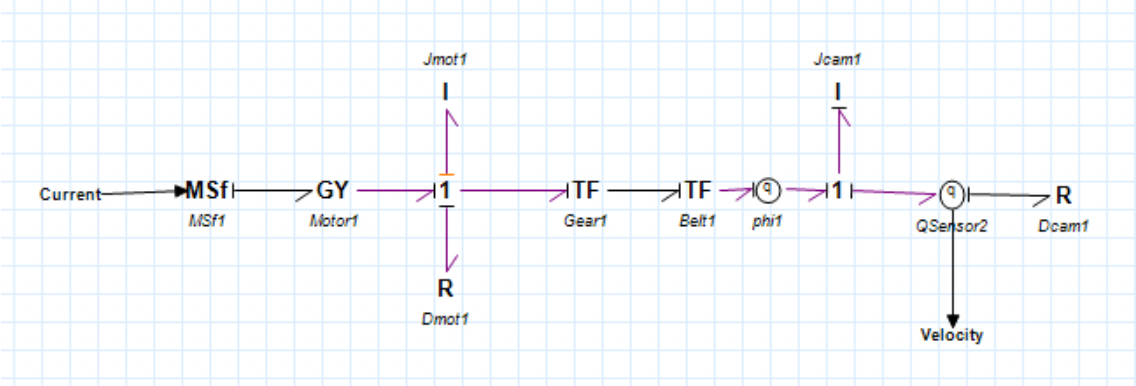

Figure 6.5: Bond graph implementation of a motor in the 20-sim editor

For the FMU generation the Euler integration method is used with a start time of 0, a stop time of 10 seconds and a step size of 0,001 second. Figure 6.6 shows the implementation of the FMU interface model which is generated by TERRA using the XML file of the FMU. It can be seen that an output and input port with the corresponding writer and reader is generated as expected. Also the port mapping and parameter mappings are visible when clicking on the FMU interface block of the CSP model. It is also possible to change the values for the simulation properties.

6.2.2 Experiments

Experiment A: Constant Input and Using the Standard Parameter Values

In this experiment the controller sends continuously a signal of 0.1 to the input of the FMU interface. This is done by adding the following code in the *execute()* function of the source file of the code block of the controller:

Current = 0.1;

where *Current* is the variable for the output of the controller. The co-simulation starts at 0 seconds and runs for 20 seconds with a time step of 0,001 seconds. The stop time of 20 seconds instead of 10 seconds is used to show that the simulation still works for stop times which are longer than the default stop time. The same simulation is also performed in 20-sim with the same conditions. A constant source with a value of 0.1 is connected to the motor model to perform the simulation.

The results for both simulations is shown in figure 6.7. The figure shows that both graphs are very similar to each other. The only difference is that the graph of the TERRA simulation is shifted 0.002 seconds to the right of the graph of the 20-sim simulation. This can be seen in figure 6.8.

