ITEA 2

INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

Functional Mock-up Interface for Co-Simulation

MODELISAR (07006)

Document version: 1.0

October 12, 2010

History

Version	Date	Remarks
1.0	2010-10-12	First version

License of this document

Abstract

This document defines the “Functional Mock-up Interface for Co-Simulation”. While the interface specification “Functional Mock-up for Model Exchange” (see MODELISAR 2010 for details) gives standardized access to simulation model equations, the basic intention of this document is to provide an interface standard for coupling two or more simulation tools in a co-simulation environment. Co-simulation is a simulation technique for coupled time-continuous and time-discrete systems that exploits the modular structure of coupled problems in all stages of the simulation process (pre-processing, time integration, post-processing).

The data exchange between subsystems is restricted to discrete communication points (sampling points, synchronization points). In the time between two communication points, the subsystems are solved independently from each other by their individual solver. Master algorithms control the data exchange between subsystems and the synchronization of all slave simulation solvers (slaves).

There are two possible ways to provide slave subsystems for co-simulation: subsystems with their specific solver, which can be simulated as stand-alone components (dll-files), or subsystems with their simulation tool, in which they have been developed. Both approaches are covered by this standard.

FMI for Co-Simulation provides interfaces between master and slaves and supports rather simple master algorithms as well as more sophisticated ones. A small set of easy to use C-functions was developed to implement the interface. Note that the master algorithm itself is *not* part of the standard FMI for Co-Simulation, but a very simple example is given and discussed in this document.

All information about the slaves, which is relevant for the communication in the co-simulation environment is provided in a slave specific XML-file. In particular, this includes a set of capability flags to characterize the ability of the slave to support advanced master algorithms, e.g. the usage of variable communication step sizes, higher order signal extrapolation, or others.

Contents

1. Overview

This document specifies a standardized Functional Mock-up Interface (FMI) for the coupling of two or more simulation models in a co-simulation environment (*FMI for Co-Simulation*). Co-simulation is a rather general approach to the simulation of coupled technical systems and coupled physical phenomena in engineering with focus on instationary (time-dependent) problems. FMI for Co-Simulation is designed both for the coupling of simulation tools (*simulator coupling*, *tool coupling*), and coupling with subsystem models, which have been exported by their simulators together with its solvers as runnable code.

Co-simulation exploits the modular structure of coupled problems in all stages of the simulation process beginning with the separate model setup and preprocessing for the individual subsystems in different simulation tools. During time integration, the simulation is again performed independently for all subsystems restricting the data exchange between subsystems to discrete *communication points* tc_i. Finally, also the visualization and post-processing of simulation data is done individually for each subsystem in its own native simulation tool. In different contexts, the communication points tc_i, the *communication steps* $tc_i \rightarrow tc_{i+1}$ and the communication step sizes $hc_i := tc_{i+1} - tc_i$ are also known as sampling points (synchronization points), macro steps and sampling rates, respectively. The term "communication point" in FMI for Co-Simulation refers to the communication between simulation tools and should not be mixed with the output points for saving simulation results to file.

FMI for Co-Simulation is an interface standard for the solution of time dependent coupled systems consisting of subsystems that are continuous in time (model components that are described by instationary differential equations) or time-discrete (model components that are described by difference equations like, e.g., discrete controllers). In a block representation of the coupled system, the subsystems are represented by blocks with (internal) state variables $x(t)$ that are connected to other subsystems (blocks) of the coupled problem by subsystem inputs $u(t)$ and subsystem outputs $y(t)$. In this framework, the physical connections between subsystems are represented by mathematical coupling conditions between the inputs $u(t)$ and the outputs $y(t)$ of all subsystems [R. Kübler, W. Schiehlen: *Two methods of simulator coupling*. - Mathematical and Computer Modeling of Dynamical Systems **6**(2000)93-113].

FMI for Co-Simulation addresses two basic aspects:

1. the data exchange between subsystems and
2. algorithmic issues to synchronize the simulation of *all* subsystems and to proceed in communication steps (macro steps) $tc_i \rightarrow tc_{i+1}$ from initial time $tc_0 := t_{start}$ to end time $tc_N := t_{stop}$.

For the first aspect, data exchange, the individual simulation tools have to be connected via MPI, TCP/IP, sockets or alternative ways of communication. In each individual simulation tool, these connections are initialized before the beginning of the time integration. In the co-simulation environment, the mapping from all subsystem outputs $y(t)$ to the subsystem inputs $u(t)$ has to be initialized to consider all physical coupling between the subsystems.

For the second aspect, a co-simulation specific software component is needed to organize the progress from $tc_0 = t_{start}$ to $tc_N = t_{stop}$ in communication steps $tc_i \rightarrow tc_{i+1}$ and the data exchange between subsystems at the communication points $t_{start} \leq tc_i \leq t_{stop}$ (exchange of subsystem outputs $y(tc_i)$). This software component is called *master* of the co-simulation environment. It may be implemented in one of the individual simulation tools (*master tool*) or in a separate *simulation backplane*. In its most general form, the coupled system may be simulated in *nested* co-simulation environments and FMI for Co-Simulation applies to each level of the hierarchy.

FMI for Co-Simulation defines interface routines for the communication between a master and individual simulation tools (*slaves*) in a co-simulation environment. A simulation tool or the part of it prepared for co-simulation by implementing the FMI is called an FMU (Functional Mock-up Unit)[1].

The most common master algorithm stops at each communication point tc_i the time integration of all slaves, collects the outputs $y(tc_i)$ from all subsystems, evaluates the subsystem inputs $u(tc_i)$, distributes these subsystem inputs to the slaves and continues the (co-)simulation with the next communication step $tc_i \rightarrow tc_{i+1} = tc_i + hc$ with fixed communication step size hc. In each slave, an appropriate solver is used to integrate one of the subsystems for a given communication step $tc_i \rightarrow tc_{i+1}$. The most simple co-simulation algorithms approximate the (unknown) subsystem inputs $u(t), (t > tc_i)$ by frozen data $u(tc_i)$ for $tc_i \leq t < tc_{i+1}$.

FMI for Co-Simulation supports this classical brute force approach as well as more sophisticated master algorithms that adapt, e.g., the communication step size $hc_i = tc_{i+1} - tc_i$ to the solution behavior (*communication step size control*), use higher order signal extrapolation to approximate the subsystem inputs $u(t), (tc_i \leq t < tc_{i+1})$, or handle the subsystems in each communication step sequentially such that intermediate results from the very first subsystems may be used to improve the approximation of subsystem inputs $u(t)$ in later stages of the communication step. FMI for Co-Simulation is designed to support a very general class of master algorithms but it does *not* define the master algorithm itself.

Subsystem inputs and subsystem outputs are described in a slave specific XML-file that contains all information about slave solver, slave model etc. being relevant for the co-simulation environment. The ability of slaves to support more sophisticated master algorithms is characterized by a set of *capability flags* that are added to the slave specific XML-file. Typical examples are the ability to handle variable communication step sizes hc_i and the ability to repeat a rejected communication step $tc_i \rightarrow tc_{i+1}$ with reduced communication step size.

The current document is structured as follows: After this general introduction and overview, Section 2 discusses the general phases of co-simulation workflow together with a more detailed description of all components of a co-simulation environment. The interface itself is defined and discussed in Section 3. Section 4 describes the structure of the archive called Functional Mock-up Unit (FMU), followed by a list

[1] This definition differs slightly from the one used in the FMI for Model Exchange in that, in the case of tool coupling the original tool is additionally required to perform the co-simulation.

of references and the glossary. Additional issues like future extensions of FMI for Co-Simulation, further examples of simulator coupling and some numerical issues are summarized in the Appendix.

2. Co-Simulation

This section gives an overview on co-simulation from a process perspective describing a sequence of phases that are part of a co-simulation task. For the subsequent phases different aspects of FMI for Co-Simulation have to be considered. Section 2 also describes different co-simulation scenarios, which are called “code generation” and “tool coupling” in this document.

2.1. Generic Co-Simulation Activity Flow

2.1.1. Process Assumptions

The following sections are meant to indicate the possible process steps that may be taken by simulation tools being part in a co-simulation setting. The overall process can be divided into a design phase, a deployment phase, and a simulation phase.

Figure 1: Co-Simulation Process Phases

2.1.1.1. Design Phase

The design phase (Figure 2) encompasses all the activities linked to the creation of a simulation model, the packaging of the simulation model into an FMU component, and the composition of a combined system model that makes use of several FMU components.

Figure 2: Design Phase Steps

Some vendors may only provide modeling and transformation capabilities for their simulation tools; the simulation tool only provides an 'FMU export' feature, and is referred to as a slave simulator.

Other vendors may only provide composition capabilities for their simulation tools; such simulators are pure co-simulation platforms, and generally provide an 'FMU import' feature. A simulator of this type is referred to as master simulator.

A simulation tool can also provide both FMU export and FMU import features. As a result, an FMU can be imported that includes itself a number of nested FMUs leading to a hierarchical composition of FMUs.

The following paragraphs describe each individual design step in more detail.

Modeling Step

The modeling step is the sole responsibility of the slave simulator. The user creates a simulation model for a certain subsystem according to the specific requirements of the simulator.

Transformation Step

Once the simulation model is ready, the user needs to decide how the subsystem model will be exported into an FMU implemented either with the FMI for Model Exchange API (see specification document for details) or with the FMI for Co-Simulation API (Figure 3). In this document only the second case is discussed.

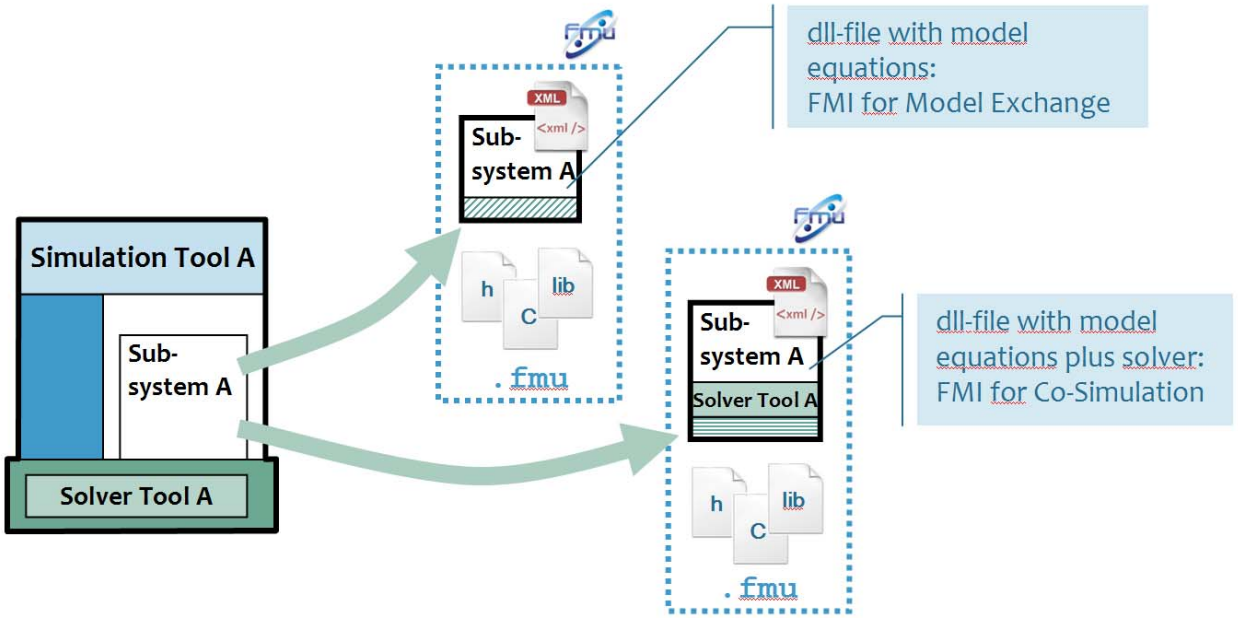


Figure 3: FMU Export Alternatives

The first decision is in terms of parameters; a list of model parameters is selected that will be made public to the master. The result is the generation of the '**Model Description**' XML file which describes the model in terms of a black box.

The second decision pertains to the form in which the model will be exposed to the master. Two alternatives are possible:

- **Code Generation**: The subsystem model is converted into code, i.e., the equations as well as the solver are compiled into a shared library for one or more targets (similar to the FMI for Model Exchange). Both model code and shared library can be included in the FMU archive (see section 4 for details). The master uses the shared library during a simulation run. In the XML-file this is indicated by the `Implementation` flag with the value `CoSimulation_StandAlone`.
- **Tool Coupling**: The subsystem model and dependencies are stored directly within the FMU. The master needs to couple to the original slave simulator that exported the FMU to be able to perform a simulation run. Instead of the compiled model code the FMU archive contains a shared library of a slave tool specific wrapper, which is to be imported by the master tool and interfaces the external tool. The XML `Implementation` flag has the value `CoSimulation_Tool` (for details see also 3.5.2).

The end-result is an FMU that contains a Model description XML file, and possibly the generated model code, compiled shared libraries, or the actual model files. The FMU may be published to some FMU library; two alternatives are possible:

- The slave simulator published the FMU to a proprietary location within the simulator environment, or
- the FMI for PLM API is used to publish the FMU to a central PLM repository.

Composition Step

In general, co-simulation platforms require some form of composition of slave simulation models in order to join subsystem models to a complete simulation system. This composition may be performed in different manners, and typically results in some form of a component-connection graph structure (Figure 4). In this specification, components denote imported FMU instances and the connections represent the communication paths used to exchange data between FMUs. The master is then responsible to schedule communication between components (master algorithm).

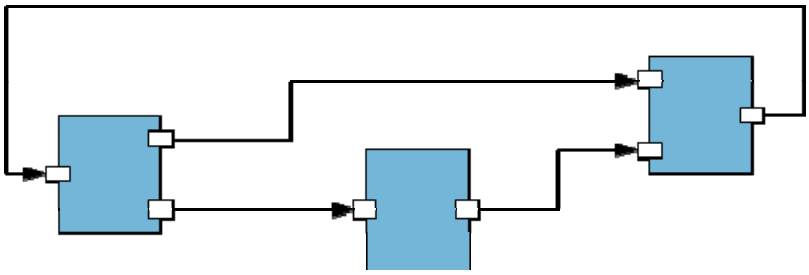


Figure 4: Component-Connection Graph Structure

A component-connection graph variant commonly used is the co-simulation with signal pools (Figure 5). Typically a component publishes a specific output variable that is subscribed by several other components as input. A co-simulation signal pool model can easily be converted to a connection graph model.

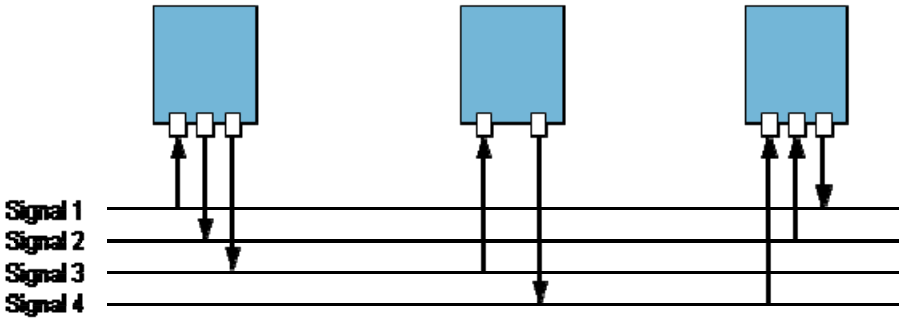


Figure 5: Signal-Pool Variant of a Component-Connection Graph Structure

A master can import an FMU by reading the FMU's zip-archive and the therein contained Model Description XML file. The model description provides the information required by the master to expose the name, the parameters, inputs and outputs of the FMU.

