A Generic FMU Interface for Modelica

Wuzhu Chen[1] Michaela Huhn[1] Peter Fritzson[2]

[1]Department of Informatics, Clausthal University of Technology, Germany,
{wuzhu.chen,michaela.huhn}@tu-clausthal.de
[2]Department of Computer and Information Science, Linköping University,Sweden, peter.fritzson@liu.se

Abstract

This paper discusses technical issues and implementation of a generic interface to import a Functional Mock-up Unit (FMU) into Modelica simulators, specifically the OpenModelica environment. Whereas other approaches for importing the FMUs rely on functionality specific to the simulator environment, this approach tries to provide a generic Modelica interface for embedding an FMU to be imported into a Modelica model. In this way any FMU conforming to the Functional Mock-up Interface (FMI) for Model Exchange v1.0 Specification for model exchange from MODELISAR can be imported into *any* Modelica simulator. When importing an FMU into a model, the resulting Modelica model can be used just like any pure Modelica models. Hence, a better reusability and interoperability for both sides, namely the external models provided via FMI and the Modelica environment, are achieved.

Keywords OpenModelica, Functional Mock-up Interface(FMI), Functional Mock-up Unit (FMU) import, Modelica code generation

1. Introduction

1.1 Modelica Models and Modelica Simulators

The Modelica language is primarily designed for modeling and simulating complex hybrid dynamic systems, so its features intend to reflect the structure, the inherent behavior, the hierarchy and the heterogeneity of those systems. Briefly speaking, Modelica language is object-oriented, equation-based, component-based and capable of modeling multi-domain problems. As a result of the modularity concepts, models described by the Modelica language can be almost freely exchanged between different Modelica tools only with some minimal restrictions, for instance, differences in the version of Modelica language or the version of the Modelica Standard Library being used. All hierarchical information as well as the behavior of models will be explicitly transferred during such exchanges, since they are delineated in the standardized Modelica language.

However, simulating complex and heterogeneous systems bears several needs for integrating models that originate from other sources: For some subsystems elaborated or optimized models may have been implemented using other domain-specific modeling languages or a general-purpose programming language.

Another reason that applies even if Modelica is used for subsystems is the protection of intellectual property. In many situations it is a business case to enable using the model for simulation without externalizing the model's structure and behavior. Thus, tool support for integrating individual sub-models into a Modelica simulation environment as well as co-simulation that facilitates the coupling of various simulators is a basic requirement for real-world multi-domain modeling.

A Modelica simulator typically contains a compiler and a run-time module. The compiler translates Modelica models into an appropriate programming language (C, C++, etc.), which will be further translated into executables by their own standard compilers. The run-time module then executes the outcomes from the compiler, handles the configuration of particular simulation runs, solves the equation system, depicts and stores the results.

1.2 Functional Mock-up Interface (FMI)

The *FMI for Model Exchange v1.0* [3] has been specified and released by the MODELISAR[1] consortium. This specification provides a series of open, standardized interfaces, through which the instantiation, initialization, execution of an individual executable (or C code) model representation called *Functional Mock-up Unit* (FMU) can be performed within a simulation environment. In order to use the FMI from both sides, the FMU and the simulator have to implement the interface functions as defined in the FMI specification. The FMU contains a concrete mathematical model described by differential, algebraic and discrete equations with possible events of a dynamic physical system transformed into explicit form and represented as C code or machine code. The idea of the FMI standard is, that an FMU can be generated from any modeling environment and im-

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

[1] MODELISAR is a ITEA2 project focusing on the interoperability between Modelica and AUTOSAR to support Vehicle Functional Mock-up

ported into any simulation environment whenever the compliance with FMI specification for model exchange is fulfilled.

Technically, a generated FMU will be distributed as a compressed .fmu file basically consisting of two parts. The first part is the model interface functions according to the FMI specification. They can be given either in C source code or in binary forms (e.g. in the form of .dll or .so files) to protect the model developers' intellectual property. The second part is the model description XML file, conforming to the schema defined in the FMI specification. The XML file contains all information about the model variables and some other optional information. Additional parts can be added and compressed into the FMU, as for instance the documentation and the icon of the model. Compared to the exchange of Modelica models, exchanging of FMUs is at a lower abstraction level and more target-oriented, and consequently less flexible due to the export and import procedures and the conversion of acausal Modelica models to causal explicit form.