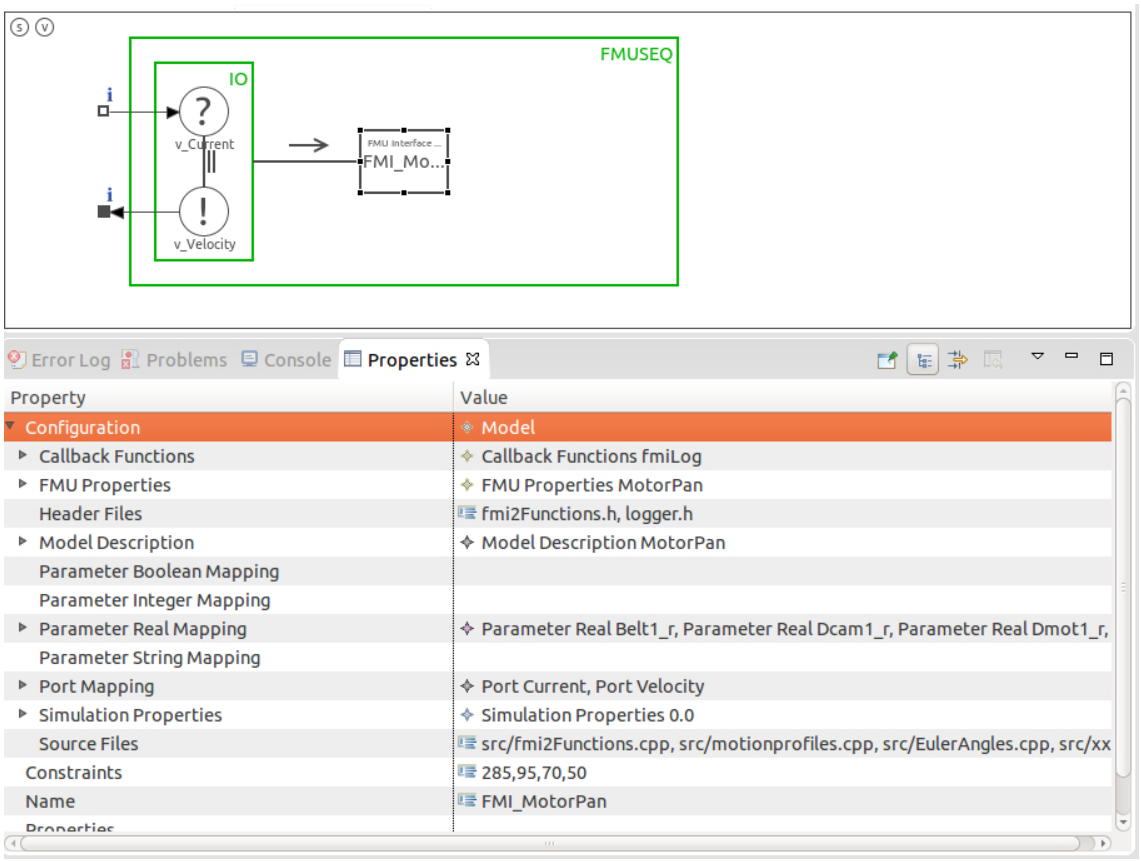

Figure 6.6: CSP model of the FMU interface to the motor model

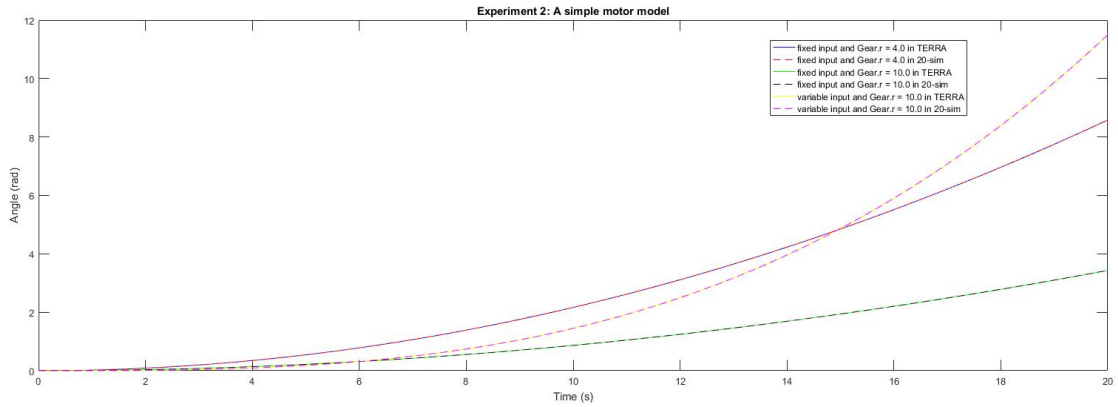

Figure 6.7: Graph with the result of experiment 2

Experiment B: Constant Input and Changing a Parameter Value

This experiment is similar to the previous experiment. Only a parameter of the motor model is changed. The value of parameter *Gear1.r* is changed from 4.0 to 10.0. This is done by opening the FMU interface model in TERRA. In this model a FMU Interface Block is visible. Properties of this block becomes visible in the properties window by clicking on the block. One of the properties is a list of parameters with their corresponding values. Here it is possible the change the value of *Gear1.r* parameter. For the simulation in 20-sim also the value of the parameter is changed

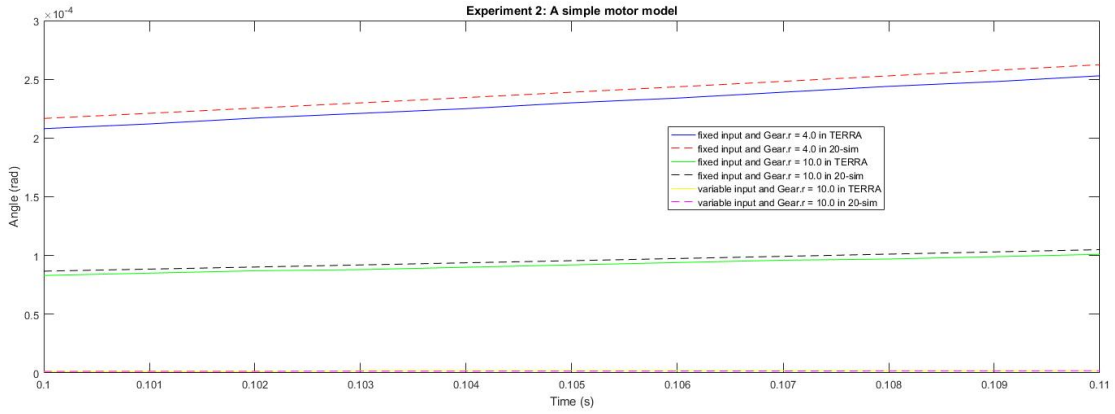

Figure 6.8: Graph with the result of experiment 2 zoomed in around time 0,1

The results for both simulations is shown in figure 6.7. The figure shows that both graphs are very similar to each other. The only difference is that the graph of the TERRA simulation is shifted 0.002 seconds to the right of the graph of the 20-sim simulation. This can be seen in figure 6.8.

Experiment C: Variable Input and Changing a Parameter Value

This experiment is similar to the previous experiment. Only now the input of the FMU interface is not constant. The input value begins at 0 and increase every cycle with 0,00005. This means that the input value have a slope of 0,05 per second. So the code in the the *execute()* function of the source file of the code block of the controller is replaced by:

Current = Current + 0.00005;

For the simulation in 20-sim the constant source is replaced by a ramp source which also have a slope of 0,05 per second.

The results for both simulations is shown in figure 6.7. The figure shows that both graphs are very similar to each other. The only difference is that the graph of the TERRA simulation is shifted 0.001 seconds to the right of the graph of the 20-sim simulation.

6.2.3 Discussion

This experiment shows that the FMU interface works for co-simulation with a simple plant model. The results of the co-simulation are similar to the results when the simulation is totally done in 20-sim. Only the results of the co-simulation of the first two experiments are shifted 0,002 seconds to the right in comparing with the results of the simulation in 20-sim. 0,002 seconds are equal to two simulation cycles of the co-simulation. The first delay of 0,001 seconds is caused by that the output of the controller starts at zero and not at 0.1 like the simulation in 20-sim. This can be prevented to set the initial value of the output port of the controller to 0,1. It takes again 0,001 seconds before the input of the FMU interface has influence on the output of the FMU interface. In the last experiment the graph is shifted by 0,001 seconds because now the input in the co-simulation and in the simulation in 20-sim begins both at zero.

This experiment shows that it is possible to change a parameter of the plant model in the TERRA itself. This give still the same results as a simulation done in 20-sim. The FMU interface also works with a input which is not constant.

6.3 Experiment 3: Using a Control System Model for Co-simulation

This experiment shows that the FMU interface also works when a co-simulation is done for a control system model. In these control system model the controller is implemented in TERRA while the plant is implemented in 20-sim. For this control system a model is used which is given as example for the 20-sim editor. This model represents a fluid level control system. Because the example is available in 20-sim, it is possible to simulate to complete system in 20-sim and compare the result with the result of the co-simulation between TERRA and 20-sim.

6.3.1 Model Description

Figure 6.9 shows the fluid level control system in 20-sim. This system consists of a controller and a plant. Figure 6.10 shows the implementation of the plant. The plant interface has two ports:

- An input port with the name *control* and the datatype *real*
- An output port with the name *height* and the datatype *real*

The plant implementation shows a two coupled tank. Fluid input is in the first tank. The fluid input is controlled by the input signal of the plant. The fluid can run through a pipe to a second tank. The second tanks has a fluid output valve. The output signal of the plant gives the value of the height of the fluid in the second tank. Figure 6.11 shows the bond-graph representation of the two coupled tank.

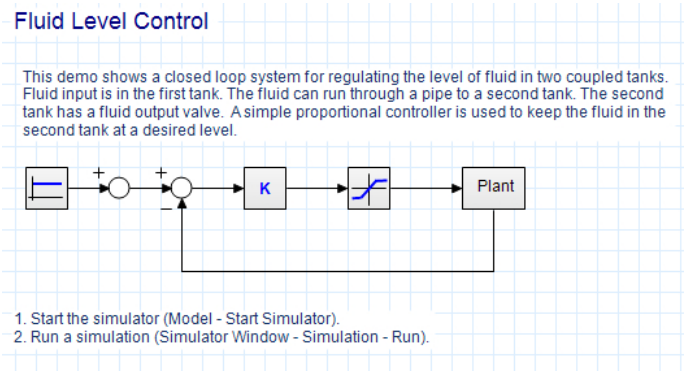

Figure 6.9: Representation of the fluid level control system in 20-sim

The goal of the controller is to keep the height of the fluid in the second tank equal to the reference value. This is done by subtracting the height value from the reference value and multiply the value with the gain factor *Kp*. The value is then limited between the 0 and 0,5 before it is used as control signal for the plant model. In this case the reference value is equal to 1,5 and the gain factor is equal to 10.

