The Functional Mockup Interface for Tool independent Exchange of Simulation Models

T. Blochwitz[1], M. Otter[2], M. Arnold[3], C. Bausch[4], C. Clauß[5], H. Elmqvist[9], A. Junghanns[6], J. Mauss[6], M. Monteiro[4], T. Neidhold[1], D. Neumerkel[7], H. Olsson[9], J.-V. Peetz[8], S. Wolf[5]

Germany: [1]ITI GmbH, Dresden; [2]DLR Oberpfaffenhofen; [3]University of Halle, [4]Atego Systems GmbH, Wolfsburg; [5]Fraunhofer IIS EAS, Dresden; [6]QTronic, Berlin; [7]Daimler AG, Stuttgart; [8]Fraunhofer SCAI, St. Augustin;

Sweden: [9]Dassault Systèmes, Lund.

Abstract

The Functional Mockup Interface (FMI) is a tool independent standard for the exchange of dynamic models and for co-simulation. The development of FMI was initiated and organized by Daimler AG within the ITEA2 project MODELISAR. The primary goal is to support the exchange of simulation models between suppliers and OEMs even if a large variety of different tools are used. The FMI was developed in a close collaboration between simulation tool vendors and research institutes. In this article an overview about FMI is given and technical details about the solution are discussed.

Keywords: Simulation; Co-Simulation, Model Exchange; MODELISAR; Functional Mockup Interface (FMI); Functional Mockup Unit (FMU);

1 Introduction

One of the objectives of the development and usage of tool independent modeling languages (e.g. Modelica®[1][1], VHDL-AMS [10]) is to ease the model exchange between simulation tools. However, modeling languages require a huge effort to support them in a tool. It is therefore common to provide also low level interfaces, to exchange models in a less powerful, but much simpler way. Another aspect of model exchange is the protection of product know-how which could be recovered from their physical models.

Several tools offer proprietary model interfaces, such as:

- Matlab/Simulink®[2]: S-Functions [3]
- MSC.ADAMS[3]: user-written subroutines [4]
- Silver: Silver-Module API [5]
- SIMPACK: user routines [6]
- SimulationX®[4]: External Model Interface [7]

Currently, no tool independent standard for model exchange (via source or binary code in a programming language) is available. The same holds for the situation in the field of co-simulation.

Vendors of Modelica tools (AMESim, Dymola, SimulationX) and non Modelica tools (SIMPACK, Silver, Exite), as well as research institutes worked closely together and recently defined the Functional Mockup Interface[5]. This interface covers the aspects of model exchange [8] and of co-simulation [9]. This development was initiated and organized by Daimler AG with the goal to improve the exchange of simulation models between suppliers and OEMs. Within MODELISAR, Daimler has set up 14 automotive use cases for the evaluation and improvement of FMI. In this article, the technical details behind FMI are discussed.

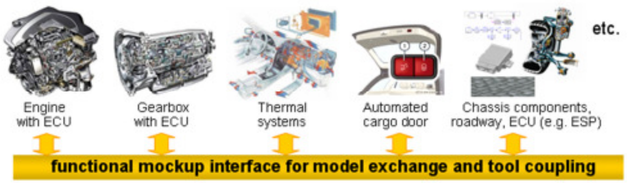
etc.
Engine with ECU
Gearbox with ECU
Thermal systems
Automated cargo door
Chassis components, roadway, ECU (e.g. ESP)
functional mockup interface for model exchange and tool coupling

Figure 1: Improving model-based design between OEM and supplier with FMI.

[1] Modelica® is a registered trademark of the Modelica Association.

[2] Matlab®/Simulink® are regist. trademarks of The MathWorks Inc.

[3] MSC® is a registered trademark and MSC.ADAMS is a trademark of MSC.Software Corporation or its subsidiaries.

[4] SimulationX® is a registered trademark of ITI GmbH.

[5] http://www.functional-mockup-interface.org

2 The Functional Mock-Up Interface

2.1 Main Design Ideas

The FMI standard consists of two main parts:

1. *FMI for Model Exchange*:
The intention is that a modeling environment can generate C-Code of a dynamic system model in form of an input/output block that can be utilized by other modeling and simulation environments. Models are described by differential, algebraic and discrete equations with time-, state- and step-events. The models to be treated can be large for usage in offline simulation; and it is also possible to use models for online simulation and in embedded control systems on micro-processors.

