ISSN 0280-5316
ISRN LUTFD2/TFRT--5869--SE

Implementation of the Functional Mock-up Interface in Matlab and Simulink

Bengt-Arne Andersson

Department of Automatic Control
Lund University
December 2010

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
December 2010

Document Number
ISRN LUTFD2/TFRT--5869--SE

Author(s)
Bengt-Arne Andersson

Supervisor
Johan Åkesson Automatic Control, Lund
Claes Führer Mathematical Sciences, Lund
(Examiner)

Sponsoring organization

Title and subtitle
Implementation of the Functional Mock-up Interface in Matlab and Simulink (Implementation aav FMI i Matlab och Simulink)

Abstract
New products on the market are likely to be simulated in a computer sometime during the development process. The environment for which the physical model of the product is developed may not always be the optimal for control simulations of the model. To be able to export models from one environment to another a common model definition must be defined.
The Functional Mock-up Interface, FMI, provides such a model definition and makes it possible to incorporate models from different environments together. In this thesis we will witness a successful implementation of the interface for model exchange, FMI, into the well known MATLAB and Simulink environment. Simulink is widely used in industry to develop control systems but not that used for physical modelling. It is therefore of great interest to be able to simulate models created from other physical modelling environments in to Simulink. A block is developed in Simulink and a user interface in MATLAB such that models created according to the FMI standard can be simulated.
The FMI is a standard for solving ODEs with events. The thesis discusses the most essential parts of the FMI standard. Events may be discontinuities that the ODE solver needs to take special care of and is therefore discussed in more detail.
In Simulink an S-function block is used with a GUI developed for the user to easily configure the model. The MATLAB interface is developed using MEX functions and is discussed based on how MATLAB's ODE solvers can be used to simulate a model. MEX functions are MATLAB's way to incorporate C, C++ and Fortran code. The FMI standard models consists of DLL functions that enforce the use of MEX functions.The implementations are verified to be correct by comparison of simulation results from different environments such as Dymola and JModelica.org. A comparison of simulation times and the number of function evaluations are also done where we can see that the S-function and the MEX interface performs on a similar level as the other simulation environments.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
60

Recipient's notes

Security classification

Preface

First of all would like to thank my supervisors Johan Åkesson at the Lund University at the Department of Automatic Control and Magnus Gäfvert at Modelon for the guidance and support of this thesis and of curse for giving me the chance to experience a real program development. I would also thank all the people on Modelon that have been contributing to this thesis, especially Tove, Christan and Jesper in the JModelica team and Hubertus for giving me lessons that I never though I would be given in undocumented Dymola.

Lund, December 2010

Bengt-Arne

Contents

1 Introduction

New products on the market are likely to be simulated in a computer sometime during the development process. The environment for which the physical model of the product is developed may not always be the optimal for control simulations of the model. To be able to export models from one environment to another a common model definition must be defined.

The Functional Mock-up Interface, FMI, provides such a model definition makes it possible to incorporate models from different development environments together. In Figure 1 we see how a typical model of a complex system may look like. It is a model of an air conditioning system in a block diagram representation in the well established modelling and simulation environment Dymola.

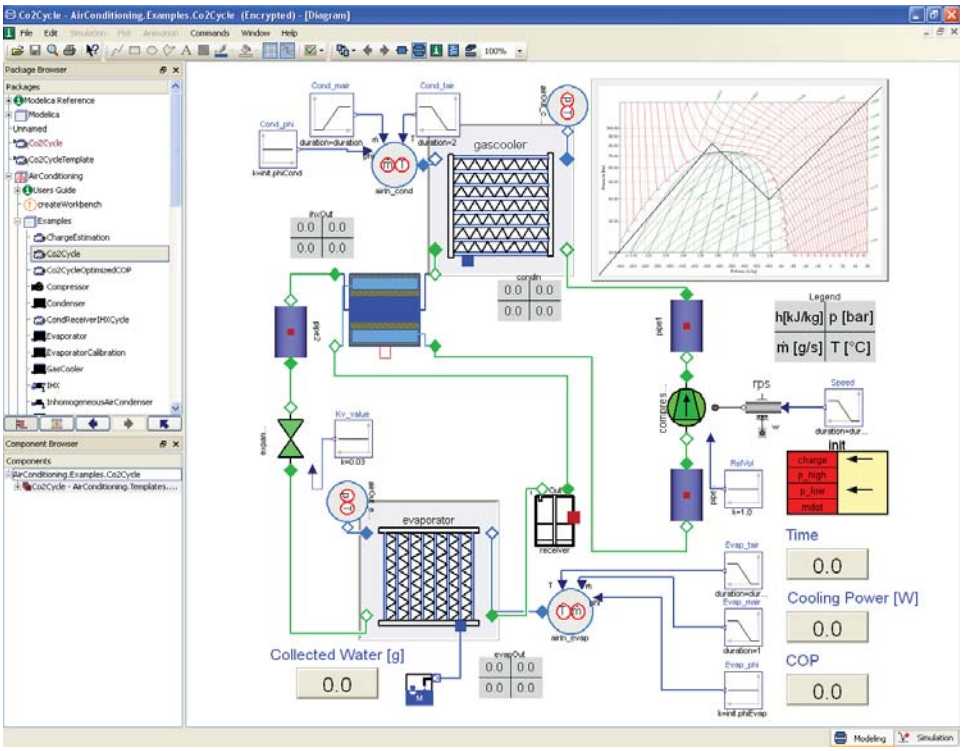

Figure 1: Block diagram model of an air conditioning system in Dymola

In this thesis we will witness a successful implementation of the interface for model exchange, FMI, into the well known MATLAB and Simulink environment. Simulink is widely used in industry to develop control systems but not that used for physical modelling. It is therefore of great interest to be able to simulate models created from other physical modelling environments

in to Simulink.

In Simulink an S-function block is developed with a GUI for simulating and interacting with the model. In MATLAB, different functions are developed for both invoking the so called FMI functions separately from the workspace and simulating a model.

A test suite of models that covers most of the FMI functionalities is simulated and compared with the corresponding results from Dymola and JModelica.org.

The Functional Mock-up Interface is a new model exchange standard developed from results of the ITEA2 project MODELISAR and was released 26.01.2010 [7]. When this is written, there are about 10 different modelling environments supporting the interface. The FMI is defined for simulations solving ordinary differential equations in state space form (ODE) with events. Differential algebraic equations(DAE) are not yet supported but might be in future releases of the FMI. For now, DAEs need to be treated within the model.

2 Background

We will describe the basic elements used for the implementation of the FMI in this section. These are the simulation environments MATLAB and Simulink for which the implementation takes place, the Function Mock-up Interface, FMI, for model exchange to be implemented, and a Software Development Kit, SDK from QTronics [4]. More detailed descriptions of the topics will be given continuously where it is needed in the rest of the thesis.

2.1 MATLAB

MATLAB is one of most used tools for simulations and fast software development. It is therefore attractive to support the FMI. It is a high-level language and an interactive environment that primarily is used for mathematical and technical calculations. MATLAB has a wide range of tools for example data analysing, algorithm developing and perform calculations. MATLAB supports many applications such as control design, signal and image processing, finance modelling and analysis. It is a so-called script language that is interpreted while it is executed. It also supports other compiled languages such as C,C++,Fortran in a special format, so called MEX. During this thesis MATLAB 7.5(R2007b) was used during the development in Windows XP32. It was also tested in MATLAB 7.10(R2010a) with Windows Vista32.

2.2 Simulink

Simulink is a tool for modelling and simulating dynamical systems. It offers tight integration with the MATLAB environment and can either drive MATLAB or be scripted from it. Simulink has a graphical block diagram interface with blocks and lines which together describe a system of equation. In Figure 2 we have a general block with an input signal that the block may use to calculate the output. And inside the block it self, it can have temporal variables such as previous values, so called states.

We give an example of how the block diagram interface works in Figure 3. The leftmost block produces a sinusoidal wave signal with amplitude 1 and frequency $\frac{1}{2\pi}$. The feedback signal from the output is then subtracted from the signal. The result is then visualised with a *Scope* block to the rightmost. The blocks are often dragged and dropped in the model workspace from the so called *Library Browser* where all the blocks are found.

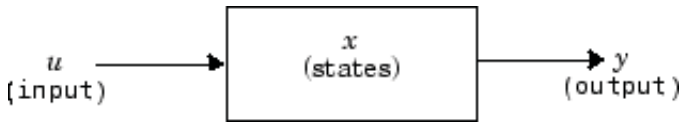

Figure 2: A graphical representation of a block with states in Simulink

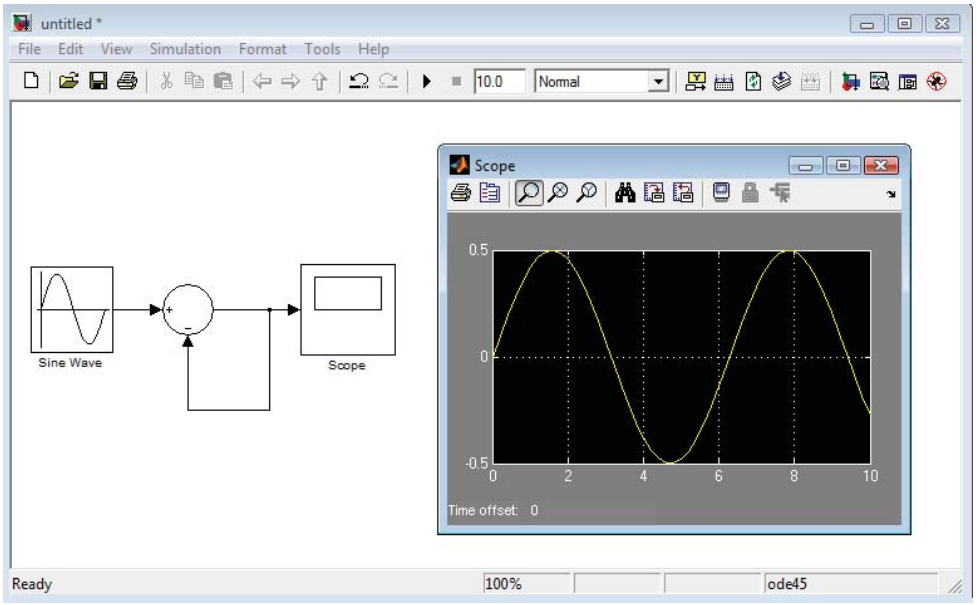

Figure 3: Coupled blocks representing a system in Simulink's model workspace. The input signal is $\sin(t)$ and the output signal should be $0.5 \cdot \sin(t)$.

2.3 Functional Mock-up Interface for Model Exchange

The Functional Mock-up Interface, FMI, defines an interface for how a Functional Mock-up Unit, FMU, is called to create a model instances and simulate it. It may be self-integrating (co-simulation) or it may require the simulator to perform numerical integration. Models are distributed in zip-files with the file name extension *.fmu. It contains a Dynamic-link library, DLL, and a model description in XML format. The xml-file contains necessary data needed for calling the FMI functions but also description of the whole model and its variables. There may be additional model data but it should not be necessary for simulation.

2.4 FMI SDK

A FMI software development kit, SDK, from QTronics(FMU SDK 1.0.1) was used during the development. The FMU extraction relay's on the SDK features for unzipping the FMU, parsing the model description file and how the FMI functions are invoked from the DLL file. The unzipping and XML-parsing is performed with two external programs called 7-zip and Expat.

2.4.1 ZIP files

ZIP is a file format for data compression. A ZIP file contains files that have been compressed to reduce the files size, using a compression algorithm. The FMUs are ZIP files with the file extension *.fmu. To restore the data to its origin from a ZIP file , the file needs to be *unzipped*.

2.4.2 XML files

Extensible Markup Language, XML, is a defined set of rules for encoding documents. An XML-file is in textual data format designed for representation of arbitrary data structures. The XML file mentioned above in the FMU, describes the model according to the XML scheme defined by the FMI standard. The XML scheme describes for instance the structure and constraints of how the data in the XML file shell be defined.

2.4.3 DLL files

Dynamic-link library, DLL, is Microsoft's implementation of shared library. Shared library is roughly described as a file containing functions that can be shared among different programs. The FMI functions are found in such a DLL file. How these functions are set up and invoked is found in the FMI documentation [7].

3 FMI and the FMI SDK

We will highlight and describe the most essential data of the FMI documentation for better understanding and discussions in later sections. A lot of the text here is reproduced from the FMI documentation [7].

3.1 Events

The FMI interface is designed for models to be simulated by solving an ODE with events (so called hybrid ODEs) numerically. This type of a system with events (often discontinuities) is called piecewise continuous system [7].

A mathematical description of events defined by the FMI is given here. An event can occur at time instants $t_0, t_1, ...t_n$ where $t_i < t_{i+1}$ [7]. Let us define the state of an ODE with $x(t)$ as the continuous state and $m(t)$ as the time-discrete state.

- $x(t)$ is a vector of real numbers and is continuous function in time inside each interval $t_i \leq t < t_{i+1}$ where $t_i = \lim_{\varepsilon \to 0} t_i + \varepsilon$, i.e., the right limit to t_i (note, $x(t)$ is continuous between the right limit to t_i and the left limit to t_{i+1} respectively) [7].
- $m(t)$ is a set of real, integer, logical and string variables that is constant inside each interval $t_i \leq t < t_{i+1}$. $m(t)$ changes values only at events [7].

At every event instance t_i, variables might be discontinuous and therefore have two values at the same time instance, a left and a right limit [7]. $x(t_i)$ and $m(t_i)$ is defined to be the right limit. In Figure 4 the different states can be seen.

FMI defines events in three kinds. These events are all triggered from the simulation environment [7]. The events are defined as follows [7]:

- **Time event** - At a predefined time instance $t_i = T_{next}(t_{i-1})$ that was defined at the previous event instance t_{i-1} either by the FMU, or by the environment of the FMU due to discontinuous changes of an input signal u_j at t_i.
- **State event** - At a time instance, where an event indicator $z_j(t)$ changes its domain from $z_j > 0$ to $z_j \leq 0$ or vice versa(see Figure 5). More precisely: An event $t = t_i$ occurs at the smallest time instance "min t" with $t > t_i$ where "$(z_j(t) > 0) \neq (z_j(t_{i-1}) > 0)$". All event indicators are piecewise continuous and are collected together in one vector of real number $z(t)$.

- **Step event** - At every completed step of an integrator, the FMI function fmiCompletedIntegratorStep must be called. An event occurs at this time instance, if indicated by the return argument callEventUpdate. Such an event is used to change continuous states because the previous states where not longer numerical suitable.

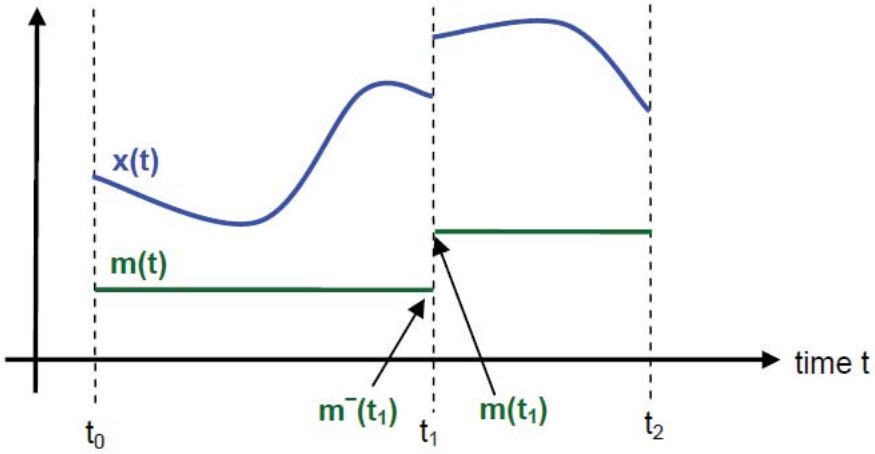

Figure 4: Piecewise-continuous states of an FMU: time-continuous (x) and time-discrete (m). The Figure is from the FMI documentation [7].

