Interfacing Functional Mock-up Units in Modelica

Simon Cöster

Master's thesis
2014:E11

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Numerical Analysis

CENTRUM SCIENTIARUM MATHEMATICARUM

Abstract

Different simulation tools have their own definition of how a model is represented. This causes complications when modeling in one tool and trying to simulate in another, or if one wants to verify simulation results in another tool. This thesis focuses on a way of interfacing the Functional Mock-up Interface (FMI) into Modelica models. Modelica is an open standard modeling language for modeling physicals model, such as electrical circuits, drive trains etc. The Functional Mock-up Interface is an interface which provides tool independent C-functions for execution of models.

In this thesis we make use of the FMI specification and Modelica's external function interface to generate a complete Modelica model. The results shows that the implementation works quite well, but with some decrease of performance.

Preface

This report is my Master Thesis for my conclusion of a degree in Master of Science in Engineering Physics at Lund Institute of Technology. It is a Master Thesis in Numerical Analysis and all the work has been performed at Modelon AB in Lund, Sweden.

My supervisor was Johan Åkesson and my assistant supervisor was Bengt-Arne Andersson, both employed at Modelon AB. Examiner was Claus Führer, professor at Numerical Analysis at Lund University.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Purpose

The purpose of this Master Thesis is to investigate how, if possible, to interface Functional Mock-up Interface (FMI) into Modelica models. There are several challenges in this task; how should the code be generated and more importantly, how to map the Functional Mock-up Interface into the Modelica model to get numerical correct results. The goal is to follow the Modelica language specification and the FMI specification as long as possible and to create a tool independent solution.

1.2 Delimitations

In this thesis, the focus is on FMI version 1.0. Additionally, only the Model Exchange feature is studied and the Co-Simulation part is omitted. This thesis also focuses on FMUs delivered with only a dynamic linked library, ergo source code FMUs are not covered. One other delimitation is that the program shall follow Modelica specification 3.2r2 [3].

1.3 Layout of this report

First, a brief background to the FMI is presented. The theory section addresses the Functional Mock-Up Interface with its mathematical description and goals. Also, relevant information about the Modelica language such as external functions and packages are addressed. Then follows a short description on a compiler-independent build process using CMake. Program requirements and a description of the different models that are tested are included in the Method-section. The results are then presented followed by a discussion of the implementation.

2 Background

Different simulation tools usually have their own way to represent models, e.g. different programming languages and data storage. This becomes a problem when users want to create models in one tool, but run the simulation in another. This problem is the underlying reason for introducing the Functional Mock-up Interface. In 2010, the first version of FMI was released, FMI 1.0. The intention is to transform a dynamic system to an FMU (Functional Mock-up Unit) that implements the FMI (Functional Mock-up Interface). A FMU can then be simulated in another tool by calling the FMI functions. There are two features of the FMI; Co-Simulation (CS) where the FMU is self integrating (the FMU is packed with a ODE solver) or Model Exchange (ME) where the FMU requires a ODE solver to perform the integration and simulation.

3 Theory

3.1 FMI

FMI - Functional Mock-up Interface is a tool independent standard for Model Exchange (ME) or Co-Simulation (CO) [4]. It is developed under the MODELISAR consortium. For model exchange the idea is that a modeling tool can generate C-code of a dynamic system that can be used by another tool. The executable that implements the interface is called a Functional Mock-up Unit (FMU). The interface consists of 24 C-functions and type definitions. These 24 functions is all that is needed to instantiate, initialize and run the simulation in the target simulator. The 24 functions are [4]

- fmiGetBoolean
- fmiGetInteger
- fmiGetReal
- fmiGetString
- fmiInitialize
- fmiInstantiateModel
- fmiFreeModelInstance
- fmiSetTime
- fmiSetBoolean
- fmiSetInteger
- fmiSetReal
- fmiSetString
- fmiGetContinuousStates
- fmiGetNominalContinuousStates
- fmiSetContinuousStates
- fmiGetDerivatives
- fmiGetEventIndicators
- fmiEventUpdate
- fmiCompletedIntegratorStep
- fmiGetStateValueReferences
- fmiGetModelTypesPlatform
- fmiTerminate
- fmiGetVersion
- fmiSetDebugLogging

The FMU is distributed in a zip-file which contains at most three parts [4]:

- A Model Description File (xml-file)
- Dynamic Linked Libraries, *.dll (Windows) or *.so (unix) and sometimes the C-source code
- Other model data such as model icons, tables and documentation

3.1.1 Model Interface

The model description file contains all the information about the model parameters and variables. C-source code and DLL can both be provided in the FMU, but at least one of them *must* be distributed.

The interface consist of two header files; `fmiModelTypes.h` and `fmiModelFunctions.h`, see Appendix C.1 and Appendix C.2.
In `fmiModelTypes.h`, all the type definitions for input and output of the FMI-specific functions are defined. Below, "standard32", a standard 32-bit platform is defined [4].

```
typedef void*                fmiComponent;
typedef unsigned int         fmiValueReference;
typedef double               fmiReal;
typedef int                  fmiInteger;
typedef char                 fmiBoolean;
typedef const char*          fmiString ;
#define fmiTrue 1
#define fmiFalse 0
#define fmiUndefinedValueReference(fmiValueReference)(-1)
```

The `fmiComponent` defines a pointer to a model specific data structure. The type `fmiValueReference` defines a unique (at least with respect to the corresponding base type) handle to a variable of the model. `fmiReal` defines a real number (64 bits), `fmiInteger` defines a integer number (32 bits), `fmiBoolean` defines a boolean number (8 bits) and `fmiString` defines a character string. Also, if `fmiValueReference` is undefined, it gets the value of `fmiUndefinedValueReference` [4]. The file `fmiModelFunctions.h` includes the header `fmiModelTypes.h` and declares all the FMI-specific functions mentioned above. In order to be able to use several models in one executable, the functions must have unique function names. In `fmiModelFunctions.h` macros are defined, that changes the actual name of the functions. This is made in the following manner [4].

```
#define MODEL_IDENTIFIER MyFMU
#include "fmiModelFunctions.h"

fmiStatus fmiSetTime(...){
...
}
```

The macro `MODEL_IDENTIFIER` is used in the macros of `fmiModelFunctions.h` to change the name of `fmiSetTime` to `MyFMU_fmiSetTime`. The `MODEL_IDENTIFIER` for each model can be found in the model description file under the attribute `modelIdentifier`. An enumeration is also defined in `fmiModelFunctions.h` that defines a status flag that is returned by all FMI functions [4].

```
typedef enum {fmiOK,
 fmiWarning,
 fmiDiscard,
 fmiError,
 fmiFatal
} fmiStatus;
```

In table 1, these statuses are explained.

Status	**Description**
`fmiOK`	The function call was problem free
`fmiWarning`	There were some complications in the function but the simulation can proceed. The function `logger` shall show a message.
`fmiDiscard`	This status is only available if it is explicitly defined for `fmiSetReal`, `fmiSetContinuousStates`, `fmiGetReal`, `fmiGetDerivatives` and `fmiGetEventIndicators`. If the function returns with this status, it is recommended that the solver shall use a smaller step size. The `logger` shall show a message.
`fmiError`	The function discovered an error and the simulation cannot continue.
`fmiFatal`	The computations are corrupted for all model instances.

Table 1: *Return values of the FMI functions.*

3.1.2 Mathematical description

The aim of the Model Exchange interface is to solve a system of hybrid ordinary differential equations numerically. This system is a *piecewise continuous system* which means that discontinuities can occur at different time instants [4]. These time instants, $t_0, t_1, ..., t_n$, are called *events*. If an event is known before hand, it is called a *time event* and if an event is defined implicitly it is called a *step event* or *state event*. In figure 1 below, a FMU instance is shown and the data flow between the solver and the instance is described with arrows [4].

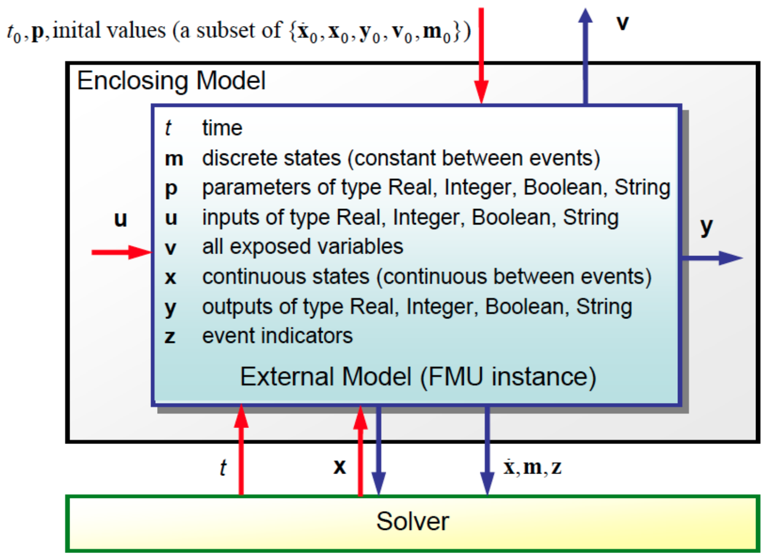

t_0, **p**, inital values (a subset of $\{\dot{\mathbf{x}}_0, \mathbf{x}_0, \mathbf{y}_0, \mathbf{v}_0, \mathbf{m}_0\}$)
v
Enclosing Model
t time
m discrete states (constant between events)
p parameters of type Real, Integer, Boolean, String
u inputs of type Real, Integer, Boolean, String
v all exposed variables
x continuous states (continuous between events)
y outputs of type Real, Integer, Boolean, String
z event indicators
u
y
External Model (FMU instance)
t
x
$\dot{\mathbf{x}}, \mathbf{m}, \mathbf{z}$
Solver

Figure 1: *The figure shows a FMU instance and the data flow.* [4]

$\mathbf{x}(t)$ is a vector of time-continuous states and inside every interval $\mathbf{x}(t)$ is a continuous function of time. Constant states are represented by a vector $\mathbf{m}(t)$, which is a set of real, integer, string and logical variables. $\mathbf{m}(t)$ only changes at events and is constant between them. Events are defined by the following conditions [4]:

1. A *time event*, t_i is defined in the previous event t_{i-1} either by the FMU or by the environment.
2. A *state event* occurs when an *event indicator* changes its domain from $z_i > 0$ to $z_i \leq 0$ or the other way round. This is also known as *zero-crossing*. All event indicators are piecewise continuous and are real numbers collected in a vector $\mathbf{z}(t)$. See figure 2.
3. *Step events* occurs if `fmiCompletedIntegratorStep` returns with `callEventUpdate = fmiTrue`. Such an event can be e.g. dynamic state selection, where the previous states are no longer suited numerically.

Events are never triggered inside the FMU, so it is the simulation environment's responsibility to handle the event triggering. [4].

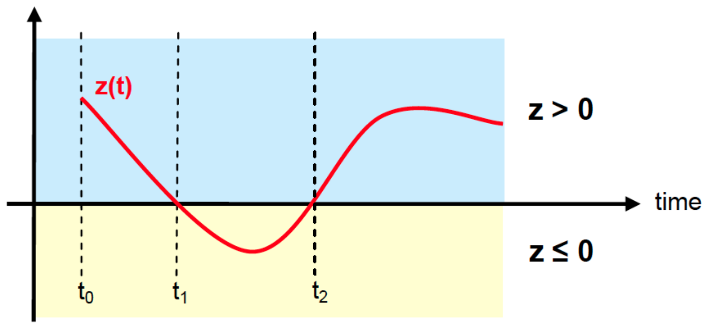


Figure 2: *An event indicator changes its domain.* [4]

3.1.3 Calling sequence

In figure 3 below the calling sequence of the FMI functions is shown in an UML State Machine[1].

