Communicating Process Architectures 2017
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2017
© 2017 The authors and IOS Press. All rights reserved.

Co-simulation Design towards Cyber-Physical Robotic Applications

Leveraging on FMI Standard and CSP Semantics

Zhou LU [1], and Jan F. BROENINK

Robotics and Mechatronics, CTIT Institute, Faculty EEMCS,
University of Twente, The Netherlands

Abstract.
Designing software controllers for multi-task automated service robotics is becoming increasingly complex. The combination of discrete-time (cyber) and continuous-time (physical) domains and multiple engineering fields makes it quite challenging to couple different subsystems as a whole for further verification and validation. Co-simulation is nowadays used to evaluate connected subsystems in the very early design phase and in an iterative development manner.

Leveraging on our previous efforts for a Model-Driven Development and simulation approach, that mainly focused on the software architecture, we propose a co-simulation approach adopting the Functional Mock-up Interface (FMI) standard to co-simulate the software controller with modelled physical plant dynamics. A model coupling approach is defined that involves the model transformation from a physical plant model implementing the FMI interface (denoted as a Functional Mock-up Unit, FMU) to a Communicating Sequential Processes (CSP) model. The Master Algorithm is (semi-)automatically generated from a co-simulation model that is formalised with CSP syntax to orchestrate the communication between different FMUs. Additionally, an optimized algorithm is defined to compensate for the artificial delay existing in a feedback loop. Finally, an example is used to illustrate the co-simulation approach, verify its working (at least, for this example) and to analyse the timing compensation algorithm.

Keywords. robotics, CPS, co-simulation, FMI, master algorithm, CSP semantics

Introduction

Designing software controllers for multi-task automated service robotics is becoming increasingly complex. The combination of cyber-physical domains and multiple engineering fields makes it quite crucial to evaluate the heterogeneous system as a whole. This is due to the combination of discrete-time, discrete-event, and continuous-time models stemming from different engineering fields, like software engineering, formal modelling, control engineering, electrical engineering, mechanical engineering. A typical top-level structure of a generic robotic Cyber-Physical System (CPS) is shown in Figure 1. Furthermore, different teams or external suppliers may use variety of tools to model different subsystems. Integration of such sets of subsystems is challenging, especially at the end of the design phase, where models are quite detailed, making this integration quite challenging and error-prone, thus costly.

[1]Corresponding Author: *Zhou Lu, Robotics and Mechatronics, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.* Tel.: +31534894419; E-mail: `z.lu@utwente.nl`.

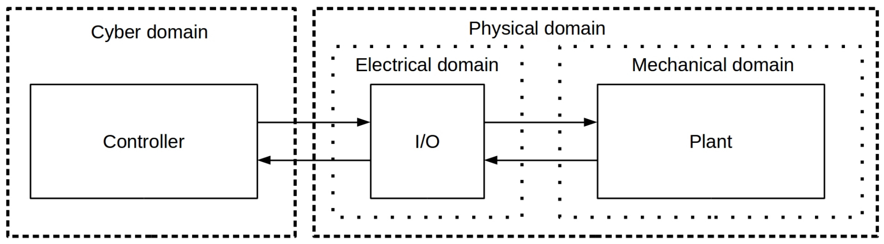


Figure 1. Typical robotic CPS: combination of multiple domains

Over the previous decade, merging control, systems, networking and software engineering built on the principles of multi-disciplinary modelling, model-driven design (MDD) and co-simulation, has become relevant research activities [1,2]. The Modelica language and the NS-2 network simulator are integrated to simulate Networked Control Systems in [3]. The Crescendo technology is used in [4] which allows discrete-event models expressed in the Vienna Development Method notation and continuous-time models expressed using 20-sim to run in their separate simulators as a true co-simulation. However, in the above work and other similar work [5], simulators involved in co-simulations are tightly coupled with each other in their own semantics that makes them less flexible and less extensible. Another example is the Ptolemy project [6,7] studying modelling and simulation of concurrent, real-time, embedded systems. It focuses on well-defined models of computation that govern the interaction between components. But similarly, the Ptolemy does not rely on standard notations.

Co-modelling and co-simulation are the promising approaches targeting the challenges with respect to heterogeneity in modelling of CPS. Combined modelling (co-modelling) allows models made in different modelling approaches be connected via a specifically defined interface. This under the condition that the semantics of the involved modelling approaches cover coupling over interfaces. Modelling approaches derived from the CPC (Component Port Connector [8]) meta-model comply with this condition. The modelling approaches used in this paper, block diagrams, CSP-based software descriptions, bond-graph based physical plant models, do comply with this condition, of course. Consequently, the reliability of software and the confidence in the design will both be increased.

However, as illustrated before, realizing a co-modelling and co-simulation methodology and infrastructure is still challenging, especially:

- Coupling of different domain models and tools in a standardized way.
- State consistency between Discrete-Event (DE) controller models and Continuous-Time (CT) plant models.

The Functional Mock-up Interface (FMI) [9] standard, initiated by auto-mobile industries within the ITEA2 MODELISAR project [10,11], and currently maintained by the Modelica Association, has been designed to support the exchange, interoperation and coordination of model components or subsystem models designed with different modelling tools [12]. Recently, with the development pushed by both industries and academia, more than 100 simulation software tools claim to support FMI [9]. In fact, FMI appears to become the de-facto standard for co-simulation and model exchange in CPS co-design.

With FMI, system engineers do not need to work with domain-specific languages or tools directly. Instead, they need to consider how to transform and import domain implementations and models, i.e. to couple different domain models specified by FMI in a co-simulation environment. Furthermore, data exchange and synchronization must be properly managed to prevent state inconsistency between different subsystem models.

The goal of this paper is to use the FMI standard for co-simulation and process-oriented approaches from the Communicating Sequential Processes (CSP) algebra, to construct a co-simulation facility for cyber-physical robotic applications. We build on our previous work on MDD and co-simulation [13,14]. We use CSP semantics to realize the synchronisation needed in this co-simulation. Towards this goal, we realize a coupling approach following the FMI standard for the software controller-architecture model and the plant model, which are formalised as *CSP* model (DE) and *bond-graph* model (CT), respectively.

Furthermore, an orchestrator skeleton can be automatically generated to build up the *Master Algorithm* (MA), which is responsible to control the data exchange between different subsystem models and to handle the synchronization during the entire co-simulation.

In Section 1, some background information about the FMI standard and about our modelling tools is presented. Then, an approach for coupling different domain models based on meta-modelling and model-to-model transformation, is illustrated in Section 2. In Section 3, our method to generate CSP-based C++ code implementing the Master Algorithm is presented. Additionally an optimized algorithm is proposed to compensate for the artificial delay that exists in a feedback loop. A practical example using our approach for co-simulation is given in Section 4. Finally, in Section 5 we provide our conclusions and sketch some future work.

1. Background

Functional Mock-up Interface

FMI is a tool-independent standard defining a generic Application Programming Interface (API) that can be adopted by different modelling and simulation tools. FMI provides two specific types of interfaces to support two time-driven control modes:

- FMI for co-simulation (FMI-CS)
- FMI for model exchange (FMI-ME)

FMI-CS is designed for both coupling of simulation tools, and coupling with subsystem models, which have been exported by their simulators together with their own solvers (i.e. numerical simulation algorithms) as runnable code. Using FMI-CS, a subsystem model can be represented as a stand-alone black-box simulation component that can be executed independently in any simulation environment.

