Nonlinear State Estimation with an Extended FMI 2.0 Co-Simulation Interface

Jonathan Brembeck[1], Andreas Pfeiffer[1], Michael Fleps-Dezasse[1],
Martin Otter[1], Karl Wernersson[2], Hilding Elmqvist[2]

[1]German Aerospace Center (DLR), Institute of System Dynamics and Control,
82234 Weßling, Germany
[2]Dassault Systèmes AB, Ideon Science Park, 22370 Lund, Sweden

Jonathan.Brembeck@dlr.de, Andreas.Pfeiffer@dlr.de, Michael.Fleps-Dezasse@dlr.de,
Martin.Otter@dlr.de, Karl.Wernersson@3ds.com, Hilding.Elmqvist@3ds.com

Abstract

In this paper we propose a method how to automatically utilize continuous-time Modelica models directly in nonlinear state estimators. The approach is based on an extended FMI 2.0 Co-Simulation Interface [1] that interacts with the state estimation algorithms implemented in a Modelica library [2]. Besides a short introduction to Kalman Filter based state estimation, we give details on a generic interface to cooperate with FMUs in Modelica, an implementation of nonlinear state estimation based on this interface, and the Dymola prototype used for the evaluation. Finally we show first results in a tire load estimation application [3] for DLR's robotic electric research platform ROMO [4].

Keywords: FMI 2.0 Co-Simulation, FMU, Inline Integration, Kalman Filter, State Estimation, Moving Horizon Estimation, Tire Load Estimation

1 Introduction

With the raise of computational power in the last decades the possibilities to implement complex control strategies in real world applications enhanced tremendously. For most of them a good knowledge of the actual states is necessary. Often these are not directly measurable due to cost limitations or missing sensors (for example, it is not practical to measure in-tire forces). In the ITEA2 project MODRIO [5] one aim is to develop state estimation technologies for plants that use the knowledge of complex models of the controlled system itself. These models are often designed, parameterized and optimized as multidomain models in Modelica. To re-use these models for estimation and control purposes the Functional Mockup Interface [1] turns out to be very useful. Three years ago, we presented a concept for state estimation [2], [6] using FMI 1.0 Model Exchange [7], using Modelica function pointers to separate the prediction model from the observer algorithms and to create an easy reconfigurable framework for state estimation purposes. This approach had several limitations and difficulties that we want to overcome based on a slightly extended FMI 2.0 Co-Simulation Interface [1] . Furthermore we introduced a different way how the user interacts with the prediction model and the desired state estimation method.

One goal of the research performed in MODRIO is to build-up a complete tool chain so that an end-user can utilize a complex model in a state estimation algorithm and download the estimator to an embedded target. To our knowledge such tool chains are not available today. There are toolboxes available for the estimation algorithms, such as [8], [9], but it is non-trivial and time-consuming to utilize them for a concrete application with a nonlinear model and download the result to a real-time target.

The following sections are organized as follows. In Section 2 we briefly recap the well-known Extended Kalman Filter, Unscented Kalman Filter and the Moving Horizon estimator algorithms. From these descriptions we deduce the requirements on a generic interface for nonlinear models. Afterwards, Section 3 gives a closer look into the Modelica implementation, as well as a proposal for a user-friendly configuration interface. Moreover, we show how the FMI 2.0 Co-Simulation needs to be extended to fit the requirements of the prediction steps sketched in Section 2. As a use-case we show an automotive tire load estimation application [3] in Section 4. To validate this approach measurement data acquired with DLR's ROboMObil [4] are used.

2 Model Evaluations in State Estimation Algorithms

It is assumed that the plant model to be used in state estimation is naturally described as nonlinear continuous-time state space system:

$$\begin{aligned} \dot{\boldsymbol{x}} &= \boldsymbol{f}(\boldsymbol{x},\boldsymbol{u}), \\ \boldsymbol{y} &= \boldsymbol{h}(\boldsymbol{x}), \end{aligned} \qquad (1)$$

$$t \in \mathbb{R}, \boldsymbol{u}(t) \in \mathbb{R}^{n_u}, \boldsymbol{x}(t) \in \mathbb{R}^{n_x}, \boldsymbol{y}(t) \in \mathbb{R}^{n_y}$$

where t is time, $\boldsymbol{u}(t)$ is the vector of inputs, $\boldsymbol{x}(t)$ is the vector of states and $\boldsymbol{y}(t)$ is the vector of outputs. Such a model shall be provided as a Functional Mockup Unit (FMU) [1]. In this paper plant models are defined in Modelica and exported as FMUs using Dymola. However, all the results are also valid if FMUs are generated by other tools and/or non-Modelica environments, as long as the FMU supports a slightly extended FMI 2.0 Co-Simulation Interface according to our proposal.

Model (1) cannot be utilized directly in a sampled data system. Instead a discrete-time representation is needed for use in a discrete-time state estimator. The following discrete-time version of (1) with additive Gaussian noise is used in the sequel:

$$\begin{aligned} \boldsymbol{x}_k &= \boldsymbol{f}_{k|k-1}(\boldsymbol{x}_{k-1},\boldsymbol{u}_{k-1}) + \boldsymbol{w}_{k-1}, \\ \boldsymbol{y}_k &= \boldsymbol{h}(\boldsymbol{x}_k) + \boldsymbol{v}_k, \\ \boldsymbol{w}_k &\sim N(0,\boldsymbol{Q}_k), \\ \boldsymbol{v}_k &\sim N(0,\boldsymbol{R}_k). \end{aligned} \qquad (2)$$

Here t_k is the k-th sample time instant of a periodically sampled data system, $\boldsymbol{u}_k = \boldsymbol{u}(t_k)$, $\boldsymbol{x}_k = \boldsymbol{x}(t_k)$, $\boldsymbol{y}_k = \boldsymbol{y}(t_k)$, $\boldsymbol{w}_k$, $\boldsymbol{v}_k$ is Gaussian noise, and

$$\boldsymbol{f}_{k|k-1} = \boldsymbol{x}_{k-1} + \int_{t_{k-1}}^{t_k} \boldsymbol{f}(\boldsymbol{x},\boldsymbol{u}_{k-1})\, dt. \qquad (3)$$

A tool chain has to support (3) because it is non-trivial to transform the natural description (1) in (2).

