The FMI++ Library: A High-level Utility Package for FMI for Model Exchange

Edmund Widl, *Member, IEEE*, Wolfgang Müller, *Student Member, IEEE*, Atiyah Elsheikh, *Member, IEEE*, Matthias Hörtenhuber and Peter Palensky, *Senior Member, IEEE*

Austrian Institute of Technology, Energy Department, Vienna, Austria

{givenname.surname}@ait.ac.at

***Abstract*—The success and the advantages of model-based design approaches for complex cyber-physical systems have led to the development of the FMI (Functional Mock-Up Interface), an open interface specification that allows to share dynamic system models between different simulation environments.**

The FMI specification intentionally provides only the most essential and fundamental functionalities in the form of a C interface. On the one hand, this increases flexibility in use and portability to virtually any platform (even embedded control systems). On the other hand, such a low-level approach implies several prerequisites a simulation tool has to fulfil in order to be able to utilize such an FMI component, for instance the availability of adequate numerical integrators. The FMI++ library presented here addresses this problem for models according to the FMI for Model Exchange by providing high-level functionalities, especially suitable for but not limited to discrete event simulation tools. The capabilities of this approach are illustrated with the help of several applications, where the FMI++ library has been successfully deployed.

This approach intends to bridge the gap between the basic FMI specifications and the typical requirements of simulation tools that do not primarily focus on continuous time-based simulation. In other words, this enables such models to be used as de-facto stand-alone co-simulation components.

I. Introduction

The FMI (Functional Mock-Up Interface) for Model Exchange [1] defines a standard that allows to exchange dynamic models between different simulation tools. This specification consists basically of two parts:

- Model interface: Each model has to provide an executable (shared library) with strictly defined functionalities, implemented in C.
- Model description scheme: Along with the executable an XML-file has to be provided, that contains all necessary information about the model.

When implemented for a model, these two elements together, wrapped up in a ZIP archive, comprise a Functional Mock-Up Unit (FMU).

Even though the actual specification includes intentionally only very fundamental functionalities, it provides all the essential features to represent e.g. Modelica [2], Simulink[1] and SIMPACK[2] models. These features include for instance access to the model parameters, its actual states and derivatives as well as event indicators. The advantages of such a low-level approach are tool independence, i.e. the model interface includes no simulator-specific functionalities, and platform independence, since C-compilers are available for virtually any operating system and processor.

Due to these advantages, the FMI for Model Exchange specification becomes increasingly popular. Several well established simulation tools already offer the possibility to import and simulate FMUs, either natively (e.g. OpenModelica [3], JModelica [4], Dymola[3] or SimulationX[4]) or via third party tools (e.g. Modelon's FMI Toolbox[5] for MATLAB/Simulink).

However, the FMI specification is primarily intended for models that comprise (sets of) hybrid ordinary differential equations (ODE). Therefore, virtually all tools that currently support the import or export of models by means of FMI for Model Exchange are simulation environments that focus on continuous time-based simulation. Hence all of these tools provide their own high-level functionalities (e.g. numerical integrators) which are necessary to handle such FMUs.

For applications that lack these features the FMI for Co-Simulation specification has been developed. As the name implies, this specification requires the models to provide their own internal utilities for simulation. However, most modelling tools only offer the possibility to export models according to the FMI for Model Exchange definition, while fewer support FMI for Co-Simulation. For this reason, the actual usage of FMUs is still mostly limited to simulation environments that are well capable of dealing with continuous time-based models anyway.

The goal of the FMI++ library is to bridge this gap, by providing model-independent functionalities for the simulation of Model Exchange FMUs, including numeric integration and event handling. This offers the possibility to include FMUs with relatively small effort into simulation environments, that by itself do not support the simulation of hybrid ODE-based models. Developers using FMI++ do not need to focus on low-level details of common FMI functionalities, but can concentrate on the task at hand.

This paper is organized as follows: Section II gives an overview of the related work regarding software for FMI

[1] `http://www.mathworks.com`, MathWorks
[2] `http://www.simpack.com`, SIMPACK Multi-Body Simulation
[3] `http://www.dymola.com`, Dassault Systèmes
[4] `http://www.itisim.com/simulationx`, SimualtionX
[5] `http://www.modelon.com`, Modelon FMI Toolbox for MATLAB

978-1-1499-1307-7/13/$26.00 ©2013 IEEE

support. In Section III the functionalities of the FMI++ library are covered in detail. Section IV gives examples of the successful deployment of the FMI++ library, demonstrating its practicality and flexibility. Finally, Section V provides the conclusions and an outlook.

