Framework for Simulation of Coupled Systems by Aggregation

Pukashawar Pannu

Master's thesis
2015:E49

LUND UNIVERSITY

Faculty of Engineering
Centre for Mathematical Sciences
Numerical Analysis

Abstract

Complex systems can be described by coupling several standalone ODE problems that communicate with input and output signals. The need for simulating such systems has increased in recent years. A major issue when simulating coupled ODE systems has been to communicate coupling relations properly throughout integration and how to handle discontinuities. In this paper a concept that aggregates several ODEs into a single problem is presented. For each right-hand-side function evaluation the aggregated problem communicates coupling relations ensuring that all inputs and outputs in the system are uptodate. Experiments are conducted on systems containing algebraic loops, discontinuities and non-linear couplings; the results suggest potential for the concept.

Acknowledgements

I would like to extend my sincerest gratitude to my supervisors Christian Andersson and Johan Åkesson at Modelon AB and Prof. Claus Führer at LTH for giving me this opportunity, believing in my success and their constant support throughout the project. They have given many ideas on concept formulation, implementation design and suitable experiments. Without you guys this thesis would not be possible. I also want to thank my colleagues at Modelon AB for creating a friendly atmosphere, helping me with technical issues and hosting interesting discussions. Surprise appearances of cakes and sweets on Friday afternoons have been especially appreciated. Finally, I want to thank my family and friends for supporting me in this undertaking.

Contents

1 Introduction

A concept for solving coupled dynamical systems described as ordinary differential equations is presented in this paper. The dynamical systems considered can be described by,

$$\dot{\bar{x}} = \bar{f}(\bar{x}, \bar{u}) \tag{1a}$$

$$\bar{y} = \bar{g}(\bar{x}, \bar{u}) \tag{1b}$$

where $\bar{x}$ are states, $\bar{y}$ outputs and $\bar{u}$ inputs. By coupling several standalone systems like Eq (1) one can model large complex systems. Coupling also makes it easier to swap specific components of a model. The major issues when it comes to simulating coupled systems are to maintain coupling relations throughout simulation and what to do about sub-system discontinuities. This paper proposes a concept of aggregation where all coupled systems are aggregated into one unified problem that handles coupling communication and sub-system discontinuities. The concept is implemented in Python and tested on a range of coupled systems.

Models used in experiments are either composed directly in Python or imported as Functional Mock-up Units (FMUs) following the Functional Mock-up Interface (FMI) [3]. FMI is an industrial standard for model exchange between different simulation tools. Usually a system is modelled in one tool that compiles the model to an FMU that can then be imported to a second tool. With FMI the same model can be tested in several different simulation environments. FMI supports two types of FMUs, Model Exchange (ME) and Co-Simulation (CS). Both types contain a model description and system equations as binaries. The difference is that in order to simulate an ME FMU the user is required to supply a numeric solver whereas the CS FMU comes with a built-in solver. However, in a CS FMU the user has no access to model equations and is forced to use the built in solver while ME FMUs gives the user access to model equations. In this paper only models compiled as ME 2.0 FMUs, 2.0 is an updated version of the standard with additional features, or written directly in Python are considered. FMU models in the experiments are written in Modelica [9], a language for modelling physical systems, and compiled with either JModelica.org [11] or Dymola [4].

There are many industrial solvers for integrating systems of ordinary differential equations. In this paper solvers available either in the open source Python package Assimulo [2] or in Dymola are used. FMUs are loaded into Python with PyFMI [10], an open source Python package for FMUs fully compliant with FMI. The implemented software intends to solve coupled systems containing coupling induced algebraic loops and sub-system events. Additionally a feature for externally adding events to a coupled system has been added.

A conference article presenting the concept discussed in this paper has been published in *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015* [7] with co-authors Christian Andersson, Johan Åkesson and Claus Führer. The article has been attached at the end of this document.

1.1 Background

Methods for integrating coupled systems modelled as ME FMUs today are quite limited. One way is to rewrite the coupled system as a monolithic model and integrate it using a standard solver. A monolithic system is when the entire model is represented by a single unit. This requires access to model source code which may be unavailable since the model dynamics are compiled to binaries when exporting a model as an FMU. It also defeats the purpose of first creating sub-systems that are then coupled together to model a dynamic system. Another way is to attach a solver to each ME sub-model and treat the entire system as a system of coupled CS FMUs that is simulated by applying a master algorithm [1]. Problems with this approach are to maintain proper coupling relations throughout the simulation as well as detecting and handling sub-system discontinuities.

As an alternative this paper presents a concept of aggregation that tries to patch the current issues of simulating coupled ME systems . The concept focuses on maintaining coupling relations at each right-hand-side function evaluation and tries to catch sub-system discontinuities. In essence, a monolithic representation of the coupled system is created without access to source code.

1.2 Reading Guidance

The paper starts with a mathematical description of the aggregation concept, followed by implementation of classes with function names written in **bold** and class names in `TypeWriter`. Next is a chapter with experiments that test a wide range of coupled systems. Source codes for all models except Formula 1 Race Car are available in the appendix. All variables mentioned in text are in *cursive.*

2 Aggregation Concept

The idea is to take N coupled systems and aggregate all system states, inputs, outputs and coupling relations to one aggregated system. The main goal of the aggregated system is to maintain sub-system coupling relations at each function evaluation. The first step is to aggregate all states, inputs and outputs of all sub-models to aggregated states, inputs and outputs vectors. Sub-model inputs are not limited to couplings, some sub-models may have external forces acting on the sub-system. To accommodate for this and at the same time keep track of which inputs are coupling related and which are time dependent external excitations, the inputs have been separated into, u, coupling related and, w, external excitations,

$$\bar{u} = [\hat{u}, \hat{w}] \tag{2}$$

The vectors are aggregated as,

$$x = \begin{bmatrix} \bar{x}_1^{[1]} \\ \vdots \\ \bar{x}_p^{[N]} \end{bmatrix}, \qquad y = \begin{bmatrix} \bar{y}_1^{[1]} \\ \vdots \\ \bar{y}_p^{[N]} \end{bmatrix}, \qquad u = \begin{bmatrix} \hat{u}_1^{[1]} \\ \vdots \\ \hat{u}_p^{[N]} \end{bmatrix}, \qquad w = \begin{bmatrix} \hat{w}_1^{[1]} \\ \vdots \\ \hat{w}_p^{[N]} \end{bmatrix} \tag{3}$$

where the super-script denotes model number, sub-script represent variable index for that sub-model and p represents the final vector index of each sub-system. The next step is to to aggregate the dynamics of the sub-systems, i.e. the state and output functions together with coupling relations. Let us take a look at a generic sub-system,

$$\dot{\bar{x}}^{[n]} = \bar{f}^{[n]}(\bar{x}^{[n]}, \hat{u}^{[n]}, \hat{w}^{[n]}) \tag{4a}$$

$$\bar{y}^{[n]} = \bar{g}^{[n]}(\bar{x}^{[n]}, \hat{u}^{[n]}, \hat{w}^{[n]}) \tag{4b}$$

where $n \in [1, N]$ of N coupled systems. The barred variables and functions belong to sub-systems. Functions f and g hold the state and output dynamics. Aggregating the sub-system functions and adding sub-model coupling relations results in,

$$\dot{x} = f(x, u, w) = \begin{bmatrix} \bar{f}_1^{[1]}(\bar{x}_1^{[1]}, \hat{u}_1^{[1]}, \hat{w}_1^{[1]}) \\ \vdots \\ \bar{f}_j^{[N]}(\bar{x}_j^{[N]}, \hat{u}_j^{[N]}, \hat{w}_j^{[N]}) \end{bmatrix} \tag{5a}$$

$$y = g(x, u, w) = \begin{bmatrix} \bar{g}_1^{[1]}(\bar{x}_1^{[1]}, \hat{u}_1^{[1]}, \hat{w}_1^{[1]}) \\ \vdots \\ \bar{g}_j^{[N]}(\bar{x}_j^{[N]}, \hat{u}_j^{[N]}, \hat{w}_j^{[N]}) \end{bmatrix} \tag{5b}$$

$$u = c(y, w) = \begin{bmatrix} \bar{c}_1^{[1]}(y, w) \\ \vdots \\ \bar{c}_j^{[N]}(y, w) \end{bmatrix} \tag{5c}$$

A more compact form of the aggregated system is,

$$\dot{x} = f(x, u, w) \tag{6a}$$

$$y = g(x, u, w) \tag{6b}$$

$$u = c(y, w) \tag{6c}$$

The aggregated system looks just like a sub-system with the addition of coupling relations. Depending on sub-model couplings the problem of algebraic loops may emerge that may in some cases be unsolvable, see Section 2.1. With aggregation one must also consider how discontinuities of all sub-systems are handled. The aggregation structure can be exploited to calculate a coupled Jacobian for use with implicit solvers or other tools requiring the Jacobian.

When analysing conditions for algebraic loops and making Jacobian calculations a linearisation of the coupled system is considered. A state-space linearized representation of Eq (6) is,

$$\dot{x} = Ax + Bu + \hat{B}w \tag{7a}$$

$$y = Cx + Du + \hat{D}w \tag{7b}$$

$$u = Ly + \hat{L}w \tag{7c}$$

where A, B, $\hat{B}$, C, D and $\hat{D}$ are matrices of the aggregated system that are block diagonal, where each block corresponds to a matching sub-system matrix. The two remaining matrices L and $\hat{L}$ come from couplings and may have any form. Issues regarding algebraic loops and Jacobian are discussed in the following sections.

2.1 Algebraic Loops

Coupling systems can at times give additional mathematical and computational problems. One problem is the introduction of algebraic loops. The loops come from feed-through where the input of a system is fed directly to an output. Coupling together systems with feed-through terms can eventually result in an output of a model coming back as an input to itself. Figure 1 displays a coupled system with a feedback loop where output $y^{[A]}$ circles back as an input to system A. To properly solve such systems one must solve an algebraic loop problem for each right-hand-side function evaluation.

Mathematically the algebraic loops are introduced in Eq (6c). To make this more evident the inputs u are eliminated by inserting Eq (6c) into Eq (6b),

$$y = g(x, c(y, w), w) \tag{8}$$

where the outputs y are present on both sides of the equation. To solve the algebraic loop is to find values of y that satisfy Eq (8). This can be done by rewriting Eq (8) to a root-finding problem,

$$y - g(x, c(y, w), w) = 0 \tag{9}$$

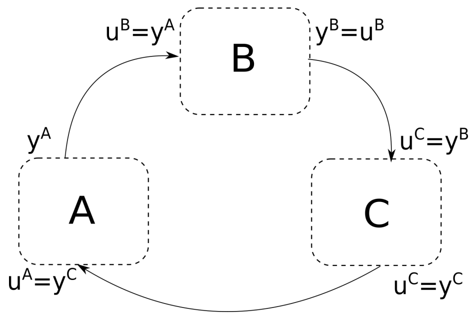

Figure 1: Coupling induced algebraic loops. Output from system A loops around to itself. Variables y represent the outputs and u the inputs.

Eq (9) can be solved by applying a root-finding scheme such as Newtons method or Fixed-Point iteration. The method implemented in the software uses a variation of Newtons method, see Section 3.4.3.

All algebraic loops cannot be solved. To find a sufficient condition for solving coupling induced algebraic loops we look at a linearised coupled system, represented in Eq (7). Inputs u are eliminated by inserting Eq (7c) into Eq (7b) giving,

$$y = Cx + DLy + D\hat{L}w + \hat{D}w \tag{10}$$

Solving for outputs y yields,

$$y = (I - DL)^{-1}(Cx + D\hat{L}w + \hat{D}w) \tag{11}$$

Solution to Eq (11) can only be found if $(I - DL)$ is non-singular. This puts a sufficient condition on coupled systems with feed-through terms. Systems that do not satisfy the condition cannot be solved by aggregation. For such systems a model redesign or a different simulation approach should be considered.

2.2 Coupled Jacobian

To efficiently integrate a system or to make use of an implicit ODE solver the system Jacobian needs to be calculated. The Jacobian is a matrix of all first-order derivatives of a system with respect to state variables, x_i. For a sub-system, all inputs $u_i^{[n]}$, are seen as time dependent and the sub-system Jacobian is given as, $J^{[n]} = \frac{\partial \bar{f}}{\partial x}$, which in matrix form becomes, $J^{[n]} = A^{[n]}$. Looking at the Jacobian of the full aggregated system, a naive approach would be to simply aggregate all sub-system Jacobians. Let us investigate the aggregated Jacobian by applying the $\frac{\partial}{\partial x}$ operator on Eq (6a)

$$J = \frac{\partial}{\partial x} f(x, u, w) = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial f}{\partial w}\frac{\partial w}{\partial x} \tag{12}$$

In the sub-system case all inputs were seen as time dependent and were disregarded. For the aggregated system only w-terms are independent of states and thus zero. The coupling equations impact the system dynamics, as evidenced by the $\frac{\partial f}{\partial u}\frac{\partial u}{\partial x}$ term in Eq (12). The coupled Jacobian cannot be constructed by simply aggregating sub-system Jacobian matrices diagonally. Further investigation is required.

Starting from Eq (6) the w-terms are removed since they have no impact on the Jacobian. Eliminating u by inserting Eq (6c) into Eq (6a) and Eq (6b) simplifies the system description,

$$\dot{x} = f(x, c(y)) \tag{13a}$$

$$y = g(x, c(y)) \tag{13b}$$

The Jacobian can be written,

$$J = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial y}\frac{\partial y}{\partial x} \tag{14}$$

The $\frac{\partial y}{\partial x}$ contribution is found by differentiating the outputs Eq (13b) with respect to x,

$$\frac{\partial y}{\partial x} = \frac{\partial g}{\partial x} + \frac{\partial g}{\partial c}\frac{\partial c}{\partial y}\frac{\partial y}{\partial x} \tag{15}$$

Solving for $\frac{\partial y}{\partial x}$ gives,

$$\frac{\partial y}{\partial x} = \left(I - \frac{\partial g}{\partial c}\frac{\partial c}{\partial y}\right)^{-1}\frac{\partial g}{\partial x} \tag{16}$$

Finally inserting into Eq (12) gives the Jacobian,

$$J = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial y}\left(I - \frac{\partial g}{\partial c}\frac{\partial c}{\partial y}\right)^{-1}\frac{\partial g}{\partial x} \tag{17}$$

When computing the Jacobian the state space representation in Eq (7) is used,

$$J = A + BL(I - DL)^{-1}C \tag{18}$$

where A, B, C and D are aggregated block diagonal matrices and L the coupling matrix.

3 Implementation

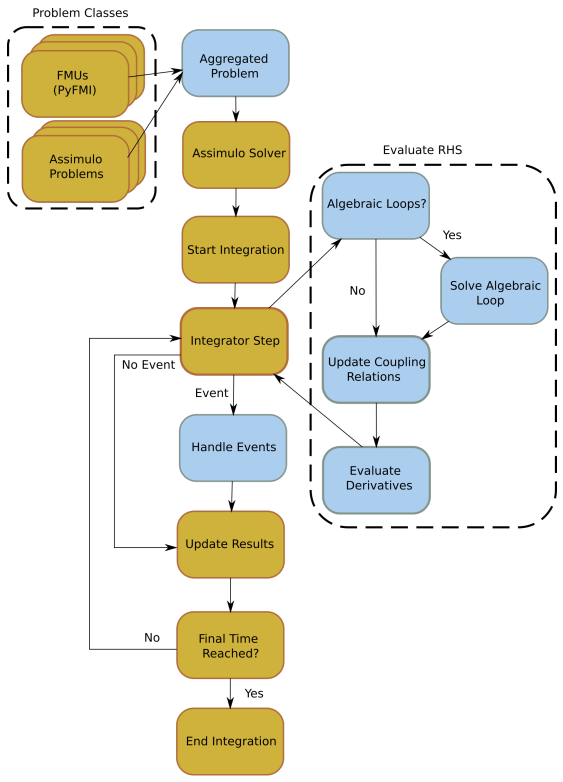

Figure 2: Simulations flow of FMU models loaded through PyFMI coupled with Assimulo Problems in `AggregatedProblem`. The blue colour indicates nodes that are added or modified due to aggregation.