2.1 The Estimation Prediction Step

In this section, we briefly summarize the steps of Kalman Filter based state estimation and will then have a closer look to the prediction step (compare Figure 1) of the Extended Kalman Filter (EKF), the Unscented Kalman Filter (UKF), and a more complex Kalman Filter based algorithm the so-called Moving Horizon Estimation (MHE) [10]. Here the need of an efficient and reliable way for the feed forward model simulation rises tremendously. For further information regarding Kalman Filter techniques, see especially the standard textbook [11]. This section is based on [12], [11] and [2].

In Figure 1 a cycle flow diagram of a recursive Kalman Filter algorithm is depicted. The filter is initialized with the initial state vector guess $\hat{\boldsymbol{x}}_0^+$ and the initial guess of the state covariance matrix $\boldsymbol{P}_0^+$. These can be seen as a stochastic expectation for believe in the first guess of the estimation task.

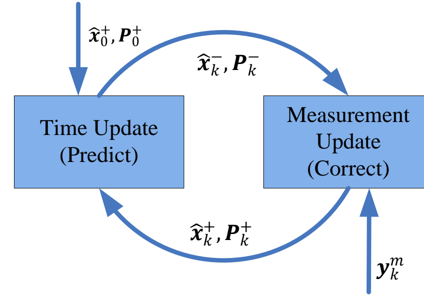


Figure 1: Principle of Kalman Filter based Estimation, $\boldsymbol{y}_k^m$ denotes the vector of measured outputs

Afterwards the cycle of the two steps *Predict* and *Correct* begins and is executed with a predetermined static sample time T_s. The additive Gaussian noise assumption in Eq. (2) is handled by the tuning covariance matrices $\boldsymbol{Q}, \boldsymbol{R}$. These enable the user to tune the filter to the specific task. For nonlinear model state estimation the widely used EKF algorithm is given as pseudo code in Table 1.

Table 1: Extended Kalman Filter Algorithm

Initialization:

$$\hat{\boldsymbol{x}}_0^+ = E(\boldsymbol{x}_0) \quad // \text{ expectation value}$$
$$\boldsymbol{P}_0^+ = E\left((\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)^T\right)$$

for $k = 1,2,\ldots$:

Predict: $\hat{\boldsymbol{x}}_k^- = \boldsymbol{f}_{k|k-1}(\hat{\boldsymbol{x}}_{k-1}^+, \boldsymbol{u}_{k-1})$

$$\boldsymbol{P}_k^- = \boldsymbol{F}_{k-1}\boldsymbol{P}_{k-1}^+\boldsymbol{F}_{k-1}^T + \boldsymbol{Q}$$

where $\boldsymbol{F}_{k-1} = e^{\left(\frac{\partial f}{\partial x}\Big|_{\hat{x}_{k-1}^+} \cdot T_s\right)}$

Correct: $\boldsymbol{K}_k = \boldsymbol{P}_k^-\boldsymbol{H}_k^T \cdot (\boldsymbol{H}_k\boldsymbol{P}_k^-\boldsymbol{H}_k^T + \boldsymbol{R})^{-1}$

where $\boldsymbol{H}_k = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\Big|_{\hat{x}_k^-}$

$$\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + \boldsymbol{K}_k \cdot (\boldsymbol{y}_k^m - \boldsymbol{h}(\hat{\boldsymbol{x}}_k^-))$$
$$\boldsymbol{P}_k^+ = (\boldsymbol{I} - \boldsymbol{K}_K \cdot \boldsymbol{H}_k) \cdot \boldsymbol{P}_k^-$$

The red marked sections indicate where the evaluation of the underlying system model equations (2) is necessary. The calculation of $\hat{\boldsymbol{x}}_k^-$ is performed by integrating model (1) from $t_{k-1}$to t_k. $\boldsymbol{F}_{k-1}$ is the state-transitions matrix of $\boldsymbol{f}$ with respect to $\boldsymbol{x}$ at $\hat{\boldsymbol{x}}_{k-1}^+$ and $\boldsymbol{H}_k$ is the partial derivative matrix of $\boldsymbol{h}$ with respect to $\boldsymbol{x}$ at $\hat{\boldsymbol{x}}_k^-$. The Jacobians $\boldsymbol{J}_{k-1}$ and $\boldsymbol{H}_k$ must either be provided directly, or they can be determined numerically, for example with a forward difference quotient:

for $i = 1,2,\ldots,n_x$; $\Delta \cong \sqrt{eps}$

$$(J_{k-1})_i = \frac{\boldsymbol{f}(\hat{\boldsymbol{x}}_{k-1} + \Delta\, \boldsymbol{e}_i, \boldsymbol{u}_{k-1}) - \boldsymbol{f}(\hat{\boldsymbol{x}}_{k-1}, \boldsymbol{u}_{k-1})}{\Delta} \qquad (4)$$

DOI 10.3384/ECP1409653

2.2 UKF Sigma Point Approach

The so called *Sigma Point Transformation* is based on the idea that it is easier to approximate a Gaussian distribution, than it is to approximate an arbitrary nonlinear function or transformation [13], [11]. The parts of the UKF algorithm, where model evaluations are necessary, are given in Table 2. The selection of the *Sigma Points* in matrix $\boldsymbol{X}$ is performed via a static scaling factor $\gamma(n, \alpha, \kappa)$ and the matrix square root of the a posteriori covariance matrix. The number of states is denoted by $n = n_x$, α is the spread around the last state value $\hat{\boldsymbol{x}}_{k-1}^{+}$ and κ is a parameter for the stochastic distribution assumption. In total $2n + 1$ points must be created and then used as initial values for $2n + 1$ simulations from t_{k-1} to t_k to compute $\boldsymbol{X}_{k|k-1}$.

