D5.6	**Implementation of multi-level scheduler for executing FMUs in parallel with suitable data transfer**
Access[1]:	**PU**
Type[2]:	**Report**
Version:	**1.0**
Due Dates[3]:	**M36**

Open Cyber-Physical System Model-Driven Certified Development

Executive summary[4]:

This report describes several master algorithms for co-simulation based on the Functional Mockup Interface (FMI) standard. In particular it presents an algorithm for parallel multi-rate real-time co-simulation which uses a channel based communication concept.

[1]Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP.
[2]Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report.
[3]Due month(s) according to FPP.
[4]It is mandatory to provide an executive summary for each deliverable.

Deliverable Contributors:

	Name	Organisation	Primary role in project	Main Author(s)[5]
Deliverable Leader	Bernhard Thiele	Linköping University (LiU)	FMI, Real-Time	X
Contributing Author(s)[6]	Adrian Pop	RISE SCISEast	WP5 Leader	
Internal Reviewer(s)[7]	Magnus Eek	SAAB	WP1 Leader	
	Lena Buffoni	LIU	WP4 Leader	
	Martin Sjölund	LIU	WP3 Leader	
	Robert Hällquist	SAAB		

Document History:

Version	Date	Reason for change	Status[8]
0.1	2018-11-23	Full draft	Draft
1.0	2018-12-03	Final version	Released

[5] Indicate Main Author(s) with an "X" in this column.
[6] Person(s) from contributing partners for the deliverable, expected contributing partners stated in project proposal.
[7] Typically person(s) with appropriate expertise to assess deliverable structure and quality.
[8] Status = "Draft", "In Review", "Released".

Contents

1 Introduction

The use of virtual prototyping methods in product development has become an indispensable tool to manage the complexity of competitive modern products and industrial processes. Modeling the dynamic behaviour of such products and processes often requires considering systems that are composed of physical subsystems (from different physical domains) together with computing and networking. These kinds of systems are nowadays often termed Cyber-Physical Systems (CPS). In general these products critically rely on software to provide their functionality. However, the development and validation of such software functions is tightly interleaved with the behavior of the interacting physical systems. Hence, it is no longer sufficient to develop self-contained software components without adequately accounting for more global system interactions.

The Modelica language, [Fri14], allows to integrate discrete-time dynamics (embedded control software) and continuous-time dynamics (process behaviour). However, a frequent problem in larger industrial projects is that although component level models are available, it is a big hurdle to integrate them in larger system simulations. This is because different development groups and disciplines, *e.g.*, electrical, mechanical, hydraulic, software, *etc.*, often use, and are required to do so, their own approach and tools for modeling and simulation.

To improve interoperability of behavioral models the MODELISAR project[9] (initiated by the Daimler AG) developed the Functional Mock-up Interface (FMI) as a standardized exchange format for behavioral models. Figure 1 illustrates the basic concept: Model components are exported as Functional Mock-up Units (FMUs) from their respective discipline specific tool, another simulator tool can import the FMUs and integrate them into a Functional Mock-up using a suitable master algorithm for coupling the individual units. In October 2014 an improved version (FMI v2.0) was released to the public [FMI14]. The FMI technology experienced a formidable market uptake and is nowadays supported by more than 100 tools (https://www.fmi-standard.org/), among them OpenModelica.

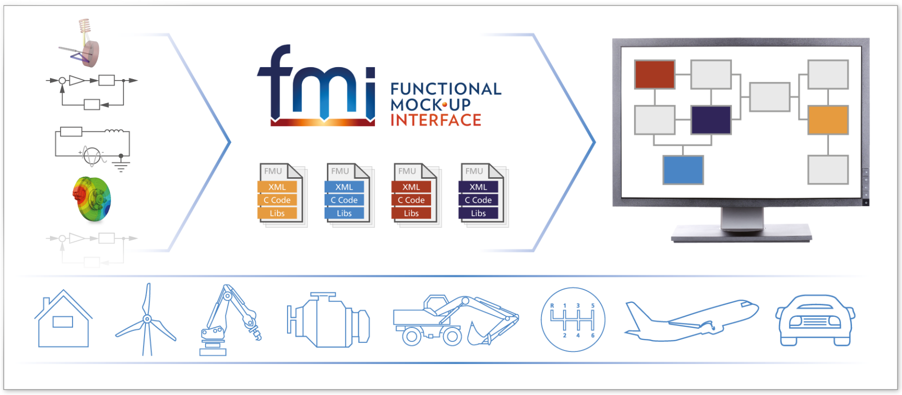


Figure 1: Model integration using FMI (source: https://www.fmi-standard.org/).

The motivation behind FMI is easily understood, however, coupling different simulator codes is a major challenge and an active research area. Modular simulation of a global system by

[9]MODELISAR consortium, *MODELISAR - From System Modeling to S/W running on the Vehicle*, ITEA 2 Office, Jul 2008 – Dec 2011, https://itea3.org/project/modelisar.html.

coupling different simulator codes may easily result in an unstable integration or may require proceeding in prohibitive small time steps [SA12]. Successful co-simulation needs

(a) a suitable module interface (this is what FMI standardizes) and

(b) a suitable master algorithm for coupling the modules (not standardized in FMI).

In previous work, Transmission Line Modeling (TLM) was integrated as one possible approach to co-simulation in OpenModelica [SFF06], and also considered as one approach to gain speed-up by simulation parallelization during the RTSIM project [SBFK10] (however, this was not based on FMI). OMSimulator now integrates both TLM and FMI into the same framework, enabling parallelization using TLM.

Difficulties arise if discrete-time models (*e.g.*, control software) are included within a co-simulation setup (*hybrid* co-simulation). The latest FMI v2.0 standard was shown to be not fully satisfactory for hybrid co-simulation. Different proposals to amend the found deficiencies are discussed, *e.g.*, [BBG+13, CLT+16, TCV+16, CLB+17].

Already at its early conception the FMI standard was also intended to be usable on real-time embedded systems [TH11]. Such FMUs typically have additional requirements (*e.g.*, memory efficiency, real-time constraints, cross-compilation) compared to standard FMUs, but they can share the same interface which allows a smoothly integrated development process for cyber-physical systems. Despite that, it appears the practical use of FMUs on embedded systems remained limited (with notable exceptions where the embedded target system is rather a PC than a severely constrained microcontroller, *e.g.*, [FWW+15]). This can be partly attributed to missing appropriate tooling support as well as on limitations in the standard with respect to typical requirements for more resource-constrained embedded systems, which has been investigated in [BNA+15]. Nevertheless, the interest in leveraging the standard for such applications remained active which is particularly evident by the recently started ITEA3 project EMPHYSIS[10] (budget: 14 M€) which major goal is to develop an extension to the FMI standard with the working name FMI for embedded systems (eFMI) which addresses the aforementioned needs.

There is not yet a generally accepted solution to the problem of hybrid co-simulation of FMUs needed for many industrial real-time applications, and suitable methods need to be devised. Furthermore, those algorithms need to be adapted to support real-time and distributed simulation scenarios. Moreover, compilation and simulation methods need to be improved using parallelization to generate code which fulfils tighter real-time deadlines. This report describes an approach for co-simulation interoperability of discrete-time (real-time) control applications and physical plant models based on the OpenModelica environment and the FMI standard. This approach can also be used for non real-time applications.

