Generic FMI-compliant Simulation Tool Coupling

Edmund Widl[1] Wolfgang Müller[2]

[1]Center for Energy, AIT Austrian Institute of Technology, Austria, edmund.widl@ait.ac.at
[2]Institute of Analysis and Scientific Computing, TU Wien, Austria, wolfgang.mueller@student.tuwien.ac.at

Abstract

The Functional Mock-up Interface (FMI) specification provides a simple yet effective definition for co-simulation APIs. Even though the number of simulation tools supporting the export of Functional Mock-up Units (FMU) is growing steadily, there is a considerable number of well-established tools that do not. This paper addresses this issue by introducing a generic and adaptable way of coupling established simulation tools in an FMI-compliant manner. The proposed concept has been implemented as part of the FMI++ library, which is used as basis for FMI-compliant wrappers for the TRNSYS simulation tool and the MATLAB environment. These examples demonstrate the potential of the proposed approach to include well-established simulation tools with minimal effort. This not only enables researchers and engineers to include a diverse range of tools more easily into their work flow, but is also an incentive for tool developers to provide FMI-compliant wrappers.

Keywords: FMI for Co-Simulation, tool coupling, front-end/back-end concept, TRNSYS, MATLAB

1 Introduction

The list of simulation tools offering FMI (Blochwitz et al., 2011) support is rapidly growing[1], demonstrating the feasibility of the approach and underlining the importance of such a specification for Co-Simulation (CS) and Model Exchange (ME). However, many established simulation tools do not yet offer APIs for co-simulation, let alone one that follows the FMI specifications. This paper explores a generic approach that facilitates the integration of FMU CS export functionalities for such tools.

The proposed approach uses a front-end that is exposed to the master algorithm as FMI component, and an appropriately linked back-end that is used by the slave application. Due to its design this approach can not only be used by tool developers who have access to the (possibly closed) source code of the core application but also by users who have only limited access to or knowledge of the underlying application layers. The only requirement is the possibility for users to provide custom objects based on C/C++ code (or languages with adequate bindings to C/C++) that can be embedded within and exchange data with the simulation environment.

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

2 The FMI-compliant front-end/back-end concept

The basic concept comprises two components: The *front-end* component to be used by the simulation master and the *back-end* component to be used by the slave application. Between these two components a proper *data management* has to be established that is responsible for the communication and data exchange between both ends. The corresponding interfaces are tailored to suit the requirements of the FMI specification. They implement the necessary functionality required for a master-slave concept, i.e., synchronization mechanisms and exchange of data. See Figure 1 for a schematic view of this concept.

2.1 Front-end component

The front-end component is the actual gateway for a master algorithm to communicate and exchange data with an external simulation application. Its interface (see Figure 2) is designed such that it can be easily used as an FMI component (FMI model type `fmiComponent`), implementing functionalities close to the requirements of the FMI specification, for instance functions `initializeSlave(...)`, `doStep(...)` or `setReal(...)`. The front-end is responsible for the following tasks:

2.1.1 Information retrieval

The front-end component parses the FMI model description and stores the relevant information. This includes general simulator attributes (e.g., executable name, event handling capabilities) as well as specific model information (e.g., simulator-specific input files, variable names and types, input/output relations).

2.1.2 Variable initialization

Once the model description information is retrieved, the memory for the variables has to be allocated. This is done with the help of the dedicated data management (see Section 2.3 below).

2.1.3 Variable handling

The front-end has to manage the mapping between the model specific variable names and the associated value references according to the FMI specification. The latter are used to refer to and access the variables through the FMI API. In addition, the front-end has to ensure during runtime that variables are accessed properly, e.g., prohibiting write requests for output variables.

