Import and Export of Functional Mock-up Units in JModelica.org

Christian Andersson[a,c] Johan Åkesson[b,c] Claus Führer[a] Magnus Gäfvert[c]

[a]Department of Numerical Analysis, Lund University, Sweden
[b]Department of Automatic Control, Lund University, Sweden
[c]Modelon AB, Sweden

Abstract

Different simulation and modeling tools often use their own definition of how a model is represented and how model data is stored. Complications arise when trying to model parts in one tool and importing the resulting model in another tool or when trying to verify a result by using a different simulation tool. The Functional Mock-up Interface (FMI) is a standard to provide a unified model execution interface. In this paper we present an implementation of the FMI specification in the JModelica.org platform, where support for import and export of FMI compliant models has been added. The JModelica.org FMI import interface is written in Python and offers a complete mapping of the FMI *C* API. JModelica.org also offers a set of Pythonic convenience methods for interacting with the model in an object-oriented manner. In addition, a connection to the simulation environment Assimulo which is part of JModelica.org is offered to allow for simulation of models following the FMI specification using state of the art numerical integrators. Generation of FMI compliant models from JModelica.org will also be discussed.

Keywords: JModelica.org; Assimulo; Sundials; FMI, FMUs

1 Introduction

In an effort to provide a unified model interface for different simulation tools and modeling environments, the MODELISAR consortium defined an open interface called the Functional Mock-up Interface. The idea is that both Modelica-based and non-Modelica-based tools may generate and exchange models that follow the FMI specification. FMI compliant models are referred to as Functional Mock-up Units (FMUs). This enables users to create specialized models in one modeling environment, connect them in a second and finally simulate the complete system using a third simulation tool. This in turn, facilitates tool interoperability and model exchange.

In this paper, we present an implementation of the Functional Mock-up Interface in the JModelica.org platform, [2]. The implementation consists of support both for exporting FMUs from JModelica.org and importing FMUs generated by other tools.

Python was selected as the implementation language for the interface. The choice of Python for the integration was based on several reasons. The main advantage is that Python is a powerful and dynamic programming language with an clear and readable syntax with a low threshold for users to create their own simulation scripts, regardless of if you come from an MATLAB environment or a low-level programming language such as *C*. There are several packages that make Python a good option for scientific computing. One example is Scipy together with Numpy [13], which contains mathematically relevant functions , another is Matplotlib [10] for visualization in a MATLAB like format. There are also Python packages aimed specifically at interfacing code written in other programming languages. One such package is CTYPES [9], which makes connection to *C* and loading of dynamic linked libraries possible. This is a necessity for implementation of the FMI standard.

Another reason for choosing Python is that for part of the implementation, functionality could be reused with little or no modification as similar functionality already exists in JModelica.org. The XML framework needed to load and generate FMUs is already in-place in JModelica.org and the connection to the simulation environment Assimulo is similar to that of the connection between a model generated from JModelica.org

and Assimulo.

The paper is outlined as follows. In Section 2, a brief background is given about the JModelica.org platform and the simulation package Assimulo together with an overview of the Functional Mock-up Interface. The implementation is described in Section 3. In Section 4, the Van der Pol oscillator and the Full Robot from the Modelica Standard Library is simulated. The result is compared with a simulation using Dymola [6]. An example from the Air Conditioning library is also given. Finally, Section 6 summarizes this paper.

2 Background

2.1 JModelica.org

JModelica.org [2] is an "extensible Modelica-based open source platform for optimization, simulation and analysis of complex dynamic systems"[1] with the mission:

> "To offer a community-based, free, open source, accessible, user and application oriented Modelica environment for optimization and simulation of complex dynamic systems, built on well-recognized technology and supporting major platforms."

The platform offers compilers for Modelica, [15] and Optimica, [1], a simulation package called Assimulo and a direct collocation algorithm for solving large-scale DAE-based dynamic optimization problems. The user interface in JModelica.org is based on Python, which provides means to conveniently develop complex scripts and applications. The platform is designed both for large-scale industrial needs and for prototyping in a research environment. It provides synergies between state of the art methods resulting from research and problems industrially relevant problems. JModelica.org supports the major platforms Windows, Mac and Linux.