FMI-ME defines an interface to the dynamic model, which is described by differential, algebraic and discrete equations. But unlike FMI-CS, the solver to the model equations should be provided by the simulation environment itself.

In the context of this paper, FMI-CS 2.0 is used as co-simulation interfacing standard.

A model implementing the FMI interface is denoted as a Functional Mock-up Unit (FMU). The subsystem models of a typical cyber-physical robotic system, e.g., the sequence-controller model, the loop-controller model and the plant model, can be encapsulated as individual FMUs. In Figure 2, an example of a co-simulation model for such a system is shown, specified as a model containing interconnected subsystem FMUs and can be simulated as a whole in a co-simulation environment. Those different subsystem FMUs can be exported from not necessarily different modelling tools.

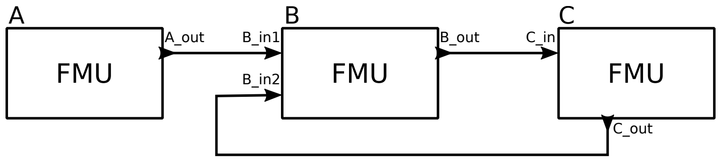


Figure 2. A co-simulation model consisting of interconnected FMUs

In a co-simulation model, each FMU consists of an XML model-description file together with a shared library and/or C source files implementing the corresponding interface. The

XML model-description file following an FMI description schema defines the static metadata of the model in a standardized way, including the definition of all (exposed) variables and their attributes such as name, unit, initial value, etc. The XML file also contains the configuration and the capability information of the FMU regarding simulation, i.e., the start and the stop time, the integration step size and the ability flag of handling variable communication step size, etc. The shared library and/or C source files, provides the model implementation (and its solver in the case of FMI-CS) together with the FMI API. The shared-library solution is especially used if the FMU provider wants to protect intellectual property rights or to allow an automatic import of the FMU in another simulation environment, hence it is preferred in our approach due to the latter reason.

In FMI standard, a Master-Slave mechanism is used during co-simulation. The communication between FMUs (slaves) is orchestrated by a master program, referred to as the *Master Algorithm* (MA). Slaves are assumed to communicate with the master only. The MA orchestrates the data exchange at specific discrete communication points, by providing calling sequences of FMI API, i.e. executing the FMUs and advancing time, getting and setting exposed variable values, to ensure the state consistency between different FMUs. In Figure 3, the Master-Slave structure when coupling subsystem models using FMI-CS interfaces is illustrated.

The MA is a crucial component, as it is essential for the co-simulation. However, it is not part of the FMI standard, implying that a sophisticated MA needs to be developed for the problem at hand.

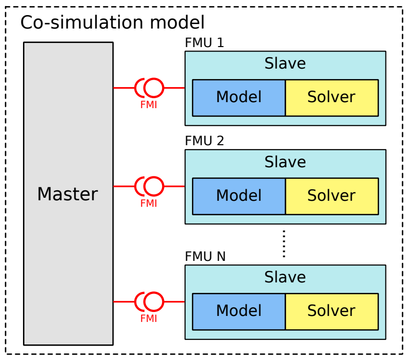


Figure 3. FMI for co-simulation: coupling with subsystem models

Tools for Modelling and Co-simulation

TERRA [15] is a CSP-algebra based graphical modelling tool suite and is designed for embedded control software architecture modelling (DE domain). Additionally, it provides model-to-model transformation and C++ code generation to facilitate further verification and validation.

TERRA provides two abstraction layers to create a hierarchical structure for a system: the *architecture* layer and the *external sub-model* layer. The latter is also identified as the functional layer in this paper. On the architecture layer, model components such as block, link, and port are used to describe the architecture composition, the component dependencies, and to define connections with the ’outside’ world (e.g., connections with an FPGA I/O board). Each model block on the architecture layer contains an external sub-model describing functional details, e.g., sets of I/O, numerical calculation and sequence of behaviours.

All sets of I/O that go to/come from the external sub-model are ’exposed’ directly to the corresponding architecture model block as *ports* with incoming/outgoing directions. Through these ports the external sub-model can be linked to other architecture components. In Figure 4, a classic loop control example modelled in TERRA is shown. The top half is the architecture model while the bottom half contains corresponding functional external sub-models. On the architecture layer, the arrowed lines each linking two ports are defined as CSP communication channels. Each port linked by an arrowed line on the architecture layer is connected to a CSPWriter or a CSPReader on the functional layer. Therefore, a link, i.e., a CSPChannel on the architecture layer is actually between a pair of CSPWriter and CSPReader that exist in different external sub-models. On both layers, models are formalized with CSP syntax, e.g., all model blocks on the architecture layer are in *parallel* and all communication channels/links on both layers are supposed to work in a *waiting-rendezvous* manner (unless the channel is buffered).

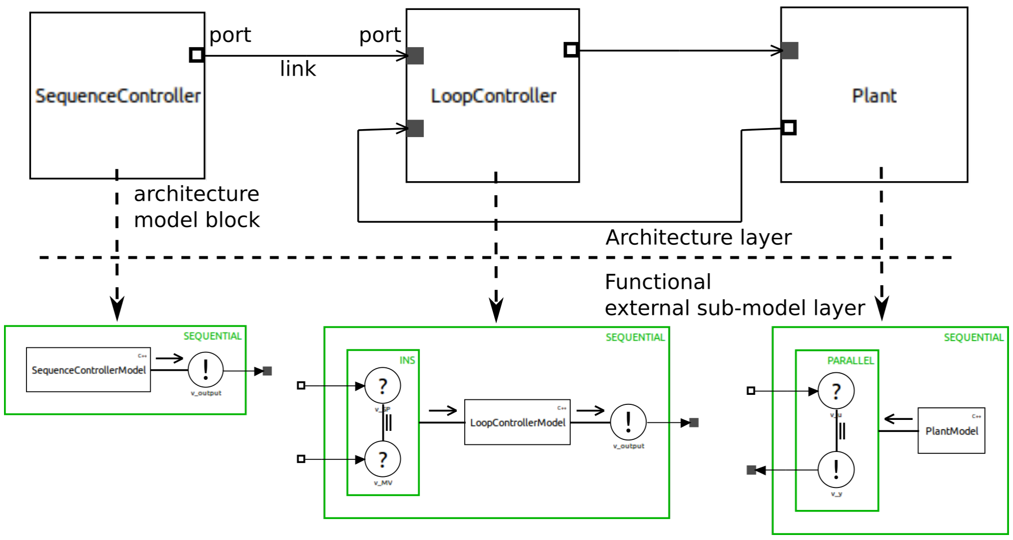


Figure 4. Example of a basic control-loop design using TERRA models

20-sim [16] is a graphical modelling tool, capable of modelling plant dynamics using bond graphs (CT domain), and modelling control laws by iconic diagrams within which the control algorithms are presented as dynamic equations.

Currently, 20-sim supports exporting both plant models and control law models as individual FMUs with interfaces specified by FMI-CS. However, this exporting has restrictions: only basic numerical solvers are exported, and not all modelling constructs are covered. Fortunately, these restrictions do *not* restrict the work for this paper.

2. Coupling Different Domain Models

In Figure 5, the co-simulation design flow derived from our previous work in [14] is shown. Step 1 to Step 3 concentrate on constituting a co-simulation model containing interconnected FMUs. Step 4 and Step 5 are designed towards generating the Master Algorithm and performing co-simulations.