2 FMI for Co-Simulation

The FMI v2.0 standard defines two interfaces [FMI14, p. 4]:

(a) *FMI for Model Exchange (FMI-ME)*: The intention is that a modeling environment can generate C code of a dynamic system model that can be utilized by other modeling and

[10]EMPHYSIS consortium, *EMPHYSIS - Embedded systems with physical models in the production code software*, ITEA 3 Office, Sep 2017 – Aug 2020, https://itea3.org/project/emphysis.html.

simulation environments.

(b) *FMI for Co-Simulation (FMI-CS)*: The intention is to provide an interface standard for coupling of simulation tools in a co-simulation environment.

The two interfaces share common parts and concepts, in particular:

- FMI C-application programming interface (API): All computations are evaluated by calling standardized C-functions.
- FMI Extensible Markup Language (XML) description schema: The schema describes the structure and content of an XML file (named modelDescription.xml) generated by the modeling environment which exports an FMU. This modelDescription.xml file contains the definition of all variables and other structural information of an FMU in a standardized form.
- The FMU is delivered as one zip file which contains the XML description file, the code that provides the C-API either in binary form (shared library, Dynamic Link Library (DLL)) or as source code, as well as potential additional resources like tables, model icon, or documentation.

Basically, FMI-ME differs from FMI-CS in that it requires that the importing tool provides a *numerical solver* for simulating the FMU. These solvers require vectors for states, derivatives and zero-crossing functions which are exposed by the FMI-ME API. In contrast, FMI-CS does not require the importing tool to provide numerical solvers. Instead, if the FMU requires solvers, they are embedded within the FMI-CS and the related information is not exposed by the FMI-CS API.

This work is concerned with *co-simulation* aspects, thus the following discussion is based on FMI-CS rather than FMI-ME.

2.1 Overview

A schematic view of an FMU is shown in Figure 2.

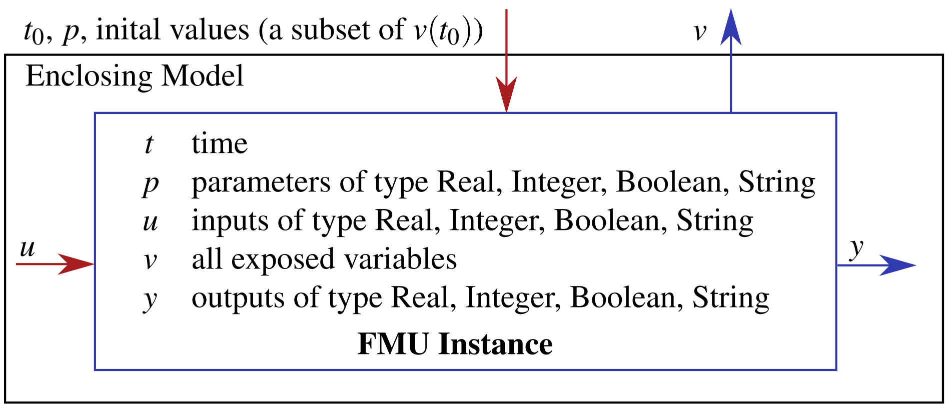

t_0, p, inital values (a subset of $v(t_0)$)
v
Enclosing Model
t time
p parameters of type Real, Integer, Boolean, String
u inputs of type Real, Integer, Boolean, String
v all exposed variables
y outputs of type Real, Integer, Boolean, String
FMU Instance
u
y

Figure 2: Data flow between the environment and an FMU. Red arrows: Values provided *to* the FMU; Blue arrows: Values provided *by* the FMU.

The FMI defines setter (`fmi2SetXXX(..)`) and getter (`fmi2GetXXX(..)`) functions for setting and getting values from an FMU where `XXX` denotes the data type and is either

Real, Integer, Boolean, or String. Initialization must be done within a dedicated initialization mode. After leaving the initialization mode, time is progressed within the FMU by calling the function fmi2DoStep(..) which has as one of its arguments the desired time step size.

For illustration consider the simple scenario in Figure 3 in which two FMUs are connected in a loop. In FMI-CS terminology these two FMUs are called *slaves*. Each FMU has one continuous real input and one continuous real output. Further, it is assumed that there is no algebraic dependency between input and output of each FMU. Note that OMSimulator can handle algebraic loops and will also use the dependencies inside the model description to detect false loops. However, the parallel execution algorithms do not support algebraic loops yet.

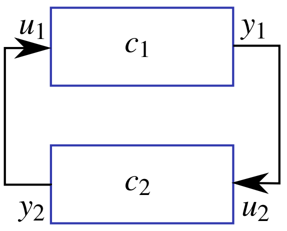


Figure 3: Connection graph of the FMUs.

The FMI specification provides a pseudo C-code example for the scenario from above (see [FMI14, Section 4.2.5]). The code demonstrates the most basic (simplest) master algorithm which after initialization simply periodically invokes the fmi2doStep(..) function using a constant communication step size (without allowing repetition of any communication steps). For convenience this pseudo C-code example is reproduced in Listing 1.

Listing 1: Pseudo C-code example for a basic master algorithm (source: FMI v2.0 specification).

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

//Set callback functions,
fmi2CallbackFunctions cbf;
cbf.logger = loggerFunction; //logger function
cbf.allocateMemory = calloc;
cbf.freeMemory = free;
cbf.stepFinished = NULL; //synchronous execution
cbf.componentEnvironment = NULL;

//Instantiate both slaves
fmi2Component c1 = c1_fmi2Instantiate("Tool1" , fmi2CoSimulation,
   GUID1, "", fmi2False, fmi2False, &cbf, fmi2True);
fmi2Component c2 = c2_fmi2Instantiate("Tool2" , fmi2CoSimulation,
   GUID2, "", fmi2False, fmi2False, &cbf, fmi2True);

if ((c1 == NULL) || (c2 == NULL)) return FAILURE;

```

```
// Start and stop time
startTime = 0;
stopTime = 10;

//communication step size
h = 0.01;

// set all variable start values (of "ScalarVariable / <type> /
    start")
c1_fmi2SetReal/Integer/Boolean/String(c1, ...);
c2_fmi2SetReal/Integer/Boolean/String(c2, ...);

//Initialize slaves
c1_fmi2SetupExperiment(c1, fmi2False, 0.0, startTime, fmi2True,
    stopTime);
c2_fmi2SetupExperiment(c1, fmi2False, 0.0, startTime, fmi2True,
    stopTime);
c1_fmi2EnterInitializationMode(c1);
c2_fmi2EnterInitializationMode(c2);
  // set the input values at time = startTime
  c1_fmi2SetReal/Integer/Boolean/String(c1, ...);
  c2_fmi2SetReal/Integer/Boolean/String(c2, ...);
c1_fmi2ExitInitializationMode(c1);
c2_fmi2ExitInitializationMode(c2);

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

tc = startTime; //Current master time

while ((tc < stopTime) && (status == fmi2OK)) {
  //retrieve outputs
  c1_fmi2GetReal(c1, ..., 1, &y1);
  c2_fmi2GetReal(c2, ..., 1, &y2);
  //set inputs
  c1_fmi2SetReal(c1, ..., 1, &y2);
  c2_fmi2SetReal(c2, ..., 1, &y1);

