D3.1.2 - Standardized interfaces for continuous-time models as needed for state and parameter estimation in the MODRIO tool chains

WP 3.1 Continuous Systems
WP 3 State estimation and system monitoring

MODRIO (11004)

Version 2.0

Date 30/09/2014

Authors

Andreas Pfeiffer, Jonathan Brembeck, Martin Otter	DLR
Karl Wernersson, Hilding Elmqvist	Dassault Systemes
Toivo Henningsson	Modelon

Executive Summary

This document describes a process and the necessary interfaces for state estimation in Modelica. For smooth model equations in Modelica, the approach is based on FMI for Co-Simulation models using an extended interface with respect to FMI 2.0. This extended FMI 2.0 interface is needed for state estimation and other methods that require simulations of the model at sample periods. The corresponding FMU is imported into a Modelica environment to be applied by a Kalman filter model in Modelica. The interface model for the Kalman filter is automatically and individually generated for each filter type and FMU. With a prototype of Dymola the described process is successfully demonstrated by examples. This approach is also described in the publication [3][1].

[1] Some figures and paragraphs from this publication are reused in this report.

Contents

1. Problem Class of Continuous-Time Models

Different filter algorithms for estimation purposes are described in D3.1.1 [1]. According to these algorithms the following dynamical system is considered for state estimation:

$$\begin{aligned} \dot{x} &= f(x,u,t), \quad x(t_0) = x_0, \\ y &= h(x,u,t) \end{aligned} \tag{1.1}$$

with continuous-time states x, continuous-time input signals u and time t. At initial time t_0 initial values x_0 are given. The derivatives of the states are modeled by a sufficiently smooth function f. The output variables y are computed by a smooth function h.

In Table 1 all needed evaluations of system (1.1) for state estimation applications are summarized. A tool chain has to provide these model evaluations. In the right column the name of the proposed Modelica function is listed to trigger the corresponding evaluation in the tool chain.

Table 1: Model evaluations for nonlinear state estimation (in the right column the name of the proposed Modelica functions are defined to trigger the evaluations in the tool chain).

Required model evaluations		**Modelica**
Integration between two sample points t_{k-1} and t_k:	$f_{k\|k-1} := x_{k-1} + \int_{t_{k-1}}^{t_k} f(x, u_{k-1}, t)\, dt$	`integrator`
Derivative evaluation:	$\dot{x} = f(x,u,t)$	`f`
Output evaluation:	$y = h(x,u,t)$	`h`
Optional model evaluations (if not provided, computed numerically by difference quotients)		
State Jacobian matrix:	$\frac{\partial f}{\partial x}(x,u,t)$	`fx`
Output Jacobian matrix:	$\frac{\partial h}{\partial x}(x,u,t)$	`hx`

2. Modelica Interfaces for Estimation Purposes

The state estimator algorithms are implemented with Modelica functions that provide the needed model evaluations. The interfaces of these functions are defined in the following (and in package `SystemFunctions` where the function prototypes are collected).

For example, the partial functions `fBase, hBase` are defined as:

```
partial function fBase "Base class of the state equation dx/dt = f(x,u,t)"
  input Integer nx "Number of states";
  input Integer nu "Number of inputs";
  input Real x[nx] "States";
  input Real u[nu] "Inputs";
  input Modelica.SIunits.Time t "Time";
  output Real dxdt[nx] "Derivatives";
end fBase;
```

```
partial function hBase "Base class of output function"
  input Integer nx "Number of states";
  input Integer nu "Number of inputs";
  input Integer ny "Number of outputs";
  input Real x[nx] "States";
  input Real u[nu] "Inputs";
  input Modelica.SIunits.Time t "Time";
  output Real y[ny] "Outputs";
end hBase;
```

Some of the estimation algorithms described in D3.1.1 make use of the Jacobian matrices f_x and h_x. Further, a one step numerical integration algorithm is needed. The following partial Modelica functions provide the necessary interfaces for the functions f_x, h_x and the integration algorithm:

```
partial function fxBase "Base class of the Jacobian matrix of the state
                                                function dx/dt = f(x,u,t)"
  input Integer nx "Number of states";
  input Integer nu "Number of inputs";
  input Real x[nx] "States";
  input Real u[nu] "Inputs";
  input Modelica.SIunits.Time t "Time";
  output Real dfdx[nx, nx] "Jacobian matrix of the continuous state
                                                    function fx = df/dx";
end fxBase;

partial function hxBase "Base class of the Jacobian of the output
                                                          function h"
  input Integer nx "Number of states";
  input Integer nu "Number of inputs";
  input Integer ny "Number of outputs";
  input Real x[nx] "States";
  input Real u[nu] "Inputs";
  input Modelica.SIunits.Time t "Time";
  output Real dhdx[ny, nx] "Jacobian matrix of the output function
                                                          hx = dh/dx";
end hxBase;

partial function integratorBase "Base class of a one step ODE solver"
  extends Modelica.Icons.Function;
  input Integer nx "Number of states";
  input Integer nu "Number of inputs";
  input Real x[nx] "States at time t, x(t)";
  input Real u[nu] "Inputs at time t, u(t)";
  input Modelica.SIunits.Time t "Time";
  input Modelica.SIunits.Time dt "Stepsize in time (> 0)";
  output Real xNew[nx] "Numerical solution of x(t+dt)";
end integratorBase;
```