This paper contributes to Step 3 and Step 4, i.e. coupling different domain models following FMI standard and generating CSP-based C++ code implementing the Master Algorithm to orchestrate co-simulations.

In this section, we first introduce an approach for coupling different domain models using TERRA and 20-sim.

To achieve the co-simulation for a CPS, e.g, a robotic system containing a software controller and a plant, the different domain models need to be coupled in a proper way to constitute a co-simulation model. We first create a CSP-based architecture model to abstract the system composition in Step 1, to interconnect different subsystem models. Therefore,

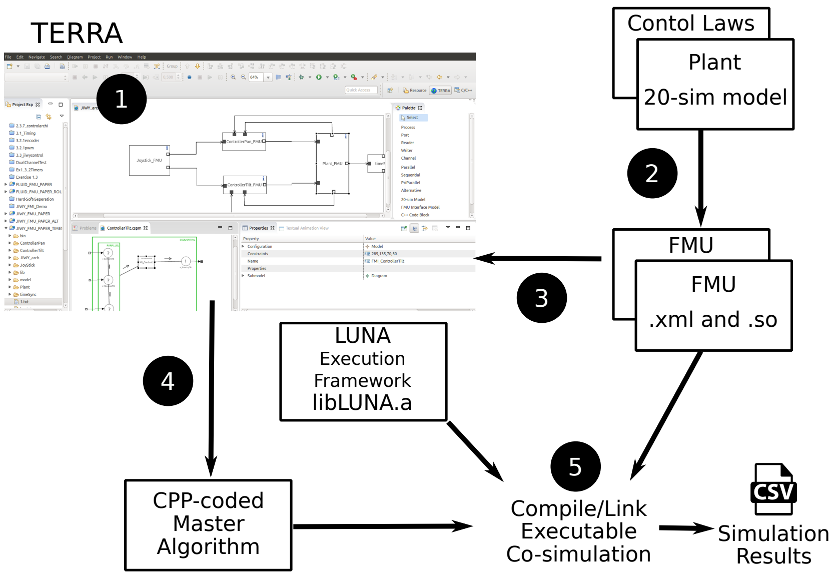


Figure 5. A co-simulation design flow

such a model is a co-simulation model. The architecture model blocks are place holders for further detailed design.

In Step 2, first detailed functional models need to be constructed with proper tools, e.g., control laws and plant dynamics are modelled in 20-sim. In this paper, we assume that these models are already available. To constitute a co-simulation model, the transformation between different formalisms must be done. For instance, the control laws and the plant dynamics are intended to be 'filled' into the corresponding functional layer of different architecture model blocks. 20-sim supports exporting FMUs specified by FMI-CS. As a result, a 20-sim model can be provided as a shared library wrapped by FMI-CS interface together with an XML model description file.

Consequently, the goal of Step 3 is to solve the transformation between the FMI XML formalism to the TERRA CSP formalism to couple these different domain models. To achieve this, sets of Eclipse plug-ins have been developed and integrated with TERRA using its Model-Driven Design methodology. A meta-model is generated from the FMI XML schema, as shown in Figure 6. This meta-model has been used to define a parser for parsing FMU XML files to TERRA CSP models to be imported as a functional external sub-models of their corresponding architecture model blocks.

In Table 1, we present the most important mapping relations between the FMI XML schema and the TERRA CSP definition. These mapping relations define the model-to-model transformation rules, which are formulated in the Epsilon Transformation Language (ETL) [17,18]. CSPModel-FMU in Table 1 represents a CSPModel process with an FMU interface configuration and it is denoted as an FMU interfacing model (the square block in the middle of the right half of Figure 7). The FMU interfacing model generally abstracts the access to an black-box FMU. In FMI XML schema, all variables of an FMU are defined as *ScalarVariables*. In TERRA, only the exposed variables with *input/output* causality type are transformed to variables flowing through CSPWriters or CSPReaders that are connected to certain *ports* and are linked by CSPChannels. The *ports* are exposed to the architecture layer and can be connected with other subsystem models.

Other model properties such as co-simulation capabilities, default experiment configuration and parameter mapping, are transformed to model properties of an FMU interfacing model. The model properties mentioned above are important when generating C++ code implementing the Master Algorithm for co-simulation. This is discussed in the following section. More technical details of the model-to-model transformation are presented in [19].

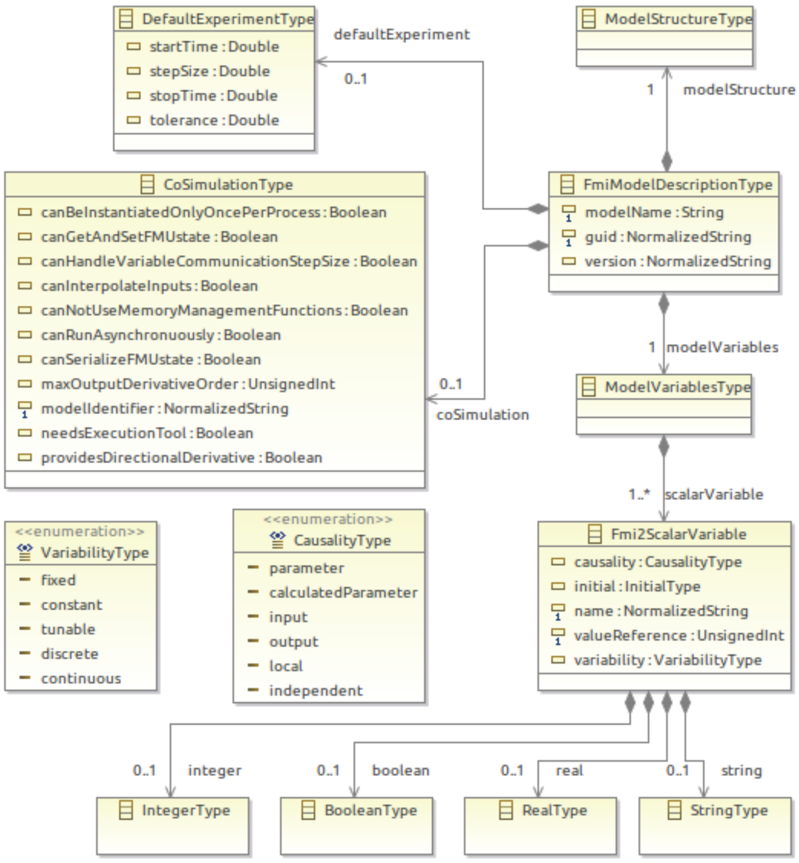


Figure 6. Simplified FMI XML schema meta-model

In Figure 7, an example of an original 20-sim model (left) and its transformed TERRA CSP model (right) is shown. The plant model represents the physical part of a fluid-level control system which is continuous in time. The input signal ’control’ and the output signal ’height’ are defined as exposed variables.

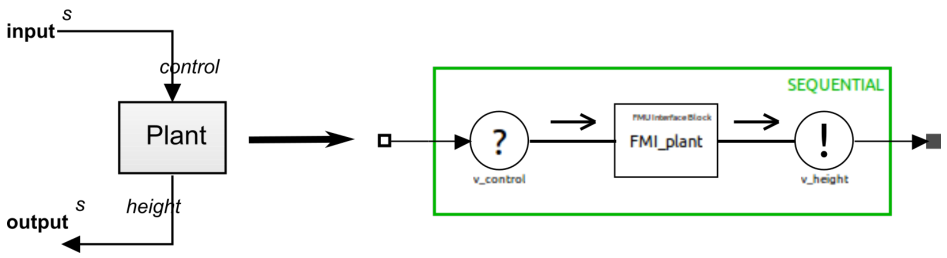


Figure 7. Example of transformation from a 20-sim model to a TERRA CSP model

3. Master Algorithm

The FMI standard only specifies the API that FMUs must implement, but does *not* specify how to perform the co-simulation. A sophisticated MA must be developed to orchestrate the co-simulation by calling the FMI API functions on interconnected FMUs. Luckily after obtaining a co-simulation model in TERRA, this can be done semi-automatically by leveraging on the CSP rendezvous communication and TERRA’s code generation facility, to generate CSP-based C++ code implementing the MA.