2.2 Assimulo

Assimulo, [3], is the default simulation package in JModelica.org. It is a Python package consisting of several solvers for solving explicit ordinary differential equations (ODEs),

$$\dot{x} = f(t,x), \quad x(t_0) = x_0 \tag{1}$$

as well as differential algebraic equations (DAEs),

$$\begin{gathered} F(\dot{x},x,w,t) = 0, \\ x(t_0) = x_0, \quad \dot{x}(t_0) = \dot{x}_0, \quad w(t_0) = w_0. \end{gathered} \tag{2}$$

Examples of solvers supported by Assimulo are a fifth-order three-stage Radau method, explicit Euler with fixed step-sizes, and a fourth-order Runge-Kutta method. By interfacing to SUNDIALS [11], state-of-the art implementations of multistep methods for ODEs and DAEs are available through Assimulo. The solvers CVode and IDA in SUNDIALS are the latest development branch of codes implementing multistep methods dating back to the 80s, also including DASSL. CVode is a variable-order, variable-step multistep algorithm for solving ordinary differential equations. CVode includes the Backward Differentiation Formulas (BDFs), which are suitable for stiff problems as well as Adams-Moulton formulae for highly accurate simulation of non-stiff systems. The solver IDA is a DAE integrator based on BDF.

Assimulo consists of mainly two parts. First, a skeleton of a simulation problem, which allows for defining all the necessary methods needed for simulation of a hybrid ODE and DAE, such as the right-hand-side, root functions and time-events. These skeletons are defined in the `Explicit_Problem` and `Implicit_Problem` classes for the ODE and DAE case respectively. The second part contains the actual integrators and interprets the information from the problem specification and performs the simulation.

In order to use Assimulo together with JModelica.org, the problem classes from Assimulo needed to be extended to allow for handling of how the models are defined in JModelica.org. In Figure 1 an overview of the implementation is shown.

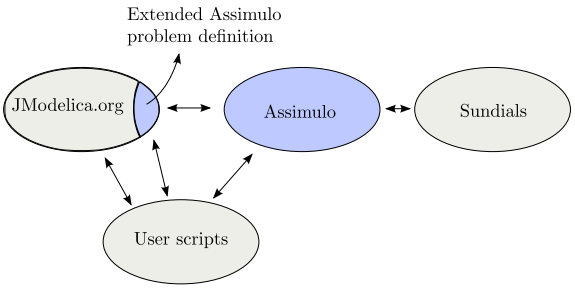

Figure 1: Overview of the interaction between JModelica.org, Assimulo and Sundials.

[1] http://www.jmodelica.org

2.3 FMI

The Functional Mock-up Interface defines a standard for model exchange consisting of a set of C functions and an XML schema for model interaction. The mathematical formalism upon which FMI is based is that of hybrid ordinary differential equations (ODEs), i.e. ordinary differential equations with some discrete states. FMUs are distributed as compressed files containing:

- A shared object file (DLL), containing implementations of the FMI functions. In addition, or alternatively, the FMU may contain the source code corresponding to the compiled DLL.
- An XML file, containing the variable definitions and meta information for the model, together with information about how it was generated. The file also contains value-references for the variables, which uniquely identifies variables and which are used when retrieving data from the model.
- Optional files containing bitmaps, documentations, tables etc.

The C functions contained in the FMU are typically called by a simulation environment, in order to perform a simulation experiment. The simulation environment needs then to be able to handle simulation of hybrid ODEs, which are often stiff. Also, the model meta data contained in the XML file needs to be loaded by the simulation environment in order to extract model information, e.g. variable names.

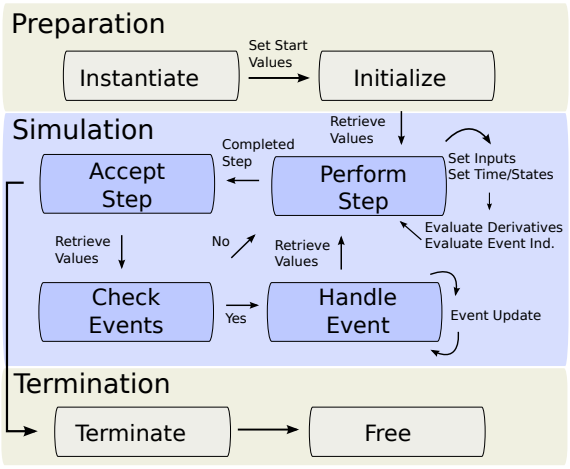

Figure 2: Overview of the calling sequence for an FMU.

In Figure 2, an overview of the calling sequence for an FMU is described. Prior to a simulation experiment, the model has to be instantiated. This includes extracting the files in the FMU, loading the DLL and XML files and calling the instantiation function available in the DLL. A model can be instantiated multiple times for which the function `fmiInstantiateModel` is provided. The model is then initialized by calling `fmiInitialize`.

Once the model is instantiated and initialized it can be simulated. The simulation is performed by updating time and states in the model, via `fmiSetTime` and `fmiSetContinuousStates`, and by calculating the derivatives via `fmiGetDerivatives`.