The dimensions `nx`, `nu` are conceptually not necessary, because the dimensions could be determined by the sizes of the vectors `x` and `u`. Currently, Dymola does not support arrays with non-fixed sizes in function calls of translated Modelica models. The function prototypes are collected in package `SystemFunctions`. They are replaceable functions that provide the needed functionality of Table 1 (the right column of this table lists the name of the function):

```
partial package SystemFunctions
  constant Integer nx=1;
  constant Integer ny=1;
  constant Integer nu=1;
```

```
  replaceable function f
    extends fBase;
  end f;

  replaceable function h
    extends hBase;
  end h;

  replaceable function fx
    extends fxBase;
  end fx;

  replaceable function hx
    extends hxBase;
  end hx;

  replaceable function integrator
    extends integratorBase;
  end integrator;
end SystemFunctions;
```

For a concrete system, the partial package `SystemFunctions` has to be extended and the system functions have to be redeclared:

```
package SystemExample
  extends SystemFunctions(nx=4, nu=1, ny=2);

  redeclare function extends f
    // Code of f
  end f;

  redeclare function extends h
    // Code of h
  end h;

  ...

  redeclare function extends integrator
    // Code of integrator
  end integrator;
end SystemExample;
```

Then, this package can be used by a Kalman filter algorithm `FilterAlgorithm` in a filter model as follows:

```
model FilterModel
  // ...
  parameter Real sampleTime = 0.01;
  package system = SystemExample;
equation
  // ...
  when sample(0, sampleTime) then
    FilterAlgorithm(..., function system.f(), function system.h(), ...);
  end when;
end FilterModel;
```

In the filter algorithm itself the functions `system.f`, `system.h`, etc. can be called with different arguments as required by some filters. Therefore not only evaluated values of the system functions but the functions themselves are transferred to the filter algorithm. Of course, the Jacobian matrices can be also transferred.

A user has to provide a Modelica package like `SystemExample` with Modelica code for the system functions. For smaller systems this can be implemented manually, but for realistic systems there are only Modelica plant models usually part of the modelling process. The code for the system functions needed for state estimation is generally not available in Modelica. But in FMI models the interfaces are rather similar to the needs of state estimation. Consequently, the proposed process to enable state estimation with general Modelica plant models is based on the FMI interface Version 2.0 [2].

3. Interfacing FMI Models for Estimation Purposes with Modelica

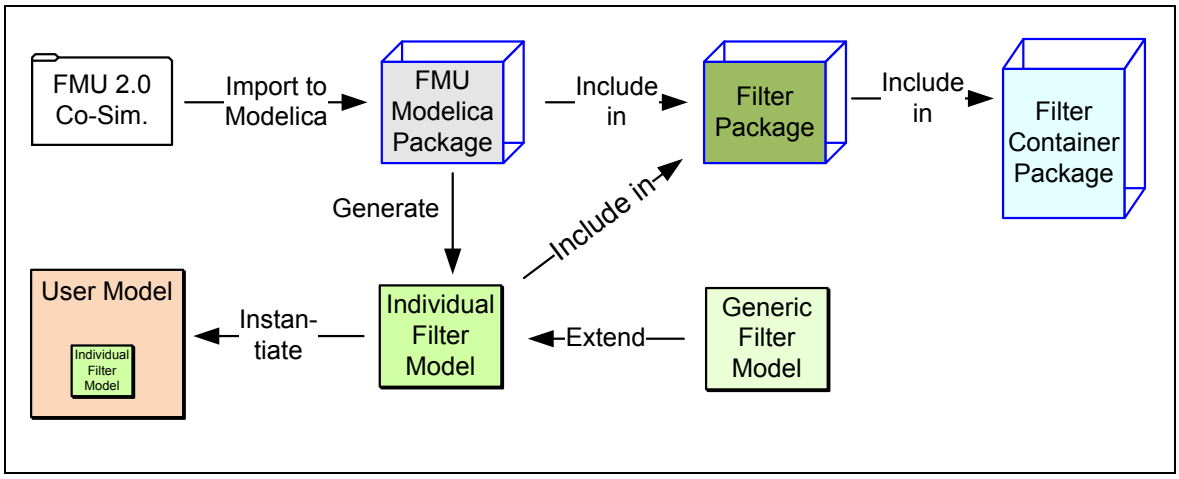
The goal of this section is to provide a tool chain for nonlinear state estimation based on the model equations of a Modelica model [3]. FMI 2.0 for Co-Simulation [2] simplifies the implementation significantly because the integration algorithm, including event handling, is embedded inside the Functional Mock-Up Interface Unit (FMU). So, the integrator function required for state estimation is already contained in the FMU. Still some features are missing in FMI 2.0. In a Dymola prototype these have been added in order that the required functions from Table 1 are supported. For each individual FMU a tailored Kalman Filter Modelica model is generated to provide the original variable names for inputs, outputs and measured variables.