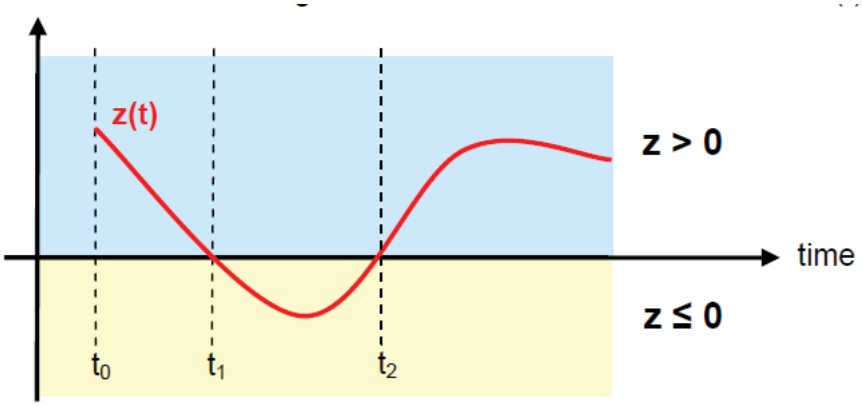

Figure 5: An event occurs when the event indicator changes its domain from $z > 0$ to $z = 0$ or vice versa. The Figure is from the FMI documentation [7]

3.2 Principle data flow

In Figure 6 we see an overview of the data flow between an FMU and a simulator. The red arrows provide the data needed by the FMU and the blue

arrows provide data needed by the simulator. For example when an FMU model is simulated, it sometimes needs input data u from the simulation environment to calculate the state derivatives $\dot{x}$. The state derivatives are used by the solver to integrate the ODE. A thorough description of the calling sequence between the FMU and the simulation environment is given in section 3.5.

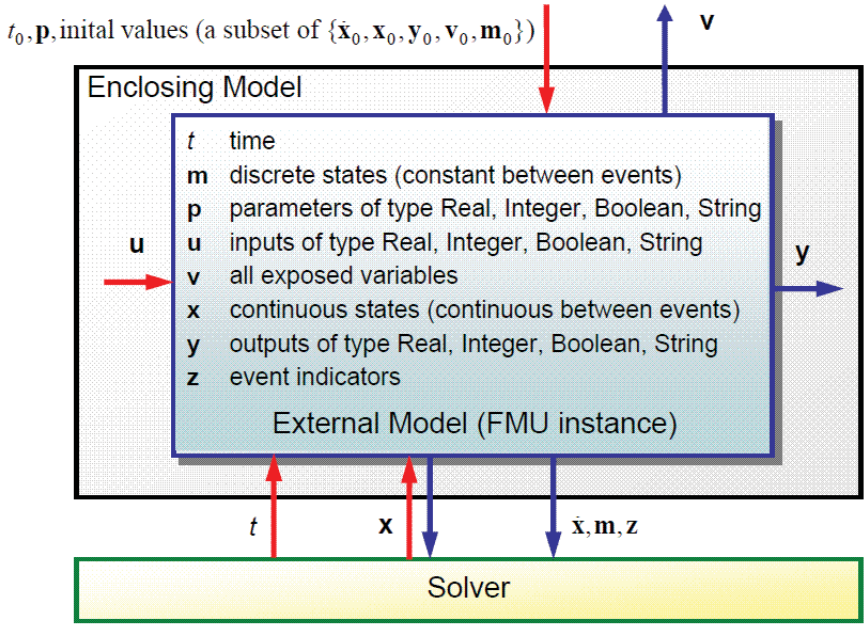

Figure 6: This shows the data flow between an FMU and the simulation environment. The figure is from the FMI documentation [7].

3.3 FMI functions

The FMI standard is defined to be a platform dependent definition. This means that the C types(int, float, double, etc.) are defined by the platform the model is simulated on. This forces the model to be compiled with same types as the simulator's platform is using and not the other way around.

We will now describe the FMI functions shortly. These functions are in some sense the ones describing the FMU model. We will not give the argument list of these functions but almost all of them return a status flag. This flag tells the simulation environment how successful the function call was and then appropriate actions may be taken [7].

- **fmiInstantiateModel** - This function instantiates a model and returns a model instance. This model instance contains the information

needed to process the model equations and is passed as an argument in many other FMI functions. The simulator environment needs to provide callback functions to the model, for memory allocation and logging. These callback functions are set during the instantiation. The FMI proved some associated functions like freeing the model instance and configure the logging.

- **fmiSetTime** - This function sets the new time for the model instance. It updates all the time dependent variables for this time instance.
- **fmiSetContinuousStates** - This function sets the new continuous state vector x. It updates all the state dependent variables.
- **fmiCompletedIntegratorStep** - This function must be called after every completed integrator step. It may notify the simulator environment to react to an event(step) by calling fmiEventUpdate.
- **fmiSetXXX and fmiGetXXX** - Every variable has a unique number associated with respect to the type(Integer, Real, Boolean, String), a so called *value reference*. The fmiSetXXX and fmiGetXXX is called with the variables *value reference* and returns or set the value, where XXX is one of the types.
- **fmiInitialize** - This function initializes the model and can only be called once for one model instance. All numerical algorithms inside the model is configured with tolerances passed as arguments to this function. It may notify the simulator environment to react on an event(time) by calling fmiEventUpdate.
- **fmiGetDerivatives** - This function computes and returns the state derivatives at the current time instant and for the current states. The elements of the returning derivatives vector are in same order as the continuous state vector(e.g. derivatives[2] corresponds to the derivative of the continuous state x[2]).
- **fmiGetEventIndicators** - This function computes and returns the event indicators z at the current time instant and states. The FMU must guarantee that at an event indicator $z_j \neq 0$.
- **fmiEventUpdate** - This function returns once new consistent states have been found and the integrator can continue. It can be configured such that it returns for every event iteration that is performed internally. In that case it notifies the simulator environment to continue call the fmiEventUpdate until it has converged. This function may not be called backwards in time.
- **fmiGetContinuousStates** - This function returns the new continuous state vector x after initialization or an event.

- **fmiGetNominalContinuousStates** - This function returns the nominal values of the state vector x.
- **fmiGetStateValueReferences** - This function returns the value references to the continuous state vector x.
- **fmiTerminate** - This function terminate the model evaluation and frees all memory allocations made since fmiInitialize was called.

3.4 FMI model description schema

All the model data, except from the model equations, are given in an XML-file. The scheme for this XML-file is well defined by the FMI [7]. This data is used to set up the environment such as inputs to the FMU and to change parameters for example. We give an extraction of the XML-schema to give the reader a feeling of the content that can be found.

- **fmiModelDescription**
  - **modellIdentifier** - Short class name
  - **numberOfContinuousStates** - unsigned integer
  - **numberOfEventIndicators** - unsigned integer
  - **ModelVariables** - List of ScalarVariables
- **ScalarVariable**
  - **name** Unique variable name.
  - **valueReference** Variable identifying number
  - **variability** Defines when the value of the variable changes
  - **causality** Defines how the variable is visible from the outside of the model
  - **One of: Real,Integer,Boolean,String,Enumeration** Type of element
- **Real**
  - **declaredType** - Providing default attributes
  - **unit** - Unit of the variable, e.g., "N.m"
  - **min** - Minimum value
  - **max** - Maximum value
  - **start** - Initial value

– **fixed** - Defines if the initial value is the same after initialization or not

3.5 FMI function calling sequence

In the FMI documentation a calling sequence for the FMI functions are given. This is the calling sequence that we want to map into the simulation environment which has its own calling sequence of callback functions. The FMI calling sequence is given in Figure 7 and a short description found in the FMI documentation [7] follows.

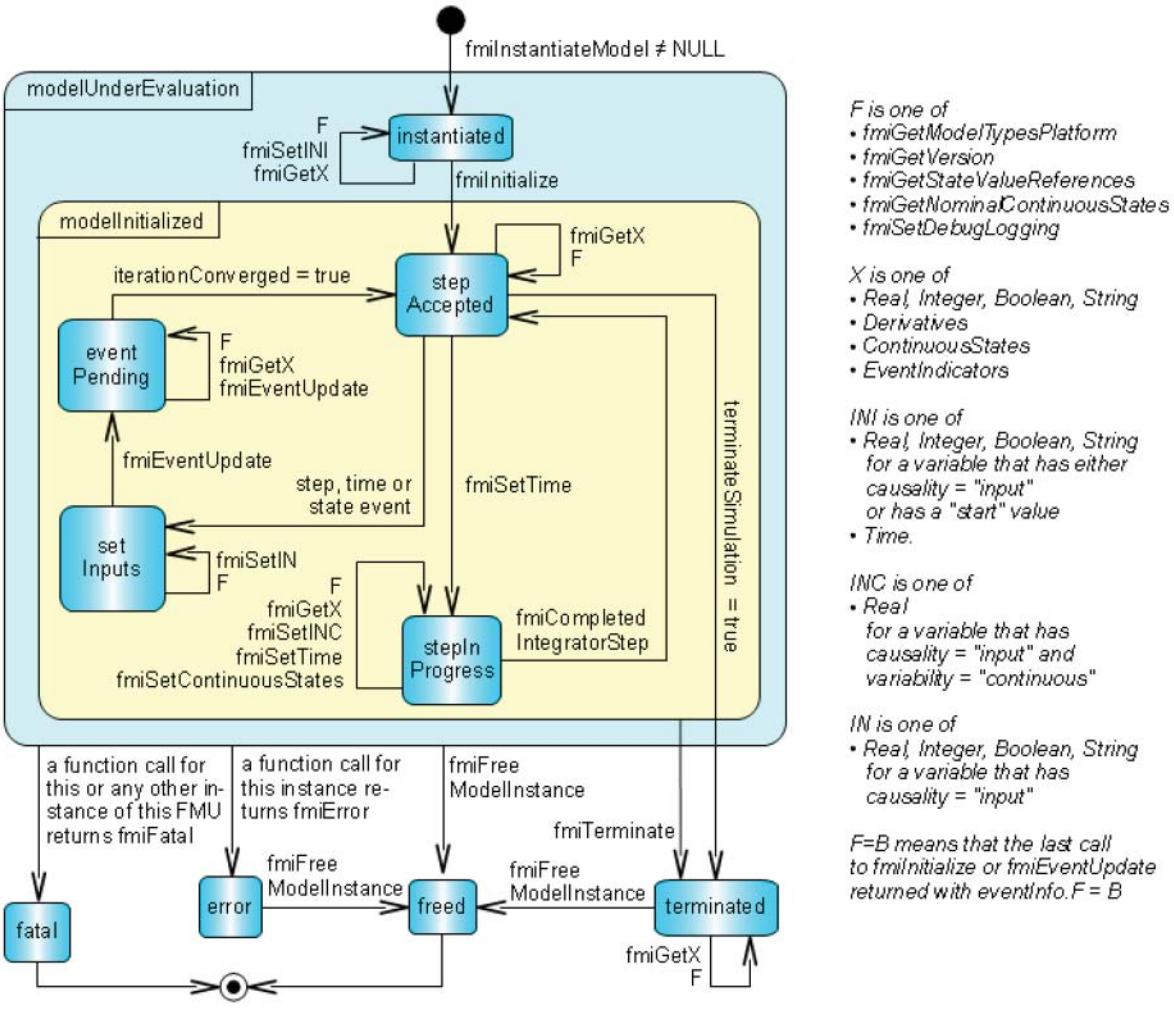

Figure 7: Calling sequence of Model Exchange C-functions in form of an UML 2.0 state machine. The Figure is taken from the FMI documentation [7].

If a transition is labelled with one or more function names (e.g. fmiGetReal, fmiGetInteger) this means that the transition is taken if any of these functions is successfully called.

- **instantiated** - In this state, inputs, start and guess values can be set.
- **stepAccepted** - In this state, the solution at initial time, after a complete integrator step, or after event iteration can be retrieved. If fmiInitialize or fmiEventUpdate return with eventInfo.terminated = fmiTrue, a transition to state *terminated* occurs.
- **stepInProgress** - In this state, an integrator step is performed. Also, the event time of a state event may be determined here after a domain change of at least one event indicator was detected at the end of a completed integrator step.
- **setInputs** - Before starting with the event handling, changed(continuous or discrete) inputs have to be set.
- **eventPending** - In this state, at least one event is waiting to be processed by a call to fmiEventUpdate. Intermediate results of the event iteration can be retrieved. If fmiEventUpdate returns with eventInfo.iterationConverged = fmiTrue, then this state is left and the state machine continues in the state *retriveSolution*.
- **terminated** - In this state, the solution at the final time of a simulation can be retrieved.

3.6 FMI SDK

An open FMI SDK is provided by QTronic [4] and can be found through the FMI web page [1]. Some structures and functions from the SDK are used in this implementation. The SDK supports the import and export of FMUs and has some basic examples of models that have been of great help while implementing the FMI.

The SDK provides a simple simulator that fulfils the basis of the FMI specifications for a correct simulation of an FMU. Some of the data structures used in this simulator are reused or slightly modified to collaborate with our FMI implementation. Especially the procedures for how the FMI functions are loaded from the DLL file and used in the simulation. The unzip function and XML parsing function are taken directly from the SDK. These were very efficient and suited the implementation well.

4 Simulink S-function implementation

In this section we will discuss the implementation of the FMI interface into Simulink. The goal is to have a robust and fully functional FMI implementation with a user friendly interface in Simulink that lets the user configure variables, ports and get information of the model. This is accomplished by implementing an S-function block with a GUI for the user to interact with the model.

4.1 The block

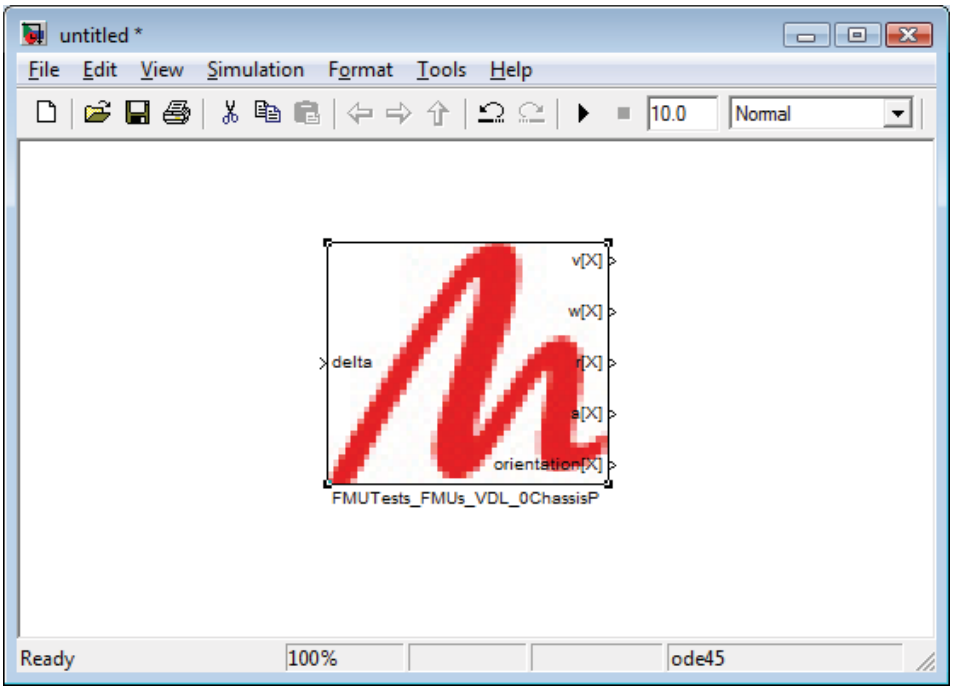

Figure 8: Implemented S-function block loaded with an FMU with input and output ports

4.1.1 Why an S-function block is used

There are at least three different types of block that the user may customize in Simulink. One of them is created by a subsystem of other blocks. The problem with implementing the FMI is then to choose which other blocks should be used in the subsystem to interact with the FMI functions, which will end up with using one of the other following custom blocks.