II. Related Work

Convenient high-level approaches like object-oriented interfaces are intentionally left out from the FMI specifications, in order to achieve the possibly highest degree of platform independence. Also the questions of how to unzip an FMU or retrieve the model information from the XML-file lie within the user's responsibility.

There are however free open-source development tools available that implement generic methods for interacting with FMUs: QTronic's FMU Software Development Kit[6] (FMU SDK) and JModelica's FMI Library[7] (FMIL). Both are intended to serve as a starting point for applications that export or import FMUs. They are written in C and offer support for unzipping, (meta) information retrieval, dynamic model loading, setting of model parameters and evaluation of model equations.

The FMIL package has been used as basis for the development of the FMU Compliance Checker[8], a free software package that checks any given FMU's compliance with the FMU standard. Also the PyFMI library[9], a Python package offering an interface for interaction with FMUs, relies on the FMIL.

Also worth mentioning is JFMI[10], a Java wrapper for FMI. Even though it does not provide functionality beyond the scope of the FMI specification, it extends the scope of FMI from C/C++ applications to Java, thus effectively adding another level of platform independence.

However, even though these packages offer lots of convenient functionalities for the practical handling of FMUs, their main features are still fairly basic. For simulation environments not focused on the numerical integration of continuous time-based components, i.e. especially discrete event-based simulators, these packages do not offer the tools needed to easily integrate FMUs. This is where the FMI++ library tries to step in, by offering generic but advanced numerical integration and event handling capabilities for FMUs.

III. Implementation

Since the FMI++ library intends to promote the FMI specification and its use in modern simulation environments, it is a freely available open-source project. This will hopefully also encourage other developers to contribute to the code base for future improvements.

The FMI++ library is written in C++, offering an object-oriented solution to interacting with FMUs. It is easily portable and has so far been tested on Linux architectures and on Windows (using MinGW/GCC). Fig. 1 gives an overview of the library.

A. Dependencies

FMI++ relies on a few well-established open-source software packages, which already provide validated concepts and solutions.

XML parsing and information retrieval: For the retrieval of model (meta) information from the description file, the FMI++ library uses the corresponding FMU SDK functionalities. The FMU SDK itself depends on eXpat[11] for the parsing of the XML model description file.

Numerical integration: For numerical integration FMI++ relies on the ODEINT library [5], which has recently become an official part of the Boost library collection [6]. It is a highly flexible and top performing C++ library for numerically solving ordinary differential equations. Since ODEINT is a header-only template library, it imposes no further dependencies at runtime on FMI++.

FMIL extension: The FMI++ library can be used on top of the FMIL, in which case the FMIL has to be already properly installed.

B. The self-integrating FMU class

The most obvious obstacle for using a bare FMU for Model Exchange is its lack of an integrator. For this reason, the FMI++ library provides a generic method for integration, encapsulated into a dedicated object. The resulting object owns the actual FMU instance and is able to advance its current state up to a specified point in time. It also provides utilities for convenient input and output handling and includes the proper handling of FMU-internal events.

Class FMUBase: This is the base for objects implementing self-integrating FMUs. It is implemented as a pure virtual interface and contains prototypes of all the functions needed by the numerical integrator and for advanced event handling (see Section III-C).

The most important features are:

- initialize/instantiate: These functions are responsible for the instantiation and initialization of the FMU and all corresponding necessary internal actions.
- integrate: Advances the state of the FMU to the specified point in time, with either a specified number of integration steps or a fixed integration step size.
- raiseEvent/handleEvents: These functionalities are the prerequisite for proper event handling. Whenever an event occurs, be it either a change of external inputs or an update of the internal state, the internal FMU instance has to be notified (via raiseEvent) and then the necessary actions have to be taken (by calling handleEvents).
- rewindTime: Event handling may in some cases involve the necessity to reset the internal FMU to a previous state. With this function the FMU-internal clock can be set back. This affects only the value of the internal time,

[6] http://www.qtronic.de, QTronic FMU Software Development Kit
[7] http://www.jmodelica.org/FMILibrary, FMI Library
[8] https://www.fmi-standard.org, FMU Compliance Checker
[9] https://www.jmodelica.org, PyFMI
[10] http://ptolemy.eecs.berkeley.edu/java/jfmi, JFMI
[11] http://expat.sourceforge.net/, The eXpat XML Parser