Table 1. Partial mapping relations between the FMI XML schema and TERRA CSP

FMI XML Schema Elements	FMU Model Properties	TERRA Model Objects	TERRA Model Properties
fmiModelDescription	*modelName* *guid*	CSPModel-FMU	*modelDescription*
CoSimulation (capabilities)	*canHandle-VariableCommunication-StepSize* *canGetAndSetFMUstate*	CSPModel-FMU	*FMUProperties*
DefaultExperiment (configuration)	*startTime* *stopTime* *stepSize*	CSPModel-FMU	*simulationProperties*
ScalarVariable *if causality type is parameter*	*name* (variable name) *start* (initial value) *valueReference* *type* (datatype)	CSPModel-FMU	*parameterXXXMapping* (XXX is datatype)
ScalarVariable *if causality type is input/output*	*name* (variable name) *valueReference* *type* (datatype)	CSPModel-FMU	*portMapping*
ScalarVariable *if causality type is input/output*	*name* (variable name) *type* (datatype)	Variable	*name* (prefix 'v') *type* (datatype)
ScalarVariable *if causality type is input/output*	*name* (variable name) *type* (datatype)	CSPWriter CSPReader	*name* (prefix 'w'/'r') *type* (datatype)
ScalarVariable *if causality type is input/output*	*name* (variable name) *type* (datatype)	Port	*name* *type* (datatype) *direction* (OUTGOING/ INCOMING)

In FMI-CS mode, the MA generally involves coordinating data exchange between interconnected FMUs and advancing time. The FMI standard provides an example state machine of calling sequence from master to slave [20]. Three sub-phases are represented as follows:

- *Instantiation and Initialisation phase* where the subsystem FMUs are instantiated and initialized by calling four FMI functions: *fmi2Instantiate*, *fmi2SetupExperiment*, *fmi2EnterInitializationMode* and *fmi2ExitInitializationMode*.
- *Simulation phase* where the simulation computation is performed. The calculation until the next communication point is performed with function *fmi2DoStep*. Arguments *currentCommunicationPoint* and *communicationStepSize* are defined as the current communication point of the master tc_i and the communication step size hc_i, respectively. The latter must be greater than zero in FMI 2.0 and the slave must integrate (simulate) until time instant $tc_{i+1} = tc_i + hc_i$. The MA can propose a fixed communication step size or a variable communication step size to the FMU, to define the communication points and the fmi2DoStep must synchronize to these points by always integrating exactly to $tc_i + hc_i$ (if the FMU can accept the proposed hc_i).
- *Termination phase* where the solution at the final time of the co-simulation can be retrieved. Normally *fmi2Terminate* and *fmi2FreeInstance* functions are called at the

end of this phase.

Besides, *fmi2SetXXX* and *fmi2GetXXX* functions are used to update inputs and retrieve output of FMUs, where XXX represents the used datatype (Real, Integer, Boolean or String). In both the Instantiation and Initialisation phase and Simulation phase, the fmi2SetXXX and fmi2GetXXX can be called by the MA, but there is an additional restriction that it is not allowed to call fmi2GetXXX functions after fmi2SetXXX functions without an fmi2DoStep call in the Simulation phase.

By carrying out Step 3 in Figure 5, the subsystem FMUs can be embedded in a TERRA architecture model. Connections between FMUs are defined by directed links representing CSP communication channels. In Figure 8 a Master-Slave structure of two interconnected FMUs in TERRA is shown, containing both an architecture model diagram and two corresponding functional external sub-model diagrams. Since the FMUs are provided as shared libraries, the MA must provide suitable ways to access them, in order to perform the different phases in correct order as well as to carry out co-simulation at specific communication points.

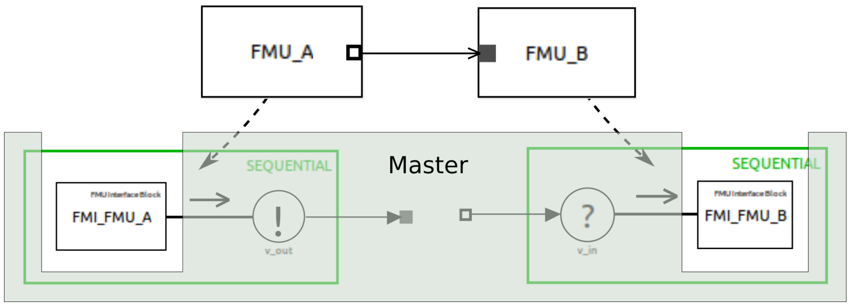


Figure 8. Example of interconnected FMUs in TERRA

We have developed an FMU interface meta-model to abstract the interfacing to an FMU as a CSPModel process (i.e., the CSPModel-FMU in Table 1), and to represent other simulation properties and co-simulation capabilities. In Figure 9, the explicit meta-model of our approach is shown. In the middle of the figure, *FMIModel* class is the abstraction of the subsystem model, i.e., the FMU which the MA is going to orchestrate. This class inherits from the CPPCodeBlockConfiguration class of the CPP meta-model implementing a C++ code block in TERRA. The parent class of CPPCodeBlockConfiguration is the ICPCExternalToolConfiguration interface class, which is part of the CPC (Component-Port-Connector) meta-model, from which the TERRA CSP meta-model is derived [15]. In this way, interfacing to the FMU is abstracted as a C++ code block in the form of a CSPModel process. Consequently, benefiting from the C++ code generation features provided by the CPP meta-model, it is possible to easily create C++ coded algorithms to access the FMU instances provided as shard libraries.

Table 2 is the mapping relations between the generated C++ class functions of each FMU interfacing model and FMI co-simulation phases as well as the FMI API functions to be invoked by the C++ class functions. In general, the Instantiation and (partial) Initialisation phase is executed in the *constructor* of the generated FMU interfacing model class. Performing of simulation steps is carried out by calling the fmi2DoStep iteratively in the *execute()* function. Simulation, i.e. executing the model, of the slave FMU is done by its own solver using the proposed communication step size until the given time instant. Before and after a call to fmi2DoStep, the fmi2SetXXX and fmi2GetXXX functions are used to update inputs and retrieve outputs of an FMU, which are assigned to reference variables flowing through CSPReaders and CSPWriters, respectively, to communicate with other FMUs. When the given final time is reached by the co-simulation, certain solutions can be triggered. Currently in our work, a simple termination of the co-simulation is invoked.

