Chapter 6
The SNiMoWrapper: An FMI-Compatible Testbed for Numerical Algorithms in Co-simulation

Stefan Hante, Martin Arnold and Markus Köbis

Abstract We introduce the SNiMoWrapper, an FMI-compatible software tool which enables the integration of models with an integrated, adapted solver in the form of a co-simulation FMU into simulation tools by conducting the co-simulation and hiding its details from the simulator. We describe the used algorithm in detail, give a short proof for the order of convergence of the SNiMoWrapper, show results for its application to an academic test example and describe an industrial proof-of-concept application.

6.1 Introduction

Co-simulation is a simulation technique for time-dependent coupled problems in engineering that restricts the data exchange between subsystems to discrete communication points in time. In the present paper we follow the block oriented framework in the industrial interface standard FMI for Model Exchange and Co-Simulation v2.0, see [6], and present the SNiMoWrapper, a testbed for numerical algorithms in co-simulation. It is designed to include FMI-compatible software components (Functional Mock-up Units or FMUs) in a simulation tool like MATLAB, Simulink or Simpack. We discuss a sophisticated implementation for hiding algorithmic details of co-simulation from the master simulation tool which does not even need to be aware that co-simulation is performed inside the SNiMoWrapper.

The paper is structured as follows: In Sect. 6.2, we will motivate the conception of the SNiMoWrapper and how it is supposed to work in a software environment. In Sect. 6.3 we will describe the co-simulation algorithm that is embedded into the SNiMoWrapper. We will discuss how the SNiMoWrapper was implemented in software in Sect. 6.4. In Sect. 6.5 we will analyze how the numerical errors behave, if the communication step size $H \to 0$ and present a proof of convergence of this algorithm, which is based on the convergence analysis in [1]. Section 6.6 describes some

S. Hante (✉) · M. Arnold · M. Köbis
Institute of Mathematics, Martin Luther University Halle-Wittenberg,
06099 Halle (Saale), Germany
e-mail: stefan.hante@mathematik.uni-halle.de

B. Schweizer (ed.), *IUTAM Symposium on Solver-Coupling and Co-Simulation*,
IUTAM Bookseries 35, https://doi.org/10.1007/978-3-030-14883-6_6

test-cases to which the SNiMoWrapper has been applied as well as some numerical tests that support the theoretical results from Sect. 6.5. Furthermore, we discuss an industrial proof-of-concept application of the SNiMoWrapper.

6.2 Approach

The Functional Mock-up Interface (FMI, [6]) is a powerful industrial standard that allows quick and easy import and export of software components in order to simulate complex physical structures that are composed of smaller sub-structures which themselves are driven by different physical concepts. A model of such a sub-structure is comprised inside a so-called Functional Mock-up Unit (FMU), often together with a time integration method specialized to the governing differential equations.

Model exchange FMUs give the importing simulation tool access to the internal states and right-hand side functions of the model, allowing the simulation tool to simply include the right-hand side calls for the model in its time integration routine and perform a monolithic time integration. This, however, is often unfavorable, because the dimension of the differential equation that has to be solved becomes large. Additionally, in this case the same time-integration method has to be applied to all the smaller submodels that may draw from different physical concepts. Often, for each submodel, there is a specialized numerical integration scheme that is known to work well with the type of model in question.

With co-simulation FMUs, this can be realized. The co-simulation FMU comes with an embedded numerical integration scheme and the importing simulation tool has no access to internal variables or right-hand side functions etc. This approach on the other hand needs an additional co-simulation algorithm that manages the data exchange between the different software modules. These topics are addressed by two standard approaches: The simulation backplane method, where there is a co-simulation code that manages all involved software modules and the master-driven method, where the co-simulation is handled by one specific simulation tool—the master simulation tool.

Our approach, however, is different: The whole co-simulation aspect is dealt with inside the SNiMoWrapper that is situated between the master simulation tool and the FMU that includes the so-called slave system. It effectively transforms the slave FMU into a function that can be called by the master simulation tool—with some restrictions—at any simulation time instance and therefore hides the details of the co-simulation from the master simulation tool.

The SNiMoWrapper is based on existing proprietary application programming interfaces (API) of industrial simulation software like m- or s-function in MATLAB/Simulink or user-defined force elements in industrial multi-body system simulation software. For a single master simulation tool an API front-end has to be developed whose only task is to communicate with the SNiMoWrapper. The SNiMoWrapper manages the co-simulation and communicates via FMI with the slave systems contained inside the FMU. The FMU comes with FMI v2.0 routines defined

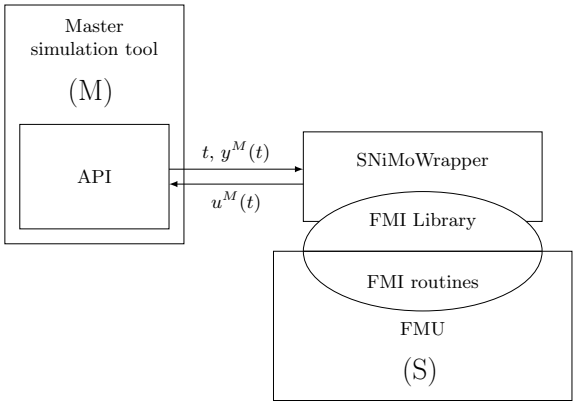


Fig. 6.1 SNiMoWrapper's workflow

in the standard [6], while the SNiMoWrapper uses a library to access these routines. Figure 6.1 shows workflow and basic software component of the SNiMoWrapper. The involved quantities $y^M(t)$ and $u^M(t)$ will be explained below.

In FMI for co-simulation, the data exchange between master and slave FMUs is restricted to discrete communication points T_i. During the communication step $(T_i, T_{i+1}]$ the slave systems are solved independently by their embedded solver [6]. The data exchange is handled by the SNiMoWrapper.

6.3 Co-simulation Algorithm

SNiMoWrapper's co-simulation algorithm works with an equidistant communication point grid $T_i = T_0 + i\,H$ with constant communication step size $H > 0$.