A second custom block is the *Embedded MATLAB Function*. This is a block using *Embedded MATLAB* code that works almost like MATLAB code. The problem is now to solve the ODE appropriately. In *Embedded MATLAB* neither of Simulink or MATLAB's ODE solvers[1] are available[2].

A third alternative is to use an S-function block. An S-function block is a block with callback functions. These functions are well defined with special purposes. They are then called by the Simulink Engine whenever it finds it appropriate during the simulation. Some of the functions are required to be implement and some are optional depending on the desired functionality. The S-function block is equipped with callback functions for solving ODEs. This property makes it superior to the other blocks and is therefore used in our implementation of the FMI.

There is different types of S-functions used for different implementations. The most flexible one is the C MEX S-function and is chosen in this implementation. This block is configured with MEX compiled C code. The fact that it is C code ease the interaction with the FMI functions. We will get into more details of the S-function block in section 4.3.

4.1.2 Adding the block to the Simulink library

To ease the user friendliness we would like to add the final product of the S-function FMI block to the Simulink library. This way the user may incorporate an FMU model into the system by adding the FMI block from the library and configure it. A description of how this *adding to library* procedure is done is given here in detail.

1. In Simulinkt's library browser choose "File -> New -> Library".
2. Locate the S-function block in the library browser and add it to the new library.
3. Right click on the S-function block and choose "S-Function Parameters..." then just set the "S-Function Name" to the S-function source file. The important here is that the compiled MEX-file and the *S_FUNCTION_NAME* macro in the S-function source file is the same that is set in "S-Function Name".

Next step is to add the new library to the library browser. This is done with a *slblocks.m* file that can be found in matlabroot/toolbox/simulink/blocks/slblocks.m and used as a template.

4. Create a new folder and add the *slblocks.m* file.

[1] The trapezoidal numerical integration function *trapz* and *cumtrapz* are available
[2] MATLAB functions may be invoked using one of the *extrinsic methods*

5. Add the library file, compiled MEX-file and slblock.m to the MATLAB path and then the block is added.

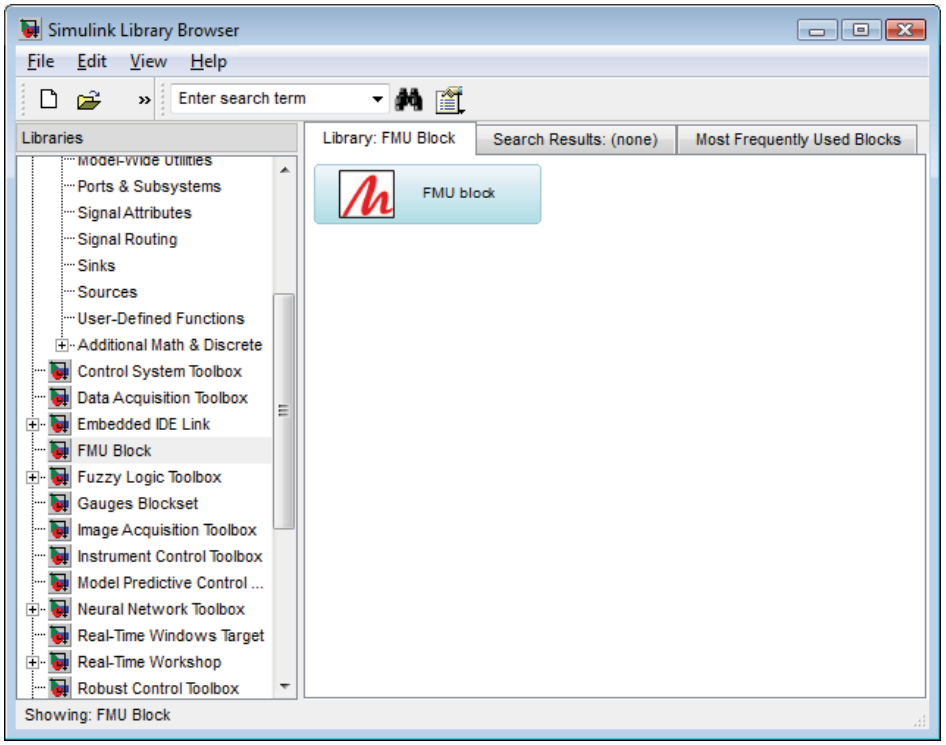

Figure 9: Implemented block added to the Simulink library

4.1.3 Block Properties

All blocks in Simulink can be configured with different parameters and functions during the model set up(not during the simulation). For example this would let the user to implement a GUI for the block from which the number of ports may be configured. A full list of these function with descriptions can be found in the MATLAB documentation [8]. The two functions used in our S-function implementation are the *LoadFcn* and *OpenFcn*. *LoadFcn* is called by the block every time the model is loaded. This function should make sure that model is available for simulation if the model was saved in such a state. *OpenFcn* is called when the user double clicks on the block. This should open a GUI.

In Section 4.3.1 we will learn that the input and output ports of the block are configured within the callback function mdlInitializeSizes(not any more) in the S-function. To be able to configure the ports from the GUI, mdlInitializeSizes has to be invoked and the configuration needs to be fetched. This

is achieved by calling the MATLAB function *set_param* that triggers the mdlInitializeSizes to run and setting the *common block parameter UserData* to a appropriate data type containing the configuration. From mdlInitialize-Sizes the MEX function *mexCallMATLAB* can then be called to fetch the configuration. In the same manner as the ports are configured, parameters and initial values are set in the S-function using the *UserData* parameter. The *UserData* parameter is set persistent such that it is saved in the model when the model is saved.

The names of the ports visible on the block that can be seen in Figure 8 is set with the *mask icon drawing*.

4.2 The implemented GUI

4.2.1 Description and Functionality

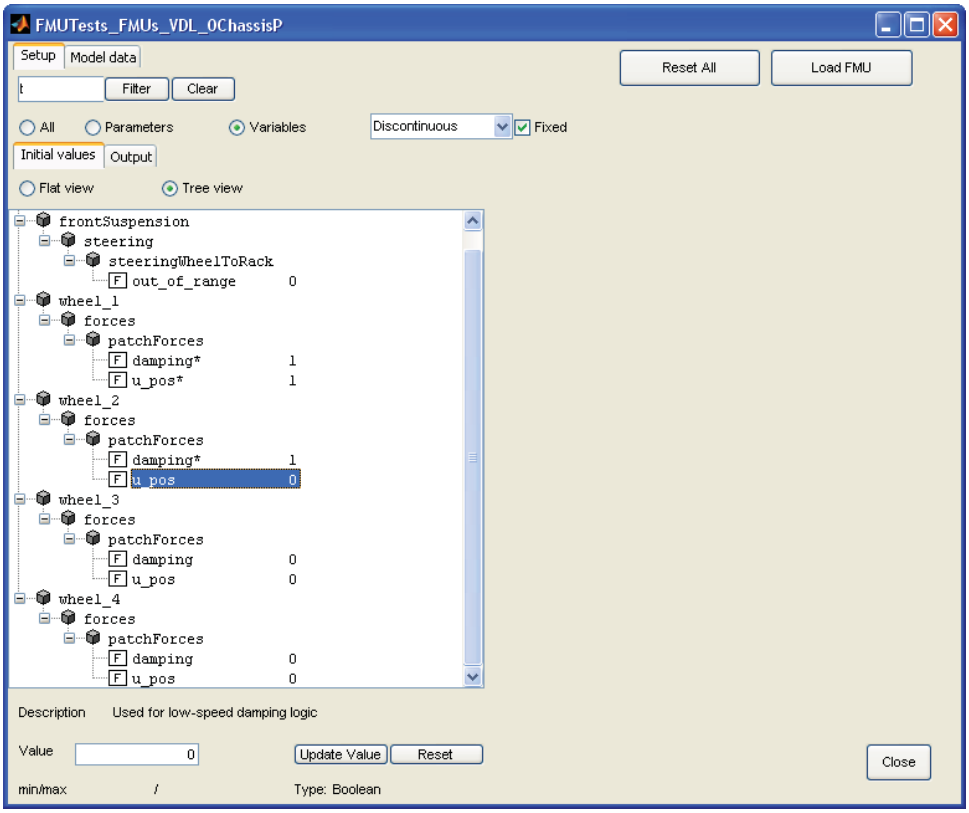

Figure 10: Showing the filtering functions of a model variables in the *Initial values* tab

In Figure 10 above we see the GUI used in the S-function. The GUI is

in the tab *Initial values* where the FMU can be loaded and all the initial values can be set. We will now describe the main functionalities of this tab in more detail. In the top left we got all the variable filter functions. For example in Figure 10 we see that only variables containing the letter "t", are discontinuous variables and have the attribute *fixed=true* are showed in the variable tree. When the variables name filtering is used, the tree expands if *Tree view* is set. These filter functions are essential for large models with many variables to eliminate tedious manual time-consuming search of a variable. The user may toggle between a "Flat tree" where all variables are listed straight up and down and the *Tree view* where the GUI take advantage of the variable naming convention defined by the FMI for structure defined FMUs. Properties and value of a selected variable are showed in the bottom left of the GUI. If the user changes the value different from its default, a "*" is added to the variable's name.

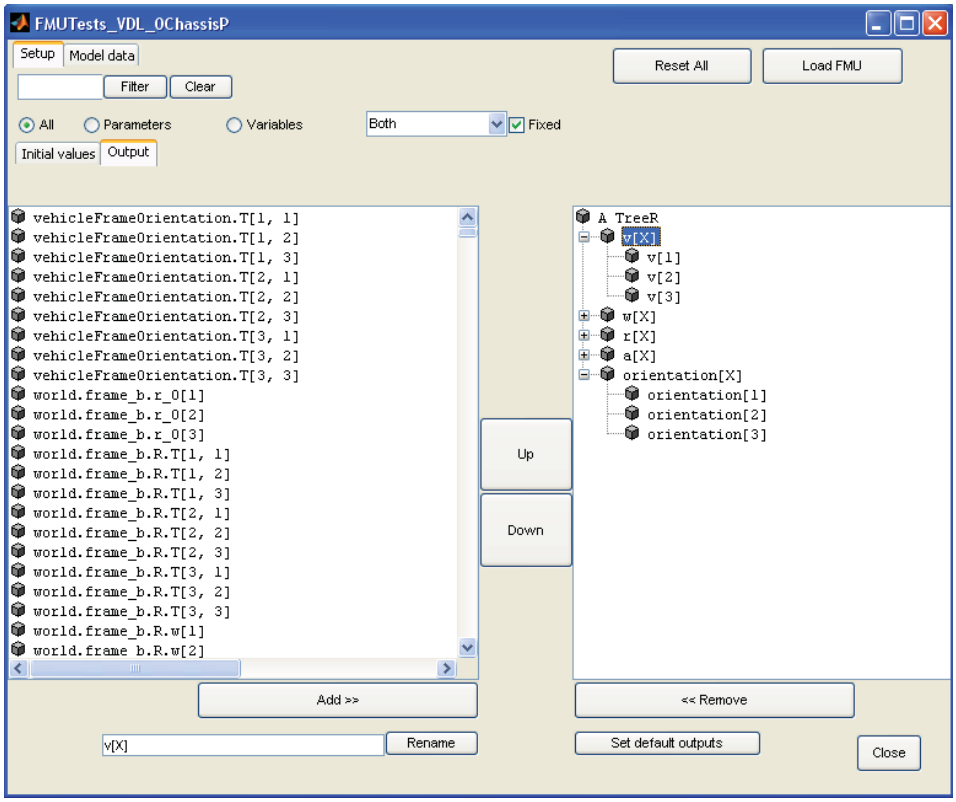

Figure 11: In the left tree the user may choose between variables to move to the right tree that is used as output ports on the block

The *Output* tab of the GUI is shown in Figure 11. This is where the output ports of the block are configured. On the left side all the variables in the model are listed in a flat tree. These can be added to the tree on the right

side of the GUI. In Simulink it is possible to have multiple signals in a port. If the user selects multiple variables(hold down ctrl while clicking the variables) from the left tree and add them. The output port will be named by the first selected variable but the port will have all the variable's signals in that output port. The name of the port can be changed as well as the position on the block. In the figure the default output ports are set using the naming convention for vector variables and the associated block is showed in Figure 8. Input ports can not be configured by the user. These are set by default using the variable name as the name of the port and are scalar signals.

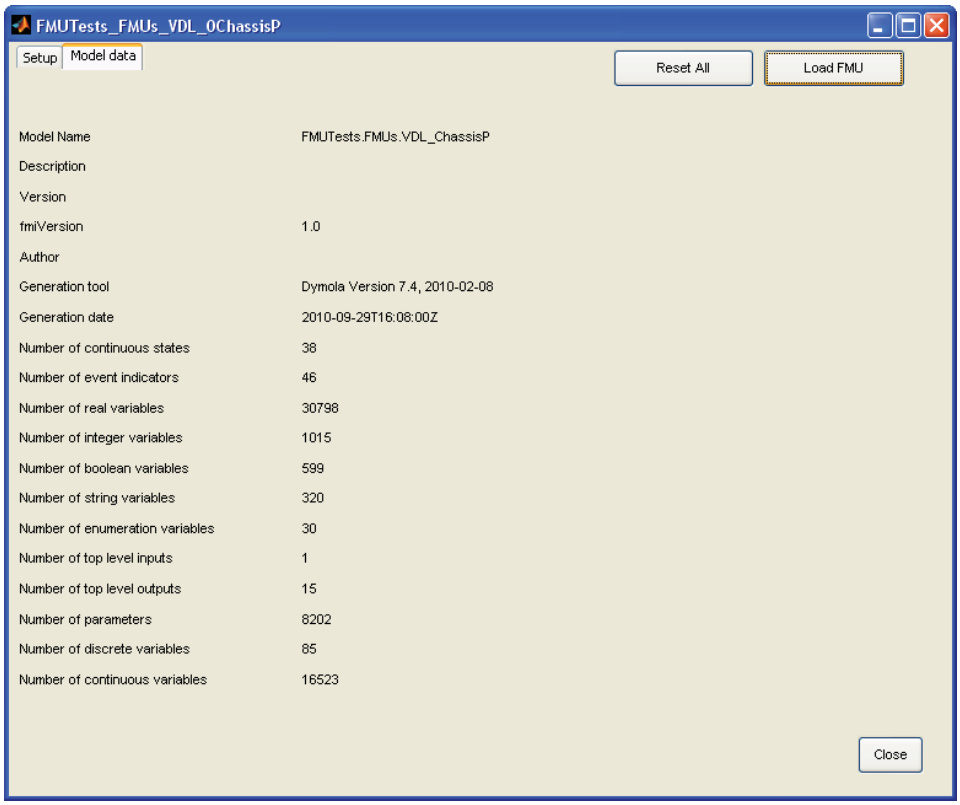

Figure 12: Summarized data extracted from the FMU

The last tab is the *Model data* in Figure 12. It shows data extracted from the model description in the FMU.

4.2.2 Technicalities

The GUI was created using *User Interface Objects*. Most of the GUI is build with *uicontrol* objects and the rest with undocumented *ui** functions. Many

GUI objects in MATLAB have roots from Java GUI objects. This makes it possible to exploit functionalities found in the Java documentation that is not documented by MATLAB.