During the simulation, events are monitored via the functions `fmiGetEventIndicators` and `fmiCompletedIntegratorStep`. State-events are detected by looking for sign changes in the event indicators while step-events are checked in the model after calling the completed step function when an integration step was sucessfully completed.

At an event, the function `fmiEventUpdate` has to be called. This function updates and re-initializes the model in order for the simulation to be continued. Information is also given about if the states have changed values, if new state variables have been selected and information about upcoming time events.

To retrieve or set variable data during a simulation, value-references are used as keys. All variables are connected to a unique number defined by the export tool and provided in the XML-file. This number can then be used to retrieve information about variables via functions in the interface or can be used to set input values during a simulation. There are functions for setting and getting values for Real, Integer, String and Boolean values, `fmi(Get/Set)(Type)`.

After a simulation, memory has to be deallocated. The function `fmiTerminate` deallocates all memory that have been allocated since the initialization and the function `fmiFreeModelInstance` dispose of the model instance.

3 FMI Implementation

3.1 Export

The FMI specification standardizes an execution interface for hybrid ODEs. Modelica models, on the other hand, are usually translated into systems of index-1 DAEs. Therefore, a Modelica-based tool needs to transform DAEs into ODE form. Starting with the DAE

$$F(\dot{x}, x, w, u, p, t) = 0 \tag{3}$$

where $x \in \mathbb{R}^{n_x}$ are the states, $w \in \mathbb{R}^{n_w}$ are the algebraic variables, $u \in \mathbb{R}^{n_u}$ are the inputs, $p \in \mathbb{R}^{n_p}$ are the parameters and $t \in \mathbb{R}$ is the time. The objective of this transformation is to obtain an ODE on the form

$$\dot{x} = f(x,u,p,t). \tag{4}$$

Notice that this transformation is conceptual in most cases in the sense that the function f cannot be computed explicitly. Rather, a commonly employed strategy is to regard the function F as a system of non-linear system equations, where $\dot{x}$ and w are unknown and x, u, p and t are known. This strategy relies on the assumptions that i) start values for the states are available, possibly computed by solving a system of initialization equations, and ii) that the matrix

$$\begin{bmatrix} \frac{\partial F}{\partial \dot{x}} & \frac{\partial F}{\partial w} \end{bmatrix} \tag{5}$$

is square and has full rank. The latter condition means that a solution to the system of equations exists, at least locally, and holds if the DAE is of index 1, see [5] for a definition. Indeed, Modelica models commonly have index higher than 1, but it is here assumed that the index has been reduced by an index reduction algorithm, [12].

The DAE (3) is often highly structured and has in addition a sparse Jacobian, properties that can be exploited in order to perform the transformation more efficiently. A common approach for exploiting this structure is to decompose the DAE system into a sequence of smaller systems. This can be done by means of Tarjans algorithm, see [14, 8, 7] for details. Additional performance is gained typically as in typical cases several of the decomposed systems can be solved directly without the need to employ iterative Newton-type solvers. The usually few remaining non-linear systems of equations are solved during simulation by means of iterative techniques. In the FMUs generated by JModelica.org the KINSOL algorithm, which is part of the SUNDIALS suite, is used. In order to increase the robustness of the algorithm, KINSOL has been extended to support regularization to handle even the case of an initially singular Jacobian. See [16] for a detailed treatment. As a result of the presence of non-linear equation systems requiring iteration, ODE form (4) is conceptual rather implicit than explicit. Nevertheless, from the point of view of the simulation environment, the model is regarded as an explicit ODE, where the derivatives are computed given values of the parameters, the states, the inputs and the current time. The algorithm for computing the derivatives as outlined above is made available in the FMI function `fmiGetDerivatives`. In addition to providing a function for evaluating the derivatives of the ODE, hybrid constructs resulting in events need to be supported in the simulation run-time system in the FMU. Examples of hybrid constructs in Modelica are instantaneous equations (expressed as `when`-clauses) and relational expressions. During continuous integration, a set of event indicator functions, provided in the function `fmiGetEventIndicators`, are monitored. If a sign change of one of the indicator functions is detected, an event has occurred and the simulation environment then informs the FMU by calling the function `fmiEventUpdate`. From the point of view of the simulation environment, this procedure is straight forward, since many integration algorithms provide native support for localization of events. For the internal simulation run-time system in the FMU, the situation is more complicated. For example, one event may trigger other events, which requires an event iteration scheme to be employed. In JModelica.org, the simulation run-time system performs a fixed point iteration to resolve dependent events. For more information about hybrid constructs in Modelica and how they are handled in the context of simulation, see [4].