3.1. Process

In practice, the user normally does not provide the system equations programmed by hand in different functions but indirectly by a hierarchical Modelica model. In order to use a Modelica model that reflects the system of equations (1.1) the Modelica model is exported as FMI 2.0 for Co-Simulation model (FMU). Consequently, we have to generate interface wrapper functions in Modelica that get access to the FMI functions of the FMU and that interact as defined in Section 2.

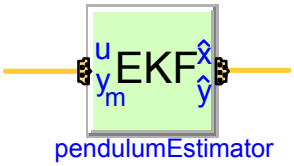
Because only smooth continuous model functions are considered for filter algorithms, the FMU has to have smooth functions, especially events may lead to unusual behavior of the state estimation filter.

The overall process of using a state estimator in Modelica is illustrated in the following figure:



An FMU (usually exported from a Modelica model) is imported into the Modelica environment by extending the package `FMUImportTemplate`. The imported package is then included in a filter package. In this filter package also *Individual Kalman Filter* models are included that are generated for the FMU Modelica package. The individual filter models provide variable names on buses and user convenient parameter menus. The algorithmic part of the state estimation is provided in a *Generic Filter Model*. The filter package may be included in a filter container package to collect several filter packages for easy access.

Finally, the individual filter model can be instantiated in the user's application model. An instance of an individual Extended Kalman Filter (EKF) model generated by the process for a pendulum model is shown in the following figure:



The bus on the left side contains the individual input and measurement variables of the FMU and the bus on the right side contains the individual estimated state and output variables of the FMU.

In the following sub-sections, the details of the process are described.

3.2. FMI for State Estimation

The standard FMI 2.0 for Co-Simulation Interface allows integrating Eq. (1.1) from sample instant t_{k-1} to t_k with function `fmi2DoStep(..)` and therefore computing $f_{k|k-1}$ in *Table 1*. In standard co-simulation the continuous-time states of a model are hidden in the co-simulation slave. However, for state estimation the states need to be explicit and it must be possible to reset the states at sample instants. In order to achieve this, a development version of Dymola 2015 FD01, that has already support for FMI 2.0 Co-Simulation according to [2], has been extended in a prototype with the needed features. Especially,

- the continuous-time states are reported in the `modelDescription.xml` file under element `ModelStructure`,
- it is possible to explicitly set the continuous-time states with `fmi2SetReal(..)` before `fmi2DoStep(..)` is called,
- it is possible to inquire the actual values of all variables with `fmi2GetReal(..)` after `fmi2SetReal(..)` was called, without an `fmi2DoStep(..)` in between,
- when importing an FMU for Co-Simulation in to Modelica, Dymola generates the Modelica code optionally according to the `FMUImportTemplate` package shown in the next sub-section. This package serves as interface to access the needed FMI functionality from a Modelica model or function.

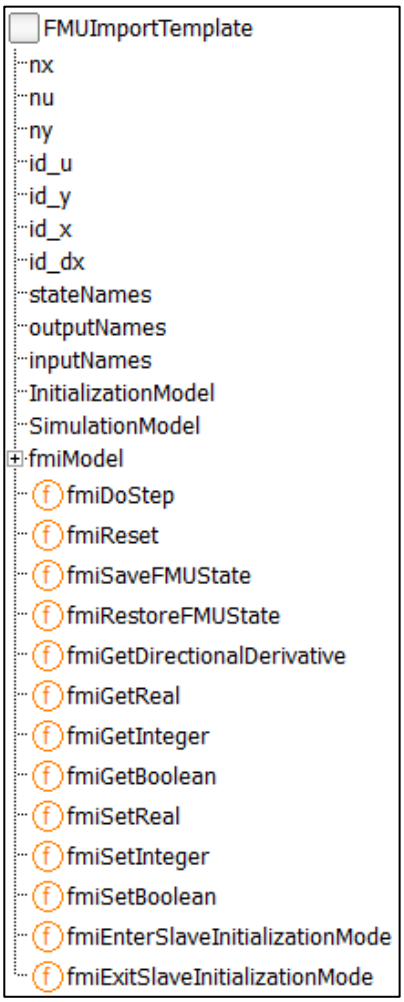
3.3. FMUImportTemplate Package

Importing an FMU means to generate a package that contains all the functionality needed to simulate the FMU or use it in a state estimator. For this the template package `FMUImportTemplate` is provided [7], see code and figure below. The imported FMU extends from the `FMUImportTemplate` and redeclares all elements.