When the block's callback function *OpenFcn* is called, it creates a MATLAB *figure* and then draw all the UI objects in it. In all of the uicontrols it is possible to associate a callback function that is called whenever the uicontrol is triggered to. A handle is returned by every call to a UI function. By using the MATLAB command *set* and *get* on the handle, it is possible to change properties of the uicontrol. In the GUI for instance, when a tab is switched, some buttons and texts are set to visible/invisible.

Every time a change is made that effects the block, the *set_param* is called to update the *UserData* parameter of the block. On the other hand, when the GUI is opened, the *get_param* is called to get the *UserData* parameter. This keep the data in the GUI and the S-function block synchronised.

To keep track of which GUI is associated to which block, the figure handle and the block's path name is saved to the *UserData* parameter. Every time a block invoke the *OpenFcn*, a check in the *UserData* if a GUI is already opened is made and highlight it in that case, otherwise a new GUI is created.

It is very time consuming to draw the variable trees and this is noticeable for big models. So the trees are saved in the *UserData* and are then reused the next time the GUI is opened. To improve the performance when a search of a variable is done, a bit search is used. For the name filter function this does not apply.

4.3 S-function API and function mapping

Simulink invokes the S-function block callback functions during the simulation to calculate the outputs. The first step in the simulation is to initialize the S-function block. The following scheme [9] in Figure 13 describes the initialization loop with respect to the callback functions.

When the initialization is done the simulation execute the simulation loop [9] found in Figure 14.

We will now list some S-function callback functions and there macros that can be set and discuss how these are set with respect to the FMI. All these functions has at least the input argument *SimStruct*. *SimStruct* is a structure that may be described as the S-function model instance. It is this structure that is *updated* in the callback functions, and then used by the solver to perform the evaluations. It may then be used to update the FMI model instance with the new states.

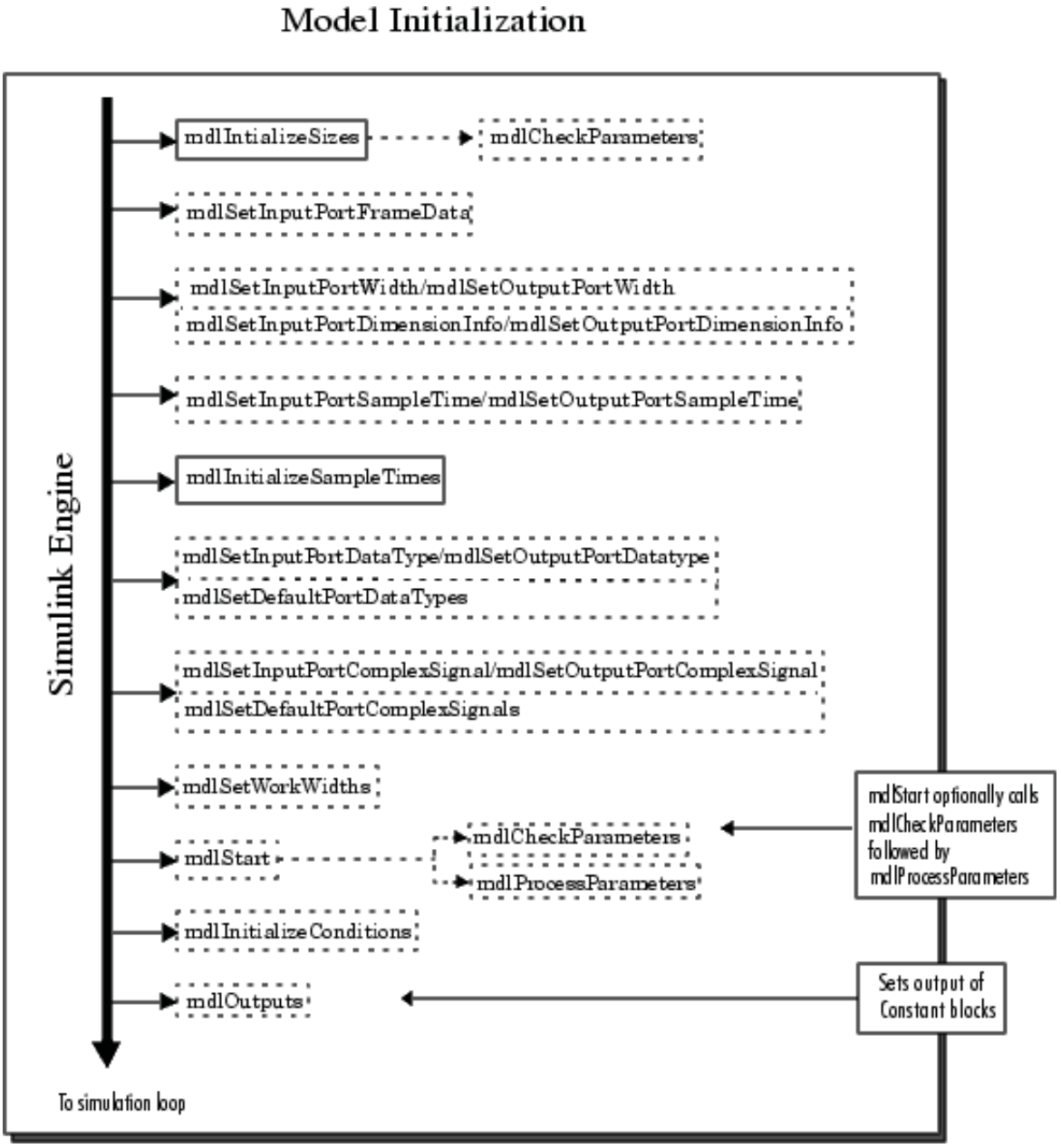

Figure 13: Solid rectangles indicate callbacks that allays are performed. Dotted rectangles indicate a callback that may be omitted.

4.3.1 mdlInitializeSizes

In *mdlInitializeSizes* [13] the number of inputs, outputs, states, port types, port widths among many other things are set. The inputs can be configured

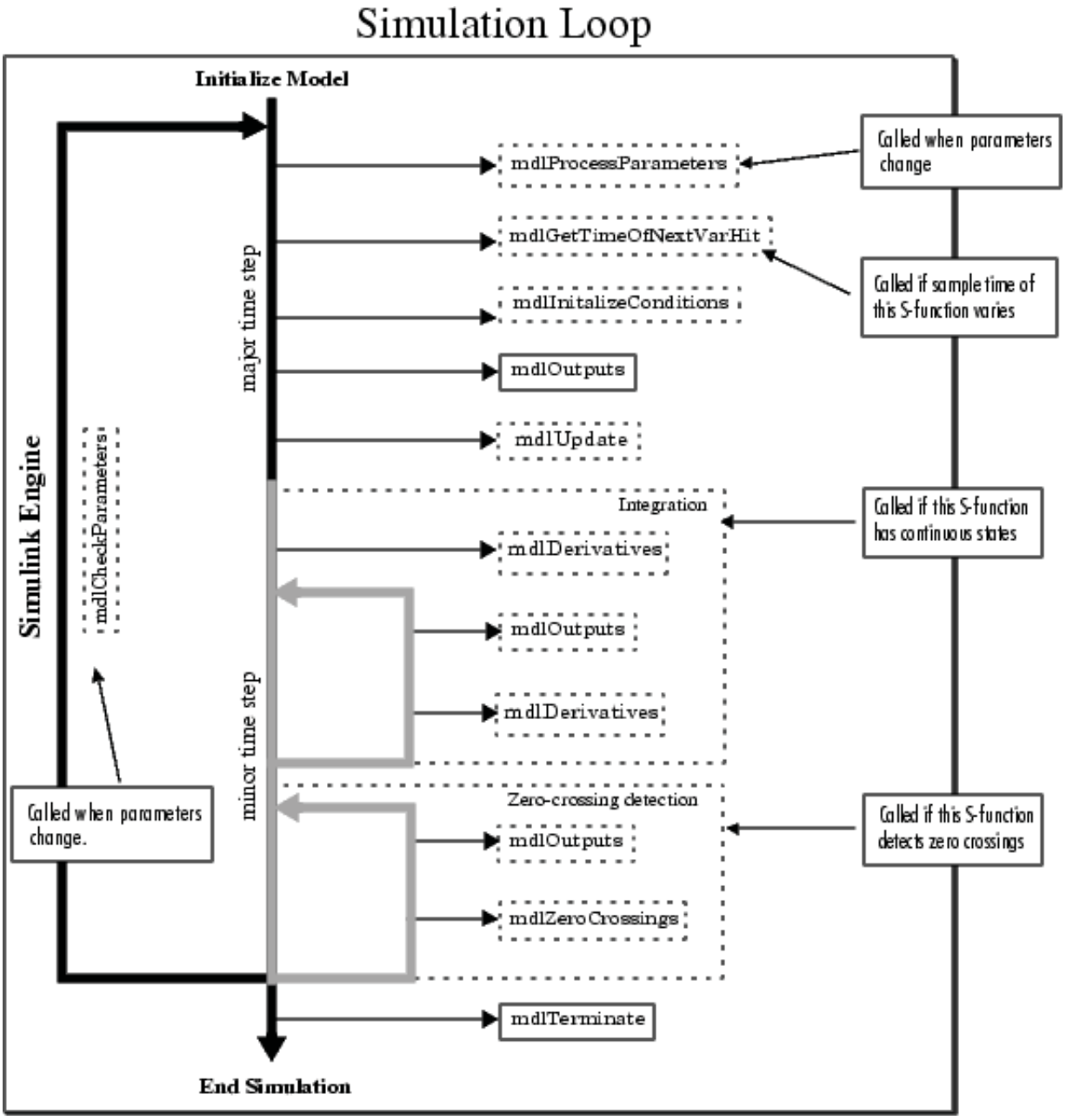

Figure 14: Solid rectangles indicate callbacks that are always performed. Dotted rectangles indicate a callback that may be omitted.

to have different dimensions and types such that it is possible to have ports of 16 bits integers 5×12 matrices. The FMI implementation sets the dimensions to $1 \times X$ for each port and of the same type as the variable is specifies to be by the model description. X is defined by the GUI which may be different

for every port.

If the input ports are used in the *mdlOutputs* which they are for this implementation, the direct feedthrough flag needs to be set to 1 in *ssSetInputPortDirectFeedThrough*. This indicate that the input signals are used in the *mdlOutputs* [12].

The number of continuous states are set with *ssSetNumContStates* and is the number of states the solver will integrate. If the number of continuous states are more then 0 the *mdlDerivatives* callback function has to return the derivatives. In the model description of an FMU the attribute *numberOfContinuousStates* are the number of continuous states of model and therefore we set the *ssSetNumContStates* to this.

The sample time are in Simulink an indicator of when an input is processed and an output are produced in a block. The internal states are updated accordingly to this. Different blocks can have different sample times. There are two methods to specify the sample times of a block.

- **Block-based sample times** - The solver uses the same rate in all the inputs and outputs.
- **Port-based sample times** - The solver can use different rates in the inputs and outputs.

The number of sample times a block has is set with *ssSetNumSampleTimes*. The different types of sample times that can be set at each port are discrete or continuous. They can then be configured i.e. such that continuous does not change in the minor time steps[3], the discrete sample times may have a constant sample time or get calculated in another callback function. It is also possible to set the sample times to be inherited, constant such that the block is only executed once, trigger and asynchronous when the block is not executed regularly.

At first, one might think using only continuous sample times and a variable-step solver is the easiest way to configure the s-function block, and yes, the discussion will show that it is a good configure for the S-function for general FMUs[4].

A reason for using a discrete sample time would be to speed up the process of hitting time events. This would be very efficient for an FMU with only discrete variables. A discrete input is only updated when an event is triggered as we can see in the FMI calling sequence in Figure 7. *mdlGetTimeOfNextVarHit* is a callback function that can be used to set the next

[3]The time steps are divided into major and minor in a continuous sample time configuration. The minor steps are used to improve the accuracy in the major steps.

[4]i.e. not making special case for FMUs with only discrete variables, continuous states, discrete variables with continuous inputs etc.

discrete time step. This could be configured such that it hits the next time event exact without any steps in between. But one should have in mind that that the *fmiCompletedIntegratorStep* could trigger events depending on time in between the two time events, and the FMI specification says nothing about *fmiCompletedIntegratorStep* for pure discrete FMUs when no integration is performed by the S-function. So due to the *fmiCompletedIntegratorStep* that should be called by the environment after every completed integrator step, to choose a continuous sample time feels more natural then a discrete sample time. The advantage of choosing a fixed-step solver and a discrete time sample for a pure discrete FMU might be minimal for a variable-step solver with continuous time sample with appropriate configuration in Simulink.

A comment on the *mdlGetTimeOfNextVarHit* is, if an FMU is simulated with both discrete sample time and continuous sample times, the next time step for the discrete sample time can not be changed until *mdlGetTimeOfNextVarHit* is called again when the next time step must be positive. So if the continuous sample time dependent simulation detects a new time event, the next discrete sample time, the time of next time event becomes obsolete [7]. The problem is that the time of next discrete sample time can not be aborted and updated. The advantage to use a discrete sample time for time events would be to reduce the continuous zero-crossing detection with one event indicator function which it would not be worth if the time event is triggered with uncontrolled accuracy or if it should not at all. So if an FMU with continuous states are used, the only solution for time event handling is to use continuous sample time and implement the triggering of time event such that the time of next time event can be aborted and updated. This is done by using a zero-crossing function to the zero-crossing detection that is only available if the S-function is using a continuous sample time.

For the curious, if the you would create an S-function with different time samples, the concerned callback functions can separate which sample time that is used. If a callback function with this property is independent of which sample time that is currently used, the *UNUSED_ARG(tid)* macro should be added. But this is not used in the this S-function since there is only one sample time.

4.3.2 mdlStart

mdlStart [14] is a function called only once every simulation and is called just before the simulation is started. This is where initialization and memory allocation that just need to be called once, may be performed. This is therefore a really good place to perform the model instantiation and do the initialization. This is also where configurations from the GUIs are handled and data is extracted in suitable manner for the S-function to increase the simulation per-

formance. By setting the *SS_OPTION_RUNTIME_EXCEPTION_FREE_CODE* macro in *mdlInitializeSizes* it allows Simulink to bypass the exception handling set up that is usually performed prior to each S-function runtime callback function. No exceptions(long jumps) are then allowed in the run-time methods. All MEX* functions have the potential for long jumping and some MX* functions. When extracting data from the GUI there is an inevitable use of a MEX* and MX* functions. The data contained in *mxArrays* are copied to *calloc* allocated memories to better suit[5] other C-functions and minimize the MX* presence.

From the blocks *UserData*, the S-function fetches the address to DLL-file and load the FMI functions. The addresses of the FMI functions are saved into a structure, we call it the *User Data structure*. This structure contains among many other things also the model instance. A pointer to this structure is then set with *ssSetUserData* so that it can be reached within all callback functions.

4.3.3 mdlZeroCrossings

This function is called by the solver to detect zero crossings. In *mdlZeroCrossings* [15] the zero crossing vector *ssGetNonsampleZCs* of length *ssGetNumNonsampledZCs* set in *mdlInitializeSizes* is updated. When a zero crossing is detected, the solver integrates up to the left edge of the zero crossing. Simulink then steps over the zero crossing and begins a new integration step.