Before an FMU can be simulated, consistent initial conditions need to be found. In the FMUs generated by JModelica.org, this is done similarly to how the derivatives are computed. The BLT transformation is applied to the initialization system, consisting of the index-1 DAE augmented by initial equations, in order to obtain a sequence of equation systems, which can be solved for the states, the derivatives and the algebraic variables. Also in this case, the modified version of KINSOL, supporting regularization, is used.

3.2 Import

Integration of the FMI to JModelica.org requires a few key features to be present.

- Ability to decompress a compressed archive.
- Ability to couple functions provided in a DLL to Python.
- Ability to read and interpret an XML-file.

These features are provided by use of several Python packages such as `ctypes` [9], `lxml` and `zipfile`. `ctypes` enables loading of a dynamic linked library (DLL) into Python. The functions of the DLL can then be retrieved and defined in Python which enables

them to be called directly. The functions have to be explicitly defined together with their arguments and return arguments so that the correct type is returned back to the DLL. `lxml` provides methods for handling of XML-files, such as querying and traverse complete files. This feature was already available in JModelica.org as an extended FMI XML format, which is used to handle generated model data from JModelica.org. Finally, `zipfile` offers methods for extracting information from compressed directories, such as an FMU.

The FMI import implementation in JModelica.org centers around a Python class, `FMUModel` where the constructor takes as input an FMU and performs the necessary tasks to enable manipulation and simulation of the FMU. The constructor also calls the XML import interface which reads the complete XML-file and populates data structures with information about all model variables including start-values, value-references, aliases and types. The interface consists of a raw mapping of the functions defined in the FMI specification easily available and accessible from Python. The methods are named according to the specification with a leading underscore. For example, the FMI function `fmiGetDerivatives(...)` corresponds to the following method in our Python class, `FMUModel`,

```
FMUModel._fmiGetDerivatives(...)
```

Providing a complete mapping to the original FMI functions enables users to create scripts tailored to their specific purposes.

JModelica.org also provides a Pythonic and object-oriented connection to an FMU with high-level methods for setting and retrieving values. We demonstrate this by computing the derivatives at time t and state y using the FMU:

```
FMUModel.time = t
FMUModel.continuous_states = y

rhs = FMUModel.get_derivatives()
```

The high-level methods propagate the information and call the underlying FMU functions.

To retrieve or set values of an arbitrary variable, instead of looking for the value reference, the name is used in the call to set/get methods:

```
FMUModel.get('der(x)')
FMUModel.set('g', 9.81)
```

The methods retrieve information about the variable, type and value reference from the XML data and then call the underlying FMU functions.

In addition to the high-level methods in JModelica.org, a connection to the simulation package Assimulo is also offered. The connection is based on extending Assimulo's problem class in the same way as models generated from JModelica.org are interfaced, see Figure 1.

As a problem class in Assimulo is just a skeleton of a model together with its methods, interfacing is just a matter of providing the information.

In order to connect the calculation of the derivatives of an FMU to Assimulo, the right-hand side function (*rhs*) must correspond to:

```
class FMIODE(Explicit_Problem):
    def f(t,y):
        #Moving data to the model
        FMUModel.time = t
        FMUModel.continuous_states = y

        #Evaluating the rhs
        rhs = FMUModel.get_derivatives()

        return rhs
```

where `Explicit_Problem` is Assimulo's skeleton class of an ODE problem. If there are any inputs, they are calculated and provided to the model as well.

Events are monitored and detected by providing methods for the state-, step- and time-events which in Assimulo correspond to implementing the methods `state_events`, `time_events` and `completed_step`. They are implemented similarly to the calculation of the derivatives.

If an event is detected, either be it state, time or step, a call is made to a method called `handle_event` in the problem which has been implemented so that it is directed to the FMI function `fmiEventUpdate`.

These methods provide a full-fledged connection to Assimulo and to state of the art numerical integrators. The current implementation is fully functional and relies on Sundials CVode solver.