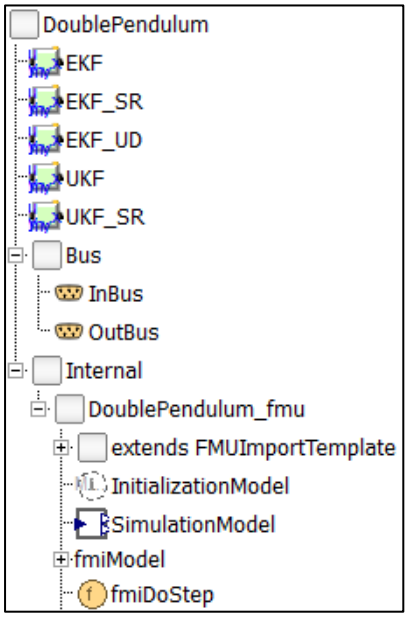
```
partial package FMUImportTemplate
  constant Integer nx=1;
  …
  constant Integer id_x[nx];
  …
  constant String stateNames[nx];
  …
  replaceable model SimulationModel
  end SimulationModel;

  replaceable model InitializationModel
    fmiModel fmi;
    parameter Real fmiInitOk(fixed=false);
  end InitializationModel;

  replaceable partial class fmiModel
    extends ExternalObject;
    function constructor
      …
```

```
    end constructor;
    …
  end fmiModel;

  replaceable function fmiDoStep
      input fmiModel fmi;
      …
  end fmiDoStep;
  …
end FMUImportTemplate;
```

Important dimensions of the FMU such as the number of continuous states `nx`, inputs `nu` and outputs `ny` are set in the imported FMU package. Furthermore, the FMI variable references are available by the vectors `id_x`, `id_dx`, `id_u` and `id_y` for state, state derivative, input and output variables. It is also important to get variable names for states, inputs and outputs. Otherwise the order of the components in the vectors `x`, `u`, `y` would be only visible by user-unfriendly reference values instead of variable names. The names are used in the parameter GUIs of the filter model in the next subsection and in the input and output bus of a filter model.



The imported FMU package contains two models: `SimulationModel` and `InitializationModel`. The model `SimulationModel` is a fully operating Modelica model (with inputs and outputs) that wraps the FMU for Co-Simulation whereas in `InitializationModel` only the FMU is instantiated by the external object `fmiModel` and the FMI initialization phase is executed. The `InitializationModel` is used in a Kalman Filter model; the `Simulation-Model` is contained for completeness to use the imported FMU package also for other applications like a "real" FMU for Co-Simulation in the Modelica simulation environment.



The FMU package provides interface functions to all (or at least most) of the functions defined in the FMI Co-Simulation standard 2.0. The FMU is imported into the `Internal` sub-package of the individual filter package, see figure on the left.

3.4. System Functions for State Estimators

For a particular model, an implementation of the functions in `SystemFunctions` has to be provided. For FMUs, this is performed with the generic package `FMISystemFunctions`. The implementation is based on the `FMUImportTemplate` package and holds therefore for every FMU that extends from this template package.

```
package FMISystemFunctions
  extends SystemFunctions;
  replaceable package FMU
           constrainedby FMUImportTemplate;
  redeclare function extends f
    input FMU.fmiModel fmi;
  algorithm
    FMU.fmiSetReal(fmi, FMU.id_u, u);
    FMU.fmiSetReal(fmi, FMU.id_x, x);
    dxdt := FMU.fmiGetReal(fmi, FMU.id_dx);
  end f;

  redeclare function extends h
    input FMU.fmiModel fmi;
  algorithm
    FMU.fmiSetReal(fmi, FMU.id_u, u);
    FMU.fmiSetReal(fmi, FMU.id_x, x);
    y := FMU.fmiGetReal(fmi, FMU.id_y);
  end h;

  redeclare function extends integrator
    input FMU.fmiModel fmi;
  algorithm
    FMU.fmiSaveFMUState(fmi);
    FMU.fmiSetReal(fmi, FMU.id_u, u);
    FMU.fmiSetReal(fmi, FMU.id_x, x);
    FMU.fmiDoStep(fmi, t, dt, 0);
    xNew := FMU.fmiGetReal(fmi, FMU.id_x);
    FMU.fmiRestoreFMUState(fmi);
  end integrator;
end FMISystemFunctions;
```

The functions `fmiSave/RestoreFMUState` in the above code fragments are auxiliary functions that call the FMI functions `fmi2Get/Set/FreeFMUstate` to enable several calls of `fmi2DoStep` starting at the same time instant, as needed, for example, for the Unscented Kalman Filter.

The system functions `f`, `h`, `integrator` can be directly implemented with functions provided in `FMUImportTemplate`. The Jacobians `fx` and `hx` are implemented by computing them numerically with finite difference quotients. Once Dymola fully supports directional derivatives for imported FMUs for the extended Co-Simulation case, that is function `fmi2GetDirectionalDerivative`, then this function can be directly called and will provide a more efficient and reliable evaluation of the Jacobians. In the following the main implementation of calling numerical Jacobians is listed:

```
  redeclare function extends fx
    input FMU.fmiModel fmi;
  protected
    function fJac
      extends  JacobianAlgorithms.fJacBase;
      input FMU.fmiModel fmi;
      input Integer nu;
      input Real u[nu];
      input Real t;
    algorithm
      F := f(nx, nu, x, u, t, fmi=fmi);
    end fJac;
  algorithm
    dfdx := JacobianAlgorithms.ForwardDifference(nx, nx, x,
```

```
                                    function fJac(nu=nu, u=u, t=t, fmi=fmi));
  end fx;

  redeclare function extends hx
    input FMU.fmiModel fmi;
  algorithm
    dhdx := JacobianAlgorithms.ForwardDifference(nx, ny, x,
                                        function h(nu=nu, u=u, t=t, fmi=fmi));
  end hx;
```

The Dymola prototype supports two techniques for the FMI function `fmi2DoStep`. Either the *Sundials* solvers [4] are used (that are integrators with variable step size and error control) to numerically integrate the model equations, or *Inline integration* [5] is applied, that means fixed step solvers are embedded in the model equations. For real-time applications, fixed-step methods have to be used and therefore a Kalman filter will usually utilize Inline integration.