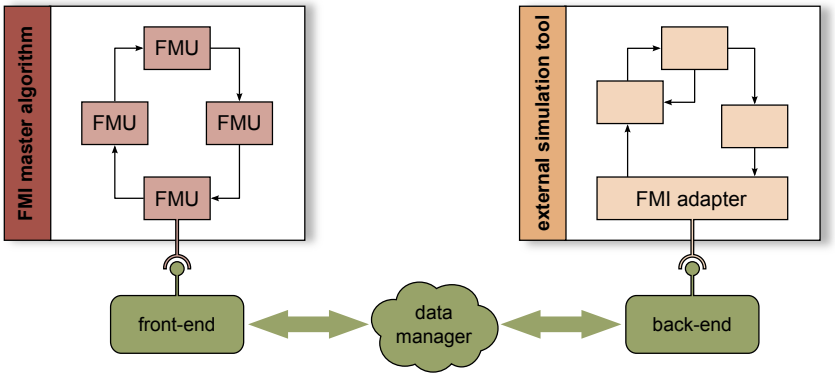

Figure 1. Schematics of the FMU CS export using the front-end/back-end concept: A simulation tool couples via an internal component to the back-end. The co-simulation master algorithm uses an instance of the front-end as FMI component. Synchronization and data exchange between the two ends is handled via a dedicated data manager.

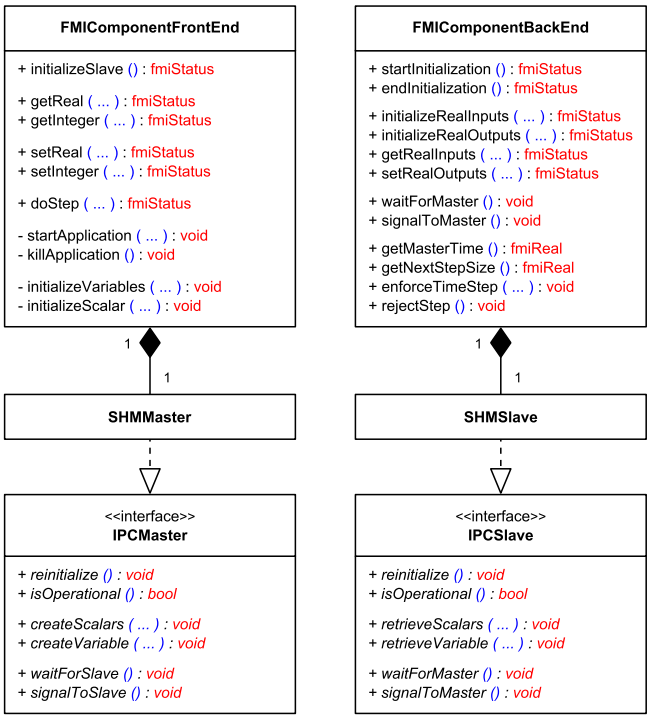

Figure 2. UML diagram of the most important features of the front-end and back-end components and the classes responsible for their data management (via shared memory in this specific case). The function arguments are not shown due to space constraints.

2.1.4 Application handling

The front-end is responsible for starting the external simulation application. It also has to establish a synchronized communication and data exchange, which is again done with the help of the dedicated data management (see Section 2.3 below)

2.2 Back-end component and FMI adapter

The back-end component functions as counterpart to the front-end component and is intended to be incorporated within the slave application as part of a dedicated simulation component, referred to as the *FMI adapter* (see Figure 1). The back-end interface is designed to make the connection with the front-end as simple as possible, focusing on synchronization and data exchange (see Figure 2). The adapter has to carry out the following tasks with the help of the back-end:

2.2.1 Information retrieval

The adapter has to be a part of the model that is loaded in the external simulator. As such is has to be able to retrieve and store information about the model it is embedded in at run-time, most importantly the names and types of the inputs and outputs that should be shared within the co-simulation.

2.2.2 Establishing the data exchange

Once the names and types of all inputs and outputs are known, the adapter has to connect to the front-end and establish the synchronized data exchange. This is done with the help of the back-end component, which retrieves pointers to automatically synchronized variables via the dedicated data management (see Section 2.3 below).

2.2.3 Data exchange during simulation

The adapter has to be designed such that it knows at which points of the simulation is has to send/receive data to/from the front-end. Using the previously retrieved pointers it can read/write data with the help of the back-end component.

2.3 Data management

The *data manager* is the crucial link between the front-end and the back-end and handles all issues regarding Inter-Process Communication (IPC). It is split in two instances (see Figure 2), implementing the purely abstract interface definitions provided by `IPCMaster` and `IPCSlave`, which are intended to be used by the front-end and the back-end, respectively.