  //call slave c1 and check status
  status = c1_fmi2DoStep(c1, tc, h, fmi2True);
  switch (status) {
    case fmi2Discard:
      fmi2GetBooleanStatus(c1, fmi2Terminated, &boolVal);
    if (boolVal == fmi2True)
      printf("Slave c1 wants to terminate simulation.");
    case fmi2Error:
    case fmi2Fatal:
      terminateSimulation = true;
      break;
  }
  if (terminateSimulation)
```

```
      break;

    //call slave c2 and check status as above
    status = c2_fmi2DoStep(c2, tc, h, fmi2True);
    ...
    //increment master time
    tc += h;
  }
  //////////////////////////
  //Shutdown sub-phase
  if ((status != fmi2Error) && (status != fmi2Fatal)) {
    c1_fmi2Terminate(c1);
    c2_fmi2Terminate(c2);
  }

  if (status != fmi2Fatal) {
    c1_fmi2FreeInstance(c1);
    c2_fmi2FreeInstance(c2);
  }
```

Listing 1 should provide a rather intuitive grasp of how an FMI-CS based co-simulation works and thus provide enough background for moving to the formalization in the following section. Details about the used API functions and data structures can be found in the specification document.

2.2 Formalized Notation

The formalized notation in this section is based on the proposal by Broman et al. [BBG+13]. The formalization facilitates focusing on the principles of the FMI-CS by introducing a higher level of abstraction and a more succinct notation.

Table 1 defines a more succinct notation for FMI-CS. Notice that the **doStep**$_c$ function extends the FMI-CS standard by allowing communication step sizes of $h, h' = 0$, while the standard requires $h, h' > 0$. This extension is part of the proposal described in [BBG+13] and allows updating the state of FMU instances without advancing physical time, enabling a *superdense* model of time (which is needed for supporting consecutive chains of events at the same continuous-time instant t_i).

3 Deterministic Composition of Hybrid Co-Simulations

A rollback master algorithm for deterministic composition of hybrid co-simulations was first described by Broman et al. [BBG+13, Algorithm 2]. The advantage of that algorithm is that it is rather simple and only requires minimal extensions to FMI-CS v2.0. The required extension is to allow communication time steps of value zero ($h \geq 0$) for enabling a superdense time model in which a consecutive chain of events can be handled without progressing the simulation time. Furthermore, the rollback strategy requires all participating FMUs to support state serialization which is an *optional* feature in FMI v2.0 (capability flag `canGetAndSetFMUstate` must be true). One disadvantage of that algorithm is that the rollback mechanism is computationally

Table 1: Formalized notation for FMI-CS.

$\mathbb{V}$	Set of values that a variable may take on (ignoring typing issues).
C	Set of FMU instances in a model.
$c \in C$	FMU instance identifier.
S_c	Set of (all possible) state valuations for instance c. Further, let $s_c \in S_c$ denote the current state valuation for instance c.
U_c	Set of input port variables for instance c.
Y_c	Set of output port variables for instance c.
$D_c \subseteq U_c \times Y_c$	I/O dependencies for instance c. $(u, y) \in D_c$ means that output y of c is directly dependent on input u of c.
$s = \bigcup_{c \in C} s_c$	Current state of all FMU instances.
$U = \bigcup_{c \in C} U_c$	Set of all input variables in a model.
$Y = \bigcup_{c \in C} Y_c$	Set of all output variables in a model.
$D = \bigcup_{c \in C} D_c$	Set of all I/O dependencies.
$\mathbb{X} = U \cup Y$	Set of all input and output variables in the model.
c_x	Denotes the (unique) FMU instance $c \in C$ to which $x \in \mathbb{X}$ belongs.
$P : U \to Y$	Port mapping. Maps every input variable $u \in U$ to exactly one (unique) connected output variable $y \in Y$.
$\mathbf{set}_c : S_c \times U_c \times \mathbb{V} \to S_c$	Given $(s_c, u, v) \in (S_c, U, \mathbb{V})$ return updated state $s'_c \in S_c$. Formally, the updated state $s'_c = s[u := v]$ is identical to s_c, except that s'_c assigns value v to variable u. Corresponds to `fmi2SetXXX(..)`.
$\mathbf{get}_c : S_c \times Y_c \to \mathbb{V}$	Given $(s_c, y) \in (S_c, Y)$ return value $v \in \mathbb{V}$ of output variable y. Corresponds to `fmi2GetXXX(..)`.
$\mathbf{doStep}_c : S_c \times \mathbb{R}_{\geq 0} \to S_c \times \mathbb{R}_{\geq 0}$	Given $(s_c, h) \in (S_c, \mathbb{R}_{\geq 0})$, where h corresponds to the `communicationStepSize` argument of `fmi2DoStep(..)`, return $(s'_c, h') \in (S_c, \mathbb{R}_{\geq 0})$ and $0 \leq h' \leq h$. Notice that this is an *extension* to the FMI-CS standard, because the standard requires $h, h' > 0$. This formalization allows $h, h' \geq 0$, enabling a *superdense* time model.

expensive and exporting FMUs with the required rollback capability can be a difficult task (if even possible) for simulation tool vendors.

In order to mitigate the disadvantages of the basic rollback master algorithm, Broman et al. propose an extension [BBG+13, Algorithm 3], which supports a more efficient execution scheme and allows to integrate (at most) one “legacy FMU” (*i.e.*, an FMU without the extensions discussed above) into the co-simulation environment. However, this algorithm requires to extend the FMI-CS v2.0 API with an additional function to retrieve the maximum step size an FMU can accept, *before* doing the step. Follow-up research related to this publication proposes further improvements, *e.g.*, by arguing for a superdense model of time using integers [CLB+17].

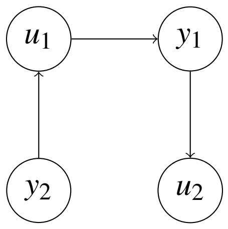
The problem of hybrid co-simulation based on FMI-CS v2.0 was also investigated by Tavella et al. [TCV+16], who proposed a different extension with a focus on computational performance and parallel execution of FMUs. Furthermore, there is ongoing work within in the FMI group discussing improved support of hybrid systems for future versions of the FMI-CS standard.

The following sections will briefly present a basic rollback master algorithm based on [BBG+13, Algorithm 2].

3.1 Scheduling the Access to Input and Output Ports

The port variable dependencies for the system of connected FMUs is given by the set of I/O dependencies D in combination with the port mapping P.

For illustration consider the simple connection graph from Figure 3 and assume that there is a direct dependency between u_1 and y_1, but not between u_2 and y_2, hence $D = \{(u_1, y_1)\}$ and $P = \{(u_1 \to y_2), (u_2 \to y_1)\}$. Figure 4 depicts the resulting dependency graph from which the valid schedule for accessing the ports is given by its topological ordering. That ordering can be directly read from the pictorial view of this graph as the sequence (y_2, u_1, y_1, u_2).



$$\begin{tikzcd} u_1 \arrow[r] & y_1 \arrow[d] \\ y_2 \arrow[u] & u_2 \end{tikzcd}$$

Figure 4: Dependency graph for Figure 3 assuming $D = \{(u_1, y_1)\}$.

Definition 1 (Port Dependency Graph)**.** The port dependency graph $G = (V, E)$ is a directed graph, where the vertices V are represented by the port variables $\mathbb{X}$ and the edges E are represented by (directed) variable dependencies $(v_1, v_2) \in \mathbb{X} \times \mathbb{X}$, meaning that v_2 is directly dependent on v_1. The set of all edges E is then constructed by $E = D \cup \{(y, u) | u \in U \land P(u) = y\}$.

A topological ordering of G is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). In this case Algorithm 1 returns a valid ordered sequence $\bar{x}$ in which the port variables $\mathbb{X}$ can be accessed (in general there exists more than one valid ordering).

Algorithm 1 Order-Variables