2. *FMI for Co-Simulation*:
The intention is to couple two or more simulation tools in a co-simulation environment. The data exchange between subsystems is restricted to discrete communication points. In the time between two communication points, the subsystems are solved independently from each other by their individual solver. Master algorithms control the data exchange between subsystems and the synchronization of all slave simulation solvers (slaves). The interface allows standard, as well as advanced master algorithms, e.g. the usage of variable communication step sizes, higher order signal extrapolation, and error control.

Both approaches share a bulk of common parts that are sketched in the next subsections.

2.2 Distribution

A component which implements the FMI is called Functional Mockup Unit (FMU). It consists of one zip-file with extension ".fmu" containing all necessary components to utilize the FMU:

1. An XML-file contains the definition of all variables of the FMU that are exposed to the environment in which the FMU shall be used, as well as other model information. It is then possible to run the FMU on a target system without this information, i.e., with no unnecessary overhead. For FMI-for-Co-Simulation, all information about the "slaves", which is relevant for the communication in the co-simulation environment is provided in a slave specific XML-file. In particular, this includes a set of capability flags to characterize the ability of the slave to support advanced master algorithms, e.g. the usage of variable communication step sizes, higher order signal extrapolation, or others.

2. For the FMI-for-Model-Exchange case, all needed model equations are provided with a small set of easy to use C-functions. These C-functions can either be provided in source and/or binary form. Binary forms for different platforms can be included in the same model zip-file.
For the FMI-for-Co-Simulation case, also a small set of easy to use C-functions are provided in source and/or binary form to initiate a communication with a simulation tool, to compute a communication time step, and to perform the da-

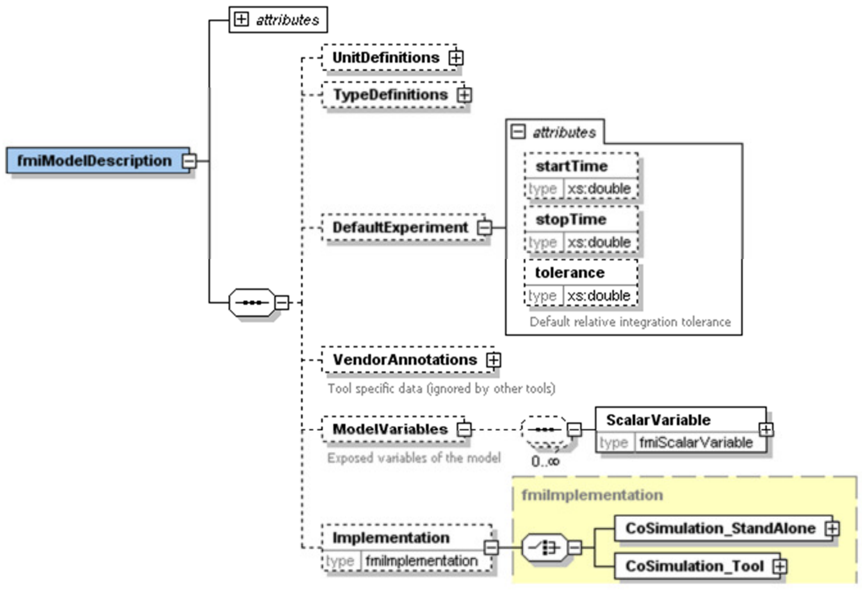

Figure 2: Top level part of the FMI XML schema

ta exchange at the communication points.

3. Further data can be included in the FMU zip-file, especially a model icon (bitmap file), documentation files, maps and tables needed by the model, and/or all object libraries or DLLs that are utilized.

2.3 Description Schema

All information about a model and a co-simulation setup that is not needed during execution is stored in an XML-file called "modelDescription.xml". The benefit is that every tool can use its favorite programming language to read this XML-file (e.g. C, C++, C#, Java, Python) and that the overhead, both in terms of memory and simulation efficiency, is reduced. As usual, the XML-file is defined by an XML-schema file called "fmiModelDescription.xsd". Most information is identical for the two FMI cases.