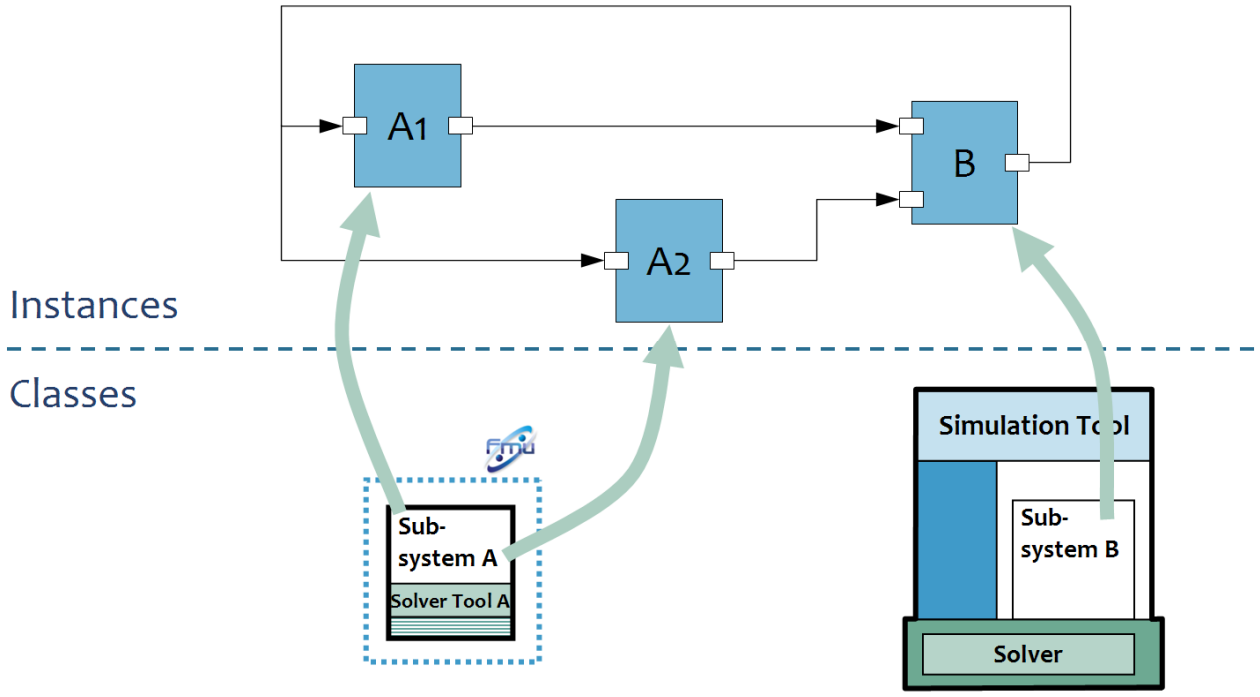


Figure 6: FMU Classifier/Instance Differences

To ensure reusability of an FMU within the same component-connection graph, a clear distinction is made between classes and instances. Each specific FMU is a subsystem class with a unique identifier (the name of an FMU is subsystem dependent). Because an FMU may appear several times within a component-connection graph, an FMU is instantiated with each instance being assigned a unique identifier. The FMU instance denotes then a component within the component-connection graph.

Additionally, each FMU instance stores the initial parameter values and the connection-graph can store the simulation parameters.

2.1.1.2. Deployment Phase

If co-simulation is enacted within a single host, all FMU components need to be accessible to that host. The master has direct file-access to the FMUs; in the simulation phase, the instantiation of FMUs can occur directly within the master process.

In the context of distributed co-simulation, the master typically communicates with slave simulators located on remote machines. The slave simulator is instructed to load the FMU in memory, and exposes the loaded FMU as an instance to the master. To do so, the slave simulation requires access to the FMU.

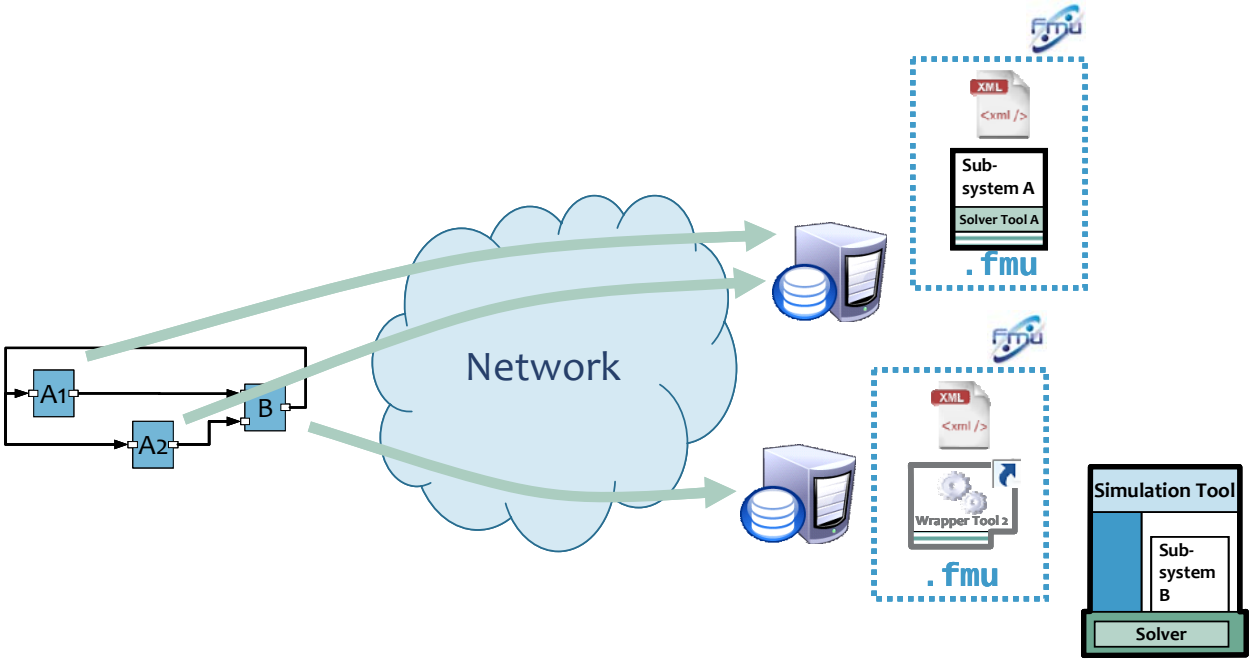


Figure 7: Distribution of FMUs across a Co-Simulation Cluster

Deployment refers to the act of making FMUs available to the slave simulators located remotely to the master; deployment can be performed in different ways.

An offline deployment refers to the manual transport of FMUs to remote locations. Some co-simulation platforms perform deployment within the composition phase. FMUs are copied remotely by the user.

An online deployment is the automatic deployment of FMUs on different hosts by the master. The user only needs to specify on which hosts the various FMU instances need to be transferred to.

Either way, the end result is that the various FMUs used by the master are distributed on the intended hosts.

2.1.1.3. Simulation Phase

The simulation phase (Figure 8) encompasses all the activities related to the execution runtime. The master is responsible for the lifecycle of FMU instances within a simulation run (experiment).

Figure 8: The Simulation Phase

The lifecycle of an FMU is comprised by the following sub-phases.

Instantiation Sub-phase

The master simulator is responsible for the instantiation of all FMU instances contained within the component-connection graph. The FMUs are then loaded into memory and instantiated.

Initialization Sub-phase

Once an FMU instance is ready, the master simulator can set the initial values for each FMU-instance parameter as defined in the component-connection graph. All FMU instances are initialized before simulation can start.

Simulation Sub-phase

The master simulator is responsible for the proper orchestration of the different FMU instances according to a so-called master algorithm (see section 2.2).

Shutdown Sub-phase

The master simulator is responsible for the proper memory deallocation locally and remotely. All FMU instances need to be shutdown; optionally, the FMUs themselves may be deleted from the operating system.

2.1.2. Distributed Infrastructure Assumptions

This section relates to the general assumptions that are made in this document about the kind of co-simulation architecture available on the market. The objective is to ensure that the FMI for co-simulation API is generic enough to be adopted as wide as possible.

Focus is given to the distributed aspect of co-simulation which is of particular interest due the different possibilities available on the market.

2.1.2.1. Generic Architecture

In the simplest compute / IT scenario, co-simulation is performed on one computer with shared memory and a shared file system. The master simulation tool can import the shared library file from the FMU (Figure 9).

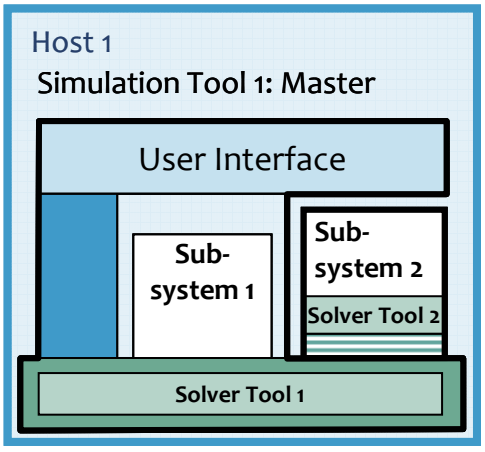


Figure 9: Co-simulation with generated code on a single computer

Figure 10 shows, how a tool coupling scenario can be performed on a single computer. From a user account the FMI co-simulators to be deployed are accessible without additional authentication.

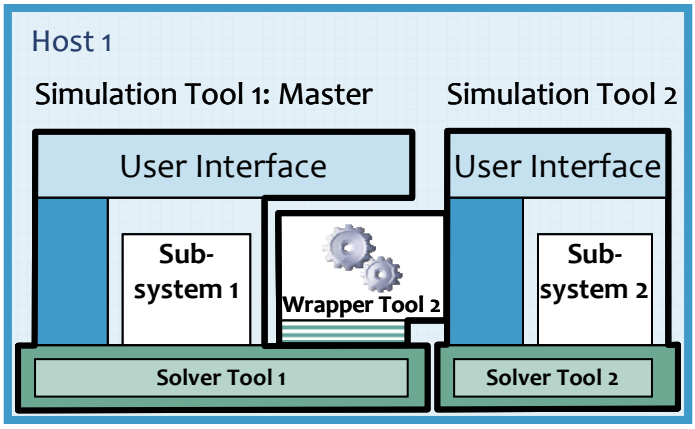


Figure 10: Co-simulation with tool coupling on a single computer

In a distributed co-simulation compute / IT scenario, the FMI co-simulators to be deployed are installed on different computers with maybe different OS (cluster computer, compute farm, computers at different locations) connected by LAN, WLAN, or WAN via TCP/IP. The user has authorized access (e.g., a user account) to the computers with the FMI simulators to be deployed.

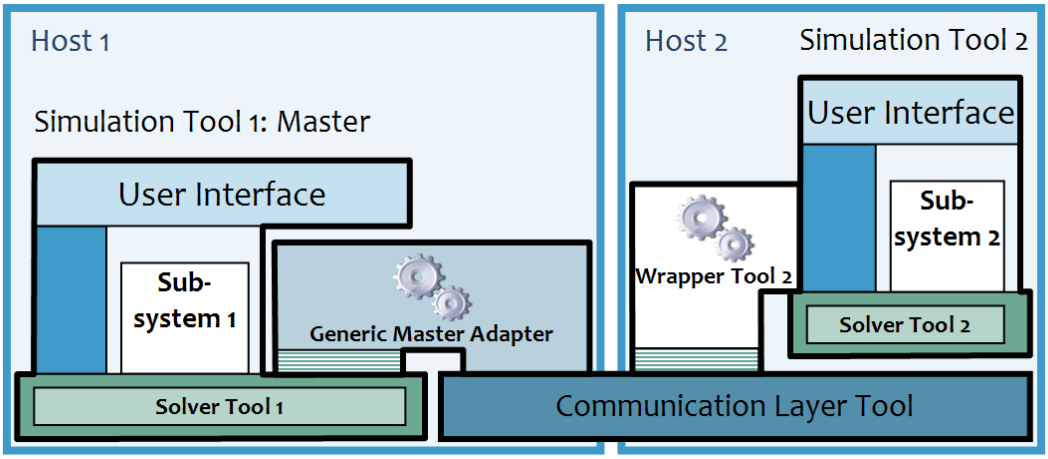


Figure 11: Distributed Co-simulation Infrastructure

In such scenario, in order to couple an FMI co-simulation slave on one computer to an FMI co-simulation master on another computer, a so called FMI co-simulation backbone or framework has to be available (see Figure 11, communication layer tool). This backbone is a special middle-ware. It consists of software on, both, co-simulation master and slave computer and performs the network communication between master and slave. In effect, the FMI co-simulation master does not notice and differentiate the location of the slave simulators.

The FMI co-simulation master (simulator) couples to the involved slave simulators through their FMI in form of a zip-archive. Therefore, for every remote co-simulation slave an FMI zip-archive has to be provided on the master's computer. This zip-archive, as well as the contained shared library file (DLL), has to be compatible to the FMI backplane deployed for the connection with the respective slave simulator. The co-simulation master reads and evaluates the XML description file in the FMI zip-archive. The DLL contained in this zip-archive provides functions according to the FMI which are able to

communicate with the remote slave simulator via the FMI co-simulation backplane. The authentication on the remote computer(s) is also performed by this backbone.

On the FMI co-simulation slave computer this backbone comprises an application server with an FMI (master side) which can couple to an FMI slave. The server accesses the zip-archive of the FMI slave. The application server loads/links the DLL to perform the communication between co-simulation master and this slave.

2.1.2.2. Assumptions

FMU Availability Assumption

The general assumption is that an FMU is already available on the host where it will be started.
This assumption is fulfilled by an online/offline deployment.

Communication Assumption

No assumption is made as to which communication protocol or transport shall be used to access the FMU instance across a network. The FMI-for-co-simulation shall not include details about host, tcp/udp ports, etc.

FMI-for-co-simulation can only include local parameter specifications. The co-simulation framework provides the remoting capabilities and is responsible to communicate with remote FMUs.

Simulator Assumption

The master simulator shall be given as little knowledge as possible about the slave simulator in a tool coupling scenario. The objective is two fold:

- wrap all specific parameters required by a slave simulator in an implementation exposing the FMI-for-co-simulation; this wrapper must be provided by the slave simulator tool vendor.
- wrap all specific parameters required by a co-simulation framework in an implementation exposing the FMI-for-co-simulation; this wrapper will be loaded by the master simulator, and must be provided by the co-simulation tool vendor.

2.1.2.3. Instantiation Sequence

The purpose of this section is to describe in more details the instantiation sequence required to remotely load an FMU instance after calling the `fmiInstantiateSlave`.

In the following scenario, the co-simulation framework has already been provided with the component-connection graph and the deployed location of FMU instances. The end result is to instantiate each FMU instance locally or remotely.

1. The master simulator loads the local FMU proxy, that is, the FMI wrapper (master adapter) provided by the co-simulation framework.
2. The co-simulation framework sends an instruction to the remote application server to load a specific FMU instance.
3. The remote application server selects the correct instantiation method. Two alternatives are possible:
   - The FMU is composed of a shared library that includes model and solver in a compiled form. The FMU shared library is directly loaded with the correct FMU instance identifier.

- The FMU represents a tool coupling. The MIME-type of the slave simulator is used to select the correct FMI wrapper provided by the slave simulator tool vendor.

4. The master simulator and slave simulator can now communicate over the FMI-for-co-simulation API.

2.2. Numerical Co-Simulation Computation Flow

Co-simulation is a simulation with more than one simulation tools which exchange intermediate results (variables, values, information) during simulation.

A **simulation tool** (**simulator)** is a tool (algorithm, executable) that computes a model's behavior, which is called simulation. In the computational sense a simulation is an autonomously running process. FMI for Co-Simulation is restricted to:

- All calculated values $v(t)$ are **time dependent** functions within an a priori defined time interval $t_{start} \le t \le t_{stop}$.
- All calculations (simulations) are carried out **time increasing** in general. The actual time t is running step by step from t_{start} to t_{stop}. A tool may have the property to be able to repeat the simulation of parts of $[t_{start}, t_{stop}]$or the whole time interval $[t_{start}, t_{stop}]$.
- After simulation the interval $[t_{start}, t_{stop}]$ is covered by subintervals $[t_i, t_{i+1}]$ with $0 < i \le N$, $t_i < t_{i+1}$, $t_0 = t_{start}$, $t_N = t_{stop}$. The subinterval length h_i is called **step size** of the i^{th} step, $h_i = t_{i+1} - t_i$. This step size is simulation tool internal.