1.3 State of the Art for FMU Import

Currently, some Modelica simulators already support FMU import, for example SimulationX ®, Dymola ®, and the open source tool JModelica.org[2], and some others. Although the FMI interface is vendor neutral, the implementations of import of the FMU realized by the above mentioned simulator tools are not. Either the import of FMU is implemented on the basis of some specific programming language [1], Python in this case, or the Modelica language is internally extended [4] because the Modelica language lacks support for the full range of functionalities specified in FMI. However, since each tool vendor offers a proprietary implementation of FMU import, we are actually keeping reinventing the wheel in the sense of multiple implementations of FMU import. Moreover, there still exists some additional difficulties in the FMI specification of FMU import that hamper the proliferation of the FMI standard in practice, for example the lack of support of multiple instances of FMU models generated by some simulation tool, the lack of coupling of imported FMUs with Modelica models, to name just some of them.

1.4 Motivation for a Generic FMU Interface to Modelica

As discussed before, the benefits of a standardized interface for model exchange are widely agreed and have led to the FMI specification by MODELISAR. But furthermore there is also a need for a generic and uniform implementation of the import functionality for Modelica tools. A holistic modeling of a heterogeneous system may incorporate the combined simulation of Modelica models and a number of imported FMUs within several Modelica simulators. In this way, the systematic behavior of the total system can be analyzed without revealing the modeling know-how embedded of the FMU components. A key role to generalize the model exchange and to glue all these benefits together is a tool-independent, uniform interface from FMUs to Modelica models, which can be also distributed along with the FMUs for potential use by Modelica modelers. Thus, the model exchange facility indeed crosses the boundary of simulators in the Modelica context.

The rest of the paper is organized as follows: In Section 2, the implementation of the generic interface are presented in detail. In section 3, a case study of a hybrid dynamic system is investigated and the results are analyzed in OpenModelica environment. At the end in Section 4, the conclusions of this FMU import approach are given.

2. Implementation of the Interface

2.1 Overview of the Implementation

As already mentioned in the introduction Section 1, we aim at a generic interface for importing FMUs into Modelica models in a way that any FMUs can be directly evaluated and simulated in any Modelica simulator. Thus the key issue is to decouple the FMU import process from specific Modelica simulator implementations. Most of the FMU interface functions will be mapped to Modelica external functions bijectively. Some extra Modelica helper classes and functions are also implemented to perform additional tasks, for example type conversion, special construct instantiation, message printing and so on. The imported FMU itself is represented as a Modelica class through the extension of an external object with variable information from parsing the model description XML file. In this way, multiple instances of this FMU can be easily declared and connected with models that are described in Modelica.

The import process contains following key steps

- To extract the archived .fmu file in some specified folder
- To parse the model description file for model information
- To load the library file or include the source file if provided
- To integrate the functions and the variable information into a Modelica block

Summing up these aspects, the concept of the implementation of FMU import is illustrated in Figure 1.

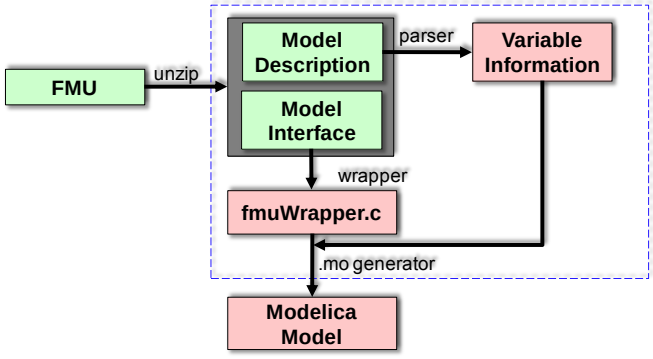

Figure 1. An overview of FMU import.