4 Examples

4.1 The Van der Pol Oscillator

The Van der Pol oscillator stated in Equation (6) is used here to demonstrate the functionality of exporting an FMU using the JModelica.org platform and also to demonstrate the import process. In addition, it will be shown how the same problem is solved using Sundials CVode solver. The problem is also solved for a set of initial values to demonstrate how to run multiple simulations in a single sweep. Simulation of per-

turbed values can be useful for analyzing and evaluating model sensitivity with respect to uncertainty in physical parameters or initial conditions. The dynamics of the Van der Pol oscillator is given by

$$\frac{d^2x}{dt^2} - (1-x^2)\frac{dx}{dt} + x = 0. \quad (6)$$

The problem can be described in the Modelica language by introducing the state variables $x_1 = x$ and $x_2 = \frac{dx}{dt}$, which gives

$$\begin{aligned} \frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= (1-x_1^2)x_2 - x_1. \end{aligned} \quad (7)$$

The Modelica specification for the Van der Pol oscillator is given in Listing 1, where the initial values are set to $x_1(t_0) = x_{1_0}$ and $x_2(t_0) = x_{2_0}$.

```
model VDP
    // The parameters
    parameter Real x1_0 = 1.0;
    parameter Real x2_0 = 0.0;

    // The states
    Real x1(start = x1_0);
    Real x2(start = x2_0);

  equation
    der(x1) = x2;
    der(x2) = (1 - x1^2) * x2 - x1;
end VDP;
```

Code Listing 1: The Van der Pol oscillator described in Modelica.

Creation of an FMU from a Modelica model consists of several steps. Primarily, the equations have to be translated and possibly manipulated by for instance an index reduction algorithm to produce a source code file, in our case a *C* file. Variable data needs to be extracted and populated into an XML structure. In JModelica.org, these steps are collected in a Python method, `compile_fmu`, which is demonstrated below.

```
from jmodelica.fmi import compile_fmu

fmu_name = compile_fmu("VDP", "VDP.mo")
```

The commands produce an FMU of the Modelica model `VDP` located in `VDP.mo` which can be distributed to any software supporting the Functional Mock-up Interface.

Steps for simulating the Van der Pol oscillator are similarly straight forward where the FMU first must be loaded into JModelica.org.

```
from jmodelica.fmi import FMUModel

model = FMUModel(fmu_name)
```

`FMUModel` takes the name of an FMU, in our case `VDP.fmu` as an argument in the constructor. A number of internal steps are then taken when a model is loaded. First, the FMU is unzipped and the XML data together with the binary containing the model functions are extracted. Second, the functions in the model binary is connected to Python and instantiated according to the FMI specification.

Simulation of the Van der Pol oscillator is then performed by the `simulate` method.

```
result = model.simulate(final_time=10)
```

The Van der Pol oscillator is simulated from $t = 0.0$ to $t = 10.0$ using default options. In Listing 2, the runtime statistics is shown, which is printed in the prompt after a simulation, when solving the Van der Pol oscillator using JModelica.org and Assimulo together with the solver CVode (BDF).

```
Final Run Statistics: VDP

 Nbr of Steps                : 148
 Nbr of Function Evaluations : 208
 Nbr of Jacobian Evaluations : 3
 Nbr of F-Eval During Jac-Eval : 6
 Nbr of Root Evaluations     : 0
 Nbr of Error Test Failures  : 11
 Nbr of Nonlinear Iterations : 204
 Nbr of Nonlinear Conv Failures : 0
```

Code Listing 2: Simulation statistics of the Van der Pol oscillator.

The result object returned from a simulation makes the simulation data and simulation trajectories easily available for either visualization or manipulation.

```
x1   = result["x1"]
x2   = result["x2"]
time = result["time"]
```

For visualization, the package Matplotlib [10] is used.

```
import matplotlib.pyplot as plt

plt.figure(1)
plt.plot(time,x1,time,x2)
plt.xlabel("Time [s]")
plt.legend(("x1", "x2"))
plt.title("Van der Pol")
plt.show()
```

The resulting trajectories for x and $\frac{dx}{dt}$ are shown in Figure 3.

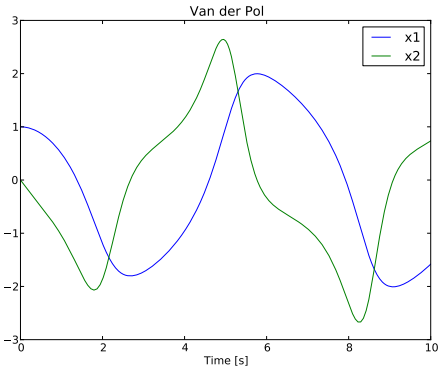

Figure 3: Solution of the Van der Pol oscillator showing x and $\frac{dx}{dt}$.