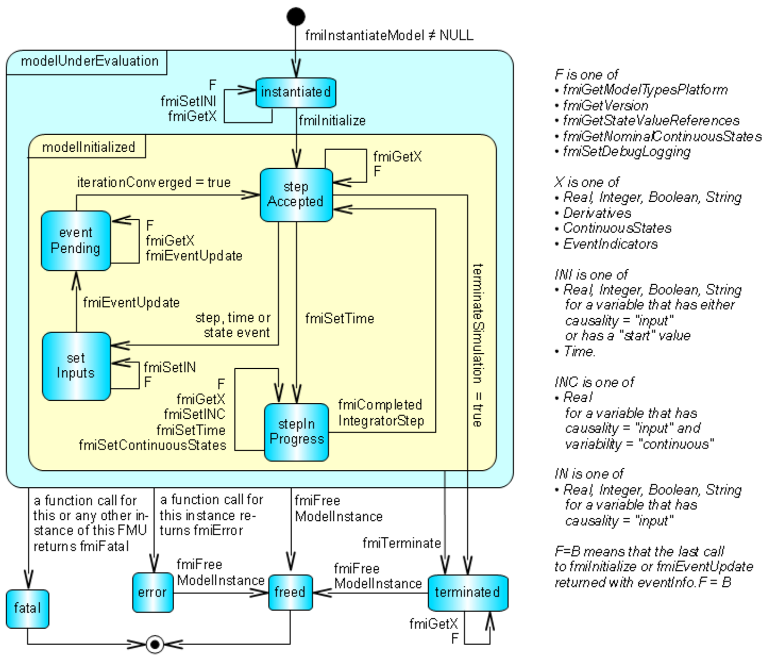

fmiInstantiateModel ≠ NULL
modelUnderEvaluation
instantiated
F
fmiSetINI
fmiGetX
fmiInitialize
modelInitialized
iterationConverged = true
step Accepted
fmiGetX
F
event Pending
F
fmiGetX
fmiEventUpdate
fmiEventUpdate
step, time or state event
fmiSetTime
terminateSimulation = true
set Inputs
fmiSetIN
F
F
fmiGetX
fmiSetINC
fmiSetTime
fmiSetContinuousStates
stepIn Progress
fmiCompleted IntegratorStep
a function call for this or any other instance of this FMU returns fmiFatal
a function call for this instance returns fmiError
fmiFree ModelInstance
fmiTerminate
fmiFree ModelInstance
fmiFree ModelInstance
fatal
error
freed
terminated
fmiGetX
F
F is one of
• fmiGetModelTypesPlatform
• fmiGetVersion
• fmiGetStateValueReferences
• fmiGetNominalContinuousStates
• fmiSetDebugLogging
X is one of
• Real, Integer, Boolean, String
• Derivatives
• ContinuousStates
• EventIndicators
INI is one of
• Real, Integer, Boolean, String for a variable that has either causality = "input" or has a "start" value
• Time.
INC is one of
• Real for a variable that has causality = "input" and variability = "continuous"
IN is one of
• Real, Integer, Boolean, String for a variable that has causality = "input"
F=B means that the last call to fmiInitialize or fmiEventUpdate returned with eventInfo.F = B

Figure 3: *The figure shows the UML State Machine of a FMU.* [4]

Every implementation must support this calling sequence. At **instantiated**, the inputs and start values can be set. The model needs to be initialized using `fmiInitialize` to reach the state **step Accepted**. The solution can be retrieved at initial time, after a completed integrator step or after an event iteration.

[1]UML State Machine - Unified Modeling Language State Machine is a way of visualize the way a program (or physical device) works. The program can be in only one state for a given time.

If `fmiInitialize` or `fmiEventUpdate` returs with `eventInfo.terminated = fmiTrue` the model should be terminated properly. After `fmiSetTime` has been called, the model is in the **step in process** state, where an integrator step is performed. When an integrator step is completed the function `fmiCompletedIntegratorStep` must be called. If the environment has detected an event, first the inputs must be set before the event iteration starts [4].

3.1.4 Model Description Scheme

Except from the model equations, all information needed about the model is stored in a XML-file. The top level of the XML-scheme is shown in figure 4.

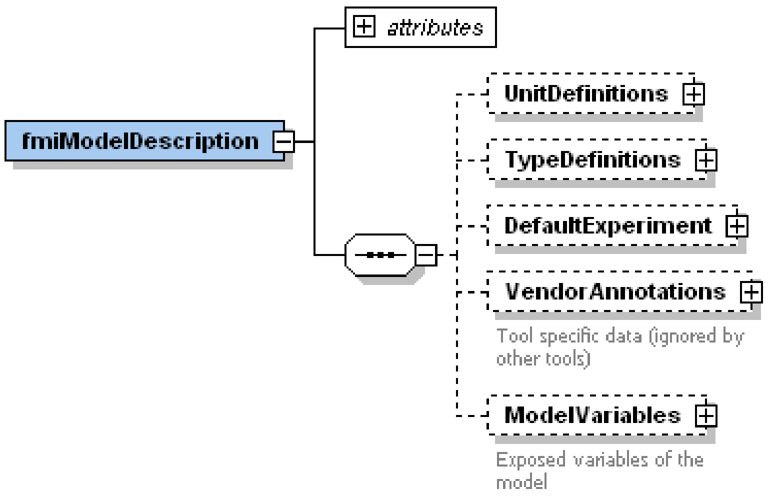


Figure 4: *Top level description schema.* [4]

The element **attributes** defines all the global properties of the model, see figure 5. **UnitDefinitions** is a list of definitions of units and **TypeDefinitions** is a list of type definitions. **VendorAnnotations** defines vendor specific data but can be ignored by other tools. **ModelVariables** are all the variables of the model [4].

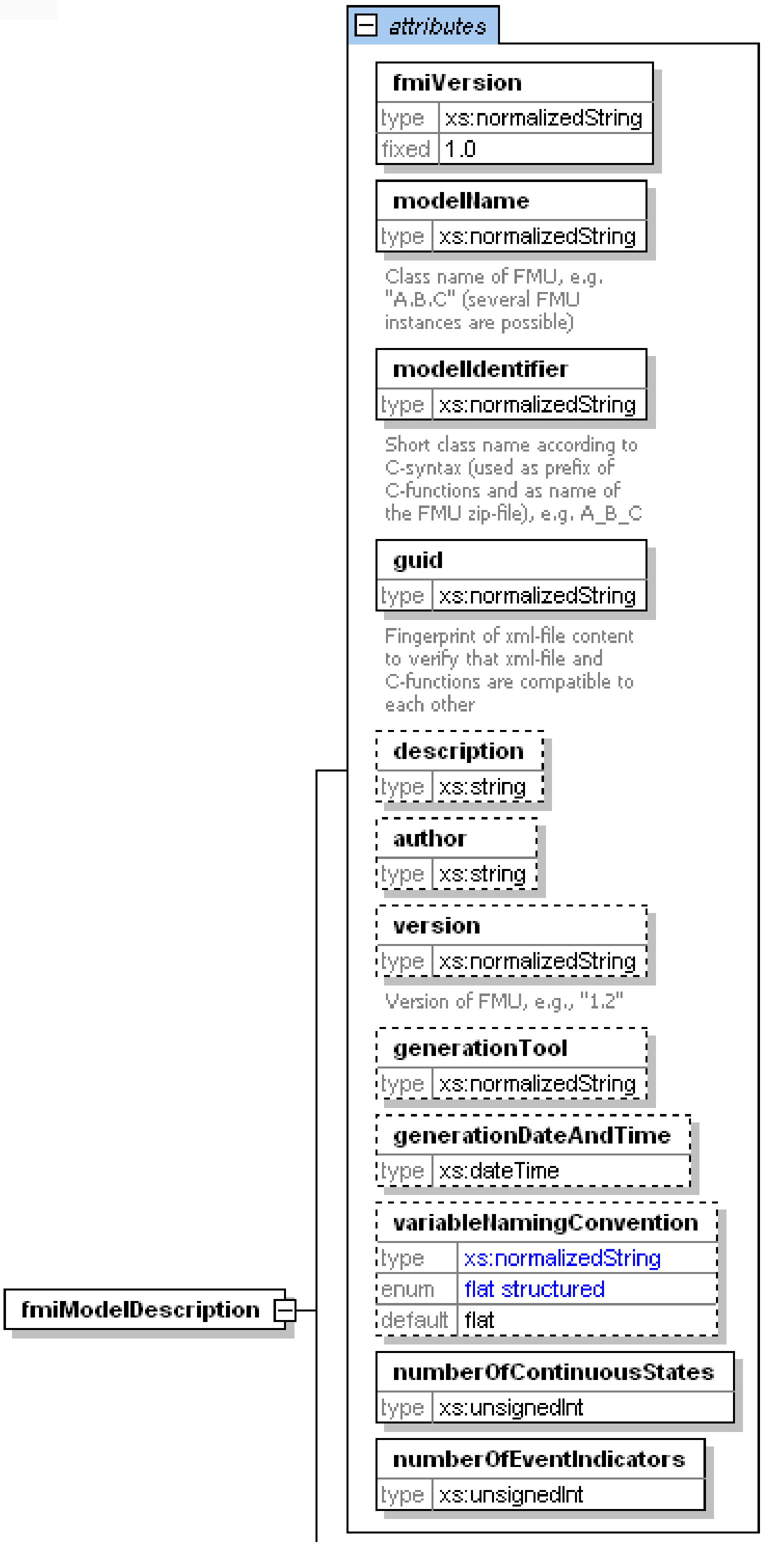


Figure 5: *The attributes of fmiModelDescription.* [4]

In figure 5 all attributes of the model are gathered. The `GUID` (Globally Unique IDentifier)[2], is a string that checks the compatibility between the XML-file and the C-functions provided.

All model variables are defined in the `ModelVariables` element which only consists of a set `ScalarVariable`. The attributes of a `ScalarVariable` can be seen in figure 6.

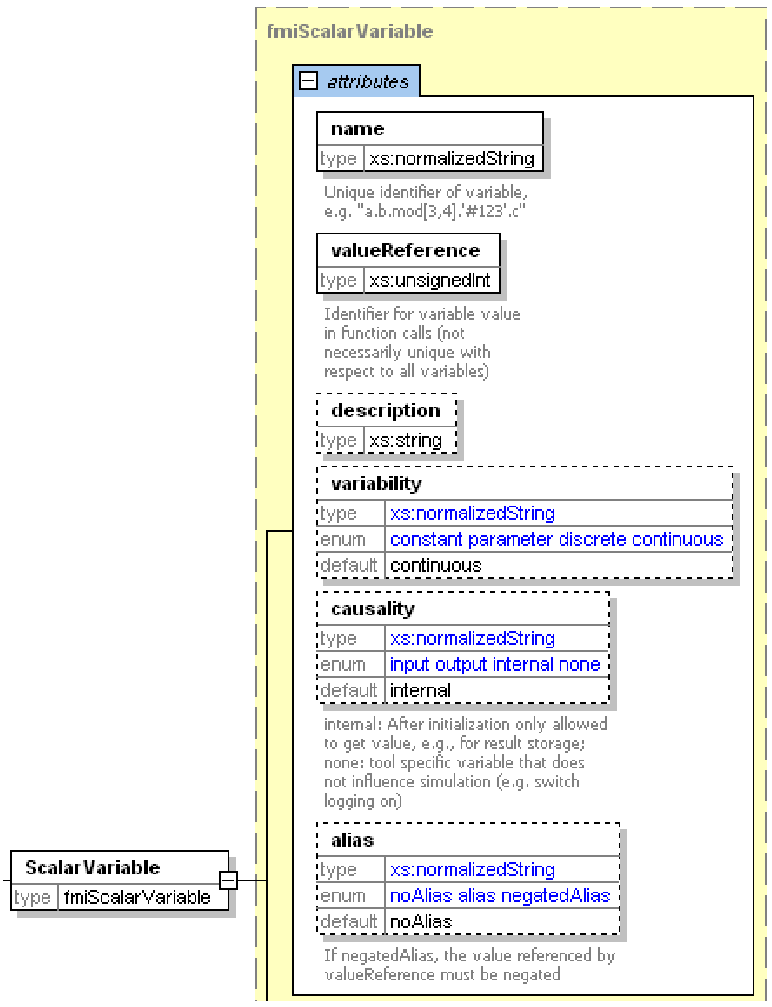

fmiScalarVariable
attributes
name
type xs:normalizedString
Unique identifier of variable, e.g. "a.b.mod[3,4].'#123'.c"
valueReference
type xs:unsignedInt
Identifier for variable value in function calls (not necessarily unique with respect to all variables)
description
type xs:string
variability
type xs:normalizedString
enum constant parameter discrete continuous
default continuous
causality
type xs:normalizedString
enum input output internal none
default internal
internal: After initialization only allowed to get value, e.g., for result storage; none: tool specific variable that does not influence simulation (e.g. switch logging on)
alias
type xs:normalizedString
enum noAlias alias negatedAlias
default noAlias
If negatedAlias, the value referenced by valueReference must be negated
ScalarVariable
type fmiScalarVariable

Figure 6: *Some of the attributes of a ScalarVariable.* [4]

The attribute `name` is the full name of the variable and it is unique within the model. The attribute `valueReference` is a handle to identify the variables value in the interface. This reference is unique for a specific base type (Real, Integer, String, Boolean) [4].
The definition of when a variable is allowed to be changed is represented by the attribute `variability`. Values of this enumeration

[2]For more information about Unique Identifiers see http://msdn.microsoft.com/en-us/library/aa373931(VS.85).aspx, retrieved 2014-02-01

are [4]:

- **constant**: The value of the variable does not change.
- **parameter**: The value of the variable does not change after initialization.
- **continuous**: The value can be changed in any way, no restrictions. Only Real variables can be Continuous.
- **discrete**: The value only change at events.

When connecting FMUs to each other, information about the `causality` is needed. The `causality` can take four different values [4]:

- **input**: the variable is an input to the model.
- **output**: the variable is an output of the model.
- **internal**: the variable cannot be used in a connection.
- **none**: tool specific variable.

A variable can be an alias variable. An alias variable occurs when physical components are connected to each other, e.g. the assignment $a := b$ and for efficiency reasons a is removed and replaced with b. The default value for a variable is `noAlias`. If a variable has attribute `alias` it is an alias variable and can be achieved by its `valueReference`. If a variable has attribute `negatedAlias` the value retrieved by its `valueReference` must be negated [4]. An example of a XML-file is shown in Appendix A.1.