The FMI function *fmiGetEventIndicators* return the event indicator vector that suits the *ssGetNonsampleZCs* zero crossing vector perfect. *fmiGetEventIndicators* computes the indicators at time instance set by *fmiSetTime* and the states *fmiSetContinuousStates*. This means, when Simulink call *mdlZeroCrossings* at time instance *ssGetT*, we have to make sure that the continuous states and time are synchronized with Simulink. According to the simulation loop found in Figure 14 there is no guaranty that *mdlZeroCrossings* are called at the same time instance as *mdlOutputs*. To make sure the time and states of the model instance are synchronised with the simulation environment, the time and states are set to *ssGetT* and *ssGetContStates* by calling *fmiSetTime* and *fmiSetContinuousStates*. The S-function input ports corresponding to *fmiSetINC* is also set before *fmiGetEventIndicators* is called.

From the discussion of sample times in *mdlInitializeSizes* the time events shell be handled as a zero-crossing, state event. If there is no time event, the function is an arbitrary positive constant. To not make the solver sensitive for this event indicator being close to a zero-crossing, it is a good idea to

[5]Not mixing MATLAB/Fortran data structures with C data structures

have a relative big value of the constant with respect to the other event indicators. If a time event is active, the event indicator is changed to be the linear function $z = nextEventTime - ssGetT(S)$.

4.3.4 mdlOutputs

In *mdlOutput* [16] is where the final computations of the output signals from the block is performed. The functions is called every time step, both minor and major time steps. This is the callback function that has to deal with the event handling. When a step is taken and the *mdlOutput* is called, the step integrated up to that time can be reached with *ssGetContStates*.

Current simulation time is fetched with *ssGeT* and the output ports can be set using the *SimStruct* macros. In this implementation of the S-function, the output type is inherited. This is accomplished with the *ssGetOutput-PortSignal* macro that returns a void pointer to the signal the port emits and then a type cast to the inherited type, *ssGetOutputPortDataType*. The output ports were configured with the inherited property in *mdlInitializesizs* with the *DYNAMICALLY_TYPED* macro in *ssSetOutputPortDataType*.

Since we use continuous sample times and zero crossing we see in Figure 14 that *mdlOutputs* iscalled for minor time steps. Since minor time steps is used to improve accuracy of solution and finding zero-crossings these calls can be separated from major time steps with *ssIsMajorTimeStep* and *ssIs-MinorTimeStep* macro.

The minor time steps corresponds to the inner loops of *stepInProgress* in Figure 7. The major time steps would then include the step from *stepIn-Progress* to *stepAccepted*. The minor step in *mdlOutput* has no function since it is almost always associated to to a call to either *mdlDerivatives* or *mdlZeroCrossings* and can therefore perform all necessary FMI calls from these instead. Otherwise, if the solver would only call *mdlOutput* in a minor step and we forget about event handling, it is an obsolete call according to the FMI specifications. The event handling is dealt with in major time steps as we will see. When a major time step is taken in *mdlOutput*, the solver has completed an integrator step. The FMU on the other hand is still in the *stepInProgress* and has to step to *stepAccepted*. The FMU model instance gets synchronised with the solver by calling *fmiSetTime*, *fmiSetContinu-ousStates* and *fmiSetINC*. Then *fmiCompletedIntegratorStep* is called and we end up in *stepAccepted*. *mdlOutput* then checks if any events are trig-gered and handle these accordingly. If no events are triggered the *fmiGetX* are used to produce the output of the S-function. If an event is triggered, the *fmiSetIN* is called and then the *fmiEventUpdate*. The *eventPending* block in the FMI calling sequence is configured to just be called once such that is

does not return for every event iteration that is performed internally. If the state values were changed after the event update these states are updated in the FMU with *fmiSetContinuousStates*. To update the states in solver, the solver needs to be restarted with *ssSetSolverNeedsReset*. This macro is only available for variables-step solvers and causes the simulation to reinitialize the variable-step size and the zero-crossing computations.

The simulation loop is started in *mdlStart* when all the initial values are set including the continuous states. The first callback function called in the simulation loop in figure 14 is *mdlOutput* at the time initial time. This is a major time step and this corresponds to *stepAccepted* in the FMI calling sequence. If the *fmiInitialize* in *mdlStart* would trigger an event, the event is handled in *mdlOutputs*.

4.3.5 mdlDerivatives

Computes the derivative of the continuous states. It is called each time step and must set the values of all derivatives. The FMU provides the function *fmiGetDerivatives* that calculates the derivatives for the current time instance and states set by *fmiSetTime* and *fmiSetContinuousStates*. According to the simulation loop found in Figure 14 there is no guaranty that *mdlDerivatives* [17] is called at the same time instance as *mdlOutputs*. To make sure the time and states of the FMU are synchronised with simulation time *ssGetT* and states *ssGetContStates* we call the *fmiSetTime* and *fmiSetContinuousStates* functions in *mdlDerivatives*. The S-function input ports corresponding to *fmiSetINC* is also set before *fmiGetDerivatives* is called.

4.3.6 mdlTerminate

mdlTerminate [18] is called last in the simulation loop and any actions required at termination may be performed here. This function may also be called if *ssSetErrorStatus* is invoked when the *SS_OPTION_CALL_TERMINATE_ON_EXIT* option is set in *mdlInitializeSizes*. The FMI function *fmiTerminate* and *fmiFreeModelInstance* should be called at the end of the simulation to deallocate all resources allocated since *fmiInitialize* and *fmiInstantiateModel* was called. Therefore these FMI functions are naturally called in *mdlTerminate*. Other resources as the *User Data structure* pointed to by *ssGetUserData*, are released here.

5 MATLAB MEX interface

In this section we will discuss the implantation of the FMI in the MATLAB environment. We will base the discussion around MATLABs ODE solvers by discussing how they work and then later how we could incorporate the FMI functions. The final goal is to be able to use the solvers to simulate an FMU.

5.1 ODE solvers in MATLAB

The ODE solvers in MATLAB are functions called like any other function in MATLAB with return values and input arguments set by the user. MATLAB can pass function handles as arguments, that means that any given function can be used as an input argument and used within the function. The ODE solvers in MATLAB use this functionality to solve ODEs. The ODE solver interface defines some input arguments to be mandatory and some to be optional.

Mandatory input arguments for the ODE solver are the initial conditions, time span to be integrated over and a function handle that evaluates the right side of the differential equation $y = f(t, y)$ [19]. The optional input arguments that are essential for the FMI implementation are the event handling(*Events*) and the output function(*OutputFcn*). The event handling in the ODE solver is just searching for the event indicators to switch domain. If such a domain switch is found, the user is given two alternatives. Either continue the integration or stop it. If we would like to handle the event(for example calling fmiEventUpdate) we would need to stop the simulation, handle the event and then start solving the ODE from where we stopped. This is the standard approach for event handling using the ODE solvers in MATLAB. There is no function to tell the ODE solver to stop integration at a certain time during the integration. For this we have implemented the time event handling as a state event as we did in the S-function.

We give here an example in pseudo MATLAB code showing a calling sequence for simulating an FMU using the MATLAB ode45 solver. We will discuss the input arguments to the ODE solver in more detail afterwards.

Listing 1: Draft of an FMU simulation using MATLAB ode45 solver

```
function FMIexampleUsingODEsolver
fmiLoadFMU('bouncingBall.fmu');

Tstart=0;
Tend=1;
```

```
fmiSetTime(Tstart);
fmiSetReal/Integer/Boolean/String
fmiInitialize();

y0=fmiGetContinuousStates();

options = odeset('Events',odeEventFMI,'OutputFcn',odeOutputFMI);

while Tstart<Tend
  [t,y] = ode45(odeDerivativeFMI,[Tstart Tend],y0,options);
  nt = length(t);
  T = [T; t(2:nt)];
  Y = [Y; y(2:nt,:)];
  fmiSetReal/Integer/Boolean/String
  fmiEventUpdate();
  y0=fmiGetContinuousStates();
  Tstart = t(nt);
end

fmiTerminate();
fmiFreeModelInstance();
end
```

5.1.1 odeDerivativeFMI

The ODE function is the first argument for the ODE solver and is a function handler that evaluates the right side of the differential equation $\dot{y} = f(t, y)$. This function has time and current states as input arguments. The *fmiGetDerivatives* shell therefore be called here to return the model's derivatives. Before *fmiGetDerivatives* is called the time, states and any top level FMU inputs of the FMU needs to be synchronized. The time and states are given by the solver but any top level inputs must be given by the user for the given time and states. This is implemented with a function *fmiSetINC* that sets the values of inputs with the FMI set functions. The values that are set are either passed as a vector with values for some given times or as a function. The vector is interpolated to give a value for the time instance but the function calculate the value for that time.

Listing 2: Draft of ODE solvers differential equation function

```
function dydt = odeDerivativeFMI(t,y)
  fmiSetTime(t);
  fmiSetINC(t);
  fmiSetContinuousStates(y);
```

```
  dydt=fmiGetDerivatives();
end
```

5.1.2 odeOutputFMI

The ODE option *'OutputFcn'* can be set to a function handle [20]. This function is called after every completed integrator step. The given input arguments for this function is the time, state and a flag. The flag is used to indicate an initial step, completed integrator step or if it is the final step. This function does not affect the solution and is often used for plotting or stop the integration. The output of this function is a state that determines whether or not the integration should stop. According to the FMI calling sequence in Figure 7, after every completed integrated step the fmiCompletedIntegratorStep should be called for the current time, states and inputs. If fmiCompletedIntegratorStep returns with the flag to call fmiEventUpdate, the integration is stopped. The synchronization procedure is the same as in 5.1.1 above.

Listing 3: Draft of ODE solvers output function

```
function status = odeOutputFMI(t,y,flag)
  if isequal(flag,[])
    fmiSetTime(t);
    fmiSetINC(t);
    fmiSetContinuousStates(y);
    status=fmiCompletedIntegratorStep();
  end
end
```

5.1.3 odeEventFMI

The *'Event'* option is set with a function handle that is used by the solver to detect zero-crossings [20]. Input arguments are the time and the current states. The output arguments of this function are the event indicators, if the event is sensitivity to the direction of the event indicator, and whether or not an event indicator should trigger a termination of the integration. If the zero-crossing is detected and the integration is stopped, solver integrates up to the right side of the event where the *fmiEventUpdate* should be called. The 'Event' function's event indicators are given by the fmiGetEventIndicators with the additionally value for the time event handling. This is done by making the time event to a state event by adding a zero-crossing function as shown in the pseudo code below.

Listing 4: Draft of ODE solvers event function

```
function [value,isterminal,direction] = odeEventFMI(t,y)
  fmiSetTime(t);
  fmiSetINC(t);
  fmiSetContinuousStates(y);
  value=fmiGetEventIndicators();
        %Added zero crossing function for time event
  if FMU.eventInfo.upcomingTimeEvent==true
    value=[value; FMU.eventInfo.nextEventTime-t];
  else
    value=[value; 100];
  end
  isterminal=ones(size(value));
  direction=zeros(size(value));
end
```

5.2 MATLAB Executable, MEX

We will now discuss how the FMI functions above are called in MATLAB. First of all, these are not the real FMI functions, these are just functions with the same name that somehow calls the real FMI function of the FMU. They are so called wrapper functions. The real FMI functions are written in C [7]. To use C-functions in MATLAB, the C-functions must be written and compiled as MEX-files, MATLAB executable files. MEX-files are DLL-files, see section 2.4.3, produced from C,C++ or Fortran. The MEX functions can be called as any other functions in MATLAB.

All MEX-files written in C must include four things [10].

1. #include mex.h
2. mexFunction, a gateway function
3. The mxArray
4. API functions

The **#include mex.h** is necessary, to use the mx* and mex* functions. **mexFunction** is the gateway function of the DLL that MATLAB uses. The MATLAB data is represented as **mxArray** in C. mxArray are the arrays you see in the MATLAB workspace(scalars, matrices, string, cell arrays, etc.). Except the data itself, mxArray contains for instance the type, dimensions, if it is structure and the field names in that case, etc. The **API functions**, mx* and mex*, are used to access the data in mxArrays and perform tasks in

MATLAB. The data in mxArray is stored columnwise, which is how Fortran stores matrices.

The code below is a template for the most basic MEX-file.

```
#include "mex.h"

//Gateway function
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
//variable declarations here

//code here
}
```

5.2.1 Void*

A possible way to invoke the FMI functions is to in each wrapper function, pass an argument of the DLL file location. Then load the DLL file, call the FMI function of interest and then return the DLL handle again. This produce some overhead that will lower the performance in a simulation. To remove this overhead, a MEX function is used to load the DLL library and save pointers to the FMI functions in the DLL file. When a wrapper FMI function is now called from the workspace, it just needs the address for where the real FMI function is found. The wrapper function can then easily invoke the FMI function without loading and unloading the DLL. Another MEX functions must then be called at last when the user want to return the DLL handle.

MATLAB do not support void* or function pointers like those we would need to save between the FMI function calls. MATLAB on the other hand do support double precision floating-point numbers. The solution is then to use a union to save the addresses of the FMI functions into doubles. In C this type of casting should be no problem since the union is the size of the biggest member. But in MATLAB I have not figured out which data type that guaranties to fit a C void pointer. Double is one of the biggest data types in MATLAB and is therefore used.

5.2.2 The FMU MATLAB structure

A MATLAB structure, we call it FMUstruct, is used to contain all the FMI function addresses and all other essential data of the FMU. For example all the model variables are saved in an array of structures. The FMI function

addresses are saved in a double array. Every wrapper FMI function takes the FMUstruct as input argument where the address to the corresponding FMI function is fetched and then invoked along with other input arguments passed through the wrapper FMI function.

When an FMU model is instantiated, a void pointer to the model instance is received. The address is saved as described above in section **??**. In the FMI function fmiInstantiateModel, the simulation environment has to set the memory allocation and releasing function that the fmiInstantiateModel should use. The practice is to use MX* functions like mxCalloc, to allocate memory in MEX-functions. Memory allocated within a MEX function that is not used as an output argument is freed when the MEX function returns if the memory is not made persistent. So a small memory allocation function is written allocating the memory and making it persistent such it suits the fmiInstantiateModel. The standard MEX function mxFree is used to free the memory.

When a new FMU is loaded, the FMU file is unzipped in a unique folder in the temporary folder at the computer. Due to the uniqueness of the folder makes it possible to load multiple FMUs into the MATLAB workspace and use them simultaneously. When the user is finished working with the FMU, the user needs to return the DLL handle. This is done by calling another function with the FMUstruct as input argument. The consequences of not doing this is that the temporary folder where the DLL file is located will not be removed.

6 Case studies

The implementation of the FMI interface in the S-function and the MEX interface are verified by comparing the results of model simulations in different environments. These other simulation environments are considered to have a correct implementation of the calling sequence of the FMI. JModelica.org and Dymola are two tools supporting FMI import that are listed on the FMI documentations page [3]. These are used in our evaluation. Dymola are used for both generating and simulating the models. The models that are used for generating FMUs are simulated both as the native model and as an exported FMU in Dymola. This *native* simulation may not contribute to the verification of a correct implemented calling sequence but may be interesting to see how well the FMU models stands against the real models and simulations. In the result figures below we denote the Dymola simulations of the native model with *Native* and the FMU model with *FMU*.

The goal here is to verify the implementations such that all possible calling sequences are tested. Since this would generate a lot of cases if we would test one calling sequence at the time, we therefore use a few models that are exploiting multiple calling sequences. We now list some model properties of interest to verify the implantations and compare the performances. We will focus on this list when discussing the model cases below. We just mention that the physical interpretation and descriptions of the model is restricted as well as the selected variables that are visualised in the results. This is in some sense out of the scope for this report and hopefully the reader is satisfied with what is given.

- Solving the ODE for a model with continuous states
- Time event handling
- State event handling
- Step event handling
- Model with inputs
- Solving algebraic loop for models with direct dependencies
- Performance of simulation

In Dymola we used the relative tolerance 1e-10 with the DASSL solver. In JModelica.org the CVode solver was used. In MATLAB and Simulink the ODE15s was used. The relative tolerances were set to 1e-10 and the absolute tolerance to 1e-12 in JModelica.org, Simulink and MATLAB. These are all BDF methods with max order 5 set. The low tolerances were used in hope to make any errors stand out even more. We also get the chance to see if

the results converge towards each other which would be a good sign of a well implemented interface.