Table 2: UKF Prediction Step

$$\boldsymbol{X}_{k-1} = \left[\hat{\boldsymbol{x}}_{k-1}, \hat{\boldsymbol{X}}_{k-1} + \gamma\sqrt{\boldsymbol{P}_{k-1}^{+}}, \hat{\boldsymbol{X}}_{k-1} - \gamma\sqrt{\boldsymbol{P}_{k-1}^{+}}\right]$$
$$\boldsymbol{X}_{k|k-1} = \boldsymbol{f}_{k|k-1}(\boldsymbol{X}_{k-1}, \boldsymbol{u}_{k-1})$$
$$\hat{\boldsymbol{x}}_k^{-} = \sum_{i=0}^{2\cdot n} \boldsymbol{w}_i^m \cdot \boldsymbol{X}_{k|k-1,i}$$
$$\boldsymbol{P}_k^{-} = \sum_{i=0}^{2\cdot n} \boldsymbol{w}_i^c \cdot \left(\boldsymbol{X}_{k|k-1,i} - \hat{\boldsymbol{x}}_k^{-}\right)\left(\boldsymbol{X}_{k|k-1,i} - \hat{\boldsymbol{x}}_k^{-}\right)^T + \boldsymbol{Q}$$
$$\boldsymbol{X}_k' = \left[\hat{\boldsymbol{x}}_k^{-}, \hat{\boldsymbol{X}}_k^{-} + \gamma\sqrt{\boldsymbol{P}_k^{-}}, \hat{\boldsymbol{X}}_k^{-} - \gamma\sqrt{\boldsymbol{P}_k^{-}}\right]$$
$$\boldsymbol{Y}_k = \boldsymbol{h}(\boldsymbol{X}_k')$$
$$\hat{\boldsymbol{y}}_k^{-} = \sum_{i=0}^{2\cdot n} \boldsymbol{w}_i^m \cdot \boldsymbol{Y}_{k,i}$$

The predicted values $\hat{\boldsymbol{x}}_k^{-}$, $\hat{\boldsymbol{y}}_k^{-}$, $\boldsymbol{P}_k^{-}$ are calculated via weighted sums with the predetermined weights $\boldsymbol{w}_i^{c,m}(n, \alpha, \kappa)$. We define $\hat{\boldsymbol{X}} := [\hat{\boldsymbol{x}}, \hat{\boldsymbol{x}}, \ldots, \hat{\boldsymbol{x}}] \in \mathbb{R}^{n\times n}$ and in our notation a vector depending function (e.g. $\boldsymbol{f}_{k|k-1}$ or $\boldsymbol{h}$) with a matrix argument returns a matrix with columns that are equal to the evaluated columns of the matrix argument.

It can be shown that the nonlinear approximation accuracy of the UKF is minimum twice higher compared to an EKF. This becomes important in case of strong nonlinearities in the prediction model (for a detailed proof see [14] – Appendix A).

2.3 MHE with NLG Method

The *Moving Horizon Estimator* (MHE) is very closely connected to *Model Predictive Control* (MPC). Instead of predicting future control inputs we have a sliding window (with M steps to the past) that moves every T_s one step ahead. Therefore, all past M measurements are taken into account.

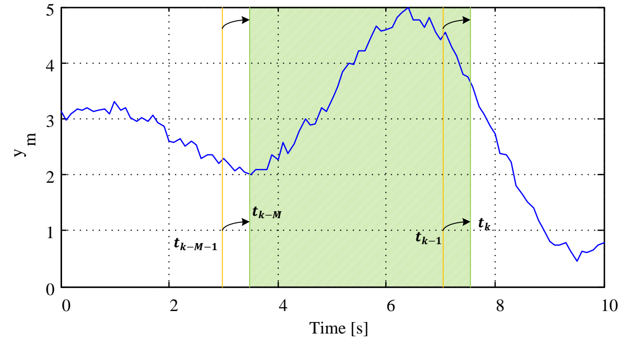


Figure 2: Moving window in MHE application

In this way the estimate gets more robust against external disturbances, delayed measurements can be incorporated and also constraints can be imposed directly [10]. Neglecting the last two points, the optimization objective can be written as follows:

Table 3: MHE Optimization Objective

$$\min_{\boldsymbol{\xi}_k} c(\boldsymbol{\xi}_k)$$
$$\boldsymbol{\xi}_k = (\boldsymbol{x}_{k-M}^T, \boldsymbol{x}_{k-M+1}^T, \ldots, \boldsymbol{x}_k^T)^T$$
$$\boldsymbol{x}_{int,k-M} = \hat{\boldsymbol{x}}_{k-M}^{+}$$
$$\boldsymbol{x}_{int,i} = \boldsymbol{f}_{i|i-1}\left(\boldsymbol{x}_{int,i-1}, \boldsymbol{u}_{i-1}\right) \quad (i = k - M + 1, \ldots, k)$$
$$c(\boldsymbol{\xi}_k) = \| \boldsymbol{x}_{k-M} - \hat{\boldsymbol{x}}_{k-M}^{+} \|_{\boldsymbol{I}_{k-M}^{+}}^2 + \sum_{i=k-M}^{k} \| \boldsymbol{y}_i^m - \boldsymbol{h}(\boldsymbol{x}_i) \|_{\boldsymbol{R}^{-1}}^2 + \sum_{i=k-M+1}^{k} \| \boldsymbol{x}_i - \boldsymbol{x}_{int,i} \|_{\boldsymbol{Q}^{-1}}^2$$

The initial constraint (first line in definition of $c(\boldsymbol{\xi}_k)$) is calculated via a Kalman Filter step i.e. an EKF from Table 1, wherein the information matrix $\boldsymbol{I}_{k-M}^{+} = (\boldsymbol{P}_{k-M}^{+})^{-1}$.