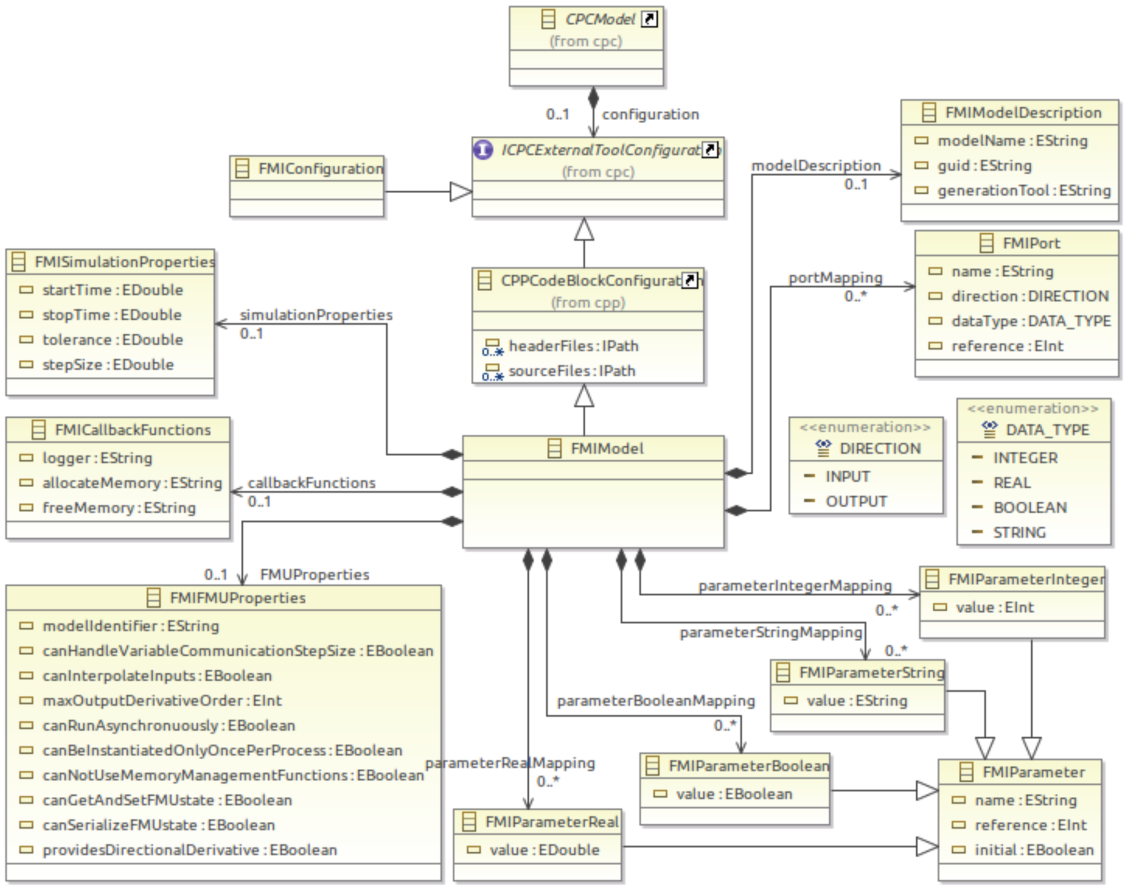


Figure 9. FMU interface meta-model

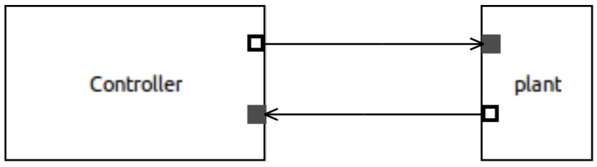
Table 2. Mapping relations between the generated C++ class functions and FMI co-simulation phases

FMU Interfacing Model: Generated C++ Class Functions	FMI Co-simulation Phases	FMI API Functions To Be Invoked
FMI_FMUName(IO parameter,...) (constructor)	Instantiation and (partial) Initialisation	*fmi2Instantiate* *fmi2SetupExperiment* *fmi2EnterInitializationMode*
execute() (at the first iteration)	(partial) Initialisation (synchronize init values only)	*fmi2ExitInitializationMode*
execute() (execute iteratively)	Simulation	*fmi2DoStep*

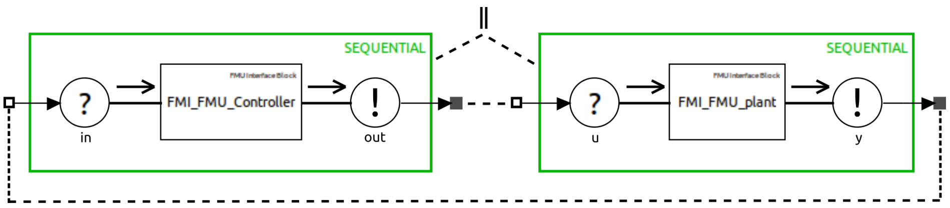
execute()
if stopTime (final time) is reached
	Terminate	*fmi2Terminate*
		fmi2FreeInstance

In our design, the MA orchestrates the data exchange between two FMUs by performing a rendezvous communication through CSPChannels. As is also shown in the functional sub-model layer diagram (the bottom half of Figure 8), the interfacing to an FMU is in SEQUENTIAL with a corresponding CSPWriter/CSPReader, which means the MA can only perform the data exchange from FMU A to FMU B in a *waiting-rendezvous* manner. In that sense, the non-trivial determinism of the MA can be ensured, meaning that the results of different co-simulation runs do not depend on any arbitrary order in which the FMUs might be chosen during the various iterations [21].

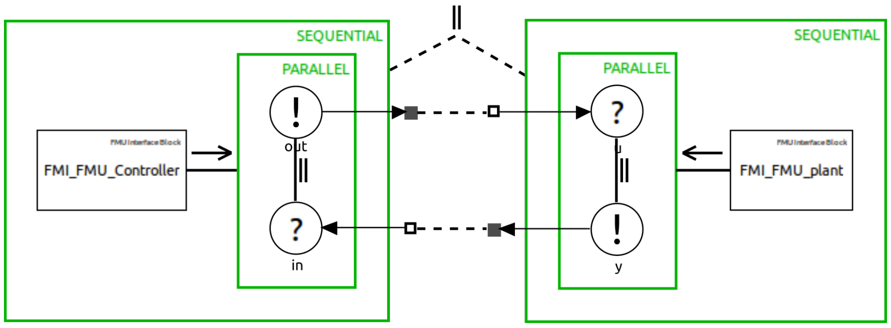
In control systems, a feedback loop is always present as shown in Figure 10a (also see the top part of Figure 4). Using the preferred IO-SEQ pattern [22], i.e. readers, computation and writers in sequence, being the obvious data flow, whereby all readers are in a PARALLEL composition as well as all writers, results in a deadlock, see Figure 10b (note that in this figure, only *one* reader and *one* writer are used per block).



(a) TERRA architecture model of a simple feedback control system



(b) Deadlock: cyclic dependencies in a feedback loop



(c) Deadlock free modification by using PARALLEL composition

Figure 10. Co-simulation models with a feedback loop

Figure 11 is the machine readable CSP generated by TERRA representing the loop control system. Clearly, in Figure 11b, i.e. in the model of Figure 10c, there is *no* deadlock due to having all readers and writers in one PARALLEL composition on the functional layer. The communication events happen in an interleaving manner.

Unfortunately, this brings an artificial delay into the simulation for each FMU, since the output y_{tc_i} of the FMU at the current communication point tc_i is calculated by the input retrieved at previous communication point $tc_i - hc_{i-1}$. Such an artificial delay is in principle wrong, but in case hc_{i-1} is small in combination with a robust control law, the simulation results might be still close to the correct one.