3.5. Tailored Kalman Filter Models in Modelica

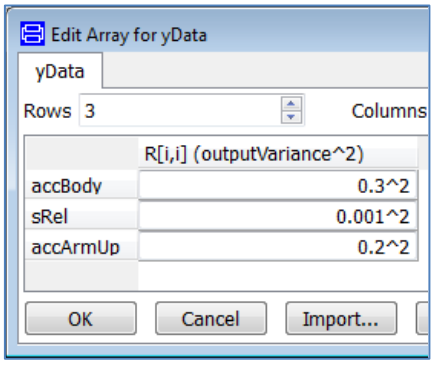
Based on the imported FMU package an individual Kalman Filter model has to be generated. In the current version of DLR's Kalman Filter Library [3] this can be performed automatically by use of a Modelica/Dymola scripting function. The idea is to define an input bus `InBus` and an output bus `OutBus` for exchanging variables between the filter model and higher level models. The names of the bus variables correspond to the variable names of the imported FMU – only ".", ",", "[", "]" and " " are replaced by "_" due to the requirement of the Modelica syntax. The bus definitions for the use-case example in Section 5.2 are listed below:

```
encapsulated expandable connector InBus
  import Modelica;
  extends Modelica.Icons.SignalBus;
  // Model Inputs
  Real u;
  // Measured Model Outputs
  Real s;
  Real phi;
end InBus;
```

```
encapsulated expandable connector OutBus
  import Modelica;
  extends Modelica.Icons.SignalBus;
  // Estimated Model States
  Real prismatic_s;
  Real prismatic_v;
  Real rev_phi;
  Real rev_w;
  Real revolute2_phi;
  Real revolute2_w;
  // Estimated Model Outputs
  Real s;
  Real phi;
end OutBus;
```

The advantage of this approach is that not vectors of anonymous variables are defined, but bus variables with meaningful names tailored to each individual FMU. The main state estimation algorithms are implemented in sub-functions and in a partial filter model, e.g. for an Unscented Kalman Filter. This model defines several variables and parameters for the filter algorithm that is called at each sample point of a sampled integration time interval. In the filter model also an instance of `InitializationModel` of the imported FMU package is included. Together with the package `FMISystemFunctions` all necessary parts are put together to run FMI based Kalman Filter algorithms within a Modelica model.

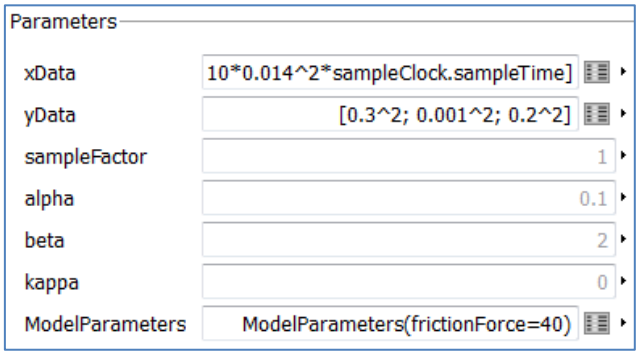
A further improvement of the user interface compared to former ones are the filter parameters like state and output variances that are shown in lists with names of the respective variables – instead of indices of vectors, see the following figure for the output variances:



Basically, a matrix is defined and via Dymola specific annotations row and column headings can be added to the parameter menu. For example the above menu is defined in the following way:

```
parameter Real yData[FMUPackage.ny,1]
  annotation(Dialog(
    __Dymola_columnHeadings =
      {"R[i,i] (outputVariance^2)"},
    __Dymola_rowHeadings =
      {"accBody", "sRel", "accArmUp"}));
```

In the parameter menu of the filter (see next figure) the user can press the button on the right side of `yData` to get the menu above. Also the model parameters of the FMU may be modified by clicking on the button on the right side of `ModelParameters`.



4. Exporting and importing models for estimation in XML format

When exchanging models for estimation purposes, a symbolic format can be used as an alternative to the FMU format. An XML-based symbolic format for model exchange is being developed jointly for deliverables D3.1.2 (estimation purposes) and D5.1.3 (optimization purposes). The principal advantage of a symbolic format is that it allows the recipient to carry out a wide range of transformations on the model. E.g., if the exporting tool did not provide Jacobians, the importing tool can derive them. Specific estimation algorithms may require other transformations.