An extension of performing a simulation of the Van der Pol oscillator is to perform a parameter sweep. A parameter sweep is executed by varying a parameter and performing a simulation for each value. In our case, we consider holding $x_1(t_0)$ fixed while varying $x_2(t_0)$ in the interval $[-3,3]$. The resulting script is shown below.

```
import numpy

nbr_points = 11
x1_0 = 0.0
x2_0 = numpy.linspace(-3.0, 3.0, 11)

for i in range(nbr_points):

    model.set('x1_0',x1_0)
    model.set('x2_0',x2_0[i])

    result = model.simulate(final_time=20)

    x1=result['x1']
    x2=result['x2']

    plt.plot(x2, x1,'b')

plt.title("Van der Pol")
plt.ylabel("x1")
plt.xlabel("x2")
plt.grid()
plt.show()
```

First, the initial values are defined as $x_{1_0} = 0.0$ and x_{2_0} being a uniformly distributed array in the interval $[-3,3]$ with 11 values. Second, the simulation command is iterated over the initial values which are set with the `model.set()` method. In each iteration, the model is simulated from 0.0 to 20 seconds and the solution trajectories for x_1 and x_2 are retrieved and plotted. The resulting phase plot is shown in Figure 4.

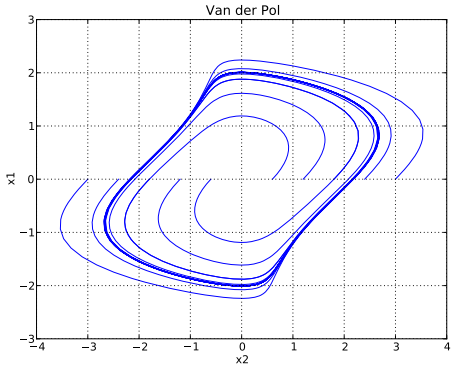

Figure 4: Solution of the Van der Pol oscillator showing a phase plot of x and $\frac{dx}{dt}$.

4.2 A Robot Model

The Full Robot from the multibody examples in the Modelica Standard Library will be used to demonstrate that the implementation can handle industrially relevant problems. The example is also intended to demonstrate that JModelica.org is able to simulate models generated by third party software supporting the FMI specification. In Figure 5, the diagram layer of the robot is depicted.

The robot consists of brakes, motors, gears and path planning. The model consists of 36 continuous states and around 700 algebraic variables together with 98 event indicators and a few thousand constants/parameters.

The FMU was generated using Dymola 7.4 [6].

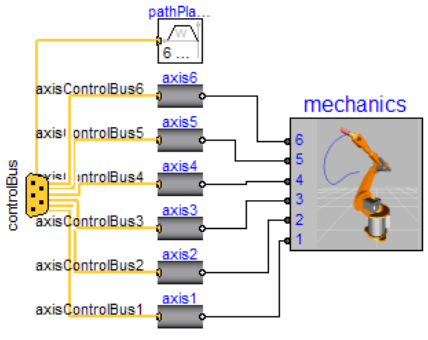

Figure 5: Overview of the Full Robot.

Simulation of the robot is performed following the same steps as in the Van der Pol example by first loading the model[2] and then invoking the simulate method.

[2]The name of the FMU has been shortened to save space in the article.

```
from jmodelica.fmi import FMUModel

robot=FMUModel('Modelica...fullRobot.fmu')
result = robot.simulate(final_time=1.8)
```

The robot is simulated from $t = 0.0$ to $t = 1.8$ using default options. In Listing 3, the run-time statistics is shown.

```
Final Run Statistics: Modelica_..._fullRobot

 Nbr of Steps                 : 1834
 Nbr of Function Evaluations  : 2386
 Nbr of Jacobian Evaluations  : 65
 Nbr of F-Eval During Jac-Eval : 2340
 Nbr of Root Evaluations      : 2223
 Nbr of Error Test Failures   : 42
 Nbr of Nonlinear Iterations  : 2202
 Nbr of Nonlinear Conv Failures : 0
```

Code Listing 3: Simulation statistics of the Full Robot using JModelica.org.

Trajectories for the joint velocities are extracted from the result object and visualized using Matplotlib in the same way as in the Van der Pol example.

```
dq1  = result['der(mechanics.q[1])']
dq6  = result['der(mechanics.q[6])']
time = result['time']

import matplotlib.pyplot as plt

plt.plot(time,dq1,time,dq6)
plt.legend(['der(mechanics.q[1])',
            'der(mechanics.q[6])'])
plt.xlabel('Time (s)')
plt.ylabel('Joint Velocity (rad/s)')
plt.title('Full Robot')
plt.show()
```

The result of the simulation is shown in Figure 6.