For the FMU generation of the plant model the Euler integration method is used with a start time of 0 seconds, a stop time of 10 seconds and a step size of 0,01 seconds. Figure 6.12 shows the implementation of the FMU interface model which is generated by TERRA using the XML file of the FMU. It can be seen that an output and input port with the corresponding writer and reader is generated as expected. Also the port mapping and parameter mappings are visible when clicking on the FMU interface block of the CSP model. It is also possible to change the values for the simulation properties.

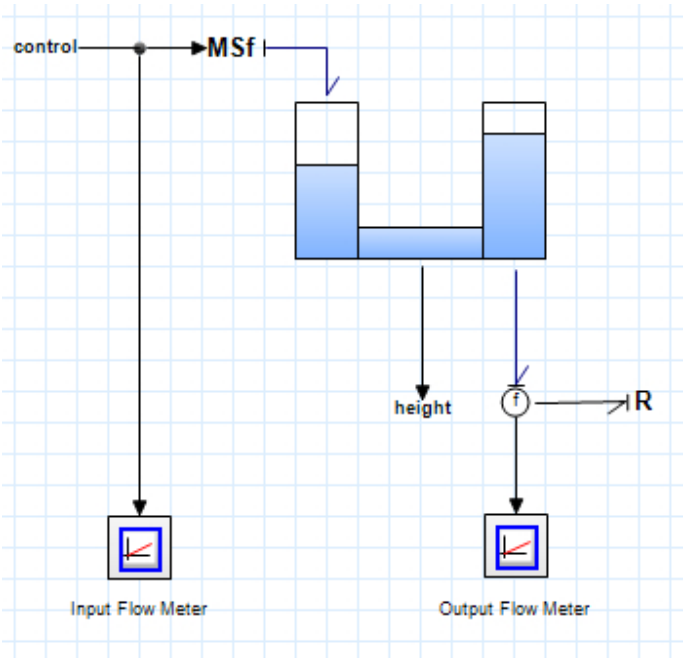

Figure 6.10: Representation of the plant model of the fluid level control system in 20-sim

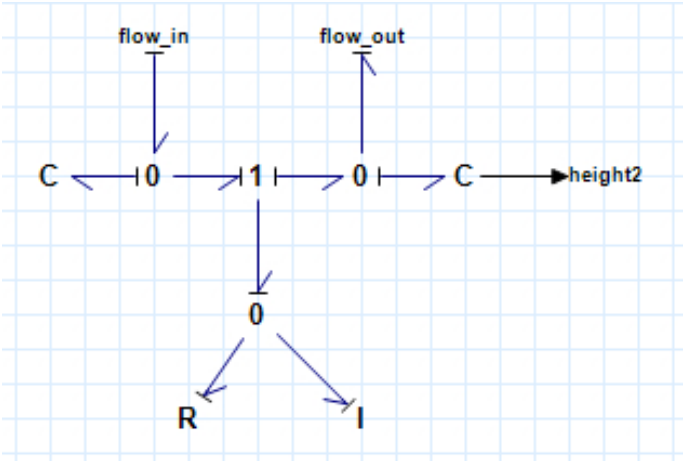

Figure 6.11: bond graph representation of the two coupled tank in 20-sim

6.3.2 Experiment

In this experiment the controller of the architecture editor in TERRA gets the same implementation as the controller used in the fluid level control system in 20-sim. This is done by adding the following code in the *execute()* function of the source file of the code block of the controller:

```
double refValue = 1.5;
double K = 10;
double tempValue = (refValue - height)*K;
if(tempValue < 0)
    control = 0;
else if(tempValue > 0.5)
    control = 0.5;
else
```

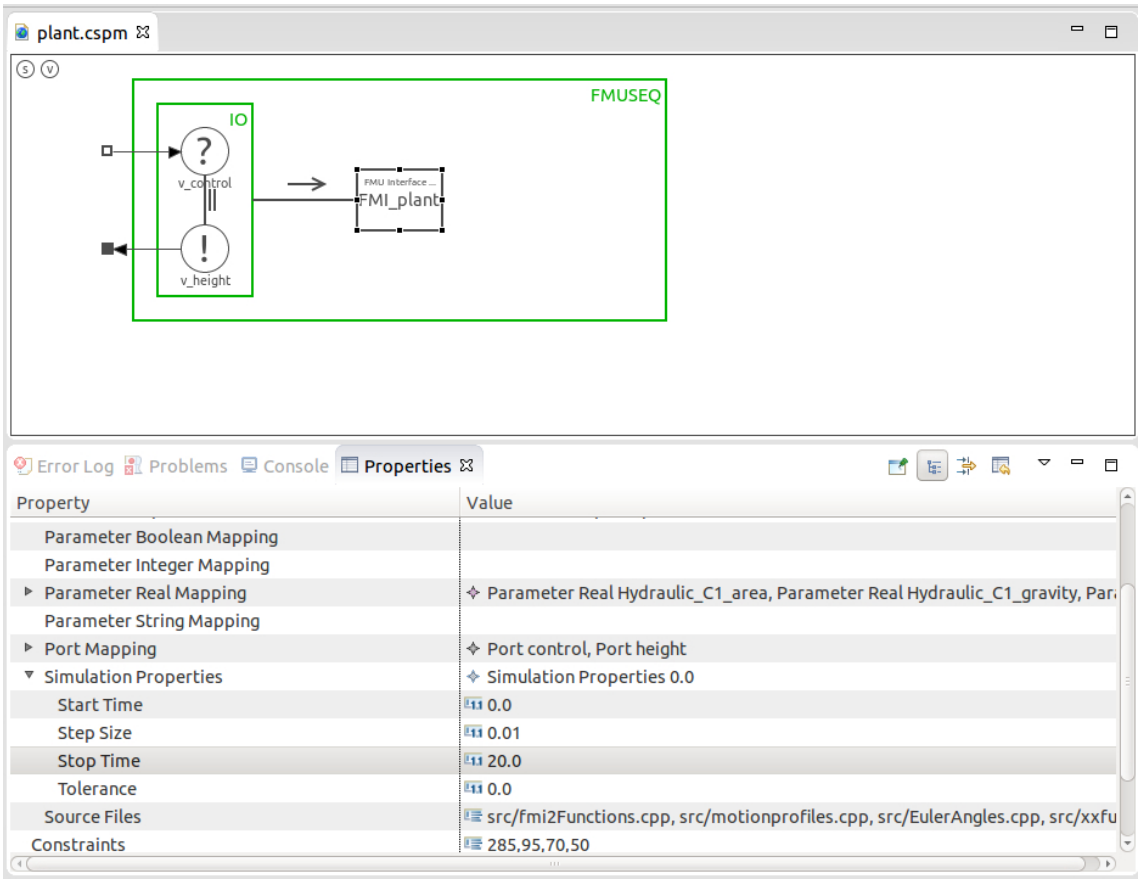

Figure 6.12: CSP model of the FMU interface to the plant model

```
    control = tempValue;
```