A simulation tool can be coupled, if it has the following properties:

- The simulation tool can be given a time value tc_i, $t_{start} \le tc_i \le t_{stop}$.
- The simulation can be interrupted when tc_i is reached.
- During the interrupted simulation the simulation tool can both receive values $u(tc_i)$ and send values $y(tc_i)$.
- During the interrupted simulation the simulation tool can be given a new time value tc_{i+1}, $tc_i \le tc_{i+1} \le t_{stop}$ to simulate the time subinterval $tc_i < t \le tc_{i+1}$
- The subinterval length hc_i is called **step size** of the i^{th} communication step, $hc_i = tc_{i+1} - tc_i$. In general, the communication step size can be positive, zero, but not negative.

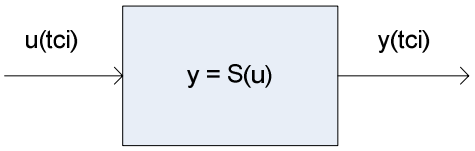


Figure 12: Data flow of a simulation tool at communication points

The item simulation tool in the sense of this description can be a huge variety of tools: a powerful simulator like AmeSim, Dymola, Simpack, SimulationX, … but also a C program, which reads data from a file without having its own solver. Within a system to be simulated many different tools should be able to interact.

2.2.1. Master-Slave Structure

Co-simulation is used to solve a coupled system by simulating each part with its own coupleable simulation tool. Once the system is established there exists a directed signal flow between the involved simulation tools. Therefore it is assumed that the signal flow between the coupled simulation tools is directed. The coupled simulation tools form a directed graph *G* the nodes of which are the simulation tools, and the directed lines describe the data flow.

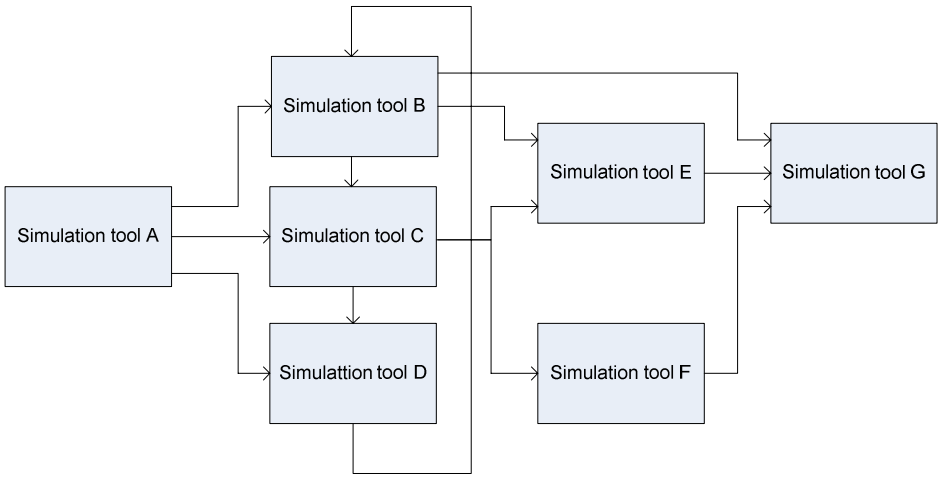


Figure 13: Example graph *G* of coupled simulation tools

Instead of directly coupling, a master is assumed to be located between the single simulation tools which are now called slaves. Each arrow of the graph *G* is regarded as to go “through” the master.

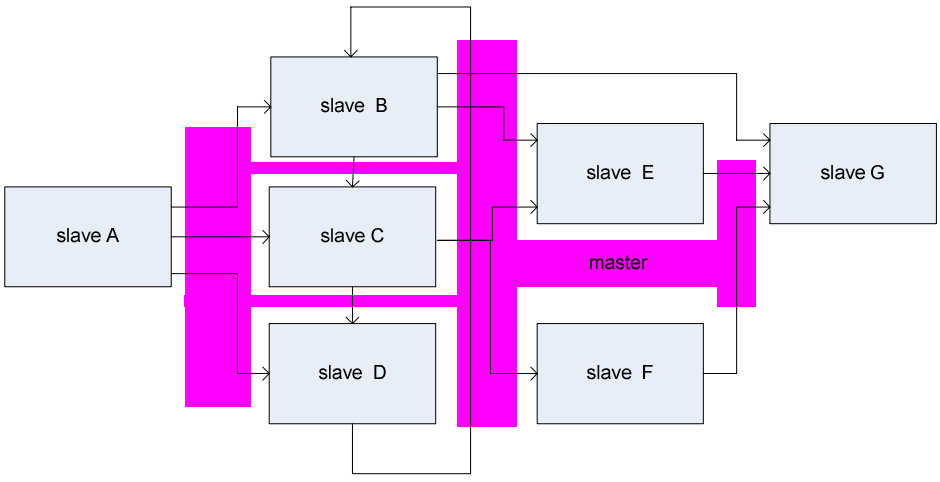


Figure 14: Master-Slave structure

Slaves are assumed to communicate with the master only. In this description the interface between master and slave is defined.

The master itself can be involved in a higher order simulation environment serving as slave. On each level of such a nested master hierarchy the FMI for Co-Simulation can be applied.

2.2.2. Basic Co-Simulation Computation Flow

The slaves will have properties which influence the possible master algorithms, especially restrict them. The master has to select suitable algorithms. In this description the master algorithms will be neither defined nor standardized. Only the interface between master and slaves is to be defined. Nevertheless, a basic co-simulation flow is assumed:

2.2.2.1. Initialization Sub-phase

All simulation tools are prepared for starting the co-simulation. The communication links between master and slaves are established. The master receives the properties of the slaves. Additionally the master receives the connection graph *G* e.g. by user input. The master chooses the master algorithm based on the capabilities of the involved slaves as well as the connection graph *G*, and possibly user inputs.

2.2.2.2. Simulation Sub-phase

The master forces the slaves to simulate the time interval $[t_{start}, t_{stop}]$ by stepwise solving master subintervals (**communication steps**) $(tc_i, tc_{i+1}]$ with $0 < i \le N$, $tc_i \le tc_{i+1}$, $tc_0 = t_{start}$, $tc_N = t_{stop}$. The subinterval length hc_i is called **communication step size** of the i^{th} step, $hc_i = tc_{i+1} - tc_i$.

The boundary points tc_i, tc_{i+1} of each subinterval are called **communication points**. It is allowed that the communication step size hc_i can be zero ($tc_{i+1} = tc_i$, iteration). In particular for the first simulation step and at an event (event iteration) a communication step size of zero length is appropriate, $hc_i = 0$.

It depends on the master algorithm how the communication step size, and the communication points are chosen. The master algorithm itself uses both the slave properties, and the graph *G*. The communication points can be chosen by the master individually for each slave, and the master can start and stop each slave independently from other slaves.

Before a subinterval is simulated, the slave receives its input values $u(tc_i)$ and possibly derivatives with respect to time ($\dot{u}(tc_i)$, $\ddot{u}(tc_i)$, ...) as well as the communication step size hc_i. After starting the slave simulation of the communication step $[tc_i, tc_{i+1}]$ the master receives the slave output values $y(tc_{i+1})$ and possibly derivatives with respect to time ($\dot{y}(tc_{i+1})$, $\ddot{y}(tc_{i+1})$, ...). Furthermore, the slave status has to be transferred to the master. Especially if the slave simulation fails, further communication is necessary.

2.2.2.3. Shutdown Sub-phase

By giving a closing information the master forces the slaves to stop.

2.2.2.4. Summary of Transferred Information via FMI for Co-Simulation

The interface between master and slave must be able to transfer the following information:

To be transferred	Direction	When
Properties of the slave	To master	Initialization sub-phase
Status of the slave	To master	After communication step
Slave input values $u(tc_i)$ and derivatives (optional)	To slave	Before communication step
Slave output values $y(tc_{i+1})$ and derivatives (optional)	To master	After communication step and after initialization
Control commands, at least - simulate communication step $[tc_i, tc_{i+1}]$ - finish simulation	To slave	At communication step Shutdown sub-phase

The connection graph *G* which specifies the directed connection between inputs and outputs of the slaves is also needed by the master. The input of this graph *G* is not standardized in this document. The graph input can be achieved e.g. by a user input.

All information regarding the (static) properties of slaves will be collected within XML-files. By reading the XML files the master gets the properties of the slaves.

2.2.3. Master

The tasks of the master are:

Tasks to be done in the **initialization sub-phase**:

- Ask the properties of the slaves.
- Analyze the graph *G*.
- Chose a master algorithm.

In the **simulation sub-phase** the master provides subintervals for each slave.

Before the slave simulation of a communication step $[tc_i, tc_{i+1}]$ the master tasks are:

- Calculate the communication step size hc_i, as well as the communication step $[tc_i, tc_{i+1}]$.
- Calculate the slave input values $u(tc_i)$ and possibly their derivatives $\dot{u}(tc_i), \ddot{u}(tc_i)$, ...
- Transfer tc_i, tc_{i+1}, $u(tc_i)$ and possibly $\dot{u}(tc_i)$, $\ddot{u}(tc_i)$, ... to the slave.
- Start the slave to simulate the communication step $[tc_i, tc_{i+1}]$.
- Wait for slave finishing.

After the slave simulation of the communication step $[tc_i, tc_{i+1}]$ the tasks are:

- Ask the status of the slave, interpret it.
- Transfer $y(tc_{i+1})$ and possibly $\dot{y}(tc_{i+1})$, $\ddot{y}(tc_{i+1})$, ... to the master, if the communication step is calculated regularly, or after initialization. [adapt State Machine]
- Transfer additional information to the master, if the communication step is not calculated regularly, e.g. error messages, or an intermediate stop time

In the **shutdown** sub-phase after the complete simulation, or in special cases

- Stop the complete simulation.

2.2.4. Slave

The tasks of the slave are:

Tasks to be done in the **initialization sub-phase**:

- Send the properties of the slave to the master.

Before the simulation of a communication step $[tc_i, tc_{i+1}]$ the tasks are:

- Stop, if stop command is received (**shutdown sub-phase**).
- Receive tc_i, tc_{i+1}, $u(tc_i)$ and possibly derivatives $\dot{u}(tc_i)$, $\ddot{u}(tc_i)$, ... from the master.
- Simulate the communication step $[tc_i, tc_{i+1}]$ after receiving the simulate-command.
- Transfer $y(tc_{i+1})$ and possibly derivatives $\dot{y}(tc_{i+1})$, $\ddot{y}(tc_{i+1})$, ... to the master, if the communication step is calculated regularly.
- Transfer additional information to the master, if the subinterval is not calculated regularly, e.g. error messages, or intermediate stop time.

After the simulation of a subinterval $[tc_i, tc_{i+1}]$ the tasks are:

- Wait for the next command.

This roughly described communication is detailed in section 3.

2.2.5. Example of Master Algorithm

One of the simplest master algorithms is like this:

- The communication step size is constant: $hc_i = hc \forall i$.
- For all slaves the first input value is chosen by the master, e.g. $u(t_{start}) = 0$.
- The input values $u(tc_i)$ are transferred to all slaves as well as the communication step size hc. The slave simulation is started, and the resulting output values $y(tc_{i+1})$ are transferred to the master. This is done for increasing i until t_{stop} is reached.
- At each communication point tc_i the master distributes the received slave results $y(tc_i)$ to the slave inputs $u(tc_i)$ according to the connection graph for the next communication step $[tc_i, tc_{i+1}]$.

The simplest way to use the input values by the slaves is to keep u constant during the slave simulation: $u(t) = u(tc_i)$ for all $tc_i \le t \le tc_{i+1}$.

For this simple master algorithm case a pseudo code example is given in the next section.

More advanced master algorithms analyze the connection graph to elaborate an effective calling order for the slaves. The communication step size can be adapted, and if possible communication steps can be repeated to allow iterative master algorithms.

3. The Application Programming Interface

The interface consists of two parts:

- Co-Simulation Interface
  A set of C-functions for exchange of in/output values and status information.
- Co-Simulation Description Schema
  The schema defines the structure and content of an XML-file. This file contains the "static" information concerning the model (dimensions, input/output variables…) and the simulator (capabilities, …) which is used to compute the model.

3.1. The Co-Simulation Interface

This chapter contains the interface description to access the in/output data and status information of a co-simulation slave from a C program.

3.1.1. Platform Dependent Definitions (fmiPlatformTypes.h)

In order to simplify porting, no C types are used in the function interfaces, but the alias types defined in this section. All definitions in this section are provided in the header file "`fmiPlatformTypes.h`"[1].

`typedef void`* `fmiComponent;`

This is a pointer to a co-simulation slave specific data structure. It contains all information needed by the slave to process the co-simulation.

`typedef unsigned int` `fmiValueReference;`

This is a handle to a (base type) variable value of the model. The handle is unique at least with respect to the corresponding base type (like `fmiReal`). All structured entities, like records or arrays, are "flattened" in to a set of scalar values of type `fmiReal`, `fmiInteger` etc. An `fmiValueReference` references one such scalar. The coding of `fmiValueReference` is a "secret" of the modeling environment that generated the model. The interface only provides access to variables via this handle. Extracting concrete information about a variable is specific to the used environment that reads the Model Variable File in which the value handles are defined.

If a function in the following sections is called with a wrong `fmiValueReference` value (e.g. setting an output with an `fmiSetReal(...)` function call), then the function has to return with an error (`fmiStatus = fmiError`), i.e., the processing of the co-simulation must be terminated.

```
typedef double       fmiReal   ;  // Real number (64 bits)
typedef int          fmiInteger;  // Integer number (32 bits)
typedef char         fmiBoolean;  // Boolean number (8 bit,
                                  // two values: fmiFalse, fmiTrue)
typedef const char*  fmiString ;  // Character string ('\0' terminated)
                                  // UTF8 encoded
#define fmiTrue  1
```

[1] This file is identical to fmiModelTypes.h from Model Exchange 1.0. In the follow up version Model-Exchange will also use this file.

```
#define fmiFalse 0
```

These are the basic data types used in the interfaces of the C-functions. More data types might be included in future versions of the interface.
If an `fmiString` variable is passed as input argument to a function and the string shall be used after the function has returned, the whole string must be copied (not only the pointer) and stored in the internal memory, because there is no guarantee for the lifetime of the string after the function has returned.

3.1.2. Status Returned by Functions

This section defines the "status" flag (an enumeration of type `fmiStatus` defined in file "`fmiModelFunctions.h`") that is returned by all functions to indicate the success of the function call.

```
typedef enum {fmiOK,
              fmiWarning,
              fmiDiscard,
              fmiError,
              fmiFatal,
              fmiPending
             } fmiStatus;
```

Status returned by functions. The status has the following meaning

- `fmiOK` – all well
- `fmiWarning` – there are things not quite right, but the computation can continue. Function "`logger`" was called in the model (see below) and it is expected that this function has shown the prepared information message to the user.
- `fmiDiscard` – can be returned by `fmiDoStep(...)` or `fmiGetSlaveStatus(..., fmiDoStepState,...)`. See section 3.2.2. Is returned also if the slave is not able to return the required status information. The master has to decide if the simulation run can be continued anyway.
- `fmiError` – the slave encountered an error. If one of the functions (except `fmiDoStep(...))` returns `fmiError`, the simulation cannot be continued and function `fmiFreeInstance(...)` must be called. Function "`logger`" was called (see below) and it is expected that this function has shown the prepared information message to the user.
- `fmiFatal` – the slave is irreparably corrupted. Function `logger` was called (see below) and it is expected that this function has shown the prepared information message to the user. It is not possible to call any other function of the slave.
- `fmiPending` – is returned if the slave executes the function in an asynchronous way. That means the slave starts to compute but returns immediately. The master has to call `fmiGetStatus(..., fmiDoStepStatus)` to find out, if the slave is ready. Can be returned only by the function `fmiDoStep(...)` and by `fmiGetStatus` (see section 3.2.