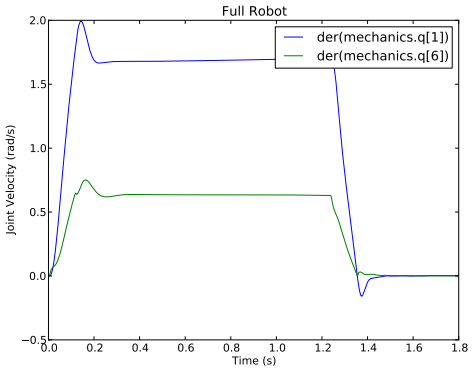

Figure 6: Solution of the joint velocities `q[1]` and `q[6]`.

4.3 Verification of the Full Robot

For verification of the result, the Robot is simulated using Dymola 7.4 and the trajectories are compared. In Listing 4, the run-time statistics is shown when simulating the robot using Dymola and the solver DASSL.

```
Number of result points        : 1001
Number of GRID points          : 1001
Number of (successful) steps   : 1482
Number of F-evaluations        : 10562
Number of H-evaluations        : 2794
Number of Jacobian-evaluations : 353
```

Code Listing 4: Simulation statistics of the Full Robot using Dymola.

In Figure 7 the resulting comparison between the simulation result from JModelica.org and Dymola is shown. The simulations are both performed with a relative tolerance of 10^{-4} and the absolute tolerance (in JModelica.org) was set to 0.01 times the relative tolerance times the nominal values of the continuous states, $0.01 \cdot \text{rtol} \cdot \text{nominal}$. The number of output points is set to 1000 in both cases.

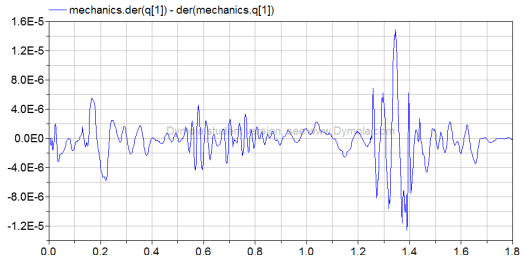

Figure 7: Difference of the state mechanics.der(q[1]) (joint velocity) between JModelica.org and Dymola. The model is simulated using default tolerances together with 1000 output points.

Timing results are shown in Table 1. Dymola native corresponds to simulating the robot directly from the Modelica standard library without using the FMI. Dymola FMU corresponds to loading a generated FMU into Dymola and performing a simulation. JModelica.org corresponds to the actual integration time and the time for I/O operations for storing the result.

Platform	Simulation Time
Dymola Native	1.26 s
Dymola FMU	4.97 s
JModelica.org	3.49 s

Table 1: Benchmark results of the Full Robot.

4.4 Twin Evaporator

The TwinEvaporatorCycle model from the commercial Air Conditioning library is demonstrated to show that the implementation can handle large models. It is a model of an A/C-cycle from a typical European premium car with individual front and rear climate zones. The model describes the cooling performance of the refrigerant cycle, and includes the compressor, front condenser, expansion control valve and two evaporators. The front evaporator is located under the dashboard and cools air for the driver and front seat passenger, while the rear evaporator is located between the seats and cools the air that flows to the back seat. The model diagram, shown in Figure 8, also includes dynamical display components for simulation analysis, e.g. the pressure-enthalpy diagram of the refrigerant R134a.

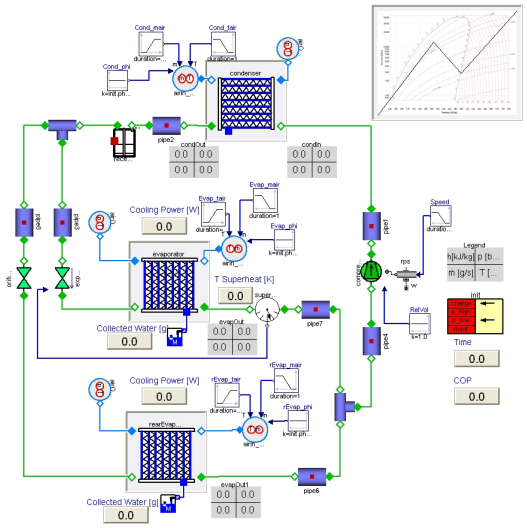

Figure 8: Overview of the Twin Evaporator.

The model consists of 130 states together with 1090 event indicator functions. The number of parameters and constants is close to 20.000, resulting in a model description file of 170.000 lines.

The model was simulated using JModelica.org from $t = 0.0s$ to $t = 180.0s$ with a relative tolerance of $rtol = 10^{-5}$ and an absolute tolerance corresponding to $atol = 0.01 \cdot rtol \cdot nominal$, as in the Full Robot example. The simulation statistics from JModelica.org can be found in Listing 5.

```
Final Run Statistics: ..._TwinEvaporatorCycle

 Nbr of Steps                 : 827
 Nbr of Function Evaluations  : 1434
 Nbr of Jacobian Evaluations  : 39
 Nbr of F-Eval During Jac-Eval : 5070
 Nbr of Root Evaluations      : 830
 Nbr of Error Test Failures   : 44
 Nbr of Nonlinear Iterations  : 1422
 Nbr of Nonlinear Conv Failures : 8
```

Code Listing 5: Simulation statistics of the TwinEvaporatorCycle using JModelica.org.