In Figure 2, the top-level part of the schema definition is shown. All parts are the same for the two FMI-cases, with exception of the element "Implementation". If present, the import tool should interpret the model description as applying to co-simulation. As a consequence, the import tool must select the C-functions for co-simulation, otherwise for model exchange. An important part of the "Implementation" is the definition of capability flags to define the capabilities that the co-simulation slave supports:

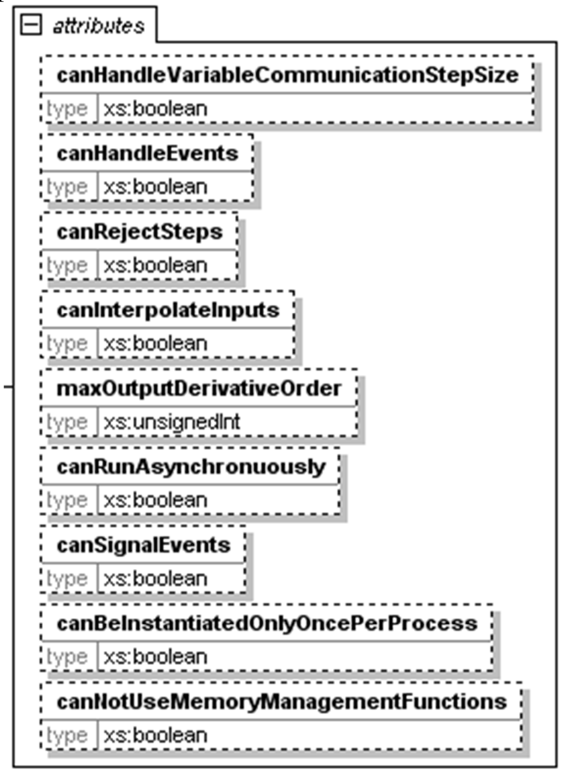

Figure 3: Capability flags of FMI for Co-Simulation.

These flags are interpreted by the master to select a co-simulation algorithm which is supported by all connected slaves.

2.4 C-Interface

The executive part of FMI consists of two header files that define the C-types and –interfaces. The header file "fmiPlatformTypes.h" contains all definitions that depend on the target platform:

```
#define fmiPlatform "standard32"
#define fmiTrue  1
#define fmiFalse 0
#define fmiUndefinedValueReference
                (fmiValueReference)(-1)

typedef void*          fmiComponent;
typedef unsigned int   fmiValueReference;
typedef double         fmiReal    ;
typedef int            fmiInteger;
typedef char           fmiBoolean;
typedef const char*    fmiString ;
```

This header file must be used both by the FMU and by the target simulator. If the target simulator has different definitions in the header file (e.g., "**typedef float** fmiReal" instead of "**typedef double** fmiReal"), then the FMU needs to be re-compiled with the header file used by the target simulator. The header file platform, for which the model was compiled, as well as the version number of the header files, can be inquired in the target simulator with FMI functions.

In this first version of FMI, the minimum amount of different data types is defined. This is not sufficient for embedded systems and will be improved in one of the follow-up versions of FMI.

The type `fmiValueReference` defines a handle for the value of a variable: The handle is unique at least with respect to the corresponding base type (like `fmiReal`) besides alias variables that have the same handle. All structured entities, like records or arrays, are "flattened" in to a set of scalar values of type `fmiReal`, `fmiInteger` etc. A `fmiValueReference` references one such scalar. The coding of `fmiValueReference` is a "secret" of the modeling environment that generated the model. The data exchange is performed using the functions `fmiSetXXX(...)` and `fmiGetXXX(...)`. XXX stands for one of the types Real, Integer, and Boolean. One argument of these functions is an array of `fmiValueReference`, which defines which variable is accessed. The mapping between the FMU variables and the `fmiValueReference`s is stored in the model description XML file.

For simplicity, in this first version of FMI a "flat" structure of variables is used. Still, the original hierarchical structure of the variables can be retrieved, if a flag is set in the XML-file that a particular convention of the variable names is used. For example, the Modelica variable name

"`pipe[3,4].T[14]`"

defines a variable which is an element of an array of records "pipe" of vector T ("." separates hierarchical levels and "[...]" defines array elements).

Header-file "fmiFunctions.h" contains the prototypes for functions that can be called from simulation environments.

The goal is that both textual and binary representations of models are supported and that several models using FMI might be present at link time in an executable (e.g., model A may use a model B). For this to be possible the names of the FMI-functions in different models must be different unless function pointers must be used. For simplicity and since the function pointer approach is not desirable on embedded systems, the first variant is utilized by FMI: Macros are provided in "`fmiFunctions.h`" to build the actual function names. A typical usage in an FMI model or co-simulation slave is:

```
#define  MODEL_IDENTIFIER MyFMU
#include "fmiFunctions.h"
< implementation of the FMI functions >
```

For example, a function that is defined as

"`fmiGetDerivatives`"

is changed by the macros to the actual function name

"`MyFMU_fmiGetDerivatives`",

i.e., the function name is prefixed with the model or slave name and an "_". The "`MODEL_IDENTIFIER`" is defined in the XML-file of the FMU. A simulation environment can therefore construct the relevant function names after inspection of the XML-file. This can be used by (a) generating code for the actual function call or (b) by dynamically loading a dynamic link library and explicitly importing the function symbols by providing the "real" function names as strings.