2.3.1 Data handling

Both interfaces are primarily designed for handling FMI scalar variables (XML type `fmiScalarVariable`),

DOI 10.3384/ecp17132321

Listing 1. Implementation of function `fmiDoStep(...)` according to the FMI 1.0 specification.

```
1  fmiStatus fmiDoStep( fmiComponent c, fmiReal currentCommunicationPoint,
2                       fmiReal communicationStepSize, fmiBoolean newStep )
3  {
4    FMIComponentFrontEnd* fe = static_cast<FMIComponentFrontEnd*>( c );
5
6    return fe->doStep( currentCommunicationPoint, communicationStepSize, newStep );
7  }
```

i.e., variables that are associated not only to a value represented by a basic data type (e.g., `fmiReal`) but also to model-related attributes (e.g., name, value reference or causality). The corresponding functionality is provided via `createScalars(...)` and `retrieveScalars(...)`.

In addition, the data manager allows to handle and access data with functions `createVariable(...)` and `retrieveVariable(...)` for internal communication between both ends (e.g., size of next time step, boolean flag for rejecting the next step).

2.3.2 Synchronization

Since the data exchange between both ends has to be synchronized, the data manager is not only responsible for allocating memory. It also has to have a way to control the access to the data, in order to prevent non-deterministic behavior.

This is realized via the functions `waitForSlave()` and `signalToSlave()` for the front-end and the functions `waitForMaster()` and `signalToMaster()` for the back-end. In both cases, variables that were instantiated via the data manager should not be read or written unless the blocking functions `waitForSlave()` or `waitForMaster()` return. Likewise, once a component is done reading or writing data, it is required to signal this via `signalToSlave()` or `signalToMaster()`, respectively, and wait again.

2.3.3 Flexibility

The abstract interfaces `IPCMaster` and `IPCSlave` have been designed such that the actual data transfer and synchronization can be achieved in various ways. For instance, shared memory access or communication via local or network sockets is feasible. In principle, this mechanism could even be used to build web applications.

3 Implementation

The above concept has been implemented for the FMI CS specification version 1.0 and version 2.0 as part of the FMI++ library[2]. The FMI++ library is an open-source software toolbox written in C++ that provides high-level functionality for handling FMUs. As such it intends to bridge the gap between the basic FMI specification and typical requirements of simulation tools. While some of the functionality offered by the FMI++ library for importing FMUs is comparable to what it is available in other software libraries (such as the FMU SDK[3] or the FMI Library[4]), the implementation of the concept for generic tool coupling as explained above is unique.

A data manager has been implemented that uses shared memory access to share data, including semaphores for the synchronization of both ends, relying on features provided by the Boost[5] library collection. In this case, both ends of the data management can physically access the same data. If the co-simulation master and the external application were executed on different machines (distributed simulation environment) both ends would have to allocate their own memory and keep their contents synchronized, e.g., by means of the Message Passing Interface (MPI Forum, 2009).

The implementation of the front-end, the back-end and the data management are generic, i.e., it is independent of the external application. FMI adapter implementations obviously depend strongly on the designated application, even though reasonably sophisticated simulation environments should offer the possibility to design it model-independent. Bindings for the generic back-end implementation to FMI adapters in other languages than C/C++ can be automatedly created with the help of the SWIG tool (Beazley, 2003).

In addition, a thin layer implementing all functions according to the FMI specification is needed, which calls the corresponding front-end component functions, see lines 4 and 5 of the code snippet in Listing 1 for an example. Since version 1.0 of the FMI specification defines the model name (FMI model description attribute `modelIdentifier`) as a prefix to all functions in the final shared library, this thin layer has to be recompiled for each individual exported model. However, this does not require any changes in the source code, as the actual functionality remains unaltered.

4 Examples

The concept explained above is very flexible and can be used within a broad context of applications. For developing an FMI adapter, only three requirements need to be satisfied by any tool:

[2] Available at `http://fmipp.sourceforge.net/`.

[3] Available at `http://www.qtronic.de/en/fmusdk.html`.

[4] Available at `http://www.fmi-library.org/`.