We have implemented an optimized algorithm to compensate for this artificial delay by storing and retrieving the state of the FMUs (*FMUstate*), as shown in Algorithm 1. FMI 2.0 provides two functions *fmi2GetFMUstate* and *fmi2SetFMUstate* to store and restore the FMU internal state. In the *Store state phase* the simulation time is *not* advanced, whereas in the *Restore state phase* the simulation time *is* advanced (Line 19), after restoring and updating internal state with up-to-date and in-time input.

In Algorithm 1, at each communication point, the calculation (simulation) of each FMU is executed twice as values before and after communication point differ. This is the current approach to handle discontinuities. With no doubt, this brings additional computation and communication costs what is investigated in the experiments discussed in the next section.

```
-- Channels
channel channel_Controller_writer_to_plant_reader
channel channel_plant_writer_to_Controller_reader
-- Processes
MainModel_PARALLEL = if (true) then (MainModel_Controller [| {|
    channel_Controller_writer_to_plant_reader ,
    channel_plant_writer_to_Controller_reader |} |]
    MainModel_plant ) ; MainModel_PARALLEL else SKIP
MainModel_Controller = Controller_SEQUENTIAL
Controller_SEQUENTIAL = Controller_reader ; Controller_FMI_FMU_Controller ; Controller_writer
Controller_reader = channel_plant_writer_to_Controller_reader -> SKIP
Controller_FMI_FMU_Controller = SKIP
Controller_writer = channel_Controller_writer_to_plant_reader -> SKIP
MainModel_plant = plant_SEQUENTIAL
plant_SEQUENTIAL = plant_reader ; plant_FMI_FMU_plant ; plant_writer
plant_reader = channel_Controller_writer_to_plant_reader -> SKIP
plant_FMI_FMU_plant = SKIP
plant_writer = channel_plant_writer_to_Controller_reader -> SKIP
```

(a) Machine readable CSP for Figure 10b

```
-- Channels
channel channel_Controller_writer_to_plant_reader
channel channel_plant_writer_to_Controller_reader
-- Processes
MainModel_PARALLEL = if (true) then (MainModel_Controller [| {|
    channel_Controller_writer_to_plant_reader ,
    channel_plant_writer_to_Controller_reader |} |]
    MainModel_plant ) ; MainModel_PARALLEL else SKIP
MainModel_Controller = Controller_SEQUENTIAL
Controller_SEQUENTIAL = Controller_FMI_FMU_Controller ; Controller_PARALLEL
Controller_FMI_FMU_Controller = SKIP
Controller_PARALLEL = Controller_writer ||| Controller_reader
Controller_writer = channel_Controller_writer_to_plant_reader -> SKIP
Controller_reader = channel_plant_writer_to_Controller_reader -> SKIP
MainModel_plant = plant_SEQUENTIAL
plant_SEQUENTIAL = plant_FMI_FMU_plant ; plant_PARALLEL
plant_FMI_FMU_plant = SKIP
plant_PARALLEL = plant_reader ||| plant_writer
plant_reader = channel_Controller_writer_to_plant_reader -> SKIP
plant_writer = channel_plant_writer_to_Controller_reader -> SKIP
```

(b) Machine readable CSP for Figure 10c

Figure 11. Machine readable CSP generated by TERRA

4. Experiments

Experiments have been carried out to verify our approach, by implementing a fluid-level control system. First, the model, and the process of generating the co-simulation is presented. Then, we do two types of experiments: (1) on the functionality and accuracy, i.e. to check whether the fluid-level signals (traces) are correct, and (2) on the performance of our algorithm compared to a MA without the timing compensation.

An architecture model is created in TERRA to represent the control architecture, as shown in Figure 10a. The functional external sub-models, namely the *LevelController* model and the *plant* model are transformed from 20-sim FMUs in later steps.

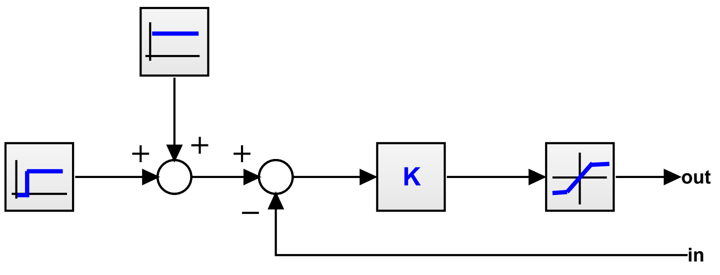
In Figure 12b, the plant model of the fluid-level control system is shown in the form of 20-sim iconic diagrams on the left and as a bond graph on the right. The two exposed signals are *control* and *height*.

The plant contains two tanks coupled through a pipe, which has a fluid input in the first tank (tank 1, the left tank in Figure 12b) and has a output valve in the second tank (tank 2). The leaking volume flow depends on the fluid level (i.e., pressure) in tank 2. The volume flow of the fluid input to tank 1, is controlled by a proportional controller as shown in Figure 12a

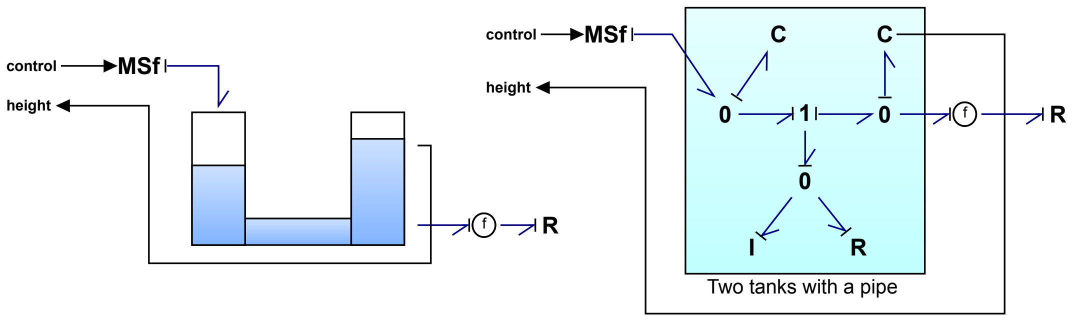
Algorithm 1 Algorithm to compensate for artificial delays at a communication point

```
 1: Current communication point $tc_i$
 2: if $rollback\_flag == true$ then
 3:     // Store state phase
 4:     if $store\_state\_done == false$ then
 5:         $fmi2SetXXX$ to set input received at $tc_{i-1}$
 6:         $fmi2GetFMUstate$ to store internal state at $tc_i$
 7:         $store\_state\_done = true$
 8:         $fmi2DoStep$ to update internal state to $tc_i + hc_i$
 9:         $fmi2GetXXX$ to retrieve output at $tc_i$
10:         Forward output to other FMUs
11:     end if
12:     // Restore state phase
13:     if $store\_state\_done == true$ then
14:         $fmi2SetXXX$ to set input received at $tc_i$
15:         $fmi2SetFMUstate$ to restore internal state to $tc_i$
16:         $store\_state\_done = false$
17:         $fmi2DoStep$ to update internal state to $tc_i + hc_i$, with updated input receive at $tc_i$
18:         $fmi2GetXXX$ to retrieve output at $tc_i$
19:         $Advance\ simulation\ time\ to\ tc_i + hc_i$
20:         Forward output to other FMUs
21:     end if
22: end if
```

with exposed in and out signals, to keep the fluid level in tank 2 at a desired level.