```
Require: Port dependency graph $G$ is a DAG
Ensure: Returns $\bar{x}$, which is an ordered sequence of valid variable accesses
1: $\bar{x}$ := TOPOLOGICAL-SORT($G$)    ▷ Algorithms are known for constructing a topological ordering of any DAG in linear time
```

As a final remark on the port dependency graph it should be noted that the graph used in the initialization phase may *differ* from the graph used during the simulation phase. The following discussion assumes a successful initialization phase and is only concerned with the simulation phase.

3.2 Rollback Master Algorithm

This section briefly presents the rollback master algorithm by Broman et al. [BBG+13, Algorithm 2].

The basic rollback master algorithm for deterministic composition of hybrid co-Simulations requires that the port dependency graph is acyclic (*i.e.*, no support of algebraic loops between FMU instances). In this case Algorithm 1 yields an ordered sequence of port variables $\bar{x}$, which is required as input to the MASTER-STEP function which is reproduced in Algorithm 2.

Algorithm 2 Master-Step

```
1: function MASTER-STEP($C, \bar{x}, P, h_{max}, s$)    ▷ $C$ is the set of FMU instances, $\bar{x}$ is an ordered sequence of valid variables accesses (from Algorithm 1), $P$ is the port mapping, $h_{max}$ the proposed time step, and $s$ the current state of the FMUs
2:     for all $u \in \bar{x}$ (in order) where $u \in U$ do    ▷ Set value for all input variables
3:         $y := P(u)$
4:         $v := \mathbf{get}_{c_y}(s_{c_y}, y)$
5:         $s_{c_u} := \mathbf{set}_{c_u}(s_{c_u}, u, y)$
6:     $r := s$    ▷ Save the states of all FMUs to enable rollback
7:     $h := h_{max}$    ▷ Set communication step size to an initial default value
8:     for all $c \in C$ do    ▷ Find $h$ acceptable by all FMUs
9:         $(s', h') := \mathbf{doStep}_c(s_c, h_{max})$
10:        $h := \min(h, h')$
11:        $s_c := s'$
12:    if $h < h_{max}$ then    ▷ roll back and perform step $h$
13:        for all $c \in C$ do
14:            $(s', h') := \mathbf{doStep}_c(r_c, h)$
15:            $s_c := s'$
16:    return $s, h$    ▷ $s$ is the updated state of the FMUs, $h$ is the actually achieved time step
```

Properties of Algorithm 2 like *determinism* or *progress* are explored in detail in [BBG+13].

4 Parallel simulation

There are many cases where simulation applications need to interact with their environment. Typical examples are Hardware-in-the-Loop (HIL), Human-in-the-Loop (HITL) or Software-in-the-Loop (SIL) simulators. In case of real-time this imposes real-time constraints on the simulation, meaning that the simulation always must meet its timing deadlines. Typically, inputs and outputs of a real-time simulation need to be processed at regular intervals. The time duration of such an interval is called the *simulation frame time*. The *worst case* computation time needs to be less than the simulation frame time. For FMI-CS the regular simulation frame can be straightforwardly mapped to executing the **doStep**(..) function using a constant communication step size. For non real-time applications these constraints are not necessary.

4.1 Overview

The goal of *parallel (real-time) simulation* is to leverage multiple CPU cores in order to improve simulation performance. Further performance gains may be achieved if the different FMUs are executed using different rates which fit their dynamic properties, hence, FMUs with slower dynamics at lower rates, and FMUs with faster dynamics at higher rates. This technique is termed multiframing or multi-rate integration. In previous work Thiele et al. have investigated multi-rate and multi-method simulation within the clocked synchronous modeling framework provided by the Modelica language [TOM14]. The following paragraph reproduces typical execution schemes.

For real-time simulation the sequence in which the computation of fast and slow frames are interspersed is important. Ledin [Led01] describes three typical execution schemes. The timing diagram in Figure 5 shows that execution schemes by means of an example where the ratio of the slow and the fast step size, k, is $k = 3$. The three schemes are described briefly below:

1. *Multiframing in a single task with no fast-frame real-time I/O*. The slow frame rate is treated as a "master" frame rate in which the slow frame is executed first, followed by a burst of the k fast frames. This scheme is only acceptable if the fast frames do not perform any real-time I/O.
2. *Multiframing in a single task with fast-frame real-time I/O*. The fast frames are executed at fixed intervals of h_f length. The computations needed in the slow frame are split into several subframes which are interspersed after the fast frame calculations. However, splitting the slow frame into several suitable subframes is rarely a simple thing to do. This is a serious drawback of this method.
3. *Multiframing in a multitasking environment with rate monotonic scheduling (RMS)*. In this case the scheduler will give CPU access to the task with the higher priority (computation of fast frames) and interrupt the lower priority task (computation of slow frames) until the higher priority task has finished its computations. During the times in which the higher priority task is idle, the CPU access is given back to the lower priority task to resume its computations. No (manual) splitting into subframes is needed which is a huge advantage compared to the previous method.

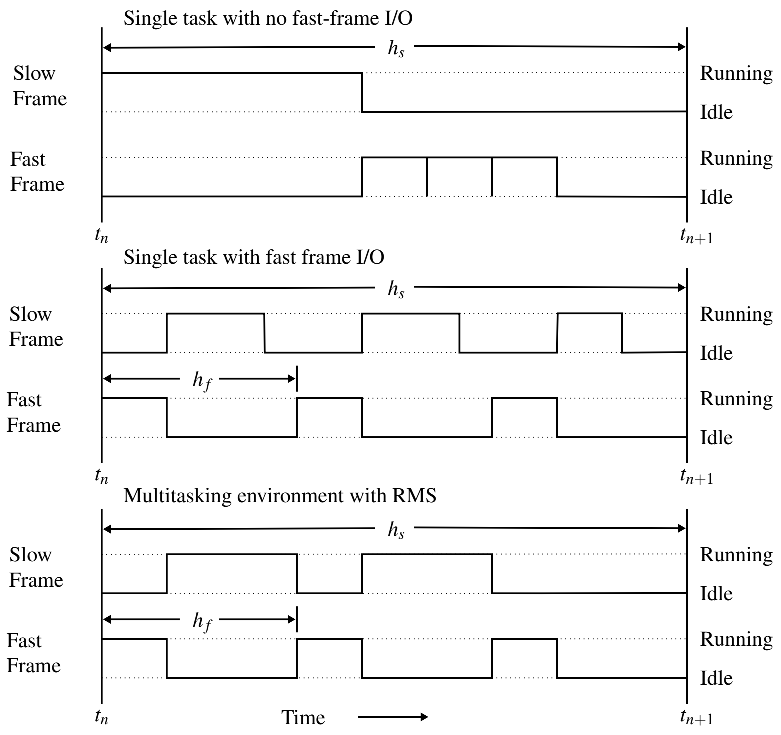


Figure 5: Three different multiframing schemes for real-time simulation.

From the basic multiframing schemes, the third one, based on RMS, will be investigated closer in the context of FMI-CS.

It may be noteworthy that more complex schemes for targeting real-time multi-core systems exist which are rather closely related to Modelica's clocked synchronous language extension. Adapting such a conceptual framework as advocated by Forget et al. [FBLP10] could allow for a very flexible and powerful composition of FMI-CS based real-time simulations. However, it would also add considerable complexities, possible requiring a dedicated architecture composition language. Therefore, such an extension is considered as out of the scope of this work.

4.2 Multi-Rate Master Algorithm

This section presents a single-thread multi-rate master algorithm. It introduces a concept of *communication channels* between input and output ports. Section 4.3 extends those algorithms to parallel versions. Table 2 extends the notation from Table 1 with notations for *clocks* and *channels*.

Table 2: Clock and communication channel notation (extends Table 1).

$h \in \mathbb{R}_{\geq 0}$	Time interval in seconds.
$h_c \in \mathbb{R}_{\geq 0}$	Time interval (communication step size) for FMU instance c in seconds.
$\mathbb{A}$	Set of (all possible) state valuations of a periodic clock. The periodicity of a clock is given by a time interval h. A clock starts (first clock tick) at the begin of the simulation.
$\mathbf{tick}_a : \mathbb{A} \to \mathbb{A}$	Progress clock $a \in \mathbb{A}$ one tick from the current time instance t_i to the next time instant $t_{i+1} = t_i + h$.
$\alpha_c \in \mathbb{A}$	Clock with interval h_c for FMU instance c.
$\alpha_{\mathbf{base}} \in \mathbb{A}$	The base clock of the model with interval $h_{\mathbf{base}} \in \mathbb{R}_{\geq 0}$. It is required that all clocks α_c in a model are integer multiples of its base clock (*i.e.*, $h_c = k \cdot h_{\mathbf{base}}, \quad k \in \mathbb{N}$).
$\alpha = \alpha_{\mathbf{base}} \cup \bigcup_{c \in C} \alpha_c$	Current state valuation of all clocks in a model.
$\_ = \_ : \mathbb{A} \times \mathbb{A} \to \mathbb{B}$	The comparison operator "=" checks if two clocks synchronize, *i.e.*, both clocks are active (tick) at this instant.
Q	Set of all possible channel states.
$q \in Q$	Current state of channels.
q_u	Denotes the current state of the channel between input $u \in U$ and output $y \in Y$ (notice that any input has exactly one corresponding output).
$M : Y \to Q$	Channel mapping. Maps every output $y \in Y$ to n connected communication channels q_{u_i}, $i = 1, 2, \cdots n$.
$\mathbf{write} : Q \times U \times \mathbb{V} \to Q$	Write value $v \in \mathbb{V}$ to channel q_u, $u \in U$.
$\mathbf{read} : Q \times U \to Q \times \mathbb{V}$	Reconstruct a value $\tilde{v} \in \mathbb{V}$ from channel q_u, $u \in U$ from a number of $m \geq 1$ values that are stored in the channel (using extrapolation or interpolation techniques).

A channel is an object which holds data and provides functions for updating the channel and retrieving information. In particular, it may hold $m \geq 1$ values inserted by the **write**(..) function and use extrapolation or interpolation techniques for *reconstructing* a value $\tilde{v}(t_i) \in \mathbb{V}$ at simulation time instant t_i as an approximation to the continuous-time simulation value $v(t_i) \in \mathbb{V}$

at that instant (the simulation value may not be available at the (real-time) communication instant when it is needed). Figure 6 illustrates that each channel q_{u_i} may hold several values and may use different reconstruction strategies (*e.g.*, q_{u_1} returns the mean of its stored values, q_{u_2} only holds one value which it returns, q_{u_3} stores two values but only returns the second value).

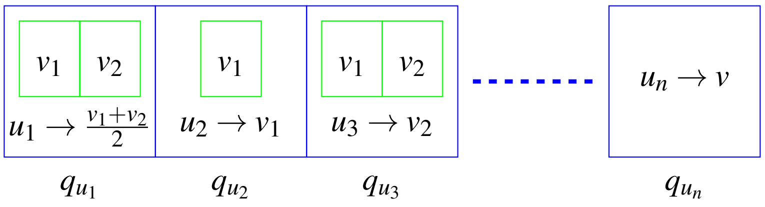


Figure 6: Illustration of the communication channel concept. Each channel q_{u_i} may hold several values and may use different reconstruction strategies.

Algorithm 3 uses these new concepts. The concrete semantics depend on the semantics given to the communication channels.

Algorithm 3 Multirate-Step