3.1.3. Inquire Platform and Version Number of Header Files

This section documents functions to inquire information about the header files used to compile its functions.

```
const char* fmiGetTypesPlatform();
```

Returns the name of the set of (compatible) platforms of the "`fmiTypes.h`" header file which was used to compile the functions of the Model Exchange interface. The function returns a pointer to the static variable "`fmiTypesPlatform`" defined in this header file. The standard header file as documented in this specification has version "standard32" (so this function usually returns "standard32").

```
const char* fmiGetVersion();
```

Returns the version of the implemented co-simulation interface functions. If a slave supports the interface as it is described in this document it has to return "1.0".

3.2. Creation and Destruction of Co-Simulation Slaves

This section documents functions that deal with instantiation and destruction of co-simulation slaves.

```
fmiComponent fmiInstantiateSlave(fmiString instanceName, fmiString fmuGUID,
                                 fmiString fmuLocation, fmiString mimeType,
                                 fmiReal timeout, fmiBoolean visible,
                                 fmiBoolean interactive,
                                 fmiCallbackFunctions functions,
                                 fmiBoolean loggingOn)
```

Returns a new instance of a co-simulation slave. If a null pointer is returned, then instantiation failed. In that case, function "`functions->logger`" was called and detailed information is transferred given there. A slave can be instantiated many times. This function must be called successfully, before any of the following functions can be called. The slave has to perform all actions which are necessary before a simulation run starts (e.g. loading the model file, compilation...).

Argument `instanceName` is a unique identifier for a given FMI Component instance. This instance identifier is used to identify a component within a co-simulation graph model, and can be used for logging messages. This argument cannot be null.

Argument `fmuGUID` is used to check that the co-simulation description file is compatible with the model file used by the slave. It is a vendor specific globally unique identifier of the co-simulation description file. It is stored in the description file as attribute guide of fmiModelDescription (See section 3.5). The `fmuGUID` read from the co-simulation description file and passed to `fmiInstantiateSlave` must be identical to the one stored in the used model (e.g., it is a "fingerprint" of the relevant information stored in the description file), otherwise the model and the description file are not consistent to each other. This argument cannot be null.

Argument `fmuLocation` is an URI according to the ietf RFC3986 syntax to indicate the access path to the FMU archive. The following protocols must be understood: (Mandatory) file:// (Optional) http(s):// ftp:// (Reserved) 'fmi://' for fmi for PLM.

Argument `mimeType` represents the MIME type (ietf RFC 2045, 2046, 2047, 2048, 2049) of the 'simulator', e.g., '`application/x-<simulator name>`', '`application/x-fmu-openmodelica`'. If the FMU contains a shared library, i.e., Model exchange + solver, the following mime-type should be used: '`application/x-fmu-sharedlibrary`'. This mimetype is typically used to help identify which simulator or FMI wrapper DLL is to be started for the specified FMU in the tool coupling scenario.
Special mimetype could be 'application/x-fmu-modelica' to be used by any modelica simulators. This argument cannot be null.

Argument `timeout` is a communication timeout value in milli-seconds to allow inter-process communication to take place. A timeout value of 0 indicates an infinite wait period.

Argument `visible` indicates whether or not the simulator application window needed to execute a model should be visible, i.e., `fmiFalse` value indicates that the simulator is executed in **batch mode**, and `fmiTrue` value indicates that the simulator is executed in **interactive mode**. Use case: in interactive mode, it should be possible to explicitly acknowledge start of simulation / instantiation / initialization; acknowledgement is non-blocking.

Argument `interactive` indicates whether the simulator application must be manually started by the user, i.e., `fmiFalse` value indicates that the co-simulation tool automatically starts the simulator application and executes the model referenced in the model description, and `fmiTrue` value indicates that the simulator indicates that the simulator application must be manually started by the user.

Argument `functions` provides callback functions to be used from the model functions to utilize resources from the environment (see type `fmiCallbackFunctions` below).

If `loggingOn=fmiTrue`, debug logging is enabled. If `loggingOn=fmiFalse`, debug logging is disabled.

```
typedef struct {
                void (*logger)(fmiComponent c, fmiString instanceName,
                               fmiStatus status, fmiString category,
                               fmiString message, ...);
                void (*stepFinished) (fmiComponent c, fmiStatus status);
                void* (*allocateMemory)(size_t nobj, size_t size);
                void  (*freeMemory)    (void* obj);
                } fmiCallbackFunctions;
```

The struct contains pointers to functions provided by the environment to be used by the slave. In the default `fmiFunctions.h` file, typdefs for the function definitions are present to simplify the usage. This is non-normative. The functions have the following meaning:

Function **logger**:
Pointer to a function that is called in the model, usually if the model function does not behave as desired. If "`logger`" is called with "`status = fmiOK`", then the message is a pure information message. "`instanceName`" is the instance name of the model that calls this function. "`category`" is the category of the message. Usually, "category" is only used for debug messages in order that the environment can filter the debug messages to be shown. The meaning of "category" is defined by the modeling environment that generated the model code. Argument "`message`" is provided in the same way and with the same format control as in "`printf(...)`". In the simplest case, this function might only print the message. It might also just store the message in a stack of buffers and via options in the environment the printing of the messages is controlled.
The logger function will append a line break to each message when writing messages after each other to a terminal or file (the messages may also be shown in other ways, e.g. as separate text-boxes in a GUI). The caller may include line-breaks (using "\n") within the message, but should avoid trailing line breaks.

Variables are referenced in a message with "#<Type><ValueReference>#" where <Type> is "r" for `fmiReal`, "i" for `fmiInteger`, "b" for `fmiBoolean` and "s" for `fmiString`. If character "#"shall be included in the message, it has to be prefixed with "#", so "#" is an escape character. Example:

A message of the form

"`#r1365# must be larger than zero (used in IO channel ##4)`"
might be changed by the environment to
"`body.m must be larger than zero (used in IO channel #4)`"
if "`body.m`" is the name of the `fmiReal` variable with `fmiValueReference = 1365`.

Function **stepFinished**:
Optional call back function to signal if the computation of a communication step is finished. A NULL pointer can be provided. In this case `fmiDoStep` has to be carried out synchronously. If a pointer to a function is provided, it must be called after a completed communication step.

Function **allocateMemory**:
Pointer to a function that is called in the model if memory needs to be allocated. It is not allowed that the model uses `malloc`, `calloc` or other memory allocation functions. One reason is that these functions might not be available for embedded systems on the target machine. Another reason is that the environment may have optimized or specialized memory allocation functions. `allocateMemory` returns a pointer to space for a vector of `nobj` objects, each of size "`size`" or `NULL`, if the request cannot be satisfied. The space is initialized to zero bytes (a simple implementation is to use `calloc` from the C standard library).

Function **freeMemory**:
Pointer to a function that must be called in the model if memory is freed that has been allocated with `allocateMemory`. If a NULL pointer is provided as input argument `obj`, the function shall perform no action (a simple implementation is to use `free` from the C standard library; in ANSI C89 and C99, the null pointer handling is identical as defined here).
The functions `allocateMemory` and `freeMemory` can be ignored by slaves. This is signalled by setting the capability flag `canNotUseMemoryManagementFunctions`.

```
fmiStatus fmiInitializeSlave(fmiComponent c, fmiReal tStart,
                             fmiBoolean StopTimeDefined, fmiReal tStop);
```

Informs the slave that the simulation run starts now.
The arguments `tStart` and `tStop` can be used to check whether the model is valid within the given boundaries or to allocate memory which is necessary for storing results. If the master tries to compute past `tStop` the slave returns `fmiError`.

```
fmiStatus fmiTerminateSlave(fmiComponent c);
```

Is called by the master to signal the slave the end of the co-simulation run.

```
fmiStatus fmiResetSlave(fmiComponent c);
```

Is called by the master to reset the slave after a simulation run. Before starting a new run, `fmiInitializeSlave` is to be called.

```
void fmiFreeSlaveInstance(fmiComponent c);
```

Disposes the given instance, unloads the loaded model, and frees all the allocated memory and other resources that have been allocated by the functions of the co-simulation interface.

```
fmiStatus fmiSetDebugLogging(fmiComponent c, fmiBoolean loggingOn);
```

If `loggingOn=fmiTrue`, debug logging is enabled, otherwise it is switched off.

3.2.1. Transfer of input / output values and parameters

Input and output variables are identified with a variable handle called “value reference”. The handle is defined in the co-simulation description file (as “ValueReference” in element “ScalarVariable”). It is a unique reference within each Slave instance for a scalar variable with respect to its base type (like `fmiReal`) and is internal information of the slave.

```
fmiStatus fmiSetReal   (fmiComponent c, const fmiValueReference vr[],
                        size_t nvr, const fmiReal value[]);
fmiStatus fmiSetInteger(fmiComponent c, const fmiValueReference vr[],
                        size_t nvr, const fmiInteger value[]);
fmiStatus fmiSetBoolean(fmiComponent c, const fmiValueReference vr[],
                        size_t nvr, const fmiBoolean value[]);
fmiStatus fmiSetString (fmiComponent c, const fmiValueReference vr[],
                        size_t nvr, const fmiString value[]);
```

Set values of inputs. Argument `vr` is a vector of `nvr` value references that define the variables that shall be set. Argument `value` is a vector with the actual values of these variables. The slave has to copy the content of the `value` array if it needs them after returning. The master may deallocate the array.

Restrictions on using the `fmiSetXXX` functions (see also section 3.3):

1. These functions can only be called after calling `fmiInstantiateSlave`(...) and before `fmiFreeSlave`(...).
2. Besides (1), they can always be called on inputs (ScalarVariable.Causality = “input”).
3. For parameters (ScalarVariable.causality = “input” and ScalarVariable.variability = “parameter”) the functions can only be called between `fmiInstantiateSlave(...)` and `fmiInitializeSlave(...)`.

If no set function is called for a variable it is initialized by the slave to its default value.

In order to enable the slave to interpolate the continuous real inputs between communication steps the derivatives of the inputs with respect to time can be provided. To allow higher order interpolation also higher derivatives can be set. Whether a slave is able to interpolate and therefore needs this information is provided by the capability `canInterpolateInputs`.

```
fmiStatus fmiSetRealInputDerivatives(fmiComponent c,
                                     const fmiValueReference vr[],
                                     size_t nvr, const fmiInteger order[],
                                     const fmiReal value[]);
```

Sets the n-th time derivative of real input variables. Argument “`vr`” is a vector of value references that define the variables whose derivatives shall be set. The array “`order`” contains the orders of the respective derivative (1 means the first derivative, 0 is not allowed). Argument “`value`” is a vector with the values of the derivatives. “`nvr`” is the dimension of the vectors.
Restrictions on using the function are the same as for the `fmiSetReal` function.

Inputs and their derivatives are set with respect to the beginning of a time step.

Output variables are handled in the same way using the following functions:

```
fmiStatus    fmiGetReal(fmiComponent c, const fmiValueReference vr[],
                        size_t nvr, fmiReal value[]);
fmiStatus fmiGetInteger(fmiComponent c, const fmiValueReference vr[],
                        size_t nvr, fmiInteger value[]);
fmiStatus fmiGetBoolean(fmiComponent c, const fmiValueReference vr[],
                        size_t nvr, fmiBoolean value[]);
fmiStatus  fmiGetString(fmiComponent c, const fmiValueReference vr[],
                        size_t nvr, fmiString value[]);
```

Get actual values of variables by providing the variable handles.

To allow interpolation/approximation of the real output variables between communication steps (if they are used as inputs for other slaves) the derivatives of the outputs with respect to time can be read. Whether the slave is able to provide the derivatives of outputs is given by the unsigned integer capability flag `MaxOutputDerivativeOrder`. It delivers the maximum order of the output derivative. If the actual order is lower (because the order of integration algorithm is low), the retrieved value is 0.

Example: If the internal polynomial is of order 1 and the master inquires the second derivative of an output, the slave will return zero.

The derivatives can be retrieved by:

```
fmiStatus fmiGetRealOutputDerivatives (fmiComponent c,
                                       const fmiValueReference vr[],
                                       size_t nvr, const fmiInteger order[],
                                       fmiReal value[]);
```

Retrieves the n-th derivative of output values. Argument "`vr`" is a vector of "`nvr`" value references that define the variables whose derivatives shall be retrieved. The array "`order`" contains the order of the respective derivative (1 means the first derivative, 0 is not allowed). Argument "`value`" is a vector with the actual values of the derivatives.
Restrictions on using the function are the same as for the `fmiGetReal` function.

The returned outputs correspond to the current slave time. E. g. after a successful `fmiDoStep(...)` the returned values are related to the end of the time step.

This standard supports polynomial interpolation and extrapolation as well as more sophisticated signal extrapolation schemes like rational extrapolation, see Appendix D.

3.2.2. Computation

The computation of time steps is controlled by the following function.

```
fmiStatus fmiDoStep( fmiComponent c, fmiReal currentCommunicationPoint,
                     fmiReal communicationStepSize, fmiBoolean newStep);
```

The computation of a time step is started.
The parameter currentCommunicationPoint is the current communication point of the master (tci). Parameter `communicationStepSize` is the communication step size. If the master carries out an event iteration the parameter `communicationStepSize` is zero. The Parameter `newStep` is `fmiTrue` if the last communication step is accepted by the master

and a new communication step is started.

Depending on the internal state of the slave and the last call of `fmiDoStep(...)` the slave has to decide which action is to be done before the step is computed.

The function returns:

- `fmiOK` - if the communication step was computed successfully until its end.
- `fmiDiscard` – if the slave computed successfully only a subinterval of the communication step. The master can call the appropriate `fmiGetXXXStatus` functions to get further information.
- `fmiError` – the communication step could not be carried out at all. The master can try to repeat the step with other input values and/or an other communication step size.
- `fmiPending` – is returned if the slave executes the function in an asynchronous way. That means the slave starts the computation but returns immediately. The master has to call `fmiGetStatus(...,fmiDoStep,...)` to find out, if the slave is ready. `fmiCancelStep(...)` can be called to cancel the current computation. It is not allowed to call any other function during a pending `fmiDoStep(…)`.

```
fmiStatus fmiCancelStep(fmiComponent c);
```

Can be called if fmiDoStep returned fmiPending in order to stop the current asynchronous execution. The master calls this function if e.g. the co-simulation run is stopped by the user or one of the slaves. Afterwards it is only allowed to call the functions `fmiTerminateSlave`, `fmiResetSlave`, or `fmiFreeSlaveInstance`.

It depends on the capabilities of the slave which parameter constellations and calling sequences are allowed (see 3.5.1).

3.2.3. Retrieving of Status Information from the Slave

Status information is retrieved from the slave by the following functions:

```
fmiStatus         fmiGetStatus(fmiComponent c, const fmiStatusKind s,
                               fmiStatus* value);
fmiStatus     fmiGetRealStatus(fmiComponent c, const fmiStatusKind s,
                               fmiReal* value);
fmiStatus  fmiGetIntegerStatus(fmiComponent c, const fmiStatusKind s,
                               fmiInteger* value);
fmiStatus  fmiGetBooleanStatus(fmiComponent c, const fmiStatusKind s,
                               fmiBoolean* value);
fmiStatus   fmiGetStringStatus(fmiComponent c, const fmiStatusKind s,
                               fmiString* value);
```

Informs the master about the actual status of the simulation run. Which status information is to be returned is specified by the argument `fmiStatusKind`. It depends on the capabilities of the slave which status information can be given by the slave (see 3.5.1). If a status is required which cannot be retrieved by the slave it returns fmiDiscard.

```
typedef enum {fmiDoStepStatus,
              fmiPendingStatus,
              fmiLastSuccessfulTime,
             } fmiStatusKind;
```

Defines which status is inquired.