For its solution a *Nonlinear Decent Search* (NLG) is useful, because only the first derivatives of the system functions are needed, which is an important constraint for the available interfaces of FMI 2.0 (compare [1]). The algorithm of the unconstrained NLG is given in Table 4, for details please see [15]:

Table 4: MHE Optimization Algorithm

1. Set $j = 0$ and define $\boldsymbol{\xi}_k^0 = \left(\boldsymbol{x}_{int,k-M}^T, \ldots, \boldsymbol{x}_{int,k}^T\right)^T$
2. Decent direction:
   $$\boldsymbol{r}_j = -\nabla c\left(\boldsymbol{\xi}_k^j\right)$$
3. Line search to determine the step size:
   $$\eta_j = \underset{0\le\eta}{argmin}\, c\left(\boldsymbol{\xi}_k^j + \eta \cdot \boldsymbol{r}_j\right)$$
4. Optimization step:
   $$\boldsymbol{\xi}_k^{j+1} = \boldsymbol{\xi}_k^j + \eta_j \boldsymbol{r}_j$$
5. If stop criterion not reached:
   $j = j + 1$ and go to step 2;

In step 1 an initial solution $\boldsymbol{\xi}_k^0$ is needed. A good approach for its calculation is the open loop integration of the prediction model from $\boldsymbol{x}_{k-M}$ to $\boldsymbol{x}_k$. The gradi-

ent $\nabla c(\xi_k^j)$ of the decent direction can be calculated as shown in Table 5 (again, all parts that need a model evaluation are marked in red).

Table 5: MHE Gradient Calculation

$$\boldsymbol{R}^* = (\boldsymbol{R}\cdot\boldsymbol{R})^{-1}\,;\ \boldsymbol{Q}^* = (\boldsymbol{Q}\cdot\boldsymbol{Q})^{-1}$$

$$\nabla c_{1:n} = (\boldsymbol{I}_{k-M}^{+} + (\boldsymbol{I}_{k-M}^{+})^T)(\boldsymbol{x}_{k-M} - \hat{\boldsymbol{x}}_{k-M}^{+}) - 2\frac{\partial \boldsymbol{h}(\boldsymbol{x}_{k-M})}{\partial \boldsymbol{x}_{k-M}}^T \boldsymbol{R}^*\left(\boldsymbol{y}_{k-M}^m - \boldsymbol{h}(\boldsymbol{x}_{k-M})\right)$$

for $i = k - M + 1, \ldots, k$:

$$\nabla c_{1+(i-k+M)\cdot n:(i-k+M+1)\cdot n} = \frac{\partial c(\boldsymbol{\xi}_k)}{\partial \boldsymbol{x}_i} = 2\boldsymbol{Q}^*(\boldsymbol{x}_i - \boldsymbol{x}_{int,i}) - 2\frac{\partial \boldsymbol{h}(\boldsymbol{x}_i)}{\partial \boldsymbol{x}_i}^T \boldsymbol{R}^*\left(\boldsymbol{y}_i^m - \boldsymbol{h}(\boldsymbol{x}_i)\right)$$

2.4 Summary of Needed Model Evaluations

In Table 6 all needed evaluations of the prediction model (compare Eq. (1), (2)) for state estimation applications are summarized. A tool chain has to provide these model evaluations. In the right column the name of the Modelica function is listed to trigger the corresponding evaluation in the tool chain proposed by this article, for details see section 3.3.

Table 6: Model evaluations for nonlinear state estimation (in the right column the name of the Modelica functions are defined to trigger the evaluations in the tool chain proposed in this article, see section 3.3).

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

3 Nonlinear Kalman Filters and FMI

In Section 2 different state estimation algorithms are summarized. The goal of this paper is to provide a tool chain for nonlinear state estimation based on the model equations of a Modelica model. In [2] it is explained why a pure Modelica solution to reach this goal is currently not possible. Using FMI helps to overcome this situation. In [2] we concentrated on FMI 1.0 for Model Exchange with the drawback that the integration algorithm for performing prediction steps of a Kalman Filter has to be implemented in Modelica, a non-trivial task. FMI 2.0 for Co-Simulation [1] simplifies the implementation significantly because the integration algorithm, including event handling, is embedded inside the FMU. Still some features are missing. In a Dymola prototype these have been added in order that the "required" functions from Table 6 are supported.

The overall process of using a state estimator in Modelica is illustrated in Figure 3:

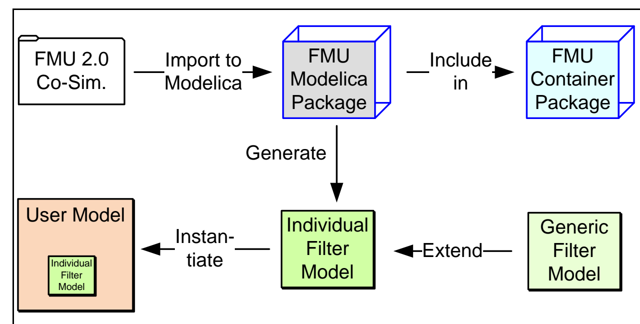


Figure 3: Process flow to generate a state estimator based on an FMU

An FMU (usually exported from a Modelica model) is imported into the Modelica environment by extending the package `FMUImportTemplate`. The imported package can be included in an FMU container package to collect several FMUs for easy access. For such an FMU package an *Individual Kalman Filter* model is generated that provides variable names on buses and user convenient parameter menus. The algorithmic part of the state estimation is provided in a *Generic Filter Model*. Finally, the individual filter model can be instantiated in the user's application model. An example is shown in the next figure:

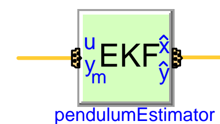


Figure 4: Instance of an individual EKF Kalman filter model generated by the process from Figure 3. The bus on the left side contains the individual input and measurement variables of the FMU and the bus on the right side contains the individual estimated state and output variables of the FMU.