The aggregated problem is intended to be simulated using Assimulo solvers. Assimulo is an open source Python package for solving ordinary differential equations. One must consider what steps need to be modified or added to the Assimulos problem solving flow to support aggregated systems. Figure 2 shows the problem solving steps using an Assimulo solver with added modifications to support aggregated systems. The boxes coloured in blue are steps specific for an aggregated system.

First step is to aggregate systems modelled as either FMUs or directly in Python as Assimulo problems. The two model types have their own structure and a different set of functions. To simplify coupling communication the two problem classes are extended with an interface that defines a common set of functions for couplings called `CouplingExtensionInterface`, see Section 3.2. The two new classes, `FMIODE2InputOutput` for FMUs and `Explicit_Problem_Model` for models written in Python can be coupled and used with the class AggregatedProblem. The class hierarchy is shown in Figure 3. The dashed line from the interface to AggregatedProblem indicates that the aggregated problem class operates on instances of classes inheriting the interface.

Following the flowchart the next interesting point is the integrator step. The purpose of this step is to evaluate the right-hand-side of an ODE and return the results to the solver. For the aggregated system this means to evaluate the right-hand-side of all sub-systems. Before this can be done the coupling relations must be updated, see Section 3.4.1, which may include a step of solving an algebraic loop problem, discussed in Section 3.4.3.

The flowchart then leads to events. AggregatedProblem must look at all sub-systems, if an event has occurred it must be handled. Support for externally adding time and state events to an aggregated problem has also been added, see Section 3.4.4. Looking at the flowchart no more modifications are needed to solve an aggregated problem with Assimulo, but to help solvers, the Jacobian of the coupled system is computed by the problem class, see Section 3.4.2.

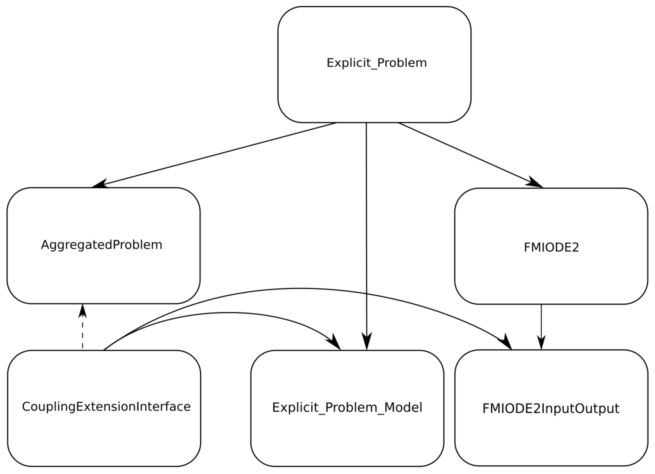

Figure 3: Class hierarchy where problem classes inherit from Explicit_Problem. FMIODE2 takes an FMU and translates it into an Assimulo problem through the PyFMI interface. Problem classes with inputs and outputs also inherit from the coupling extension interface. The dashed line indicates that AggregatedProblem operates on CouplingExtensionInterface instances.

3.1 Software Background

Before making any implementations an example of how to solve an ODE system with Assimulo is presented. For demonstration purposes the bouncing ball system in the Assimulo documentation is used. The system is modelled both as an Assimulo problem directly in Python and as an FMU using the Modelica language.

A bouncing ball system can be described as,

$$\dot{x_1} = x_2 \tag{19a}$$
$$\dot{x_2} = -g \tag{19b}$$

where x_1 is the ball's height above ground in metres, x_2 the ball's velocity and g the gravitational acceleration. When the ball hits the ground a state dependent discontinuity occurs. At this point the velocity variable changes sign and the system loses energy if the impact is inelastic. The energy loss is modelled with an elasticity constant that reduces the balls velocity. In Assimulo solvers discontinuities can be handled as events [5]. Events dependent on state variables are modelled as zero-crossings called event indicators. Each time the event indicator changes sign the solver stops and calls a handle event method. For this demonstration the event indicator can be modelled as,

$$event = x_1 \tag{20}$$

where each time the height variable changes sign it is seen as an impact on the ground. The initial conditions are,

$$x_1 = 1 \tag{21a}$$
$$x_2 = 0 \tag{21b}$$
$$e = 0.8 \tag{21c}$$

where e is the elasticity constant. For more information regarding events and Assimulo consult the Assimulo documentation.

3.1.1 Bouncing Ball as Assimulo Problem

The bouncing ball system described in Eq (19) is here written as an Assimulo problem. A problem class called `Explicit_Problem` in the Assimulo package is suitable for modelling ODEs and is used as demonstration. The problem class requires the user to define a right-hand-side function of the problem, the relations described in Eq (19), initial conditions and for this specific problem event indicators for the impact to the ground and a handle event function. The **rhs** is easily modelled using `numpy`, a class for numerical data types in Python, as

```
def rhs(t, x, sw):
   g = 9.82
   return np.array([x[0], -g])
```

where *np* is used as an alias for the numpy class and x is a numpy array with the first element representing height of the ball e.g. x_1 and the second representing the velocity, x_2. An additional argument has been provided to the function called *sw*, it stands for switches and should contain a list of boolean values that can be used to activate and deactivate certain model behaviour. Switches will be used with the event handling.

Initial conditions are provided as,

```
   x0 = np.array([1.0, 0.0])
   t0 = 0.0
```

where the first element of *x0* corresponds to the height of the ball and the second element the velocity.

Events can be modelled in different ways, this demonstration makes use of switches. The reason for using switches is to prevent what is knows as chattering. In short, chattering occurs when an event is retriggered a second time shortly after the first. The problem comes from how events are detected and the fact that the system needs time to cope with model changes after event handling. For more in-depth explanation consult the Assimulo documentation as chattering is not within the scope of this paper. The event indicator function is written as,

```
def state_events(t, y, sw):
   """
   State event function. Defines zero-crossings.
   """
   event1 = y[0] if sw[0] else 5 # y[0] if ball is falling.
   event2 = y[1] if sw[1] else 5 # y[1] if ball is going up.
   return np.array([event1, event2])
```

where the first event, $event1$, monitors for when the ball impacts the ground and the second event is used to turn switches when the ball reaches maximum height. Switches prevent chattering since the event indicators are only active when the corresponding switch is active. The switches are, $sw[0]$ which is true when the ball is falling down and false otherwise and $sw[1]$, true if ball is moving upwards and false otherwise.

After the event has been detected it has to be handled. This is done by changing sign of the velocity, reducing energy with the elasticity constant and flipping switches,

```
def handle_event(solver, event_info):
   """
   Event handling function. Changes switches and
   velocity direction.
   """
   ev_info = event_info[0] # Only looking at state events.
   if ev_info[0] !=0:
      # Ball bounces up.
      solver.sw[0] = False
      solver.sw[1] = True
      solver.y[1] = -0.8*solver.y[1] # 0.8 elasticity
   else:
      # Ball is at top, starting to fall downwards.
      solver.sw[0] = True
      solver.sw[1] = False
```

With a complete system description an Assimulo problem can be created as,

```
model = Explicit_Problem(rhs, x0, t0)
model.state_events = state_events
model.handle_event = handle_event
```

where initial conditions and functions are the same as the above. The next step is to attach a solver and simulate the problem. A solver in the Assimulo package is `CVode`,

```
sim = CVode(model)

sim.simulate(tf)
```

where tf is the final time of the simulation. A full script of the demonstration is available in the appendix, see Section 6.4.1.

3.1.2 Bouncing Ball with PyFMI

In this section the bouncing ball system described in Eq (19) is modelled in Modelica and compiled to an FMU with JModelica.org. The system is then loaded into Python with PyFMI and simulated using the FMU-Assimulo interface, `FMIODE2`, and an Assimulo solver.

Modelica code for the bouncing ball system is,

```
model BouncingBall
    Real h(start = 1.0);
    Real v(start = 0.0);

    constant Real g = 9.81;
equation
    der(h) = v;
    der(v) = -g;

    when h < 0 then
        reinit(v, -0.8*v);
    end when;
end BouncingBall;
```

where the *when* statement acts as the event indicator and *reinit* as the event handler. Variables h and v are used for the balls height and velocity. For more in-depth explanation consult the Modelica language definition [9].

A compiled FMU is loaded into Python with PyFMI. This gives the user an FMU instance with standard FMU functions. In order to simulate an FMU with Assimulo an interface has to be used. For FMI 2.0 Model Exchange FMUs the interface FMIODE2 in PyFMI is used. When simulating an FMU with Assimulo a cutom result handler is commonly used. Compiling, loading and interfacing the bouncing ball system together with a custom result handler is done as,

```
# Compile model
name = compile_fmu('BouncingBall', 'model.mo', target='me',
    version='2.0')

# Load FMU
fmu = load_fmu(name)

# Result handler
opts = fmu.simulate_options()
res = ResultHandlerFile(fmu)
res.set_options(opts)

# Create Assimulo-FMU interface instance
mod = FMIODE2(fmu, result_handler=res)
```

where the result handler stores data in a file. For compiler options see JModelica.org documentation. Before an ME 2.0 FMU can be simulated it has to be initialized,

```
# Initialize FMU
fmu.setup_experiment()
fmu.initialize()
fmu.event_update()
fmu.enter_continuous_time_mode()
```

where the first two lines are needed to initialize the system. The last two lines are needed to setup event detection with FMU and Assimulo. Once the FMU has been initialized the model can be simulated as a standard Assimulo problem,

```
# Star result handler
res.simulation_start()

# Solver instance
sim = CVode(mod)

# Simulation
sim.simulate(tf)

# Stop result handler
res.simulation_end()
```

where the result handler is started and stopped appropriately. A script demonstrating the full process is available in the appendix, see Section 6.4.2.

3.2 Coupling Extension Interface for ODEs

Assimulo problems describing ODEs lack structure and functions for handling inputs and outputs. For FMUs, the structure is present but when loaded in Python with PyFMI's Assimulo interface, `FMIODE2`, the necessary functions for setting inputs and getting outputs are concealed. The two problem classes can be extended to include functions for handling inputs and outputs. For this purpose `CouplingExtensionInterface` has been developed that defines the set of functions given in Table 1.

Table 1: Functions defined by `CouplingExtensionInterface`. Arguments for state space matrix functions are described in text.

Functions	Description
get(var_list, t, x)	Get inputs/outputs.
set($var\_list, list\_values$)	Set inputs.
get_state_list()	Get a list of state variables.
get_input_list()	Get a list of input variables.
get_output_list()	Get a list of output variables.
get_state_space_matrix_A(...)	Get state space matrix A.
get_state_space_matrix_B(...)	Get state space matrix B.
get_state_space_matrix_C(...)	Get state space matrix C.
get_state_space_matrix_D(...)	Get state space matrix D.
get_output_dependencies()	Get output dependency.
get_derivatives_dependencies()	Get derivatives dependency.

Arguments for state space matrix functions are: t time, x states, sw switches, var a list of variables; default value is $None$, h approximation step size; default 10^{-10} and $limit\_functions$ a list of function names; default []. The var argument determines which variables to perturb, with it one can choose which variables should be part of the matrix. As an example consider matrix B in a model with seven states and seven inputs. Each column in matrix B corresponds to a specific input variable. If one is only interested in the B matrix for inputs u_1 and u_4 this can be chosen with var,

```
model.get_state_space_matrix_B(t0, x0, var=['u1', 'u4'])
```

where *t0* and *x0* are initial conditions. The function call above returns a matrix of size (7, 2) instead of a full (7,7) matrix. The $limit\_functions$ argument is similar as it can be used to limit which functions are part of the matrix. Consider now a model with seven states, inputs and outputs where one is interested in the D matrix for outputs y_1, y_4 and inputs u_1, u_3, u_6. Making the call,

```
model.get_state_space_matrix_C(t0, x0, var=['u1', 'u3', 'u6'],
    limit_functions=['y1', 'y4'])
```

results in a (2, 3) matrix instead of (7, 7). The default values for *var* and *limit_functions* are *None* and [] respectively. In those cases the full matrices are returned.

The interface also defines a set of attributes expected in a problem with inputs and outputs, shown in Table 2. Options defined by the interface are given in Table 3.

Table 2: Shows attribute defined by `CouplingExtensionInterface`.

Attributes	Description
_f_nbr	Number of state functions.
_u_nbr	Number of inputs.
_y_nbr	Number of outputs.

Table 3: Options defined by `CouplingExtensionInterface`.

Option	Description
has_outputs	True if system has outputs.
use_custom_result_handler	True if system uses custom result handler.

Note that the interface is intended as an extension to Assimulo like ODE problems. It is not enough to describe problems such as Eq (1) on its own.

3.3 Problem Classes for ODEs with Inputs and Outputs

In this section problem classes for describing Eq (1) are implemented by extending base classes with the coupling extension interface, as shown in Figure 3. Class `Explicit_Problem_Model` is for models written directly in Python, discussed in Section 3.3.1. The class for FMUs is `FMIODE2InputOutput`, discussed in Section 3.3.2.

3.3.1 Assimulo ODE Problem with Coupling Extension

The goal is to implement `Explicit_Problem_Model`, an Assimulo problem class for modelling systems like Eq (1). As base `Explicit_Problem` is used, a problem class for describing ODEs such as,

$$\dot{\bar{x}} = \bar{f}(\bar{x}) \tag{22}$$

where the right-hand-side is a function of states. By extending the base with the coupling extension interface, the desired problem class can be created.

As previously mentioned Assimulo problem classes lack structure for inputs and outputs. Before making structural changes one must consider how and when inputs and outputs are used. In Eq (1) inputs are used in the right-hand-side function. Evidently the **rhs** in the class must be altered to support inputs. One can additionally see that inputs are used in the outputs function. This means that the inputs must be accessible by at least two functions. From this it was decided to store inputs internally and represent them as a numpy array with each element representing an input term. The **rhs** is a user provided function and in order to support inputs the user is required to write it as,

```
def rhs(t, x, u=None, sw=None):
   return x
```

where t is time, x states, u inputs and sw switches. Parameter u for inputs must be present even if the system is without inputs. However, this solution raises an issue of compatibility with Assimulo solvers. They expect **rhs** to be a function of at most time, states and switches. Since the inputs are handled internally a dummy **rhs** can be created that is a function of time and states that updates inputs and then calls the user provided right-hand-side function. This allows the problem class to support existing Assimulo solvers. An example of creating an instance of `Explicit_Problem_Model` and making a call to **rhs** is demonstrated below,

```
def rhs(t, x, u=None, sw=None):
   return x

model = Explicit_Problem_Model(rhs, x0, t0)

model.rhs(t0, x0)
```

where $t0$ is initial time and $x0$ initial state. The example does not have any inputs and the right-hand-side is evaluated at initial time and states.

Inputs can be either time dependent external excitations, w, or come from coupling u. Since the time dependent function is known the two input types can be distinguished. In PyFMI there is a class `Trajectory` for handling time dependent inputs with FMUs. `Trajectory` is a class that interpolates values. It has two components, an abscissa that defines a time grid and an ordinate that is a data matrix where each column holds data for a specific input. A sine function can be modelled with a trajectory object as,

```
abscissa = np.linspace(0, 10, 20)
ordinate = np.vstack(np.sin(abscissa))
```

```
traj = TrajectoryLinearInterpolation(abscissa, ordinate)
```

where *np* is an alias for the numpy package, *abscissa* is a time-vector from $t_0 = 0$ to $t_f = 10$ with 20 data points, *ordinate* is the sine function and the trajectory has linear interpolation. The following demonstrates how to create a model with three inputs, one coming from coupling and two external excitations, by using the *inputs* argument,

```
# Right-Hand-Side function
def rhs(t, x, u, sw=None):
   return x + u

# Time dependent external excitations
abscissa = np.linspace(0, 10, 20)
ordinate = np.array(np.sin(abscissa),
    np.cos(abscissa)).transpose()

traj = TrajectoryLinearInterpolation(abscissa, ordinate)

# Initial Conditions
x0 = np.array([1., 2., 3.])
t0 = 0

# Model
model = Explicit_Problem_Model(rhs, x0, t0,
inputs=(3, [('u', 0), ('w1', 1), ('w2', 2)]),
         (['w1', 'w2'], traj))
```

The *inputs* argument is a tuple with three components. First an integer indicating number of inputs. The second is a list with tuples for naming input variables. In the example, input at index 0 is named u and inputs at indexes 1 and 2 are named w_1 and w_2 respectively. If an empty list is sent in, variables are named automatically starting from u_0 and increasing the sub index. The final entry is a tuple with the first component as a list of variable names that are external excitations and the trajectory as the second.