The following status information can be retrieved from a slave:

Status	*Type of retrieved value*	*Description*
`fmiDoStepStatus`	`fmiStatus`	Can be called when the `fmiDoStep` function returned `fmiPending`. The function delivers `fmiPending` if the computation is not finished. If the computation is finished meanwhile the function returns the result of the asynchronous executed `fmiDoStep(...)` call.
`fmiPendingStatus`	`fmiString`	Can be called when the `fmiDoStep` function returned `fmiPending`. The function delivers a string which informs about the status of the currently running asynchronous fmiDoStep computation.
`fmiLastSuccessfulTime`	`fmiReal`	Returns the time until the last communication step was computed successfully. Can be called after `fmiDoStep(...)` returned `fmiDiscard`.
`...`		

3.3. State Machine of Calling Sequence from Master to Slave

The following state machine demonstrates the possible calling sequence. The following abbreviations are used:

- `fmiFunc(...)` is one of the functions `fmiGetVersion()`, `fmiGetTypesPlatform()`, `fmiSetDebugLogging(...)`
- XXX is one of `Real`, `Integer`, `Boolean`, `String`
- `ts`, `tm`, `h` are internal variables of the slave

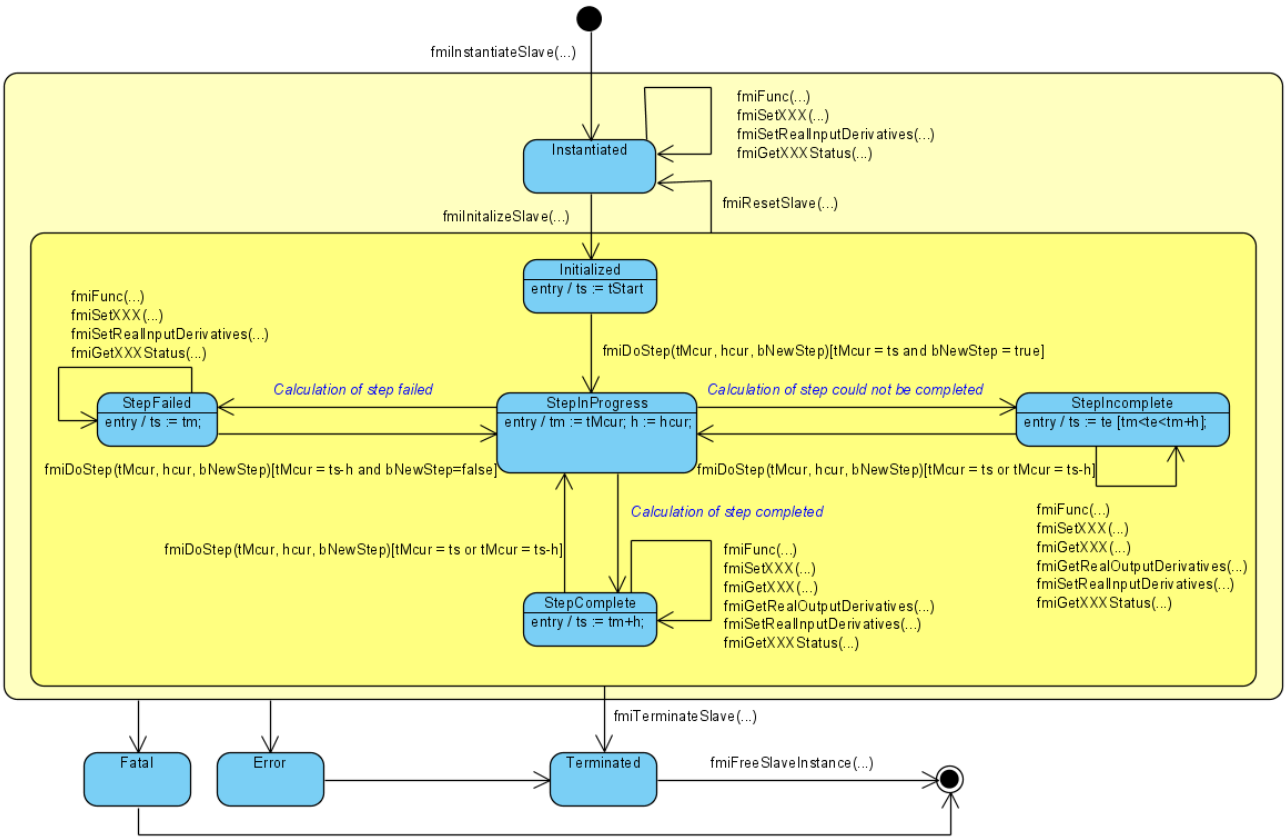


Figure 15: State-machine for the calling sequence of co-simulation interface C-functions

3.4. Pseudo Code Example

In the following example, the usage of the FMI functions is sketched in order to clarify the typical calling sequence of the functions in a simulation environment. The example is given in a mix of pseudo-code and “C”, in order to keep it small and understandable. We consider two slaves. Both have one continuous real input and one continuous real output which are connected in the following way:

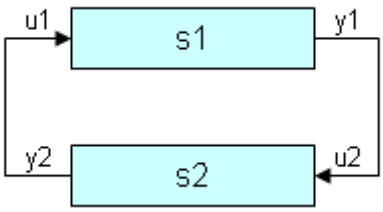


Figure 16: Connection graph of the slaves

We assume no algebraic dependency between input and output of each slave. The slaves do not support asynchronous execution of `fmiDoStep(...)`. The code demonstrates the simplest master algorithm as shown in section 2.2.5.

- Constant communication step size.
- No repeating of communication steps.
- The slaves do not support asynchronous execution of `fmiDoStep`.

The error handling is implemented in a very rudimentary way.

```
//////////////////////
//Initialization sub-phase

//Instantiate both slaves
fmiComponent s1 = fmiInstantiateSlave("Tool1", "", "Model1", "", ...);
fmiComponent s2 = fmiInstantiateSlave("Tool1", "", "Model2", "", ...);
// tStart needs to be between startTime and stopTime from the XML-file
tStart = 0;
// tStop needs to be between startTime and stopTime from the XML-file
tStop = 10;
// communication step size
h = 0.01;

//Initialize slaves
status = fmiInitializeSlave(s1, tStart, fmiTrue, tStop);
if(status == fmiOK)
      ret = fmiInitializeSlave(s2, tStart, fmiTrue, tStop);

//////////////////////
//Simulation sub-phase

//Current master time
tc = tStart;

while((tc < tStop) && (status == fmiOK))
      //retrieve outputs
      fmiGetReal(s1, ..., 1, &y1);
      fmiGetReal(s2, ..., 1, &y2);
      //set inputs
      fmiSetReal(s1, ..., 1, &y2);
      fmiSetReal(s2, ..., 1, &y1);

      //call slaves
      status = fmiDoStep(s1, tc, h, fmiTrue);
      if(status == fmiOK)
            status = fmiDoStep(s2, tc, h, fmiTrue);

      //increment master time
      tc+=communicationStepSize;
}

//////////////////////
//Shutdown sub-phase
if (status == fmiOK)
{
      fmiTerminateSlave(s1);
      fmiTerminateSlave(s1);
      //Reset slaves
      fmiResetSlave(s1);
      fmiResetSlave(s2);
}

if (status != fmiFatal)
{
      //cleanup slaves
      fmiFreeSlaveInstance(s1);
      fmiFreeSlaveInstance(s2);
}
```

3.5. The Co-Simulation Description Schema

The FMI for co-simulation reuses the XML schema encoding conventions and data types as defined by the FMI for model exchange (in section 3).

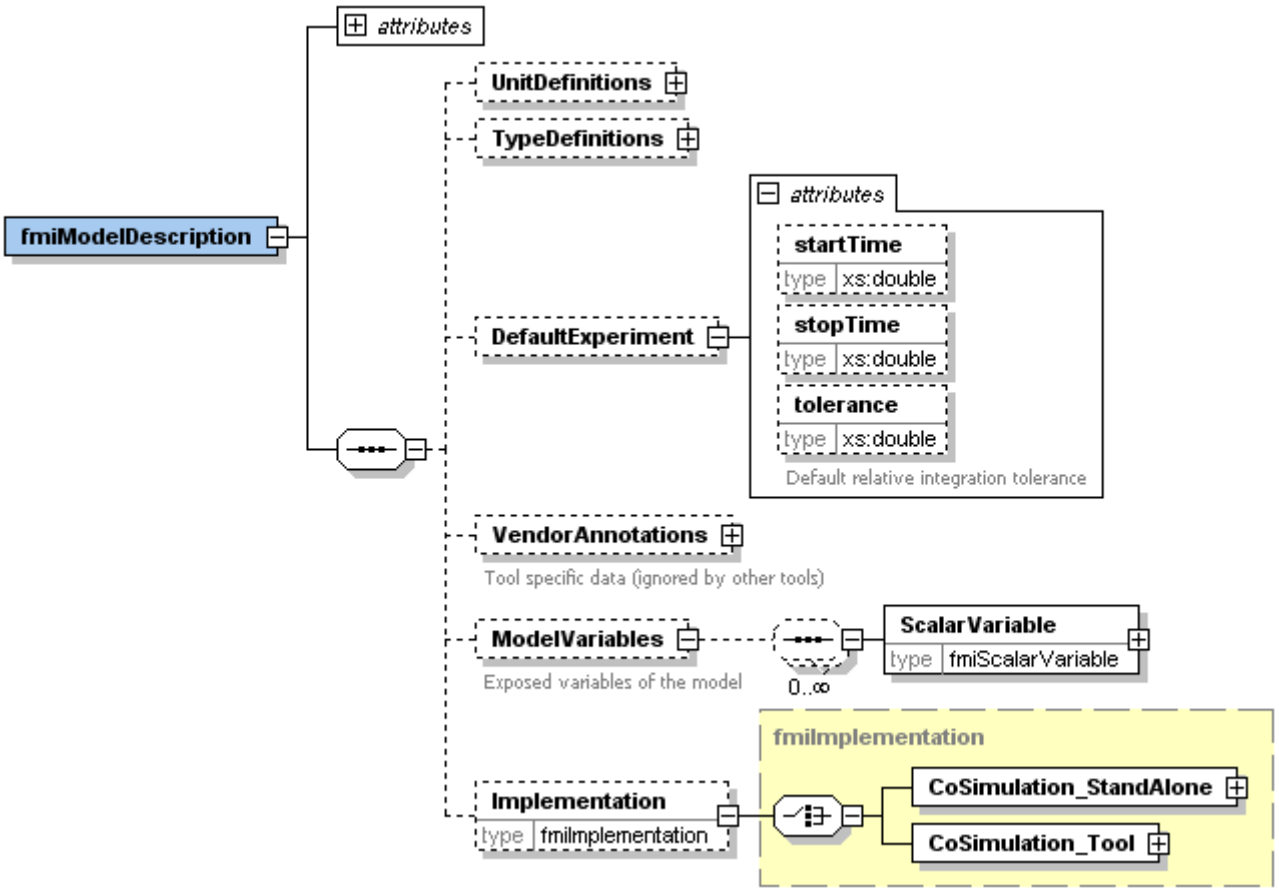
However, there are two important differences:

1. The “fmiModelDescription.xsd” definition has been modified to include an `Implementation` element.
2. An additional schema file “fmiImplementation.xsd” has been added to include the elements required to support co-simulation description.

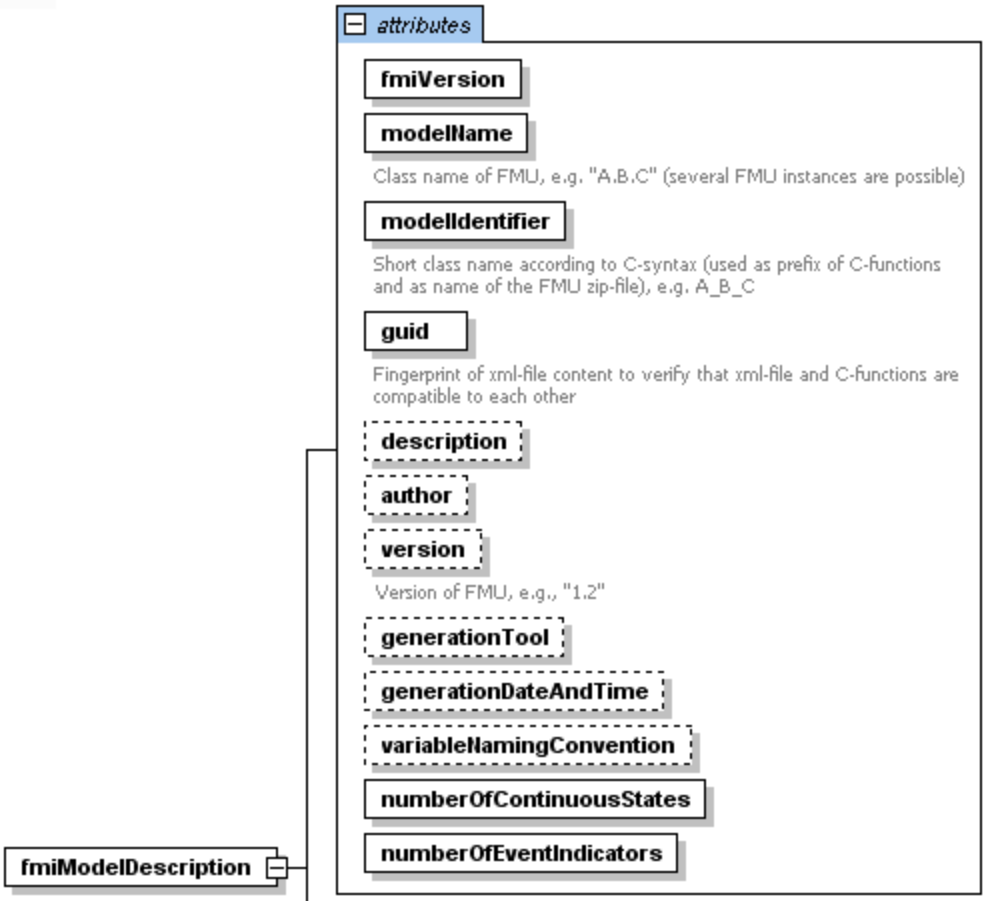
The following sections describe the amendments made to the `fmiModelDescription` schema and detailed information related to the co-simulation implementation element (`fmiImplementation`).

3.5.1. Description of a Model for Co-Simulation (fmiModelDescription)

The FMI for Co-Simulation modifies the model description format of FMI for Model Exchange, by appending an `Implementation` element; the reader is referred to section 3.1 of FMI for Model Exchange specification to understand the details of the top level description.



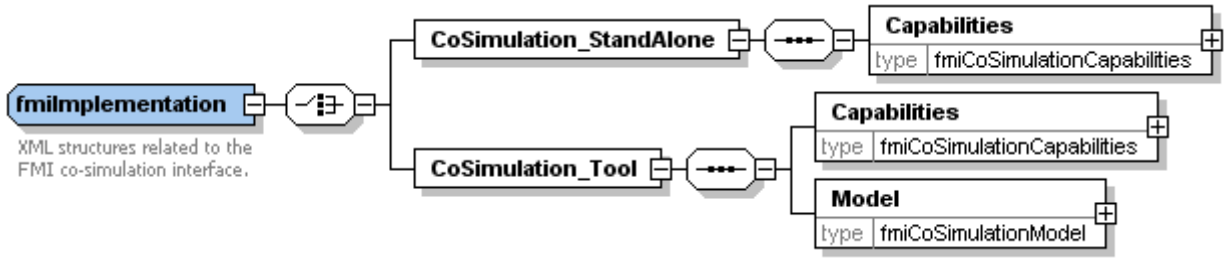
The `Implementation` element is optional; if present, the import tool should understand the model description as applying to co-simulation. As a consequence, the import tool must select the proper FMI API. The “attributes” part of fmiModelDescription is not changed.



3.5.2. Definition of an Implementation

An 'Implementation' in the co-simulation context can be either `CoSimulation_Tool` or `CoSimulation_StandAlone`.

The main difference between these implementations relates to the existence of the original model. A tool execution requires that the original tool is available to be executed in co-simulation mode; in a stand-alone execution, the slave is completely contained inside the FMU in source code or binary format (shared library).



The `Implementation` element can have one of the element choices `CoSimulation_StandAlone` or `CoSimulation_Tool`, which are described in the following table.