[5] Available at `http://www.boost.org/`.

- *Modularity:* The targeted tool has to offer a mechanism for including user-defined code (including the possibility to access memory), in order to define an FMI adapter.
- *Execution control:* User-defined code has to be able to impact the tool's execution. This can be achieved either actively (e.g., by accessing methods that directly control the execution) or passively (e.g., by halting the execution through a blocking function).
- *Language compatibility:* The language of the user-defined code has to be compatible or have bindings to an existing front-end/back-end implementation.

In the following, this is demonstrated for two distinct tools.

4.1 TRNSYS FMI adapter

4.1.1 Implementation

TRNSYS (Klein et al., 1976) is a popular and well established thermal building and system simulation environment that comes with a validated components library. It uses instances of so-called *types* to model the individual components of a building or a system. Unfortunately, it does not provide an API that allows to use it as a slave application within a co-simulation. However, TRNSYS fulfills all the prerequisites to use the above discussed concept to implement an FMI adapter that overcomes this limitation:

- *Modularity:* In addition to providing a rich library of validated types, TRNSYS also offers the possibility to include user-defined types. Since TRNSYS is based on Fortran, these types are not object-oriented components in a strict sense but follow a sufficiently similar design pattern based on specialized function calls.[6]
- *Execution control:* The overall simulation execution is steered by the TRNSYS core, which calls the individual instances of the types included within a model. During these calls the instances are told at which stage the simulation currently is, especially whether it is the initialization phase, a standard call during a time step or the last call of a time step. This information can be used by TRNSYS types to take actions accordingly.
- *Language compatibility:* Due to the TRNSYS simulation core being implemented in Fortran, user-defined types can be implemented using C/C++. Even though the ability of Fortran programs to call compiled C/C++ functions is limited, for the task at hand all conditions are met to establish sufficient interoperability.

[6] Basically, every TRNSYS type is implemented as a function. Individual simulation components based on the same type are handled via the same function call, using a unique ID and appropriate memory storage utilities that allow to differentiate between the instances.

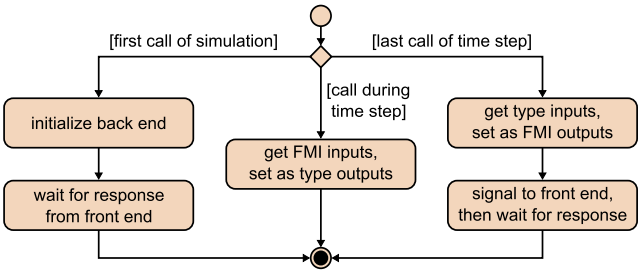

Figure 3. Schematic view of the functionality of the TRNSYS FMI adapter type in dependence on the simulation step.

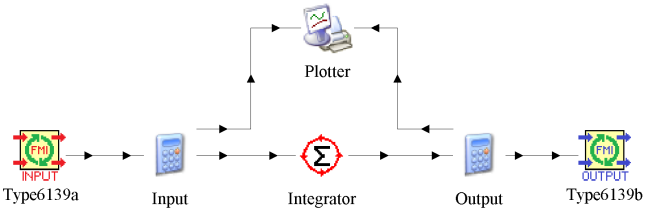

Figure 4. Example of a simple TRNSYS model containing blocks of Type6139 for FMU export.

Therefore the implementation of the front-end and back-end concept discussed in Section 3 can be used to develop a TRNSYS type that acts as FMI adapter. Figure 3 depicts the internal use of the back-end component within this type and its interaction with the simulation master. Please note that inputs to the TRNSYS FMI adapter type are the FMU's outputs and vice versa. Due to the strict fixed step size simulation paradigm of TRNSYS the adapter enforces time steps accordingly using the back-end component's `enforceTimeStep(...)` function. The front-end handles this information accordingly and rejects calls of `doStep(...)` in case they do not conform. The model description flag `canHandleVariableCommunicationStepSize` is set accordingly.

This FMI adapter has been implemented on top of the FMI++ library and is available online[7]. The provided FMI adapter – referred to as *Type6139* – can be included within a TRNSYS model like any other type, with ordinary inputs and outputs coming from and going to other types. In addition, the names of the input and output variables have to be provided (as part of the *Special Cards* in the type's *Proforma*) according to the definition that is also used in the model description. Apart from the additional input and output block of this type, TRNSYS models are constructed in the usual way. Given such a model, an FMU can be generated with the help of a dedicated Python script.