where *height* is the variable for the input of the controller and *control* is the variable for the output of the controller. The co-simulation starts at 0 seconds and runs for 20 seconds with a time step of 0,01 seconds. The same simulation is also performed in 20-sim with the same conditions.

The results for both simulations is shown in figure 6.13. The figure shows that both graphs are very similar to each other at the beginning. But in the end the graph of the co-simulation is delayed in comparing with the graph of the 20-sim simulation. But both graphs shows that the height of the fluid in the second tank is kept around the 1,5 meters.

6.3.3 Discussion

The result of this experiment shows that the FMU interface also can be used for co-simulation of controlled systems where the controller is implemented in TERRA and the plant is implemented in 20-sim. The result of the co-simulation shows after some time a delay in the output values of the plant in comparing with the result of the simulation which is done completely in 20-sim. This delays are caused because there exists a delay of one step between the input values of the FMU interface and the output of the FMU interface. And there also exists a delay of one step because the control signal of the controller is based on the value of the height of the previous simulation cycle of the physical system model. So in total a delay of two steps caused the delay in the co-simulation. Figure 6.14 shows the representation of the fluid level control system in 20-sim with two step delays added. These delays represents the same delays which

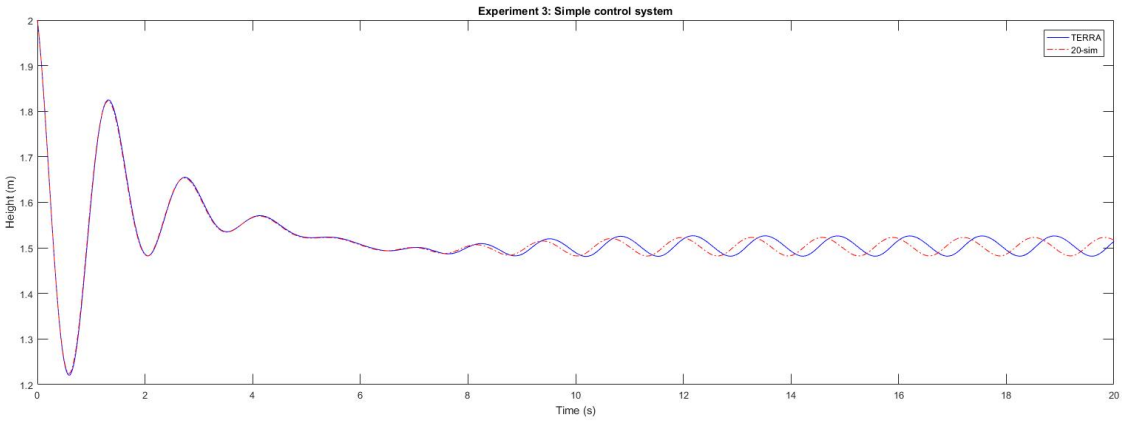

Figure 6.13: Graph with the result of experiment 3

occurs in the co-simulation between TERRA and 20-sim. Figure 6.15 shows the result of the co-simulation together with the result of the 20-sim simulation of the model with the added delays. The results are very similar. So the difference in Figure 6.13 are indeed caused by the added delays. Figure 6.16 shows the results for a simulation time of 100 seconds. Also the results of these simulation are similar. So also for longer simulation times the difference in the results are caused by the delays in the co-simulation.

The delay of two steps is too much for the co-simulation. The ideal case is to have zero delay in the co-simulation. This is not possible because the delay of one time step, which is caused by using the control value for the physical system which is based on the output value of the physical system a simulation cycle back, always exists. But the second delay of a step needs to make as small as possible. So further research must be performed to investigate where the second delay exactly comes from and how it can be prevented.

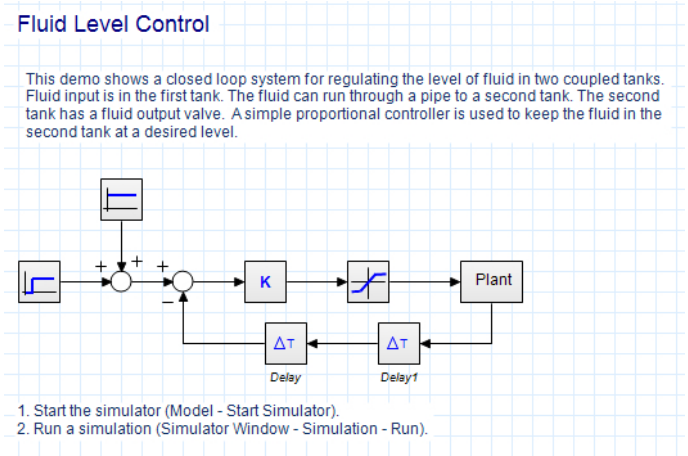

Figure 6.14: Representation of the fluid level control system in 20-sim with some added delays

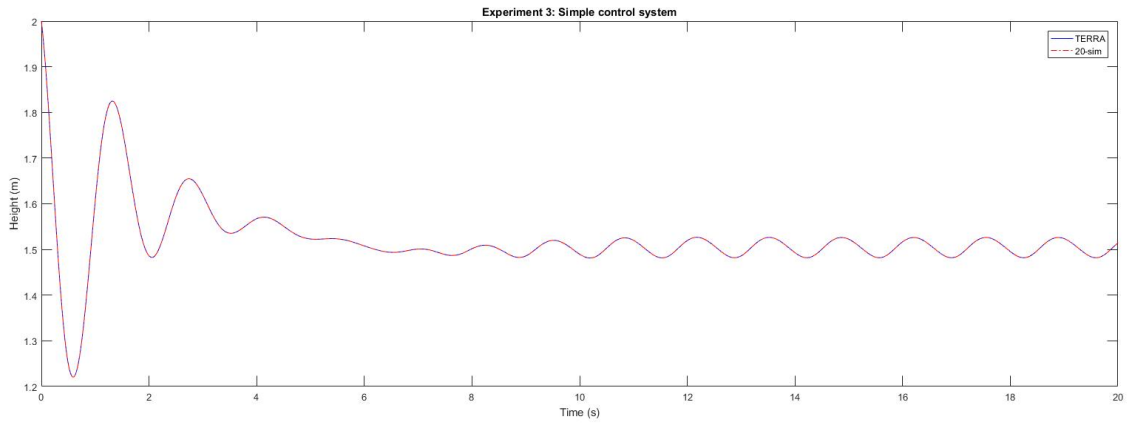

Figure 6.15: Graph with the result of experiment 3 with added delays in the 20-sim simulation