Name	*Description*
`CoSimulation_StandAlone`	This element is used when "FMI for Co-Simulation" code

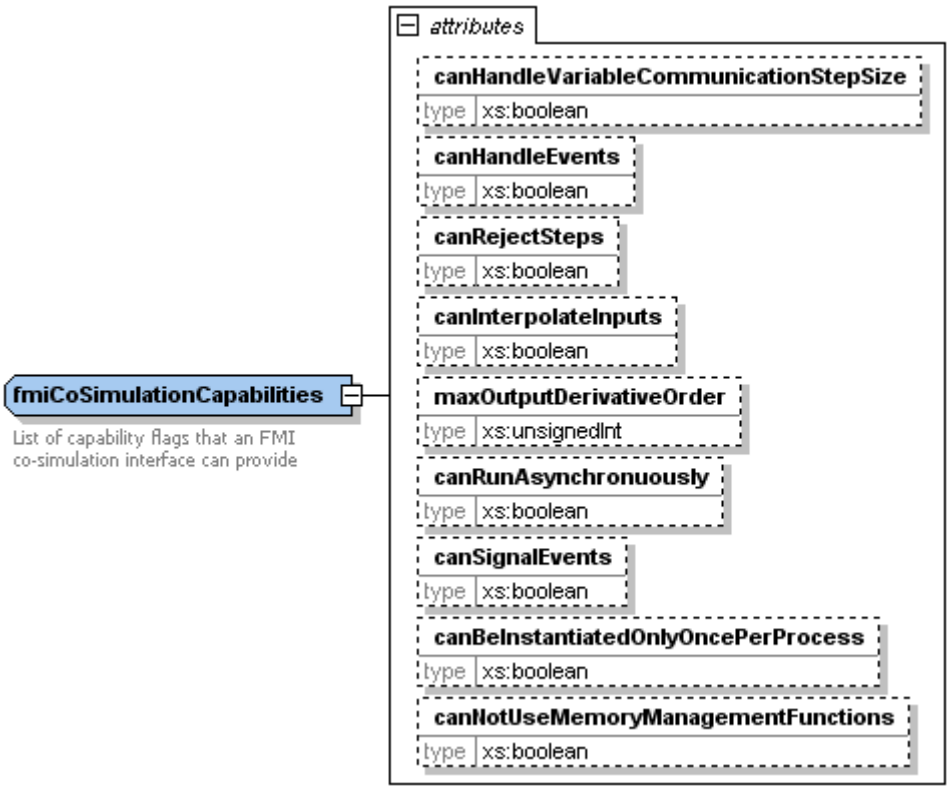
	generators are used to transfer models into compilable source code. The slave is available either in source code form or binary format (shared library or executable).
`CoSimulation_Tool`	This element is used when a slave simulation tool implements the "FMI for Co-Simulation" API and models can be directly executed without code generation being required.

The Element `CoSimulation_StandAlone` consists of a `Capabilities` element, the element `CoSimulation_Tool` consists of a sequence of `Capabilities` and `Model` elements.

The elements `Capabilities` and `Model` are described in the following sections.

3.5.2.1. Capability Flags

The `Capabilities` element is based on the type definition `fmiCoSimulationCapabilities`, which is defined as follows.



The `Capabilities` element can contain the following optional attributes.

Attribute Name	*Description*
`canHandleVariableCommunicationStepSize`	The slave can handle variable communication step size. The communication step size (parameter `communicationStepSize` of `fmiDoStep(...) )` has not to be constant for each call.
`canHandleEvents`	The slave supports event handling during co-simulation. The communication step size

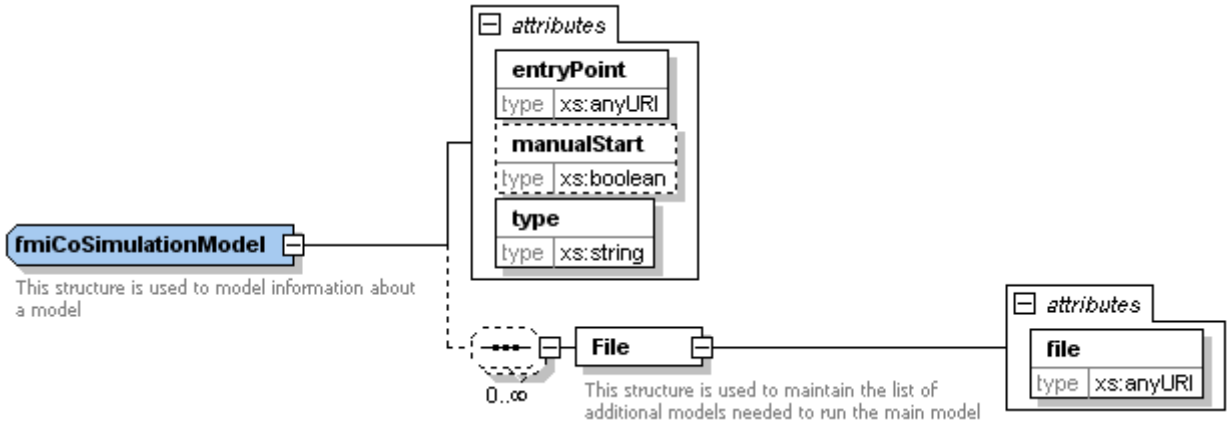
	(parameter `communicationStepSize` of `fmiDoStep(...))` can be zero.
`canRejectSteps`	This flag indicates the slave's capability to discard and repeat a communication step. The parameter `newStep` of `fmiDoStep(...)` can be `fmiFalse`. The parameter `currentCommunicationTime` can be constant in consecutive `fmiDoStep(...)` calls.
`canInterpolateInputs`	The slave is able to interpolate continuous inputs. Calling of `fmiSetRealInputDerivatives(...)` has an effect for the slave.
`maxOutputDerivativeOrder`	The slave is able to provide derivatives of outputs with maximum order. Calling of `fmiGetRealOutputDerivatives(...)` is allowed.
`canRunAsynchronuously`	This flag describes the ability to carry out the `fmiDoStep(...)` call asynchronously.
`canSignalEvents`	If a slave is able to provide information about events during a communication step, this flag has to be set `true`.
`canBeInstantiatedOnlyOncePerProcess`	This flag indicates cases (especially for embedded code), where only one instance per FMU is possible (multiple instantiation is default `= false`; if multiple instances are needed, the FMUs must be instantiated in different processes).
`canNotUseMemoryManagementFunctions`	If `true`, the slave uses its own functions for memory allocation and freeing only. The callback functions `allocateMemory` and `freeMemory` given in `fmiInstantiateSlave` are ignored.

All flags are optional. The flags have the following default values.

- `boolean: false`
- `unsignedInt: 0`

3.5.2.2. Model description

The Element `Model` is based on the type definition `fmiCoSimulationModel`, which is defined as follows.



Attribute Name	*Description*
`entryPoint`	The URI of the model to be executed by the slave simulator. Examples of URIs are: • `"fmu://resources/model/controller.mdl"` refers to a model within the FMU archive. • `"file://c:/model/controller.mdl"` refers to a model located externally to the FMU archive. • `"http://myserver:6456/models/controller.mdl"` refers to a model accessible via a web server.
`manualStart`	Indicates whether the model should be manually loaded and started by the user on the slave simulator. By providing this flag, the master tool can choose the adequate start sequence on the master side. By default, this flag is set to `false`.
`type`	A mime type that indicates the needed simulator and FMI wrapper for a simulator that needs to be started to instantiate an FMI Component.

In some cases, several model files may be transported, e.g. calibration files. In a tool coupling scenario, the master tool may need to know, which model needs to be opened to get the top level system.

Element `Model` contains an optional sequence of `File` elements. Each `File` element is used to represent an additional file required by the slave simulator.

Attribute Name	*Description*
`file`	The URI of a file needed by the slave simulator to execute the native model. An example of file URI entry is `"fmu://resources/model/myReferencedModel.mdl"` that refers to a model within the FMU archive.

4. Model Distribution

The major part of this section is directly taken from the specification document FMI for Model Exchange, as FMI for Co-Simulation builds upon concepts of the previous. Additional remarks will point out, where changes were made specifically for the co-simulation case.

An FMU description consists of several files. An FMU may be distributed in textual and/or in binary format. All relevant files are stored in a zip-file with a pre-defined structure. The name of the zip-file must be identical to the "modelIdentifier" stored as xml-attribute in the Model Description File and used as defined symbol MODEL_IDENTIFIER with header file `fmiFunctions.h`. The extension of the zip-file must be ".fmu", e.g., "HybridVehicle.fmu". The compression method used for the zip-file must be "deflate" (most free tools, e.g. zlib, offer only the common compression method "deflate").

Every FMU is distributed by its own zip-file. This zip-file has the following structure:

```
// Structure of zip-file of an FMU
modelDescription.xml           // Description of model (required file)
model.png                      // Optional image file of model icon
documentation                  // Optional directory containing the model documentation
    _main.html                 // Entry point of the documentation
    <other documentation files>
sources                        // Optional directory containing all C-sources
  // all needed C-sources and C-header files to compile and link the model
  // with exception of: fmiPlatformTypes.h and fmiFunctions.h
binaries                       // Optional directory containing the binaries
    win32                      // Optional binaries for 32-bit Windows
        <modelIdentifier>.dll  // DLL of the model interface implementation
        // Optional object Libraries for a particular compiler
        VisualStudio8          // Binaries for 32-bit Windows generated with
                               // Microsoft Visual Studio 8 (2005)
        <modelIdentifier>.lib  // Binary libraries
        gcc3.1                 // Binaries for gcc 3.1
          ...
    win64                      // Optional binaries for 64-bit Windows
      ...
    linux32                    // Optional binaries for 32-bit Linux
      ...
    linux64                    // Optional binaries for 64-bit Linux
      ...
resources                      // Optional resources needed by the model
    < data in model specific files which will be read during initialization >
```

The FMU must be distributed with at least one implementation, i.e., either sources or one of the binaries for a particular machine[1]. It is also possible to provide the sources and binaries for different target machines altogether in one zip-file. The names "win32", "win64", "linux32", "linux64" are standardized, as well as the names "VisualStudioX" and "gccX" that define the compiler with which the binary has been generated. Further names can be introduced by vendors. Typical scenarios are to provide binaries only for one machine type (e.g. on the machine where the target simulator is running and for which licenses of run-time libraries are available) or to provide only sources (e.g. for translation and download for a particular micro-processor). If run-time libraries cannot be shipped due to licensing, special handling is needed, e.g., by providing the run-time libraries at appropriate places by the receiver.

[1] Note that the implementation can be either according to FMI for Model Exchange or FMI for Co-Simulation. For the second, see section 3.5.2 for details. Appendix B gives an outlook of a possible future generalization.

FMI for Co-Simulation provides the means for two kinds of implementation: `CoSimulation_Tool` and `CoSimulation_StandAlone`. In the first scenario a slave tool specific wrapper dll has to be provided as the binary, in the second a compiled or source code version of the model with its solver is stored (see section 2.1 for details).

In directory "`resources`", additional data can be provided in model specific formats, typically for tables and maps used in the model. This data must be read into the model at the latest during initialization (`fmiInitializeSlave`). The actual file names in the zip-file to access the data files can either be hard-coded in the generated model functions, or the file names can be provided as string parameters via the `fmiSetString` function (see Functional Mock-up Interface for Model Exchange **MODELISAR** (ITEA 2 - 07006) January 26, 2010 Page 41 of 56).

In the case of a co-simulation implementation of `CoSimulation_Tool` type, the "`resources`" directory can contain the model source file in the tool specific file format.

Note that the header files `fmiPlatformTypes.h` and `fmiFunctions.h` are not included in the FMU due to the following reasons:

`fmiPlatformTypes.h` makes no sense in the "`sources`" directory, because if sources are provided, then the target simulator defines this header file and not the FMU. This header file is not included in the "`binaries`" directory, because it is implicitly defined by the platform directory (e.g. win32 for 32-bit machine or linux64 for 64-bit machine). Furthermore, the version that was used to construct the FMU can also be inquired via function `fmiGetModelTypesPlatform()`.

`fmiFunctions.h` is not needed in the "`sources`" directory, because it is implicitly defined by atttribute `fmiVersion` in file **`modelDescription.xml`**. Furthermore, in order that the C-compiler can check for consistent function arguments, the header file from the target simulator should be used when compiling the C-sources. It would therefore be counter productive (unsafe), if this header file would be present. This header file is not included in the "`binaries`" directory, since this header file was already utilized to build the target simulator executable. Via attribute `fmiVersion` in file **`modelDescription.xml`** or via function call `fmiGetVersion()` the version number of this header file used to construct the FMU can be deduced.

5. Literature

AMESim: www.lmsintl.com/

AUTOSAR: www.autosar.org

Dymola: www.dynasim.se

EXITE: www.extessy.com

R. Kübler, W. Schiehlen: *Two methods of simulator coupling*. - Mathematical and Computer Modelling of Dynamical Systems **6**(2000)93-113.

MODELISAR Glossary (2009): MODELISAR WP2 Glossary and Abbreviations. Version 1.0, June 9, 2009.

MODELISAR 2010: Functional Mock-up Interface for Model Exchange, Version 1.0, January 26, 2010, http://www.functional-mockup-interface.org/fmi.html

Silver: www.qtronic.de/de/silver.html

Simpack: www.simpack.com

SimulationX: www.simulationx.com

XML: www.w3.org/XML, en.wikipedia.org/wiki/Xml

Appendix A Contributors

A.1 Version 1.0

The Functional Mock-up Interface subproject inside MODELISAR was initiated and organized by Daimler AG. The development FMI for Co-Simulation version 1.0 was performed within WP200 of the MODELISAR ITEA2 project, organized by the WP200 work package leader Dietmar Neumerkel (Daimler). FMI for Co-Simulation was developed in three subgroups: “Solver Coupling” headed by Martin Arnold (University Halle) and Torsten Blochwitz (ITI), “Tool Coupling” headed by Jörg-Volker Peetz (Fraunhofer SCAI), and “Control Logic” headed by Manuel Monteiro (Atego). The essential part of the design of this version was performed by (alphabetical list):

Martin Arnold, University Halle, Germany
Constanze Bausch, Atego Systems GmbH, Wolfsburg, Germany
Torsten Blochwitz, ITI GmbH, Dresden, Germany
Christoph Clauß, Fraunhofer IIS EAS, Dresden, Germany
Manuel Monteiro, Atego Systems GmbH, Wolfsburg, Germany
Thomas Neidhold, ITI GmbH, Dresden, Germany
Jörg-Volker Peetz, Fraunhofer SCAI, St. Augustin, Germany
Susann Wolf, Fraunhofer IIS EAS, Dresden, Germany

This version was evaluated with prototypes implemented for (alphabetical list):

SimulationX by Torsten Blochwitz and Thomas Neidhold (ITI GmbH),
Master algorithms by Christoph Clauß (Fraunhofer IIS EAS)

The following MODELISAR partners participated at FMI design meetings and contributed to the discussion (alphabetical list):

Martin Arnold, University Halle, Germany
Jens Bastian, Fraunhofer IIS EAS, Dresden, Germany
Constanze Bausch, Atego Systems GmbH, Wolfsburg, Germany
Torsten Blochwitz, ITI GmbH, Dresden, Germany
Christoph Clauß, Fraunhofer IIS EAS, Dresden, Germany
Manuel Monteiro, Atego Systems GmbH, Wolfsburg, Germany
Thomas Neidhold, ITI GmbH, Dresden, Germany
Dietmar Neumerkel, Daimler AG, Böblingen, Germany
Martin Otter, DLR, Oberpfaffenhofen, Germany
Jörg-Volker Peetz, Fraunhofer SCAI, St. Augustin, Germany
Tom Schierz, University Halle, Germany
Klaus Wolf, Fraunhofer SCAI, St. Augustin, Germany

Appendix B Features for Future Versions

In this appendix, features are summarized that are already known to be missing and might be added in a future release.