Output function **y** is user provided and must be a function of time, states, inputs and switches. It has the same structure as **rhs** and a model with outputs is created as,

```
def y(t, x, u=None, sw=None):
   return x

t0 = 0
x0 = np.array([1, 2])

model = Explicit_Problem_Model(rhs, x0, t0,
   outputs=(y, [('y1', 0), ('y2', 1)]))
```

where the *outputs* argument is used, rhs is the same as in the previous example. The *outputs* argument takes a tuple with the first part the actual output function **y** and the second a list for naming variables the same way as for inputs.

Variable names are stored in `OrderedDicts` and are accessible with get-list functions, one for each variable type,

```
model.get_states_list()
model.get_input_list()
model.get_output_list()
```

where *model* is the system in the example above. The dictionaries hold variable names and indexes to their corresponding position in data arrays. With a structure for inputs and a naming practice the two functions **get** and **set** can be implemented. The first is used for acquiring current input or output value by calling,

```
model.get(['u12'], t0, x0)
```

where u_{12} is the desired input. It takes either a single variable or a list of variable names and returns the results. The **set** function has two parameters, the first being a list of variable names and the second a list of corresponding values. It can be used to update coupling related inputs,

```
model.set('u12', 42)
```

here input variable u_{12} is set to 42. Both functions search the dictionaries to find indexes pointing them to the correct values in the data arrays.

Functions for getting state space matrices are implemented by approximation. The method used is forward differences that is generally stated as,

$$\phi'(x) \approx \frac{\phi(x+h) - \phi(x)}{h} \tag{23}$$

for a small h. Matrix A in Eq (7) is computed by perturbing states and computing Eq (23) where ϕ is replaced with the systems right-hand-side function. Perturbation of one state yields a column in the matrix. Repeating for all states, one at a time, all columns can be put together into a matrix. Matrix B is computed the same way by replacing states with inputs. For matrices C and D **rhs** is replaced with the output function **y**.

When constructing an instance of `Explicit_Problem_Model` dependency information can be provided with the *dependency_info* argument. The argument takes a tuple with state dependencies and output dependencies. State and output dependencies are split in two parts to distinguish dependencies on states or inputs. In total the information is stored in four `OrderedDicts`. Creation of a model with supplied derivatives and input dependencies is demonstrated,

```
# State dependencies on states
s_dep = OrderedDict()
```

```
s_dep['x'] = ['x']

# State dependencies on inputs
u_dep = OrderedDict()
u_dep['x'] = ['u']

# State dependencies
state_dep = (s_dep, u_dep)

# Output dependencies on states
s_dep = OrderedDict()
s_dep['y'] = []

# Output dependencies on inputs
u_dep = OrderedDict()
u_dep['y'] = ['u']

# Output dependencies
output_dep = (s_dep, u_dep)

# Model
model = Explicit_Problem_Model(rhs, x0, t0,
    dependency_info=(state_dep, output_dep))
```

where $t0$ and $x0$ are initial conditions and rhs a right-hand-side function. Functions for getting derivatives and output dependencies return tuples with `Dictionarys`. In case no dependency information is provided the functions return $None$.

A demonstration of creating a system with two states, two inputs, one as external excitation w and the other as coupling term u and two outputs is shown below,

```
def rhs(t, x, u, sw=None):
   return -x + u

def y(t, x, u, sw=None):
   return x

t0 = 0
x0 = np.array([9.0, 2.0]

abscissa = np.linspace(0, 10, 20)
ordinate = np.bstack(np.sin(abscissa))

traj = TrajectoryLinearInterpolation(abscissa, ordinate)

model = Explicit_Problem_Model(rhs, x0, t0,
   inputs=(2, [('u', 0), ('w', 1)]), (['w'], traj)),
   outputs=(y, [('y', 0)]))
```

```
model.rhs(t0, x0)
```

with the external excitation on w as a sine wave modelled with the `Trajectory` class.

3.3.2 PyFMI Problem with Coupling Extension

FMUs loaded with PyFMIs `FMIODE2` have a built in structure for inputs and outputs, however, some important functions are hidden. The focus of the implementation is to make functions for accessing inputs and outputs visible. Using `FMIODE2` as base, `FMIODE2InputOutput` can be created by extending the class with the coupling interface. As in the case of `Explicit_Problem_Model` a dummy **rhs** has to be created in order to support inputs and be compatible with Assimulo solvers. An instance of `FMIODE2InputOutput` is created as,

```
# Load FMU
fmu = load_fmu("Model.fmu")

# Setup and Initialize FMU
fmu.setup_experiment()
fmu.initialize()

# Setup result handler for FMU, in this case on file
res = ResultHandlerFile(fmu)
res.set_options(fmu.simulate_options())

# Create FMIODE2InputOutput Problem
mod = FMIODE2InputOutput(fmu, result_handler=res)
```

During initialization, `FMIODE2InputOutput` calls the base class initialization function. This is done in order make use of all initialization features in the base class. For a full list of constructor arguments consult the PyFMI documentation for `FMIODE2`.

Variable names are stored in `OrderedDicts` and are accessed with get-list functions that return entire dictionaries. Function for setting value is as expected. It takes two arguments, the first a list of or a single variable name and as the second argument a list or single value. A model's value can be updated as,

```
mod.set('u8', 9)
```

where variable u_8 is set to value 9.

Function for retrieving values follows the `CouplingExtensionInterface` where the arguments are a list of variable names, time, states and switches. A big difference with Assimulo problems and FMUs is that FMUs continuously store state data whereas Assimulo problems do not. When simulating a pure

Assimulo problem all state data is stored in the solver, not the sub-model. This is the reason for time and states being part of the **get** function in the coupling interface. Each time the get function is called the FMU model states are updated and the requested variables returned. This keeps the solver and the model on the same level. The function call for getting inputs of an `FMIODE2InputOutput` model becomes,

```
mod.get(['u1', 'u4', 'u9'], t0, x0)
```

where *t0* and *x0* are initial time and states. Inputs u_1, u_4 and u_9 are returned as a numpy array.

The state space matrix implementation tries to take advantage of FMI 2.0. In the standard there is an option to compile FMUs with directional derivatives. Using the option each column of a matrix can be computed with high precision. The columns are then combined to form a state space matrix. FMUs that are not compiled with directional derivatives approximate matrices the same way as in `Explicit_Problem_Model`, with a forward difference step on each variable.

Dependency information is read from the FMU. If it is unavailable dependency information variables are stored as *None*.

3.4 Aggregated Problem Class

This is a problem class for coupled systems that are modelled with the interface described in Section 3.2. The idea is to handle sub-model communication internally and integrate the problem class with Assimulo solvers like any standard Assimulo problem class. `AggregatedProblem` inherits from `Explicit_Problem` in Assimulo and extends the problem class for coupled systems, giving it the needed structure for Assimulo problems.

As input when instantiating the problem the user provides a `Dictionary` with user chosen keys and models that follow `CouplingExtensionInterface` as values together with coupling relations. The couplings can be provided as a list or a function, see Section 3.4.1. A coupled system can be formulated,

```
dic = OrderedDict()
dic['ModelA'] = modA
dic['ModelB'] = modB

couplings = [('ModelA', 'u1', 'ModelB', 'y2')]

agg_prob = AggregatedProblem(dic, couplings)
```

where *ModelA* and *ModelB* are string keys and *modA*, *modB* are sub-models following `CouplingExtensionInterface`. The couplings say that *ModelA*'s input variable u_1 gets its data from *ModelB*'s output variable y_2.

To correctly evaluate the right-hand-side of an aggregated problem the coupling relations must be updated and the results from all evaluated sub-model

rhs-functions be aggregated. This has been done by creating an **rhs** that first calls **update_couplings** and then evaluates and aggregates sub-model right-hand-side results. The function **update_couplings** is responsible for maintaining proper coupling relations. If algebraic loops are present **update_couplings** makes a call to solve them, see Section 3.4.3. With a correctly evaluating **rhs** it is enough for solvers like `CVode` to solve the coupled problem. `CVode` is an implicit solver that uses the system Jacobian for integration. It has built in functions for approximating the Jacobian but does allow users to supply a Jacobian function. To make computations more efficient a function that exploits the aggregated structure in order to compute the coupled system Jacobian has been implemented, see Section 3.4.2.

During initialization of an `AggregatedProblem` instance some data is collected. Total number of inputs and outputs are saved and depending on coupling type dictionaries containing data of coupling related inputs and outputs are stored. If variables part of coupling cannot be determined, all inputs and outputs are assumed to be part of coupling. These dictionaries are used when approximating the coupled Jacobian matrix. Another important aspect is to check couplings for dependency order and presence of algebraic loops. Depending on coupling type, discussed in Section 3.4.1, an attempt is made to find coupling execution order. In systems containing feed-through, proper execution order is vital. Consider the following example with models A, B, and C where model B has direct feed-through,

```
## Outputs of models in system.
# A: y = x
# B: y = x + u
# C: y = -

couplings = [('C', 'u', 'B', 'y'),
             ('B', 'u', 'A', 'y')]
```

here model B gets its input from model A and model C from model B. Rewriting the problem in mathematical terms yields,

$$u_A = 0 \tag{24a}$$
$$u_B = y_A \tag{24b}$$
$$u_C = y_B = x_B + u_B = x_B + y_A \tag{24c}$$

where u_i are inputs and y_j outputs. From Eq (24c) it becomes evident that the input to model C is dependent on model A. To get proper coupling relations the couplings must be executed,

```
A -> B
B -> C
```

This is determined by walking through couplings and using the models dependency information to find a previous node in a coupling chain. For the example above starting from coupling C -> B, dependency information on output $y^{[B]}$ indicates that input $u^{[B]}$ should be set before using output $y^{[B]}$ on another model. With this information a search is made to find a coupling that updates input $u^{[B]}$. If found the process is repeated by looking at the source models output dependency to determine if another coupling should come before it. Eventually a model is found that has no dependency on its output or the couplings have looped back to themselves and an algebraic loop has been found. The latter case is flagged so **update_couplings** can make the necessary calls to solve the loop problem during coupling update by iterating all coupling relations. In case the dependency information is unavailable the system is assumed to contain algebraic loops by default.

3.4.1 Couplings

What differs the aggregated system from other systems is the presence of coupling equations. Naturally this becomes the central part of the problem. The couplings are user provided and must be user friendly, efficient and give enough freedom to the user to model reasonably advanced behaviour. To satisfy these conditions the couplings were separated into two parts, `one-to-one` for simple linear coupling and `advanced` for non-linear or specialized couplings. Coupled systems relying on `one-to-one` couplings have a constant coupling matrix L. This is exploited when computing the aggregated Jacobian matrix.

Couplings of type `one-to-one` let the user set outputs from one model as inputs to another as a list of tuples. For example, $u_4^{[A]} = y_7^{[B]}$, where input u_4 of model A is set to output y_7 of model B. In code it would look like,

```
couplings = [('A', 'u4', 'B', 'y7')]

#couplings = [('m_dst', 'var_dst', 'm_src', 'var_src')]
```

where the first term in the tuple is a keyword for a sub-model, provided by the user at initialization, then comes the input variable name as it is in the sub-model, a keyword for model with outputs and lastly variable name of the output. Internally all inputs are set by looping through the coupling list. This method allows for a user-friendly interface.

In some cases simple `one-to-one` coupling is not enough. To model non-linear coupling behaviour an `advanced` coupling scheme has been developed. It requires the user to provide the coupling function, $c(y, w)$, as a function that takes sub-model outputs and sub-model external excitation inputs as parameters and returns a dictionary with sub-models as key and another dictionary as value. The second dictionary should then hold variable name as key and variable value as value. Coupling example with $u_4^{[A]} = y_7^{[B]}$ would in the `advanced` method look like,

```
def coupling(y, w):
   u = {'A':{}}
   u['A']['u4'] = y['B']['y7']
```

```
    return u
```

This does allow for more advanced coupling behaviour but since all inputs are gathered by a single call, coupling execution order cannot be implemented. Neither can systems containing algebraic loops be detected. To solve this issue all systems coupled with `advanced` type are assumed to contain algebraic loops. Systems without dependency order can change "algebraic_loops" flag in option to *False*, otherwise the entire coupled system is assumed to be one large loop.

To distinguish between `one-to-one` and `advanced` coupling the user makes the call,

```
agg_prob = AggregatedProblem(dic, couplings,
    coupling_type='advanced')
```

where *dic* is a dictionary of models and couplings a function defined as above. See Section 4.1 for a demonstration of the aggregation concept and Section 4.3 for use of `advanced` couplings.

3.4.2 Jacobian Computation

The Jacobian is computed with Eq (18). Matrices A, B, C and D are block diagonal matrices where each block is a corresponding sub-model state space matrix. The aggregated state space matrices are constructed by asking each sub-system for its state space matrix and putting it into correct place. The remaining matrix L must be approximated by `AggregatedProblem`. It is done by perturbing outputs and making calls to $c(y, w)$. Applying Eq (23) on each variable yields a column in L that is used to build up the matrix. Once all matrices have been found Eq (18) is solved as a linear system with a call to Python's sparse.linalg.spsolve method. The Jacobian is then returned in sparse format to the solver. All matrices are stored as sparse in `AggregatedProblem`.

Looking at Eq (18) and considering that the coupling matrix is only non-zero for coupled terms one realises that only inputs and outputs part of coupling have an effect on the Jacobian. With this in mind, matrix sizes can be limited to reduce computations. Systems where the coupling matrix remains constant can be flagged in options with *approximate_coupling_matrix_continuously* set to *False*.

3.4.3 Algebraic Loop Solver

Aggregated systems containing algebraic loops must solve the algebraic loop problem to maintain proper input-output values. This has to be done for each function evaluation. To solve an algebraic loop problem is to solve Eq (9) which is a root-finding problem. One way of solving such problems is by using a root-finding solver. `AggregatedProblem` uses `Kinsol` [2] an advanced root-finding solver part of the Assimulo library. The solver finds outputs, y, that satisfy Eq (9) that are then used to calculate new inputs u. Step by step instructions are presented below,

1. Define residual problem (Eq (9)).
2. Solve residual problem with `Kinsol` to get new outputs y_{new}.
3. Calculate new inputs by calling $u_{new} = c(y_{new}, w)$.
4. Update u_{new} to all sub-models.

Step 1 is solved by defining a function that takes the outputs y as a numpy array argument, updates inputs, calculates new outputs and returns $y - y_{updated}$, where $y_{updated} = g(x, u, w)$. In step 2 the solver iterates with a variation of a Newton method [8] to find values of y that satisfy $|y - y_{updated}| <$ tol where tol is a predetermined tolerance. With outputs y_{new}, inputs u_{new} are calculated in step 3. Finally the inputs are updated to models in step 4.

The loop problem must be solved for each right-hand-side function evaluation making this an essential part of a coupled system. For a demonstration of a coupled system with algebraic loops see Section 4.2.

3.4.4 Event Detection and Handling

Discontinuities can be handled as events and several solvers in Assimulo support events [6]. `Explicit_Problem_Model` and `FMIODE2InputOutput` both inherit from the Assimulo problem class `Explicit_Problem` giving them built in event functionality. Events are designed to stop the integrator, force a call to a handle event function in the model and then resume the integration. When handling an event the model could change values of variables or even swap entire equations when entering different domains. There are three types of events to consider,

- State events.
- Time events.
- Step events.

State events occur when there is a discontinuity dependent on the state variables in the right-hand-side function, f. For an example of how to simulate an ODE system with discontinuities using events with Assimulo see Section 3.1.1. As previously mentioned events are modelled as zero-crossings called event indicators. `AggregatedProblem` aggregates all sub-model event indicators in a **state_events** function that is monitored by Assimulo solvers.