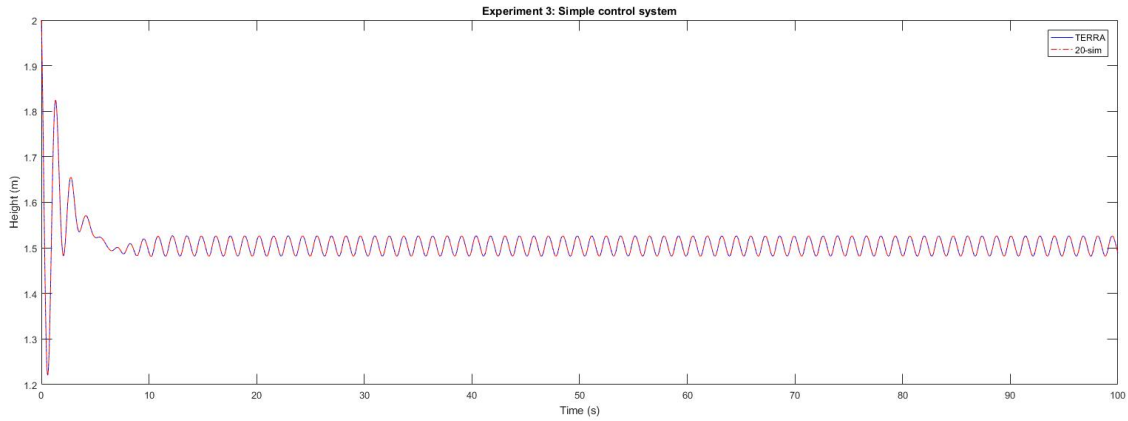

Figure 6.16: Graph with the result of experiment 3 with added delays in the 20-sim simulation and a run of 100 seconds

7 Conclusion and Recommendations

7.1 Conclusions

For the first goal a design is developed which can be used to have support for physical system co-modeling in the architecture editor of TERRA. This design is based on creating a plant model interface in the architecture editor and translate this interface to 20-sim. This can be done by generating a 20-sim file from the plant model interface in the architecture editor. This interface can be used in 20-sim to give implementation to the plant. A meta-model is designed which can be used to create in plan interface model. This model can be used to generate a 20-sim file. With this method it is possible to generate an interface for a model in 20-sim with multiple ports of different data types which are also supported in the architecture editor. Due time constraints and the limit documentation of the design of TERRA, it was not possible to implement this method in the architecture editor.

For the second goal a communication interface is implemented in the architecture editor of TERRA to use 20-sim for a co-simulation. With this communication interface it is possible to simulate a physical system model in combination with controller software which is based on CSP models. This communication interface is based on the FMI standard which describes an interface to an FMU which is based on a model created in 20-sim. The XML file which is part of the FMU can be used to automatically generate a FMU interface in TERRA by using model-to-model translation an model-to-text translation. So the implementation of the communication interface support the model-driven development. This method is reusable for all kind of models in 20-sim. Only the delay of two time steps which occurs during the co-simulation of a control system is to much.

7.1.1 Reflection of the Requirements

In this section a reflection is given on the requirements which are defined in section 3.3.

A Must have

I All meta-models must be based on the CPC meta-model
The meta-model for the plant model interface contains the same attributes which are also used in the CPC meta-model. So the meta-model of the plant model interface is compatible with the CPC meta-model. The base class of the FMU interface meta-model inheriting from the class 'CPPCodeBlockConfiguration'. The class 'CPPCodeBlockConfiguration' is a class from the CPP meta-model which is based on the CPC meta-model. This means that the FMU interface meta-model is also based on the CPC meta-model.

II No code must be added by hand
Meta-models are defined to create 20-sim files and FMU interfaces in the architecture editor of TERRA. These meta-models are translated to code by using the epsilon framework of eclipse. So it is not necessary that code is added by the user to get a working 20-sim file or FMU interface.

III The architecture editor must support the 20-sim tool
The 20-sim tool is used to use the plant model in combination with the architecture editor of TERRA. The design and simulation of the plant model is done in 20-sim itself while there is a connection with TERRA. So there is support for 20-sim in the architecture editor.

IV The architecture editor must support code generation for the LUNA framework
The FMU interface is based on a CSP model. It was already possible to transform CSP

models to code based on the LUNA framework. Only an new FMU interface block is added to the CSP block where also code is generated from. This code works also in combination with the LUNA framework.

V Verification of the models must be supported by the architecture editor
Because the FMU interface is based on CSP models, the verification methods of the CSP models can also ne used for the validation of the FMU interface. So it is not possible to create two ports with the same name or to connect two ports which not support the same data type.

B Should have

I The architecture editor should have the possibility to switch between the real plant and a model of the plant.
In this assignment, all the effort was put in the support to use a physical system model in the architecture editor of TERRA.

II The architecture editor should have support for hardware ports simulation
In this assignment, all the effort was put in the support to use a physical system model in the architecture editor of TERRA. It is possible to model some hardware ports in 20-sim an use the FMU interface to use this models in the architecture editor.

III The architecture editor should have support for simulations on the general computer platform and on a dedicated embedded platform in combination with a model of the plant
At the moment it is only possible to run the FMU interface in combination with the controller software on a general computer platform. It is possible to run the controller software on a dedicated embedded platform and use the model of the plant on a general computer. But to make this possible a communication channel must be created between the controller software on the dedicated embedded platform and the FMU interface on a general computer. An IP/TCP connection can be used for example.

C Could have

I The architecture editor could have support for code generation to Cλash
In this assignment, all the effort was put in the support to use a physical system model in the architecture editor of TERRA. No time was put into the support for code generation to Cλash in the architecture editor. Also the research to make Cλash code from CSP models was performed in parallel during this assignment.