3 FMI for Model Exchange

3.1 Mathematical Description

The goal of the Model Exchange interface is to numerically solve a system of differential, algebraic and discrete equations. In this version of the interface, ordinary differential equations in state space form with events are handled (abbreviated as "hybrid ODE").

This type of system is described as a piecewise continuous system. Discontinuities can occur at time instants $t_0, t_1, \ldots, t_n$, where $t_i < t_{i+1}$. These time instants are called "events". Events can be known before hand (= time event), or are defined implicitly (= state and step events).

The "state" of a hybrid ODE is represented by a continuous state $\mathbf{x}(t)$ and by a time-discrete state $\mathbf{m}(t)$ that have the following properties:

- $\mathbf{x}(t)$ is a vector of real numbers (= time-continuous states) and is a continuous function of time inside each interval $t_i \le t < t_{i+1}$, where $t_i = \lim_{\varepsilon \to 0}(t_i + \varepsilon)$, i.e., the right limit to t_i (note, $\mathbf{x}(t)$ is continuous between the right limit to t_i and the left limit to t_{i+1} respectively).
- $\mathbf{m}(t)$ is a set of real, integer, logical, and string variables (= time-discrete states) that are constant inside each interval $t_i \le t < t_{i+1}$. In other words, $\mathbf{m}(t)$ changes value only at events. This means, $\mathbf{m}(t) = \mathbf{m}(t_i)$, for $t_i \le t < t_{i+1}$.

At every event instant t_i, variables might be discontinuous and therefore have two values at this time instant, the "left" and the "right" limit. $\mathbf{x}(t_i)$, $\mathbf{m}(t_i)$ are always defined to be the right limit at t_i, whereas $\mathbf{x}^-(t_i)$, $\mathbf{m}^-(t_i)$ are defined to be the "left" limit at t_i, e.g.: $\mathbf{m}^-(t_i) = \mathbf{m}(t_{i-1})$. In the following figure, the two variable types are visualized:

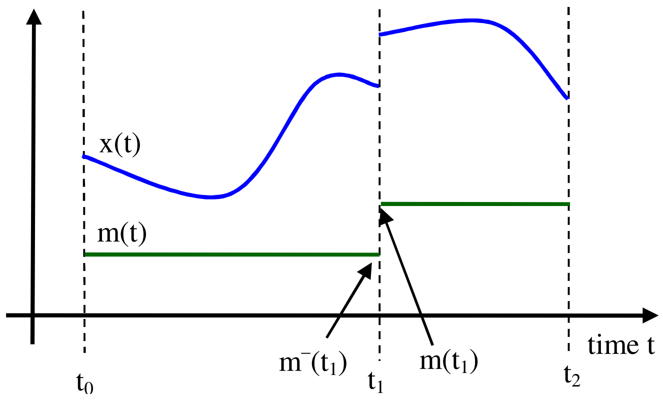

Figure 4: Piecewise-continuous states of an FMU: time-continuous (x) and time-discrete (m).

An event instant t_i is defined by one of the following conditions that gives the smallest time instant:

1. At a predefined time instant $t_i = T_{next}(t_{i-1})$ that was defined at the previous event instant t_{i-1} either by the FMU, or by the environment of the FMU due to a discontinuous change of an input signal u_j at t_i. Such an event is called time event.
2. At a time instant, where an event indicator $z_j(t)$ changes its domain from $z_j > 0$ to $z_j \le 0$ or vice versa (see Figure **5** below). More precisely: An event $t = t_i$ occurs at the smallest time instant "min t" with $t > t_{i-1}$ where "$(z_j(t) > 0) \ne (z_j(t_{i-1}) >$

0)". Such an event is called state event. All event indicators are piecewise continuous and are collected together in one vector of real numbers **z**(t).