```
1: function MULTIRATE-STEP($C, \bar{x}, s, \alpha, q, M$)    ▷ $C$ is the set of FMU instances, $\bar{x}$ is an ordered sequence of valid variables accesses (from Algorithm 1), $s$ the current state of the FMUs, $\alpha$ the current state of clocks, $q$ the current channel state, $M$ is the channel mapping.
2:     for all $\{x \in \bar{x} \mid \alpha_{c_x} = \alpha_{\mathbf{base}}\}$ do
3:         if $x \in Y$ then
4:             $v := \mathbf{get}_{c_x}(s_{c_x}, x)$
5:             for all $q_u \in M(x)$ do
6:                 $q_u := \mathbf{write}(q_u, v)$    ▷ Write value for output $x$ to channel $q_u$
7:         else if $x \in U$ then
8:             $(q_x, v) := \mathbf{read}(q_x)$    ▷ Read value of output $y$ from channel
9:             $s_{c_x} := \mathbf{set}_{c_x}(s_{c_x}, x, v)$
10:    for all $\{c \in C \mid \alpha_c = \alpha_{\mathbf{base}}\}$ do
11:        $h_c = \mathbf{interval}(\alpha_c)$
12:        $(s_c, h'_c) := \mathbf{doStep}_c(s_c, h_c)$
13:        $\mathbf{assert}(h'_c = h_c)$
14:        $\alpha_c := \mathbf{tick}(\alpha_c)$
15:    $\alpha_{\mathbf{base}} := \mathbf{tick}(\alpha_{\mathbf{base}})$
16:    return $s, \alpha, q$    ▷ $s$ updated state of the FMUs, $\alpha$ updated clocks, $q$ updated channels
```

4.3 Parallel Multi-Rate Master Algorithm

In a next step Algorithm 3 and the communication channel concept from Section 4.2 will be extended for allowing parallel execution of the co-simulation.

First, an extension of the channel concept is needed.

Definition 2 (Channel)**.** A channel $Q_u = (r, k, n, \mathfrak{m}_p, \mathfrak{m}_c, v)$ connects an output $y \in Y$ with an input $u \in U$. The set of all possible channel states is denoted by Q. There are three rate transition cases r that are discriminated:

$$r = \begin{cases} \texttt{fastToSlow} & \text{The clock of output } y \text{ is } \textit{faster} \text{ than that of input } u, \\ \texttt{slowToFast} & \text{The clock of output } y \text{ is } \textit{slower} \text{ than that of input } u, \\ \texttt{same} & \text{The clocks of } y \text{ and } u \text{ run at the } \textit{same} \text{ rate.} \end{cases}$$

Since any input u is connected to exactly one output y a channel instance q_u can be uniquely identified by its connected input $u \in U$. The remaining elements in the tuple Q_u are:

$k \in \mathbb{N}$ is the activation ratio, defined as

$$k = \frac{\Delta\alpha_{\text{slow}}}{\Delta\alpha_{\text{fast}}}$$

where $\Delta\alpha_{\text{slow}}$ is the interval of the slow clock and $\Delta\alpha_{\text{fast}}$ is the interval of the fast clock. Only integer ratios are supported.

$n \in \mathbb{N}_0$ A counter with the property $n \leq k$.

$\mathfrak{m}_p \in \{\texttt{locked}, \texttt{unlocked}\}$ is a mutex supporting operations **lock**(..) (locks the mutex, blocks if the mutex is not available) and **unlock**(..) (unlocks the mutex).

$\mathfrak{m}_c \in \{\texttt{locked}, \texttt{unlocked}\}$ is a mutex, same as above.

$v \in \mathbb{V}^j$ storage place for j values. For the algorithms described below $j = 1$.

Two functions, **write**(..) and **read**(..), need to be defined for interacting with channels. They are defined in Algorithm 4.

Algorithm 5 sets up the necessary channels before spawning one task for each FMU instance in line 14. The scheduler can use a rate monotonic scheduling (RMS) algorithm with the task priorities being picked, so that tasks with faster clocks have a higher priority. For avoiding priority inversion problems the RMS algorithm should further have support for *priority inheritance*. Notice that only integer ratios $k \in \mathbb{N}$ between fast and slow clocked FMUs are supported

Finally, Algorithm 6 describes the periodic task which is executed for each FMU instance. Line 15 denotes the synchronization with the real-time clock.

4.4 Implementation

Three master algorithm variants of the communication channel based approach to parallel multi-rate co-simulation have been implemented as an experimental extension to the OMSimulator[11] tool. The OMSimulator is an FMI-based co-simulation environment developed by OpenCPS partners: LiU and SICSEast under the umbrella of the OpenModelica project and the Open Source Modelica Consortium (OSMC).

OMSimulator is internally implemented in C++ and exposes a pure C API which facilitates developing high-level interfaces to popular scripting languages. At present there are interfaces to the Lua and the Python scripting languages.

The master algorithm can be set by calls to the function `oms2_setMasterAlgorithm(..)`. The implemented channel based master algorithm variants are

[11] OMSimulator Github repository, https://github.com/OpenModelica/OMSimulator.

- `pmrchannelm`: This variant uses mutexes for synchronization. It is an implementation of the algorithm outlined in Section 4.3.
- `pmrchannela`: This variant use atomic variables for synchronization. The algorithm is described in Appendix A.
- `pmrchannelcv`: A variant which uses C++ condition variables for synchronization. The algorithm is not described in this document.

The implementation is still experimental and the API is likely to change.

First experiments have been made using a basic dual-mass oscillator example model which was split into two FMUs. For that example a significant speed-up (~1.5x, see table 3) could be observed for the `pmrchannela` variant, but not for the `pmrchannelm` and `pmrchannelcv` variant. It is expected that the observed speed-up will depend strongly on the ratio of the computational effort within an FMU and the communication/synchronization overhead introduced due to the parallelization. Therefore, suitable benchmark models need to be developed in future work in order to gain meaningful insight about potential performance gains for practically relevant use-cases.

Runs	standard (s)	pmrchannela (s)	pmrchannelcv (s)	pmrchannelm (s)	pctpl (s)
1	9.174	5.733	12.217	11.374	19.500
2	9.163	5.858	12.138	11.663	19.801
3	9.124	6.539	12.142	11.425	19.894

Table 3: Benchmark of parallel execution for test DualMassOscillatorEq_cs_oms2.lua, stopTime at 5s. Computer: Intel(R) Core(TM) i7-4702MQ CPU at 2.20GHz (4 cores), 16Gb RAM, Ubuntu 16.04 LTS.

5 Conclusion

This report describes several master algorithms for co-simulation based on the FMI standard. In particular it presents a channel based communication concept for realizing algorithms that support parallel and/or multi-rate co-simulation schemes. It further discusses constraints under which real-time co-simulation can be supported by the presented master algorithms. Several variants of the channel based master algorithm have been implemented in C++11 as an experimental feature of the OMSimulator tool. They differ in the technical solution which was used for synchronizing the communication: Mutexes, atomic variables, or condition variables. First experiments with a basic dual-mass oscillator example model which was split into two FMUs only showed performance speed-up for the variant using atomic variables. However, more thorough experiments with realistic real-world models need to be conducted in future work to gain meaningful insight about potential performance gains. This work will be part of the 4-months limited project extension.

Algorithm 4 Write and read