II The architecture model could have support for the ROS framework
In this assignment, all the effort was put in the support to use a physical system model in the architecture editor of TERRA. No time was put into the support of the ROS framework in the architecture editor. Also the research to combine ROS with LUNA was performed in parallel during this assignment.

D Won't have

I The architecture editor won't have support for multiple plant models
No effort was put into the support of multiple plant models in the architecture editor. However with the current implementation of the FMU interface, it should be possible to include multiple plant models in on architecture model.

7.2 Recommendations

Throughout the project several points have been identified that need further improvements.

7.2.1 FMU interface functionalities

During the co-simulation of a control system using the FMU interface a delay of two steps occurs. The ideal case is to have zero delay in the co-simulation. This is not possible because the delay of one time step, which is caused by using the control value for the physical system which is based on the output value of the physical system a simulation cycle back, always exists. But the second delay of a step needs to make as small as possible. So further research must be performed to investigate where the second delay exactly comes from and how it can be prevented.

The 20-sim tool generates a FMU from the model of the physical system with the solver included in the source code. The FMI standard also describes the possibility to use the FMU with a simulation tool. At the moment this option is not supported by 20-sim, but it will probably be supported at the end of the year. This means that 20-sim can be used to simulate the plant model and the FMU is used for the communication between TERRA and 20-sim. The interface of the FMU is for both situations the same, so it should not be necessary to change the implementation of the code generation for the FMI interface. Only the source files which are generated for the FMU by 20-sim have to be changed.

7.2.2 Co-modeling support in TERRA

The translation between the plant model interface in the architecture editor of TERRA to the 20-sim editor needs to be implemented. In chapter 4 a meta-model with the necessary information for this translation is proposed. This translation can directly be implemented in the architecture editor because the meta-model of the plant interface is similar to the meta-model for a component model in the architecture editor. Only the parameters are not included in the component model of the architecture model. So a choice can be made to also implement the support for parameters in the component model or to leave out the parameters in the plant interface.

7.2.3 TERRA functionality

When generating an FMU from a plant model in 20-sim, an amount of C files is generated containing the source code of the dynamics of the plant model. It is also possible that the FMU contains some libraries which implements all the FMI functions. The current version of the meta model for a C++ block contains only support for .h and .cpp files. The Makefile which is automatically generated by TERRA to perform the code compiling and linking is only useful for .cpp files. To support .c files and libraries files it is necessary to include entries for these files in the meta-model for the C++ block. Also the code generation for the Makefile needs to be changed so it can also use .c files and libraries files.

7.2.4 LUNA functionality

For the FMU interface a logger function is necessary which is created during this assignment. At the moment the header and source file for this logger function needs to be included by hand to the rest of the code. It is possible to include this function in the LUNA framework so it is not necessary anymore to include the files by hand.

A Using a Generated FMU from 20-sim in TERRA

This appendix describes how a FMU from a simple model can be generated in 20-sim. And how to use the FMU in TERRA.

A.1 Step 1: Creating a Model in 20-sim

First a simple model of a motor is created in 20-sim. Figure A.1 shows a created *Motor* submodel in the 20-sim editor with an input port with the name *Current* for the current and an output port with the name *Velocity* for the velocity. Figure A.2 shows the implementation of the *Motor* model. The motor is modeled with bond graphs. The model takes as input a current and gives as output a velocity.

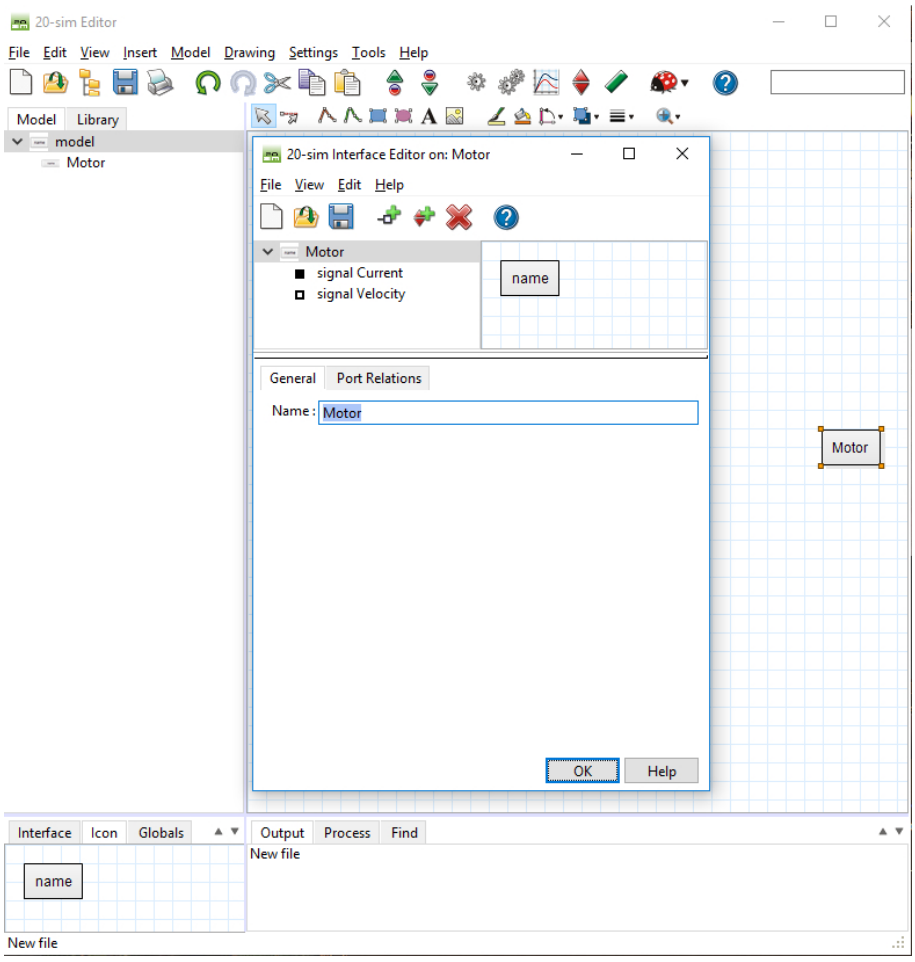

Figure A.1: Motor submodel with interface shown in 20-sim.

A.2 Step 2: Generating a FMU in 20-sim

All the input ports of the model needs to be connected to a signal source. Otherwise it is not possible to start the simulator. Figure A.3 shows the *Motor* model with a source signal connected to the input. Then go to *Tools* -> *Simulator* to start the simulator. The 20-sim Simulator window appears. Go to *Properties* -> *Run* to configure the simulator. Take care that not every integration method is supported by the FMI in 20-sim. For this example the *Euler* integration method is selected. Close the configuration window to go back to the 20-sim Simulator itself.