Time events work differently. It is known when a time event occurs prior to integration, instead of monitoring event indicators during integration, the time of the closest event is used as a break point for the solver. The event handler is then called and the solver resumes integration until final time is reached. Assimulo problems handle time events with a **time_event**-function that returns the time of the closest upcoming time event. In `AggregatedProblem` it is enough to call all sub-system **time_event**s to find and return the closest event.

Step events are different and do not affect model behaviour. They are used to help the solver with numerical integration [1]. It could for example be to change coordinate system internally if the current coordinates are no longer feasible numerically. The aggregated problems' step event function calls all sub-system **step_events** that both detect and handle an event if it occurs.

State and time events are monitored with event indicators or a shortest time until next event. This structure allows for adding events to a problem externally. The user can provide a function with event indicators to add additional state events to the aggregated problem or a list with time events. By also adding an external event handler one can define special behaviour at these events. An example of externally added state events is described in Section 4.5. To add time events to a coupled system one would write,

```
agg.set_time_events([1.0, 5.0, 9.0])
```

where the argument is a list of time events and *agg* is an aggregated system.

3.4.5 Result Handling

When solving problems with Assimulo the results are normally stored in the solver. Simulations of FMUs with Assimulo usually have a separate result handler object for this. `AggregatedProblem` must support both systems when saving results after each step. In the coupling extension interface there is a flag *use_custom_result_handler* that can be set to *True/False*. When *True* a subset of the solver specific for the sub-model in question is sent to the models result handler. Otherwise the results are stored in normal Assimulo fashion in the solver. Using a flag for each model allows results to be stored after model preferences.

4 Simulation Examples and Experiments

In this section the described framework is demonstrated by showing simulations of coupled systems modelled with FMUs and Assimulo problems. Aggregated systems containing events, externally added events, and systems containing algebraic loops are tested. The test systems are compared either to analytic solutions or monolithic systems written in Modelica and compiled to FMUs in Dymola or JModelica.org. By a monolithic system is meant that the entire system is modelled in a single FMU. The simulations are made with *Dassl* in Dymola or in Python with `CVode`, a numerical solver in Assimulo, that lets the user define *atol* for local absolute tolerance and *rtol* for local relative error tolerance.

All model source codes except for the Formula 1 Race Car system are available in appendix.

4.1 Simple Coupled System

As a demonstration of the aggregation concept two models with only one state variable each, are coupled and simulated. The two systems are called A and B and are modelled as,

$$\dot{x}^{[A]} = -x^{[A]} \quad (25a) \qquad \dot{x}^{[B]} = u^{[B]} \quad (25d)$$

$$y^{[A]} = x^{[A]} \quad (25b) \qquad y^{[B]} = 0 \quad (25e)$$

$$u^{[A]} = 0 \quad (25c) \qquad u^{[B]} = y^{[A]} \quad (25f)$$

where system A is independent of system B and is recognized as an exponentially decreasing function. In contrast, system B is directly dependent on output signal from system A, with the coupling relation $u^{[B]} = y^{[A]}$. System B contains no output terms and the complete system is void of algebraic loops. The coupled system can be solved analytically and with initial conditions $x^{[A]} = 9$ and $x^{[B]} = 2$, the solution is given as,

$$x^{[A]}(t) = 9e^{-t} \quad (26a)$$

$$x^{[B]}(t) = -9e^{-t} + 11 \quad (26b)$$

The coupled system was solved twice, with the two systems modelled in the Modelica language and compiled to FMUs with the open source tool JModelica.org and as Assimulo problems. Figures 4-5 show the error of simulating the coupled system with the `CVode` solver in Assimulo with tolerances atol = rtol = 10^{-8}. As reference the analytic solution was used.

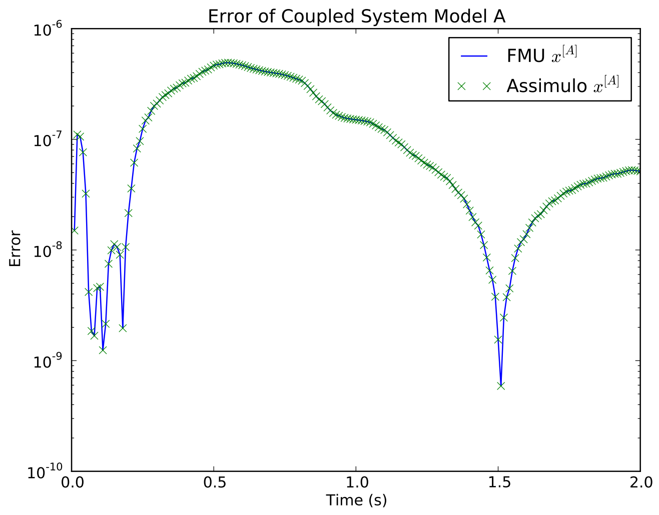

Figure 4: Error of aggregated system A described in Section 4.1 compared to analytic solution. Simulated with `CVode` with tolerances atol = rtol = 10^{-8} for two seconds.

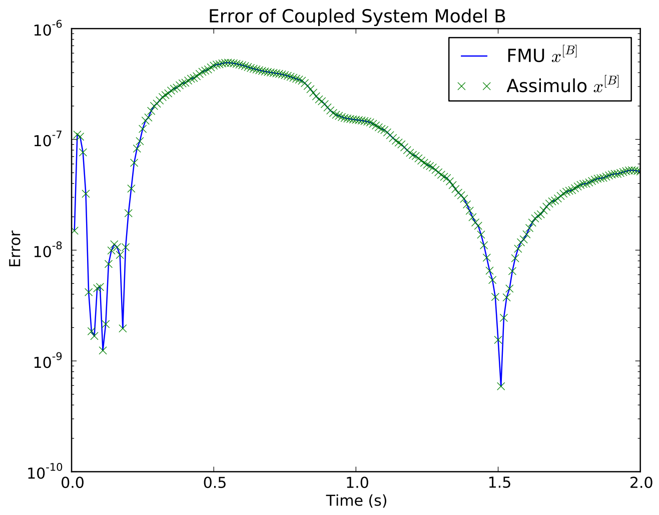

Figure 5: Error of aggregated system B described in Section 4.1 compared to analytic solution. Simulated with `CVode` with tolerances atol = rtol = 10^{-8} for two seconds.

4.2 Coupling Induced Algebraic Loop System

This examples is designed to demonstrate simulation of coupled systems where the couplings bring about algebraic loops. Two systems A and B are described as,

$$\dot{x}^{[A]} = -x^{[A]} + u^{[A]} \tag{27a}$$
$$y^{[A]} = x^{[A]} - u^{[A]} \tag{27b}$$
$$u^{[A]} = y^{[B]} \tag{27c}$$
$$\dot{x}^{[B]} = -x^{[B]} + u^{[B]} \tag{27d}$$
$$y^{[B]} = \frac{1}{2}u^{[B]} \tag{27e}$$
$$u^{[B]} = y^{[A]} \tag{27f}$$

The coupled system can be solved algebraically by making use of the coupling relations to first solve the algebraic loop problem,

$$y^{[A]} = x^{[A]} - u^{[A]} = x^{[A]} - y^{[B]} = \ldots = x^{[A]} - \frac{1}{2}y^{[A]} \tag{28}$$

Solving for $y^{[A]}$ gives,

$$y^{[A]} = \frac{2}{3}x^{[A]} \tag{29}$$

Using Eq (29) with Eq (27a) and applying the integrating factor method gives a solution to system A. Using the results with system B gives the analytic solution to the coupled system,

$$x^{[A]}(t) = Ce^{-\frac{2}{3}t} \tag{30a}$$
$$x^{[B]}(t) = 2Ce^{-\frac{2}{3}t} + De^{-t} \tag{30b}$$

The two systems were modelled in Modelica and compiled to FMUs with the JModelica.org tool and as Python problems with `Explicit_Problem_Model`. An integration was conducted with `CVode` in Assimulo using tolerances atol = rtol = 10^{-8} for two seconds with initial conditions,

$$x^{[A]} = 9.0 \tag{31a}$$
$$x^{[B]} = 2.0 \tag{31b}$$

resulting in $C = 9$ and $D = -16$ in Eq (30). The algebraic loops were solved using `Kinsol` at each right-hand-side function evaluation. Results are compared to the analytic solution in Figure 6 that shows the error of $x^{[A]}$ and $x^{[B]}$ for both FMU and Assimulo sub-systems in log-scale.

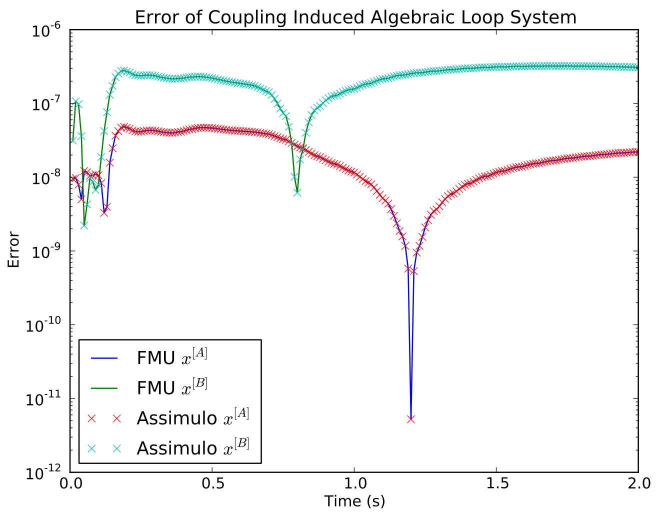

Figure 6: Error of aggregated system with coupling induced algebraic loops described in Section 4.2 in log-scale. Simulated with `CVode` with tolerances atol = rtol = 10^{-8} for two seconds. Simulation results are compared to the analytic solution.

4.3 Coupled Pendulums as FMUs

This example demonstrates application of advanced coupling with FMU models. Two systems, each describing a pendulum in Cartesian coordinates, are coupled with a linear spring to an aggregated system. The setup can be seen in Figure 7. Each pendulum, modelled with unit mass and length, and with a forced excitation on the pivot, is described as,

$$\dot{\bar{x}}_1 = \bar{x}_3 \tag{32a}$$
$$\dot{\bar{x}}_2 = \bar{x}_4 \tag{32b}$$
$$\dot{\bar{x}}_3 = \bar{u}_1 - 2\bar{x}_1\lambda + \bar{u}_2 \tag{32c}$$
$$\dot{\bar{x}}_4 = -g - 2\bar{x}_2\lambda + \bar{u}_3 \tag{32d}$$
$$0 = \bar{x}_1^2 + \bar{x}_2^2 - 1 \tag{32e}$$
$$\bar{y}_1 = \bar{x}_1 \tag{32f}$$
$$\bar{y}_2 = \bar{x}_2 \tag{32g}$$

where $\bar{x}_1$, $\bar{x}_2$ are positions and $\bar{x}_3$, $\bar{x}_4$ are velocities relative to the pendulums pivot. To properly define system constraints λ has been added as an algebraic variable. Inputs $\bar{u}_2$ and $\bar{u}_3$ are forces acting on the bob in the x and y directions respectively and $\bar{u}_1$ is an acceleration due to forced motion of the pivot. Inputs

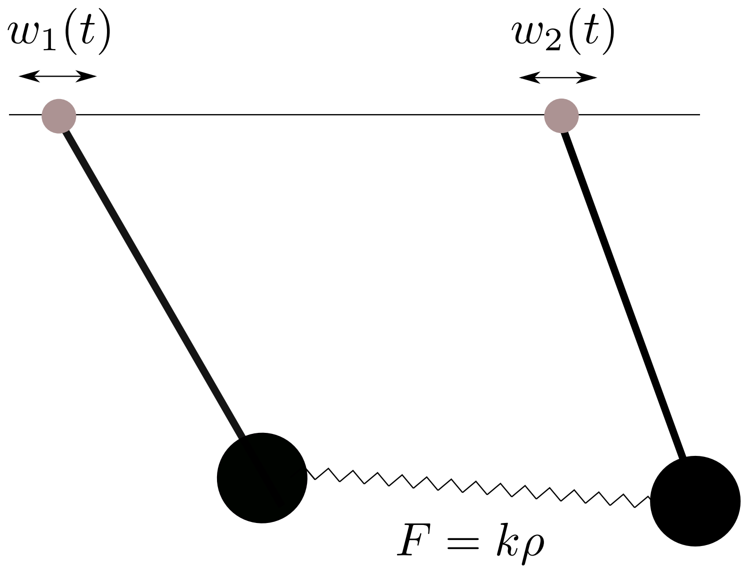

Figure 7: Setup of spring coupled pendulums.

are separated into external excitations and coupling related inputs,

$$\bar{u}^{[i]} = [\underbrace{\bar{u}_1^{[i]}}_{\hat{w}^{[i]}}, \underbrace{\bar{u}_2^{[i]}, \bar{u}_3^{[i]}}_{\hat{u}^{[i]}}]. \tag{33}$$

The pendulums are coupled with a linear spring describing the coupling equations $u = c(y, w)$ as,

$$\begin{bmatrix} \hat{u}_2^{[1]} \\ \hat{u}_3^{[1]} \\ \hat{u}_2^{[2]} \\ \hat{u}_3^{[2]} \end{bmatrix} = k \underbrace{\begin{bmatrix} (\bar{y}_1^{[1]} - a + w^{[1]}) - (\bar{y}_1^{[2]} - b - w^{[2]}) \\ \bar{y}_2^{[1]} - \bar{y}_2^{[2]} \\ -((\bar{y}_1^{[1]} - a + w^{[1]}) - (\bar{y}_1^{[2]} - b - w^{[2]})) \\ -(\bar{y}_2^{[1]} - \bar{y}_2^{[2]}) \end{bmatrix}}_{=:\rho} \tag{34}$$

where k is the stiffness ratio. Variables a and b represent the x-coordinates of the two pendulums' pivots in a Cartesian coordinate system. External input vector is,

$$u_1 = [w^{[1]}, w^{[2]}, \hat{w}^{[1]}, \hat{w}^{[2]}]. \tag{35}$$

From Eq (34), the benefit of including time dependent inputs in couplings is evident. It must be noted that for this example $\hat{w}^{[i]}$ must be chosen $\ddot{w}^{[i]}$.

The pendulum is modelled in Modelica and compiled to an FMU using JModelica.org. The pendulum is described as a DAE of index 3 and the JModelica.org tool is responsible for transforming the model to an FMI supported ODE.

The coupled aggregated system was integrated using `CVode` in Assimulo with Jacobian approximated with forward differences and initial conditions,

$$\bar{x}_1^{[1]} = 1 \tag{36a}$$

$$\bar{x}_2^{[1]} = 0 \tag{36b}$$

$$\bar{x}_1^{[2]} = -1 \tag{36c}$$

$$\bar{x}_2^{[2]} = 0 \tag{36d}$$

It must be noted that the initial conditions are from the reference point of each pendulums pivot. The pivots are positioned at $a = (-2, 0)$ for the pendulum to the left and $b = (2, 0)$ for the one to the right. To model external excitations the function $\sin(t)$ was chosen giving time dependent inputs,

$$w^{[i]} = \sin(t) \tag{37a}$$

$$\hat{w}^{[i]} = -\sin(t) \tag{37b}$$

Stiffness ratio of the spring $k = 1.0$ N/m.

The system was simulated for five seconds using `CVode` with atol = rtol = 10^{-8}. As reference the coupled system was modelled as a monolithic system in Modelica and simulated in Dymola using the solver `Dassl` with tolerance tol = 10^{-12}. Figure 8 shows the error of x, y positions in log-scale.