```
function WRITE($q, v$)                    ▷ $q \in Q_u$ is a channel instance, $v \in \mathbb{V}$ a value
    $q_{\mathfrak{m}_\mathfrak{p}} :=$ lock$(q_{\mathfrak{m}_\mathfrak{p}})$
    if $q_r =$ fastToSlow then
        $q_v := v$
        $q_n := q_n + 1$
        if $q_n = q_k$ then
            $q_{\mathfrak{m}_\mathfrak{c}} :=$ unlock$(q_{\mathfrak{m}_\mathfrak{c}})$
        else
            $q_{\mathfrak{m}_\mathfrak{p}} :=$ unlock$(q_{\mathfrak{m}_\mathfrak{p}})$
    else if $q_r =$ slowToFast then
        $q_v := v$
        $q_n := 0$
        $q_{\mathfrak{m}_\mathfrak{c}} :=$ unlock$(q_{\mathfrak{m}_\mathfrak{c}})$
    else
        $q_v := v$
        $q_{\mathfrak{m}_\mathfrak{c}} :=$ unlock$(q_{\mathfrak{m}_\mathfrak{c}})$
    return $q$
function READ($q$)                        ▷ $q \in Q_u$ is a channel instance
    $q_{\mathfrak{m}_\mathfrak{c}} :=$ lock$(q_{\mathfrak{m}_\mathfrak{c}})$
    if $q_r =$ fastToSlow then
        $v := q_v$
        $q_n := 0$
        $q_{\mathfrak{m}_\mathfrak{p}} :=$ unlock$(q_{\mathfrak{m}_\mathfrak{p}})$
    else if $q_r =$ slowToFast then
        $v := q_v$
        $q_n := q_n + 1$
        if $q_n = q_k$ then
            $q_{\mathfrak{m}_\mathfrak{p}} :=$ unlock$(q_{\mathfrak{m}_\mathfrak{p}})$
        else
            $q_{\mathfrak{m}_\mathfrak{c}} :=$ unlock$(q_{\mathfrak{m}_\mathfrak{c}})$
    else
        $v := q_v$
        $q_{\mathfrak{m}_\mathfrak{p}} :=$ unlock$(q_{\mathfrak{m}_\mathfrak{p}})$
    return $q, v$
```

Algorithm 5 Spawn-Tasks