Event Handling

Event Handling is not supported at the moment. Modelisar SWP202 will work on event handling algorithms. Useful extensions could be:

`fmiEventHappened`	`fmiBoolean`	Delivers `fmiTrue` if during computation of the last communication step a discontinuity happened, that affects an output. Is delivered only if the capability flag `canSignalEvents` is set.
`fmiEventTime`	`fmiReal`	Time of the event, happened during the last communication step. Is delivered only if the capability flag `canSignalEvents` is set.

Other extensions are e.g. error criteria of the slave which can be used for sophisticated co-simulation master algorithms.

Efficient Handling of Time Events

We postpone the efficient handling of time events in order to avoid overloading of the discussion. Efficient time event handling should be developed together with the FMI ME.

The efficient and numerical robust handling of time events is essential to include controller algorithms in a co-simulation scenario. At first we consider time events with a constant sample rate. The number of sample rates is defined in the slave description file.

```
fmiStatus fmiGetSampleRates(fmiComponent c, const fmiSampleRateInfo st[]);
```

Retrieves the sample rates of the slave. Parameter "`st`" is an array of fmi `fmiSampleRateInfo` structures. The dimension of the array has to be consistent with the number of sample rates given in the slave description file. The function can be called after `fmiLoadModel` and must be called before the simulation run is started.

```
typedef struct {
                fmiInteger expTimeBase;
                fmiUnsigned startTime;
                fmiUnsigned sampleRate;
               } fmiSampleRateInfo;
```

This structure contains the information about one sample rate of the slave. To avoid inaccuracies an integer representation is used. The sample rate and start time are defined by integer multipliers of a time base. The time base is given by its exponent of base 10.

This structure contains the information about one sample rate of the slave:

- `expTimeBase`: is the exponent of 10 of the time base in seconds

- `startTime`: defines the occurrence of the first sample hit
- `startTime`: defines the sample rate

The startTime counts from the start time of the simulation run. It is defined by the parameter `tStart` of `fmiInitializeSlave(...)`.

A sample rate of 10 ms is e.g. given by:

- `base=-3`
- `sampleRate=10`

`or:`

- `base=-4`
- `sampleRate=1`

Remark: In the Modelica Language Design group a discussion about integer time representation (at least for a numerical robust definition of time events) is going on. Here we think about using (similar to SPICE or VHDL) special characters ('n'… nano, 'u'… mikro, 'm'... mili, ...) of the time base. This would be a possibility too.

Possibly we should consider how sample rates are specified in AUTOSAR.

If the slave exposes at least one sample rate it has to be informed by the master when the sample time instance is achieved:

```
fmiStatus setSampleTimeStatus(fmiComponent c, const fmiBoolean* s[]);
```

Informs the model if one of the sample times is reached. "`s`" is a boolean array. The dimension of the array has to be consistent with the number of sample rates given in the slave description file. If one of the sample time instances is reached the corresponding element in "`s`" is set to `fmiTrue`.

Can be called by the master after an event step is signaled by a call of `fmiSetBooleanStatus(s, fmiEvent, fmiTrue)`.

Sample time instances are defined by:

$ts = startTime*10^{expTimeBase}+i*sampleRate*10^{expTimeBase}$

$(i=0,1...)$

Discarding and Repeating of Communication Steps

If the slave sets the capability flag `canRejectSteps` to `fmiTrue` the master can use more sophisticated co-simulation algorithms which require the repeating of communication steps. Currently the master signals that by calling `fmiDoStep` with parameter `newStep = fmiFalse`. In this case the slave has to reject its last computed communication step and repeat the computation.

This mechanism is not efficient for the following use case. If a master will only go forward, the slave should be informed about that. Otherwise it has to store its state at the beginning of each computation of a communication step, because the next `fmiDoStep` call could require a discarding of the last communication step. This could be time consuming. It would be better to have special functions for

storing and restoring several states of the slave, e.g.:

```
fmiSaveState(fmiComponent c, size_t index)

fmiRestoreState(fmiComponent c, size_t index)
```

which can be called explicitly by the master. The parameter index identifies which state the slave has to restore.

Also for usage of FMU's in training simulators (e.g. for nuclear power plants) an explicit save and restore mechanism could be useful. The training master (a human being) may want to have a snap shot at a particular time point in order to restart from this point at some other time instant.

Appendix C Further Examples for Simulator Coupling

In the following, two further examples demonstrating the coupling of three simulators are given in a mix of pseudo-code and “C”.

C.1 Example 1: Parallel simulation and input/output of different kinds

The three slaves are connected in the following way:

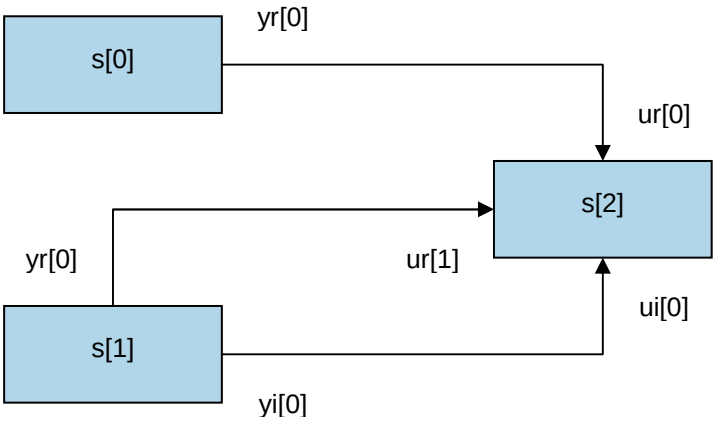


Figure 17: Connection graph of the slaves of example 1

Simulator s[0] has one continuous real output yr[0], simulator s[1] has one continuous real output yr[0] and one integer output yi[0], and simulator s[2] has two real inputs ur[0], ur[1] and one integer input ui[0]. Simulators s[0] and s[1] have the same priority and there does not exist a cycle, so that both simulators can work in parallel.

C.2 Example 2: Cycle (feedback)

The three slaves are connected in the following way:

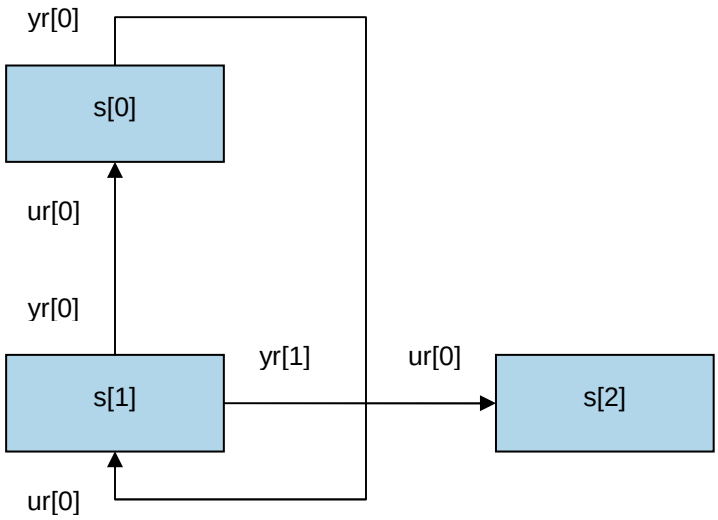


Figure 18: Connection graph of the slaves of example 2

Simulator s[0] has one continuous real input ur[0] and one continuous real output yr[0], simulator s[1] has one continuous real input ur[0] and two continuous real outputs yr[0] and yr[1], and simulator s[2] has one real inputs ur[0]. Simulators s[0] and s[1] have the same priority but this time a cycle exists, so that both simulators cannot work in parallel.

C.3 Pseudo Code for both examples

The code demonstrates a more elaborate master algorithm than shown in section 2.2.5.

- Constant communication step size.
- Repeating of communication steps / iteration.
- Parallelization / multiple threads

The error handling is again implemented in a very rudimentary way.

```
//////////////////////
// Initialization sub-phase

// Graph structure (taken from configuration file)
// Number of slaves
nsim = 3;
// Priority of slaves 0...nsim-1
priority[0] = 0;
priority[1] = 0;
priority[2] = 1;
// At priority i do cycles exist? yes: cycles[i] = 1, no: cycles [i] = 0
cycles[1] = 0;
#ifdef Example1
cycles[0] = 0;
#else
cycles[0] = 1;
#endif
// Read the ModelDescription XML files of the FMUs
// Instantiate slaves
```

```
for (i = 0; i < nsim; ++i) {
      s[i]->component = fmiInstantiateSlave("Instance_i", "", "FMU_i.dll",
            "",...);
      if (s[i]->component == NULL)
            // error
}
// tStart needs to be between startTime and stopTime from the XML-file
tStart = 0;
// tStop needs to be between startTime and stopTime from the XML-file
tStop = 10;
// Communication step size
h = 0.01;
// Number of inputs and outputs of slave s[i] (taken from XML-file),
// n[u|y][r|i|b|s] is the number of components of [real|integer|boolean|string]
// [input|output] array [u|y][r|i|b|s]
#ifdef Example1
s[0]->nyr = 1;
s[1]->nyr = 1;
s[1]->nyi = 1;
s[2]->nur = 2;
s[2]->nui = 1;
#else
s[0]->nur = 1;
s[0]->nyr = 1;
s[1]->nur = 1;
s[1]->nyr = 2;
s[2]->nur = 1;
#endif

// Initialize slaves
for (i = 0; i < nsim; ++i) {
      status = fmiInitializeSlave(s[i]->component, tStart, fmiTrue, tStop);
      if (status != fmiOK)
            // error
}

////////////////////
// Simulation sub-phase

// Current master time
tc = tStart;

while ((tc < tStop) && (status == fmiOK)) {
      // Zero communication step size at first step and for event iteration
      if (firstStep || event)
            hStep = 0;
      else
            hStep = communicationStepSize;
      // Call slaves regarding priority
      for (prior = 0; prior < maxPriority; ++prior) {
            if (cycles[prior] == 0) { // no cycle, parallel execution of slaves
                  // Call slaves of priority prior
                  for (i = 0; i < nsim; ++i)
                        if (priority[i] == prior) {
                              // Open thread
                              // Set inputs for slaves of priority prior
                              fmiSetReal(s[i]->component, ..., s[i]->nur,
                                    s[i]->ur);
                              fmiSetInteger(s[i]->component, ..., s[i]->nui,
                                    s[i]->ui);
                              status = fmiDoStep(s[i]->component, tc, hStep,
```

```
                                    fmiTrue);
                              if (status == fmiError || status == fmiFatal)
                                    // error
                              // Retrieve outputs for slaves of priority prior
                              fmiGetReal(s[i]->component, ..., s[i]->nyr,
                                    s[i]->yr);
                              fmiGetInteger(s[i]->component, ..., s[i]->nyi,
                                    s[i]->yi);
                              // Close thread
                        }
            } else { // cycle, serial execution of slaves, iteration
                  itSteps = 0;
                  newStep = fmiTrue;
                  // Iteration
                  do {
                        ++itSteps;
                        // Backup of values exchanged between slaves for error
                        // check
                        oldValues = values;
                        // Call slaves of priority prior
                        for (i = 0; i < nsim; ++i)
                              if (priority[i] == prior) {
                                    // Set inputs for slaves of priority prior
                                    fmiSetReal(s[i]->component, ..., s[i]->nur,
                                          s[i]->ur);
                                    fmiSetInteger(s[i]->component, ...,
                                          s[i]->nui, s[i]->ui);
                                    status = fmiDoStep(s[i]->component, tc,
                                          hStep, newStep);
                                    if (status==fmiError || status==fmiFatal)
                                          // error
                                    // Get outputs for slaves of priority prior
                                    fmiGetReal(s[i]->component, ..., s[i]->nyr,
                                          s[i]->yr);
                                    fmiGetInteger(s[i]->component, ...,
                                          s[i]->nyi, s[i]->yi);
                              }
                        newStep = fmiFalse;
                        // Check error between old and new values of iteration
                        err = errorCheck(values, oldValues);
                  } while (err > 0 && itSteps < maxItSteps);
            }
      }
      //increment current master time
      tc += hStep;
}
```

```
//////////////////////
// Shutdown sub-phase
if (status == fmiOK) {
      // Terminate slaves
      for (i = 0; i < nsim; ++i)
            fmiTerminateSlave(s[i]->component);
      // Reset slaves
      for (i = 0; i < nsim; ++i)
            fmiResetSlave(s[i]->component);
}

if (status != fmiFatal)
      // Cleanup slaves
      for (i = 0; i < nsim; ++i)
            fmiFreeSlaveInstance(s[i]->component);
```

Appendix D Higher Order Signal Extrapolation

Within each communication step $tc_i \rightarrow tc_{i+1}$ the slave inputs $u(t)$ are approximated using function values at $t = tc_i$ and possibly up to $r-1$ more previous communication points $t = tc_{i-1}, t = tc_{i-2}, ..., t = tc_{i+1-r}$ for some $r > 1$. In a serial implementation, it is even possible that some slaves may use function values $u(t)$ at the new communication point $t = tc_{i+1}$.

In most co-simulation algorithms, polynomial approximations of slave inputs are used:

- Constant ("zero order") extrapolation based on data at $t = tc_i$:

$$u(t) \approx u_{\mathrm{E},0}(t) := u(tc_i) \ , \ (tc_i \le t \le tc_{i+1}) \ ,$$

- Linear ("first order") extrapolation based on data at $t = tc_{i-1}$ and $t = tc_i$:

$$u(t) \approx u_{\mathrm{E},1}(t) := u(tc_i) + \dot{u}(tc_i)(t - tc_i) \ \text{ with } \ \dot{u}(tc_i) := \frac{u(tc_i) - u(tc_{i-1})}{tc_i - tc_{i-1}} \ , \ (tc_i \le t \le tc_{i+1}) \ ,$$

- Linear ("first order") interpolation based on data at $t = tc_i$ and $t = tc_{i+1}$:

$$u(t) \approx u_{\mathrm{I},1}(t) := u(tc_i) + \dot{u}(tc_i)(t - tc_i) \ \text{ with } \ \dot{u}(tc_i) := \frac{u(tc_{i+1}) - u(tc_i)}{tc_{i+1} - tc_i} \ , \ (tc_i \le t \le tc_{i+1}) \ ,$$

- Quadratic ("second order") extrapolation based on data at $t = tc_{i-2}$, $t = tc_{i-1}$ and $t = tc_i$:

$$u(t) \approx u_{\mathrm{E},2}(t) := u(tc_i) + \dot{u}(tc_i)(t - tc_i) + \frac{1}{2}\ddot{u}(tc_i)(t - tc_i)^2 \ , \ (tc_i \le t \le tc_{i+1}) \text{ with}$$

$$\ddot{u}(tc_i) := \left(\frac{u(tc_i) - u(tc_{i-1})}{tc_i - tc_{i-1}} - \frac{u(tc_{i-1}) - u(tc_{i-2})}{tc_{i-1} - tc_{i-2}} \right) \Bigg/ \left(\frac{tc_i - tc_{i-2}}{2} \right) \ \text{ and}$$

$$\dot{u}(tc_i) := \frac{u(tc_i) - u(tc_{i-1})}{tc_i - tc_{i-1}} + \frac{1}{2}\ddot{u}(tc_i)(tc_i - tc_{i-1}) \ ,$$

- Quadratic ("second order") interpolation based on data at $t = tc_{i-1}$, $t = tc_i$ and $t = tc_{i+1}$:

$$u(t) \approx u_{\mathrm{I},2}(t) := u(tc_i) + \dot{u}(tc_i)(t - tc_i) + \frac{1}{2}\ddot{u}(tc_i)(t - tc_i)^2 \ , \ (tc_i \le t \le tc_{i+1}) \ \text{ with}$$

$$\ddot{u}(tc_i) := \left(\frac{u(tc_{i+1}) - u(tc_i)}{tc_{i+1} - tc_i} - \frac{u(tc_i) - u(tc_{i-1})}{tc_i - tc_{i-1}} \right) \Bigg/ \left(\frac{tc_{i+1} - tc_{i-1}}{2} \right) \ \text{ and}$$

$$\dot{u}(tc_i) := \frac{u(tc_{i+1}) - u(tc_i)}{tc_{i+1} - tc_i} - \frac{1}{2}\ddot{u}(tc_i)(tc_{i+1} - tc_i)$$

and so on. In all these examples, a *Nordsieck* like representation of the interpolating and extrapolating polynomials was used that expresses the approximation of $u(t)$ in terms of powers of $(t-tc_i)$ with coefficients being defined by difference quotients of u. Note, that the denominators of these difference quotients may be further simplified in the case of equidistant communication points $tc_{i-2}, tc_{i-1}, tc_i, tc_{i+1}, \ldots$ with fixed communication step size hc:

$$tc_{i+1} - tc_i = tc_i - tc_{i-1} = tc_{i-1} - tc_{i-2} = hc\,, \qquad \frac{tc_{i+1} - tc_{i-1}}{2} = \frac{tc_i - tc_{i-2}}{2} = hc\,, \qquad \ldots\ .$$

The Nordsieck like representation of the slave inputs is favourable since it abstracts from algorithmic details (like data interpolation vs. data extrapolation) and requires at a communication point $t = tc_i$ just the transfer of the derivative vector $\left(u(tc_i), \dot{u}(tc_i), \ddot{u}(tc_i), \ldots, \frac{d^k u}{dt^k}(tc_i)\right)$ from master to slave to define the extrapolated or interpolated slave inputs $u(t)$ in communication step $tc_i \rightarrow tc_{i+1}$. For polynomial slave inputs $u(t)$, the length $k+1$ of this derivative vector determines the degree k of the polynomial and the components of the derivative vector contain in increasing order the coefficients of $(t-tc_i)^j / j!$ for $j = 0,1,\ldots,k$:

$$u(t) \approx \sum_{j=0}^{k} \frac{1}{j!} \frac{d^j u}{dt^j}(tc_i)\,(t-tc_i)^j\ .$$

The Nordsieck representation of polynomials is not restricted to classical interpolation polynomials but may be used as well for more sophisticated co-simulation techniques like the extrapolated interpolation (S. Dronka, J. Rauh: *Co-Simulation-Interface for User-Force-Elements*. – SIMPACK User Meeting 2006, http://www.simpack.com/uploads/media/um06_dc_research-dronka_05.pdf) or interpolated extrapolation of slave inputs. Also the extension to interpolation by rational functions and related approaches is straightforward.