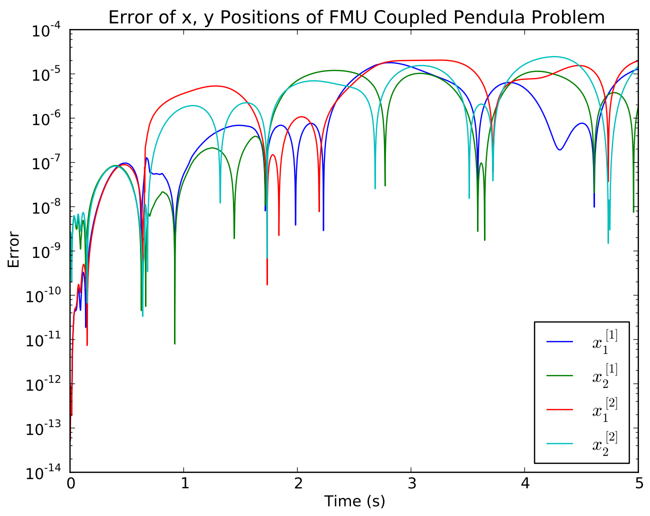

Figure 8: Error of x, y positions of pendulums coupled with linear spring modelled as FMUs. Simulated using `CVode` solver in Assimulo with tolerances atol = rtol = 10^{-8} for five seconds. Variables $x_1^{[i]}$ represents the x-coordinate and $x_2^{[i]}$ the y-coordinate.

4.4 Coupled Pendulums with Mixed Models

To demonstrate coupled systems of mixed FMU and Assimulo models the aggregated system described in Section 4.3 is here remodelled in polar coordinates. Both pendulums are modelled with length 1 m, mass 1 kg and a forced excitation on the pivot. As outputs the models give the angle and angular velocity. The ODE system of a pendulum is described as,

$$\dot{\bar{x}}_1 = \bar{x}_2 \tag{38a}$$
$$\dot{\bar{x}}_2 = (-g + \bar{u}_3)\sin(\bar{x}_1) + (\bar{u}_1 + \bar{u}_2)\cos(\bar{x}_1) \tag{38b}$$
$$\bar{y}_1 = \bar{x}_1 \tag{38c}$$

where $\bar{x}_1$ is the angular displacement with respect to the pivot, $\bar{x}_2$ the angular velocity, g gravitational acceleration, $\bar{u}_1$ is an input of acceleration due to forced motion of the pivot, $\bar{u}_2$ force in x-direction acting on the bob and $\bar{u}_3$ force in y-direction on the bob.

Splitting the input vector into external inputs and coupling related inputs gives,

$$\bar{u}^{[i]} = [\underbrace{\bar{u}_1^{[i]}}_{\hat{w}^{[i]}}, \underbrace{\bar{u}_2^{[i]}, \bar{u}_3^{[i]}}_{\hat{u}^{[i]}}]. \tag{39}$$

The linear spring coupling the pendulums is described as,

$$\begin{bmatrix} \hat{u}_2^{[1]} \\ \hat{u}_3^{[1]} \\ \hat{u}_2^{[2]} \\ \hat{u}_3^{[2]} \end{bmatrix} = k \underbrace{\begin{bmatrix} (\sin(\bar{y}_1^{[1]}) - a + w^{[1]}) - (\sin(\bar{y}_1^{[2]}) - b - w^{[2]}) \\ (-\cos(\bar{y}_1^{[1]}) - (-\cos(\bar{y}_1^{[2]})) \\ -((\sin(\bar{y}_1^{[1]}) - a - w^{[1]}) - (\sin(\bar{y}_1^{[2]}) - b - w^{[2]})) \\ -((-\cos(\bar{y}_1^{[1]}) - (-\cos(\bar{y}_1^{[2]}))) \end{bmatrix}}_{=:\rho} \tag{40}$$

where k is the stiffness ratio. Variable a represents the left-hand-side pendulums pivot point in x-coordinates, variable b is the same for the pendulum to the right. The time dependent external inputs are the same as for the system described in Section 4.3,

$$\bar{u}_1 = [w^{[1]}, w^{[2]}, \hat{w}^{[1]}, \hat{w}^{[2]}] \tag{41}$$

where $w^{[i]}$ has to be chosen $\ddot{w}^{[i]}$. The system was modelled in Modelica and compiled to FMUs with JModelica.org, as Assimulo models and mixed FMU-Assimulo systems. Initial conditions of the angles and angular velocities were chosen as,

$$\bar{x}_1^{[1]} = \frac{\pi}{2} \tag{42a}$$

$$\bar{x}_2^{[1]} = 0 \tag{42b}$$

$$\bar{x}_1^{[2]} = -\frac{\pi}{2} \tag{42c}$$

$$\bar{x}_2^{[2]} = 0 \tag{42d}$$

with spring stiffness ratio $k = 1.0$ N/m and a $\sin(t)$ signal as forced motion on the pivot.

The three aggregated systems were integrated with `CVode` with tolerances atol $=$ rtol $= 10^{-8}$ for five seconds and the Jacobian approximated with forward differences. Results were then compared to a monolithic reference model written in Modelica and simulated with `Dassl` in Dymola using tolerance tol $= 10^{-12}$. Figure 9 shows the error in log-scale of the angle $\bar{x}_1^{[1]}$ of all three systems. Error of angle $\bar{x}_1^{[2]}$ is shown in Figure 10.

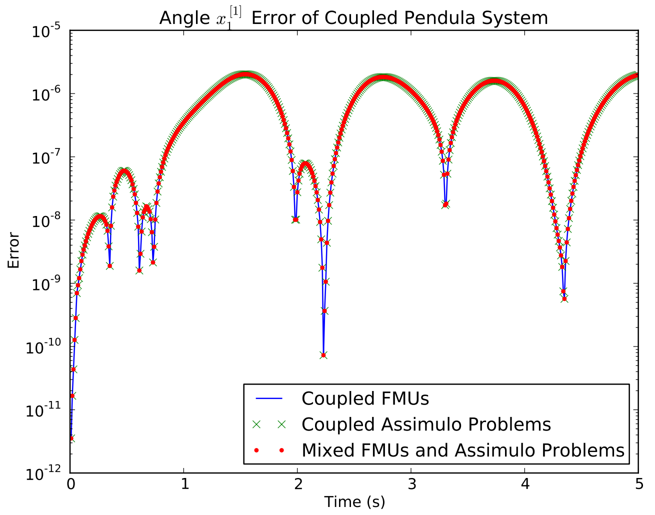

Figure 9: Error of angle $x_1^{[1]}$ of coupled pendulums modelled as FMUs, Assimulo models and mixed FMU and Assimulo models. Simulated with `CVode` in Assimulo with tolerances atol = rtol = 10^{-8} for five seconds. Variable $x_1^{[1]}$ represents angular displacement of pendelum to the left.

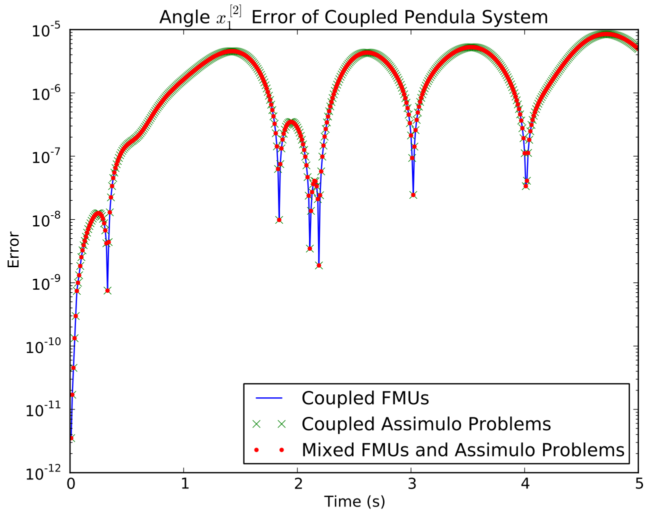

Figure 10: Error of angle $x_1^{[2]}$ of coupled pendulums modelled as FMUs, Assimulo models and mixed FMU and Assimulo models. Simulated with `CVode` in Assimulo with tolerances atol = rtol = 10^{-8} for five seconds. Variable $x_1^{[2]}$ represents angular displacement of pendelum to the right.

4.5 Coupled Pendulums with Externally Added Events

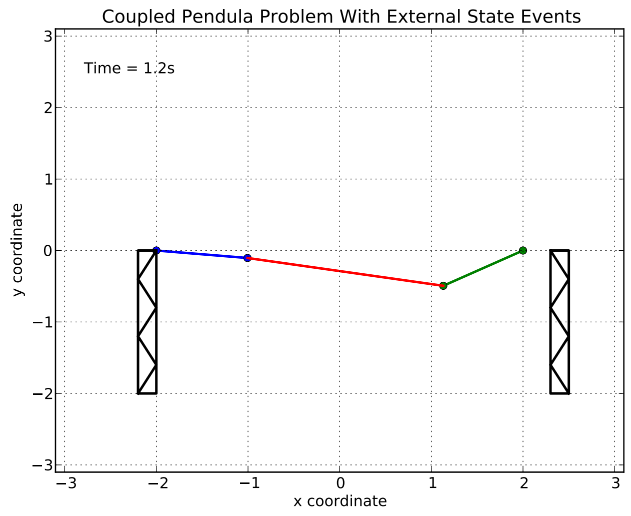

Figure 11: Setup of coupled pendulum system with externally added events as walls blocking the swinging motion of the pendulums.

This example is for demonstrating externally added events to a coupled aggregated system. The coupled pendulum system described in Section 4.3 is reused with the modification that the forced excitations of the pivots have been removed. The spring constant remains at $k = 1.0$ N/m and so do the pivot coordinates, $a = (-2, 0)$ and $b = (2, 0)$.

External events added to the system are walls blocking the pendulum motion, one wall for each pendulum. When the bob swings into the wall a discontinuity occurs. This is handled by defining event indicators that trigger each time an impact occurs. The event indicators are defined as zero-crossings,

$$event^{[i]} = wall^{[i]} - \bar{x}_1^{[i]} \tag{43}$$

where $wall^{[i]}$ is the x-coordinate of the wall blocking pendulum $[i]$ with respect to the pendulums pivot. The impact is modelled as an elastic collision and the event is handled by reversing the direction of the velocity. The walls are placed asymmetrically with the wall blocking the left pendulum placed directly below its pivot meaning that an event occurs when the bobs x-coordinate reaches $\bar{x}_1^{[1]} = 0$ with respect to its pivot. The wall blocking the motion of the pendulum on the right side is placed slightly to the right of its pivot giving a discontinuity when the bob impact the wall at x-coordinate $\bar{x}_1^{[2]} = 0.3$ with respect to its pivot. The setup can be seen in Figure 11. Initial conditions and parameters are the same as in Section 4.3.

The system was integrated using `CVode` with tolerances atol = rtol = 10^{-8} for five seconds. Figure 12 shows the x-displacement of the two pendulums with respect to their pivots. The horizontal lines represent each pendulums' respective wall.

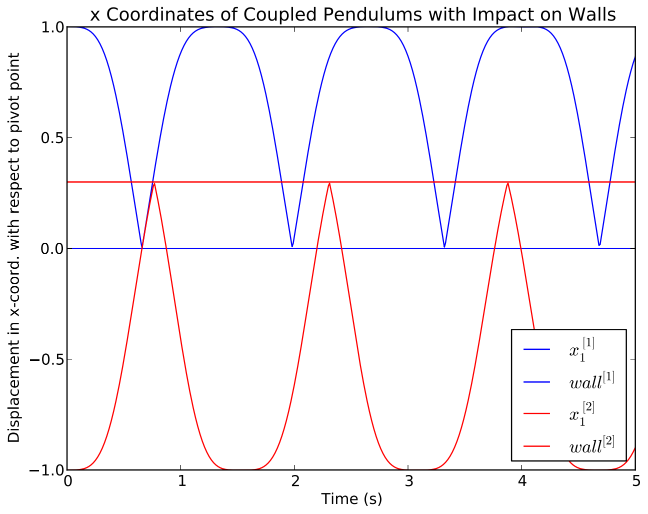

Figure 12: Coupled pendulums with externally added events as walls. Shows x-coordinate displacement with respect to each pendulums' pivot, [1] is pendulum to the left and [2] is to the right. The horizontal lines represent walls blocking the swinging motion.

4.6 Formula 1 Race Car

This example demonstrates the software being used with a large complex model. The test model has been used in previous work [1], and describes a Formula 1 race car driven around an eight shaped race course with increasing velocity. The system containing 47 continuous states and 44 event indicators was modelled as two FMUs, one representing the chassi and the other a wheel. Both sub-systems were then compiled to ME 2.0 FMUs with Dymola. By defining 43 necessary coupling relations between a wheel and the chassi the complete system with 172 couplings was created. Figure 13 shows the phase plot of the car driven around the course simulated for 20 seconds with atol = rtol = 10^{-6} using `CVode` in Assimulo.

As reference the complete system was modelled as a monolithic system and compiled to an FMU. The reference system was the loaded into Python with PyFMI and simulated for 20 seconds using `CVode` in Assimulo with atol = rtol = 10^{-10}. Figure 14 shows the error of the coupled race car system compared to the monolithic system.

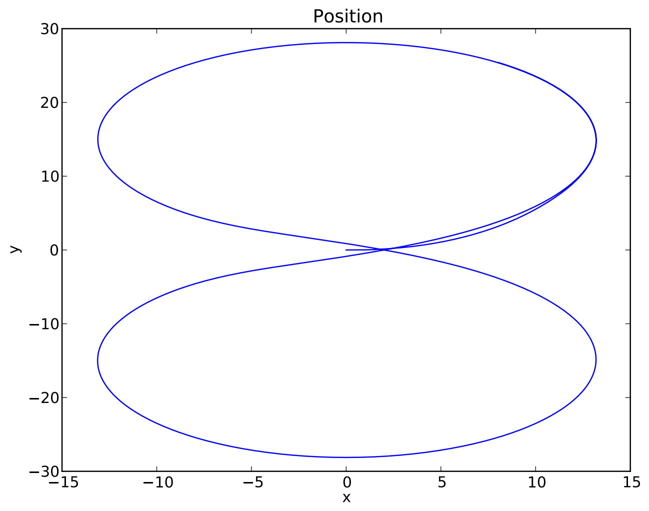

Figure 13: Phase plot of Formula 1 race car simulation with atol = rtol = 10^{-6} for 20 seconds. Shows the race cars position on the race track over time.

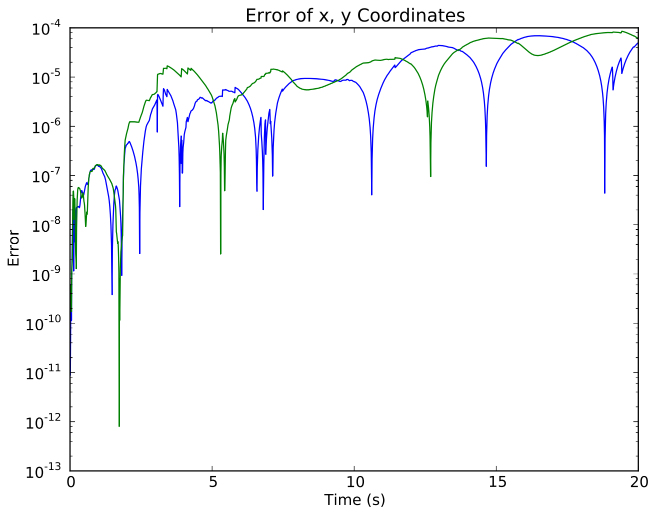

Figure 14: Error of x, y coordinates of Formula 1 model simulated in `CVode` for 20 seconds with atol = rtol = 10^{-6}. The monolithic reference model was simulated in `CVode` with atol = rtol = 10^{-10}.

5 Conclusions

The main goal has been to solve coupled ODE systems while maintaining coupling relations throughout the simulation. Looking at the results in Section 4 the proposed concept delivers in this regard. The concept and the implementation can be used to simulate coupled systems with discontinuities, externally added events, algebraic loops and non-linear couplings. All experiments have been tested against result accuracy and not computation time. The race car simulation, a large complex model, gives reasonable results but took a long time to compute. This has more likely to do with poor implementation than faults in concept.

Sub-models can be coupled in many different ways. This allows for reuse of existing sub-systems for many different applications, as was the case in Sections 4.3-4.5. It is also of huge benefit to be able to swap out specific components of a complex system to compare differences in performance without remodelling the entire system.