4.1.2 Example application

The example uses a simple thermal model from TRNSYS (see Figure 4) that implements the following first order

[7] Available at `http://trnsys-fmu.sourceforge.net/`.

DOI 10.3384/ecp17132321

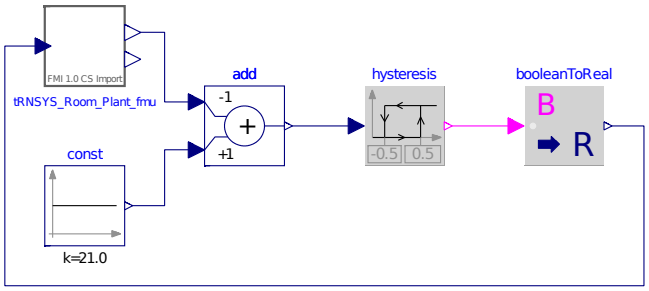

Figure 5. Dymola model.

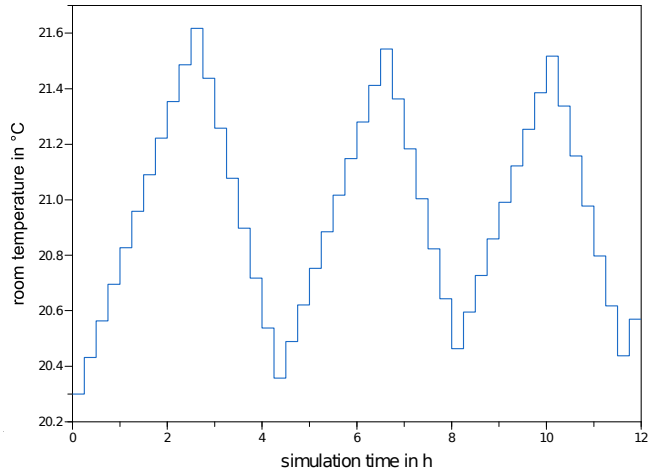

Figure 6. Example TRNSYS FMU output.

ODE:

$$\dot{T}_{\text{room}} = \begin{cases} -Q_{\text{loss}} & \text{if heater is off,} \\ Q_{\text{heater}} - Q_{\text{loss}} & \text{if heater is on.} \end{cases} \quad (1)$$

T_{room} is the room air temperature, Q_{loss} the difference between losses to the environment and inner loads, and Q_{heater} is the power of the heating unit. Both Q_{loss} and Q_{heater} are normalized w.r.t. the thermal capacity of the room air. The model was exported as an FMU with one input variable (associated to the on/off signal of the heater, called `control_signal`) and one output variable (associated to the room temperature, called `room_temperature`).

To test its functionality, the FMU was used as a plant model in a simple closed-loop control system implemented in Dymola, see Figure 5. Depending on the room temperature provided by FMU output variable `room_temperature`, the controller turns the room's heating on or off by setting the FMU input variable `control_signal` to either 0 or 1. More precisely, the model implements a hysteresis controller that turns the heater on as soon as the room temperature falls below 20.5 °C and turns it off when it exceeds 21.5 °C.

Figure 6 shows the results of the simulated Dymola model. Depicted is the room temperature as computed by TRNSYS, which is kept within 21.0 °C ± 0.5 °C by the Dymola controller. Due to the fixed simulation step size of 15 minutes, the switching of the controller state does not happen at the exact edges of the controller's dead-band (i.e., at 20.5 °C and 21.5 °C). Please be aware that this is not a shortcoming of the FMU itself, but due to TRNSYS's restriction to fixed simulation time steps. Such simulation artifacts are unavoidable in fixed-step co-simulation and have to be taken into account by the modeler (e.g., by choosing an adequate simulation step size).