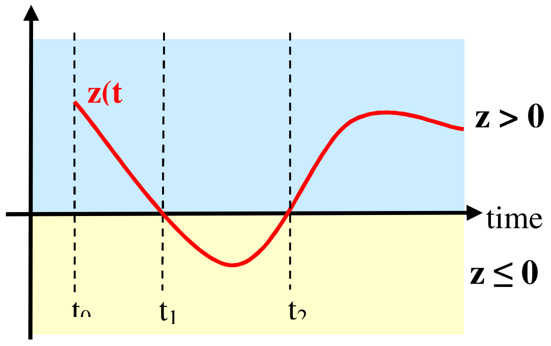

Figure 5: **An event** occurs when the event indicator changes its domain from z > 0 to z ≤ 0 or vice versa.

3. At every completed step of an integrator, `fmiCompletedIntegratorStep` must be called. An event occurs at this time instant, if indicated by the return argument `callEventUpdate`. Such an event is called step event. Step events are, e.g., used to dynamically change the (continuous) states of a model, because the previous states are no longer suited numerically.

An event is always triggered from the environment in which the FMU is called, so it is not triggered inside the FMU. A model (FMU) may have additional variables **p**, **u**, **y**, **v**. These symbols characterize sets of real integer, logical, and string variables, respectively. The non-real variables change their values only at events. For example, this means that $u_j(t) = u_j(t_i)$, for $t_i \leq t < t_{i+1}$, if u_j is an integer, logical or string variable. If u_j is a real variable, it is either a continuous function of time inside this interval or it is constant in this interval (= time-discrete). "**p**" are parameters (data that is constant during the simulation), "**u**" are inputs (signals provided from the environment), "**y**" are outputs (signals provided to the environment that can be used as inputs to other subsystems), and "**v**" are internal variables that are not used in connections, but are only exposed by the model to inspect results. Typically, there are a few inputs **u** and outputs **y** (say 10), and many internal variables **v** (say 100000).

3.2 Caching of Variables

Depending on the situation, different variables need to be computed. In order to be efficient, FMI is designed so that the interface requires only the computation of variables that are needed in the present context. For example, during the iteration of an integrator step, only the state derivatives need to be computed provided the output of a model is not connected. It might be that at the same time instant other variables are needed. For example, if an integrator step is completed, the event indicator functions need to be computed as well. For efficiency it is then important that in the call to compute the event indicator functions, the state derivatives are not newly computed, if they have been computed already at the present time instant. This means, the state derivatives shall be reused from the previous call. This feature is called "caching of variables".

Caching requires that the model evaluation can detect when the input arguments, like time or states, have changed. This is achieved by setting them explicitly with a function call since every such function call signals precisely a change of the corresponding variables. A typical call sequence to compute the derivatives

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, \mathbf{p}, t)$$

as function of states, inputs, and parameters is therefore:

```
// Instantiate FMU
// ("M" is the MODEL_IDENTIFIER)
m = M_fmiInstantiateModel("m", ...);
 ...
// set parameters
M_fmiSetReal(m, id_p, np, p);

// initialize instance
M_fmiInitialize(m, ...);
 ...
// set time
M_fmiSetTime(m, time);
 ...
// set inputs
M_fmiSetReal(m, id_u, nu, u);
 ...
// set states
M_fmiSetContinuousStates(m, x, nx);
 ...
// get state derivatives
M_fmiGetDerivatives(m, der_x, nx);
```

To obtain the FMU outputs:

$$\mathbf{y} = \mathbf{f}(\mathbf{x},\mathbf{u},\mathbf{p},t)$$

as function of states, inputs, and parameters, the environment would call:

```
...
// get outputs
M_fmiGetReal(m, id_y, ny, y);
...
```

The FMU decides internally which part of the model code is evaluated in the present context.

4 FMI for Co-Simulation

Co-simulation is a simulation technique for coupled time-continuous and time-discrete systems that exploits the modular structure of coupled problems in all stages of the simulation process (pre-processing, time integration, post-processing).

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

Examples for co-simulation techniques are the distributed simulation of heterogeneous systems, partitioning and parallelization of large systems, multi rate integration, and hardware-in-the-loop simulations.

A simulation tool can be coupled if it is able to communicate data during simulation at certain time points (communication points, t_{Ci}), see Figure 6.

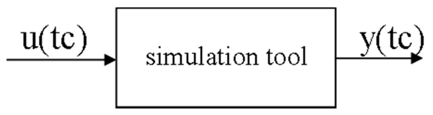

Figure 6: Coupleable simulation tool

4.1 Master Slave Principle

Instead of coupling the simulation tools directly, it is assumed that all communication is handled via a master (Figure 7).