In the following sub-sections, the details of the process from Figure 3 are described.

DOI 10.3384/ECP1409653

3.1 FMI for State Estimation with Dymola

The standard FMI Co-Simulation Interface allows integrating (1) from sample instant t_{k-1} to t_k with function `fmiDoStep(..)` and therefore computing (2). 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, see Section 2. In order to achieve this, Dymola 2014 FD01, that has already support for FMI 2.0 Co-Simulation according to [1], 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 `fmiSetReal(..)` before `fmiDoStep(..)` is called,
- it is possible to inquire the actual values of all variables with `fmiGetReal(..)` after `fmiSetReal(..)` was called, without an `fmiDoStep(..)` 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 section. This package serves as interface to access the needed FMI functionality from a Modelica model or function.

3.2 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, see code and figure below. The imported FMU extends from the `FMUImportTemplate` and redeclares all elements.

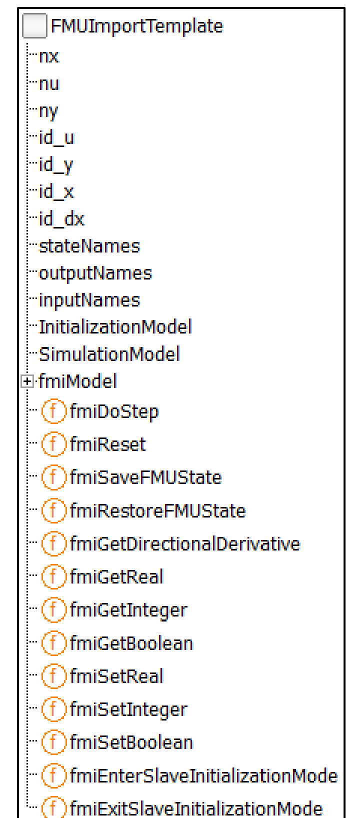
```
partial package FMUImportTemplate
  constant Integer nx=1;
  …
  constant Integer id_x[nx];
  …
  constant String stateNames[nx];
  …
  replaceable model SimulationModel
  end SimulationModel;

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

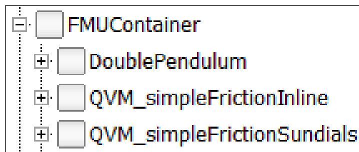
```
  replaceable partial class fmiModel
    extends ExternalObject;
    function constructor
      …
    end constructor;
  …
  end fmiModel;

  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 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 `SimulationModel` 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. For a user-convenient handling of the FMU import process, it is desirable to import an FMU as a sub-package into an existing Modelica package. The default package is the package FMUContainer that hosts several imported FMUs, see figure above.

3.3 Model Functions for State Estimators

The state estimator algorithms are implemented with Modelica functions that provide the needed model evaluations. In partial package `BaseFunctions` the interfaces of these functions are defined and in package `SystemFunctions` the function prototypes are collected. The latter are replaceable functions that provide the needed functionality of Table 6 (the right column of this table lists the name of the function).

For example, partial function `fBase` is 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;
```

The dimensions `nx`, `nu` are conceptually not necessary, because the dimensions could be determined by the size 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`:

```
partial package SystemFunctions
  replaceable function f
    extends fBase;
  end f;
  …
  replaceable function integrator
     extends integratorBase;
  end integrator;
end SystemFunctions;
```

For a particular model, an implementation of the `SystemFunctions` functions 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 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 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 supports directional derivatives for imported FMUs for the extended Co-Simulation case, that is function `fmiGetDirectionalDerivatives`, then this function can be directly called and will provide a more efficient and reliable evaluation of the Jacobians.

The Dymola prototype supports two techniques for the FMI function `fmiDoStep`. Either the *Sundials* solvers [16] are used (that are integrators with variable step size and error control) to numerically integrate the model equations, or *Inline integration* [17] is applied, that means fixed step solvers are embedded in the model equations. The Kalman Filter library works with both techniques. For real-time applications, fixed-step methods have to be used and therefore a Kalman filter will usually utilize Inline integration.

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

3.4 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 the Kalman Filter Library 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

DOI 10.3384/ECP1409653

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 Modelica syntax. The bus definitions for the use-case example in Section 4 are listed below:

```
encapsulated expandable connector InBus
  import Modelica;
  extends Modelica.Icons.SignalBus;
  // Model Inputs
  Real u;
  // Measured Model Outputs
  Real accBody;
  Real sRel;
  Real accArmUp;
end InBus;
encapsulated expandable connector OutBus
  import Modelica;
  extends Modelica.Icons.SignalBus;
  // Estimated Model States
  Real mass_wheel_s;
  Real mass_wheel_v;
  Real mass_body_s;
  Real mass_body_v;
  Real …FirstOrderShapingFilter_s;
  // Estimated Model Outputs
  Real accBody;
  Real sRel;
  Real accArmUp;
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 UKF (see Section 2.2). 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 `FMISystemFunction` all necessary parts are put together to run FMI based Kalman Filter algorithms within a Modelica model.