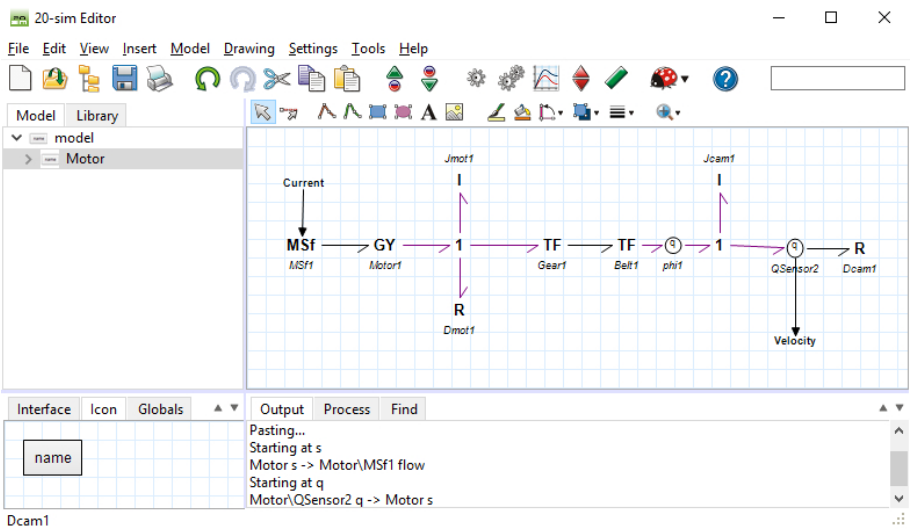

Figure A.2: Bond graph model of a motor in 20-sim.

Go to *Tools -> Real Time Toolbox -> C-Code Generation* to open the C-Code Generation window. Figure A.4 shows the window where it is possible to generate a FMU from a model. Select in the *Target* list the option: *FMU 2.0 export for 20-sim submodel.* Select at *Submodel* the submodel of the motor. It is also possible to change the output directory. Click on *OK* to generate the FMU. **Take care that at the moment 20-sim only supports the generation of FMU's under Windows.**

In the output directory a new directory is generated with the FMU of the motor model. The FMU contains an XML file which gives information about the motor model and some information for the simulation. The FMU also contains c files and header files which contains the functions and implementations to simulate the motor model.

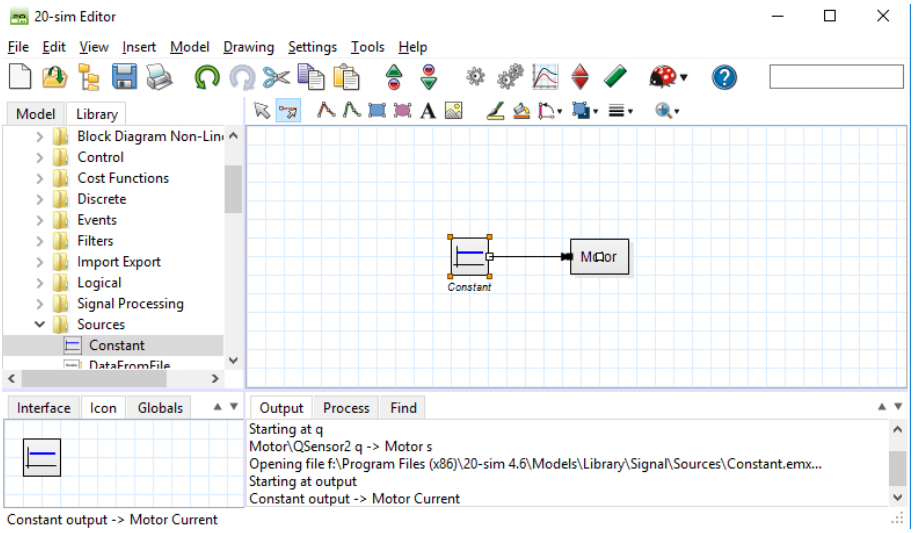

Figure A.3: Adding a source to the Motor submodel.

A.3 Step 3: Translate the XML-file of the FMU to CSPm

Create in TERRA a new TERRA project. A new directory for the project is created in the eclipse workspace. Copy the XML file from the generated FMU to the model directory which is located in the TERRA project directory. Change the name of the XML file to the same name as how

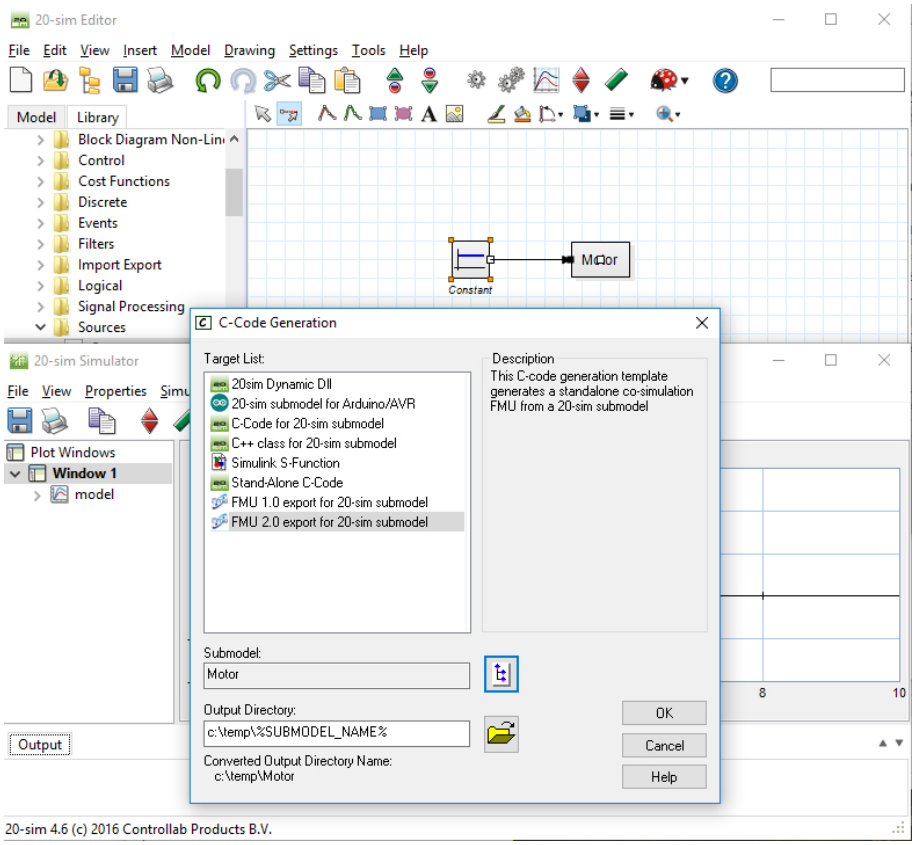

Figure A.4: Generate a FMU of the Motor submodel.