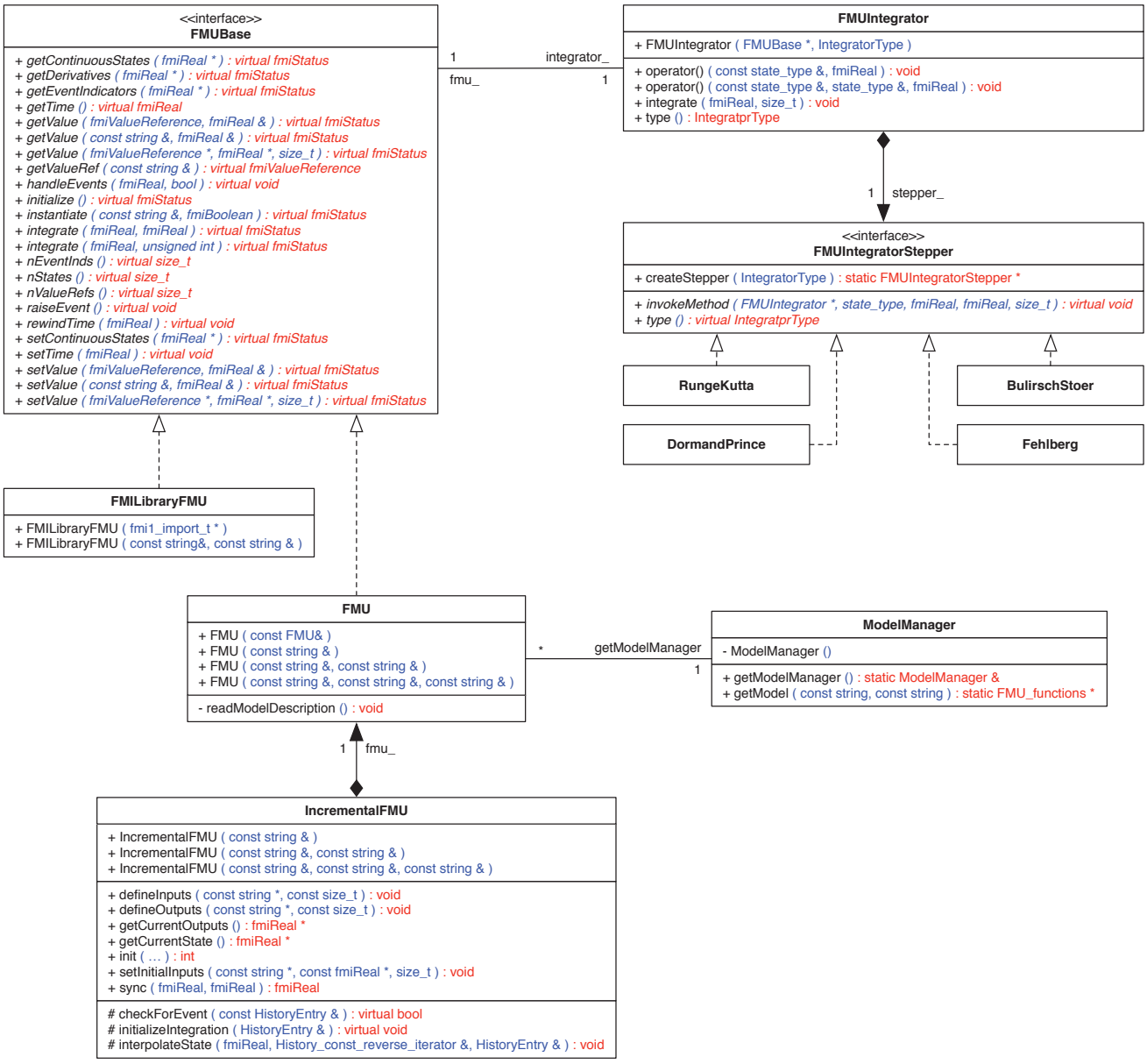

Fig. 1. Overview of the most important features of the FMI++ library.

but not the internal state of the FMU (which has to be changed via setContinuousState, etc.).

- getValue/setValue: Several convenient getter and setter functions are defined, allowing e.g. to set or get values by referring to their names.

Class FMUIntegrator: This class provides the link between ODEINT's numerical integration routines and all classes inherited from FMUBase. It is implemented as a functor object, that provides the necessary inputs (i.e. the FMU's continuous states and the according derivatives) to ODEINT. It also updates the internal state of the FMU with the corresponding result.