[2] JModelica.org is an open source Modelica simulation environment tailored for optimization problems. For more information please refer to http://www.jmodelica.org

2.2 Details of the Prototype Implementation

A prototype of the generic FMU interface for Modelica has been implemented and tested in the OpenModelica environment. The details of the implementation are going to be discussed in this section.

Even in the open-source domain, we find numerous possibilities for the decompression of a .fmu archive file, e.g. *zlib*, *gzip*, *PowerArchiver*, *7-zip*, etc. Since this step is rather trivial compared to the others, it will not be discussed further.

After having decompressed the archive file, the model description XML file and the model interface implementation are stored in a specified directory, e.g. in OpenModelica under Microsoft Windows®the directory will be "<OPENMODELICAHOME> \tmp\fmu". Information about model variables and other additional information can be extracted from the XML file by an appropriate XML parser, we will use the parser in the *libxml2* library provided by the OpenModelica environment.

For the current prototype, the model interface implementation is supposed to be a shared dynamic link library (DLL). By successfully parsing the XML and loading the DLL, the handle of the DLL, the addresses of the exported interface functions and the information data from the XML file are stored in a data structure called *FMI*, which is defined in a wrapper header file. The instantiation of this structure in Modelica is achieved by a class, which extends the Modelica *External Object* as shown below.

```
class fmuFunctions "List of interface functions"
  extends ExternalObject;
  function constructor
    input String dllPath;
    output fmuFunctions fmufun;
    external "C" fmufun = instantiateFMIFun(dllPath)
    annotation(Include = "#include <fmuWrapper.c>");
  end constructor;
  function destructor
    input fmuFunctions fmufun;
    external "C" freeFMIFun(fmufun)
    annotation(Include = "#include <fmuWrapper.c>");
  end destructor;
end fmuFunctions;
```

An FMU is represented as a Modelica class called `fmuModelInst` in the prototype. In the generic interface, the detailed structure of this class is however hidden behind the so-called *Modelica External Object*. Consequently, the internal structure of the instance cannot be accessed and manipulated directly by Modelica. Provided the necessary inputs are given in Modelica, the instance of an FMU can be declared and immediately instantiated, e.g. `fmuModelInst inst=fmuModelInst(argument_list);`. Later on, this instance reference may be passed as an argument to some external code for various purposes such as data updating, data evaluation, memory freeing, etc. In fact, the actual instantiation is done by calling the external C function `fmiInstantiate(...)` defined in the constructor section of the class `fmuModelInst`, see the following Modelica code for details:

```
class fmuModelInst
  extends ExternalObject;
  function constructor
    input fmuFunctions in_fmufun;
    input String in_instName;
    input String guid;
    input fmuCallbackFuns functions;
    input Boolean logFlag;
    output fmuModelInst inst;
    external "C" inst=fmiInstantiate(in_fmufun,
     in_instName,guid,functions,logFlag)
     annotation(Include="#include <fmuWrapper.c>");
  end constructor;
  function destructor
    input fmuModelInst in_inst;
    external "C" fmiFreeModelInst(in_inst)
    annotation(Include="#include <fmuWrapper.c>");
  end destructor;
end fmuModelInst;
```

There are 24 standard interface functions defined in the *FMI specification v1.0*. Most of them are wrapped in the `fmuWrapper.c` and then mapped one-to-one into Modelica external functions with a slight difference in naming convention to indicate the wrapped functions are not the implementation of FMI but exclusive for FMU import. For instance, the `fmiGetDer(...)` in C

```
void fmiGetDer(void* in_fmi, void* in_fmu,
     double* der_x, size_t nx, const double* x){
  FMI* fmi = (FMI*) in_fmi;
  fmiStatus status;
  if(fmi->getDerivatives){
    status = fmi->getDerivatives(in_fmu,der_x,nx);
    if(status>fmiWarning){
      printf("fmiGetDerivatives(...) failed...\n");
      exit(EXIT_FAILURE);} }
  return; }
```

will be mapped in Modelica as the following snippet:

```
function fmuGetDer "calculate the derivatives"
  input fmuFunctions fmufun;
  input fmuModelInst inst;
  input Integer nx;
  input Real x[nx];
  output Real der_x_out[nx];
  external "C" fmiGetDer(fmufun,inst,der_x_out,nx,x)
  annotation(Include = "#include <fmuWrapper.c>");
end fmuGetDer;
```

2.3 Implementation of FMI Calling Sequence

The UML 2.0 compliant state machine in Figure 2 shows the calling sequence of the interface functions specified in *FMI for Model Exchange v1.0*. This calling sequence has to be implemented in the prototype in Modelica as well.