The aim is to create a single unified format that allows representing Modelica models at different stages of processing, from source code to transformed and flattened models; the latter being the most natural for import into most estimation algorithms. The format should provide full coverage of the Modelica 3.x language, be uniform and minimalistic, and the resulting XML should not be overly verbose.

Draft XML schemas that cover a majority of Modelica constructs have been constructed and

presented at the 79th Modelica Design Meeting in Coventry, June 3-5, 2013. The schemas are modeled quite closely on the syntax of Modelica, but an effort has been made to have as few ways as possible to syntactically express the same semantic construction, to ease the implementation burden on the receiving end. The schemas are made more uniform and at the same time leave some room for future extensions by modeling a superset of Modelica syntax.

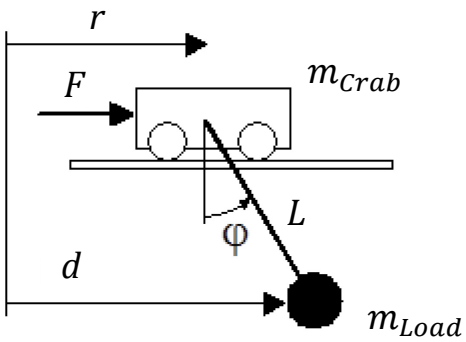
A simple model might look like

```
<class kind="model">
     <component name="x" variability="continuous" visibility="public">
          <builtin name="Real"/>
     </component>
     <equation initial="true">
          <equal>
               <local name="x"/>
               <real value="1"/>
          </equal>
     </equation>
     <equation>
          <equal>
               <operator name="der">
                    <local name="x"/>
               </operator>
               <call builtin="-">
                    <local name="x"/>
               </call>
          </equal>
     </equation>
</class>
```

5. Examples

5.1. Analytical Crane

The following crane example is considered to show how a Kalman Filter model can be applied to hand-coded system functions:



A crab with mass m_{Crab} moves on a horizontal line. On the crab a pendulum of length L with a mass point m_{Load} is attached. An external force F acts on the crab. Output variables of the system are the angle φ and the horizontal position d of the mass point whereas the force F is an input to the system.

An approximated model of the crane can be derived by applying impulse and angular momentum conservation. Four states (angle φ, angular velocity $\dot{\varphi}$, position r, velocity $\dot{r}$) describe the dynamical system by the following equations implemented in Modelica functions to be used by filter algorithms. The Jacobian matrices of f and h are found by analytical differentiation.

```
package SystemCrane
```

```
function f
  extends fBase;
  input Real p[3];
protected
  Real m_Load = p[1];
  Real m_Crab = p[2];
  Real L = p[3];
algorithm
  dxdt := {x[2],
           -(L*m_Load*cos(x[1])*sin(x[1])*x[2]^2 + 981*sin(x[1])
           *(m_Load+m_Crab)/100 +u[1]*cos(x[1]))/(L*(m_Crab+m_Load*
           sin(x[1]))),
           x[4],
           (L*m_Load*sin(x[1])*x[2]^2+u[1]+981*m_Load*sin(2*x[1])/200)
           /(m_Crab+m_Load*sin(x[1]))};
end f;

function h
  extends hBase;
  input Real p[3];
protected
  Real L = p[3];
algorithm
  y := {x[1], x[3]+L*sin(x[1])};
end h;

function fx
  extends fxBase;
  input Real p[3];
protected
  Real m_Load = p[1];
  Real m_Crab = p[2];
  Real L = p[3];
algorithm
  dfdx :=[[0,1,0,0];
          [-1/100*(-100*u[1]*sin(x[1])*m_Crab-100*sin(x[1])*L*x[2]^2
           *m_Load^2+100*m_Load^2*sin(x[1])*L*x[2]^2*cos(x[1])^2
           +200*m_Load*L*x[2]^2*cos(x[1])^2*m_Crab+981*cos(x[1])
           *m_Crab^2+981*cos(x[1])*m_Load*m_Crab-100*u[1]*m_Load
           -100*m_Load*L*x[2]^2*m_Crab)/L/(2*m_Crab*m_Load*sin(x[1])
           +m_Crab^2+m_Load^2-m_Load^2*cos(x[1])^2),-2*m_Load*x[2]*
           sin(x[1])*cos(x[1])/(m_Crab+m_Load*sin(x[1])),0,0];
          [0,0,0,1];
          [1/100*m_Load*(-981*m_Load*sin(x[1])+981*m_Load*sin(x[1])
           *cos(x[1])^2-100*u[1]*cos(x[1])+1962*cos(x[1])^2*m_Crab
           +100*L*x[2]^2*cos(x[1])*m_Crab-981*m_Crab)/(2*m_Crab*m_Load
           *sin(x[1])+m_Crab^2+m_Load^2-m_Load^2*cos(x[1])^2),
           2*m_Load*L*x[2]*sin(x[1])/(m_Crab+m_Load*sin(x[1])),0,0]];
end fx;

function hx
  extends hxBase;
  input Real p[3];
protected
  Real L = p[3];
algorithm
  dhdx := [1, 0, 0, 0; L*cos(x[1]), 0, 1, 0];
```

```
  end hx;

  function integrator
    extends integratorBase;
  // assumes, that u is constant between t and t+dt
    protected
    Real k1[size(x,1)];
    Real k2[size(x,1)];
    Real k3[size(x,1)];
  // Rule of Simpson, order 3
  algorithm
    k1 := f(x,                          u, t);
    k2 := f(x + 0.5*dt*k1,              u, t+0.5*dt);
    k3 := f(x -     dt*k1 + 2*dt*k2, u, t+    dt);
    xNew := x + (k1 + 4*k2 + k3)/6*dt;
  end integrator;
end SystemCrane;
```