the model is called in 20-sim to prevent getting warnings from the TERRA application. In the TERRA application right click on the model directory of the TERRA project and select *Refresh*. Now the XML file appears in the overview. Right click on the XML file and go to *TERRA -> Transform FMI to CSPm*. A CSPm file is generated which contains a CSP model with the FMU interface block as shown in figure A.5

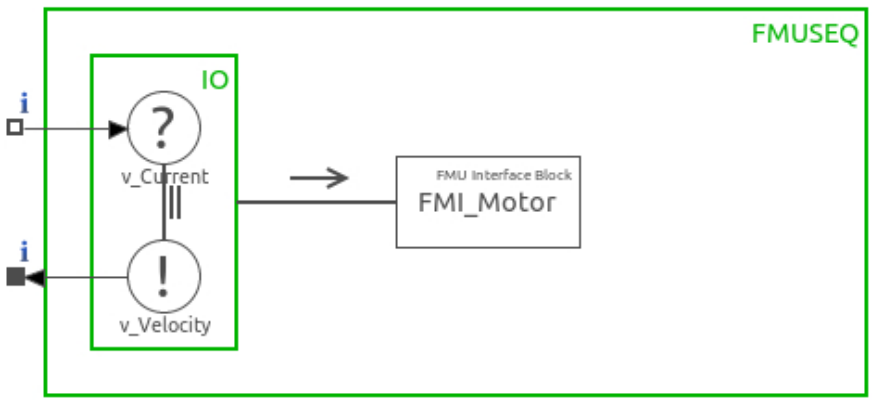

Figure A.5: CSP model of the FMU interface for the motor model.

Clicking on the FMU interface block in the CSP model shows the properties of the FMI component. The callback functions can be changed if the current functions are not supported on the platform where the model will be executed on. Also the simulation properties can be changed

but take care that the it can generate errors during simulation because it is not always possible to change the settings. The step size for the simulation can be fixed for some models. For the stop time the value which is getted from the FMI file is used as standard value. When this value is changed to zero no stop time is defined for the co-simulation.

A.4 Step 4: Adding the FMI CSPm model to an Architecture Model

A new architecture model can be added in the TERRA project created in the previous step. To include the FMU interface model in the architecture editor, add a *External Model* to the architecture editor. Select as external model file the CSPm file which contains the CSP model of the FMU interface. Now it is also possible to add for example a controller block to the architecture editor.

A.5 Step 5: Code Generation of the Architecture Model

After implementing the complete architecture model, it is possible to generate code from the complete model. For the architecture model code can be generated by clicking on *Diagram -> Code generration -> CPP LUNA* in the architecture editor. Also for every CSP model which is included in the architecture model code generation can be performed in the same way as for the architecture editor. This also for the FMU interface model. After code generation for the FMU interface model a directory is visible in the TERRA project which contains the code for the FMU interface. The source files which are located in the in FMU needs to be added to the source directory and header directory which are created during the code generation. The following files needs to be added to the source directory:

- EulerAngles.c
- fmi2Functions.c
- motionprofiles.c
- xxfuncs.c
- xxinteg.c
- xxincerse.c
- xxmatrix.c
- xxmodel.c
- xxsubmodel.c
- xxTable2D.c

It is also necessary to replace the .c extension of each file to the .cpp extension because the current MakeFile does not support .c files. The following files needs to be added to the include directory:

- EulerAngels.h
- fmi2Functions.h
- fmi2FunctionTypes.h
- fmi2TypesPlatform.h
- fmiGUID.h

- motionprofiles.h
- xxfuncs.h
- xxinteg.h
- xxmatrix.h
- xxmodel.h
- xxsubmodel.h
- xxTable2D.h
- xxtypes.h

Also add the files *logger.cpp* and *logger.h* to the rest of the files because otherwise it is not possible to log information and error messages which are generated by the FMU interface. Without these files the FMU interface does not work.

A.6 Step 6: Running the Architecture Model

Now it is possible to run the software and using the FMU interface for co-simulation. The compilation and execution of the application is done in the same way as for all application created by TERRA.

Bibliography

Bezemer, M. M. (2013), *Cyber-physical systems software development: way of working and tool suite*, Ph.D. thesis, University of Twente, Enschede.

Bezemer, M. M., R. J. Wilterdink and J. F. Broenink (2011), Luna: Hard real-time, multi-threaded, csp-capable execution framework.

Broenink, J. F., Y. Ni and M. A. Groothuis (2010a), On Model-driven Design of Robot Software using Co-simulation, in *SIMPAR, Workshop on Simulation Technologies in the Robot Development Process*, Ed. E. Menegatti, pp. 659 – 668, ISBN 978-3-00-032863.
http://www.2010.simpar.org/ws/sites/SimTechRDP/04-SimTechRDP.pdf

Broenink, J. F., Y. Ni and M. A. Groothuis (2010b), On model-driven design of robot software using co-simulation.

Bruyninckx, H., M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi and D. Brugali (2013), The BRICS component model: a model-based development paradigm for complex robotics software systems, in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ACM, pp. 1758–1764.

FMI-standard (2014), Functional Mock-up Interface for Model Exchange and Co-Simulation, Technical report, Modelica Association,
https://svn.modelica.org/fmi/branches/public/specifications/v2.0/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf.

FMI-standard (2016), FMI.
https://www.fmi-standard.org/

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice Hall International.
http://www.usingcsp.com/cspbook.pdf

Kuipers, F. P., T. Wester, J. Kuper and J. Broenink (2016), Mapping CSP Models for Embedded Control Software to Hardware Using CλaSH.

Kranenburg-de Lange, D. et al. (2012), Dutch Robotics Strategic Agenda-Analysis, Roadmap and Outlook.

Ni, Y. (2015), *System Design Support of Cyber-Physical Systems, a co-simulation and co-modelling approach*, Ph.D. thesis, University of Twente, Enschede, Netherlands, doi:10.3990/1.9789036538588.

Scattergood, B. and P. Armstrong (2011), CSPm: A Reference Manual.
http://www.cs.ox.ac.uk/ucs/cspm.pdf

The Eclipse Foundation (2014), Epsilon.
http://www.eclipse.org/epsilon/

The Eclipse Foundation (2016), Eclipse Modeling Framework (EMF).
http://www.eclipse.org/modeling/emf/

van der Werff, W. M. (2016), *Connecting ROS to the LUNA embedded real-time framework*, Msc report 018ram2016, University of Twente.