The model was also simulated with Dymola using the same options of the tolerances. The result can be found in Listing 6. Note that the Dymola simulation was performed on the Modelica model, not the FMU.

```
Number of result points          : 1004
Number of GRID points            : 1001
Number of (successful) steps     : 302
Number of F-evaluations          : 3859
Number of H-evaluations          : 1303
Number of Jacobian-evaluations   : 136
```

Code Listing 6: Simulation statistics of the TwinEvaporatorCycle using Dymola.

In Table 2, timing results of simulations with both JModelica.org and Dymola are listed. The simulation time corresponds to the actual integration time, including writing the result. The total time also includes time for the initialization.

Platform	Simulation Time	Total Time
Dymola Native	57.6 s	79 s
Dymola FMU	125 s	175 s
JModelica.org	90 s	130 s

Table 2: Benchmark results of the Twin Evaporator.

The execution time measurements indicate that the performance of the JModelica.org FMU import is on par with state of the art commercial tools.

5 Limitations

While fully FMI compliant FMUs are generated by JModelica.org, both in terms of the DLL functions and in terms of the XML files, the Modelica language compliance of the compiler front-end is not complete. The support is continuously improving and recent additions include support for hybrid and sampled systems.

The FMI standard specifies how several FMUs can be simulated jointly by connecting their inputs and outputs. One application of this feature is to include FMUs in Modelica models, and another application is co-simulation. These features remains to be implemented.

6 Summary

In this paper, an implementation of the Functional Mock-up Interface for Model Exchange in the JModelica.org platform has been presented. The export functionality enables users to generate FMI compliant models, FMUs from Modelica models and to use them in different FMI compliant tools.

The FMU import is based on Python. Models can be imported into the Python environment, where FMUs are represented by objects of a Python class. The FMU model class provides the user with a one to one mapping of the FMI functions, as well as convenient high-level methods for setting parameter values and simulating models.

This paper also shows that simulation of Functional Mock-up Units using JModelica.org produce results comparable to those produced by a state of the art commercial tool.

Future extensions include support for sparse Jacobians in the FMI specification.

This work was partially funded by the ITEA2 project OPENPROD.

References

[1] Johan Åkesson. Optimica—an extension of modelica supporting dynamic optimization. In *In 6th International Modelica Conference 2008*. Modelica Association, March 2008.

[2] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749, November 2010. Doi:10.1016/j.compchemeng.2009.11.011.

[3] Christian Andersson. A new Python-based class for simulation of complex hybrid daes and its integration in jmodelica.org. Master's thesis, Department of Mathematics, Lund University, Sweden, 2010.

[4] Willi Braun, Bernhard Bachmann, and Sabina Pross. Synchronous events in the OpenModelica compiler with a petri net library application. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, 2010.

[5] K.E. Brenan, S.L. Campbell, and L.R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, 1996.

[6] Dassault Systemes. Dymola web page, 2010. `http://www.3ds.com/products/catia/portfolio/dymola`.

[7] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund University, Sweden, May 1978.

[8] Jan Eriksson. A note on the decomposition of systems of sparse non-linear equations. *Bit Numerical Mathematics*, 16(4):462–465, 1976. DOI: 10.1007/BF01932730.

[9] Python Software Foundation. ctypes: A foreign function library for Python, 2009. `http://docs.python.org/library/ctypes.html`.

[10] J. Hunter, D. Dale, and M. Droettboom. Matplotlib: Python plotting, 2010. `http://matplotlib.sourceforge.net/`.

[11] Center for Applied Scientific Computing Lawrence Livermore National Laboratory. SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers), 2009. `https://computation.llnl.gov/casc/sundials/main.html`.

[12] Sven Erik Mattsson and Gustaf Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM J. Sci. Comput*, 14(3):677–692, May 1993.

[13] T. Oliphant. Numpy Home Page, 2009. `http://numpy.scipy.org/`.

[14] R.E Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

[15] The Modelica Association. The Modelica Association Home Page, 2010. `http://www.modelica.org`.

[16] Johan Ylikiiskilä, Johan Åkesson, and Claus Führer. Improving Newton's method for initialization of Modelica models. In *8th International Modelica Conference*, 2011.