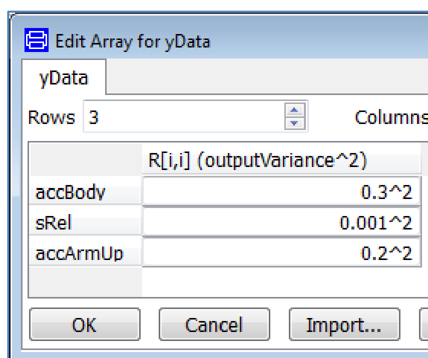


Figure 5: Parameter menu for output variances with names of output variables

A further improvement of the user interface compared to [2] 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 Figure 5. Basically, a matrix is defined and via Dymola specific annotations row and column headings can be added to the parameter menu. For example the menu in Figure 5 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 in Figure 6 the user can press the button on the right side of `yData` to get the menu of Figure 5. Also the model parameters of the FMU may be modified by clicking on the button on the right side of `ModelParameters`.

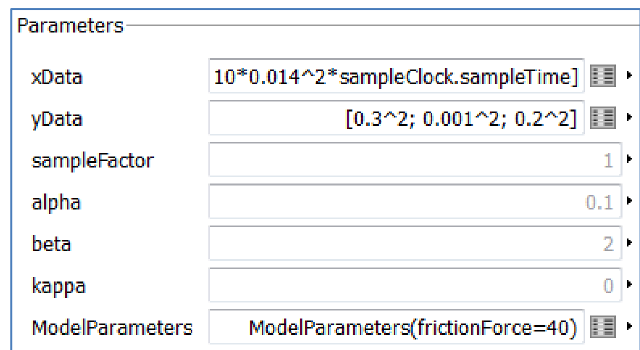


Figure 6: Menu of a UKF SR Kalman Filter model

4 Example: Vehicle Vertical Dynamics States Estimation

As an example application, the above described Kalman Filter Library is used to develop an advanced state estimator for the vertical dynamics of the ROboMObil. A more detailed version of this application case is available in [3]. This section is based on [3], but now the method and software from Section 3 are used. The ROboMObil is a robotic electric vehicle concept (see Figure 7) developed at the Robotics and Mechatronics Center of the German Aerospace Center DLR. It is comprised of four Wheel Robots (Figure 8 – left), which integrate traction motor, steering, brake system, spring and semi-active damper. Further details on the ROboMObil can be found in [4].

Figure 7: ROMO on the four-post test rig

In contrast to fully active suspension systems they need far less energy [19]. Therefore a diversity of control strategies for semi-active dampers is presented in literature. An overview about these control strategies can be found in [20] and [21]. As most of these strategies need state feedback, but not all states can be measured, state estimation becomes an important topic during the design of semi-active suspension systems.

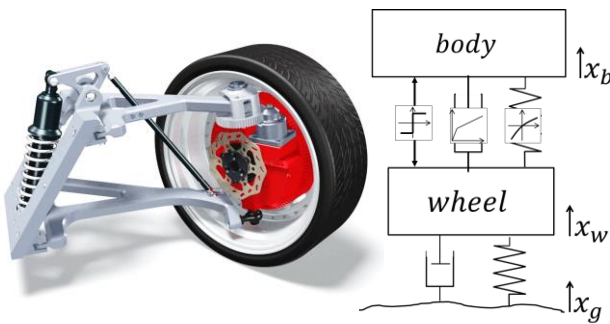


Figure 8: Left: the "Wheel Robot" concept, right: nonlinear two mass system [3]

The DLR *Kalman Filter Library* as presented in Section 3 offers an easy to use framework for the development of state estimators for nonlinear systems like this semi-active suspension system. Especially a square root utilizing implementation of the UKF (Subsection 2.2) algorithm SR-UKF is well suited for highly nonlinear systems, because of its higher order linearization accuracy of mean and covariance. Furthermore the nonlinear parts can be taken into account more easily than in an EKF algorithm (Subsection. 2.1) since no derivatives and Jacobians are needed.

4.1 The Nonlinear Quarter Vehicle Model

The suspension system of the Wheel Robot is modeled as a nonlinear two mass system (see Figure 8 – right) as described in [3]. The corresponding implementation in Modelica is shown in Figure 9. The model consists of the two masses *mass_body* and *mass_wheel*, a linear spring damper component, which approximates the wheel behavior, a road model as explained in [18] or [22] and the *body_spring* and *body_damper*.

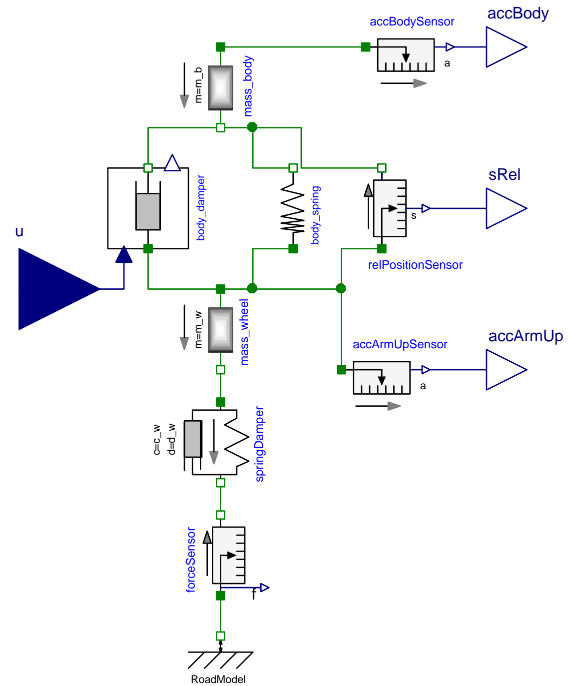


Figure 9: Nonlinear two mass system in Modelica

As the motion of these two components are connected to the wheel and body motion by a push rod-rocker kinematic (compare Figure 8 – left), the motion and the force of these components is scaled by a transmission ratio. Details on the nonlinear characteristic of the *body_damper* are shown in [3].