4.2 MATLAB FMI adapter

4.2.1 Implementation

Despite the popularity and widespread use of the numerical computing environment MATLAB, there is so far only comparably little support within the context of FMI. The Modelon FMI Toolbox[8] and the FMI Kit for Simulink[9] offer the export of Simulink models as FMUs for Model Exchange, but so far there is no tool available that allows to provide MATLAB's full functionality via an FMI-compliant co-simulation interface. In the following, a description is given of how the proposed front-end/back-end concept can be utilized to solve this issue.

- *Modularity:* Since MATLAB is a multi-purpose, multi-paradigm computing and programming environment, there are potentially many possible ways to implement an FMI adapter. Within the context of this work, an object-oriented approach has been chosen that relies on a base class called `FMIAdapter`, which provides the full functionality of the FMI adapter. In order to utilize its functionality, the abstract methods `init(...)` and `doStep(...)` have to be implemented by a derived class.

- *Execution control:* In contrast to Simulink, MATLAB defines itself no general notion of time. With the proposed concept, calls to the FMU's `doStep(...)` function are associated to a call to method `doStep(...)` of class `FMIAdapter` (or rather the class derived from it). For such a function call, the current communication point and communication step size are provided as input arguments.

- *Language compatibility:* MATLAB provides many ways for interfacing. Within the context of this work, the SWIG tool has been used to create S-Function bindings to a generic back-end implementation in C++ that can be called from within MATLAB. Even though these bindings can be used directly from MATLAB scripts, it is recommended to utilize their functionality through class `FMIAdapter`.

The MATLAB FMI adapter has been implemented on top of the FMI++ library (for Windows with 32-bit MATLAB) and is available online[10]. As mentioned above, it requires to implement the abstract methods `init(...)` and

[8] Available at `http://www.modelon.com/`.
[9] Available at `http://www.3ds.com/`.
[10] Available at `http://matlab-fmu.sourceforge.net/`.

Listing 2. MATLAB implementation of the FMI adapter for a simple on/off controller.

```
classdef SimpleController < fmipputils.FMIAdapter

  methods

    function init( obj, currentCommunicationPoint )
      obj.defineRealInputs( { 'T' } );
      obj.defineRealOutputs( { 'Pheat' } );
    end

    function doStep( obj, currentCommunicationPoint, communicationStepSize )
      realInputValues = obj.getRealInputValues();
      T = realInputValues(1);
      if ( T >= 90 )
        obj.setRealOutputValues( 0 );
      elseif ( T <= 80 )
        obj.setRealOutputValues( 1e3 );
      end
    end

  end

end
```

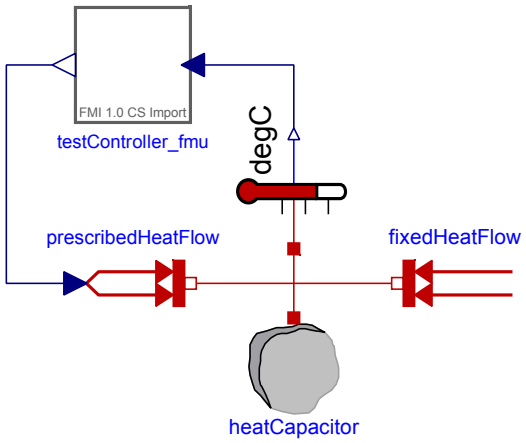

Figure 7. Example Modelica thermal system model.

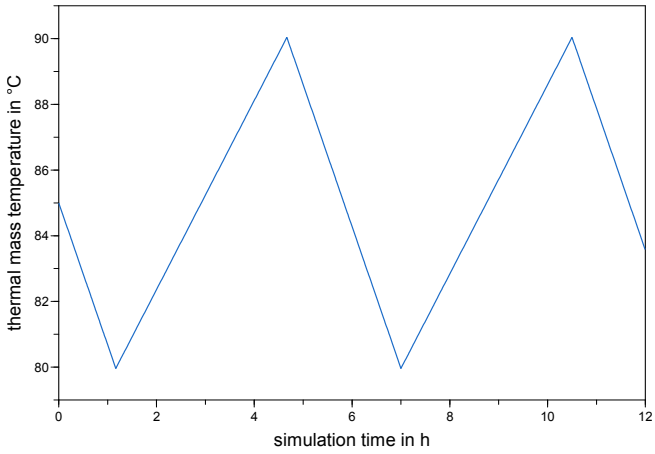

Figure 8. Example Dymola output.