3.1.5 The FMI functions

The FMI defines 24 C-functions that interact with the FMU in different ways.

```
fmiComponent fmiInstantiateModel(fmiString instanceName,
                 fmiString GUID,
                 fmiCallbackFunctions functions,
                 fmiBoolean loggingOn);
```

This function returns a new instance of the model. If the function returns a null pointer, the instantiation failed and no other function can be called. The argument `instanceName` is the name of the instance and the argument `GUID` is the *Globally Unique IDentifier* as mentioned in section 3.1.4. Argument `functions` is an instance of `fmiCallbackFunctions` which is described below. `loggingOn` can be either `fmiTrue` or `fmiFalse` and enables/disables debug logging respectively [4].

```
void fmiFreeModelInstance(fmiComponent c);
```

The function above is used to deallocate all the memory for the instance `c`. Switching between `loggingOn = fmiTrue`/`loggingOn = fmiFalse` can be achieved by using the function

```
fmiStatus fmiSetDebugLogging(fmiComponent c,
                        fmiBoolean loggingOn);
```

These three functions deal with creation and destruction of model instances and to set the desired debugging option [4].

In `fmiInstantiateModel`, an argument `function` is present, which is an instance of `fmiCallbackFunctions`. This is a struct defined in the following manner:

```
typedef struct {
  void  (*logger)(fmiComponent c, fmiString instanceName,
                  fmiStatus status, fmiString category,
                  fmiString message, ...);
  void* (*allocateMemory)(size_t nobj, size_t size);
  void  (*freeMemory)(void* obj);
} fmiCallbackFunctions
```

This struct holds function pointers to three functions. The first, `logger`, is a function that is called in the model, usually when the model function encounters some problem. The other two function pointers handles memory allocation and deallocation [4]. In C, those will point to e.g. `calloc` and `free` respectively.

During the simulation, different variables needs to be computed and re-initialized. The following functions handles this type of interaction with the model.

```
fmiStatus fmiSetTime(fmiComponent c, fmiReal time);
```

This function sets the new time for the instance and re-initializes variables that depend on the time [4].

```
fmiStatus fmiSetContinuousStates(fmiComponent c,
                const fmiReal x[], size_t nx);
```

This function sets a new continuous states vector, `x`, with size `nx` and re-initializes caching of variables that are state dependent.

```
fmiStatus fmiCompletedIntegratorStep(fmiComponent c,
                        fmiBoolean* callEventUpdate);
```

The environment must call this function after every completed integrator step. The function `fmiEventUpdate` must be called if `fmiCompletedIntegratorStep` returns with `callEventUpdate = fmiTrue` [4].

```
fmiStatus fmiSetXXX(fmiComponent c,
                    const fmiValueReferences vr[],
                    size_t nvr, const fmiXXX value[]);
```

Inputs, start values and independent variables are set with the function `fmiSetXXX`. Note that `XXX` can be any of `Real`, `Integer`, `Boolean` or `String`. The `ValueReferences` are passed to the function as `vr`, with `nvr` number of values. The argument `value` is a vector containing the values that shall be set. [4]

The following functions are used to evaluate the model equations.

```
fmiStatus fmiInitialize(fmiComponent c,
                        fmiBoolean toleranceControlled,
                        fmiReal relativeTolerance,
                        fmiEventInfo* eventInfo);
```

This function initializes the model, which means that it computes the start values for all the variables. `fmiSetTime` must be called before this function. Furthermore, variables with a start attribute in the ModelDescription file can be set before the model is initialized. The function returns with `eventInfo` once the initialization is done. [4]

The structure `fmiEventInfo` is used in two functions; `fmiInitialize` and `fmiEventUpdate`. This structure is defined as [4]

```
typedef struct {
  fmiBoolean iterationConverged;
  fmiBoolean stateValueReferencesChanged;
  fmiBoolean stateValuesChanged;

  fmiBoolean terminateSimulation;
  fmiBoolean upcomingTimeEvent;
  fmiReal        nextEventTime;
} fmiEventInfo;
```

The top three variables in this structure are only meaningful for `fmiEventUpdate` and will always be `fmiTrue` for `fmiInitialize`. If `terminateSimulation = fmiTrue`, the simulation shall successfully be terminated. If `upcomingTimeEvent = true`, the simulator shall integrate to `nextEventTime` at most. Another event can occur before `nextEventTime`, e.g. a step event, then `nextEventTime` becomes meaningless. If `upcomingTimeEvent = fmiFalse` there are no time events in the model. [4]

```
fmiStatus fmiEventUpdate(fmiComponent c,
                         fmiBoolean intermediateResults,
                         fmiEventInfo* eventInfo);
```

After an event has been triggered, this function shall be called and will return with `eventInfo`. This function uses the three other variables in `fmiEventInfo`. It shall be called until

`eventInfo->iterationConverged = fmiTrue`.
If `eventInfo->stateValuesChanged = fmiTrue`, then the new states have to be achieved with the function `fmiGetContinuousStates`.
If `eventInfo.stateValueReferencesChanged = fmiTrue` the meaning of the states has changed. [4]

```
fmiStatus fmiGetDerivatives(fmiComponent c,
                            fmiReal derivatives[],
                            size_t nx);
```

Returns the states derivatives for the current states.
The element `derivative[i]` is corresponding to the derivative of the state `x[i]`.

```
fmiStatus fmiGetEventIndicators(fmiComponent c,
                           fmiReal eventIndicators[],
                           size_t ni);
```

Retrieves the vector containing the event indicators of the model.

```
fmiStatus fmiGetXXX(fmiComponent c,
                           const fmiValueReference vr[],
                           size_t nvr,
                           fmiXXX value[]);
```

Gets the values of the variables with corresponding `ValueReference`.

```
fmiStatus fmiGetContinuousStates(fmiComponent c,
                                    fmiReal x[],
                                    size_t nx);
```

Retrives the vector of continuous states. This function has to be called after initialization and after an event iteration and the states have changed indicated by
`eventInfo->stateValuesChanged = fmiTrue`. [4]

```
fmiStatus fmiGetNominalContinuousStates(fmiComponent c,
                                   fmiReal x_nominal[],
                                   size_t nx);
```

Returns the nominal values of the continuous states.

```
fmiStatus fmiGetStateValueReferences(fmiComponent c,
                                   fmiValueReference vrx[],
                                   size_t nx);
```

Returns the vector containing the value references of the state vector.

```
fmiStatus fmiTerminate(fmiComponent c);
```

Terminates the simulation. Also releases all the memory that has been allocated during the simulation.

3.2 Modelica

Modelica is an Object Orientated programming language for modeling of physical systems [3]. Modelica Standard Library defines different components such as electrical circuits, drive trains, hydraulic circuits etc. These components contains connectors which can be connected between components. It is mainly a *declarative* programming language, meaning the program expresses the logic of computation without any information about the computation flow, it is the compiler's responsibility to chose the order of computation. Furthermore, a Modelica tool is free to evaluate expression several times in an integrator step and omit evaluation of variables if their value doesn't influence the result.This is in contrast to imperative programming which describes statements that should be performed after each other. [3]

3.2.1 Equations, algorithms and when clauses

In Modelica, equations are listed inside an `equation`-block. As mentioned before, it is the Modelica compiler's task to decide in which order the expressions of an `equation`-block should be calculated. However, Modelica also offers an imperative way of stating relationships and expressions, in `algorithm`-blocks. This works essentially like `equation`-block, but the order is defined from top to bottom.

Before any calculations can be carried out, initialization takes place to assign values to all variables in the model. Constrains to determine the initial value of variables can be set using the word `initial` before either `equation` or `algorithm`. The initial value for a variable can be set using the attribute `start = someExpression`.

Events are triggered using the `when`-statement. A `when`-statement has the following syntax [3]:

```
when expression then
  { statement ";" }
{ elsewhen expression then
  { statement ";" } }
end when;
```

Statements inside a `when`-statement are active only when an element in the `expression` vector becomes true. `when`-statements cannot be nested and they are not allowed inside while, if or for-clauses. Furthermore, they cannot occur inside functions. At an event instant, the continuous states should be reinitialized. Modelica offers the `reinit`-operator that only can be used inside a `when`-statement. The operator has the following syntax [3]:

```
reinit(x, expression);
```

This operator reinitializes `x` with `expression`. `x` must be of type `Real` or an array of `Real` variables. Below, an example written in Modelica is shown.

```
model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Boolean impact;
  Real v_new;
 equation
  impact = h <= 0;
  der(v) = if flying then -g else 0;
  der(h) = v;
  when {impact, h <= 0 and v <= 0} then
   v_new = if edge(impact) then -e*pre(v) else 0;
   flying = v_new > 0;
   reinit(v, v_new);
  end when;
end BouncingBall;
```

The operator `der` is used to represent the symbolical derivative of the variable its operating on, and the `pre` operator returns the value of the variable after the last event iteration. The `edge(a)` operator expands to the Boolean expression `a and not pre(a).` [3] This model gets the result presented in figure 7.

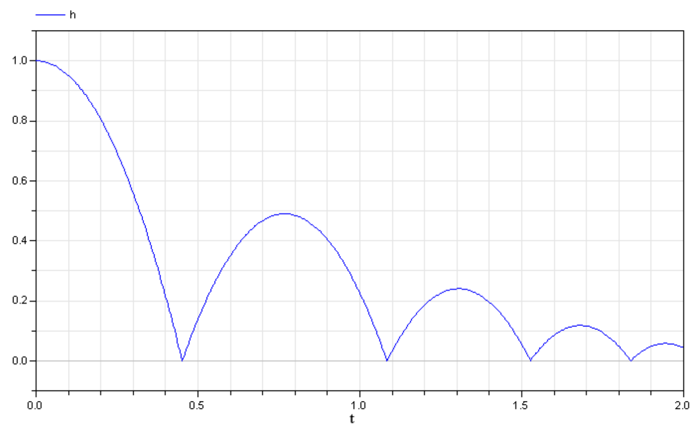


Figure 7: *The figure shows the height of the ball in model* BouncingBall.

3.2.2 External Functions

An interface for external function calls is provided in the Modelica language. This interface provides support for external functions written in C and FORTRAN 77 and mapping of types between Modelica and the external language. The declaration of a external function is defined in [3]:

```
function IDENT string_comment
  { component_clause ";" }
  [ protected { component_clause ";" } ]
external[language_specification] [external_function_call]
  [annotation ] ";"
  [annotation ";"]
end IDENT;
```

Here, statements inside `[]` are optional and statements inside `{}` can be repeated zero or several times. By default, C is chosen as external language and it is not necessary to specify this as the `language_specification`. In the public part of the declaration, all the components must be declared either as `input` or `output`. Annotations is used to include necessary header files or include different libraries. [3] The annotations for interacting with external functions are:

- `annotation(Library="NameOfLibrary")`
- `annotation(Library={"NameOfLibrary1", "NameOfLibrary2"})`
- `annotation(Include="includeDirective")`
- `annotation(IncludeDirectory="modelica://URI/to/IncludeDir")`
- `annotation(LibraryDirectory="modelica://URI/to/LibraryDir")`

The default URI[3] to the `IncludeDir` is
`modelica://LibraryName/Resources /Include`,
and to `LibraryDir` is
`modelica://LibraryName/Resources /Library` [3]. More about packages and libraries in section 3.2.3. External functions can use internal memory if the function is defined as an instance of the built in Modelica class `ExternalObject`. This is a *partial* class which means that no instance can be defined by this class, it has to be used as an extension to `ExternalObjects`. The external object class that extends `ExternalObject` must contain a `constructor` and a `destructor`. External functions can operate on the class that extends `ExternalObject`. If an `ExternalObject` is used as input or output of and external function written in C, it is mapped to `void*`. [3] To be able to use external functions, some argument type mapping is essential, see table 2 and 3.

Modelica	*C*	
	input	*output*
`Real`	`double`	`double*`
`Integer`	`int`	`int*`
`Boolean`	`int`	`int*`
`String`	`const char*`	`const char**`
`Enumeration Type`	`int`	`int*`
`size(..., ...)`	`size_t`	`size_t`

Table 2: *Argument mapping types.* [3]

[3]URI = Uniform Resource Identifier, see http://www.w3.org/TR/uri-clarification/

The type returning from an external function is mapped according to table 3.

Modelica	*C*
`Real`	`double`
`Integer`	`int`
`Boolean`	`int`
`String`	`const char*`
`Enumeration Type`	`int`

Table 3: *Return types.* [3]

Vectors can only be passed as argument to an external function and not as a return type. When passing a vector to an external C-function, the syntax in table 4 must be followed.

Modelica	*C*
	Input and Output
`T[n]`	`T'*, size_t n`
`T[n,m]`	`T'*, size_t n, size_t m`

Table 4: *Mapping of vectors.* [3]

In table 4, the type `T` must be one of the simple types, `Real`, `Integer`, `Boolean` or `String` and the mapped type `T'` must follow the mapping types defined in table 2 under the *Input* column.