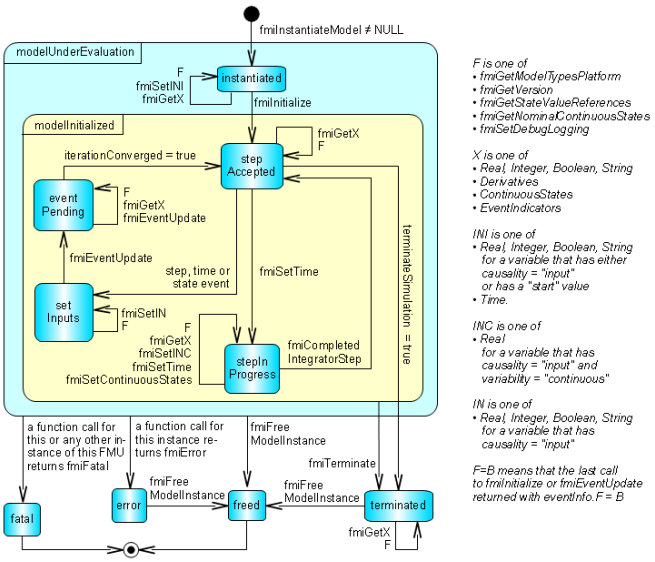

Figure 2. Calling sequence of interface functions[3].

A schematic view of communication between an FMU instance and the environment is also given in the FMI Specification in Figure 3. In this figure, it is shown that different kinds of variables are defined to describe the state of an FMU and the information flowing into\out of an FMU.

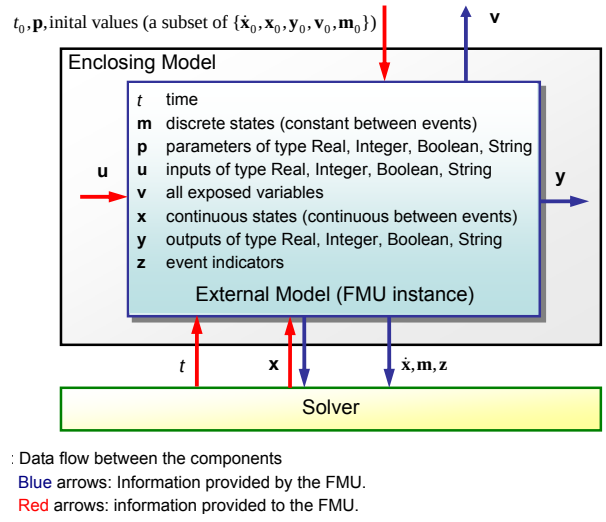

Figure 3. Data flow for information exchange[3].