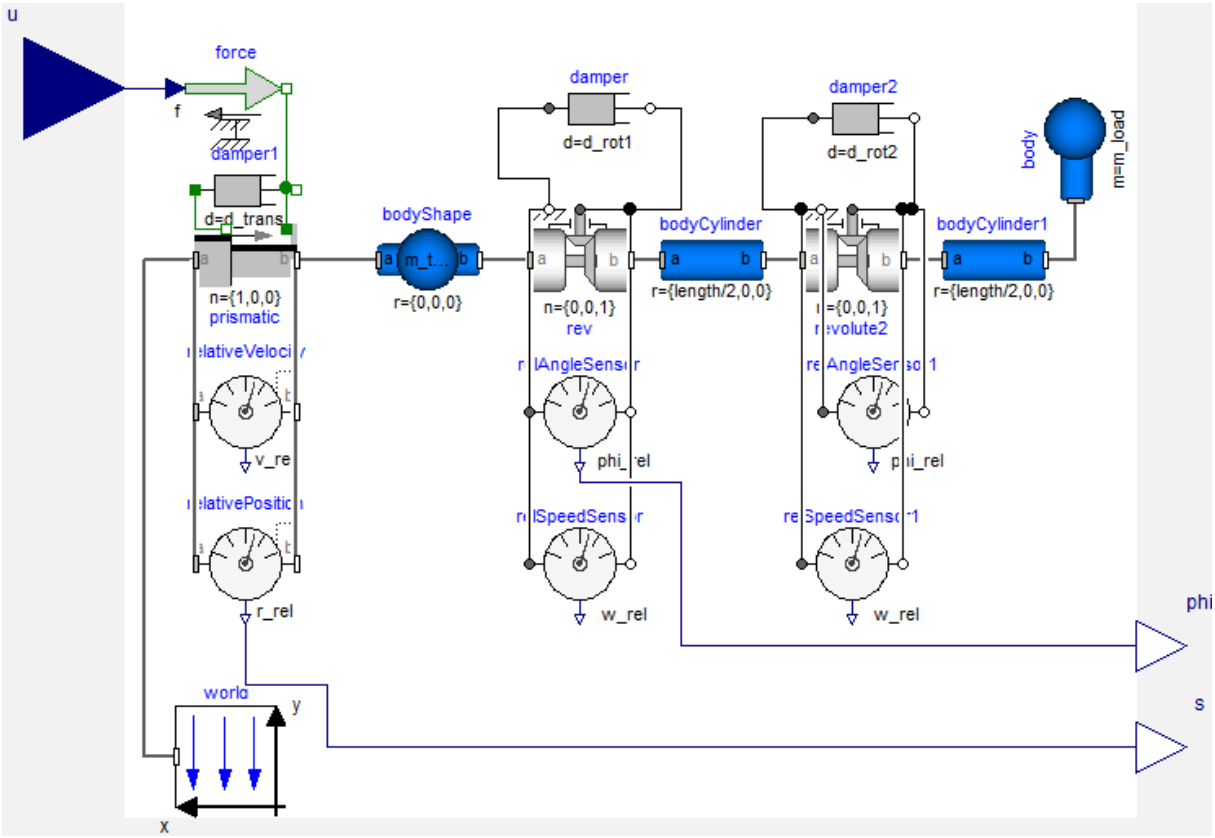
Here, the implemented one step integration algorithm is a third order explicit Runge-Kutta method that does not use the Jacobian matrix of the system function f. For the numerical integration from t to $t + dt$ it is assumed that the input u is constant within this time interval.

The following model shows how a filter algorithm can be called including *function partial applications* (according to Section 12.4.2.1 of [6]) that allow to transfer additional input variables (e.g. parameters) to the system functions:

```
model FilterModelCrane
  parameter Real p[3] = {4000,1000,10};
  parameter Real sampleTime = 0.01;
  package system = SystemCrane;
equation
  // ...
  when sample(0, sampleTime) then
    FilterAlgorithm(..., function system.f(p=p), function system.h(p=p),
                         function system.fx(p=p), function system.hx(p=p),
                         function system.integrator());
  end when;
end FilterModelCrane;
```

5.2. Double Pendulum

In this section a double pendulum is modeled by components of the Modelica Standard library, especially by components of the Multibody library, see the following figure. The model is exported to an FMU to get access to the system functions necessary for the application of filter algorithms.