Class FMUIntegratorStepper: The actual integration algorithms provided by ODEINT are encapsulated in objects inheriting from this class. Currently implemented are a basic 4th-order method with constant step size (class RungeKutta), a 5th-order method with adaptive step size (class DormandPrince), an 8th-order method with adaptive step size (class Fehlberg) and a high-precision method with controllable order and adaptive step size (class BulirschStoer).

Class FMU: With this class, the FMI++ library provides a completely implemented realization of the interface defined by FMUBase. At construction time it dynamically loads the model, thus avoiding the need to define the model identifier as macro at compile time. It also has access to the complete set of information from the model description.

Class ModelManager: This is a utility class designed for the use by instances of class FMU. It is implemented as a singleton and handles the dynamic loading of FMU models

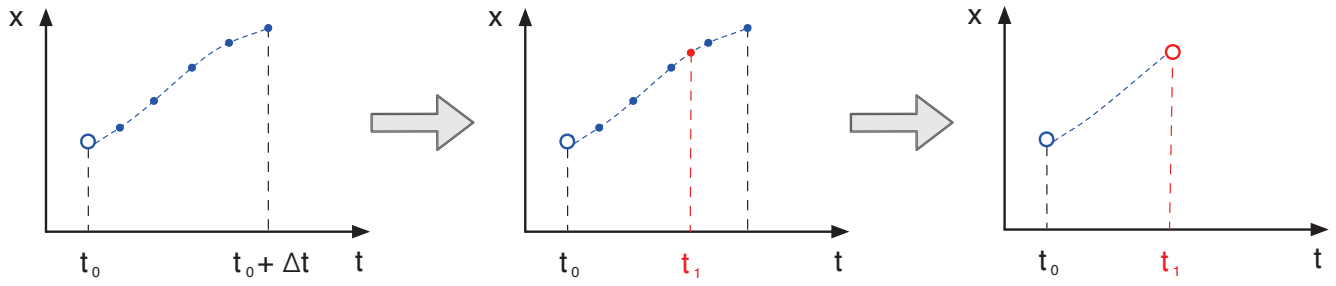

Fig. 2. Schematic view of an incremental update.

and the parsing of the corresponding model description. The singleton instance actually loads and parses each model only once during runtime and stores this information in case another instance of the same model is instantiated. This saves time in cases where simulations include many instances of the same model.

Class FMILibraryFMU: The abstract definition provided by class FMUBase makes it possible to combine other FMI tools with the FMI++ library. The class FMILibraryFMU gives an example for such a hybrid use, by utilizing the FMIL implementation for the internal handling of FMUs. This allows for example to integrate the functionalities provided by FMI++ into PyFMI (see Section IV-B).

C. The advanced event-handling FMU class

The FMI++ library offers the possibility to combine the basic ability to integrate the state of an FMU with advanced event handling capabilities. This is especially useful when using FMUs within discrete event-based simulation environments, where the time difference between updates is not constant.

The class IncrementalFMU implements a lookahead mechanism, where predictions of the FMU's state are incrementally computed and stored. In case an event occurs, these predictions are then used to interpolate and update the state of the FMU. If no event occurs, the latest prediction can be directly used to update the FMU's state.

Fig. 2 shows a schematic view of such an incremental update. Shown on the left, at time t_0 the FMU's state x is represented by a blue circle. According to this state, several predictions (blue dots) up to the time $t+\Delta t$ are computed and stored, with Δt referred to as *lookahead horizon*. In the current implementation the time steps between these internal predictions are constant and have to be specified at instantiation time. Next, depicted in the middle, an (external) event occurs at time t_1. Since the exact time of the event does in general not coincide with one of the predictions, the state at that time is interpolated using the available predictions, depicted by the red dot. Finally, shown on the right, this interpolated prediction is used to update the actual state of the FMU, depicted by a red circle, and the old predictions are deleted.

It is important to note that the actual state of the FMU is not changed when the predictions are calculated. This is only done during the next update.

The most important features of class IncrementalFMU are:

- sync: This function call updates the associated FMU from time t_0 to time t_1. It first uses the previous predictions to update the state of the FMU. Subsequently it calculates a new set of predictions according to the current inputs.
- checkForEvent: This function checks for each new prediction whether an FMU-internal event has occurred. In case it returns true, no further prediction is computed. It is implemented as a virtual function, which enables the user to customize its behaviour.
- handleEvent: This function is called in case checkForEvent has returned true. It is implemented as a virtual function, which enables the user to customize its behaviour.
- initializeIntegration: This function initializes the integration by defining the first prediction. By default, this is the current state of the FMU. It is implemented as a virtual function, which enables the user to customize its behaviour.