In order to satisfy the requirements of the function calling sequence and the data exchange of an FMU, a suitable construct in Modelica should be defined and standardized. Despite the fact that Modelica is primarily designed as a modeling language based on acausal equation systems[2], Modelica does offer non-declarative algorithmic constructs, in which procedural methods may be called in a specific order. Due to the arguments discussed previously, the calling sequence and specific data-flows can be realized as a Modelica block incorporating with internal variables, **input****output** signals, **algorithm** and **equation** sections. A snippet of such a Modelica block is given below for illustration.

```
block FMUBlock "a prototype"
  output Real y[ny];
  input Real u[nu];
protected
  constant String dllPath="fmu\\bouncingBall
   \\binaries\\win32\\bouncingBall.dll";
  constant String instName="bouncingBall";
  constant String guid=
     "{9d95f943-e636-4f71-8d7d-6f54c4128f13}";
  parameter Boolean logFlag=false,tolControl=true;
  parameter Integer nx = 2;
  parameter Integer ni = 1;
  replaceable parameter Real relTol = 0.0001;
  replaceable Real x[nx](start={1.0,0.0}),der_x[nx];
  Real out_x[nx],out_der_x[nx];
  ...
  fmuModelInst inst=fmuModelInst(fmufun,instName,
                          guid,functions,logFlag);
  fmuEventInfo evtInfo = fmuEventInfo();
  ...
  fmuFunctions fmufun = fmuFunctions(dllPath);
  fmuCallbackFuns functions = fmuCallbackFuns();
  initial algorithm
  fmuSetTime(fmufun, inst, time);
  fmuInit(fmufun,inst,tolControl,relTol,evtInfo);
  x:=fmuGetContStates(fmufun, inst);
  algorithm
    der_x:=fmuGetDer(fmufun, inst, x);
  equation
    der(x) = der_x;
  algorithm
    fmuSetContStates(fmufun, inst, x);
    fmuCompIntStep(fmufun, inst, stepEvt);
    prez:=z;
    z:=fmuGetEventInd(fmufun, inst, ni);
    fmuStateEvtCheck(stateEvt, ni, z, prez);
    fmuEvtUpdate(fmufun,inst,evtInfo,timeEvt,
           stepEvt,stateEvt,interMediateRes);
    ...
    when zeroCrossing<0 then
      reinit(x[1], out_x[1]);
      reinit(x[2], out_x[2]);
      ...
    end when;
end FMUBlock;
```

2.4 Other Technical Issues

In the FMI Specification v1.0, the *fmiCallbackFunctions* is defined as a *struct* containing function pointers provided by the environment. In Modelica there is no corresponding **record** to store information about function pointers. Nevertheless, Modelica provides the *external object construct*, which allows the user to define and access his or her own external data structures. Thus it is possible to declare a subclass of this `ExternalObject` to instantiate and de-instantiate the *fmiCallbackFunctions* data structure via externally defined C functions.

The FMI specification also states that some functions, such as *fmiInitialize(...)*, should be called only once, but this can not be ensured by the Modelica simulation environment, because during the initialization phase the runtime module may traverse the `initial` **algorithm** section several times. A solution to solve this issue is to define a flag as an initialization indicator in the above mentioned **class** `fmuFunctions`. Whenever this flag records a call for initialization, the *fmiInitialize(...)* will not be called again.

Since the predefined type `Boolean` in Modelica and *fmiBoolean* in FMI are implemented differently in terms of C types, when importing an FMU into most Modelica simulators, the conversion between these two types needs to be considered carefully. The former, i.e. the `Boolean` type, corresponds to *unsigned char* in C in the OpenModelica context. The latter, i.e. the *fmiBoolean* in FMI, corresponds to *char* in C. As a consequence, external functions have been implemented in the wrapper to perform the necessary bi-directional conversions and these functions can be called directly from Modelica.

Connecting imported FMU blocks together hierarchically might arise the difficulty in determining the correct call sequence of setter and getter methods in FMUs for input and output signals. A general alternative is to determine the call sequence according to the connection graph based on some causal relationships between these signals. As soon as the call sequence has been sorted, flags associating with the setter or getter methods can be added in the argument list to indicate the correct order. In this paper, since the imported FMUs are transformed into Modelica models, the correct call sequence of setter and getter methods of input and output signals can be automatically determined by the Modelica simulation environment, [3] in Appendix B.5.

2.5 Difficulties in the Implementation

Some functionalities defined in FMI specification cannot be mapped directly into Modelica without specific treatment.

FMI functions *fmiCompletedIntegratorStep*, *fmiEventUpdate* and *fmiTerminate* are not constant functions, so they do not behave as mathematical functions and there might be a need to mark these functions as "impure" in Modelica via some extension to the language.

The FMI function *fmiCompletedIntegratorStep* cannot be defined and called properly in Modelica, since Modelica language does not provide the functionalities, through which the status of a integration step can be queried and when an integrator step is finished can be detected.