The following Modelica package shows the typical structure of a package that interfaces an FMU for estimation purposes. It consists of a model `InitializationModel`, some constants of the FMU, an `fmiModel` class to incorporate the FMU as Modelica external object and the Modelica functions to interface the FMU system functions. Because only smooth continuous model functions are considered (cp. System (1.1)) for estimation purposes, it is not necessary to interface the FMI functions handling events, like `fmiEventUpdate`, `fmiGetEventIndicators` or `fmiCompletedEventIteration`.

The model `InitializationModel` has mainly two tasks. Firstly, it enables to set the model parameters of the FMU. Secondly, it instantiates and initializes the FMU when `InitializationModel` is instantiated in a Modelica model. The functions `fmiDoStep`, etc. are wrapper functions to the FMI interface functions in C. If the FMU is provided including C-Code, then the code can be compiled together with an overall model where the Kalman filter uses the package `DoublePendulumFMU`. Or, if a compiled version of the FMU is provided, the overall model has to be linked against the compiled FMU code.

```
package DoublePendulumFMU
  extends FMUImportTemplate(nx = 6, nu = 1, ny = 2,
        id_x = {369098876, ..., 369100155}, id_u = {352321536},
        id_y = {335544320, 335544321}, id_dx = {369098877, ..., 369100156},
        stateNames = {"prismatic.s", ..., "revolute2.w"},
        inputNames = {"u"},
        outputNames = {"s", "phi"});

  redeclare model extends InitializationModel
    parameter ...
    // All model parameters of the DoublePendulum Modelica model
    parameter ...
    parameter String fmiInstanceName = "DoublePendulumFMU";
    parameter Boolean fmiLoggingOn = false;
    fmiModel fmi;
```

```
  initial equation
    fmiEnterSlaveInitializationMode(...);
    fmiExitSlaveInitializationMode(...);
  equation
    when initial() then
      fmi = fmiModel(...);
    end when;
  end initializeModel;

  redeclare class fmiModel
      extends ExternalObject;
      function constructor "Initialize FMI model"
        input String instanceName;
        input Boolean loggingOn;
        input String resourceLocation;
        output fmiModel fmi;
        external"C" fmi = DoublePendulum_fmiInstantiateModel2(
                                  instanceName, loggingOn, resourceLocation)
      end constructor;

      function destructor "Release storage of FMI model"
        extends Modelica.Icons.Function;
        input fmiModel fmi;
        external"C" DoublePendulum_fmiFreeModelInstance2(fmi);
      end destructor;
  end fmiModel;

  redeclare function fmiDoStep
    input fmiModel fmi;
    input Real currentTime;
    input Real stepSize;
    input Real preAvailable;
    output Boolean stepOK;
    output Real postAvailable=preAvailable;
    external"C" stepOK= DoublePendulum_fmiDoStep2(fmi, currentTime,
                                                  stepSize);
    annotation (…);
  end fmiDoStep;
  // ... all necessary fmi functions
  // ...

end DoublePendulumFMU;
```

In the filter model the filter algorithm can be called in the same way as for the analytical example in Section 5.1 by use of the package `FMISystemFunctions` from Section 3.4. Only the FMU external object ID (`fmi`) has to be transferred to the system functions by a function partial application.

```
model FilterModelDoublePendulumFMU
  // ...
  parameter Real sampleTime = 0.01;
  package fmuPackage = DoublePendulumFMU;

  package system = FMISystemFunctions(redeclare package FMU = fmuPackage);
  fmuPackage.InitializationModel ModelParameters;
equation
  // ...
  when sample(0, sampleTime) then
    FilterAlgorithm(..., function system.f(fmi=ModelParameters.fmi),
```

```
                        function system.h(fmi=ModelParameters.fmi), ...);
  end when;
end FilterModelDoublePendulumFMU;
```

6. References

[1] Deliverable D3.1.1: *Method to extend models for system design to models for system operation*, ITEA2 project MODRIO (11004), Version 1.0, 2013.

[2] Modelica Association Project "FMI": *Functional Mock-up Interface for Model Exchange and Co-Simulation*, Version 2.0, July 2014. www.fmi-standard.org

[3] J. Brembeck, A. Pfeiffer, M. Fleps-Dezasse, M. Otter, K. Wernersson and H. Elmqvist: *Nonlinear State Estimation with an Extended FMI 2.0 Co-Simulation Interface*. In: Proceedings of the 10th International Modelica Conference. pp. 53-62, Lund, Sweden, March 2014.

[4] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban and D. E. Shumaker: *SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers*, ACM Transactions on Mathematical Software, vol. 31(3), pp. 363-369, 2005.

[5] H. Elmqvist, F. Cellier and M. Otter, *Inline Integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems*, in European Simulation Multiconference, Prague, 1995.

[6] Modelica Association: *Modelica - A Unified Object-Oriented Language for Systems Modeling*, Language Specification, Version 3.3, May 2012. www.modelica.org

[7] Deliverable D3.1.3: *Estimation blocks for continuous-time FMUs*, ITEA2 project MODRIO (11004), Version 1.0, 2014.