5.1 Improvements

The implementation can be improved in many aspects. A big problem has been to find correct execution order for couplings. The implemented method tries a backtracking technique using dependency information to find an order of how inputs must be set. If a loop is detected during tracking all inputs and outputs in the coupled system are seen as part of the loop. Instead of iterating all variables in the coupled system when solving an algebraic loop problem one could try to identify which variables are part of the loop and iterate them only. Going further one could find several unrelated loops that can be iterated separately. Tarjan's algorithm from graph theory could be used for this purpose.

A lot of computation time is spent approximating state space matrices. Currently all sub-system state space matrices are re-approximated each time a call to compute the Jacobian is made. Considering that some models may have one or more constant state space matrices they can be stored in memory for faster access. Another important detail when it comes to matrices is approximation accuracy. The method implemented is a standard forward difference scheme. For better accuracy a higher order scheme can be implemented, unfortunately it usually costs in computation time. It would be interesting to see an implementation of automatic differentiation for approximating state space matrices, both in regards to computation time and approximation accuracy.

5.2 Future Work

The implemented concept does deliver when it comes to simulating coupled systems. One thing that has not been analysed or discussed is simulation performance. By that is meant total computation time, number of steps, number of Jacobian evaluations etc., compared to simulation of a monolithic system.

Performance is something one usually considers when choosing a numeric solver or a method for solving a problem and should be analysed.

Currently a challenge in the field of computational modelling is to find a way of coupling ME and CS FMUs. The target is to find a master algorithm that sufficiently handles coupling relations and discontinuities. Perhaps the concept of aggregating MEs can be used as part of the solution.

References

[1] Christian Andersson. *A Software Framework for Implementation and Evaluation of Co-Simulation Algorithms.* Licentiate thesis, Centre for Mathematical Sciences, Lund University, Lund, Sweden, 2013.

[2] Christian Andersson, Claus Führer, and Johan Åkesson. Assimulo: A unified framework for ode solvers. *Math. Comput. Simulat.*, 2015. doi: 10.1016/j.matcom.2015.04.007. In press.

[3] Torsten Blochwitz, Martin Otter, Johan Åkesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, Hans Olsson, and Antoine Viel. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *In 9th International Modelica Conference 2012*. Modelica Association, 2012.

[4] Dassault Systèmes. Dymola - Multi-Engineering Modeling and Simulation - Version 2016. `http://www.dymola.com/`, 2016. Accessed: 2015-08-01.

[5] Edda Eich-Soelner and Claus Führer. *Numerical Methods in Multibody Dynamics.* European Consortium for Mathematics in Industry (ECMI). Teubner, 1998. ISBN 3-519-02601-5.

[6] Emil Fredriksson, Christian Andersson, and Johan Åkesson. Discontinuities handled with events in Assimulo. In Hubertus Tummescheit and Karl-Erik Årzén, editors, *Proceedings of the 10th International Modelica Conference*, number 96 in Linköping Electronic Conference Proceedings, pages 827–836. Linköping University Electronic Press, Linköpings universitet, 2014. URL `http://dx.doi.org/10.3384/ECP14096827`.

[7] Peter Fritzon and Hilding Elmqvist, editors. *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, Linköping Electronic Conference Proceedings, 2015. Linköping University Electronic Press, Linköpings universitet. doi: http://dx.doi.org/10.3384/ecp15118.

[8] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, September 2005. ISSN 0098-3500. doi: 10.1145/1089014.1089020.

[9] Modelica Association. Modelica: A unified object-oriented language for physical systems modeling, language specification version 3.3 revision 1, 2014. URL `http://www.modelica.org/`.

[10] Modelon AB. PyFMI - Version 2.1. Technical report, 2015. Accessed: 2015-05-18.

[11] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Comput. Chem. Eng.*, 34(11):1737–1749, November 2010. doi: http://dx.doi.org/10.1016/j.compchemeng.2009.11.011.

6 Appendix

In this section the models described in Section 4 are shown in Modelica or Python code. The Formula 1 Race Car system described in 4.6 is not shown in code.

6.1 Coupling Induced Algebraic Loops

Models described in Section 4.2 modelled in Modelica and in Python as Assimulo problems.

6.1.1 System A in Modelica

System A used in Section 4.2 in Modelica.

```
model A
    Real x(start = 9.0);
    input Real u;
    output Real y;
equation
    der(x) = -x + u;
    y = x - u;
end A;
```

6.1.2 System B in Modelica

System B used in Section 4.2 in Modelica.

```
model B
    Real x(start = 2.0);
    input Real u;
    output Real y;
equation
    der(x) = -x + u;
    y = 0.5*u;
end B;
```

6.1.3 System A in Python as Assimulo Problem

System A used in Section 4.2 as Assimulo problem.

```
t0 = 0

def rhsA(t, x, u=None, sw=None):
   return -x + u

def yA(t, x, u=None, sw=None):
   return x - u
```

```
x0A = np.array([9.])

modA = Explicit_Problem_Model(rhsA, x0A, t0,
   outputs=(yA, [('y', 0)]), inputs=(1, [('u', 0)], None))
```

6.1.4 System B in Python as Assimulo Problem

System B used in Section 4.2 as Assimulo problem.

```
t0 = 0

def rhsB(t, x, u=None, sw=None):
   return -x + u

def yB(t, x, u=None, sw=None):
   return 0.5*u

x0B = np.array([2.])

modB = Explicit_Problem_Model(rhsb, x0B, t0,
   outputs=(yB, [('y', 0)]), inputs=(1, [('u', 0)], None))
```

6.2 Coupled Pendulums as FMUs

The coupled pendulum model described in Cartesian coordinates in Section 4.3 modelled in Modelica.

6.2.1 Pendulum Model in Modelica in Cartesian Coordinates

Model of coupled pendulum problem used in Section 4.3 in Cartesian coordinates.

```
model PendulumCartesianCoordinates
    Real x1(start=1.0, fixed = false); // x coordinate
    Real x2(start=0.0, fixed = false); // y coordinate

    Real x3, x4;                       // x3=xvel, x4=yvel

    // Lambda
    Real lambda;
    constant Real g = 9.82;

    input Real w, u1;                 // w, u1=aw
    input Real u2, u3;                // u2=Fx, u3=Fy

    output Real y1, y2;
equation
    der(x1) = x3;
    der(x2) = x4;

    der(x3) = u1 - 2*x1*lambda + u2;
    der(x4) = -g -2*x2*lambda + u3;
    0 = x1^2 + x2^2 - 1;

    y1 = x1;                          // x coordinate
    y2 = x2;                          // y coordinate
end PendulumCartesianCoordinates;
```

6.2.2 Monolithic Reference Model in Modelica

Monolithic reference model described in Modelica for example in Section 6.2.

```
model CoupledPendulumsReferenceCartesian
    // x, y coordinates of left pendulum
    Real x1(start = 1.0, fixed = true);
    Real y1(start = 0.0, fixed = false);

    Real vx1, vy1;  // Velocities, left pendulum

    // x, y coordinates of right pendulum
    Real x2(start = -1.0, fixed = true);
    Real y2(start = 0.0, fixed = false);

    Real vx2, vy2;  // Velocities, right pendulum

    Real lambda1;
    Real lambda2;

    constant Real g = 9.82;
    parameter Real k = 1.0;     // spring constant

    // Pendulum pivot x-coordinates
    parameter Real a = -2.0;    // Left pendulum
    parameter Real b = 2.0;     // Right pendulum

    // Coupling, Spring Forces
    Real Fx, Fy;

    // External excitations on pivots
    input Real w1, aw1;
    input Real w2, aw2;
equation
    // Left Pendulum Dynamics
    der(x1) = vx1;
    der(vx1) = aw1 - 2*x1*lambda1 + Fx;

    der(y1) = vy1;
    der(vy1) = -g -2*y1*lambda1 + Fy;

    0 = x1^2 + y1^2 - 1;

    // Right Pendulum Dynamics
    der(x2) = vx2;
    der(vx2) = aw2 - 2*x2*lambda2 - Fx;

    der(y2) = vy2;
    der(vy2) = -g -2*y2*lambda2 - Fy;

    0 = x2^2 + y2^2 - 1;

    // Coupling with Linear Spring

    Fx = k*((x1 - a + w1) - (x2 - b - w2));
    Fy = k*(y1 - y2);

    // Inputs handled internally
    w1 = Modelica.Math.sin(time);
    aw1 = -Modelica.Math.sin(time);

    w2 = Modelica.Math.sin(time);
    aw2 = -Modelica.Math.sin(time);
end CoupledPendulumsReferenceCartesian;
```

6.3 Coupled Pendulums with Mixed Models

The coupled pendulum model described in Polar coordinates in Section 4.4 modelled in Modelica and as Assimulo problem in Python.

6.3.1 Pendulum Model in Modelica in Polar Coordinates

Coupled pendulums problem in polar coordinates used in Section 4.4.

```
model PendulumPolarCoordinates
  Real x1(start = 0.0, fixed=false); // theta angle
  Real x2(start = 0.0, fixed=false); // thetadot angular velocity

  constant Real g = 9.82;

  input Real u2, u3;                 // u2 = Fx, u3 = Fy
  input Real w, u1;                  // u3 = aw

  output Real y;                     // theta angle
equation
    // System dynamics
    der(x1) = x2;
    der(x2) = (-g + u3)*Modelica.Math.sin(x1) +
              (u1 + u2)*Modelica.Math.cos(x1);

    // Outputs
    y = x1;                         // theta angle
end PendulumPolarCoordiantes;
```

6.3.2 Coupled Pendulum in Polar Coordiantes as Assimulo Problem

Coupled pendulum in Polar coordinates in Python as Assimulo problem:

```
##########################################################
# Dynamics
def rhs(t, x, u=None, sw=None):
   g = 9.82

   theta = x[0]
   theta_dot = x[1]

   w = u[3]
   aw = u[0]
   Fx = u[1]
   Fy = u[2]

   return np.array([theta_dot, (-g + Fy)*np.sin(theta) +
       (Fx+aw)*np.cos(theta)])

# Outputs; theta and theta_dot
def y(t, x, u=None, sw=None):
   return x

# Initial conditions
x0left = np.array([theta_L0, thetaDot_L0])

##########################################################

p_left = Explicit_Problem_Model(rhs, x0left, t0,
   outputs=(y, [('y', 0),    # y = theta; angle
             ('y1', 1)]),    # y1 = thetadot; angular velocity
   inputs=(4,
         [('u1', 0),        # aw
         ('u2', 1),         # Fx
         ('u3', 2),         # Fy
         ('w', 3)],
         (['w', 'u1'], traj)) )

##########

x0right = np.array([theta_R0, thetaDot_R0])

p_right = Explicit_Problem_Model(rhs, x0right, t0,
   outputs=(y, [('y', 0),   # y = theta; angle
             ('y1', 1)]),   # y1 = thetadot; angular velocity
   inputs=(4,
        [('u1', 0),         # aw
         ('u2', 1),         # Fx
         ('u3', 2),         # Fy
```

```
          ('w', 3)],
          (['w', 'u1'], traj)) )
```

6.3.3 Monolithic Reference Model in Modelica

Monolithic reference model for coupled pendulum problem in polar coordinates used in Section 4.4.

```
model CoupledPendulumsReferencePolar
 // Left Pendulum
 Real x1(start = 3.14*0.5, fixed=true);
 Real x2(start = 0.0, fixed=false);

 // Right Pendulum
 Real x3(start = -3.14*0.5, fixed=true);
 Real x4(start= 0.0, fixed=false);

 constant Real g = 9.82;

 parameter Real k = 1.0; // Spring constant

 // Pivot x-coordinates
 parameter Real a = -2.0; // Left Pendulum
 parameter Real b = 2.0;  // Right Pendulum

 // Forces From Coupled Spring
 Real Fx, Fy;

 // External Excitations
 Real w1, aw1;
 Real w2, aw2;

equation
 // Left Pendulum Dynamics
 der(x1) = x2;
 der(x2) = (-g + Fy)*Modelica.Math.sin(x1) +
            (Fx + aw1)*Modelica.Math.cos(x1);

 // Right Pendulum Dynamics
 der(x3) = x4;
 der(x4) = (-g - Fy)*Modelica.Math.sin(x3) +
            (-Fx + aw2)*Modelica.Math.cos(x3);

 // Spring Coupling
 Fx = k*( (Modelica.Math.sin(x1) - a + w1) -
            (Modelica.Math.sin(x3) - b - w2));

 Fy = k*( -Modelica.Math.cos(x1) + Modelica.Math.cos(x3));

 // Inputs Handled Internally

 w1 = Modelica.Math.sin(time);
 w2 = Modelica.Math.sin(time);
```

```
  aw1 = -Modelica.Math.sin(time);
  aw2 = -Modelica.Math.sin(time);
end CoupledPendulumsReferencePolar;
```

6.4 Bouncing Ball Demonstration

In this section scripts for the bouncing ball demonstrations are presented.

6.4.1 Bouncing Ball as Assimulo Problem

A script of the demonstration described in Section 3.1.1 is presented:

```
####
#### Bouncing Ball as Assimulo Problem

# Imports
from assimulo.problem import Explicit_Problem
from assimulo.solvers import CVode

import matplotlib.pyplot as plt
import numpy as np

# Dynamics
def rhs(t, x, sw):
   """
   Right-Hand-Side function of bouncing ball problem.

   x[0] : Height of ball (m)
   x[1] : Velocity of ball (m/s)
   """
   return np.array([x[1], -9.82])

# Event indicators
def state_events(t, y, sw):
   """
   State event function. Defines zero-crossings.
   """
   event1 = y[0] if sw[0] else 5 # y[0] if ball is falling
       down.
   event2 = y[1] if sw[1] else 5 # y[1] if ball is going up.
   return np.array([event1, event2])

# Event handling
def handle_event(solver, event_info):
   """
   Event handling function. Changes switches and
   velocity direction.
   """
   ev_info = event_info[0] # Only looking at state events.
```

```
    if ev_info[0] !=0:
        # Ball bounces up.
        solver.sw[0] = False
        solver.sw[1] = True
        solver.y[1] = -0.8*solver.y[1] # 0.8 elasticity
    else:
        # Ball is at top, starting to fall downwards.
        solver.sw[0] = True
        solver.sw[1] = False

# Initial conditions
x0 = np.array([1.0, 0.0])
t0 = 0

# Defining problem with Assimulo problem class
prob = Explicit_Problem(rhs, x0, t0)
prob.state_events = state_events
prob.handle_event = handle_event
prob.sw0 = [True, False]

# Creating solver instance
sim = CVode(prob)

# Simulating problem
t, h = sim.simulate(2.0, 40) # tf=2.0, ncp=40;

# Plot results
plt.figure()
plt.plot(t, h[:,0])
plt.title('Bouncing Ball as Assimulo Problem')
plt.xlabel('Time (s)')
plt.ylabel('Height above ground (m)')
plt.plot(t, [0]*len(t))  # Plot ground
plt.show()
```

6.4.2 Bouncing Ball with PyFMI

Code for demonstration described in Section 3.1.2. Modelica model:

```
model BouncingBall
    Real h(start = 1.0);
    Real v(start = 0.0);

    constant Real g = 9.81;
equation
    der(h) = v;
    der(v) = -g;

    when h < 0 then
```

```
        reinit(v, -0.8*v);
    end when;
end BouncingBall;
```

Python script demonstrating import and simulation of an FMU with Assimulo solvers:

```
####
#### BouncingBall with FMU

# Imports
from pymodelica import compile_fmu

from pyfmi import load_fmu
from pyfmi.simulation.assimulo_interface import FMIODE2
from pyfmi.common.io import ResultHandlerFile

from assimulo.solvers import CVode
import matplotlib.pyplot as plt
#######################################################
# Compile model
name = compile_fmu('BouncingBall', 'model.mo', target='me',
    version='2.0')

# Load FMU
fmu = load_fmu(name)

# Result handler
opts = fmu.simulate_options()
res = ResultHandlerFile(fmu)
res.set_options(opts)

# Initialize FMU
fmu.setup_experiment()
fmu.initialize()
fmu.event_update()
fmu.enter_continuous_time_mode()

# Create Assimulo-FMU interface instance
mod = FMIODE2(fmu, result_handler=res)

# Star result handler
res.simulation_start()

# Solver instance
sim = CVode(mod)

# Simulation
sim.simulate(2.0, 40) # tf=2.0, ncp=40;
```

```python
# Stop result handler
res.simulation_end()

# Plot result
h = res.get_result().get_variable_data('h').x
t = res.get_result().get_variable_data('time').t

plt.figure()
plt.title('Bouncing Ball as FMU')
plt.xlabel('Time (s)')
plt.ylabel('Height above ground (m)')
plt.plot(t, h)
plt.plot(t, [0]*len(t))
plt.show()
```

Coupling Model Exchange FMUs for Aggregated Simulation by Open Source Tools

Pukashawar Pannu[1] Christian Andersson[1,2] Claus Führer[1] Johan Åkesson[2]

[1]Centre for Mathematical Sciences, Lund University, Sweden
[2]Modelon AB, Sweden

Abstract

The Functional Mock-up Interface standard allows to generate stand-alone sub-systems which can be simulated and verified individually. In this paper we present a design of a model aggregation which allows to simulate several Functional Mock-up Units as a coupled model. The formulation is based on Assimulo as a numerical integration environment. Assimulo problem classes are extended to a class for aggregated problems which collects information provided by the Functional Mock-up Units through the tool PyFMI together with Python based problem classes defined by Assimulo. This allows to set-up test environments of complex models composed of several sub-systems.