6.1 ODE with time and state events - Coupled clutches model

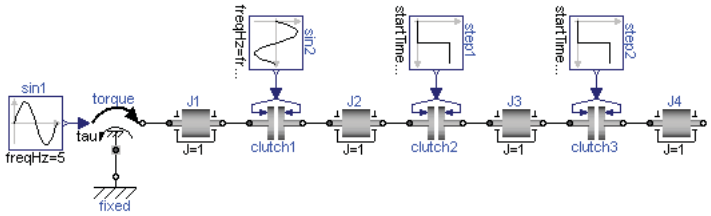

Figure 15: Block diagram model Coupled clutches in Dymola

The coupled clutches model in Figure 15 is a standard example found in the Modelica standard library. It demonstrates how variable structure drive trains are handled. The drive train consists of four inertias and three clutches, where the clutches are controlled by some signal. This model has 8 continuous states with 54 state event indicators and time events. Time events and state events do occurs within the 10 seconds the model is simulated. We choose to visualize the results from one variable, w1, that is the angular velocity of inertia 1. The results of the whole simulation is found in Figure 16. The solutions seem to lie on top of each other. In Figure 17 we have taken the difference of our implementations and the Dymola Native solution. This is done by first interpolating the results. Now we see that there some spikes. The first couple of spikes are verified to be related to a time for when an event occurred and that is probably the rest of the spikes as well. A compilation of the absolute value of the differences are visualized in a histogram in Figure 18. We can see that the differences greater then 1e-8 can be found for the MEX interface but not for the S-function. If we now zoom as in Figure 19 we see a phenomenon that is seen around all the spikes and also some spikes where the S-function have a greater difference then the MEX interface. This is most likely to be caused due to the spline interpolation around discontinuities. Unfortunately I have not figured out a good way to compare the results without interpolating over the events. We will discuss the spikes more in section 7.1. If we for the moment neglect the biggest differences we see that the difference is in the order of 1e-8. I would say that this is satisfying with respect to the low tolerances and the interpolation of the results and that it indicates the S-function and the MEX interface to be correctly implemented.

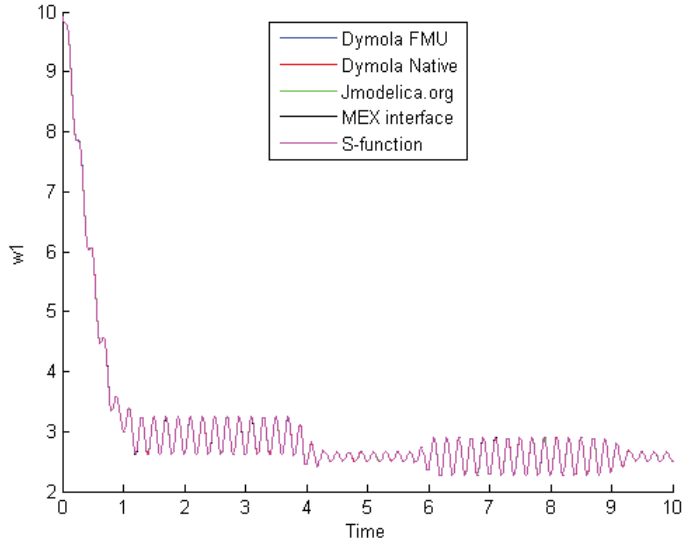

Figure 16: Simulation results of Coupled clutches model variable w1, angular velocity of inertia 1

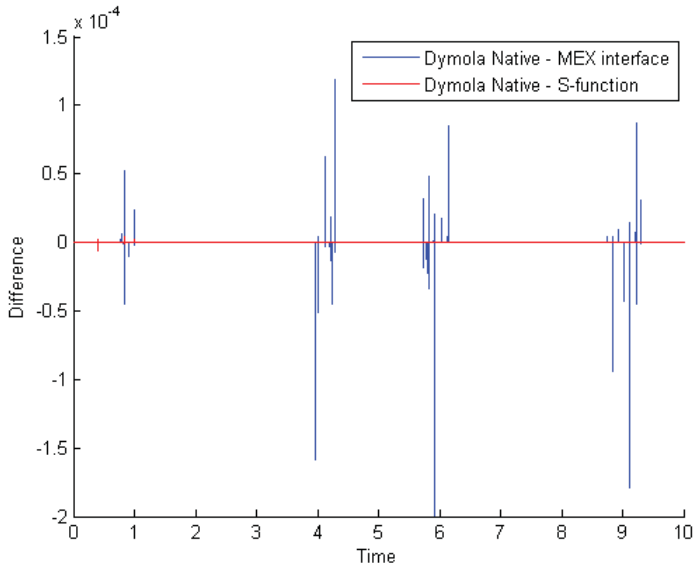

Figure 17: Interpolated results of variable w1 are differentiated

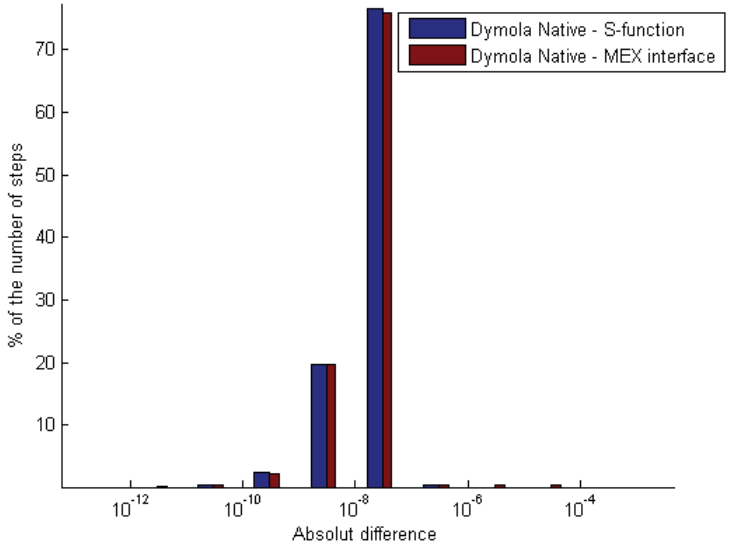

Figure 18: Absolute value of the differentiated results are summarized in a histogram

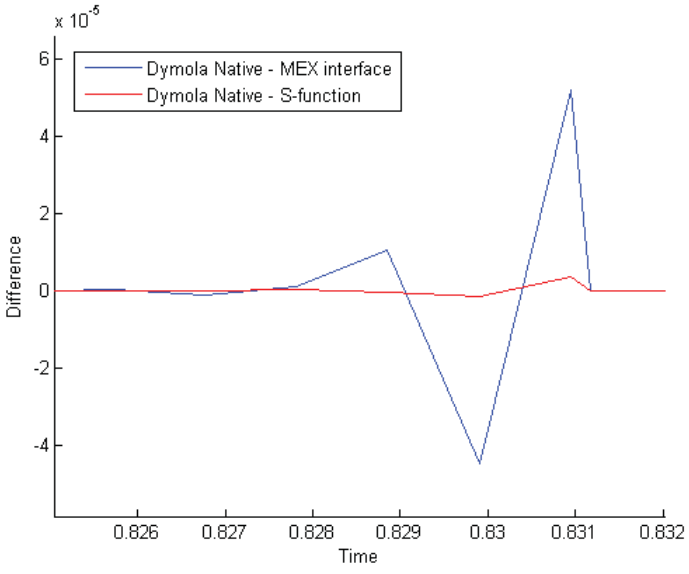

Figure 19: A *Spike* is zoomed. *Spike* is described here as the difference of the interpolated results around an event

6.2 Step events - Pendulum

The pendulum model used is a demonstration of dynamic state selection. This is a standard example for how model may change states because they are not numerical suitable any more. This model has 2 continuous states and step events that are triggered a couple of times during the simulation of 10 seconds. The variable we chose to compare the results with, x, is one of the cartesian coordinates of the pendulum weight. In Figure 20 we see the whole simulation. The results are not noticeable different. In Figure 21 we see the difference of results after interpolation. We see that the S-function is more alike the Dymola Native simulation than what the MEX interface is. The MEX interface seem to have spike like differences seen in Figure 21. We can see how this effect the spread in the difference in Figure 22 where we have steps for differences greater than 1e-7. If we zoom, see Figure 23, it can be seen that these spikes do all have the same characteristics of starting to "oscillating" just before the event and then disappear afterwards. This characterize the S-function as well if we would zoom further. If we for the moment neglect the biggest differences we see that the difference is in the order of 1e-7. I would say that this is satisfying with respect to the low tolerances and the interpolation, and that this would indicate that the MEX interface and the S-function are correctly implemented.

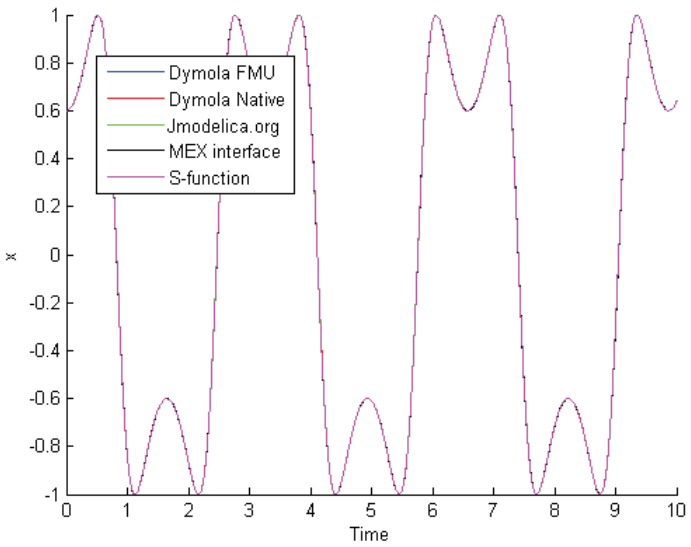

Figure 20: Simulation results of Pendulum model cartesian coordinate x

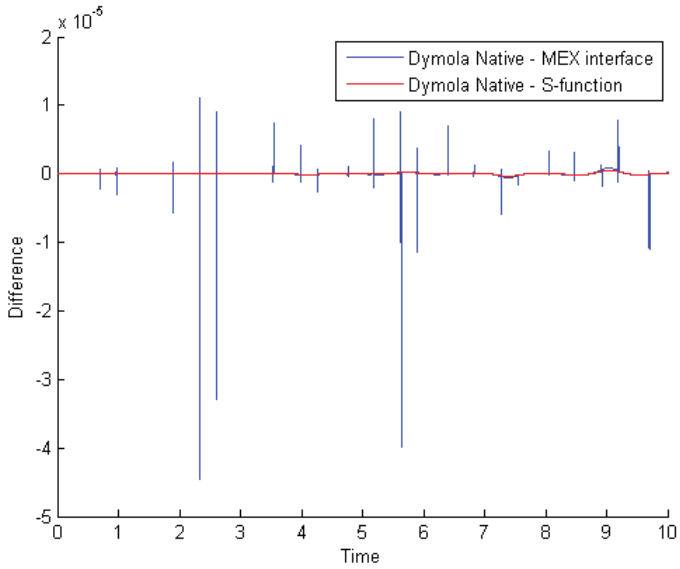

Figure 21: Interpolated results of variable x are differentiated

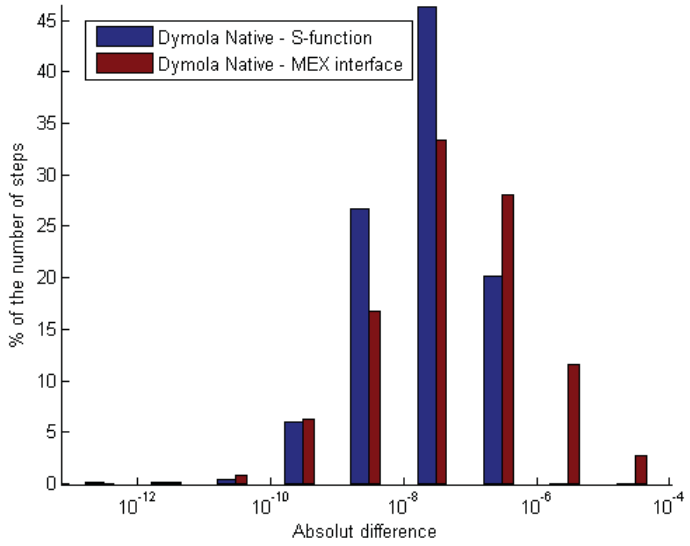

Figure 22: Absolute value of the differentiated results are summarized in a histogram

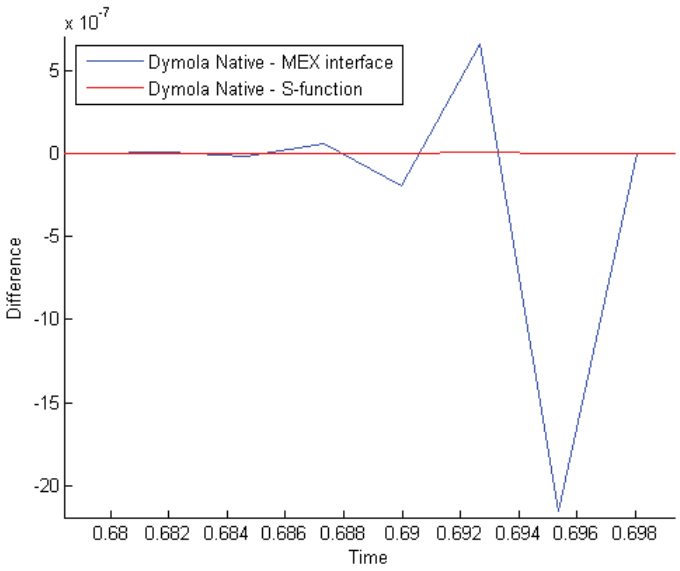

Figure 23: A *Spike* is zoomed. *Spike* is described here as the difference of the interpolated results around an event

6.3 Inputs - Mechanics model

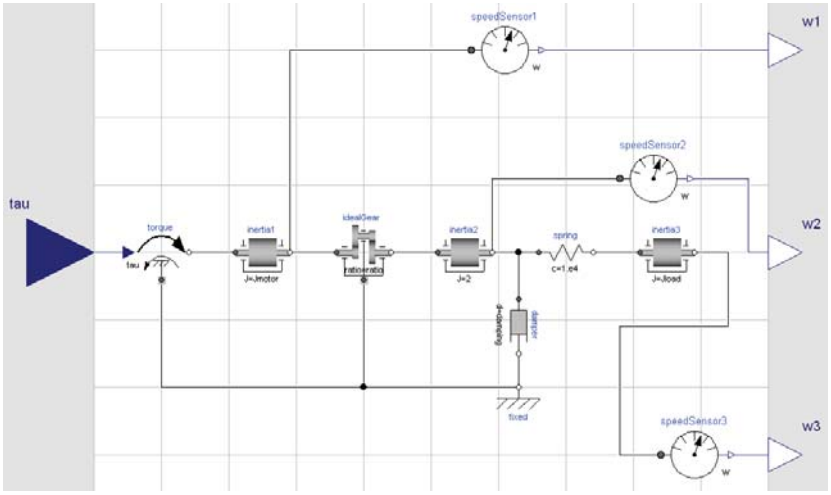

Figure 24: Block diagram the Mechanics model in Dymola

This is a model we named Mechanics, see Figure 24, and is a model of a drive train consisting of a motor inertia which is driven by a sine-wave motor torque. Via a gearbox the rotational energy is transmitted to a load inertia. Elasticity in the gearbox is modelled by a spring element. A linear

damper is used to model the damping in the gearbox bearing. This is a model without events with a top level input which we will focus on. As input signal to the motor torque we used a sinus signal. The input signal is generated for those times the solver do calculations. This was simulated for 10 seconds and we show the angular velocity of inertia 1, w1, in Figure 25. We can just note that the results can not be distinguished by the naked eye. We now interpolate the results and take the difference in Figure 26 to see that the solutions really are different. In Figure 27 we can more comfortable see that the S-function and the MEX interface seems to give very similar results. The maximum differences from the Dymola Native result seems to be in the order of 1e-6. Noteworthy is that we do not have any spikes as in the models with events.