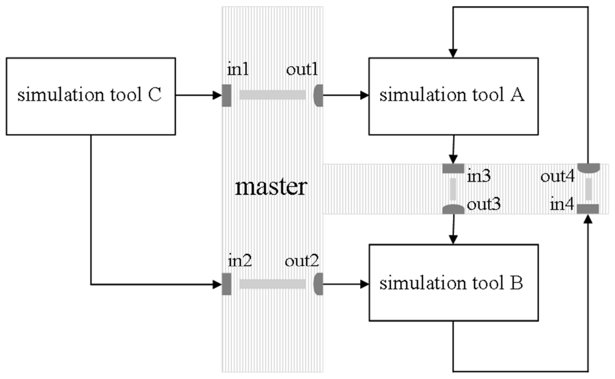

Figure 7: Three tools are controlled by one master

The master plays an essential role in controlling the coupled simulation. Besides distribution of communication data, the master analyses the connection graph, chooses a suitable simulation algorithm and controls the simulation according to that algorithm. The slaves are the simulation tools, which are prepared to simulate their subtask. The slaves are able to communicate data, execute control commands and return status information.

4.2 Interface

The FMI for Co-Simulation defines similar functions like FMI for Model Exchange for creation, initialization, termination, and destruction of the slaves. In order to allow distributed scenarios, the co-simulation functions provide some more arguments. E.g. the function `fmiInstantiateSlave(...)` provides the string argument `mimeType` which defines the tool which is needed to compute the FMU in a tool based co-simulation scenario (see section 4.3).

For data exchange, the `fmiGet…/fmiSet…` functions of FMI for Model Exchange are used here too. In order to allow higher order extrapolation/interpolation of continuous inputs/outputs additional Get/Set functions are defined for input/output derivatives.

The computation of a communication time step is initiated by the call of `fmiDoStep(...)`. The function arguments are the current communication time instance, the communication step size, and a flag that indicates whether the previous communication step was accepted by the master and a new communication step is started. Depending on the internal state of the slave and the previous call of `fmiDoStep(...)`, the slave has to decide which action is to be done before the step is computed. E.g. if a communication time step is repeated, the last taken step is to be discarded.

Using this interface function, different co-simulation algorithms are possible:

- Constant and variable communication step sizes.
- Straightforward methods without rejecting and repetition of communication steps.
- Sophisticated methods with iterative repetition of communication steps.

4.3 Use Cases

The FMI for Co-Simulation standard can be used to support different usage scenarios. The simplest one is shown in the following figure (in order to keep it simple, only one slave is drawn).

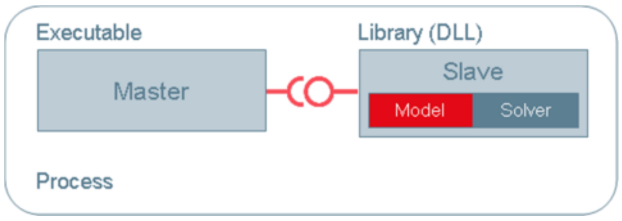

Figure 8: Simple stand alone use case.

Master and slave are running in the same process. The slave contains model and solver, no additional tool is needed.

The next use case is carried out on different processes. A complete simulation tool acts as slave. In this case, the FMI is implemented by a simulation tool dependent wrapper which communicates by a (proprietary) interface with the simulation tool.

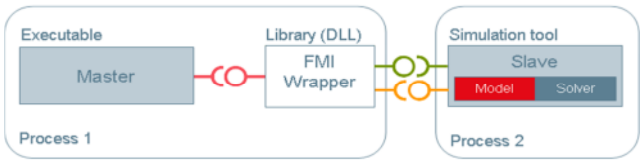

Figure 9: Slave is a simulation tool

The green and orange interfaces can be inter-process communication technologies like COM/DCOM, CORBA, Windows message based interfaces or signals and events. The co-simulation master does not notice the usage of an FMI wrapper here. It still utilizes only the FMI for Co-Simulation.

Figure 10 demonstrates a distributed co-simulation scenario.