The default implementation of class IncrementalFMU recognizes FMU-internal events and stops the prediction at the corresponding time. This implementation uses a linear interpolation technique to estimate the state from the stored predictions. By inheriting from class IncrementalFMU and customizing checkForEvent, handleEvent and initializeIntegration, it is possible to extend this functionality. Section IV-A gives an example where this feature is used to implement an additional controller.

IV. Examples

The functionality of the FMI++ library has been tested by applying it within various simulation environments. The following examples give a brief glimpse of the accomplished results.

A. Inclusion into GridLAB-D

GridLAB-D [7] is a discrete event-based micro-simulation tool with a focus on power distribution systems. It comes with a variety of plug-in modules for modeling and simulating energy generation, distribution and consumption as well as related topics such as controls, network communication or markets. By including the functionalities offered through FMI++ it is possible to include continuous time-based simulation components via FMI.

For the purpose of illustration a simple thermal system has been simulated. It consists of a simplified building model,

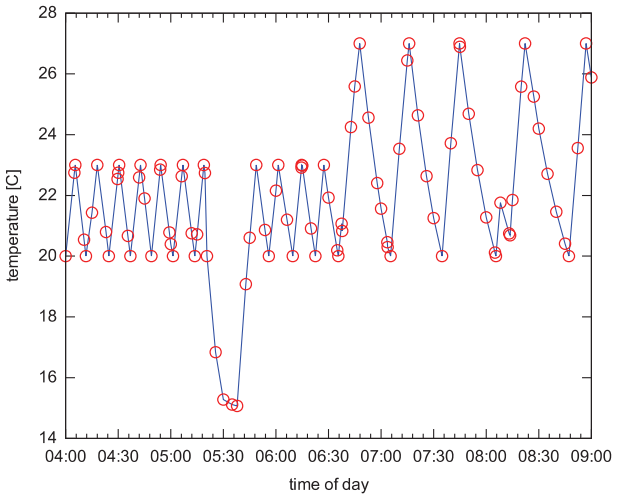

Fig. 3. Temperature profile of a building simulated in GridLAB-D using an FMU model via FMI++. Each red circle corresponds to an update (sync) of the FMU.

whose heater is turned on/off by a two point controller. The controller's set point can be altered by an agent, and at random times the building's windows are opened and closed, effectively changing the thermodynamic properties of the building. A detailed description of the model can be found in Reference [8].

Fig. 3 shows the results of such a simulation for an individual building. The thermal volume was modeled in Modelica, and exported as an FMU with the help of OpenModelica. The FMU was wrapped by a dedicated class derived from IncrementalFMU in order to be usable with GridLAB-D. The two point controller was not part of the FMU, but has been implemented by customizing the functionalities of checkForEvent, handleEvent and initializeIntegration (see Section III-C). Furthermore, the effects of external events have been included this way, i.e. the opening/closing of a window or changes due to the decisions made by the controller's agent.

B. Extension of PyFMI

Using class FMILibraryFMU (see Section III-B) it is possible to include the features offered by FMI++ in PyFMI. Adding the possibility to integrate an FMU only needs 5 additional lines of code in PyFMI's source code (plus several changes in its setup to compile properly). Fig. 4 shows a simple Python script that uses the integrator feature of this modified version of PyFMI.

```
import pyfmi

model = pyfmi.load_fmu( "Simple.fmu" )
model.initialize()

for t in range( 1, 10 ):
    model.integrate( t, 0.01 )
    print str( model.get( "x" ) )
```

Fig. 4. Simple example script for PyFMI, testing FMI++'s integrating feature.

C. Interaction with Ptolemy II's DE domain

Ptolemy II [9] is a generic open source simulation framework for studying the interplay of concurrent processes. These concurrent processes are represented by so-called actors, whose implementations have to obey certain guidelines (referred to as *abstract semantics*) to ensure a well-defined behaviour. The principles behind the design of the FMI++ library allow to define a dedicated actor for Ptolemy II's discrete event (DE) domain, that respects these abstract semantics.