Since, at this point, we want to couple a single co-simulation FMU with a master simulation tool, we have to consider two systems: The master system that represents the black-box system of the master simulation tool and the slave system that represents the co-simulation FMU. In the context of co-simulation, this block-oriented description was introduced in [9]. The coupled master-slave system is modeled mathematically by the following coupled set of differential equations

$$\left.\begin{aligned} \dot{x}^M(t) &= f^M\left(t, x^M(t), u^M(t)\right) \\ y^M(t) &= g^M\left(t, x^M(t), u^M(t)\right) \end{aligned}\right\} \tag{6.1}$$

$$\left.\begin{aligned} \dot{x}^S(t) &= f^S\left(t, x^S(t), u^S(t)\right) \\ y^S(t) &= g^S\left(t, x^S(t), u^S(t)\right) \end{aligned}\right\} \tag{6.2}$$

Fig. 6.2 Co-simulation setup of the SNiMoWrapper

where the coupling conditions are

$$\left.\begin{aligned} u^M(t) &= y^S(t) \\ u^S(t) &= y^M(t) \end{aligned}\right\}. \tag{6.3}$$

Here, x^M and x^S are internal states, u^M and u^S are inputs and y^M and y^S are outputs of the master and slave system, respectively. In Fig. 6.2, the black-boxes and their coupling is shown.

6.3.1 Prerequisites

In this paper, we are not dealing with coupled systems that have algebraic loops [1]. A system without direct feed-through is always free from algebraic loops, for instance.

The master tool may call the SNiMoWrapper through the API at any time instance t inside the current communication step $(T_i, T_{i+1}]$ in order to obtain $u^M(t)$. The master can call the SNiMoWrapper at any $T_i < t \leq T_{i+1}$ in any order but restricted to these bounds. This means, in particular, it may not go back to a previous communication step. The FMI standard does, in principle, allow for saving and reinitializing an FMU to a previous time instance. However, for many industrial models, it is a rather strong condition that the necessary routines `fmi2GetState`, `fmi2SetState` are implemented.

Before the master simulation tool can advance to the next communication step $(T_{i+1}, T_{i+2}]$, it has to call the SNiMoWrapper at T_{i+1}, so the SNiMoWrapper can obtain the master system's output $y^M(T_{i+1})$.

These prerequisites can typically easily be fulfilled by industrial simulation software.

6.3.2 Capabilities of the Slave System

The slave system is part of a co-simulation FMU, therefore there are only a few things that we can do with it. For co-simulation, FMI offers functions for setting slave inputs, performing time integration and retrieving slave outputs:

Setting inputs If the slave system is currently at time t^*, then we may set the inputs $u^S(t^*)$ at this time instance via the FMI function `fmi2SetReal`. If we were to perform a time integration of the slave system, then the inputs will be considered constant for the whole slave communication step. In order to provide inputs that vary in time, we will need to use the FMI function `fmi2SetRealInputDerivatives`. Using this, we can give a polynomial $Q(t)$ of maximum degree ρ to the slave system, that is represented by its derivatives at t^* (Nordsieck representation, see [6]) rather than by its coefficients:

$$Q(t) = \sum_{R=0}^{\rho} \frac{1}{k!} Q^{(R)}(t^*)(t - t^*)^R.$$

Via the FMI function `fmi2SetRealInputDerivatives`, we can give the polynomial's derivatives $Q^{(R)}(t^*)$ of order $R > 0$ to the slave system. Note that FMUs only support input derivatives up to a certain order `maxSlaveInputDerivatives`, which we denote by r. This effectively means that they can only handle inputs that are polynomials of degree up to $r =$ `maxSlaveInputDerivatives`.

Performing time integration The time integration is triggered using the FMI function `fmi2DoStep`. It will run the time integration method of the FMU from $t^* \to t^* + H_S$ with a given slave communication step size $H_S > 0$, using the previously given inputs.

Getting outputs After the time integration was performed, we need to retrieve the outputs $y^S(t^* + H_S)$, where $t^* + H_S$ is the new current time of the slave system. This is done using the FMI function `fmi2GetReal`.

6.3.3 The Algorithm

The co-simulation algorithm of the SNiMoWrapper is a serial Gauss–Seidel co-simulation method involving higher order inter- and extrapolation of the inputs and outputs. We will present two variants of the algorithm: The basic algorithm and the extended algorithm. In the extended algorithm, we have put special attention on FMUs that do not support time-varying inputs, thus $r =$ `maxSlaveInputDerivatives` $= 0$ as well as on FMUs that have highly-oscillating outputs. Pseudo code of the basic algorithm can be found in Algorithm 1 and of the extended algorithm in Algorithm 2. Furthermore, the extended algorithm is visualized in Fig. 6.3.

Note that due to the coupling conditions (6.3) we only use the terms y^S and y^M for the data that is stored inside the SNiMoWrapper. Throughout this paper we will use $p > 0$ as the order of interpolation of the master outputs y^M, $q > 0$ as the order of extrapolation of the slave outputs y^S and $r =$ `maxSlaveInputDerivatives` the maximal degree of polynomial that the slave can handle as input data. Also, we introduce the notation of the inter- and extrapolation polynomial π of maximum degree m in the following form