3.2.3 Packages

A Modelica package is basically a namespace for classes, functions and other definitions. It can be used to avoid name collisions and to map class structures to a hierarchical File System [3].
There are two types of packages:

- Structured entities
- Non-structured entities

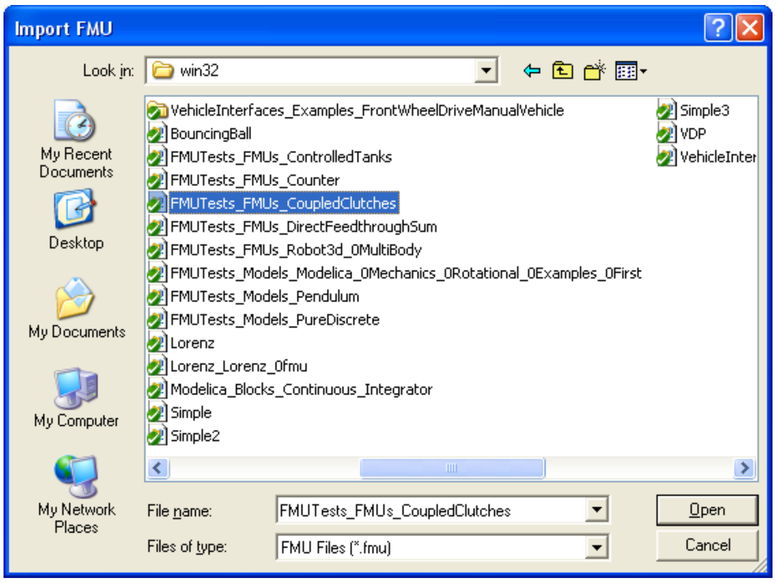
A non-structured entity is represented by a file `classname.mo` and shall only contain definition of a class `classname`. A structured entity are represented as a file structure with a directory `packagename` with a file `package.mo` in it. In `package.mo` a package with a name `packagename` shall be defined. Using this structure with a package containing external functions enables the user to easily include header files or use libraries with `annotation` defined in section 3.2.2. [3]

3.3 CMake

CMake is a cross-platform, open source, build system. CMake creates makefiles for Unix based systems and Visual Studio Projects for Windows based system. All commands are listed in a file `CMakeLists.txt` which is used to generate the project. To every project one can add executables, libraries and add include directories etc. [2] In this project, CMake was used to generate platform independent projects and to add the external library FMIL (Functional Mock-up Interface Library) provided by Modelon AB. [2] It is outside the scope of this Thesis to go deeper on this topic.

4 Pre-study

Several simulation tools offers an 'Import FMU' feature which generates Modelica models ready to be simulated. Two of those, Dymola and OpenModelica, has been studied. Both of these tools offers import of FMU into their modeling environment. In Dymola, one can either choose the 'Import FMU' function or just drag the FMU file to the window and release and the generation will begin. The import feature is shown in figure 8



VehicleInterfaces_Examples_FrontWheelDriveManualVehicle
BouncingBall
FMUTests_FMUs_ControlledTanks
FMUTests_FMUs_Counter
FMUTests_FMUs_CoupledClutches
FMUTests_FMUs_DirectFeedthroughSum
FMUTests_FMUs_Robot3d_0MultiBody
FMUTests_Models_Modelica_0Mechanics_0Rotational_0Examples_0First
FMUTests_Models_Pendulum
FMUTests_Models_PureDiscrete
Lorenz
Lorenz_Lorenz_0fmu
Modelica_Blocks_Continuous_Integrator
Simple
Simple2
Simple3
VDP
VehicleInter

File name: FMUTests_FMUs_CoupledClutches
Files of type: FMU Files (*.fmu)

Figure 8: *FMU import feature of Dymola.*

This generates declarations of all the variables (including parameters and enumerations) in the model. A package `fmi_Functions` is defined with some of the FMI functions and some help functions. There is not a complete mapping of the FMI functions. All functions are calling external C-functions, but the annotation is not including a particular C file or library. Instead, in the annotation Dymola uses the identifier `Header` and generates all the code for the specific C-function inside the annotation. An example for the function `fmiSetTime` is shown below:

```
function fmiSetTime
      input fmiModel fmi;
      input Real ti;
      input Real preAvailable;
      output Real postAvailable=preAvailable;
    external "C"
        BouncingBall_fmiSetTime2(fmi, ti);
      annotation (Header =    "
#ifndef BouncingBallFMI_C
#define BouncingBallFMI_C 1
#include \"FMI/fmiModelFunctions_.h\"
#include \"FMI/fmiImport.h\"
#endif
#ifndef BouncingBall_SetTime_C
#define BouncingBall_SetTime_C 1
#include <stdlib.h>
void BouncingBall_fmiSetTime2(void*m, double ti) {
  struct dy_Extended*a=m;
  fmiStatus status=fmiFatal;
  if (a) {
    a->dyTime=ti;
    status=a->dyFmiSetTime(a->m, ti);
  }
  if (status!=fmiOK && status!=fmiWarning) {
   ModelicaError(\"SetTime failed\");
  }
}
#endif", Library="BouncingBall",
LibraryDirectory="modelica://BouncingBall_fmu/
Resources/Library/FMU/binaries");
end fmiSetTime;
```

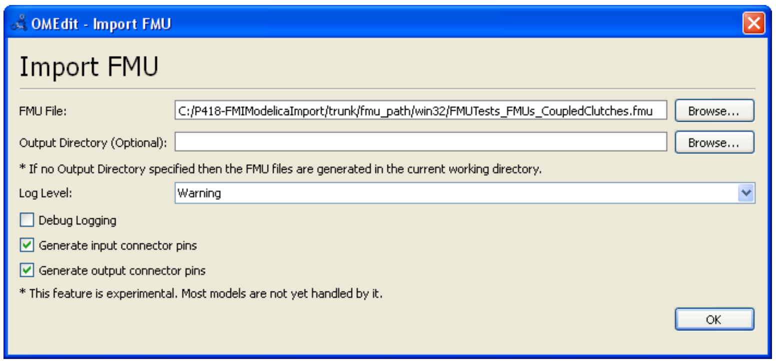
One can see that Dymola uses Modelicas `ExternalFunction` to hold internal memory of the FMU. A structure `dy_Extended` is used in the wrapper function to typecast the object from Modelica. After a NULL pointer check, the call to the FMI function is performed.

The calling sequence is the same for every model generated by Dymola. Initial values and parameters are set in `initial algorithm` and the model is initialized under `initial equation`. After that, the main calling sequence is generated as:

```
equation
  when initial() then
    fmi = fmi_Functions.fmiModel(fmi_instanceName,
                                    fmi_loggingOn);
  end when;
  fmi_ATime = fmi_Functions.fmiSetTime(fmi, time, 1);
  fmi_AStates = fmi_Functions.fmiSetContinuousStates(
                  fmi, fmi_x, fmi_AParamsAndStart +
                  fmi_Initialized + fmi_ATime);
  der(fmi_x) = fmi_Functions.fmiGetDerivatives(fmi,
                size(fmi_x, 1), fmi_AStatesInputs);
  fmi_z  = fmi_Functions.fmiGetEventIndicators(fmi,
    fmi_NumberOfEventIndicators, fmi_AStatesInputs);
  for i in 1:size(fmi_z,1) loop
  fmi_z_positive[i] = fmi_z[i] > 0;
  end for;
algorithm
  when cat(1, change(fmi_z_positive),
          {time>=pre(fmi_TNext), fmi_StepEvent,
        fmi_DiscreteInputChanged, initial()}) then
 (fmi_TNext, fmi_NewStates) :=
   fmi_Functions.fmiEventUpdate(fmi, fmi_AStatesInputs);
    if fmi_NewStates then
     reinit(fmi_x, fmi_Functions.fmiGetContinuousStates(
          fmi, size(fmi_x,1), fmi_AStatesInputs));
    end if;
  end when;
equation
  fmi_StepEvent = fmi_Functions.fmiCompletedStep(fmi,
   fmi_AStatesInputs)>0.5;
```

Since there is no way of guarantee that the calling sequence is as stated, Dymola have introduced some variables that make dependencies. For example, the function `fmi_Functions.fmiSetContinuousStates` cannot be called until the variables `fmi_AParamsAndStart`, `fmi_Initialized` and `fmi_ATime` have been evaluated. This way, the calling sequence can be controlled completely.

OpenModelica's FMI Import is experimental and there are limitations when trying to simulate FMUs. The import interface is shown in figure 9.



Import FMU

FMU File: C:/P418-FMIModelicaImport/trunk/fmu_path/win32/FMUTests_FMUs_CoupledClutches.fmu

Output Directory (Optional):

* If no Output Directory specified then the FMU files are generated in the current working directory.

Log Level: Warning

☐ Debug Logging

☑ Generate input connector pins

☑ Generate output connector pins

* This feature is experimental. Most models are not yet handled by it.

Figure 9: *The figure shows OpenModelica's FMI import.*

When trying to simulate FMUs generated by Dymola in OpenModelica, the result is valid when the simulation time is chosen to `DefaultExperiment.stopTime` defined under `fmiModelDescription`. Any longer simulation will give a constant zero result. The FMU is imported as a Co-Simulation model, which is outside the scope of this thesis.

5 Method

5.1 Design of program and requirements

The program should be able to perform several tasks. Considering Modelica's structured packages, a design such that the *.dll (or *.so) could be found easily is preferable. Since there are no way of loading *.dll-files directly in Modelica, some type of wrapper functions must be generated for every model that maps Modelica functions to C-functions that loads the *.dll and calls the right FMI - function. Another requirement is that the Modelica model, with its wrapper functions and other files, should be packaged in a structural, compact way. The default search path in Modelica structured packages/libraries is defined as a file structure with the package file `package.mo` and directory `Resources` with subdirectories `Include` and `Library`. This is best explained by the example in figure 10 below.

```
ExternalFunctions
  package.mo   // contains the Modelica code from above
  Resources
    Include    // contains the include files
       ExternalFunc1.h   // C-header file
       ExternalFunc2.h   // C-header file
       ExternalFunc3.c   // C-source file
    Library    // contains the object libraries for different platforms
       win32
          ExternalLib1.lib      // static link library for VisualStudio
          ExternalLib2.lib      // statically linking the dynamic link library
          ExternalLib2.dll      // dynamic link library (with manifest)
       linux32
          libExternalLib1.a     // static link library
          libExternalLib2.so    // shared library
```

Figure 10: *The figure shows an example of external functions in a Modelica package.*

The example in figure 10 shows a package `ExternalFunctions` that is mapped to the file system as a structured entity. Since this is the default structure of a structured package in Modelica the program should be able to generate a package with the model defined inside. The program should also move or copy the FMI specific header files `fmiModelFunctions.h` and `fmiModelTypes.h` to the Include directory along with the generated C-file with the wrapper functions.

Also, the program should be able to unzip the FMU file in some way to get access to the *.dll file and the model description schema (XML). Modelon offers a library written in C to interact with all parts of FMUs including unzipping and parsing of XML files. [1] Using this API, the program can unzip the FMU, parse the XML file and receive all data needed to generate a Modelica model.

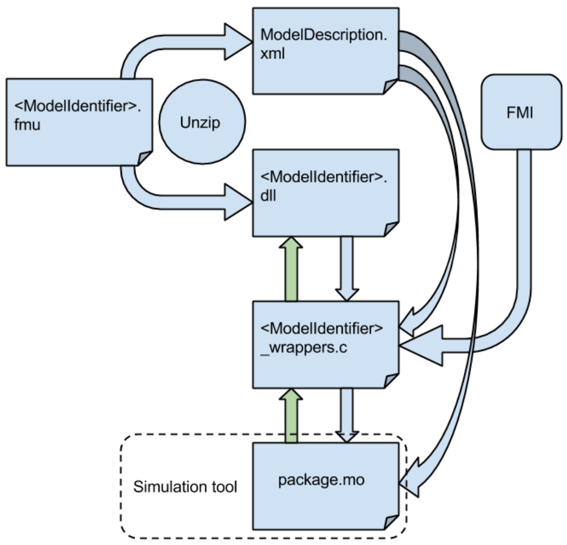


Figure 11: *The figure shows the implementation process.*

Figure 11 illustrates the idea of the implementation process. The FMU instance should be an instance of Modelica's ExternalObject which in C is mapped as `void*`. The FMU should be able to hold information about the FMI functions, a handle to the *.dll file and other type of information. The handle to the *.dll file is represented by the type HANDLE on windows system [5] and by **`void*`** on unix systems [6]. The most natural choice to represent a FMU in C-code is to use a structure. The idea is to have function pointers to the corresponding function in the loaded *.dll file. The naming of these functions are the same as the FMI function, but with underscores between the words, e.g. the function pointer to `fmiSetTime` is named `fmi_Set_Time`. The wrapper functions is named as the FMI function appended with `_wrapper`. At instantiation, an instance of the structure is allocated and the function pointers are set. The structure is defined in appendix B. It holds a handle to the *.dll file, function pointers to all the FMI-functions and a `fmiComponent`.
As mentioned in section 3.1.1, `fmiComponent` is a 32 bit pointer to a model specific structure. This structure holds all information to process model equation etc. It is the exporting tools task to implement this structure and the importing environment does not know its content. Furthermore three flags, `currentTime`, `lastIntegratorStepTime` and `stepEventTrigger`, are present in the FMU structure. These three flags will restrict the calls to `fmiCompletedIntegratorStep`, as explained later.