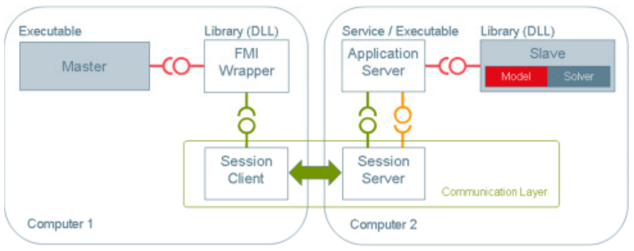

Figure 10: Distributed co-simulation scenario.

The slave (which can be a simulation tool as in Figure 9 too) and the master run on different computers. The data exchange is handled by a communication layer which is implemented by a special FMI wrapper. Neither the master nor the slave notice the technology used by the communication layer. Both utilize the FMI for Co-Simulation only.

5 Comparing Model-Exchange Approaches

As shown in section 1, there are multiple, proprietary approaches for exchanging executable models. Here we discuss the differences with respect to the following properties:

1. Interface coverage:
   a) Model representation as ordinary differential equation (ODE) or differential algebraic equation (DAE).
   b) Does the API support querying Jacobian matrix information?
   c) Is it possible to transfer structural data via the API? If information about algebraic dependencies between outputs and inputs are supplied, the importing tool is able to detect and handle algebraic loops automatically.
2. Event handling: Does the API support transporting event handling information between model and simulation environment for
   a) Time events?
   b) State events?
   c) Step events?
   d) Event iteration?
3. Step-size control: Does the API support
   a) Rejecting of time step by the model?
   b) Variable step sizes?
4. Efficiency: Does the API support
   a) efficient computing (e.g. allowing value cashing)?
   b) efficient result storage (e.g. via alias mechanism and storing large numbers of internal variables)?
   c) efficient argument handling (data copy required)?
5. Programming languages: Which programming languages are supported by the API
6. Documentation: Is documentation sufficient for
   a) using the APIs data types?
   b) building models with this API (export)?
   c) using models with this API in other simulation environments (import)?
   d) each models variables (inputs, outputs, states etc.)?
7. Miscellaneous:
   a) Interoperability: which platforms are supported?
   b) Does changing the (version of the) simulation environment require model recompilation?
   c) Compact and flexible file format, that allows the inclusion of additional data.
8. Status of API:
   a) License?
   b) Developed and maintained by only one tool vendor?

	Property	Simulink: S-Function	ADAMS: user routines	Silver: Silver API	SIMPACK: user routines	SimulationX: EMI	**FMI for Model Exchange**
1a	Representation	ODE	ODE	Co-Sim[1]	ODE/DAE	ODE	ODE[2]
1b	Jacobian support	no	no	no	no	no	no[3]
1c	Structural data	yes	no	no	no[4]	yes	yes
1a	Time events	yes[5]	no	no	yes	yes[5]	yes[5,6]
1b	State events	yes	no	no	yes	yes	yes
2c	Step events	yes	no	no	yes	yes	yes
2d	Event iteration	no	no	no	yes	yes	yes
3a	Discard time step	no	no	no	no	yes	yes
3b	Variable step size	yes	yes	no	yes	yes	yes
4a	Eff. computing	no[7]	no[7]	no[7]	no[7]	no[7]	yes
4b	Eff. Result storage	no[8]	no	no	no[8]	no[8]	yes
4c	Data copy required	no	no	no	no	no	yes
5	API language	C, C++, Fortran	C, Fortran	C, Python	C, Fortran	C	C
6a	Doc API data types	yes	yes	yes	yes	yes	yes
6b	Doc model generation	yes	yes	yes	yes	yes	yes
6c	Doc model use	no	no	yes	yes	yes	yes
6d	Doc model variables	no		yes	no	no	yes
7a	Platform support	independent[9]	Windows, Linux	Windows	Windows, UNIX	Windows	independent.[9,10]
7b	Recompilation	yes	yes	no	yes[11]	yes	no
7c	Compact file format	no	no	no	no	no	yes
8a	Legal status	proprietary	proprietary	open	proprietary	open	open
8b	Proprietary	yes	yes	yes	yes	yes	no

1. Silver supports only exchanging models including their own solvers, communicating at project-dependent, equidistant time intervals.
2. DAE support is planned for one of the next versions.
3. Planned for one of the next FMI versions.
4. Loops are detected and can be handled using additional user intervention on the model.
5. API supports time events, but definition is based on real variables which could lead to inexact results.
6. Efficient and numerically robust time event handling is planned for FMI Version 2.0
7. Caching mechanism is possible, but the component itself has to observe which data have changed since the last call.
8. API is not designed to transfer more data than inputs, outputs, and states. Internal variables are not published by the external model.
9. The component can be provided as binary for one operating system and/or as source code.
10. The model source code can be part of the FMU archive file.
11. Recompilation is only needed for major version changes.