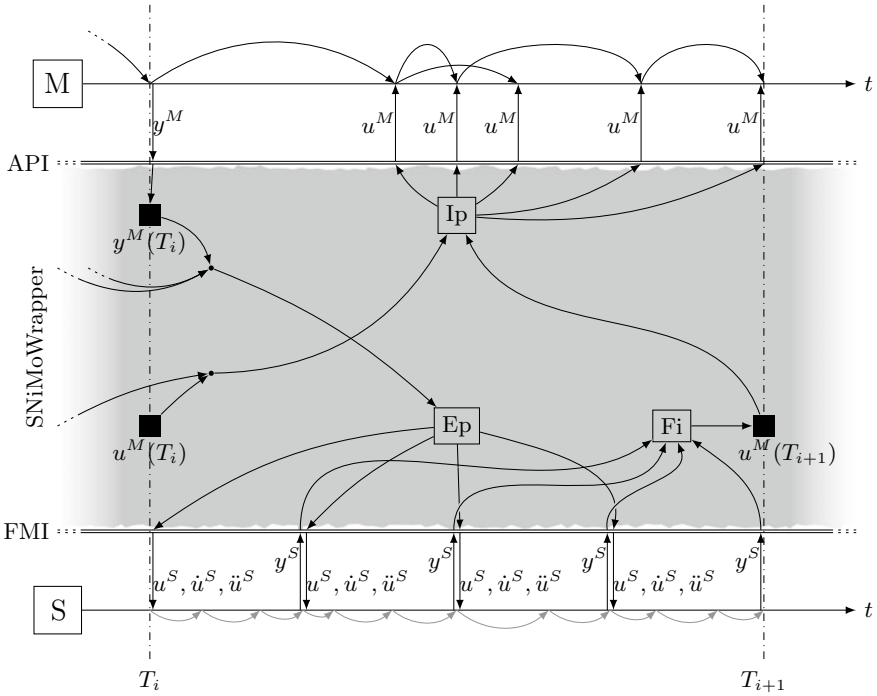


Fig. 6.3 Visualization of the SNiMoWrapper's extended algorithm with $n = 4$ and $p = q = r = 2$. Ip stands for interpolation, Ep for extrapolation and Fi for filtering

$$\pi(\tau_j; \varphi; \tau_0, \ldots, \tau_m) = \varphi(\tau_j), \qquad (j = 0, \ldots, m),$$

i.e., $\pi(\tau; \varphi; \tau_0, \ldots, \tau_m)$ denotes the interpolation polynomial of a given function φ using the supporting points $\tau_0, \ldots, \tau_m$. The maximum degree of the polynomial is determined by the number of arguments $\tau_0, \ldots, \tau_m$.

The basic algorithm This variant of the algorithm is a well-known approach to co-simulation and uses higher order interpolation of the slave outputs and higher order extrapolation of the master outputs.

Calculating master inputs Assume that the master simulation tool requests input $u^M(t)$ with $t \in (T_i, T_{i+1}]$. The SNiMoWrapper will then interpolate the $p + 1$ master inputs $u^M(T_{i+1}), \ldots, u^M(T_{i+1-p})$, that are already known to the SNiMoWrapper with an interpolation polynomial of maximum degree p:

$$u^M(t) = \pi(t; u^M; T_{i+1-p}, \ldots, T_{i+1}).$$

The result of this calculation is then passed to the master simulation tool.

Calculating slave outputs When the master simulation tool requests input $u^M(t)$ with $t \in (T_i, T_{i+1}]$ for the first time, then $u^M(T_{i+1})$ has not been calculated yet. In order

to do so, we want to calculate the interpolation polynomial

$$u^S(t) = \pi(t; y^M; T_{i-q}, \ldots, T_i)$$

of the $q+1$ master outputs $y^M(T_i), \ldots y^M(T_{i-q})$, that have been passed to the SNiMoWrapper at the end of all preceding communication steps. This interpolation polynomial is then passed to the slave system in its Nordsieck form of degree r

$$\big((u^S)^{(R)}(T_i)\big)_{R=0}^{r} = \big(u^S(T_i),\ (u^S)'(T_i),\ \ldots,\ (u^S)^{(r)}(T_i)\big),$$

because the slave only accepts polynomial of order up to r. Now the integration of the slave system from $T_i \to T_i + H = T_{i+1}$ is triggered. After the integration is finished, the master inputs at T_{i+1} are the slave outputs $u^M(T_{i+1}) := y^S(T_{i+1})$, which are retrieved and stored in the SNiMoWrapper.

As a variant, we can use linear interpolated extrapolation [5]

$$\begin{aligned} u^S(t) &= \pi_{ie}(t; y^M; T_i, T_{i-1}, T_{i-2}) \\ &= 2y^M(T_{i-1}) - y^M(T_{i-2}) + \frac{t - T_i}{H}\big(2y^M(T_i) - 3y^M(T_{i-1}) + y^M(T_{i-2})\big), \end{aligned}$$

that interpolates the results of two extrapolations at adjacent communication points. This approach will result in a continuous input signal [4, 5], but will restrict the order of the co-simulation algorithm to second order, since π_{ie} is a polynomial of maximum degree one, see Sect. 6.5 below.

In Algorithm 1 we have used the symbol $\tilde{\pi}$ to denote either the classical interpolation polynomial ($\tilde{\pi} = \pi$) or the interpolated extrapolation polynomial ($\tilde{\pi} = \pi_{ie}$).

Algorithm 1 Basic algorithm of the SNiMoWrapper

1: **if** $t = T_{i+1}$ **then** Save $y^M(T_{i+1})$
2: **if** $t > T_{i+1}$ **then**
3: $\quad i \leftarrow i + 1$
4: $\quad$ Run slave from T_i to $T_i + H$ with input data $\big((u^S)^{(R)}(T_i)\big)_{R=0}^{r}$, where

$$u^S(\tau) = \tilde{\pi}(\tau; y^M; T_{i-q}, \ldots, T_i)$$

5: $\quad$ Retrieve slave output $y^S(T_{i+1})$
6: $\quad u^M(T_{i+1}) := y^S(T_{i+1})$
7: **return** $u^M(t) = \pi(t; u^M; T_{i-(p-1)}, \ldots, T_i, T_{i+1})$

The extended algorithm Our extended algorithm is focused on FMUs that do not support higher order signal extrapolation, i.e. when $r = 0$ or very small, and on FMUs with highly oscillating behaviour, where even high order polynomial signal extrapolation can not be used to approximate the dynamics in a stable manner. The main idea is to introduce substeps in each master communication step $(T_i, T_{i+1}]$.

Throughout this paper let $n > 0$ be the number of substeps per master communication step and $H_S = H/n$ the slave communication step size.