The generated Modelica model will define a package `fmiFunctions` with all the FMI functions as external function calls. Considering the mapping types discussed in section 3.2.2, all FMI-functions shall be mapped to the Modelica model. In Modelica, the FMU is represented as an `ExternalObject`. The implementation looks like this:

```
class FMU
 extends ExternalObject;
 function constructor
   input String instanceName;
   input Boolean loggingOn;
   input String dll_path;
   input Sring GUID;
   output FMU fmu;
 external "C" fmu =
   fmiInstantiateModel_wrapper(instanceName, loggingOn,
                                         dll_path, GUID)
   annotation (Include="#include <VDP_wrappers.c>");
 end constructor;
 function destructor
   input FMU fmu;
 external "C" fmiFreeModelInstance_wrapper(fmu);
   annotation (Include="#include <VDP_wrappers.c>");
 end destructor;
end FMU;
```

This class contains two functions; the constructor and the destructor. The constructor and destructor are mapped to the FMI functions `fmiInstantiateModel` and `fmiFreeModelInstance` respectively. The function `fmiInstantiateModel` allocates memory for the FMU using the `calloc`-function. If allocation is successful, all the function pointers are set to the correct function in the *.dll file.

One of the requirements of the program is that the FMI-functions shall be mapped 1:1 in the Modelica code. An example of this is shown below

```
function fmiGetContinuousStates
  input FMU fmu;
  input Integer n;
  output Real x[n];
external "C" fmiGetContinuousStates_wrapper(fmu, x, n)
  annotation(Include="#include <BouncingBall_wrappers.c>");
end fmiGetContinuousStates;
```

This function is mapped to the C-function

```
void fmiGetContinuousStates_wrapper(void* fmu,
                                    double* states,
                                    size_t num)
  {
  struct FMU* tmp = fmu;
  fmiStatus status;
  if(tmp != NULL){
    status = tmp->fmi_Get_Continuous_States(tmp->component,
                                            states,
                                            num);
  }
  if(status != fmiOK && status != fmiWarning){
    ModelicaError("Error using fmiGetContinuousStates");
  }
}
```

This function follows the rules of mapping of arguments as mentioned in table 2. The first argument `fmu` is an instance of `ExternalObject` in Modelica and is mapped to `void*` in C. First, this variable has to be casted to the type `struct FMU*`. If the pointer is not a NULL-pointer the actual FMI-function is called and returns with `status`. If `status` is not `fmiOK` and not `fmiWarning`, the simulation shall be stopped.

The calling sequence is mainly based on algorithms because one can control the evaluation order. Though, some operators such as `reinit` are not allowed in algorithm sections, equations are used for these. An example of the calling sequence is shown below

```
initial algorithm
  fmiFunctions.fmiSetReal(fmu, 637534210, vy_start, 1);
  fmiFunctions.fmiSetReal(fmu, 637534212, x_start, 1);
  fmiFunctions.fmiSetReal(fmu, 637534213, y_start, 1);
  fmiFunctions.fmiSetReal(fmu, 16777216, g, 1);
  fmiFunctions.fmiSetTime(fmu, startTime);
  initNextTime := fmiFunctions.fmiInitialize(fmu);
  states := fmiFunctions.fmiGetContinuousStates(fmu,
                                  size(states,1));
algorithm
  fmiFunctions.fmiSetTime(fmu, time);
  states_set := fmiFunctions.fmiSetContinuousStates(fmu,
                                          states, 1);
  der_states := fmiFunctions.fmiGetDerivatives(fmu,
                                  size(states,1));
  vx := fmiFunctions.fmiGetReal(fmu, 637534208);
  der_vx_ := fmiFunctions.fmiGetReal(fmu, 637534209);
  vy := fmiFunctions.fmiGetReal(fmu, 637534210);
  der_vy_ := fmiFunctions.fmiGetReal(fmu, 637534211);
  x := fmiFunctions.fmiGetReal(fmu, 637534212);
  der_x_ := fmiFunctions.fmiGetReal(fmu, 637534208);
  y := fmiFunctions.fmiGetReal(fmu, 637534213);
  der_y_ := fmiFunctions.fmiGetReal(fmu, 637534210);
  F := fmiFunctions.fmiGetReal(fmu, 637534214);
  for i in 1:size(eventIndicators, 1) loop
    domain_change[i] := eventIndicators[i] > 0;
  end for;
  stepEvent := fmiFunctions.fmiCompletedIntegratorStep(
                  fmu, states_and_inputs_set) > 0.5;
equation
  der(states) = der_states;
  when cat(1, change(domain_change),
           {time >= pre(tNext) and pre(tNext) > 0,
              initial(), stepEvent}) then
    (tNext, newStates, terminateSimulation) :=
                  fmiFunctions.fmiEventUpdate(fmu);
  end when;
  when cat(1, change(domain_change),
          {time >= pre(tNext) and pre(tNext) > 0,
              initial(), stepEvent}) then
    if newStates then
        tempStates =
        fmiFunctions.fmiGetContinuousStates(fmu,
                                size(states, 1));
    else
        tempStates = pre(tempStates);
    end if;
  end when;
  when cat(1, change(domain_change) and
           {newStates for i in 1:size(domain_change)},
           {time >= pre(tNext) and pre(tNext) > 0 and
           newStates, initial() and
           newStates, stepEvent and newStates} then
    reinit(states, tempStates);
  end when;
  when terminateSimulation then
        fmiFunctions.fmiTerminate(fmu);
        terminate("Terminated by FMU");
  end when;
```

Initially, start values and parameters are set under the `initial algorithm` section. After that, the start time is set and the model is initialized with `fmiInitialize`. Then the continuous states are retrieved. In the algorithm section, the time is set, the continuous states are set and the derivatives are retrieved. The different variables can be retrieved after this, with the `fmiGetXXX` functions. The event indicators are retrieved and an iteration over this vector takes place that check if the indicator is greater than zero. If so, the value `True` is stored in a temporary vector `domain_change`, if not, the value `False` is stored. Events are handled by `when` sections, see discussion in section 7.

In Modelica, there is no way to know when an integrator step has been completed, hence it it not straightforward to call the function `fmiCompletedIntegratorStep`. This function must be called when an integrator step has been completed according to the FMI standard. The solution to this is to have the three flags `currentTime`, `lastIntegratorStepTime` and `stepEventTrigger` in the FMU structure to check if it is legal to call the function from the wrapper function. The idea is that at initialization the flag `lastIntegratorStepTime` is set to `currentTime` (most likely $t_{start} = 0$) and `stepEventTrigger = 0`. Every time `fmiSetTime` is called, `currentTime` is updated with the actual time. Then, `fmiCompletedIntegratorStep` is called from the Modelica model several times in one integrator step but the actual call to the function `fmi_Completed_Integrator_Step` in the FMU structure is only performed if `currentTime > lastIntegratorStepTime`.
If `fmi_Completed_Integrator_Step` returns with `callEventUpdate = fmiTrue`, the flag `stepEventTrigger` is set to 1 and `lastIntegratorStep` is set to `currentTime`. Then function returns with 1 if a step event has been triggered and `currentTime >= lastIntegratorStepTime`. This is shown below:

```
int fmiCompletedIntegratorStep_wrapper(void* fmu){
  struct FMU* tmp = fmu;
  fmiStatus status;
  fmiBoolean callEventUpdate;
  if(tmp != NULL){
    if(tmp->currentTime > tmp->lastIntegratorStepTime){
      status = tmp->fmi_Completed_Integrator_Step(
                      tmp->component, &callEventUpdate);
      tmp->lastIntegratorStepTime = tmp->currentTime;
      if(callEventUpdate == fmiTrue){
        tmp->stepEventTrigger = 1;
      }
    } else {
      status = fmiOK;
    }
  }
  if(status != fmiOK && status != fmiWarning){
    ModelicaError("Error using fmiCompletedIntegratorStep");
  }
  return tmp->stepEventTrigger &&
         tmp->currentTime >= tmp->lastIntegratorStepTime;
}
```

5.2 Testing Models

By generating some test models, one can compare with e.g. models generated by Dymola to see if there are any numerical differences. The goal with these different models that are tested is to have a good mix of models with different behavior, i.e. different type of events. This means that some models only have time events and other may only have state events, or maybe a mixture of the both. Another aspect to consider is whether the model have inputs and outputs and to see how the model handles this. In this section, the models that are tested are described briefly and the purpose to why this model was chosen.

5.2.1 PureDiscrete

This model is a simple model with only one discrete variable. It was chosen in order to test that the program can handle purely discrete models with no continuous states. Furthermore, this model lacks inputs and outputs and contains only time events.

```
model PureDiscrete
  discrete Real d(start=1);
equation
  when sample(0, 1) then
    d =  sign(sin(time));
  end when;
end PureDiscrete;
```

5.2.2 Bouncing Ball

The Bouncing Ball model was chosen to test if models with state events can be handled. Every time the ball touches the ground a reinitialization must be made such that the ball change direction upwards.

```
model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Boolean impact;
  Real v_new;
 equation
  impact = h <= 0;
  der(v) = if flying then -g else 0;
  der(h) = v;
  when {impact, h <= 0 and v <= 0} then
   v_new = if edge(impact) then -e*pre(v) else 0;
   flying = v_new > 0;
   reinit(v, v_new);
  end when;
end BouncingBall;
```

5.2.3 Pendulum

The pendulum model is special because of its change of state variables. This model tests how the program handles step events.

```
model Pendulum
  Real vx;
  Real vy;
  Real x(start=0.6);
  Real y(start=0.8);
  Real F;
  parameter Real g = 9.82
equation
  der(vx) = -x*F;
  der(vy) = -y*F - g;
  der(x) = vx;
  der(y) = vy;
  x*x+y*y=1;
end Pendulum;
```

5.2.4 Controlled Tanks

This model is used in order to test models using both continuous and discrete states. The model is shown in figure 12.

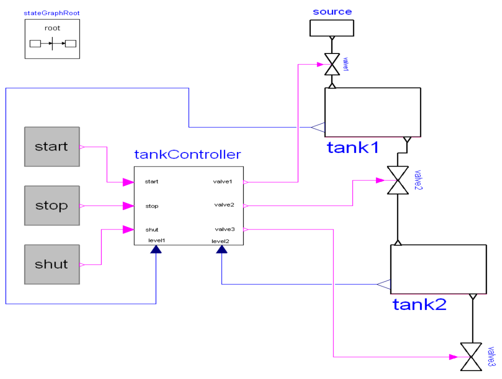


Figure 12: *The figure shows the model* Controlled Tanks.

5.2.5 Direct Feedthrough with Continuous Integrator

This model is included in this test because it consists of three sub-models that are coupled together. The block `Constant` gives a constant signal to the `Integrator` which then goes to the `Feedthrough sum`. The flow is shown in figure 13.

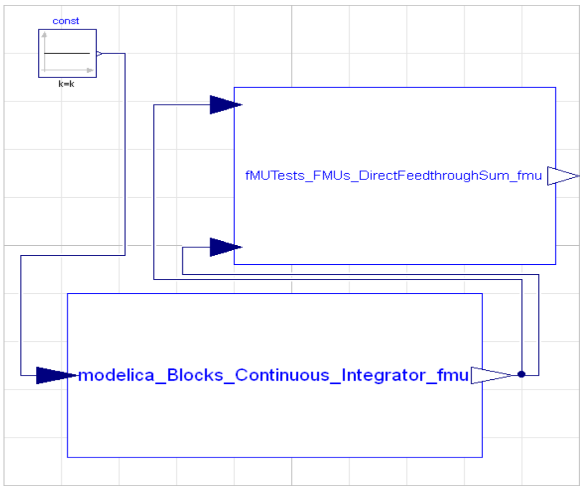

const
k=k
fMUTests_FMUs_DirectFeedthroughSum_fmu
modelica_Blocks_Continuous_Integrator_fmu

Figure 13: *The figure shows the connected blocks.*

5.2.6 Direct Feedthrough with Feedback

To test if the model can handle a feedback signal. This model consists of a sinus signal coupled to one of the inputs of the Feedthrough model and the output is feed backed to the outer input, as in figure 14.

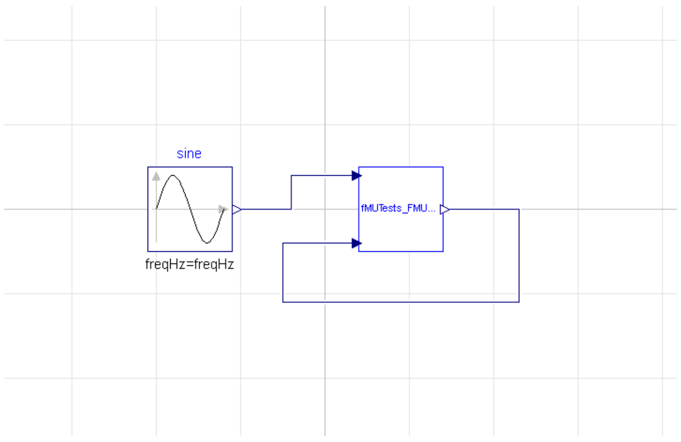


Figure 14: *The figure shows a sinus signal feed backed.*

5.2.7 Coupled Clutches

`Coupled clutches` is a classic example that often is used in tests examples. The model is shown in figure 15. This model contains both time events and state events.