(a) Level controller model



(b) Plant model

Figure 12. A fluid level control system modelled in 20-sim

After carrying out Step 2 and Step 3 in Figure 5, two TERRA CSP models with exactly the same structure as shown in the left and right half of Figure 10c, respectively, are imported into the architecture model of Figure 10a, resulting a fluid level control co-simulation model in TERRA. Consequently, since the system contains a feedback loop, Algorithm 1 is implemented when generating C++ code for the MA.

The simulation experiment is as follows: the initial fluid levels of tank 1 and tank 2 are $1.0m$ and $2.0m$, respectively. The maximum fluid volume flow rate from the controller input to tank 1 is limited to $0.5m^3/s$. From the simulation time $0.000s$ the desired level in tank 2 is $1.5m$, until the simulation time $10.000s$. At the simulation time $10.000s$, a time event is triggered to change the desired level in tank 2 to $2.5m$. The co-simulation finishes at $20.000s$, with a fixed communication step size of $0.001s$.

In Figure 13 the co-simulation results by performing Step 5 in Figure 5 are shown. The input and output from the plant model, i.e. the fluid input to tank 1 and the fluid level in tank 2 are presented in the graphs. The top part is plot for the whole co-simulation period, $20.000s$. As illustrated by the figure the height signal changes as expected, meaning that the desired change of the fluid level in tank 2 is controlled properly.

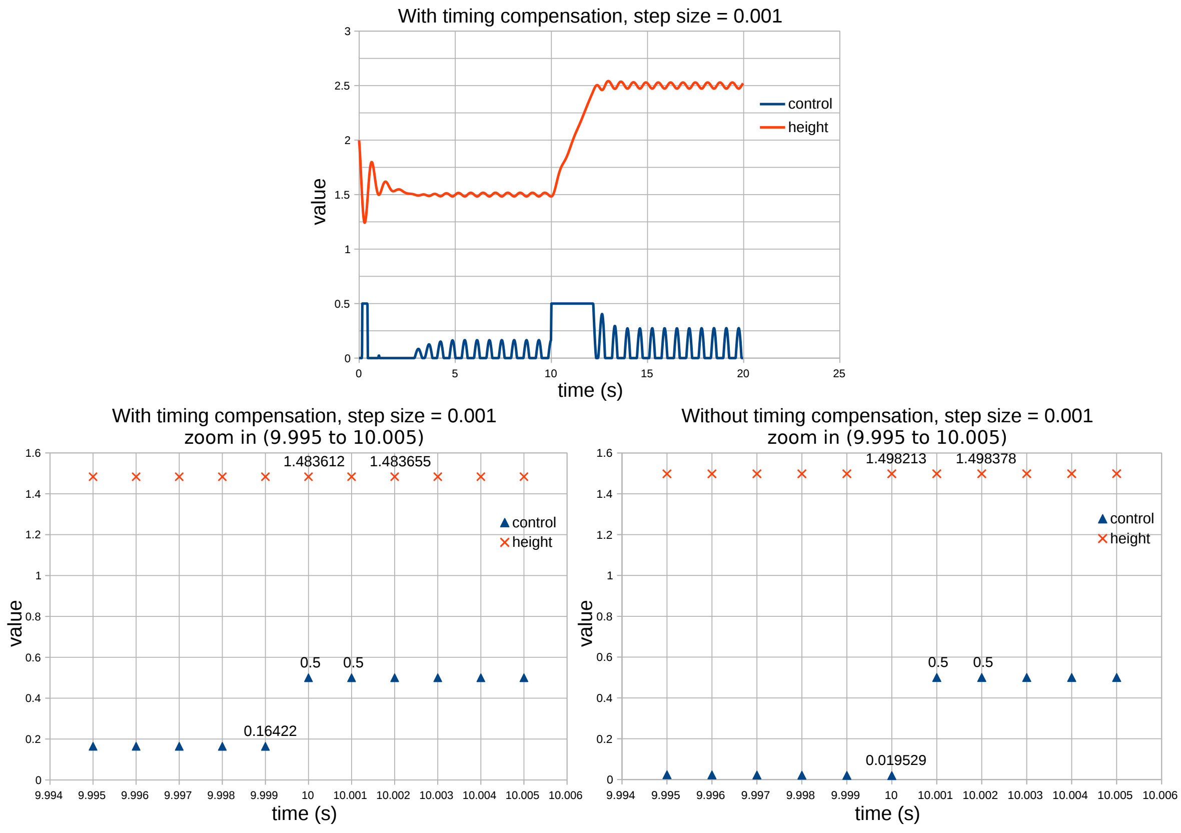


Figure 13. Fluid level control: co-simulation with/without timing compensation, step size = 0.001

In the bottom left, the zoom-in results from time $9.995s$ to $10.005s$ with timing compensation are shown. At time $10.000s$, the control signal jumps to the maximum of $0.5m^3/s$. However in contrast to this, in the bottom right, without any timing compensation, the control signal jumps to the maximum at time $10.001s$, so with one step delay. The comparison is consistent with what we have discussed with respect to Algorithm 1.

In this fluid-level control system, this one-step delay does *not* cause a too much deviation from the correct simulation results. However, this would have been the case when the communication step size would have been large.

To investigate the effect of this time delay, the experiment is performed again, but now the communication step size is changed to $0.3s$. By comparing results in Figure 14, it shows a significant influence when not engaging the timing compensation. The one step delay ($0.3s$) as shown in the right half of Figure 14 makes the control response time almost $2.5s$ longer. Besides, in the zoom-in part on the left, there is also a $0.2s$ delay. The reasoning for this delay is that the time event happened exactly at time $10.000s$, and at the previous communication point $9.999s$ the event did not happen yet. This brings a risk of missing event if the event is not triggered by time (or unpredictable). The store-restore-state mechanism used in Algorithm 1 can still be used to prevent missing events but we still need to make further studies.

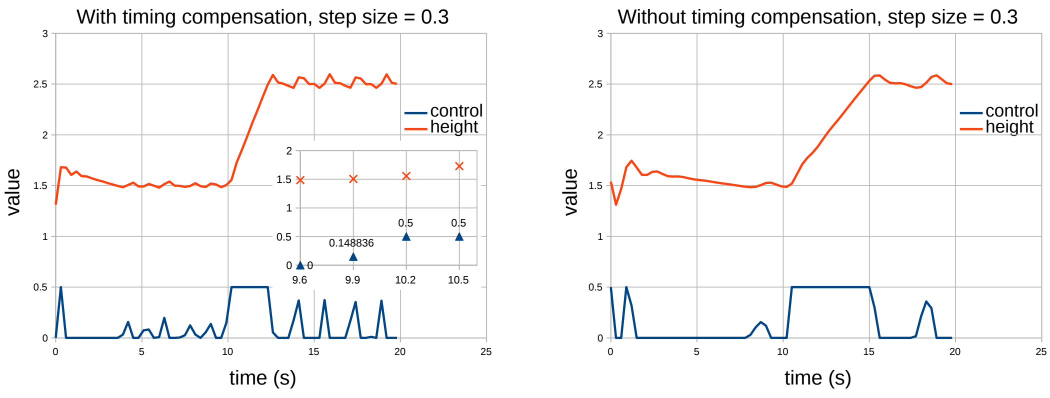


Figure 14. Co-simulation comparison: with/without timing compensation, step size = 0.3

The second experiment is on the performance of our algorithm compared to a MA without timing compensation. Here, we run the co-simulation with different communication step sizes and with or without engaging timing compensation. The results are shown in Figure 15.