The calculation of master inputs is done in exactly the same way, as in the basic algorithm. The main difference lies in the calculation of the slave outputs:

When the master simulation tool requests input $u^M(t)$ with $t \in (T_i, T_{i+1}]$ for the first time, as before, $u^M(T_{i+1})$ has not been calculated yet. Again, we consider the polynomial

$$u^S(t) = \tilde{\pi}(t; y^M; T_i, \ldots, T_{i-q}),$$

which is either the classical interpolation polynomial ($\tilde{\pi} = \pi$) of maximum degree q or the linear interpolated extrapolation polynomial ($\tilde{\pi} = \pi_{ie}$). Now, for $k = 0, \ldots, n-1$, we will recursively let the slave system integrate from $T_i + kH_S \to T_i + (k+1)H_S$ and afterwards retrieve the output $y^S(T_i + (k+1)H_S)$. The input data for each integration is the Nordsieck representation of order r of the polynomial $u^S(t)$ at the point $T_i + kH_S$:

$$\left((u^S)^{(R)}(T_i + kH_S)\right)_{R=0}^{r}.$$

After the slave system has reached the time instance $T_i + nH_S = T_i + H = T_{i+1}$, we can either use

$$u^M(T_{i+1}) := y^S(T_i + nH_S)$$

in order to continue the algorithm, or the retrieved slave outputs $y^S(T_i + kH_S)$ can be filtered. We have implemented the mean value filter

$$u^M(T_{i+1}) := \frac{1}{n} \sum_{k=1}^{n} y^S(T_i + kH_S)$$

in order to better support FMUs with highly oscillating outputs, where the high frequency oscillations are not relevant to the master system. In Algorithm 2 and Fig. 6.3 the filter function is denoted by the symbol Fi.

6.3.4 Initialization

The first call of the SNiMoWrapper should be at $t = T_0$ and the master tool is supposed to pass $y^M(T_0)$ to the SNiMoWrapper, in order to use it in its co-simulation algorithm. The SNiMoWrapper will then initialize the FMU and retrieve $y^S(T_0)$ from it. Since the algorithm needs y^S and y^M from previous communication points, we introduce the ghost communication points and ghost quantities

Algorithm 2 Extended algorithm of the SNiMoWrapper

1: **if** $t = T_{i+1}$ **then** Save $y^M(T_{i+1})$
2: **if** $t > T_{i+1}$ **then**
3: $i \leftarrow i + 1$
4: **for** $k = 0, \ldots, n-1$ **do**
5: Run slave from $T_i + kH_S$ to $T_i + (k+1)H_S$ with input data

$$\left((u^S)^{(R)}(T_i + kH_S)\right)_{R=0}^{r},$$

where $u^S(\tau) = \tilde{\pi}(\tau; y^M; T_{i-q}, \ldots, T_i)$
6: Retrieve slave output $y^S(T_i + (k+1)H_S)$
7: $u^M(T_{i+1}) := \mathrm{Fi}\left(y^S(T_i + H_S), \ldots, y^S(T_i + nH_S)\right)$
8: **return** $u^M(t) = \pi(t; u^M; T_{i-(p-1)}, \ldots, T_i, T_{i+1})$

$$\begin{aligned} T_{-j} &:= T_0 - jH, & j &= 1, \ldots, \max\{p-1, q\} \\ y^S(T_{-j}) &:= y^S(T_0), & j &= 1, \ldots, p-1, \\ y^M(T_{-j}) &:= y^M(T_0), & j &= 1, \ldots, q. \end{aligned}$$

Although these starting values are only approximations of first order, for problems being dominated by time-dependent external excitations, the co-simulation algorithm still reaches its higher order. This can be seen in the numerical experiments in Sect. 6.6. For other problems the order of the algorithm will drop to first order if no higher-order initialization is used.

6.4 Implementation

The SNiMoWrapper was implemented in plain C due to the great portability of C code and the possibility to use the libraries listed below. The SNiMoWrapper library was compiled using the GNU C compiler from the GNU compiler collection (https://gcc.gnu.org/). We have tested the implementation on Windows as well as on Linux systems.

The SNiMoWrapper can be compiled into either a shared library, which then can be loaded by the master simulation tool, or can be compiled into a static library that can be linked into the simulation tool. Of course, the simulation tool has to be compatible with the compiler that was used to compile the SNiMoWrapper library.

For the incorporation of the FMI v2.0 routines, we used the open-source FMI Library FMIL from Modelon AB (http://jmodelica.org/FMILibrary). The setting of parameters for the SNiMoWrapper is handled by initialization files. In order to load and interpret the ini files, we have used the open-source library inih by Brush Technology (http://code.google.com/p/inih/).

The SNiMoWrapper is designed to support the handling of multiple FMUs, so the master simulation tool only needs to load the SNiMoWrapper once. This is done

using an ID that refers to a specific FMU. The usage of multiple instances of the same FMU is possible as well.

The APIs should be designed to give the SNiMoWrapper the time t at which the master system needs input data as well as the output data $y^M(t)$ at this time. If the master tool calls the SNiMoWrapper in a new communication step $(T_{i+1}, T_{i+2}]$, the SNiMoWrapper recognizes this and infers, that the last master output data must have been $y^M(T_{i+1})$.

We implemented a simple API for MATLAB, that uses MATLAB's capabilities to load external shared libraries. Based on this API, we implemented a custom Simulink block that enables the usage of the SNiMoWrapper in Simulink and acts as an API.

Furthermore, we implemented an API for the industrial multi-body system tool Simpack using the user-force-elements that are written in Fortran code. The SNiMoWrapper was compiled into a static library and linked to the compiled user-force-element code.

6.5 Accuracy

The convergence of co-simulation algorithms for systems without algebraic loops was studied in [1]. There it was assumed that the subsystems are solved with sufficiently small tolerances, so the analytic solution in each time-integration step is considered in order to concentrate on the co-simulation algorithm, following [2].