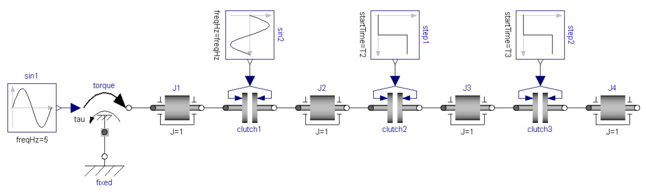


Figure 15: *The figure shows Coupled Clutches.*

5.2.8 3D robot

A bigger model to show that the program can handle industrial sized models. Just like the `Coupled clutches` model, this model contains both time- and state events. Furthermore, this model reaches a stopping criteria when the robot reaches the end of the movement. This will check if the models call the `fmiTerminate` correct.

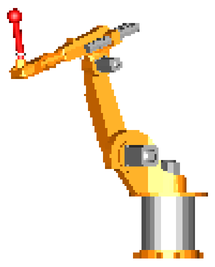


Figure 16: *The figure shows a model of the* 3D Robot

6 Results

6.1 Testing Models

Several models, described in section 5.2, where exported as FMUs from Dymola 2014. These FMUs was then unzipped and parsed with the program to generate the desired file structure with the Modelica model.

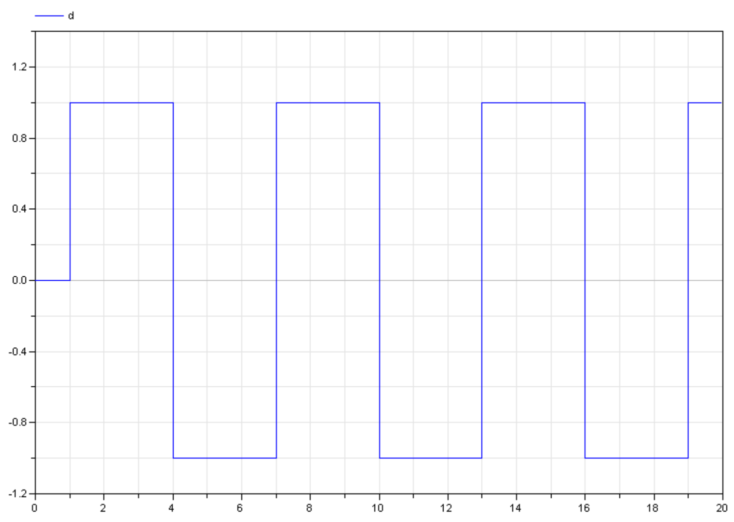


Figure 17: *The figure shows the simulation result of the model* PureDiscrete*.*

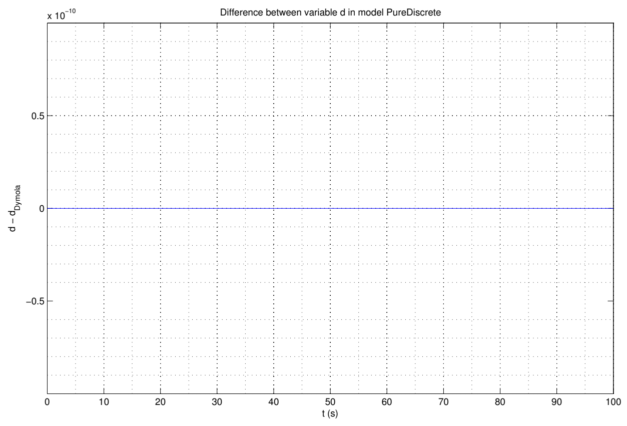


Figure 18: *The difference between variable* d *in model* Pure-Discrete*.*

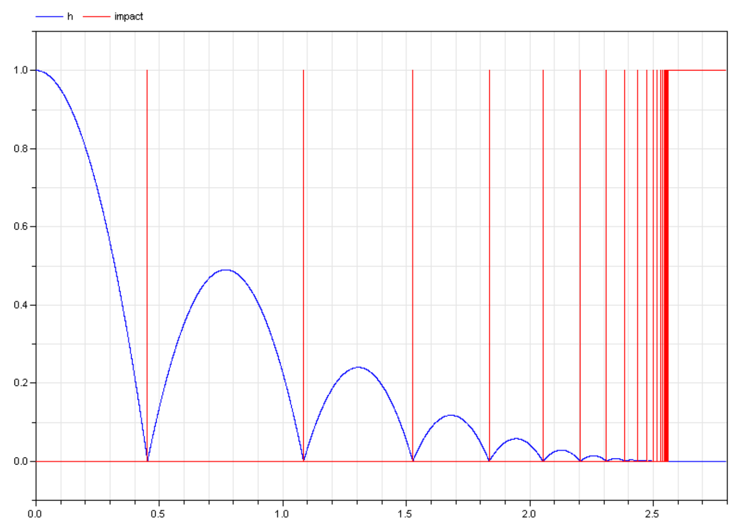


Figure 19: *The figure shows the simulation result of the model* BouncingBall. *Both height and impact variable is plotted.*

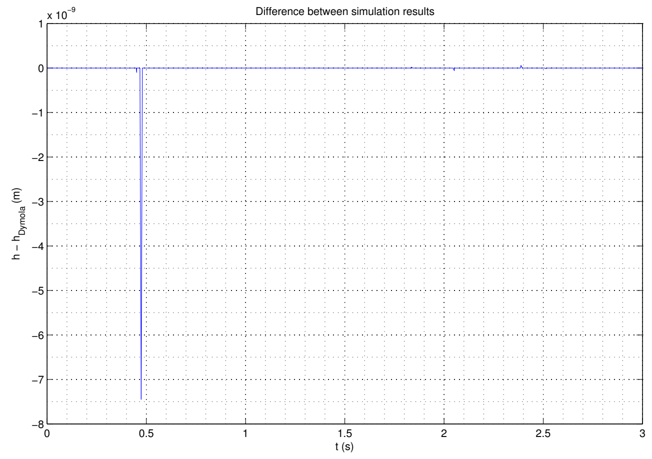


Figure 20: *The figure shows the difference in variable* h *of the BouncingBall model.*

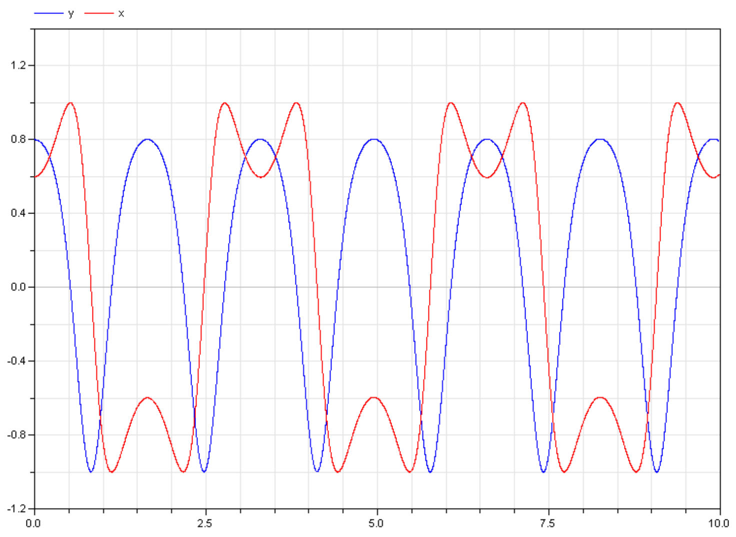


Figure 21: *Both* x *and* y *variable of the pendulum model is plotted.*

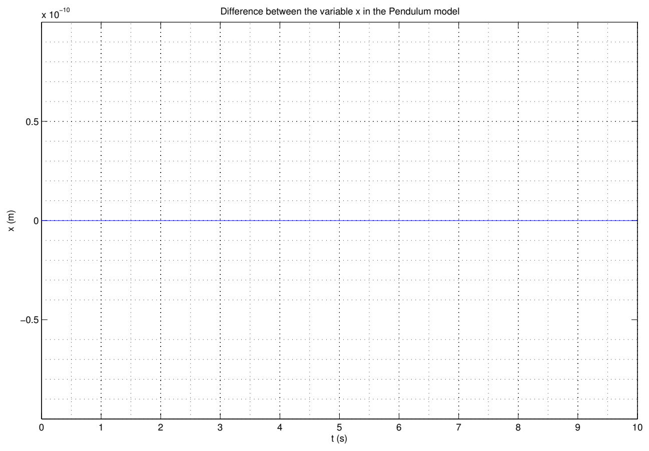


Figure 22: *The figure shows the difference in variable* x *from the* Pendulum *model.*

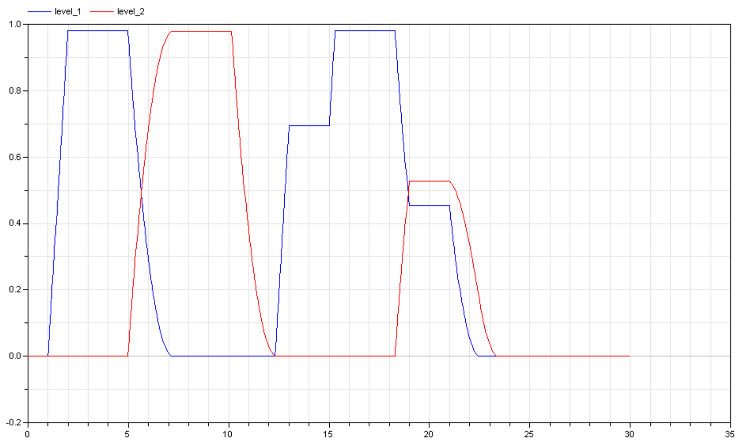


Figure 23: *A plot of the levels in two controlled tanks.*

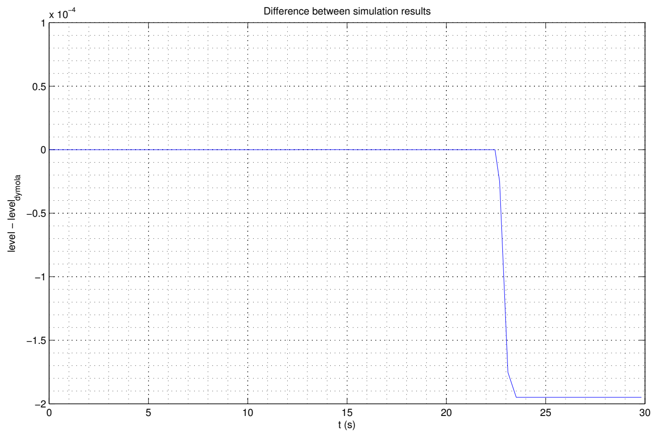


Figure 24: *The difference in level in the model* ControlledTanks.

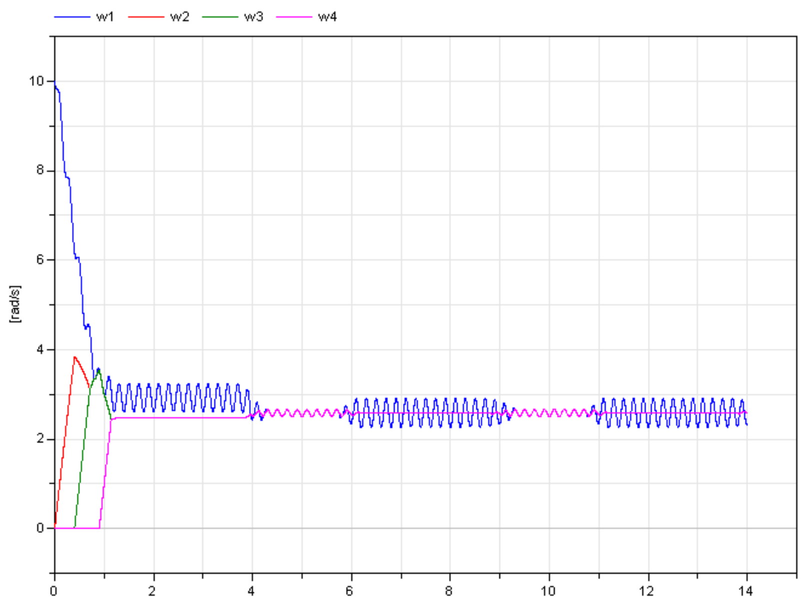


Figure 25: *The simulation result for* Coupled Clutches.

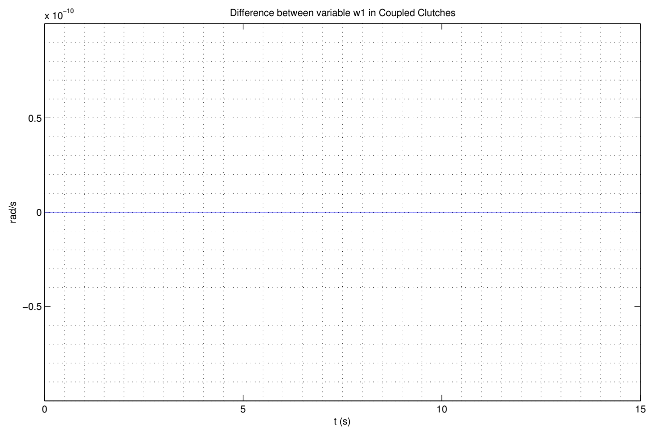


Figure 26: *The figure shows the difference of variable w1 in model* Coupled Clutches.