The third main nonlinearity of the two mass system besides the transmission ratio and the damper characteristic is the friction of the suspension system. It covers the friction of the damper and of all joints of the suspension system. For the state estimation the friction force F_f is modeled without stiction by a smooth tanh-switching function:

$$F_f = F_{f,const} * \tanh(v_d/\epsilon_f). \tag{5}$$

Here $F_{f,const}$ represents a constant sliding friction. The direction of the friction force is determined according to the current velocity difference v_d between body and wheel. The parameter ϵ_f is used to define the transitional behavior of the tanh-function.

4.2 Experimental Setup and Results

The nonlinear two mass system described in Section 4.1 is integrated in an SR-UKF state estimator using the DLR Kalman Library including the FMI 2.0 for Co-Simulation interface and Inline-integration as described in Section 3. Subsequently the resulting estimators are applied to the measurement data recorded with the ROboMObil on a four-post test rig. Figure 10 shows the Modelica model of the SR-UKF

DOI 10.3384/ECP1409653

estimator. On the left-hand side the measurement data is read by a *CombiTimeTable* and on the right-hand side the estimator, called *Filter*, and its corresponding settings block *observerControl* can be found. The estimator uses three measurement inputs: the acceleration of the body above the wheel, the wheel acceleration and the damper deflection.

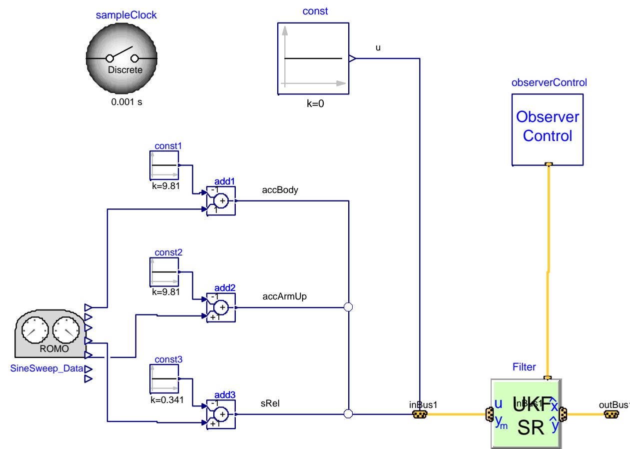


Figure 10: SR-UKF estimator in Modelica

The parameters of the estimators, as well as the system covariances, are tuned according to the optimization procedure presented in [3]. The measurement covariances are set according to the sensor noise. The experimental setup is shown in Figure 11.

The performance of the estimator, subject to a sine sweep excitation, is shown in Figure 11 by comparing the measured and the estimated tire contact forces in the last plot. Please notice that the tire force $F_{z_{measure}}$ is only available on the four-post test rig (Figure 7). It is used for validating the estimator performance and not as a measurement output y^m to it (compare experimental setup in Figure 10). Additionally the measurements are compared to the estimated measurements. It can be seen that the estimator reproduces the measurements and the tire contact force with a good accuracy.

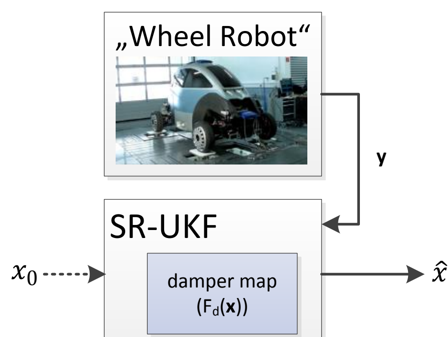


Figure 11: State estimator setup with measurement data from the four-post test rig

As the body accelerometer has the largest noise level, the weighting of its measurements *accBody* was chosen in such a way that the estimator relies more on the damper deflection *sRel* and the wheel acceleration *accArmUp*.

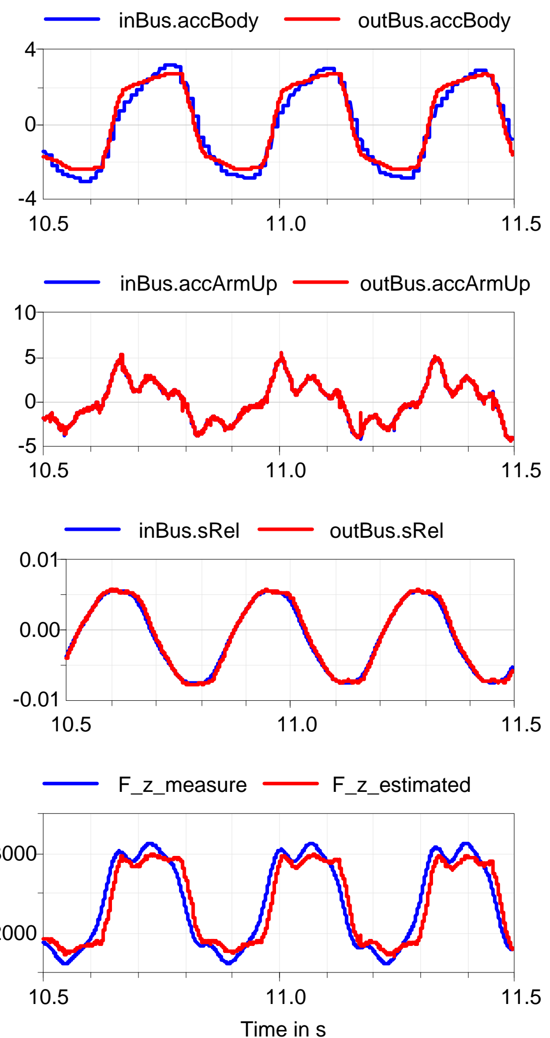


Figure 12: Comparison of measurements to estimated measurement (plot 1-3) and contact force estimation (plot 4) on the four-post test rig – "sine sweep" excitation