The result from [1] can be directly applied to the basic algorithm: The interpolation of the slave outputs is done with a polynomial of maximum degree q. The extrapolation of the master outputs is done with a polynomial of maximum degree p in the case of classical polynomial extrapolation and in the case of interpolated extrapolation the polynomial is at most linear. However, we can only give a polynomial of order r to the slave system. From this it follows that the overall interpolation error must be of order $\min\{q, p, r\} + 1$ in the polynomial extrapolation case and $\min\{q, 1, r\} + 1$ for interpolated extrapolation. From [1] it follows that the global error of the basic algorithm is of size $\mathcal{O}(H^{\min\{q,p,r\}+1})$ or $\mathcal{O}(H^{\min\{q,1,r\}+1})$, respectively.

In the following, we will adapt the convergence analysis from [1] considering filtered slave output data and the substeps of the extended algorithm. Throughout the proof we will only use the inputs of the systems rather than the outputs, since they can easily be obtained by the coupling conditions (6.3). Let x^M, u^M, x^S and u^S be the analytic solutions of (6.1)–(6.3). The numerical solutions will be written with capital letters X^M, U^M, X^S, U^S and are for $t \in (T_i, T_{i+1}]$ the analytic solutions to

$$\dot{X}^M(t) = f^M\big(t, X^M(t), \Psi^M(t)\big), \tag{6.4a}$$

$$\Psi^M(t) = \pi(t; U^M; T_{i+1-p}, \ldots, T_{i+1}), \tag{6.4b}$$

$$U^S(T_{i+1}) = g^M\big(X^M(T_{i+1}), U^M(T_{i+1})\big), \tag{6.4c}$$

$$\dot{X}^S(t) = f^S\big(t, X^S(t), \Psi^S(t)\big), \tag{6.4d}$$

$$\Phi(t) = \pi(t; U^S; T_{i-q}, \ldots, T_i), \tag{6.4e}$$

$$\Psi^S(t) = \mathcal{T}_r(t; T_k + kH_S; \Phi), \qquad \begin{cases} t \in (T_i + kH_S, T_i + (k+1)H_S], \\ k = 0, \ldots, n-1, \end{cases} \tag{6.4f}$$

$$U^M(T_{i+1}) = \mathrm{Fi}\Big[\big(g^S\big(X^S(T_i + kH_S), \Psi^S(T_i + kH_S)\big)\big)_{k=1}^{n}\Big], \tag{6.4g}$$

where $\mathcal{T}_r(\tau; \tau^*; \varphi)$ is the Taylor polynomial of order r around τ^* of function φ evaluated at τ. This Taylor polynomial describes the truncation of the Nordsieck representation to order r. Because the functions Ψ^S and Ψ^M are fundamentally different, we will consider the components of the master and slave system separately.

First, we want to consider the error in the internal states for a step $T_i \to T_i + H$, so let $t \in (T_i, T_{i+1}]$. A perturbation analysis of the ordinary differential equations (6.4a) and (6.4d) together with their dependence on the initial value [13] gives for $B \in \{S, M\}$

$$\begin{aligned} \|X^B(T_{i+1}) - x^B(T_{i+1})\| \le{}& e^{L_1 H}\|X^B(T_i) - x^B(T_i)\| \\ &+ L_2 \frac{e^{L_1 H} - 1}{L_1} \max_{t \in [T_i, T_{i+1}]} \|\Psi^B(t) - u^B(t)\| \end{aligned} \tag{6.5}$$

with some constants $L_1, L_2 > 0$. For the master system the last difference can be estimated by

$$\Psi^M(t) - u^M(t) = \mathcal{O}(1) \sum_{j=0}^{p} \big(U^M(T_{i+1-j}) - u^M(T_{i+1-j})\big) + \mathcal{O}(H^{p+1}), \tag{6.6}$$

because the interpolation polynomial is linear in its values of support. Furthermore, we need a standard result on the error of polynomial interpolation. For the slave system, we need to deal with the truncation to degree r:

$$\Psi^S(t) - u^S(t) = \mathcal{O}(1) \sum_{j=0}^{q} \big(U^M(T_{i-j}) - u^M(T_{i-j})\big) + \mathcal{O}(H^{q+1} + H_S^{r+1}). \tag{6.7}$$

Here, we have the term H_S^{r+1} that results from the fact that the truncation to degree r happens in an interval of length H_S. Of course, asymptotically it is $\mathcal{O}(H_S^{r+1}) = \mathcal{O}(H^{r+1})$, but the more careful way of writing H_S^{r+1} explains the results for coarse H and larger n and therefore smaller H_S in Test 2 in Sect. 6.6.

We introduce the following notation for the errors in the states and inputs for $B \in \{S, M\}$:

$$\epsilon_i^B := \|X^B(T_i) - x^B(T_i)\|,$$
$$\eta_i^B := \|U^B(T_i) - u^B(T_i)\|.$$

From (6.4c) and (6.4g) it now follows

$$U^S(T_{i+1}) - u^S(T_{i+1}) = \mathcal{O}(1)\epsilon^M_{i+1} + J^M_{i+1}\big(U^M_{i+1} - u^M(T_{i+1})\big), \tag{6.8a}$$

$$U^M(T_{i+1}) - u^M(T_{i+1}) = \mathcal{O}(1)\epsilon^S_{i+1} + \mathcal{O}(1)J^S_{i+1}\sum_{j=0}^{q}\big(U^S_{i-j} - u^S(T_{i-j})\big) + \mathcal{O}\big(H^{q+1} + H_S^{r+1} + F(H)\big), \tag{6.8b}$$

where J^M_{i+1} and J^S_{i+1} are Jacobians of $g^M = g^M(x^M, u^M)$ and $g^S = g^S(x^S, u^S)$ with respect to u^M and u^S, respectively. The term $F(H)$ represents the error of the filtering, that is applied in (6.4g). If there is no filtering, then we have $F(H) = 0$ and in the case of the mean value filter, we get $F(H) = H$, because the averaging is a first order approximation to each element.