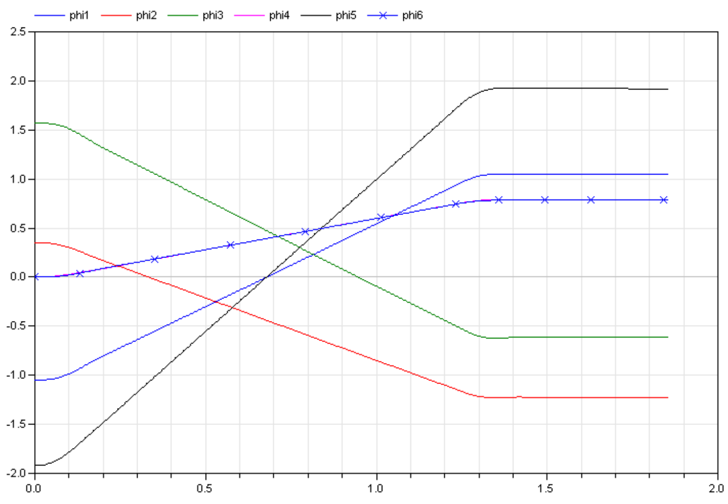


Figure 27: *The figure shows the simulation result for the* 3D Robot *model.*

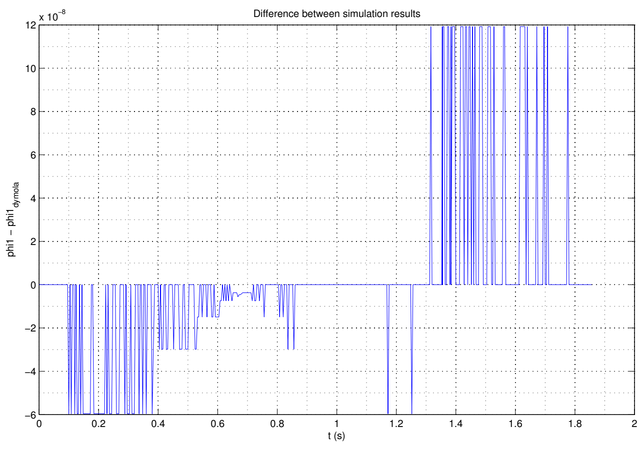


Figure 28: The figure shows the difference between the variable *phi1* in the model *3D Robot*

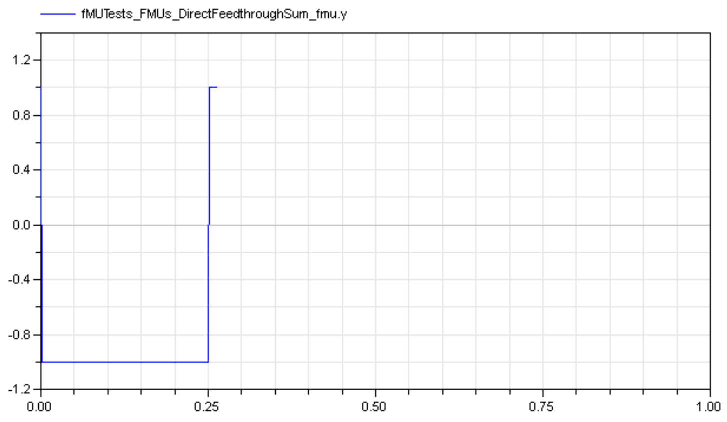


Figure 29: *The figure shows the simulation result for a feed backed signal.*

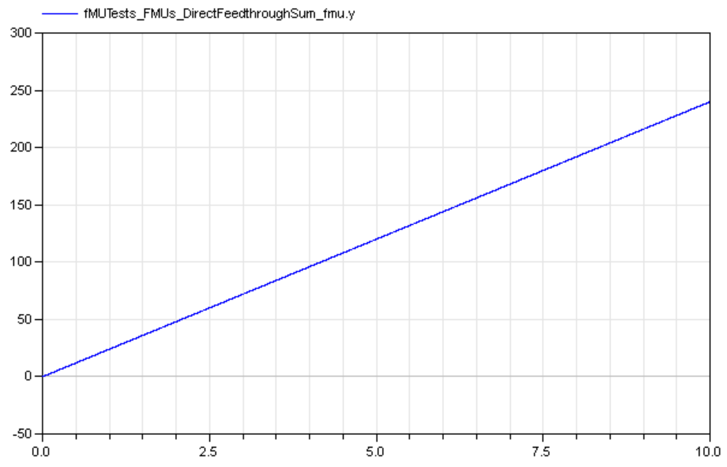


Figure 30: *The figure shows the simulation result of a signal that is integrated and summed.*

The simulation times for the different models are shown in table 5.

Model	Dymola (s)	Mine (s)	Native (s)
Pure Discrete	0.031	0.031	0.031
Bouncing Ball	0.031	0.063	0.031
Pendulum	0.047	0.281	0.234
Controlled Tanks	0.015	0.313	0.015
Coupled Clutches	0.125	1.61	0.016
3D Robot	0.796	32.6	0.375

Table 5: *The CPU times for some of the models.*

7 Discussion

The calling sequence was chosen to be in an algorithm section to be able to control the order of evaluation. However, it is not allowed to have the `reinit` operator in algorithms, so the event handling is stated in equations to follow the Modelica standard. Although it is not allowed to have `reinit` in algorithms, Dymola choose to do so anyway. This leads me to believe that Dymola is less strict when it comes to this, or that Dymola supports Modelica 3.3 in which `reinit` is allowed in algorithms. Comparing the generated models with models generated by Dymola, we see that the results are quite consistent. There are small numerical errors in some of the models and one model, `Feedthrough with feedback`, was completely wrong. The numerical solver could not proceed the simulation longer than 0.27. This model should be quite simple, but something makes it fail and I am not shure why.

It is clear that the generated models requires more CPU time to be simulated than both models generate by Dymola and native models. This can be due to the function calls to `fmiGetXXX` and `fmiSetXXX`. Now, one variable is set/get per function call which probably is not the most efficient way to do. There might be some loop iterations going on inside `fmiGetXXX` and `fmiSetXXX` for lookup of variables etc. which will increase the computation time, especially in larger models. A probably better way to do this is to pass vectors of `valueReferences` and the values. Hence, there will only be one call per base type and the simulation will probably be a lot faster.

The main goal was to generate models that were independent of simulation tool, which means that the models should be able to simulate in e.g. Dymola, OpenModelica and JModelica.org. It turns out, that this is not the case. In OpenModelica, the models are getting error **type in operand to change must be simple type in component <NO COMPONENT>**, and I have not figured out what is causing this error. OpenModelica obvious has some problems with

FMI models, which maybe related to this problem. JModelica.org does not support `fmiXXXString`, which leads to problem when trying to simulate the models. To simulate a model in JModelica one first compile the model to an FMU, which then is loaded and simulated. The generated models can be compiled without problem, but when the compiled FMU shall be loaded, JModelica.org writes the error: Could not load the FMI function 'fmiGetString'. First I thought this was because in `fmiInstantiate` the function tries to set the pointer to `fmiGetString`. But the error remains after those rows of code are commented.

Another goal was to have a platform independent solution which should work on both Windows system and UNIX system. Unfortunately, there has been no time or equipment to test this program on UNIX systems yet.

7.1 Future work

Some things that can be added in this kind of implementation are support for Co-Simulation and for FMI 2.0 in the future when it is released. The function calls should really be changed so that `fmiGetXXX` and `fmiSetXXX` are called with vectors of values instead of one value at a time. I strongly believe that this will improve simulation speed. It would be nice to give this program a GUI to make it more user friendly.

References

[1] *FMI Library*, `http://www.jmodelica.org/FMILibrary`

[2] *CMake documentation*, `http://www.cmake.org/cmake/help/documentation.html`

[3] Modelica Association, *Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling*, Version 3.2 revision 2. July 30, 2013.

[4] ITEA (Information Technology for European Advancement), *Functional Mock-up Interface for Model Exchange*, Version 1.0, January 26, 2010.

[5] `http://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx`

[6] `http://pubs.opengroup.org/onlinepubs/009695399/functions/dlopen.html`

A Model Description Schema

A.1 Example of ModelDescription.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<fmiModelDescription
  fmiVersion="1.0"
  modelName="Lorenz"
  modelIdentifier="Lorenz"
  guid="{c9045968-fbcc-485e-b15f-2b708c32cf21}"
  generationTool="Dymola Version 2014 (32-bit), 2013-03-25"
  generationDateAndTime="2014-01-15T12:54:21Z"
  variableNamingConvention="structured"
  numberOfContinuousStates="3"
  numberOfEventIndicators="0">
  <DefaultExperiment startTime="0.0"
    stopTime="50.0"
    tolerance="1E-010"/>
  <ModelVariables>
    <ScalarVariable
      name="x"
      valueReference="33554432">
      <Real start="10"
        fixed="true"/>
    </ScalarVariable>
    <ScalarVariable
      name="der(x)"
      valueReference="587202560">
      <Real/>
    </ScalarVariable>
    <ScalarVariable
      name="y"
      valueReference="33554433">
      <Real start="1"
        fixed="true"/>
    </ScalarVariable>
    <ScalarVariable
      name="der(y)"
      valueReference="587202561">
      <Real/>
    </ScalarVariable>
    <ScalarVariable
      name="z"
      valueReference="33554434">
      <Real start="5"
        fixed="true"/>
    </ScalarVariable>
    <ScalarVariable
      name="der(z)"
      valueReference="587202562">
      <Real/>
    </ScalarVariable>
    <ScalarVariable
      name="sigma"
      valueReference="16777216"
      variability="parameter">
      <Real start="10"
        fixed="true"/>
    </ScalarVariable>
    <ScalarVariable
      name="rho"
      valueReference="16777217"
      variability="parameter">
      <Real start="28"
        fixed="true"/>
    </ScalarVariable>
    <ScalarVariable
      name="beta"
      valueReference="16777218"
      variability="parameter">
      <Real start="2.6666666666666665"
        fixed="true"/>
    </ScalarVariable>
  </ModelVariables>
  <Implementation>
    <CoSimulation_StandAlone>
      <Capabilities
        canHandleVariableCommunicationStepSize="true"
        canHandleEvents="true"
        canBeInstantiatedOnlyOncePerProcess="true"/>
    </CoSimulation_StandAlone>
  </Implementation>
</fmiModelDescription>
```

B FMU Structure

```
struct FMU {
 DLL_HANDLE dll_handle;

 fmiComponent (*fmi_Instantiate_Model)(fmiString instanceName, fmiString GUID, fmiCallbackFunctions functions, fmiBoolean loggingOn);
 void (*fmi_Free_Model_Instance)(fmiComponent c);

 fmiStatus (*fmi_Set_Debug_Logging)(fmiComponent c, fmiBoolean loggingOn);
 fmiStatus (*fmi_Set_Time)(fmiComponent c, fmiReal time);
 fmiStatus (*fmi_Set_Continuous_States)(fmiComponent c, const fmiReal x, size_t nx);
 fmiStatus (*fmi_Completed_Integrator_Step)(fmiComponent c, fmiBoolean* callEventUpdate);
 fmiStatus (*fmi_Set_Real)(fmiComponent c, const fmiValueReference vr, size_t nvr, const fmiReal    value);
 fmiStatus (*fmi_Set_Integer)(fmiComponent c, const fmiValueReference vr, size_t nvr, const fmiInteger value);
 fmiStatus (*fmi_Set_Boolean)(fmiComponent c, const fmiValueReference vr, size_t nvr, const fmiBoolean value);
 fmiStatus (*fmi_Set_String)(fmiComponent c, const fmiValueReference vr, size_t nvr, const fmiString  value);

 fmiStatus (*fmi_Initialize)(fmiComponent c, fmiBoolean toleranceControlled, fmiReal relativeTolerance, fmiEventInfo* eventInfo);

 fmiStatus (*fmi_Get_Derivatives)(fmiComponent c, fmiReal derivatives, size_t nx);
 fmiStatus (*fmi_Get_Event_Indicators)(fmiComponent c, fmiReal eventIndicators, size_t ni);
 fmiStatus (*fmi_Get_Real)(fmiComponent c, const fmiValueReference vr, size_t nvr, fmiReal    value);
 fmiStatus (*fmi_Get_Integer)(fmiComponent c, const fmiValueReference vr, size_t nvr, fmiInteger value);
 fmiStatus (*fmi_Get_Boolean)(fmiComponent c, const fmiValueReference vr, size_t nvr, fmiBoolean value);
 fmiStatus (*fmi_Get_String) (fmiComponent c, const fmiValueReference vr, size_t nvr, fmiString  value);
 fmiStatus (*fmi_Event_Update)(fmiComponent c, fmiBoolean intermediateResults, fmiEventInfo* eventInfo);
 fmiStatus (*fmi_Get_Continuous_States)(fmiComponent c, fmiReal states, size_t nx);
 fmiStatus (*fmi_Get_Nominal_Continuous_States)(fmiComponent c, fmiReal x_nominal, size_t nx);
 fmiStatus (*fmi_Get_State_Value_References)(fmiComponent c, fmiValueReference vrx, size_t nx);

 fmiStatus (*fmi_Terminate)(fmiComponent c);
 fmiComponent component;
 double currentTime;
 double lastIntegratorStepTime;
 int stepEventTrigger;
};
```

C Model Interface

C.1 fmiModelFunctions.h

```
#ifndef fmiModelFunctions_h
#define fmiModelFunctions_h

/* This header file must be utilized when compiling a model.
   It defines all functions of the Model Execution Interface.
   In order to have unique function names even if several models
   are compiled together (e.g. for embedded systems), every "real" function name
   is constructed by prepending the function name by
   "MODEL_IDENTIFIER"  "_" where "MODEL_IDENTIFIER" is the short name
   of the model used as the name of the zip-file where the model is stored.
   Therefore, the typical usage is:

      #define MODEL_IDENTIFIER MyModel
      #include "fmiModelFunctions.h"

   As a result, a function that is defined as "fmiGetDerivatives" in this header file,
   is actually getting the name "MyModel_fmiGetDerivatives".