Practical experience and recent theoretical investigations (M. Arnold: *Stability of sequential modular time integration methods for coupled multibody system models*. - Journal of Computational and Nonlinear Dynamics, **5**(2010)031003, doi:10.1115/1.4001389) show that higher order signal extrapolation increases the risk of numerical instability in co-simulation. Therefore, polynomial signal extrapolation is typically restricted to constant, linear or quadratic polynomials. In principle, however, interpolation polynomials of arbitrary degree could be computed and evaluated very efficiently using their Newton representation that may be found in any textbook on numerical mathematics. The coefficients $u(tc_i), \dot{u}(tc_i), \ldots$ of the Nordsieck representation are obtained by Taylor expansion of the interpolation polynomial at $t = tc_i$.

Appendix E Communication Step size Control

In contrast to classical (mono-disciplinary) simulation techniques in system dynamics, state-of-the-art master algorithms in co-simulation are even today based on constant communication step sizes hc and do not provide any automatic error control. Constant communication step sizes may restrict strongly the efficiency of co-simulation algorithms if the solution behavior changes considerably during time integration. Furthermore, the selection of an "optimal" constant communication step size hc requires much practical experience or time-consuming numerical tests.

Therefore, error control and the adaptive selection of (variable) communication step sizes hc_i may contribute to more reliable and more efficient master algorithms. The basic ideas of classical step size control in time integration are described in great detail in the literature on numerical solution of ordinary differential equations (U. Ascher, L.R. Petzold: *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. - SIAM Philadelphia, 1998). The practical implementation in the explicit Runge-Kutta code DOPRI5 (http://www.unige.ch/~hairer/prog/nonstiff/dopri5.f) may be considered as an advanced reference implementation in classical ODE time integration.

Step size control is based on the component based comparison of an error estimate EST with user defined bounds ATOL, RTOL in each time step:

$$\mathrm{err} := \sqrt{\frac{1}{m}\sum_{j=1}^{m}\left(\frac{\mathrm{EST}_j}{\mathrm{ATOL}_j + \mathrm{RTOL}_j\,|y_j|}\right)^2}\ .$$

The error indicator err shows if the (estimated) error EST is below the given error bounds ATOL, RTOL (resulting in $\mathrm{err} \le 1$). If $\mathrm{err} > 1$, then the (estimated) error is too large and the current step should be repeated with smaller step size.

The crucial part of this error control strategy is the efficient evaluation of a reliable error estimate EST that may be obtained comparing two numerical solutions of different accuracy. In ODE and DAE time integration, the nominal numerical solution in a time step $T \to T + h$ is compared

- with the predictor of a linear multistep method in predictor-corrector form,
- with an embedded Runge-Kutta solution of different order in the case of Runge-Kutta methods or
- with the result of two time steps of reduced step size ($T \to T + h/2$ and $T + h/2 \to T + h$, Richardson extrapolation).

The details of an efficient implementation are sophisticated, see the above given references. The use of Richardson extrapolation for communication step size control in co-simulation is discussed in (R. Kübler: *Modulare Modellierung und Simulation mechatronischer Systeme*. Fortschritt-Berichte VDI Reihe 20, Nr. 327. VDI-Verlag Düsseldorf, 2000).

In the context of co-simulation, vector EST should estimate in each communication step $tc_i \to tc_{i+1}$ all errors in the slave outputs $y(tc_{i+1})$ that result from the use of approximated slave inputs

$u(t), (tc_i \le t \le tc_{i+1})$. Then, the error indicator err shows if the communication step size hc_i was sufficiently small to meet some user defined error bounds ATOL, RTOL or not. Furthermore, the ratio between the error indicator err and its optimal value 1.0 may be used to define a posteriori an "optimal" communication step size hc_{opt} :

$$hc_{\text{opt}} := \alpha\, hc_i \left(\frac{1}{err}\right)^{\frac{1}{k+1}}$$

with a safety factor $\alpha \in [0.8, 0.9]$ and k denoting the approximation order of the signal extrapolation for slave inputs $u(t)$. Note, that hc_{opt} is always smaller than the current communication step size hc_i if the error estimate EST exceeds the given tolerances ($\text{err} > 1$).

If all slaves in a co-simulation environment support variable communication step sizes hc_i (capability flag `canHandleVariableCommunicationStepSize`), then the master algorithm may use this optimal communication step size hc_{opt} for the next communication step $tc_{i+1} \rightarrow tc_{i+2} := tc_{i+1} + hc_{i+1}$ with $hc_{i+1} := hc_{\text{opt}}$. (At least) a warning message should be generated whenever the error indicator err exceeds its critical value 1.0.

In a really error controlled master algorithm, however, a communication step resulting in an error indicator $\text{err} > 1$ has to be repeated with smaller communication step size ("rejected" communication steps). FMI for Co-Simulation supports such step rejections by repeated calls of `fmiDoStep(…)` with one and the same input parameter `currentCommunicationPoint` and different input parameters `communicationStepSize`. To keep the discussion in this appendix compact the parameters `currentCommunicationPoint` and `communicationStepSize` are abbreviated by $t_{\text{cur}}^{\text{M}}$ and h_{cur}, respectively. I.e., `fmiDoStep(…)` is called to perform one communication step $t_{\text{cur}}^{\text{M}} \rightarrow t_{\text{cur}}^{\text{M}} + h_{\text{cur}}$.

In a practical implementation of advanced error controlled master algorithms, all slaves of the co-simulation environment have to support repeated calls with one and the same current communication time t_{cur}^{M} and different communication step sizes h_{cur} (capability flag `canRejectSteps`). It is mandatory for a successful co-simulation with communication step size control that all slaves in the co-simulation environment guarantee that repeated calls of `fmiDoStep(…)` with identical input data (i.e. with identical t_{cur}^{M} and h_{cur} and identical slave inputs $u(t)$) result in *exactly* identical output data. Therefore, the capability to discard and to repeat communication steps (capability flag `canRejectSteps`) requires substantial modifications and extensions of existing simulation software that is typically designed to solve model equations and to store simulation data going step by step forward in time from initial time t_{start} to end time t_{stop}.

With advanced error controlled master algorithms there are two fundamentally different types of communication steps $t_{\text{cur}}^{\text{M}} \rightarrow t_{\text{cur}}^{\text{M}} + h_{\text{cur}}$:

- Accepted communication steps: All slaves perform successfully the communication step and generate simulation data that should be saved to file. At $t_{\text{cur}}^{\text{M}} + h_{\text{cur}}$ the error estimate EST and the error indicator err are evaluated resulting in $\text{err} \leq 1$. Then, the current communication point $t_{\text{cur}}^{\text{M}}$ is updated to $t_{\text{cur}}^{\text{M}} + h_{\text{cur}}$ and co-simulation proceeds with the next communication step and "optimal" communication step size hc_{opt}, input parameter `newStep` of `fmiDoStep(…)` is set to `fmiTrue`.
- Rejected (or "discarded") communication steps: All slaves perform the communication step but do not generate any simulation data for file output. If all slaves complete successfully the full communication step $t_{\text{cur}}^{\text{M}} \rightarrow t_{\text{cur}}^{\text{M}} + h_{\text{cur}}$ then the error estimate EST and the error indicator err are evaluated but the error indicator exceeds its critical value: $\text{err} > 1$. The communication step has to be repeated with the same current communication point $t_{\text{cur}}^{\text{M}}$ as before but reduced communication step size $h_{\text{cur}} := hc_{\text{opt}}$. The communication step has to be repeated as well if at least one slave fails to complete the communication step successfully. Again, the current communication point $t_{\text{cur}}^{\text{M}}$ is left unchanged and the communication step size h_{cur} is reduced appropriately.

A technically challenging problem in the design and implementation of error controlled master algorithms is caused by the fact that during a communication step $t_{\text{cur}}^{\text{M}} \rightarrow t_{\text{cur}}^{\text{M}} + h_{\text{cur}}$, i.e. during a call to `fmiDoStep(…)`, neither the master nor any slave know if the communication step will finally be accepted or not since this decision is based on the output of *all* slaves. The output of simulation data to file, updates of model parameters etc. have to be postponed until all slaves have completed the current call of `fmiDoStep(…)` and the error criterion err is evaluated. In a practical implementation, the file output of simulation data during the communication step may be redirected to a data buffer. If the communication step is accepted, the buffered data are written to file, otherwise the data buffer is cleared.

In nested co-simulation environments with nested communication step size control, the situation gets even more complicated since the output of simulation data has to be postponed until *all* nested master algorithms accept the (nested) communication steps. In FMI for Co-Simulation, the information that the *previous* communication step $t_{\text{cur}}^{\text{M}} \rightarrow t_{\text{cur}}^{\text{M}} + h_{\text{cur}}$ was accepted may be given to the slaves setting parameter `newStep` to `fmiTrue` in the next call to `fmiDoStep(…)`. I.e., if a slave is called by function `fmiDoStep(…)` with input argument `newStep` set to `fmiTrue`, then the *previous* call of this slave by function `fmiDoStep(…)` resulted in an accepted communication step and data buffers should be written to file, model parameters should be updated (if applicable) etc. before starting the computation of the current communication step. This implementation scheme is applicable as well at the end time t_{stop} performing a call of `fmiDoStep(…)` with $t_{\text{cur}}^{\text{M}} = t_{\text{stop}}$ and $h_{\text{cur}} = 0$ and `newStep` = `fmiTrue` before terminating the co-simulation.

The specific problem in nested co-simulation environments is the fact that an *accepted* communication step of the *inner* co-simulation environment may belong to a (larger) *rejected* communication step of the *outer* co-simulation environment. Currently, all practical experience with communication step size control

in co-simulation is restricted to master algorithms generating *non-decreasing* sequences $t_{\text{cur}}^{\text{M}}$. More sophisticated algorithms for nested master algorithms are still under development.

Glossary

This glossary is a subset of (*MODELISAR Glossary, 2009*) with some extensions specific to this document.

Term	Description
algorithm	A formal recipe for solving a specific type of problem.
application programming interface (API)	A set of functions, procedures, methods or classes together with type conventions/declarations (e.g., C-header files) that an operating system, library or service provides to support requests made by computer programs.
communication points	Time grid for data exchange between master and slaves in a co-simulation environment (also known as “sampling points” or “synchronization points”).
communication step size	Distance between two subsequent *communication points* (also known as “sampling rate” or “macro step size”).
co-simulation	Coupling (i.e., dynamic mutually exchange and utilization of intermediate results) of several *simulation programs* including their numerical solvers in order to simulate a system consisting of several subsystems.
co-simulation interface	The set of interfaces within the MODELISAR framework to perform a *co-simulation.*
co-simulation platform	Software, which obtains means for coupling several *simulation programs* for *co-simulation.*
functional mock-up environment (FMUE)	In the general scheme of a *simulation program* FMUE is the part, which is responsible for all control activities and computations of the simulation, including data exchange between coupled simulation programs. It does include neither a user interface nor a logic for a user interaction.
functional mock-up interface for co-simulation	One of the MODELISAR *functional mock-up interfaces.* It connects the *master solver* component with one or more *slave solvers*.
functional mock-up interface for model exchange	One of the MODELISAR *functional mock-up interfaces.* It consists of the *model description interface* and the *model execution interface*. It connects the *external model* component with the *solver* component.
functional mock-up trust center (FMTC)	As defined in the MODELISAR framework, FMTC describes a closed system providing *model* and *simulation* access to authenticated users and functional mock-up authorities through dedicated cryptographic interfaces.
functional mock-up unit (FMU)	A “model class” from which one or more “model instances” can be build for simulation. A FMU is stored in one zip-file as defined in section 4 consisting basically of one xml file (see section 3) that defines the model variables and a set of C-functions (see section 2), in source or binary form, to execute the model equations or the simulator slave. In case of tool exection, additionally, the original simulator is required to perform the co-simulation (compare section 3.5.2).
gateway	A link between two computer programs allowing them to share information and bypass certain protocols on a host computer.
integration algorithm	The numerical algorithm to solve differential equations.
integrator	A *software component*, which implements an *integration algorithm*.
interface	An abstraction of a *software component* that describes its behavior without dealing with the internal implementation. *Software components* communicate with each other via interfaces.

master/slave	A method of communication, where one device or process has unidirectional control over one or more other devices. Once a master/slave relationship between devices or processes is established, the direction of control is always from the master to the slaves. In some systems a master is elected from a group of eligible devices, with the other devices acting in the role of slaves.
model	A model is a mathematical or logical representation of a system of entities, phenomena, or processes. Basically a model is a simplified abstract view of the complex reality. It can be used to compute its expected behavior under specified conditions.
model description file	The model description file is an XML-file, which supplies a description of all properties of a *model* (e.g. input/output variables).
model description interface	An interface description to write or retrieve information from the *model description file*.
model execution interface [from model interface working group]	An interface description to access the equations of a dynamic system from an external program.
numerical solver	see *solver*
output points	Tool internal time grid for saving output data to file (in some tools also known as "*communication points*" – but this term is used in a different way in FMI for Co-Simulation, see above).
output step size	Distance between two subsequent *output points*.
parameter	A quantity within a *model*, which remains constant during *simulation*, but may be changed between simulations. Examples are a mass, stiffness, etc.
slave	see *master/slave*
simulation	Compute the behavior of one or several *models* under specified conditions. (see also *co-simulation*)
simulation model	see *model*
simulation program	Software to develop and/or solve simulation *models*. The software includes a *solver*, may include a user interface and methods for post processing (see also: *simulation tool*, *simulation environment*). Examples of simulation programs are: Amesim, Dymola, Simpack, SimulationX, Simulink.
simulation tool	see *simulation program*
simulator	A simulator can include one or more *simulation programs*, which solve a common simulation task.
solver	*Software component,* which includes algorithms to solve *model*s, e.g. *integration algorithms* and *event handling* methods.
user interface	The part of the simulation program that gives the user control over the simulation and allows watching results.