Since the system is supposed to be free of algebraic loops, we can then plug the Eqs. (6.8a) and (6.8b) into each other repeatedly and end up with

$$\eta^M_{i+1} + \eta^S_{i+1} = \mathcal{O}(1)\sum_{j=0}^{m(q+1)-1}(\epsilon^S_{i-j} + \epsilon^M_{i-j}) + \mathcal{O}\big(H^{q+1} + H_S^{r+1} + F(H)\big) \tag{6.9}$$

for $B \in \{S, M\}$ and an $m > 0$ by further estimation [1]. Now it follows

$$\epsilon^M_{i+1} + \epsilon^S_{i+1} = \big(1 + \mathcal{O}(H)\big)(\epsilon^M_i + \epsilon^S_i) + \mathcal{O}(H)\sum_{j=1}^{m\max\{p,q+1\}}(\eta^S_{k+1-j} + \eta^M_{k+1-j}) + \mathcal{O}\big(H^{p+2} + H^{q+2} + H\cdot H_S^{r+1} + H\cdot F(H)\big). \tag{6.10}$$

The term $\mathcal{O}(H)(\epsilon^M_{k+1} + \epsilon^S_{k+1})$ that would appear on the right-hand side can be brought to the left side. Dividing by $1 - \mathcal{O}(H) = \mathcal{O}(1)$ leads to (6.10). This analysis roughly followed the convergence analysis for linear multistep methods in the DAE case [8].

Now, just like in [1], it follows

$$\epsilon^M_{i+1} + \epsilon^S_{i+1}, \eta^M_{i+1} + \eta^S_{i+1} \in \mathcal{O}\big(H^{p+1} + H^{q+1} + H_S^{r+1} + F(H)\big).$$

This means that the extended algorithm is of order $\min\{p, q, r\} + 1$. In the case of interpolated extrapolation, the same argument holds and we get convergence of order $\min\{p, 1, r\} + 1$. If the filtering is applied, we get a convergence of first order.

The asymptotic analysis for $H \to 0$ is not insightful in the case of a highly oscillating slave system, where a filtering would be applied. Here, a convergence analysis in the frequency range would be more appropriate.

6.6 Applications

In the following, we will present applications of the SNiMoWrapper. Firstly, we have conducted numerical tests on an academic example in order to support the theoretical results of Sect. 6.5. Then, as a proof of concept we discuss two real-world problems in Sect. 6.6.2.

6.6.1 Academic Example

The test example is a quarter car model [12], see also [3] . The chassis and wheel are represented by two point masses m_M and m_S and can only move in the vertical direction. The chassis is connected to the wheel by a damper and a spring with coefficients d_M and k_M, while the wheel is connected to the ground by a very stiff spring with stiffness k_S. The master subsystem only consists of the chassis, while the slave subsystem consists of the wheel and the ground. We apply a force-displacement coupling. This means that the master subsystem outputs the height of the chassis and the slave subsystem outputs the force that is applied by the spring and damper that connect wheel and chassis. The situation is depicted in Fig. 6.4. The input of the master is, as above, the output of the slave and vice versa. In order to model a moving quarter car, the height of the ground $h(t)$ varies in time. For this, we have used two options:

$$\text{smooth profile:} \quad h(t) = \begin{cases} 0.1\,\text{m} \cdot \exp\left(\frac{1}{(t-2)^2-1}\right), & t \in (1\,\text{s}, 3\,\text{s}), \\ 0\,\text{m}, & \text{else}, \end{cases}$$

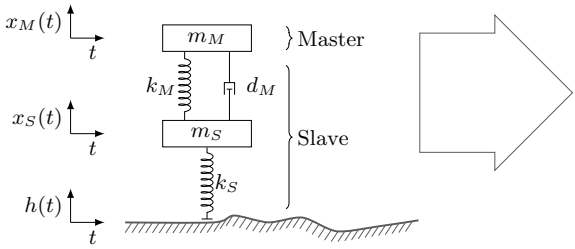
$$\text{step profile:} \quad h(t) = \begin{cases} 0.04\,\text{m}, & t < 4\,\text{s}, \\ 0\,\text{m}, & \text{else}, \end{cases}$$

see Fig. 6.5. The model parameters are given by [10]:

$$m_M = 256\,\text{kg}, \qquad k_M = 2020\,\text{N/m}, \qquad d_M = 1140\,\text{Ns/m},$$
$$m_S = 31\,\text{kg}, \qquad k_S = 128\,\text{kN/m}.$$

The master subsystem was implemented in Fortran. For the time integration of the master subsystem we used DASSL [11], a popular Fortran implementation of the BDF multistep time integration method. The slave subsystem was implemented in an FMU using plain C. Here, the time integration method is DOPRI5 [7], an implementation of the Runge–Kutta time integration method of Dormand and Prince with fifth order.

In the following, we have conducted numerical experiments in order to verify the theoretical analysis and to fvillustrate the effects of some of the co-simulation options. All tests have been performed with tight absolute and relative tolerances of



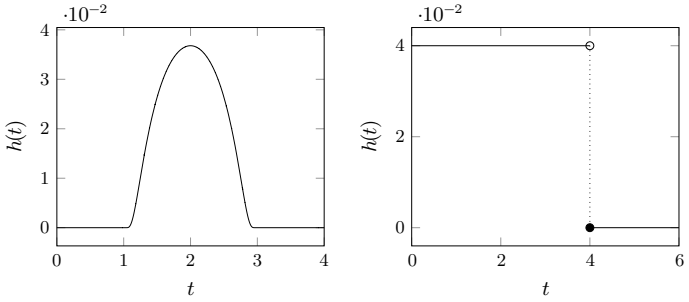


Fig. 6.4 Quarter car model

Fig. 6.5 Plot of the smooth road profile (left) and of the step road profile (right)

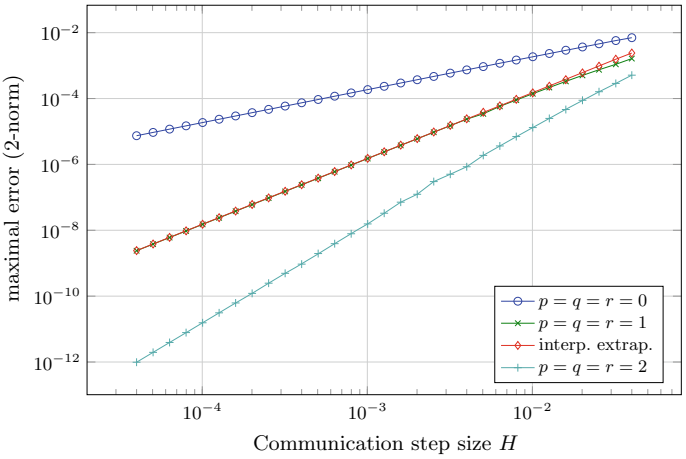


Fig. 6.6 Error plot for Test 1 (basic algorithm)

10^{-8} in the master system and 10^{-9} in the slave system. We have used a reference solution that was obtained by solving the monolithic system with MATLAB's Adams-Bashforth-Moulton PECE solver `ode113` of order up to 13 with very tight absolute and relative tolerances of 2.2×10^{-14}.