When calling the FMI function *fmiGetEventIndicators* from Modelica model, it will introduce the hysteresis twice to the event indicators. According to the FMI specification, FMUs will add a small hysteresis to the event indicators to avoid event "chattering". And a Modelica tool will do the same when calculating the "zero crossing". So the resulted event triggered by the imported FMU is slightly inaccurate. This problem can be solved by calling external functions, e.g. `fmuStateEventCheck(...)`, which will check the events according to the current values and the previous values of the event indicators given as function arguments.

This paper focuses mainly on FMU import in OpenModelica environment, some of these tricky issues requiring specific treatments are temporarily handled in OpenModelica.

In the presented prototype, the Modelica model is partially automatically generated by the code generator. A full automation of the generation process will be completed soon. After the translation from an FMU to a Modelica model, any instances of this FMU can now be declared as " `FMUBlock fmu1,fmu2,...;` " and connected with other Modelica models. Such capabilities are shown below about the the Bouncing Ball case study.

In the case study discussed in Section 3, the only kind of events are state events. The state event is triggered by the simulation environment outside the FMU model via the `FMUBlock`. The function `fmuStateEvtCheck(...)` will check for the state event and return information through the formal *fmiBoolean* parameter `stateEvt`. If a state event occurs, the `fmuEvtUpdate(...)` is called and updates the states accordingly. Other events, such as time events and step events, might also occur during simulations of dynamic systems, but they are not covered in the case study Bouncing Ball.

3. Case Study: Bouncing Ball

The bouncing ball example is a good candidate to show the behavior of a hybrid dynamic system and the capability of event handling in the presented FMU import prototype. The bouncing ball FMU is generated using the free software development kit *FMU SDK* provided by QTronic GmbH. [5]

3.1 Multiple Instances of an FMU

As mentioned previously, multiple instances of an FMU block can be easily created as below by the standard variable declaration in Modelica. Furthermore, modifications can also be applied to the declared variables e.g.:

```
model FMUMultipleInstance
  FMUBlock fmu1,fmu2(redeclare parameter Real
  relTol = 0.002, redeclare Real x[nx]
  (start = {2.0,0.0}));
end FMUMultipleInstance;
```

The simulation results of the Modelica code above in *OpenModelica 1.7.0* are illustrated in Figure 4, which shows the expected property of multiple instances of an FMU block.

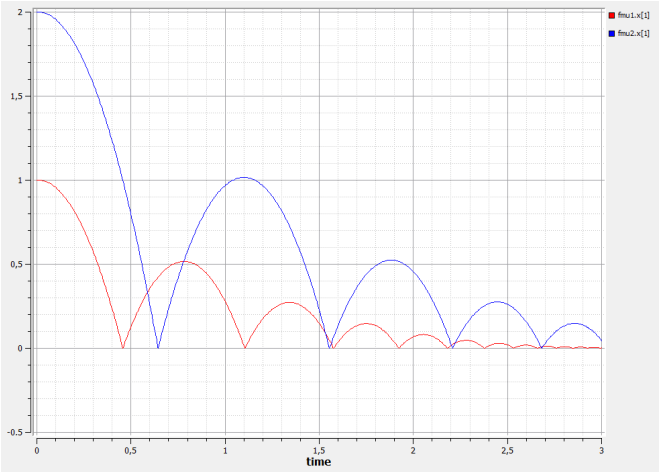

Figure 4. Results of multiple instances of an FMU

3.2 Connection with another Modelica Model

A simple connection between an FMU instance and a nonlinear block from the Modelica Standard Library 3.1 (MSL 3.1) is shown in Figure 5. From the user perspective, the connection between an FMU instance and any Modelica models becomes nearly trivial when using our generic interface. The Modelica code is:

```
model FMUConnection
  FMUBlock fmu;
  Modelica.Blocks.Nonlinear.VariableDelay vd
    (delayTime = 0.2);
equation
  vardelay.u = fmu.x[1];
end FMUConnection;
```

and the simulation result from *OpenModelica* is given in Figure 5, which also produced the expected delay of the input signal.