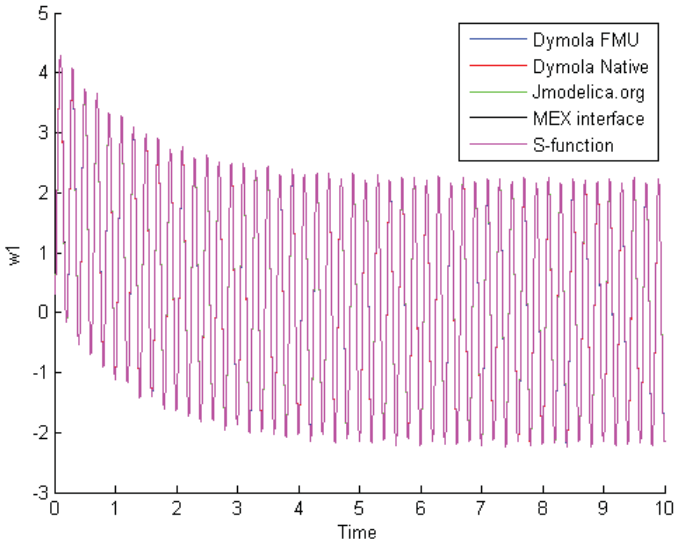

Figure 25: Simulation results of the Mechanics model variable w1, angular velocity of inertia 1

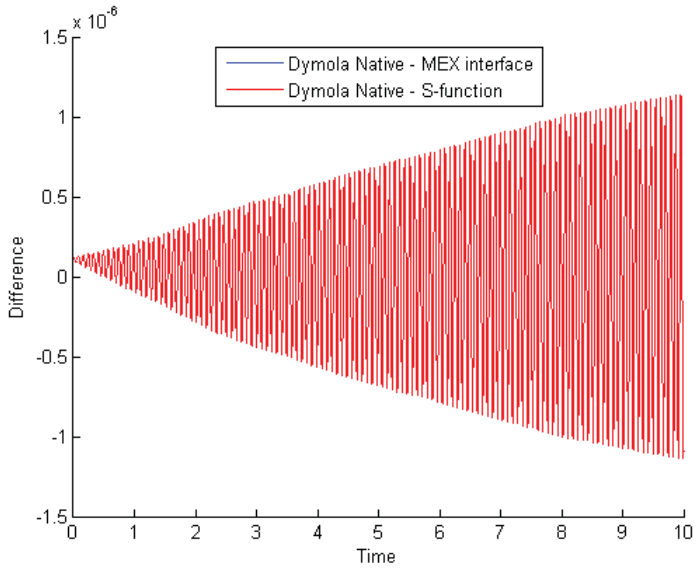

Figure 26: Interpolated results of variable w1 are differentiated

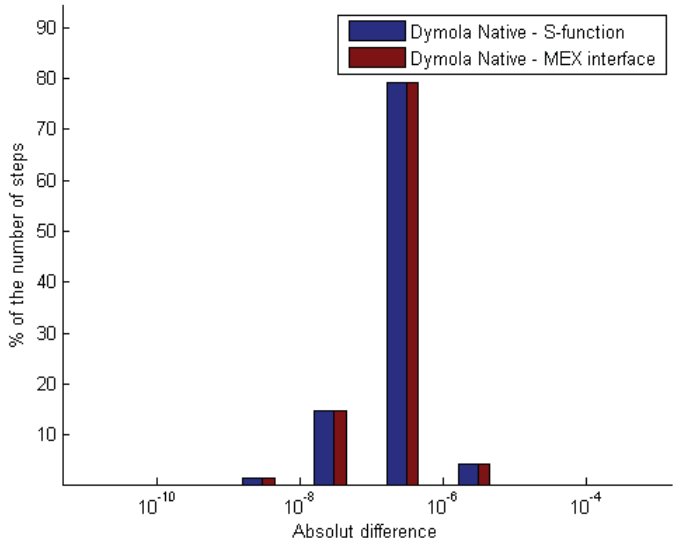

Figure 27: Absolute value of the differentiated results are summarized in a histogram

6.4 Algebraic loop - Feedback model

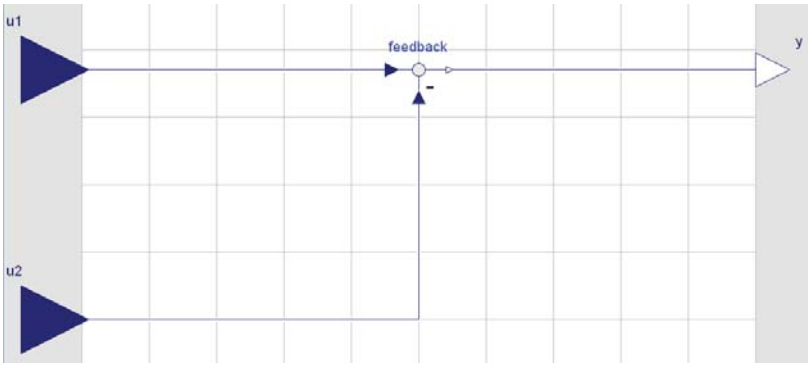

Figure 28: Dymola model exported as FMU and referred to as the Feedback model

The feedback model is used to verify that algebraic loops are solved. The model exported as an FMU is seen in Figure 28. The FMU subtracts two inputs that becomes the output as in Figure 3. The simulation is then set up with the FMU with one input signal generated from a sinus and the other input is connected directly to the output of the FMU. This set up generates an algebraic loop. Since there is no support for solving algebraic loops in neither of JModelica.org or the MEX interface, the results from these are omitted and we added the exact solution instead. We were also forced to limit the step size in Simulink and force Dymola to produce equidistant points get enough result points to produce a nice plot. The results of the output variable y is seen with the exact solution in Figure 29. The first thing one might notice is that Dymola FMU is constant zero. This imply that Dymola did not solve the algebraic loop for the FMU. In Figure 30 we neglect the Dymola FMU results and compare the Dymola Native and the S-function results with the exact solution. Noticeable here is that the Dymola Native took only 19 steps when Simulink took 500 steps which may explain why Dymola Native produce some what more error in the result.

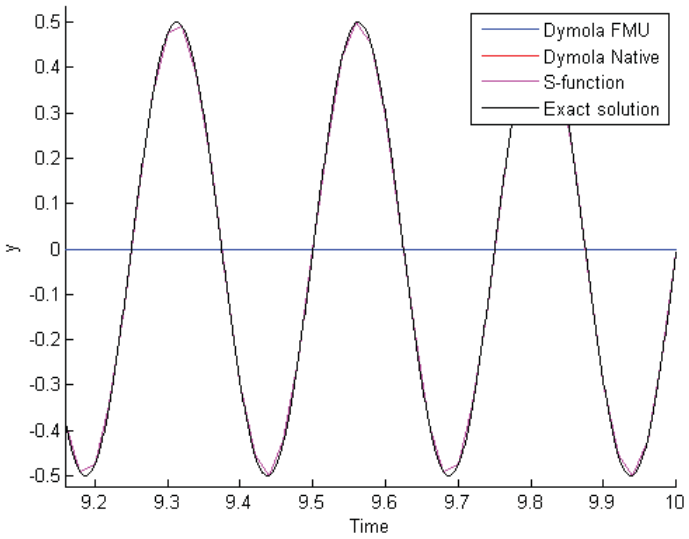

Figure 29: Simulation results of the output variable y from the Feedback model and an exact solution

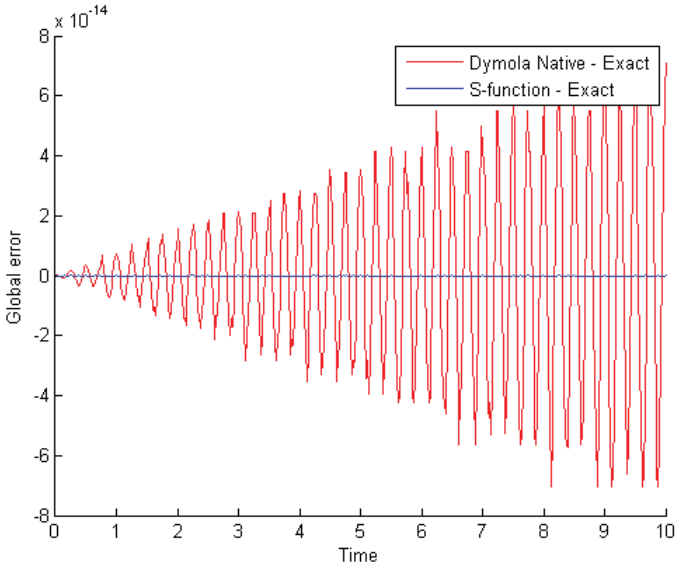

Figure 30: Global error of the simulation results

6.5 Performance - Robot model

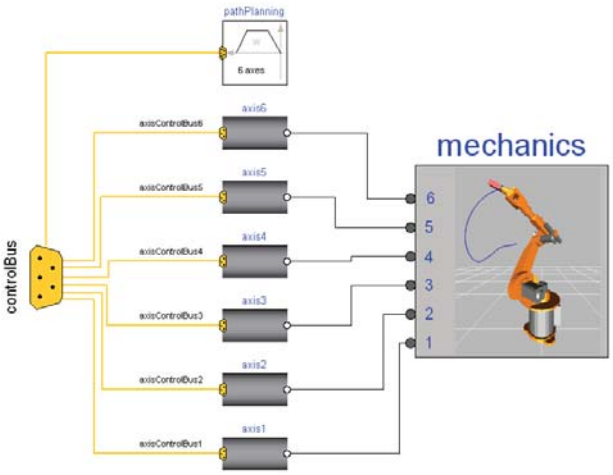

Figure 31: Block diagram of the Robot model in Dymola

Since we have simulated models with all listed properties from above for verification of the FMI calling sequence we now turn our interests to the performance. We simulate a big model such as the Robot example model in Figure 31 found in Dymolas MultiBody library. This model has over 7000 variables with 36 continuous states and 98 event indicators. The model is simulated to around 1.8 seconds and the angle of robot joint 1, phi1, is seen in Figure 32. We also give the difference between the Dymola Native results and the interpolated MEX interface and S-function results in Figure 33. We see that the differences is within the order of 1e-7.

We have summarized some of the logging data from the simulations in the table below. The table should just give a rough picture of the performances. The MEX interface and JModelica.org is the only one including the time for loading the FMU by unzipping and parsing the xml-file. For the MEX interface this takes less then 2s for the big robot model.

In the table below we have listed the simulation times and the number of function evaluations for the FMUs we discussed above and also added the TwinEvap. TwinEvap is a example model of a 2 evaporator system. This is a big and complex model with 130 continuous states and 1090 event indicators that show some different results in comparison.

The S-function produce results for the variables set as output of the block, which is only one in each of the simulations above. Depending on the efficiency of writing results, the performance may vary more or less depending on the number of results that are saved. We therefore choose to omit to pro-

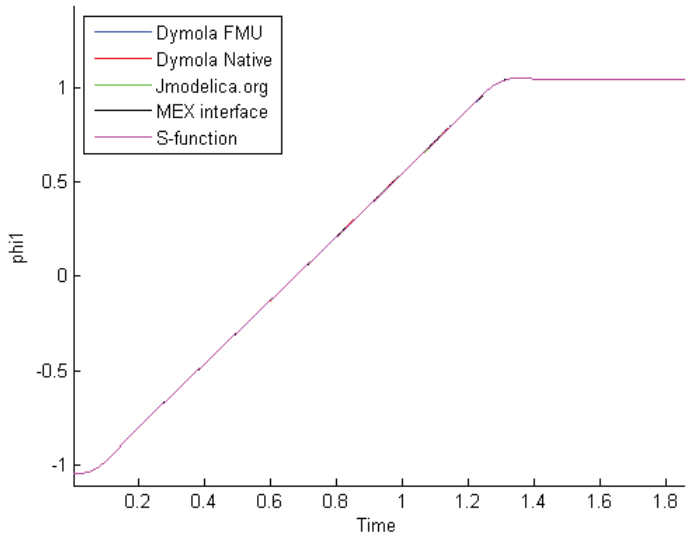

Figure 32: Simulation results of the Robot model's angle joint 1, phi1

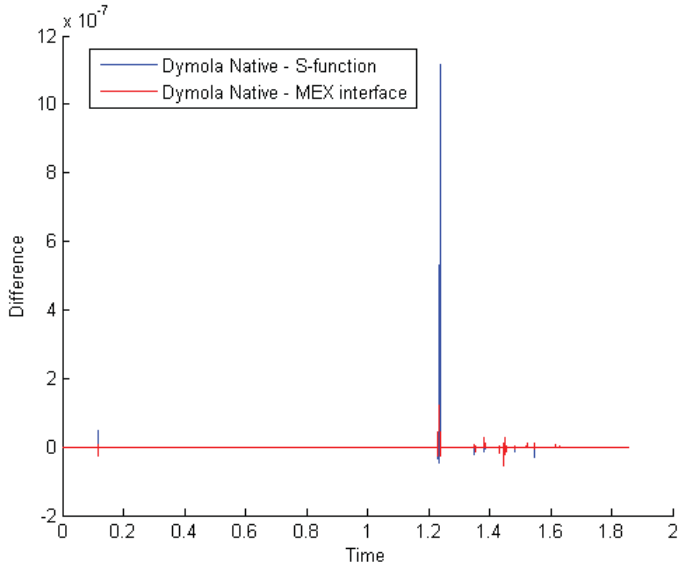

Figure 33: Interpolated results of variable phi1 are differentiated

duce the results for the MEX interface and JModelica.org when we compare the simulation times. But we unfairly let Dymola produce the results for all variables.

In both Dymola Native and Dymola FMU **F-evals** below corresponds to the

number of F-evaluations retrieved from the logging data from the simulation. In JModelica.org **F-evals** is the sum of *Number of Function Evaluations* and *Number of F-Eval During Jac-Eval* found in the simulation logging data. In the S-function and the MEX interface **F-evals** is the number of calls to the derivative function.