Test 1 In this test, we compared the results of the basic algorithm with the smooth road profile for equal inter- and extrapolation order $p = q$ of different magnitude. We

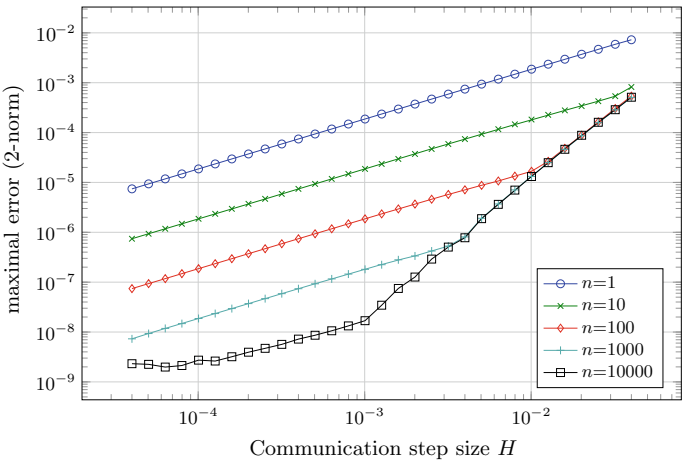


Fig. 6.7 Error plot for Test 2 (extended algorithm)

also assumed that the FMU is able to take inputs of sufficient order, thus $p = q = r$. Furthermore, we have also compared this to the case of linear interpolated extrapolation [5] and $p = r = 2$. The integration was done over $t \in [0, 4]$. The maximum of the errors in the 2-norm are plotted over the communication step size in Fig. 6.6. For polynomial extrapolation, we observe numerical convergence of order $p + 1$ for $p = q = r$. For interpolated extrapolation, we only get second order. This is consistent with the theoretical investigations in Sect. 6.5.

Test 2 Here, we want to show the effect of the number of substeps on the accuracy in the extended algorithm. We used polynomial inter- and extrapolation of second order $p = q = 2$ and assumed that the FMU only accepts piecewise constant inputs $r = 0$. Using the smooth road profile and the same tolerances and integration time span as above, we compared the results for varying amounts of substeps $n = 1, 10, \ldots, 10000$. The resulting errors are, in the same fashion as before, depicted in Fig. 6.7.

The theory only gives us convergence of order $\mathcal{O}(H^3) + \mathcal{O}(H_S)$ and we can numerically observe the first order term in all cases for sufficiently small communication step sizes H. On the other hand we see, that an increase in the number of substeps decreases the term $\mathcal{O}(H_S)$ significantly. If the number of substeps is sufficiently large in relation to the communication step size, the overall result behaves as if the slave system would be able to take nonconstant inputs. What we see is that the resulting error is, for coarser communication step sizes bounded from below by the error we would get for $r > 0$, which is at most of third order, because $p = q = 2$.

Test 3 In this test, we wanted to check how the co-simulation algorithm behaves, when there are discontinuities in one of the models. Note that the theoretical investigations act on the assumption that all occurring functions are sufficiently smooth. We have used the same tolerances for solving the master and slave subsystems as above, but now the discontinuous step road profile was chosen and the integration time span was

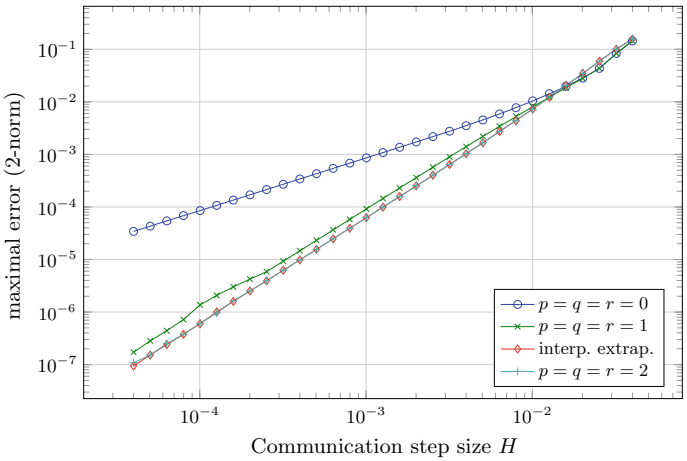


Fig. 6.8 Error plot for test 3

$t \in [0, 6]$. As in Test 1, we compared the results for $p = q = r$ for different values, and included interpolated extrapolation as well. Furthermore, we used the extended algorithm with $n = 10$ substeps for the test. The results are depicted in Fig. 6.8. We can see that the best convergence behaviour that can be numerically observed is quadratic. For $p = q = r = 0$, as before, only first order can be reached. Higher order inter- and extrapolation allows only for a slightly smaller error constant. Here, the interpolated extrapolation performs similarly as the polynomial extrapolation. This is mainly due to the fact that both algorithms use polynomial interpolation of second order to calculate the inputs for the master subsystem.

Verification with industrial tool We have verified these results by implementing a 3D version of the quarter car model with the industrial multi-body system tool Simpack. In Simpack, we created a mass and used a "user-force-element" in order to apply user-defined forces to the mass. The user force element is a Fortran code that can be programmed by the Simpack user. We used it to implement a Simpack API for the SNiMoWrapper and were able to simulate the system, where the SNiMoWrapper conducted the co-simulation of the slave system inside the FMU. We obtained similar results to the implementation above.