`doStep(...)` of class `FMIAdapter` with the help of an inherited class, see for instance the example code in Listing 2.

Method `init(...)` is intended to initialize input and output variables needed for co-simulation. For instance, input and output variables of type `fmiReal` can be initialized with the help of methods `defineRealInputs(...)` and `defineRealOutputs(...)`, whose input arguments are cell arrays containing the associated variable names. Method `doStep(...)` is called at every simulation step (as requested by the master algorithm). During such a call, methods `getRealInputValues()` and `setRealOutputValues(...)` can be used to get input and set output values for instance.

Since the `init(...)` and `doStep(...)` methods may contain any MATLAB-compliant code, virtually any MATLAB functionality can be made available with the help of this concept. In order to create an FMU from such an implementation, the dedicated script `createFMU.m` has to be called from within MATLAB. Its inputs arguments are only the intended FMI model identifier of the FMU and the path to the class file implementing the FMI adapter. Additional MATLAB files may also be specified, e.g., containing data or further function definitions.

It is also noteworthy that an FMI adapter's functionality can be tested and debugged directly from within MATLAB. Unless explicitly activated, instances of FMI adapters do not try to connect to a back-end component. In this state, the input (output) variables defined by calling the `init(...)` method can be set before (read after) a call to the `doStep(...)` method from within MATLAB with a set of dedicated methods.

DOI 10.3384/ecp17132321

4.2.2 Example application

This example uses a simple on/off controller implemented in MATLAB, to control a thermal system implemented in Modelica. The Modelica model consists of a thermal mass that is connected to a constant negative heat flow (heat sink) and a heater, see Figure 7. The underlying equations of this model are analogous to the previous example, cf. Equation 1. The temperature of the thermal mass is sent as input to the controller, which can set the heater's power output. The MATLAB implementation of the controller is shown in Listing 2. Method `init(...)` defines in line 6 an input variable called `T`, associated to the temperature of the thermal mass, and in line 7 an output variable called `Pheat`, which controls the heater's power output. Method `doStep(...)` retrieves the value of previously defined input variable (lines 11 and 12) and sets the values of the previously defined output variables according to its simple internal logic (lines 14 and 16, respectively).

To test its functionality, the MATLAB controller was exported as FMU and imported into the Modelica model. Figure 8 shows the simulation results. Shown is the temperature of the thermal mass as computed by Dymola, which is kept within the range specified by the controller implementation.

5 Conclusion and Outlook

This work presented a generic approach for FMI-compliant tool coupling for a broad spectrum of tools. The approach is based on the concept of a generic front-end and back-end, with the front-end being directly accessed by a master algorithm as an FMI component. The back-end, which is synchronized to the front-end via a data manager, is associated to the coupled tool. The tool itself interacts with the back-end via a dedicated FMI adapter.

The proposed concept has been implemented as part of the FMI++ library according to the Functional Mock-up Interface version 1.0 specification and adapted to two distinct tools, TRNSYS and MATLAB. In both examples the same front-end, data manager and back-end have been used, with customized FMI adapters to meet the requirements of the specific tools. With the help of two simple co-simulation setups the functionality of both approaches has been shown.

Future work will comprise the extension of the FMI++ implementation to support optional functionality, e.g., handling of input derivatives.

Acknowledgments

Part of this work emerged from the Annex 60 project, an international project conducted under the umbrella of the International Energy Agency (IEA) within the Energy in Buildings and Communities (EBC) Programme. Annex 60 develops and demonstrates a new generation of computational tools for building and community energy systems based on Modelica and the Functional Mock-up Interface standard.

References

D.M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599 – 609, 2003.

T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proceedings of the 8th International Modelica Conference*, 2011.

S. A. Klein, J. A. Duffie, and W. A. Beckman. TRNSYS: A transient simulation program. *ASHRAE Transactions*, 82:623 – 633, 1976.

The MPI Forum. MPI: A Message-Passing Interface Standard. Technical Report Version 2.2, Sept. 2009. URL `http://www.mpiforum.org/`.