Keywords: FMI, Jacobian, Algebraic loops, Events, Model Exchange 2.0, Assimulo

1 Introduction

The Functional Mock-up Interface (FMI) (Blochwitz et al., 2012) has gained momentum in simulation of dynamical systems and in exchanging dynamic simulation models between tools. The standard has proven to be highly successful as it fills a gap where there were costly custom integrations before. The open source tools PyFMI [1] together with Assimulo (Andersson et al., 2015) provide a solid foundation for performing simulations and experiments on single Functional Mock-up Units (FMUs).

A key feature that is currently lacking is the ability to easily simulate coupled systems and thus fully taking advantage of the standard.

In this article, an extension to the open-source tools PyFMI and Assimulo is presented that allows for simulation of coupled model exchange FMUs following the FMI 2.0 standard. The extension enables coupling of FMUs and models written directly in Python to a so-called aggregated model.

[1]`http://www.pyfmi.org` PyFMI - Version 2.1. Accessed, 2015-05-18

The dynamical models considered here can be described as,

$$\dot{\bar{x}} = \bar{f}(\bar{x},\bar{u}) \tag{1a}$$
$$\bar{y} = \bar{g}(\bar{x},\bar{u}) \tag{1b}$$

where $\bar{x}$ represents the states, $\bar{u}$ the input signal and $\bar{y}$ the output, consistent with the FMI.

Commonly, a full system model is represented by several stand-alone sub-systems coupled together by coupling equations to a model for a global system. This results in the following general system description,

$$\dot{x} = f(x,u,w) \tag{2a}$$
$$y = g(x,u,w) \tag{2b}$$
$$u = c(y,w) \tag{2c}$$

where x represents the combined states from the separate models. The local inputs for the ith model, $\bar{u}^{[i]}$, has here been separated into two vectors, $\bar{u}^{[i]} = [\hat{u}^{[i]}, \hat{w}^{[i]}]$, and subsequently combined into the global vectors (for N models), $u = [\hat{u}^{[1]}, \ldots, \hat{u}^{[N]}]$ and $w = [\hat{w}^{[1]}, \ldots, \hat{w}^{[N]}]$. This as to separate between inputs determined by the coupling, u, and external inputs acting on the coupled system, w. In general the external inputs can not only influence the model behaviour directly but also the coupling, Eq (2c), which is highlighted in Section 3.

When solving a coupled system, an approach is co-simulation as is explored in (Andersson, 2013) where the systems have their own integrator and the focus is on communication between systems. In this paper however, the focus is on coupling model exchange FMUs under a single solver.

2 Concept

The idea is to take N coupled sub-systems, either FMUs or Python models, and aggregate them into a single system and treating the final full system as any other model. In order to facilitate the general description of a sub-system, as is defined in FMI, for the aggregated system, care needs to be considered in how for example the Jacobian is defined. The Jacobian is a necessity when using implicit methods for solving the resulting system and is discussed in Section 2.3. Additionally, the events for each sub-system and external events need to be considered, discussed in Section 2.2, as well as algebraic loops which can occur due to the coupling, Section 2.4.

Now, looking at N sub-systems,

$$\dot{\bar{x}}_1^{[1]} = \bar{f}_1^{[1]}(\bar{x}_1^{[1]}, \hat{u}_1^{[1]}, \hat{w}_1^{[1]}) \tag{3a}$$
$$\bar{y}_1^{[1]} = \bar{g}_1^{[1]}(\bar{x}_1^{[1]}, \hat{u}_1^{[1]}, \hat{w}_1^{[1]}) \tag{3b}$$
$$\bar{u}_1^{[1]} = \bar{c}_1^{[1]}(\bar{y}^{[1]}, \hat{w}_1^{[1]}) \tag{3c}$$

$$\vdots$$

$$\dot{\bar{x}}_N^{[N]} = \bar{f}_N^{[N]}(\bar{x}_N^{[N]}, \hat{u}_N^{[N]}, \hat{w}_N^{[N]}) \tag{4a}$$
$$\bar{y}_N^{[N]} = \bar{g}_N^{[N]}(\bar{x}_N^{[N]}, \hat{u}_N^{[N]}, \hat{w}_N^{[N]}) \tag{4b}$$
$$\bar{u}_N^{[N]} = \bar{c}_N^{[N]}(\bar{y}^{[N]}, \hat{w}_N^{[N]}) \tag{4c}$$

and the resulting aggregated system,

$$\dot{x} = f(x,u,w) = \begin{bmatrix} \bar{f}_1^{[1]}(\bar{x}_1^{[1]}, \hat{u}_1^{[1]}, \hat{w}_1^{[1]}) \\ \vdots \\ \bar{f}_1^{[N]}(\bar{x}_1^{[N]}, \hat{u}_1^{[N]}, \hat{w}_1^{[N]}) \end{bmatrix} \tag{5a}$$

$$y = g(x,u,w) = \begin{bmatrix} \bar{g}_N^{[1]}(\bar{x}_N^{[1]}, \hat{u}_N^{[1]}, \hat{w}_N^{[1]}) \\ \vdots \\ \bar{g}_N^{[N]}(\bar{x}_N^{[N]}, \hat{u}_N^{[N]}, \hat{w}_N^{[N]}) \end{bmatrix} \tag{5b}$$

$$u = c(y,w) = \begin{bmatrix} \bar{c}_1^{[1]}(\bar{y}^{[1]}, \hat{w}_1^{[1]}) \\ \vdots \\ \bar{c}_N^{[N]}(\bar{y}^{[N]}, \hat{w}_N^{[N]}) \end{bmatrix} \tag{5c}$$

The vectors x, y, u and w of the aggregated system are defined as:

$$x = \begin{bmatrix} \bar{x}_1^{[1]} \\ \vdots \\ \bar{x}_N^{[N]} \end{bmatrix}, \quad y = \begin{bmatrix} \bar{y}_1^{[1]} \\ \vdots \\ \bar{y}_N^{[N]} \end{bmatrix}, \quad u = \begin{bmatrix} \hat{u}_1^{[1]} \\ \vdots \\ \hat{u}_n^{[N]} \end{bmatrix}, w \;\; = \begin{bmatrix} \hat{w}_1^{[1]} \\ \vdots \\ \hat{w}_N^{[N]} \end{bmatrix}$$

2.1 Aggregated Problem

Using the open-source tools PyFMI together with Assimulo, an FMU can be accessed from Python together with being solved using solvers available in Assimulo. With this in mind two Assimulo problem classes have been worked on. One that creates an input/output problem structure called `ExplicitProblemModel`. The other aggregates several FMUs, or `ExplicitProblemModels`, to one large problem that can be integrated using one of Assimulos available solvers, called `AggregatedProblem`. For simplicity an already existing problem class, `ExplicitProblem`, was extended to handle the aggregation. To define an aggregated problem class some basic data is required:

- Aggregated states.
- RHS (Right-Hand-Side) function of aggregation.
- Coupling handling.

Through PyFMI there exists already a wrapper interface that can load an FMU ME 2.0 and integrate it using Assimulo. When instantiating the `AggregatedProblem` class a list of FMUs is provided from which the initial states are easily accessible and aggregated,

```
for model in models:
    aggregated_x0 aggregate model.x0
```

The crucial part of the aggregated problem class is how to handle the right hand side function. The first major difference between an aggregated problem and an Assimulo problem is the presence of couplings. For each call to the RHS, coupling terms must be up to date. The condition can be satisfied by updating the coupling relations within the RHS-function.

Since the separate problem classes already have an RHS-function structure, computing the RHS-function of the aggregated system is simply to call the RHS-function of each sub-system,

```
set_connections()
for model in models:
    aggregated_rhs aggregate model.rhs
```

Coupling handling is done in *set_connections*(). For simple cases when for example system A input $u_2^{[A]}$ needs inputs from system B output $y_4^{[B]}$, the function simply sets $u_2^{[A]} = y_4^{[B]}$. However, this is not always the case which is further discussed in Section 2.4.

For implicit solvers a Jacobian is required and must be provided by `AggregatedProblem`. More advanced models require `AggregatedProblem` to take into account events and algebraic loops. The three mentioned topics are affected by aggregation and are discussed in the following sections.

2.2 Events

Many models include discontinuities. One way of integrating such systems is by using events (Eich-Soelner and Führer, 1998) which requires that a set of event indicators are monitored during the integration. The integration is interrupted when conditions on the event indicators are violated and the event (discontinuity) is appropriately handled, finally the integration is restarted. Most of the solvers available in Assimulo have this functionality (Fredriksson et al., 2014).

The aggregated problem can access event indicators of an FMU. When asked for event indicators by an Assimulo solver the `AggregatedProblem` combines all event indicators from the sub-systems and hands them to the solver. Once an event has been detected and the integration stopped, the problem class identifies the triggered event and calls the corresponding sub-system's event handling.

Another type of events in FMUs are time events, which are known at the start of a simulation. They split up the integration into segments by setting up end times for the simulation at which point an event is handled (Andersson, 2013). This could be, for instance, a force periodically applied to the system. It is up to the aggregated system to search through all sub-systems for the closest time event to define the end time of the next integration segment and handle the event.

From the Assimulo problem design it is simple to add events to a problem. For the aggregated problem, adding of events would be to add external events to a coupled system. Events can not only be provided through the sub-systems but also through how the system is coupled. Consider a pendulum with no knowledge of its surroundings. Now, in the system model the pendulum is positioned such that its degree of freedom is limited by for instance positioning close to a wall. The limitation can be considered as an external event that needs to be taken into account. In the problem formulation this is easily done by providing extra sets of event indicators for the integrator to monitor.

2.3 Jacobian

When solving an ODE the Jacobian can be explicitly provided or numerically approximated. For an uncoupled input/output system where the inputs are only time dependent the Jacobian, $\frac{\partial \bar{f}}{\partial \bar{x}}$, is computed. When looking at a coupled system the dynamic changes. Due to coupling some input terms are state dependent instead of time dependent as in the uncoupled case. Consider the coupled system,

$$\dot{x} = f(x,u,w) \tag{6a}$$
$$y = g(x,u,w) \tag{6b}$$
$$u = c(y,w) \tag{6c}$$

Inserting Eq (6c) into Eq (6a) and Eq (6b) gives:

$$\dot{x} = f(x,c(y),w) \tag{7a}$$
$$y = g(x,c(y),w) \tag{7b}$$

Differentiating Eq (7a) with respect to x yields:

$$J = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial y}\frac{\partial y}{\partial x} \tag{8}$$

The term $\frac{\partial y}{\partial x}$ is found by differentiating Eq (7b):

$$\frac{\partial y}{\partial x} = \frac{\partial g}{\partial x} + \frac{\partial g}{\partial c}\frac{\partial c}{\partial y}\frac{\partial y}{\partial x} \tag{9}$$

Solving for $\frac{\partial y}{\partial x}$ gives:

$$\frac{\partial y}{\partial x} = \left(I - \frac{\partial g}{\partial c}\frac{\partial c}{\partial y}\right)^{-1}\frac{\partial g}{\partial x} \tag{10}$$

Resulting in the Jacobian:

$$J = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial y}\left(I - \frac{\partial g}{\partial c}\frac{\partial c}{\partial y}\right)^{-1}\frac{\partial g}{\partial x} \tag{11}$$

For the system to be solvable there is necessary condition that $(I - \frac{\partial g}{\partial c}\frac{\partial c}{\partial y})$ is non singular. The $\frac{\partial g}{\partial c}$ term handles the coupling relations and $\frac{\partial c}{\partial y}$ the sub-system feed-through terms.

With FMI 2.0 models have an option to provide directional derivatives. In case they are provided `AggregatedProblem` uses directional derivatives to approximate the aggregated Jacobian matrix. If directional derivatives are unavailable a forward difference scheme is applied. The same applies for non-FMI models.

2.4 Algebraic Loops

When a system contains feed-through, i.e. when the partial derivative of Eq (6b) with respect to u is not the zero matrix, then, in general, an equation system needs to be solved to maintain consistent input and output values satisfying,

$$y = g(x,u,w) \tag{12a}$$
$$u = c(y,w). \tag{12b}$$

By rewriting Eq (12a) to,

$$y - g(x,c(y,w),w) = 0 \tag{13}$$

the algebraic loop can be solved by an iterative method. `AggregatedProblem` creates a residual function of the left-hand-side of Eq (13) and uses the `Kinsol` solver in Assimulo to solve the problem. `Kinsol` is a nonlinear algebraic equation solver, part of the `SUNDIALS` suite (Hindmarsh et al., 2005). When the outputs are known, Eq (12b) is used to update the inputs.

2.5 Workflow

The simulation flow of coupled systems using the aggregated problem class and an Assimulo solver is illustrated in Figure 1. The simulation flow is essentially equivalent to that of simulating an ODE with Assimulo, however, some nodes are affected by aggregation and these are coloured blue in the figure.

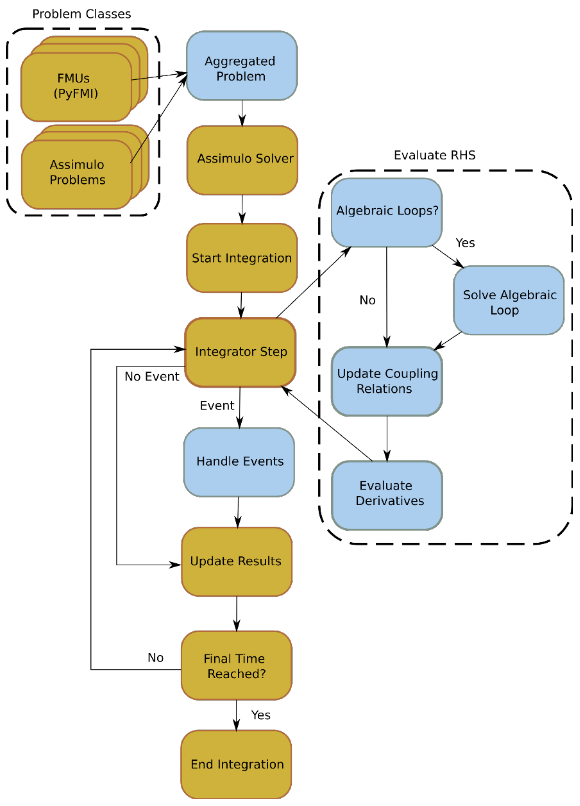

Figure 1. Assimulo simulation flow of coupled systems using FMUs, Assimulo problems and the aggregated problem class. The blue color represents nodes that are affected by aggregation.

3 Examples

In this section, the proposed framework is demonstrated. The ability to couple model exchange FMUs is shown together with coupling of FMUs with models directly defined in Python. Additionally, simulation of coupled models with externally defined events is demonstrated.

3.1 Coupled Pendula

This example demonstrates how two FMUs, each describing a pendulum, are coupled to an aggregated model. The full system consists of two pendula coupled by a force and excited by two inputs acting on the pivots.