6 Tools supporting FMI

The following tools are currently supporting the FMI version 1.0 (for an up-to-date list see www.functional-mockup-interface.org/tools.html):

- **Dymola 7.4** (FMU export and import, www.3ds.com/products/catia/portfolio/dymola),
- **JModelica.org 1.3** (FMU import, www.jmodelica.org),
- **Silver 2.0** (FMU import, www.qtronic.com/en/silver.html),
- **SimulationX 3.4** (FMU import and export, FMU co-simulation, www.simulationx.com, see [10])
- **Simulink** (FMU export by Dymola 7.4 via Real-Time Workshop, www.mathworks.com/products/simulink),
- **Fraunhofer Co-Simulation Master** (see [12])

The respective vendors plan to support FMI with their following tools:

- **AMESim** (FMU export and import, www.lmsintl.com/imagine-amesim-1-d-multi-domain-system-simulation),
- **EXITE**, **EXITE ACE** (FMU export and import, www.extessy.com),
- **OpenModelica** (FMU export and import; a prototype for FMU export is available, www.openmodelica.org),
- **SIMPACK** (FMU import, FMU co-simulation, www.simpack.com),
- **TISC** (FMU import, www.tlk-thermo.com).
- **MpCCI** (FMU import and export, www.mpcci.de)

7 Conclusions

The FMI eases the model exchange and co-simulation between different tools. It has a high potential being widely accepted in the CAE world:

- It was initiated, organized and pushed by Daimler to significantly improve the exchange of simulation models between suppliers and OEMs.
- It was defined in close collaboration of different tool vendors within the MODELISAR project.
- Industrial users were involved in the proof of concept within MODELSAR.
- FMI can already be used with several Modelica tools, Simulink, multi-body and other tools.
- In the ITEA2 project OPENPROD [10], FMI extensions are currently developed with respect to optimization applications.

Current priorities of the further development inside MODELISAR are:

- Unification and harmonization of FMI for Model Exchange and Co-Simulation
- Improved handling of time events.
- Improved support for FMUs in embedded systems.
- Efficient interface to Jacobian matrices.

8 Acknowledgements

This work is supported by the German BMBF (Förderkennzeichen: 01IS08002) and the Swedish VINNOVA (funding number: 2008-02291) within the ITEA2 MODELISAR project (http://www.itea2.org/public/project_leaflets/MODELISAR_profile_oct-08.pdf). The authors appreciate the partial funding of this work.

9 References

[1] Modelica Association: **Modelica – A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification, Version 3.2**. March 24, 2010. Download: https://www.modelica.org/documents/ModelicaSpec32.pdf

[2] **VHDL-AMS: IEEE Std 1076.1-2007**. Nov. 15, 2007. VHDL-AMS web page: http://www.vhdl.org/vhdl-ams/

[3] The Mathworks: **Manual: Writing S-Functions**, 2002

[4] **Using ADAMS/Solver Subroutines**. Mechanical Dynamics, Inc., 1998.

[5] A. Junghanns: **Virtual integration of Automotive Hard- and Software with Silver**. ITI-Symposium, 24.-25.11.2010, Dresden.

[6] http://www.simpack.com

[7] Blochwitz T., Kurzbach G., Neidhold T. **An External Model Interface for Modelica**. 6th International Modelica Conference, Bielefeld 2008. www.modelica.org/events/modelica2008/Proceedings/sessions/session5f.pdf

[8] MODELISAR Consortium: **Functional Mock-up Interface for Model Exchange. Version 1.0**, www.functional-mockup-interface.org

[9] MODELISAR Consortium: **Functional Mock-up Interface for Co-Simulation**. Version 1.0, October 2010, www.functional-mockup-interface.org

[10] **OPENPROD - Open Model-Driven Whole-ProductDevelopment and Simulation Environment**, www.openprod.org

[11] Ch. Noll, T. Blochwitz, Th. Neidhold, Ch. Kehrer: **Implementation of Modelisar Functional Mock-up Interfaces in SimulationX.** 8th International Modelica Conference. Dresden 2011.

[12] J. Bastian, Ch. Clauß, S. Wolf, P. Schneider: **Master for Co-Simulation Using FMI.** 8th International Modelica Conference. Dresden 2011.