6.6.2 Industrial Application: Proof of Concept

The SNiMoWrapper was designed as testbed for numerical algorithms in academic as well as industrial co-simulation scenarios. We have used it in a proof-of-concept application as a way to include an FMU into a simulation tool.

During the SNiMoRed project we worked with a Simulink model of a front axle from an industrial partner. The model did not include a proper tire model, however.

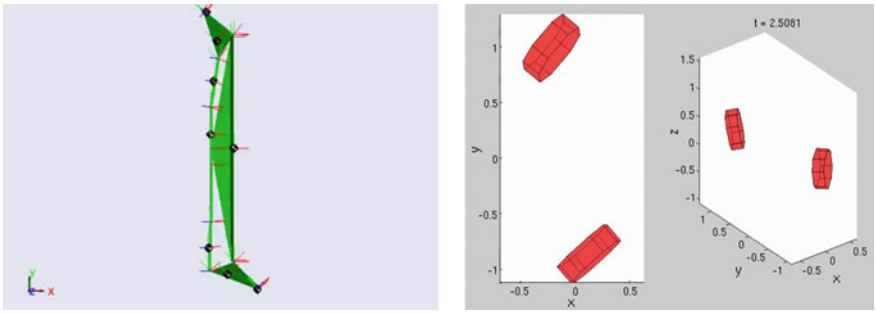


Fig. 6.9 Screenshots of the visual representation of the front axle model in Simulink (left) and of two of the commercial wheel models (right)

We wanted to combine a commercial tire model with the front axle model. In order to accomplish this, we have programmed an FMU that includes the commercial tire model by using its API. We have used the custom Simulink block which loads the SNiMoWrapper as shared library, see Sect. 6.4. With this configuration, we were able to perform a steering capacity test, where real-life data for the steering maneuver was used. The test consisted of fully steering to one side and then fully steering to the other side. The tire model worked properly and we were able to obtain meaningful data from this numerical experiment. Screenshots of the test are depicted in Fig. 6.9. Note that here we used simultaneously two instances of the same tire model FMU.

6.7 Conclusions

We have implemented an FMI v2.0 compatible tool that allows for easy incorporation of virtually any co-simulation FMU in standard simulation tools of nonlinear system dynamics. The co-simulation algorithm is contained entirely inside the SNiMoWrapper and thus hides the details of the co-simulation from the master simulation tool, essentially transforming the FMU into a function of time. For this to work, we only require mild prerequisites, such as that the integrator of the master simulation tool must step on each communication point before advancing to the next communication step. The co-simulation algorithm is of higher order and has some parameters which can easily be fine-tuned. The algorithm is of Gauss-Seidel type and incorporates the idea of substepping in order to deal with FMUs, that do not allow higher order signal extrapolation, and the idea of filtering the outputs of the slave system in order to deal with slave systems that are highly oscillating. We have given a proof of convergence that extends the proof given in [1].

Moreover, we have shown that the analytically predicted order of the algorithm can be observed numerically as well. For this, we have used the academic example of the quarter car model. Furthermore, we have shown an industrial proof-of-concept

example that goes beyond the world of academic examples. This shows that the SNiMoWrapper is applicable for real-life industrial problems as well.

Acknowledgements This work was supported by BMBF grant 05M10NHA [SNiMoRed] "Multidisciplinary simulation, nonlinear model reduction and pro-active control in vehicle dynamics." The authors are grateful to the MDF team at Fraunhofer ITWM for providing the industrial test case "steering capacity" (Sect. 6.6.2) and for fruitful discussions on the SNiMoWrapper.

References

1. Arnold, M., Clauß, C., Schierz, T.: Error analysis and error estimates for co-simulation in FMI for model exchange and co-simulation v2.0. Arch. Mech. Eng. **LX**, 75–94 (2013). https://doi.org/10.2478/meceng-2013-0005
2. Arnold, M., Günther, M.: Preconditioned dynamic iteration for coupled differential-algebraic systems. BIT **41**, 1–25 (2001). https://doi.org/10.1023/A:1021909032551
3. Arnold, M., Hante, S., Köbis, M.: Error analysis for co-simulation with force-displacement coupling. Proc. Angew. Math. Mech. **14**, 43–44 (2014). https://doi.org/10.1002/pamm.201410014
4. Busch, M.: Continuous approximation techniques for co-simulation methods: analysis of numerical stability and local error. ZAMM - J. Appl. Math. Mech./Zeitschrift für Angewandte Mathematik und Mechanik **96**(9), 1061–1081 (2016). https://doi.org/10.1002/zamm.201500196
5. Dronka, S., Rauh, J.: Co-simulation-interface for user-force-elements. In: Proceedings of SIMPACK User Meeting, Baden-Baden (2006)
6. FMI: The Functional Mock-up Interface. http://fmi-standard.org/
7. Hairer, E., Nørsett, S.P., Wanner, G.: Solving Ordinary Differential Equations I, Nonstiff Problems, 2nd edn. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-78862-1
8. Hairer, E., Wanner, G.: Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems, 2nd edn. Springer, Berlin (1996). https://doi.org/10.1007/978-3-642-05221-7
9. Kübler, R., Schiehlen, W.: Two methods of simulator coupling. Math. Comput. Model. Dyn. **6**, 93–113 (2000). https://doi.org/10.1076/1387-3954(200006)6:2;1-M;FT093
10. Mitschke, M., Wallentowitz, H.: Dynamik der Kraftfahrzeuge, 4th edn. Springer, Berlin (2004). https://doi.org/10.1007/978-3-662-06802-1
11. Petzold, L.R.: A Description of DASSL: a differential/algebraic system solver. In: Proceedings of IMACS World Congress (1982)
12. Popp, K., Schiehlen, W.: Ground Vehicle Dynamics. Springer, Berlin (2010). https://doi.org/10.1007/978-3-540-68553-1
13. Walter, W.: Ordinary Differential Equations. Springer, Berlin (1998). https://doi.org/10.1007/978-1-4612-0601-9