```
Require: Global model initialization done.
Require: All FMU instances have a task priority $s_{c_p}$ for RMS assigned, faster clocks have a
    higher priority
 1: function SPAWN-TASKS($C, \bar{x}, P, s, \alpha, q$)
 2:     for all $u \in \bar{x}$ (in order) where $u \in U$ do
 3:         $y := P(u)$
 4:         if $\frac{\Delta\alpha_y}{\Delta\alpha_u} < 1$ then
 5:             $r :=$ fastToSlow
 6:             $k := \frac{\Delta\alpha_u}{\Delta\alpha_y}$
 7:         else if $\frac{\Delta\alpha_y}{\Delta\alpha_u} > 1$ then
 8:             $r :=$ slowToFast
 9:             $k := \frac{\Delta\alpha_y}{\Delta\alpha_u}$
10:         else
11:             $r :=$ same
12:             $k := 1$
13:         assert($k \in \mathbb{N}$)
14:         $q_u = (r, k, 0, \texttt{unlocked}, \texttt{locked}, u_{\text{initial}})$  ▷ $u_{\text{initial}}$ is the value from the initialization
15:     for all $c \in C$ do
16:         spawn(TASK-INSTANCE$_{s_{cp}}(c, \bar{x}, P, s_c, \alpha_c, q, M)$)▷ Spawn task with RMS priority $s_{c_p}$
    for real-time task scheduling
```

Algorithm 6 Task-Instance

```
 1: function TASK-INSTANCE($c, \bar{x}, P, s_c, \alpha_c, q, M$)                                ▷
    $c$ is the FMU instance identifier, $\bar{x}$ is an ordered sequence of valid variables accesses (from
    Algorithm 1), $P$ is the port mapping, $s_c$ the current state of the FMU, $\alpha_c$ the current state
    of the clock, $q$ the current state of communication buffers, $M$ is the buffer mapping.
 2:     loop
 3:         for all $\{x \in \bar{x} \mid x \in \mathbb{X}_c\}$ do
 4:             if $x \in Y$ then
 5:                 $v :=$ get$_{c_x}(s_{c_x}, x)$
 6:                 for all $q_u \in M(x)$ do
 7:                     $q_u :=$ WRITE$(q_u, v)$                        ▷ Potentially blocking
 8:             else if $x \in U$ then
 9:                 $(q_x, v) :=$ READ$(q_x)$                           ▷ Potentially blocking
10:                 $s_{c_x} :=$ set$_{c_x}(s_{c_x}, x, v)$
11:         $h_c =$ interval$(\alpha_c)$
12:         $(s_c, h'_c) :=$ doStep$_c(s_c, h_c)$
13:         assert$(h'_c = h_c)$
14:         $\alpha_c :=$ tick$(\alpha_c)$                              ▷ Progress clock for instance $c$
15:         wait$(\alpha_c = rtclk)$           ▷ Synchronize clock for instance $c$ with real-time
```

A Parallel Multi-Rate Master Algorithm using Atomic Operations

This sections presents an alternative multi-rate master algorithm which uses atomic operations instead of the mutex based approach from Section 4.3. First performance tests comparing the mutex based implementation with the atomic operations based implementation suggest a significantly better performance for the atomic operations based approach. However, the mutex based approach allows to avoid priority inversion problems by using a standard RMS algorithm with priority inheritance support which is important in a multi-rate real-time context (the “fast frames” need to be executed with a higher priority as the “slow frames” and if a “slow” task blocks a resource needed by a “fast” task, the priority of the “slow” task needs to be raised, see also Section 4.1). Hence, the decision to present the mutex based algorithm more prominently in the main part of the document.

In case all FMUs are executed with the same rate the priority inversion problem does not exist and the performance of the atomic operations based algorithm is likely to be better. The channel definition from Definition 2 (page 17) needs to be slightly adapted.

Definition 3 (AtomicOpsChannel)**.** A channel $Q_u = (r, k, n, \mathtt{p}, v)$ connects an output $y \in Y$ with an input $u \in U$. The set of all possible channel states is denoted by Q. There are three rate transition cases r that are discriminated:

$$r = \begin{cases} \mathtt{fastToSlow} & \text{The clock of output } y \text{ is } \textit{faster} \text{ than that of input } u, \\ \mathtt{slowToFast} & \text{The clock of output } y \text{ is } \textit{slower} \text{ than that of input } u, \\ \mathtt{same} & \text{The clocks of } y \text{ and } u \text{ run at the } \textit{same} \text{ rate.} \end{cases}$$

Since any input u is connected to exactly one output y a channel instance q_u can be uniquely identified by its connected input $u \in U$. The remaining elements in the tuple Q_u are:

$k \in \mathbb{N}$ is the activation ratio, defined as

$$k = \frac{\Delta\alpha_{\text{slow}}}{\Delta\alpha_{\text{fast}}}$$

where $\Delta\alpha_{\text{slow}}$ is the interval of the slow clock and $\Delta\alpha_{\text{fast}}$ is the interval of the fast clock. Only integer ratios are supported.

$n \in \mathbb{N}_0$ A counter with the property $n \leq k$.

$\mathtt{p} \in \mathbb{B}$ is a flag ($\mathbb{B} = \{\mathtt{false}, \mathtt{true}\}$). It is an *atomic* object, meaning that if one task writes to it while another task reads from it, the behavior is well-defined.

$v \in \mathbb{V}^j$ storage place for j values. For the algorithms described below $j = 1$.

Two functions, **write**(..) and **read**(..), need to be defined for interacting with channels. These functions provide semantic variation points, since different definitions of these functions allow different communication semantics.

Algorithm 7 provides basic definitions for the functions **write**(..) and **read**(..) in which $v \in \mathbb{V}$. These definitions are sufficient for the parallel multi-rate master algorithm developed in this section.

Algorithm 8 sets up the necessary channels before spawning one task for each FMU instance in line 14. Notice that the third argument “$k-1$” is only relevant if $r = \mathtt{fastToSlow}$, where it

Algorithm 7 Write and read (atomic operations variant)

```
1:  function WRITE($(q, v)$)                 ▷ $q \in Q_u$ is a channel instance, $v \in \mathbb{V}$ a value
2:      while $q_\mathfrak{p}$ do                          ▷ Busy-waiting while $q_\mathfrak{p} = \texttt{true}$
3:          yield()                    ▷ Hint to the scheduler to allow other threads to run
4:      if $q_r = \texttt{fastToSlow}$ then
5:          $q_v := v$
6:          $q_n := q_n + 1$
7:          $q_\mathfrak{p} := q_n = q_k$
8:      else if $q_r = \texttt{slowToFast}$ then
9:          $q_v := v$
10:         $q_n := 0$
11:         $q_\mathfrak{p} := \texttt{true}$
12:     else
13:         $q_v := v$
14:         $q_\mathfrak{p} := \texttt{true}$
15:     return $q$
16: function READ($(q)$)                                 ▷ $q \in Q_u$ is channel instance
17:     while $\neg q_\mathfrak{p}$ do                        ▷ Busy-waiting while $q_\mathfrak{p} = \texttt{false}$
18:         yield()                    ▷ Hint to the scheduler to allow other threads to run
19:     if $q_r = \texttt{fastToSlow}$ then
20:         $v := q_v$
21:         $q_n := 0$
22:         $q_\mathfrak{p} := \texttt{false}$
23:     else if $q_r = \texttt{slowToFast}$ then
24:         $v := q_v$
25:         $q_n := q_n + 1$
26:         $q_\mathfrak{p} := \neg(q_n = q_k)$
27:     else
28:         $v := q_v$
29:         $q_\mathfrak{p} := \texttt{false}$
30:     return $q, v$
```

ensures that flag $\mathfrak{p} = \texttt{true}$ after the first write. Notice that only integer ratios $k \in \mathbb{N}$ between fast and slow clocked FMUs are supported.

Finally, Algorithm 9 describes the periodic task which is executed for each FMU instance. Line 15 denotes the synchronization with the real-time clock.

Algorithm 8 Spawn-Tasks (atomic operations variant)