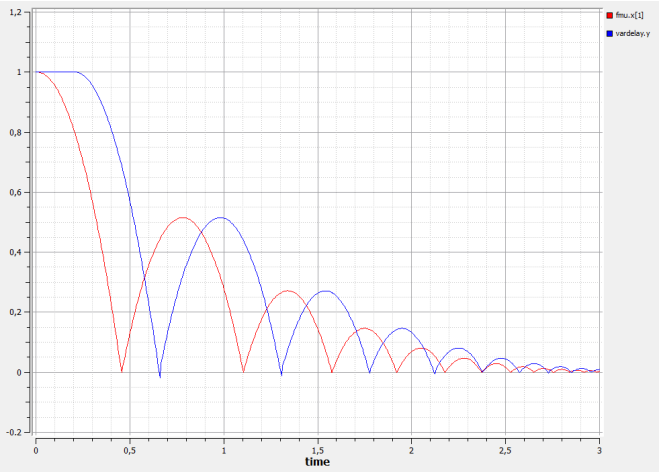

Figure 5. Results of connection between an FMU instance and Modelica model

4. Conclusions and Outlook

In this paper, we presented the concept of a generic interface in Modelica and a prototypical implementation. The interface enables multiple instances of an imported FMU and a simple connection between FMU instances and Modelica models. Of course, any properties provided by the Modelica modeling language, e.g. inheritance, modification, extension and so on, can be applied to the FMU block and its declared variables. The approach presented here fills the gap between the vendor-neutral FMI for model exchange and currently existing vendor-dependent FMU import implementations. An obvious drawback of our approach is the the higher memory consumption compared to simulating a pure Modelica model or simulating an FMU in a stand-alone simulator, because storages have to be assigned for both Modelica objects and FMU instances. Future work includes more test cases of the prototype, full automation of the code generation and tuning to support other Modelica simulators. It is also worth mentioning that FMI specification is not Modelica specific, so a similar approach for FMU import may be checked with other behavioral languages, e.g. VHDL-AMS, provided they support some mechanism to access external functions.

However, the following fundamental requirements need to be satisfied to facilitate FMU import. Obviously, for the generic FMU interface for Modelica to work properly, the provided FMU needs to be fully compliant with the *FMI for Model Exchange v1.0 Specification*. For instance, the FMUs generated from Dymola®7.4 do not support multiple instances of an FMU at the moment. Secondly, the simulator must not modify the generated Modelica models during loading the FMU block, which is not guaranteed by some simulators. Some simulators adapt the loaded Modelica code according to their specific rules, which creates problems in this situation. And thirdly, but not last, the construction of the *Modelica External Object* construct has to be well supported by the simulator.

Acknowledgments

This work is funded by the Federal Ministry of Education and Research (BMBF), Germany, in the ITEA2 project OPENPROD (grant 01IS09029D).

We are also thankful to Martin Sjölund and Mohsen Torabzadeh-Tari from Linköping University for the productive discussions on this topic.

References

[1] Christian Andersson, Johan Åkesson, Claus Führera, and Magnus Gäfvert. Import and Export of Functional Mock-up Units in JModelica.org. In *Proceedings of 8th Modelica Internationl Conference*. Modelica Consortium, 2011.

[2] Peter Fritzson. *Principle of Object-Oriented Modeling and Simulation with Modelica 2.1*. WILEY-INTERSCIENCE, 2003.

[3] MODELISAR. Functional Mock-up Interface for Model Exchange 1.0. http://www.functional-mockup-interface.org.

[4] Christian Noll, Torsten Blochwitz, Thomas Neidhold, and Christian Kehrer. Implementation of Modelisar Functional Mock-up Interfaces in SimulationX. In *Proceedings of 8th Modelica Internationl Conference*, Webergasse1, 01067 Dresden, Germany, 2011. Modelica Consortium.

[5] QTronic. FMU SDK (FMU Software Development Kit). `http://www.qtronic.de/de/fmusdk.html`.