   Revisions:
   - Jan. 20, 2010: stateValueReferencesChanged added to struct fmiEventInfo (ticket #27)
                    (by M. Otter, DLR)
                    Added WIN32 pragma to define the struct layout (ticket #34)
                    (by J. Mauss, QTronic)
   - Jan.  4, 2010: Removed argument intermediateResults from fmiInitialize
                    Renamed macro fmiGetModelFunctionsVersion to fmiGetVersion
                    Renamed macro fmiModelFunctionsVersion to fmiVersion
                    Replaced fmiModel by fmiComponent in decl of fmiInstantiateModel
                    (by J. Mauss, QTronic)
   - Dec. 17, 2009: Changed extension "me" to "fmi" (by Martin Otter, DLR).
   - Dez. 14, 2009: Added eventInfo to meInitialize and added
                    meGetNominalContinuousStates (by Martin Otter, DLR)
   - Sept. 9, 2009: Added DllExport (according to Peter Nilsson's suggestion)
                    (by A. Junghanns, QTronic)
   - Sept. 9, 2009: Changes according to FMI-meeting on July 21:
                    meInquireModelTypesVersion     -> meGetModelTypesPlatform
                    meInquireModelFunctionsVersion -> meGetModelFunctionsVersion
                    meSetStates                    -> meSetContinuousStates
                    meGetStates                    -> meGetContinuousStates
                    removal of meInitializeModelClass
                    removal of meGetTime
                    change of arguments of meInstantiateModel
                    change of arguments of meCompletedIntegratorStep
                    (by Martin Otter, DLR):
   - July 19, 2009: Added "me" as prefix to file names (by Martin Otter, DLR).
   - March 2, 2009: Changed function definitions according to the last design
                    meeting with additional improvements (by Martin Otter, DLR).
   - Dec. 3 , 2008: First version by Martin Otter (DLR) and Hans Olsson (Dynasim).


   Copyright © 2008-2009, MODELISAR consortium. All rights reserved.
   This file is licensed by the copyright holders under the BSD License
   (http://www.opensource.org/licenses/bsd-license.html):

   ----------------------------------------------------------------------------
   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   - Redistributions of source code must retain the above copyright notice,
     this list of conditions and the following disclaimer.
   - Redistributions in binary form must reproduce the above copyright notice,
     this list of conditions and the following disclaimer in the documentation
     and/or other materials provided with the distribution.
   - Neither the name of the copyright holders nor the names of its
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
   "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
   TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
   CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
   OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
   WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
   OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
   ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
   ----------------------------------------------------------------------------
   with the extension:

   You may distribute or publicly perform any modification only under the
   terms of this license.
*/

#include "fmiModelTypes.h"
#include <stdlib.h>

/* Export fmi functions on Windows */
#ifdef _MSC_VER
#define DllExport __declspec( dllexport )
#else
#define DllExport
#endif

/* Macros to construct the real function name
   (prepend function name by MODEL_IDENTIFIER  "_") */

#define fmiPaste(a,b)     a ## b
#define fmiPasteB(a,b)    fmiPaste(a,b)
#define fmiFullName(name) fmiPasteB(MODEL_IDENTIFIER, name)

#define fmiGetModelTypesPlatform      fmiFullName(_fmiGetModelTypesPlatform)
#define fmiGetVersion                 fmiFullName(_fmiGetVersion)
#define fmiInstantiateModel           fmiFullName(_fmiInstantiateModel)
#define fmiFreeModelInstance          fmiFullName(_fmiFreeModelInstance)
#define fmiSetDebugLogging            fmiFullName(_fmiSetDebugLogging)
#define fmiSetTime                    fmiFullName(_fmiSetTime)
#define fmiSetContinuousStates        fmiFullName(_fmiSetContinuousStates)
#define fmiCompletedIntegratorStep    fmiFullName(_fmiCompletedIntegratorStep)
#define fmiSetReal                    fmiFullName(_fmiSetReal)
#define fmiSetInteger                 fmiFullName(_fmiSetInteger)
#define fmiSetBoolean                 fmiFullName(_fmiSetBoolean)
#define fmiSetString                  fmiFullName(_fmiSetString)
#define fmiInitialize                 fmiFullName(_fmiInitialize)
#define fmiGetDerivatives             fmiFullName(_fmiGetDerivatives)
#define fmiGetEventIndicators         fmiFullName(_fmiGetEventIndicators)
#define fmiGetReal                    fmiFullName(_fmiGetReal)
#define fmiGetInteger                 fmiFullName(_fmiGetInteger)
#define fmiGetBoolean                 fmiFullName(_fmiGetBoolean)
#define fmiGetString                  fmiFullName(_fmiGetString)
#define fmiEventUpdate                fmiFullName(_fmiEventUpdate)
#define fmiGetContinuousStates        fmiFullName(_fmiGetContinuousStates)
#define fmiGetNominalContinuousStates fmiFullName(_fmiGetNominalContinuousStates)
#define fmiGetStateValueReferences    fmiFullName(_fmiGetStateValueReferences)
#define fmiTerminate                  fmiFullName(_fmiTerminate)


/* Version number */
#define fmiVersion "1.0"

/* Inquire version numbers of header files */
   DllExport const char* fmiGetModelTypesPlatform();
   DllExport const char* fmiGetVersion();

/* make sure all compiler use the same alignment policies for structures */
#ifdef WIN32
#pragma pack(push,8)
#endif

/* Type definitions */
   typedef enum  {fmiOK,
                  fmiWarning,
                  fmiDiscard,
                  fmiError,
                  fmiFatal} fmiStatus;

   typedef void  (*fmiCallbackLogger)        (fmiComponent c, fmiString instanceName,
                                               fmiStatus status,
                                              fmiString category,
                                              fmiString message, ...);
   typedef void* (*fmiCallbackAllocateMemory)(size_t nobj, size_t size);
   typedef void  (*fmiCallbackFreeMemory)    (void* obj);

   typedef struct {
     fmiCallbackLogger         logger;
     fmiCallbackAllocateMemory allocateMemory;
     fmiCallbackFreeMemory     freeMemory;
   } fmiCallbackFunctions;

   typedef struct {
      fmiBoolean iterationConverged;
      fmiBoolean stateValueReferencesChanged;
      fmiBoolean stateValuesChanged;
      fmiBoolean terminateSimulation;
      fmiBoolean upcomingTimeEvent;
      fmiReal    nextEventTime;
   } fmiEventInfo;

/* reset alignment policy to the one set before reading this file */
#ifdef WIN32
#pragma pack(pop)
#endif

/* Creation and destruction of model instances and setting debug status */
   DllExport fmiComponent fmiInstantiateModel (fmiString            instanceName,
                                           fmiString            GUID,
                                           fmiCallbackFunctions functions,
                                           fmiBoolean           loggingOn);
   DllExport void      fmiFreeModelInstance(fmiComponent c);
   DllExport fmiStatus fmiSetDebugLogging  (fmiComponent c, fmiBoolean loggingOn);


/* Providing independent variables and re-initialization of caching */
   DllExport fmiStatus fmiSetTime                (fmiComponent c, fmiReal time);
   DllExport fmiStatus fmiSetContinuousStates    (fmiComponent c,
                 const fmiReal x, size_t nx);
   DllExport fmiStatus fmiCompletedIntegratorStep(fmiComponent c,
                 fmiBoolean* callEventUpdate);
   DllExport fmiStatus fmiSetReal                (fmiComponent c,
                 const fmiValueReference vr, size_t nvr, const fmiReal    value);
   DllExport fmiStatus fmiSetInteger             (fmiComponent c,
                 const fmiValueReference vr, size_t nvr, const fmiInteger value);
   DllExport fmiStatus fmiSetBoolean             (fmiComponent c,
                 const fmiValueReference vr, size_t nvr, const fmiBoolean value);
   DllExport fmiStatus fmiSetString              (fmiComponent c,
                 const fmiValueReference vr, size_t nvr, const fmiString  value);


/* Evaluation of the model equations */
   DllExport fmiStatus fmiInitialize(fmiComponent c, fmiBoolean toleranceControlled,
                              fmiReal relativeTolerance, fmiEventInfo* eventInfo);

   DllExport fmiStatus fmiGetDerivatives    (fmiComponent c, fmiReal derivatives,
                                                       size_t nx);
   DllExport fmiStatus fmiGetEventIndicators(fmiComponent c, fmiReal eventIndicators,
                                                       size_t ni);

   DllExport fmiStatus fmiGetReal   (fmiComponent c, const fmiValueReference vr,
                                 size_t nvr, fmiReal    value);
   DllExport fmiStatus fmiGetInteger(fmiComponent c, const fmiValueReference vr,
                                 size_t nvr, fmiInteger value);
   DllExport fmiStatus fmiGetBoolean(fmiComponent c, const fmiValueReference vr,
                                 size_t nvr, fmiBoolean value);
   DllExport fmiStatus fmiGetString (fmiComponent c, const fmiValueReference vr,
                                 size_t nvr, fmiString  value);

   DllExport fmiStatus fmiEventUpdate               (fmiComponent c,
           fmiBoolean intermediateResults, fmiEventInfo* eventInfo);
   DllExport fmiStatus fmiGetContinuousStates       (fmiComponent c,
           fmiReal states, size_t nx);
   DllExport fmiStatus fmiGetNominalContinuousStates(fmiComponent c,
           fmiReal x_nominal, size_t nx);
   DllExport fmiStatus fmiGetStateValueReferences   (fmiComponent c,
           fmiValueReference vrx, size_t nx);
   DllExport fmiStatus fmiTerminate                 (fmiComponent c);

#endif
```

C.2 fmiModelTypes.h

```
#ifndef fmiModelTypes_h
#define fmiModelTypes_h

/* Standard header file to define the argument types of the
   functions of the Model Execution Interface.
   This header file must be utilized both by the model and
   by the simulation engine.

   Revisions:
   - Jan.  4, 2010: Renamed meModelTypes_h to fmiModelTypes_h (by Mauss, QTronic)
   - Dec. 21, 2009: Changed "me" to "fmi" and "meModel" to "fmiComponent"
                    according to meeting on Dec. 18 (by Martin Otter, DLR)
   - Dec.  6, 2009: Added meUndefinedValueReference (by Martin Otter, DLR)
   - Sept. 9, 2009: Changes according to FMI-meeting on July 21:
                    Changed "version" to "platform", "standard" to "standard32",
                    Added a precise definition of "standard32" as comment
                    (by Martin Otter, DLR)
   - July 19, 2009: Added "me" as prefix to file names, added meTrue/meFalse,
                    and changed meValueReferenced from int to unsigned int
                    (by Martin Otter, DLR).
   - March 2, 2009: Moved enums and function pointer definitions to
                    ModelFunctions.h (by Martin Otter, DLR).
   - Dec. 3, 2008 : First version by Martin Otter (DLR) and
                    Hans Olsson (Dynasim).


   Copyright © 2008-2010, MODELISAR consortium. All rights reserved.
   This file is licensed by the copyright holders under the BSD License
   (http://www.opensource.org/licenses/bsd-license.html)

   ----------------------------------------------------------------------------
   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   - Redistributions of source code must retain the above copyright notice,
     this list of conditions and the following disclaimer.
   - Redistributions in binary form must reproduce the above copyright notice,
     this list of conditions and the following disclaimer in the documentation
     and/or other materials provided with the distribution.
   - Neither the name of the copyright holders nor the names of its
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
   "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
   TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
   CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
   OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
   WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
   OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
   ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
   ----------------------------------------------------------------------------

   with the extension:

   You may distribute or publicly perform any modification only under the
   terms of this license.
*/

/* Platform (combination of machine, compiler, operating system) */
#define fmiModelTypesPlatform "standard32"

/* Type definitions of variables passed as arguments
   Version "standard32" means:

   fmiComponent     : 32 bit pointer
   fmiValueReference: 32 bit
   fmiReal          : 64 bit
   fmiInteger       : 32 bit
   fmiBoolean       :  8 bit
   fmiString        : 32 bit pointer

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

```
/* Values for fmiBoolean  */
#define fmiTrue  1
#define fmiFalse 0

/* Undefined value for fmiValueReference (largest unsigned int value) */
#define fmiUndefinedValueReference (fmiValueReference)(-1)

#endif
```

Master's Theses in Mathematical Sciences 2014:E11
ISSN 1404-6342

LUTFNA-3026-2014

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/