5 Conclusions and Outlook

In this paper we have shown how the FMI 2.0 standard for Co-Simulation [1] can be extended to use its capabilities for modern state estimation problems. We discussed the parts of the different estimation algorithm in detail, the needed evaluation of the system functions and integration between two sample instants via the FMI interface. Furthermore, we gave implementation details on a user friendly workflow as well as a set of necessary Modelica models and functions. The use case example of our vertical dynamics state estimation [3] application for our ROboMObil [4] showed good results in accuracy and computational efficiency. As a next step we plan to test them on a commercial real-time platform for stability and deterministic execution.

6 Acknowledgement

This paper is based on research performed within the ITEA2 project MODRIO. Partial financial support of the German BMBF and the Swedish VINNOVA for this development are highly appreciated.

Furthermore, the authors would like to thank Bernhard Thiele from DLR for his support to implement and cross-compile FMUs on real-time hardware and Marcus Baur for his support in implementing Kalman Filter algorithms in Modelica.

References

[1] **Functional Mockup Interface for Model Exchange and Co-Simulation 2.0 RC1**, Modelica Association, 18.10.2013. [Online]. Available [Accessed 25.1.2014]: https://www.fmi-standard.org/downloads#version2.

[2] J. Brembeck, M. Otter and D. Zimmer, **Nonlinear Observers based on the Functional Mockup Interface with Applications to Electric Vehicles**, in *Proceedings of 8th International Modelica Conference*, Dresden, 2011. Available [Accessed 25.1.2014]: http://www.ep.liu.se/ecp/063/053/ecp11063053.pdf.

[3] M. Fleps-Dezasse and J. Brembeck, **Model based vertical dynamics estimation with Modelica and FMI**, in *IFAC ACC*, National Olympics Memorial Youth Center, Tokyo, Japan, 2013.

[4] J. Brembeck, L. M. Ho, A. Schaub, C. Satzger, J. Tobolar, J. Bals and G. Hirzinger, **ROMO - the robotic electric vehicle**, in *22nd IAVSD International Symposium on Dynamics of Vehicle on Roads and Tracks*, Manchester, 11.-14. Aug. 2011.

[5] **Modrio ITEA2**, [Online]. Available [Accessed 25.1.2014]: https://modelica.org/external-projects/modrio http://www.itea2.org/project/index/view/?project=10114.

[6] C. Engst, **Object-Oriented Modelling and Real-Time Simulation of an Electric Vehicle in Modelica**, Masters Thesis, TUM EAL Munich, 2010.

[7] **Functional Mock-up Interface for Model Exchange,Version 1.0**, 25 January 2010. [Online]. Available [Accessed 25.1.2014]: https://www.fmi-standard.org/downloads#version1.

[8] J. Hartikainen and S. Särkkä, **Optimal filtering with Kalman filters and smoothers - A Manual for Matlab toolbox EKF/UKF**, February 2008. [Online]. Available [Accessed 25.1.2014]: http://www.lce.hut.fi/research/mm/ekfukf/.

[9] B. Houska, H. J. Ferreau and M. Diehl, **ACADO toolkit – An open-source framework for automatic control and dynamic optimization**, *Optimal Control Applications & Methods,* vol. 32, no. 3, pp. 298-312, 2010.

[10] D. Simon, **Kalman filtering with state constraints: a survey of linear and nonlinear algorithms**, *IET Control Theory & Applications,* vol. 4, no. 8, p. 1303+, 2010.

[11] D. Simon, **Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches**, 1. Edition ed., Wiley & Sons, 2006.

[12] J. Brembeck, **Method to extend models for system design to models for system operation**, MODRIO First Project Report, Munich, 2013.

[13] S. J. Julier and J. K. Uhlmann, **Unscented filtering and nonlinear estimation**, *Proceedings of the IEEE,* vol. 92, no. 3, pp. 401-422, March 2004.

[14] S. Haykin, **Kalman Filtering and Neural Networks**, Wiley-Interscience, 2001.

[15] J. Rosen, **The Gradient Projection Method for Nonlinear Programming. Part I. Linear Constraints**, *Journal of the Society for Industrial and Applied Mathematics,* vol. 8, no. 1, pp. 181-217, 1960.

[16] 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.
Software: Available [Accessed 25.1.2014] http://computation.llnl.gov/casc/sundials/main.html.

[17] 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. Available [Accessed 25.1.2014] http://www.inf.ethz.ch/personal/fcellier/Pubs/OO/esm_95.pdf.

[18] M. Mitschke and H. Wallentowitz, **Dynamik der Kraftfahrzeuge**, Berlin: Springer, 2004.

[19] R. Williams, **Electronically controlled automotive suspensions**, *Computing Control Engineering Journal,* vol. 5, no. 3, pp. 143-148, jun 1994.

[20] S. Savaresi, C. Poussot-Vassal, C. Spelta, O. Sename and L. Dugard, **Semi-Active Suspension Control Design for Vehicles**, Elsevier Science, 2010.

[21] E. Guglielmino, **Semi-active suspension control - Improved vehicle ride and road friendliness**, London: Springer, 2008.

[22] G. Koch, T. Kloiber, E. Pellegrini and B. Lohmann, **A nonlinear estimator concept for active vehicle suspension control**, in *Proc. American Control Conf. (ACC)*, Baltimore, 2010.