At smaller communication step sizes, the timing compensation takes more execution time, as the extra work for this timing compensation has a larger relative contribution in the execution time: this extra work must be done at every communication point between the two FMUs.

For a co-simulation with large step size, using timing compensation to eliminate the significant influence brought by the artificial delay as discussed before should be prioritized.

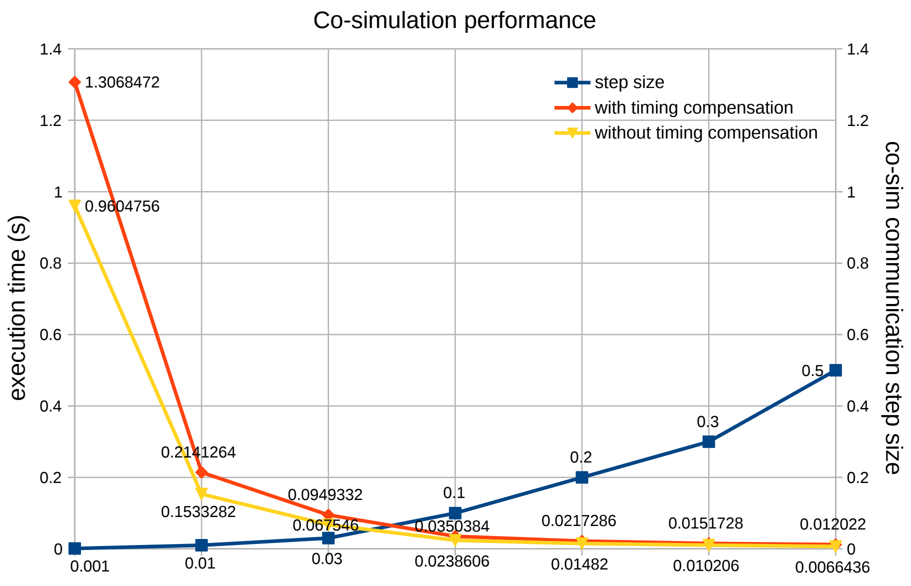


Figure 15. Co-simulation performance: different communication step sizes with/without timing compensation

5. Conclusions

In this paper, we presented a co-simulation approach which is compliant to the FMI 2.0 standard. By coupling different domain models into our TERRA tool, and leveraging on its model-driven features, a CSP-based C++ code skeleton implementing the Master Algorithm can be automatically generated.

The MA orchestrates the co-simulation between different FMUs benefiting from CSP semantics, to break the cyclic I/O dependencies and to perform the synchronization at specific communication points.

To correctly handle timing, we developed an optimized algorithm for the MA to compensate for the artificial delay, which is present when simply connecting the two models to be co-simulated.

A fluid level control model was co-modelled and co-simulated using the proposed approach. Experiments were designed and performed to evaluate on the functionality and accuracy of the co-simulation, and to analyse performance with or without engaging the timing compensation.

Experiment results verified that the fluid level was controlled properly and the one-step artificial delay was handled correctly with engaging the timing compensation. Without engaging the timing compensation, the one-step delay did not cause a too much deviation with a small communication step size, i.e. $0.001s$, but when the communication step size changed to $0.3s$ it had a more significant influence to the control response time. Additionally, co-simulation performance was discussed by measuring execution time with different communication step sizes. We can conclude that for a co-simulation with large step size, using the timing compensation to eliminate the artificial delay should be prioritized.

Future work is to further develop Master Algorithms that can handle arbitrary state events. In that case, at least one of the simulation algorithms must be capable of going backwards in time. Furthermore, after FMI-ME is supported by 20-sim (work in progress by Controllab Products B.V.), coupling of simulation tools, i.e. TERRA and 20-sim, should be implemented whereby more sophisticated co-modelling and more advanced co-simulation calculations can be supported.

References

[1] J. F. Broenink and Y. Ni. Model-Driven Robot-Software Design using Integrated Models and Co-Simulation. In *Proceedings of SAMOS XII*, pages 339 – 344, Samos, Greece, July 2012.

[2] Z. Lu, M. M. Bezemer, and J. F. Broenink. Model-Driven Design of Simulation Support for the TERRA Robot Software Tool Suite. In *37th WoTUG Technical Meeting - Communicating Process Architectures 2015*, pages 257–272, Canterbury, UK, August 2015. Open Channel Publishing Ltd.

[3] A. Al-Hammouri, M. Branicky, and V. Liberatore. Co-simulation Tools for Networked Control Systems. *Hybrid Systems: Computation and Control*, pages 16–29, 2008.

[4] J. Fitzgerald, K. Pierce, and P. G. Larsen. Co-modelling and Co-simulation in the Engineering of Systems of Cyber-Physical Systems. In *2014 9th International Conference on System of Systems Engineering (SOSE)*, pages 67–72, June 2014.

[5] M. Bombino and P. Scandurra. A Model-Driven Co-simulation Environment for Heterogeneous Systems. *International Journal on Software Tools for Technology Transfer*, 15(4):363–374, 2013.

[6] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity - the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[7] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[8] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali. 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 (SAC)*, pages 1758–1764. ACM, 2013.

[9] FMI-standard. http://www.fmi-standard.org/, 2017.

[10] T. Blochwitz, M. Otter, M Arnold, et al. The Functional Mockup Interface for Tool Independent Exchange of Simulation Models. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, number 063, pages 105–114. Linköping University Electronic Press, 2011.

[11] T. Blochwitz, M. Otter, J. Akesson, et al. Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models. In *Proceedings of the 9th International MODELICA Conference;*

September 3-5; 2012; Munich; Germany, number 076, pages 173–184. Linköping University Electronic Press, 2012.

[12] F. Cremona, M. Lohstroh, S. Tripakis, C. Brooks, and E. A. Lee. FIDE: An FMI Integrated Development Environment. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1759–1766. ACM, 2016.

[13] J. F. Broenink, Y. Ni, and M. A. Groothuis. On Model-Driven Design of Robot Software Using Co-Simulation. In *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), workshop on Simulation Technologies in the Robot Development Process*, pages 659–668. TU Darmstadt, November 2010.

[14] Z. Lu, T. C. Ran, and J. F. Broenink. Simulation and Visualisation Tool Design for Robot Software. In *38th WoTUG Technical Meeting - Communicating Process Architectures 2016*, Concurrent System Engineering Series, pages 63 – 82, Copenhagen, Denmark, August 2016. Open Channel Publishing Ltd.

[15] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink. Design and Use of CSP Meta-Model for Embedded Control Software Development. In *Proceedings of the Communicating Process Architectures 2012*, pages 185–199. Open Channel Publishing Ltd., August 2012.

[16] Controllab Products B.V. 20-sim website. http://www.20sim.com/, visited on 2017-06-01.

[17] D. S. Kolovos, L. M. Rose, and R. F. Paige. The Epsilon Book (2010).

[18] Epsilon. Documentation. http://www.eclipse.org/epsilon/doc/, visited on 2017-06-01.

[19] K. J. Kok. TERRA Support for Architecture Modeling. Master's thesis, EEMCS, University of Twente, August 2016. http://essay.utwente.nl/71105/.

[20] Modelica Association. Functional Mock-up Interface for Model Exchange and Co-Simulation V2.0, July 2014. http://www.fmi-standard.org/.

[21] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate Composition of FMUs for Co-simulation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, page 2. IEEE Press, 2013.

[22] P.H. Welch, G. R. R. Justo, and C. J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. pages 1 – 24, 1993.