	Coupled clutches		Pendulum		Mechanics	
	F-evals	Time	F-evals	Time	F-evals	Time
Dymola Native	18455	0.3s	9587	0.2s	31250	0.3s
Dymola FMU	24815	0.5s	9385	0.3s	30996	0.4s
JModelica.org	12955	4.3s	4733	1.2s	22234	7.2s
MEX interface	25244	6.2s	5385	1.8s	42968	11.9s
S-function	25216	0.3s	5937	0.1s	42968	0.3s

	Feedback		Robot		Twin Evap	
	F-evals	Time	F-evals	Time	F-evals	Time
Dymola Native	37	0.08s	54467	3.3s	7884	79.5s
Dymola FMU	37	0.08s	64237	3.3s	31342	216s
JModelica.org	-	-	26233	8.8s	6543	77s
MEX interface	-	-	27319	8.9s	11580	123s
S-function	0	0.03s	28575	2.0s	11467	132s

6.6 State event in detail

With this model we want to enlighten the difficulties with event detection and at the same time strengthen the implementation of state event handling. This bouncing ball model comes with the QTronic SDK as C code and a script to compile and export it to an FMU. The model has two continuous states and one event indicator. The event indicator value is given as the hight of the ball plus a small perturbation (the perturbation is only used to make the event indicator different from 0 in accordance with the FMI standard). This means that the ball fall through the "floor" at every bounce and causes a domain switch of the event indicator. When the ball is beneath the floor the fmiEventUpdate is called and changes the direction of the velocity. The solver starts integrating again and then another domain switch occur since the ball is on the way up through the "floor". Therefore there is two domain switches at every bounce. With this model we can verify that the event update function is called at the correct right side of the zero-crossing. For the interested reader, we give an extraction of the model's C-code that is modified. r is a double vector containing all the real variables and *pos* is

a boolean vector used by the state event indicator. h is the hight variable, v is ht's velocity and e is the coefficient the velocity is multiplied with at bounces. These are all real variables. Notice the underscore defined macros that is the element number of the variable's value in the double vector r.

```
#define h_      0
#define v_      2
#define e_      4

#define EPS_IND 1e-14
fmiReal getEventIndicator(ModelInstance* comp, int z) {
    switch (z) {
        case 0 : return r(h_) + (pos(0) ? EPS_IND : -EPS_IND);
        default: return 0;
    }
}

void eventUpdate(ModelInstance* comp, fmiEventInfo* eventInfo) {
    if (pos(0)) {
        r(v_) = - r(e_) * r(v_);
    }
    pos(0) = r(h_) > 0;
    eventInfo->iterationConverged  = fmiTrue;
    eventInfo->stateValueReferencesChanged = fmiFalse;
    eventInfo->stateValuesChanged  = fmiTrue;
    eventInfo->terminateSimulation = fmiFalse;
    eventInfo->upcomingTimeEvent   = fmiFalse;
 }
```

Unfortunately Dymola relay on a static wrapper library to import the FMI functions. Therefore Dymola can not simulate the bouncingBall FMU and is omitted from this simulation. This is not in accordance with the FMI standard.

In Figure 34 we see the ball dropped from the hight 1 and is simulated for 4 seconds. Around 3 seconds we see how the ball fall trough the "floor". This is accordance with the model and the limitations of numerical computations when the right side of the domain switch is too far under the floor for the ball to bounce up through the floor again. We may also notice that results from JModelica.org, the S-function and MEX-interface are very similar. The dashed lines the results from using a relative tolerance of 1e-10 and an absolute tolerance of 1e-12. The solid lines the results from using a relative tolerance of 1e-12 and absolute tolerance of 1e-14.

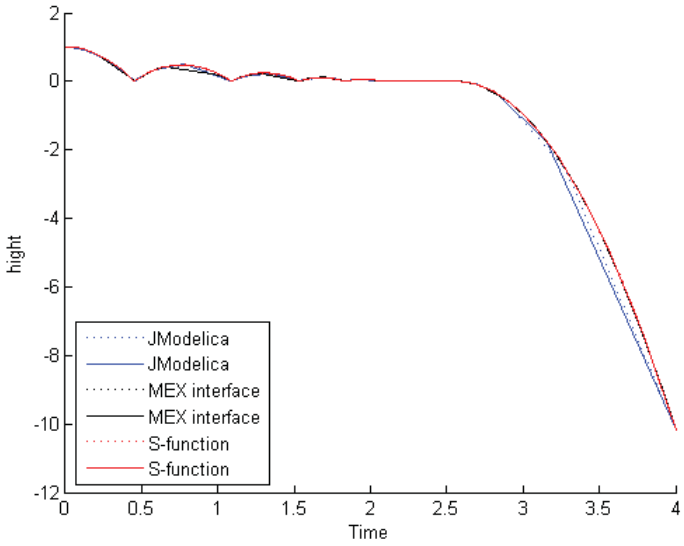

Figure 34: QTronics bouncing ball FMU simulated

In Figure 35 we have zoomed at the first bounce to show how different tolerances effect the state events.

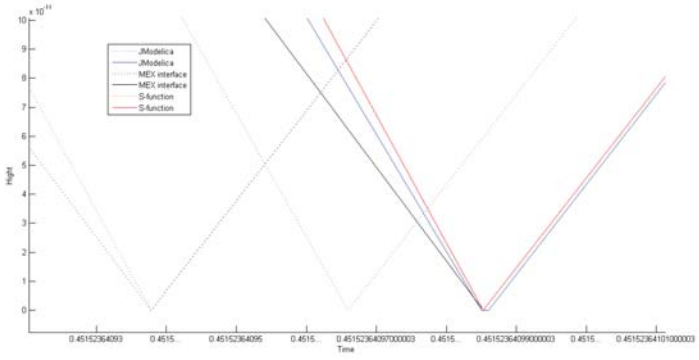

Figure 35: Close zoom up of a bounce

We zoom even more in Figure 36 and can now demonstrate how the state events are handled. The MEX-interface detects both zero crossings and integrate to the right side of these. However the S-function integrates up to the left side of the event and then to the right side. Therefore we see an extra step at both zero crossings. Notice that the parturition of the event indicator causes the ball to cross the floor even though the event indicator indicates it is above the floor when the ball is falling downwards and vice

versa on the way up.

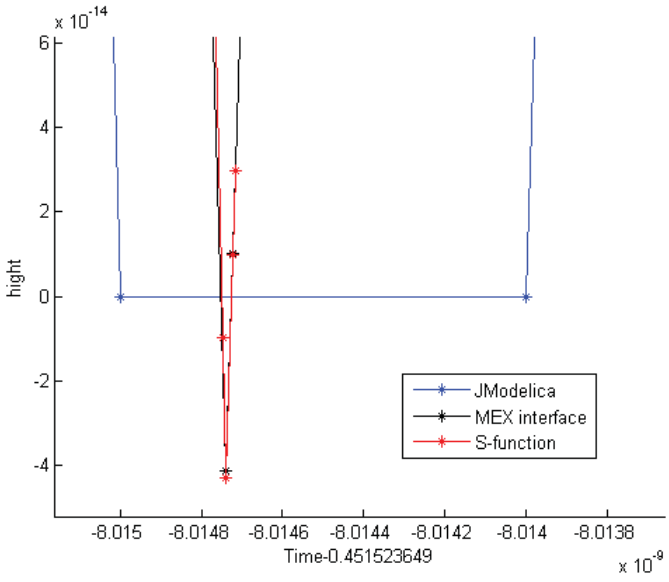

Figure 36: The S-function integrates up to the left side of the event and then steps over and continues at the right side. The MEX-interface integrates up to the right side of the event and then continuous.

7 Summary and conclusions

7.1 Simulation results

The results from the verification of a correctly implemented FMI calling sequence are satisfying. At first sight from the result figures of the whole simulations, all simulation environments gave such results that they could not be distinguished. When we then interpolated the results to compare the difference we saw that they really were different. We then saw some spikes occurring around events. To show that this is most likely to be caused by the interpolation the difference results from the Coupled clutches are showed one more time here, zoomed at a time event occurring at the time 0.4 seconds. We now also added the interpolated difference of the same results which we have limited to only contain the results up to left edge of the time event. This is seen in Figure 37. The results from the limited interpolated difference is

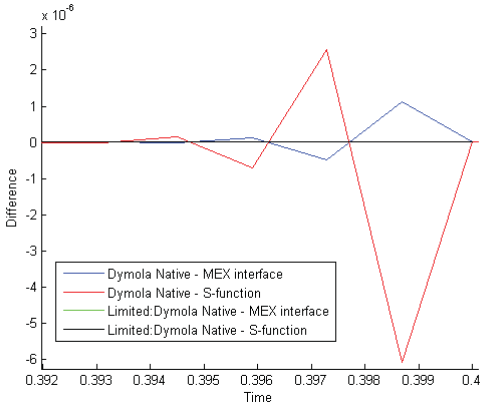

Figure 37: Interpolated results differentiated

in the order of 1e-9 which is more similar to the difference that is in between events. At the right side of the event the difference is in the order of 1e-9 for the unlimited results. This means that if we could stop the interpolation over the event we would have a maximum difference of 1e-9 instead of 1e-4. If all the spikes is caused by this interpolation phenomenon, we may explain some of the lower staples with higher absolute difference in the histograms like in the Figure 18 from the Coupled clutches model.

Now when we tried to explain the spikes in the result comparisons we have not found any other notable deviations that would indicate that the MEX interface or the S-function is not implemented correct. The differences we have seen in the results are possible to explain by the use of different solvers and how the tolerances may effect the solvers differently. My conclusion is that the S-function and the MEX interface are correctly implemented.

7.2 Performance

In tables on page 50 we could see the time it took for the solvers to complete the simulations and how many function evaluations were needed. The different simulation configurations and how the result generation affect the times and results differently makes a really fear comparison hard. The reader should keep this in mind in the next section.

If we consider the simulation times we see that the S-function is faster then the MEX interface for all simulations except for the TwinEvap model. The S-function is expected to be faster than the MEX interface since it is a compiled block used in a compiling simulator. The MEX interface uses the MATLAB scripting language, that is often slower than a compiling language, between the FMI function calls.

The S-function seems to qualify in the same performance league as JModelica.org and Dymola. For the minor models, the S-function, Dymola Native and Dymola FMU is almost equally fast. For the really big model TwinEvap, JModelica.org and Dymola Native is almost twice as fast as the rest. We should notice that the rest also do twice as many function evaluations. The MEX interface is slightly slower then the other simulators for the minor models but performs better then the S-function for the big TwinEvap model. I have no good explanation for this behavior. I tested to simulate the same model in MATLAB 2010a, Simulink 7.5. The simulation time for the MEX Interface was 200 seconds and for the S-function 181 seconds. This is more reasonable and the expected results.

The conclusion from this is that both the S-function and the MEX interface are fast enough to be an alternative for simulating FMU models.

7.3 Models used during the development

The models we used here is not all the models that were used during the development. One group of models that were left out were the four FMUs from the Qtronics SDK. These four FMUs were used in the first stage of development and where referred to be fundamental for being solved correctly by both the S-function and the MEX-interface. One of these FMUs, inc.fmu, is a pure discrete model with no continuous sates or event indicators. Some other models exported from Dymola were also tested.

7.4 Optimization

So far the focus of the implementation has been on getting a correct and a so robust implementation as possible. But if we would like to optimize the

S-function a little we could investigate if the Simstruct has any flag that can be used to notify a zero-crossing event in mdlOutput rather then to check all the event indicators if indicator has switched domain. If this is not found an alternative could be to introduce a flag that is set by a check if the simulator takes a minor step back in time in mdlZeroCrossing. This would indicate that the simulator has detected a domain switch and is trying to accurately detect where the event occurred. This last proposal would work for the MEX interface as well.

In the MEX interface one could try to create MEX functions that combine the wrapper FMI functions that are now called after each other in the MATLAB code. This would first of all minimize the overhead of running MATLAB code and also eliminate some procedures that is repeated in every MEX file to call the real FMI function. To optimize the input handling one could try to write a MEX function that perform the function evaluations and that sets the values to the model.

7.5 Linux implementation

An implementation of the S-function and the MEX interface would probably be easy since we already have the Windows implementations based on the QTronics windows SDK. There is an SDK adapted from the QTronics windows version by Michael Tiller [5] found through the FMI homepage [1]. This could be used to make the changes that are needed such as unziping and loading the FMI functions.

7.6 My reflections

What is not discussed in this theses is the different states the development been through before it became what it is now. The most time consuming thing with this project has been the creation of the GUI. This was developed continuously with increasing number of functionalities. This was a very good experience for building bigger programs that must be designed for future developments, even though I learned the hard way. At some point the whole GUI was remade due to is has become too complex and hard to overview. Now afterwards I think a Java GUI could have been a serious alternative due to both the development tools available for this and the possibility to create better looking GUIs. There is still some work to do in the GUI and the S-function before it can be used for real, for example the output tab is still in some development phase.

What the reader also miss is what is done without contributing to the S-function and MEX interface implementation. For example the investigation

whether or not the cell renderer in the GUI trees were available to use to make nice aligned variable values. Or if the callback function mdlGetTimeOfNextVarHit could be used to hit time events exactly in the S-function. The answers to these questions are hard to find in the MATLAB documentation and were tested in simple examples. Once I needed to ask MathWorks for technical support for a phenomenon that even they thought was hard to find a answer for.

Some of the implementation is still not found in any documentation like the triggering of the mdlInitializeSizes when *set_param* is used. But the writing of this thesis has contributed to better documentation or even finding documentation at all for the implementation.

The development has also led to a few bug reports to *Dassault Systemes Lund* regarding the FMU export mechanism. Dymola already knew of some of the bugs, some were new and for some am I still waiting for response. In QTronics SDK a minor bug in the models were found.

References

[1] Functional Mock-up download site - 9 sep 2010.
http://functional-mockup-interface.org/fmi.html

[2] ITEA2 project MODELISAR, FMU project profile pdf - 9 sep 2010.
http://www.itea2.org/public/project_leaflets/MODELISAR_profile_oct-08.pdf

[3] FMI support in tools - 9 nov 2010.
http://www.functional-mockup-interface.org/tools.html

[4] QTronic FMU SDK - 23 nov 2010.
http://www.qtronic.de/en/fmusdk.html

[5] Michael Tiller adapted QTronics FMU SDK - 25 nov 2010.
https://github.com/mtiller/fmusdk

[6] How Simulink Works, Modeling Dynamic Systems - 23 nov 2010.
http://www.mathworks.com/help/toolbox/simulink/ug/f7-20739.html

[7] FMI document version 1.0 - 25 nov 2010.
http://www.functional-mockup-interface.org/fmi.html

[8] MATLAB Using Callback Functions - 12 okt 2010.
http://www.mathworks.com/help/toolbox/simulink/ug/f4-122589.html

[9] MATLAB How the Simulink Engine Interacts with C S-Functions - 18 sep 2010.
http://www.mathworks.com/help/toolbox/simulink/sfg/f8-37326.html

[10] MATLAB MEX-files Guide - 9 sep 2010.
http://www.mathworks.com/support/tech-notes/1600/1605.html#ingredients

[11] MATLAB Controlling and Displaying the Sorted Order - 18 sep 2010.
http://www.mathworks.com/help/toolbox/simulink/ug/f13-91940.html#brbj4u4

[12] MATLAB ssSetInputPortDirectFeedThrough - 18 sep 2010.
http://www.mathworks.com/help/toolbox/simulink/sfg/sssetinputportdirectfeedthrough.html

[13] MATLAB mdlInitializeSizes - 18 sep 2010.
http://www.mathworks.com/help/toolbox/simulink/sfg/mdlinitializesizes.html

[14] MATLAB mdlStart - 18 sep 2010.
http://www.mathworks.com/help/toolbox/simulink/sfg/mdlstart.html

[15] MATLAB mdlZeroCrossings - 18 sep 2010.
http://www.mathworks.se/help/toolbox/simulink/sfg/mdlzerocrossings.html

[16] MATLAB mdlOutputs - 18 sep 2010.
http://www.mathworks.com/help/toolbox/simulink/sfg/mdloutputs.html

[17] MATLAB mdlDerivatives - 18 sep 2010.
http://www.mathworks.com/help/toolbox/simulink/sfg/mdlderivatives.html

[18] MATLAB mdlTerminate - 18 sep 2010.
http://www.mathworks.com/help/toolbox/simulink/sfg/mdlterminate.html

[19] MATLAB ode initial value problem solvers - 21 okt 2010.
http://www.mathworks.com/help/techdoc/ref/ode23.html

[20] MATLAB odeset solvers - 21 okt 2010.
http://www.mathworks.com/help/techdoc/ref/odeset.html