```
Require: Global model initialization done.
 1: function SPAWN-TASKS($C, \bar{x}, P, s, \alpha, q$)
 2:     for all $u \in \bar{x}$ (in order) where $u \in U$ do
 3:         $y := P(u)$
 4:         if $\frac{\Delta\alpha_y}{\Delta\alpha_u} < 1$ then
 5:             $r :=$ fastToSlow
 6:             $k := \frac{\Delta\alpha_u}{\Delta\alpha_y}$
 7:         else if $\frac{\Delta\alpha_y}{\Delta\alpha_u} > 1$ then
 8:             $r :=$ slowToFast
 9:             $k := \frac{\Delta\alpha_y}{\Delta\alpha_u}$
10:         else
11:             $r :=$ same
12:             $k := 1$
13:         assert($k \in \mathbb{N}$)
14:         $q_u = (r, k, k-1, \texttt{false}, u_{\text{initial}})$        ▷ $u_{\text{initial}}$ is the value from the initialization
15:     for all $c \in C$ do
16:         spawn(TASK-INSTANCE$_{s_{c_p}}(c, \bar{x}, P, s_c, \alpha_c, q, M)$)▷ Spawn task with RMS priority $s_{c_p}$
    for real-time task scheduling
```

Algorithm 9 Parallel-Instance (atomic operations variant)

```
 1: function PARALLEL-INSTANCE($c, \bar{x}, P, s_c, \alpha_c, q, M$)        ▷ $c$ FMU
    instance identifier, $\bar{x}$ is an ordered sequence of valid variables accesses (from Algorithm 1),
    $P$ is the port mapping, $s_c$ the current state of the FMU, $\alpha_c$ the current state of the clock, $q$
    the current state of communication buffers, $M$ is the buffer mapping.
 2:     loop
 3:         for all $\{x \in \bar{x} \mid x \in \mathbb{X}_c\}$ do
 4:             if $x \in Y$ then
 5:                 $v :=$ get$_{c_x}(s_{c_x}, x)$
 6:                 for all $q_u \in M(x)$ do
 7:                     $q_u :=$ write$(q_u, v)$        ▷ Potentially blocking
 8:             else if $x \in U$ then
 9:                 $(q_x, v) :=$ read$(q_x)$        ▷ Potentially blocking
10:                 $s_{c_x} :=$ set$_{c_x}(s_{c_x}, x, v)$
11:         $h_c =$ interval$(\alpha_c)$
12:         $(s_c, h'_c) :=$ doStep$_c(s_c, h_c)$
13:         assert$(h'_c = h_c)$
14:         $\alpha_c :=$ tick$(\alpha_c)$        ▷ Progress clock for instance $c$
15:         wait$(\alpha_c = rtclk)$        ▷ Synchronize clock for instance $c$ with real-time
```

References

[BBG+13] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of fmus for co-simulation. Technical Report UCB/EECS-2013-153, EECS Department, University of California, Berkeley, Aug 2013. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-153.html.

[BNA+15] Christian Bertsch, Jonathan Neudorfer, Elmar Ahle, Siva Sankar Arumugham, Karthikeyan Ramachandran, and Andreas Thuy. FMI for physical models on automotive embedded targets. In Peter Fritzson and Hilding Elmqvist, editors, 11th *Int. Modelica Conference*, Versailles, France, September 2015. doi:10.3384/ecp1511843.

[CLB+17] Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. Hybrid co-simulation: it's about time. *Software & Systems Modeling*, November 2017. doi:10.1007/s10270-017-0633-6.

[CLT+16] Fabio Cremona, Marten Lohstroh, Stavros Tripakis, Christopher Brooks, and Edward A. Lee. Fide: An fmi integrated development environment. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1759–1766, New York, NY, USA, 2016. ACM. doi:10.1145/2851613.2851677.

[FBLP10] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A real-time architecture design language for multi-rate embedded control systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 527–534, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1774088.1774196, doi:10.1145/1774088.1774196.

[FMI14] FMI development group. Functional Mock-up Interface for Model Exchange and Co-Simulation v2.0. Modelica Association Project "FMI", October 2014. Standard Specification. URL: https://www.fmi-standard.org/.

[Fri14] Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley IEEE Press, 2014.

[FWW+15] Rüdiger Franke, Marcus Walther, Niklas Worschech, Willi Braun, and Bernhard Bachmann. Model-based control with FMI and a C++ runtime for Modelica. In Peter Fritzson and Hilding Elmqvist, editors, 11th *Int. Modelica Conference*, Versailles, France, September 2015. doi:10.3384/ecp15118339.

[Led01] Jim Ledin. *Simulation Engineering*. CMP Books, Lawrence, Kansas 66046, first edition, 2001.

[SA12] Tom Schierz and Martin Arnold. Stabilized overlapping modular time integration of coupled differential-algebraic equations. *Applied Numerical Mathematics*, 62(10):1491 – 1502, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0168927412001067, doi:10.1016/j.apnum.2012.06.020.

[SBFK10] Martin Sjölund, Robert Braun, Peter Fritzson, and Petter Krus. Towards efficient distributed simulation in modelica using transmission line modeling. In *3rd In-*

ternational Workshop on Equation-Based Object-Oriented Languages and Tools., Oslo, Norway, October 2010. URL: http://www.ep.liu.se/ecp/047/.

[SFF06] Alexander Siemers, Dag Fritzson, and Peter Fritzson. Meta-Modeling for Multi-Physics Co-Simulation applied for OpenModelica. In *International Congress on Methodologies for Emerging Technologies in Automation (ANIPLA2006)*, Rome, Italy, November 13–15 2006.

[TCV$^+$16] J. P. Tavella, M. Caujolle, S. Vialle, C. Dad, C. Tan, G. Plessis, M. Schumann, A. Cuccuru, and S. Revol. Toward an accurate and fast hybrid multi-simulation with the fmi-cs standard. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–5, September 2016. doi:10.1109/ETFA.2016.7733616.

[TH11] Bernhard Thiele and Dan Henriksson. Using the Functional Mockup Interface as an Intermediate Format in AUTOSAR Software Component Development. In Christoph Clauß, editor, 8th *Int. Modelica Conference*, Dresden, Germany, March 2011. doi:10.3384/ecp11063484.

[TOM14] Bernhard Thiele, Martin Otter, and Sven Erik Mattsson. Modular Multi-Rate and Multi-Method Real-Time Simulation. In Hubertus Tummescheit and Karl-Erik Årzén, editors, 10th *Int. Modelica Conference*, Lund, Sweden, March 2014. doi:10.3384/ECP14096381.