Fig. 5 shows a basic discrete event-based Ptolemy II model containing such an FMU actor (labelled events FMU). In the upper left corner a green box visualizes the DE director that governs the execution of the model. The FMU actor is connected to a plotter, that observes and records the FMU's output. In this particular case the FMU contained within the FMU actor has been generated with the help of OpenModelica. The model simply integrates a constant, thus producing a linear output. However, the model raises internal events, that periodically reverses the constant's sign, which effectively changes the slope of the output signal. Fig. 6 shows the resulting output, where each of the red dots correspond to the firing of the actor. As can be seen, the actor is either fired once the integration's lookahead horizon is reached or when an internal event occurs.

The FMU actor uses internally an instance of class IncrementalFMU, so that the (potentially) costly numerical integration is completely executed in C++. The bindings between Java and FMI++ have been done using SWIG [10]. See Reference [11] for details on the implementation of the FMU actor.

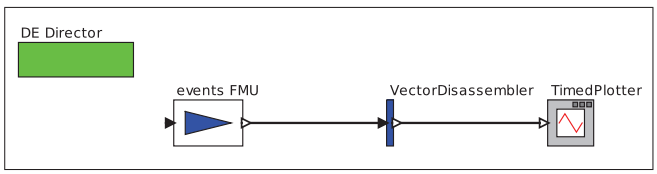

Fig. 5. Ptolemy II model including an FMU actor based on FMI++.

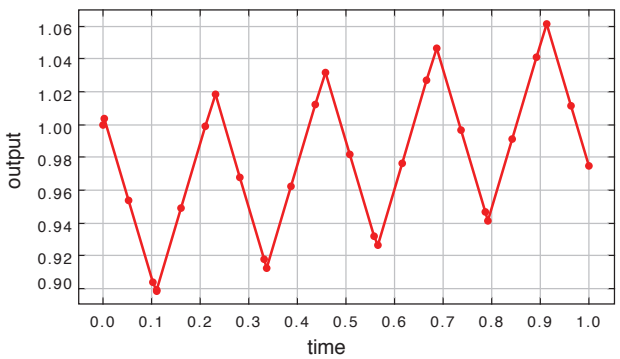

Fig. 6. Resulting output from the FMU actor. Each red dot corresponds to a firing (sync) of the FMU actor.

V. Conclusions and Outlook

This work presents the basic functionalities of the FMI++ library that extends the functionalities provided via the FMI for Model Exchange specifications. Its features allow to utilize FMUs in a convenient way, simplify their integration in existing simulation environments and enable their use as independent co-simulation components.

Currently, the FMI++ library is still in an early prototype phase. The concepts presented here still need to be thoroughly validated. Further improvements are already planned, e.g. support for automatically unzipping FMUs or improved interpolation algorithms for lookahead predictions.

The FMI++ library intends to promote the FMI specification and encourages the use of FMUs in modern simulation environments. The source code will soon be publicly available, and valuable contributions are heartly welcome.

References

[1] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clau, 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, March 20th-22nd, Technical Univeristy, Dresden, Germany*, 2011.

[2] P. Fritzson, *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*, 1st ed. Wiley-IEEE Press, 2011.

[3] P. Fritzson, P. Aronsson, H. Lundvall, K. Nystrm, A. Pop, L. Saldamli, and D. Broman, "The OpenModelica Modeling, Simulation, and Software Development Environment," *Simulation News Europe*, vol. 44, no. 45, Dec. 2005.

[4] J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit, "Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem," *Computers and Chemical Engineering*, vol. 34, no. 11, pp. 1737–1749, Nov. 2010.

[5] K. Ahnert and M. Mulansky, "ODEINT - Solving Ordinary Differential Equations in C++," in *AIP Conf. Proc.*, vol. 1389, 2011, pp. 1586–1589.

[6] B. Schäling, *The boost C++ libraries*. XML Press, 2011.

[7] D. Chassin, K. Schneider, and C. Gerkensmeyer, "Gridlab-d: An open-source power systems modeling and simulation environment," in *Transmission and Distribution Conference and Exposition, 2008. IEEE/PES*, april 2008, pp. 1 –5.

[8] P. Palensky, E. Widl, and A. Elsheikh, "Simulating cyber-physical energy systems: challenges, tools and methods," *submitted to IEEE Transactions on Systems, Man and Cybernetics*, 2012.

[9] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127 – 144, 2003.

[10] D. Beazley, "Automated scientific software scripting with SWIG," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 599 – 609, 2003.

[11] E. Widl and W. Müller, "Linking FMI-based components with discrete event systems," in *Proceedings of the International IEEE Sytems Conference (SysCon)*, 2013, accepted.