The pendulum, with mass 1 kg and length 1 m, is described by,

$$\dot{\bar{x}}_1 = \bar{x}_3 \tag{14a}$$
$$\dot{\bar{x}}_2 = \bar{x}_4 \tag{14b}$$
$$\dot{\bar{x}}_3 = \bar{u}_1 - 2\bar{x}_1\lambda + \bar{u}_2 \tag{14c}$$
$$\dot{\bar{x}}_4 = -g - 2\bar{x}_2\lambda + \bar{u}_3 \tag{14d}$$
$$0 = \bar{x}_1^2 + \bar{x}_2^2 - 1 \tag{14e}$$
$$\bar{y}_1 = \bar{x}_1 \tag{14f}$$
$$\bar{y}_2 = \bar{x}_2 \tag{14g}$$

where $\bar{x}_1, \bar{x}_2$ are positions and $\bar{x}_3, \bar{x}_4$ velocities relative to the pendulum's pivot. The inputs are forces, $\bar{u}_2$ and $\bar{u}_3$, acting on the body's center and an acceleration, $\bar{u}_1$ due to a forced motion of the pivot. The outputs, $\bar{y}$, are the positions.

In order to couple two pendula, $i = [1,2]$, the input vector is split for each pendulum into external excitations and inputs determined by the coupling,

$$\bar{u}^{[i]} = [\underbrace{\bar{u}_1^{[i]}}_{\hat{w}^{[i]}}, \underbrace{\bar{u}_2^{[i]}, \bar{u}_3^{[i]}}_{\hat{u}^{[i]}}]. \tag{15}$$

The two pendula are coupled by a linear spring which is determined by the equation, $u = c(y,w)$,

$$\begin{bmatrix} \hat{u}_2^{[1]} \\ \hat{u}_3^{[1]} \\ \hat{u}_2^{[2]} \\ \hat{u}_3^{[2]} \end{bmatrix} = k \underbrace{\begin{bmatrix} \bar{y}_1^{[1]} - a + w_1 - (\bar{y}_1^{[2]} - b - w_2) \\ \bar{y}_2^{[1]} - \bar{y}_2^{[2]} \\ -(\bar{y}_1^{[1]} - a + w_1 - (\bar{y}_1^{[2]} - b - w_2)) \\ -(\bar{y}_2^{[1]} - \bar{y}_2^{[2]}) \end{bmatrix}}_{=:\rho} \tag{16}$$

where k is the stiffness ratio. Variable a represents the pivot points x-coordinate of the left pendulum and b the point of the right pendulum. The external input vector is,

$$w = [w_1, w_2, \hat{w}^{[1]}, \hat{w}^{[2]}]. \tag{17}$$

The setup is shown in Figure 2. As previously mentioned, it is necessary to include the external inputs into the coupling as is made evident in this example. Note also, that in this example $\hat{w}^{[i]}$ has to be chosen as $\ddot{w}^{[i]}$.

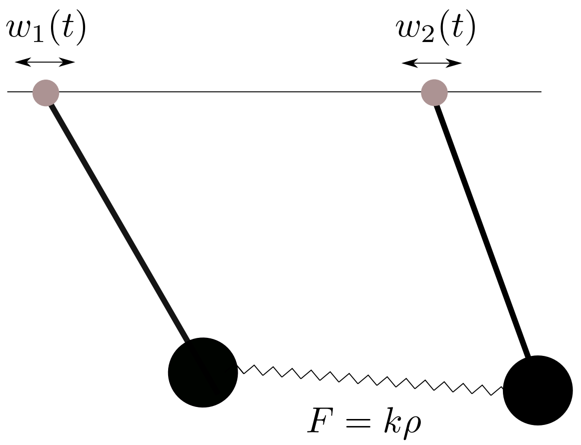

Figure 2. Two pendulums coupled via a spring.

The pendulum is modelled in the Modelica language and using the open-source tool JModelica.org (Åkesson et al., 2010) the Modelica model is compiled into an FMU. The tool is responsible for transforming the pendulum which is described as a DAE of index 3 into an ODE that FMI supports.

The aggregated system was integrated using Assimulo `CVode` solver with tolerances atol = rtol = 10^{-8} for 5 seconds and the Jacobian approximated with forward differences. Initial conditions for the system,

$$\bar{x}_1^{[1]} = 1 \tag{18}$$

$$\bar{x}_2^{[1]} = 0 \tag{19}$$

$$\bar{x}_1^{[2]} = -1 \tag{20}$$

$$\bar{x}_2^{[2]} = 0 \tag{21}$$

note that the initial conditions are from the reference point of each pendulums pivot. The pivot points are located at $(-2,0)$ for the left pendulum and $(2,0)$ for the pendulum to the right. As external forces acting on the pivots the $sin(t)$ function was chosen. Stiffness ratio of the spring is set to $k = 1.0$ N/m.

As reference a monolithic model of the system was created in Modelica and simulated in Dymola (Dassault Systèmes, 2016) using the solver `Dassl` with tolerance tol $= 10^{-12}$. Error of both pendulums x, y positions is presented in Figure 3 in log-scale.

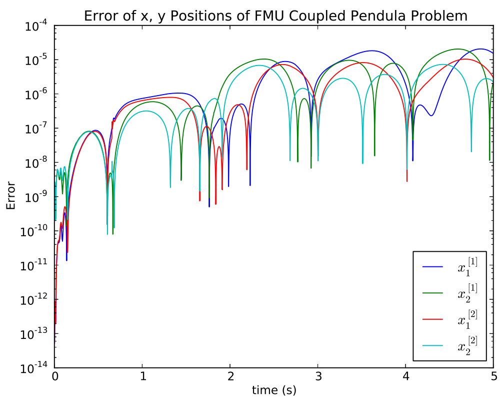

Figure 3. Error of x, y positions of aggregated coupled pendulum described in Section 3.1, simulated with CVode with atol = rtol $= 10^{-8}$ for 5 seconds. $x_1^{[i]}$ denotes the x-coordinate and $x_2^{[i]}$ the y of model i. Model [1] is the pendulum to the left and [2] the one to the right.

3.2 Coupled Pendula with Different Model Types

Three aggregated systems of coupled pendulas were modelled and compared to a monolithic reference model. The first system built by two FMU models modelled in Modelica and compiled with JModelica.org. The second by two Assimulo models and the third with the left pendulum as an Assimulo model and the right pendulum as an FMU. For this example the pendulums were modelled as ODEs in polar coordinates with unit mass and length,

$$\dot{\bar{x}}_1 = \bar{x}_2 \tag{22a}$$

$$\dot{\bar{x}}_2 = (-g+\bar{u}_3)sin(\bar{x}_1) + (\bar{u}_2+\bar{u}_1)cos(\bar{x}_1) \tag{22b}$$

$$\bar{y}_1 = \bar{x}_1 \tag{22c}$$

where g is gravitational acceleration, $\bar{x}_1$ is angular displacement with respect to the pivot point, $\bar{x}_2$ angular velocity. The inputs $\bar{u}_2$ and $\bar{u}_3$ are forces acting on the bob horizontally and vertically respectively. $\bar{u}_1$ is an input of acceleration due to a forced motion of the pivot. The output, $\bar{y}_1$, is the angular displacement.

Similarly to the example described in Section 3.1 the input vector is split into external excitations and inputs by coupling.

$$\bar{u}^{[i]} = [\underbrace{\bar{u}_1^{[i]}}_{\hat{w}^{[i]}}, \underbrace{\bar{u}_2^{[i]}, \bar{u}_3^{[i]}}_{\hat{u}^{[i]}}]. \tag{23}$$

The linear spring coupling the two pendulas is determined by,

$$\begin{bmatrix} \hat{u}_2^{[1]} \\ \hat{u}_3^{[1]} \\ \hat{u}_2^{[2]} \\ \hat{u}_3^{[2]} \end{bmatrix} = k \underbrace{\begin{bmatrix} (sin(\bar{y}_1^{[1]}) - a + w_1) - (sin(\bar{y}_1^{[2]}) - b - w_2) \\ (-cos(\bar{y}_1^{[1]}) - (-cos(\bar{y}_1^{[2]})) \\ -((sin(\bar{y}_1^{[1]}) - a - w_1) - (sin(\bar{y}_1^{[2]}) - b - w_2)) \\ -((-cos(\bar{y}_1^{[1]}) - (-cos(\bar{y}_1^{[2]}))) \end{bmatrix}}_{=:\rho} \tag{24}$$

where k is the stiffness ratio. Variables a and b represent the pivot points x-coordinate for the left-hand-side and right-hand-side pendulas respectively. The external input vector is,

$$u_1 = [w_1, w_2, \hat{w}^{[1]}, w^{\hat{[2]}}]. \tag{25}$$

As with example in Section 3.1, $\hat{w}^{[i]}$ has to be chosen as $\ddot{w}^{[i]}$.

Initial conditions for the aggregated system were chosen for the pendulas to mirror each other with angles $\frac{\pi}{2}$ and $-\frac{\pi}{2}$ for the left and right pendulas and zero initial angular velocity.

$$\bar{x}_1^{[1]} = \frac{\pi}{2} \tag{26a}$$

$$\bar{x}_2^{[1]} = 0 \tag{26b}$$

$$\bar{x}_1^{[2]} = -\frac{\pi}{2} \tag{26c}$$

$$\bar{x}_2^{[2]} = 0 \tag{26d}$$

As external force exciting the pendula pivots a $sin(t)$ signal was chosen and the springs stiffness ratio $k = 1.0$ N/m.

The aggregated system was integrated using the `CVode` solver in the Assimulo package with tolerances, atol = rtol $= 10^{-8}$ for a time of 5 seconds and the Jacobian approximated using forward differences. Results were then compared to a reference where the coupled pendulas were modelled as a monolithic system in Modelica and simulated with `Dassl` in Dymola using tolerance tol $= 10^{-12}$. Figure 4 shows the error in log-scale of the angle $\bar{x}_1^{[1]}$ of all three systems compared to the control. The same plot for angle $\bar{x}_1^{[2]}$ is shown in Figure 5.

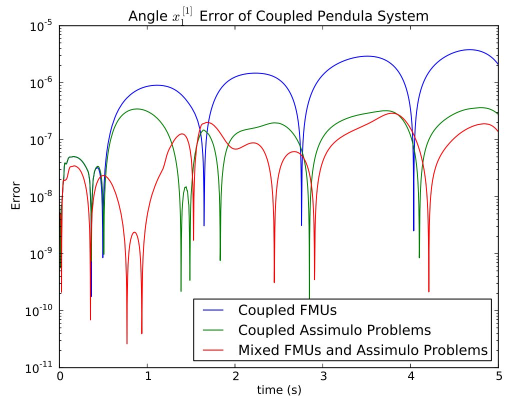

Figure 4. Error of angle $x_1^{[1]}$ of aggregated FMU, Assimulo and mixed systems, simulated for 5 seconds with tolerances atol $=$ rtol $= 10^{-8}$ with CVode solver in Assimulo package.

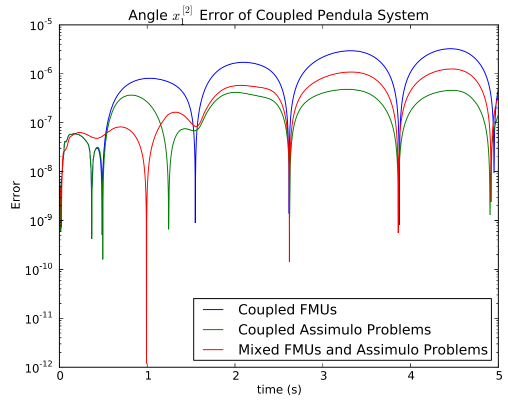

Figure 5. Error of angle $x_1^{[2]}$ of aggregated FMU, Assimulo and mixed systems, simulated for 5 seconds with tolerances atol $=$ rtol $= 10^{-8}$ with CVode solver in Assimulo package.

3.3 Coupled Pendula Impact on Wall

The aggregated system constructed with two FMUs in Section 3.1 is here reused with the addition of discontinuities as two walls are externally placed in the path of the two pendulums' swinging motion. One wall for each pendulum. This as to highlight the possiblity of externally adding state events to the coupled problem. Also the external forces acting on the pivots have been removed.

When the bob hits the wall a discontinuity occurs. This is handled by defining event indicators that trigger an event when the impact occurs. Event indicators are defined as zero-crossings as,

$$event^{[i]} = wall^{[i]} - \bar{x}_1^{[i]} \qquad (27)$$

where $wall^{[i]}$ is the x-coordinate of the wall blocking pendulum $[i]$. The impact itself is elastic and the event handling is done by simply reversing the velocity of the bob. For the pendulum to the left a wall is placed directly below the pivot point and the impact occurs when

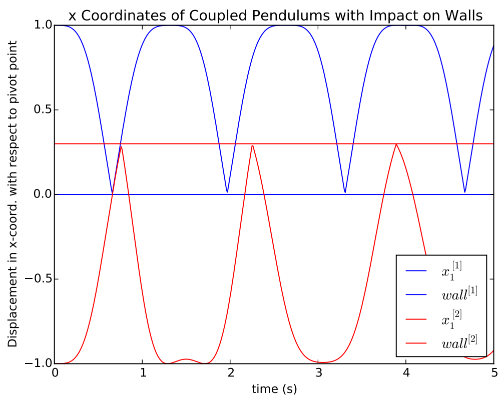

Figure 6. Shows the x-coordinate displacement, with respect to their own pivots, of the pendulums [1], to the left, and [2], to the right, has they hit a wall. The horizontal lines represent walls blocking each pendulums path.

the bobs x-coordinate reaches $\bar{x}_1^{[1]} = 0$ with respect to its pivot. The right-hand-side pendulum wall is placed slightly to the right of its pivot and the bob impacts the wall when its x-coordinate reaches $\bar{x}_1^{[2]} = 0.3$ with respect to its pivot. Initial conditions and parameters are the same as for the example described in Section 3.1.

The aggregated system was integrated with the `CVode` solver with tolerances atol $=$ rtol $= 10^{-8}$ for 5 seconds. Figure 6 shows the x-coordinate displacement with respect to each pendulums pivot. The two horizontal lines represent each pendulums respective walls.

4 Conclusion

In this paper, a framework has been presented for simulation of coupled systems by aggregation. Care needs to be taken when a coupled system contains feed-through as an equation system needs to be solved in order to compute the derivatives of the system. This puts a condition on the sub-system feed-through terms that also presents itself when computing the Jacobian.

The sub-system events are handled by aggregation. A benefit of this approach is that events from all sub-systems together with external events can be monitored at once and handled through the aggregated system. Example described in Section 3.3 shows that external events can be added to an aggregated coupled system.

The FMI has all functionality needed to carry out the presented scheme. By combining the discussed ideas with Assimulo and allowing direct coupling of FMUs and Python based problems one gets a flexible and powerful environment for solving coupled dynamical problems.

References

Christian Andersson. *A Software Framework for Implementation and Evaluation of Co-Simulation Algorithms*. Licentiate thesis, Centre for Mathematical Sciences, Lund University, Lund, Sweden, 2013.

Christian Andersson, Claus Führer, and Johan Åkesson. Assimulo: A unified framework for ode solvers. *Math. Comput. Simulat.*, 2015. doi:10.1016/j.matcom.2015.04.007. In press.

Torsten Blochwitz, Martin Otter, Johan Åkesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, Hans Olsson, and Antoine Viel. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *In 9th International Modelica Conference 2012*. Modelica Association, 2012.

Dassault Systèmes. Dymola - Multi-Engineering Modeling and Simulation - Version 2016. `http://www.dymola.com/`, 2016. Accessed: 2015-08-01.

Edda Eich-Soelner and Claus Führer. *Numerical Methods in Multibody Dynamics*. European Consortium for Mathematics in Industry (ECMI). Teubner, 1998. ISBN 3-519-02601-5.

Emil Fredriksson, Christian Andersson, and Johan Åkesson. Discontinuities handled with events in Assimulo. In Hubertus Tummescheit and Karl-Erik Årzén, editors, *Proceedings of the 10th International Modelica Conference*, number 96 in Linköping Electronic Conference Proceedings, pages 827–836. Linköping University Electronic Press, Linköpings universitet, 2014. URL `http://dx.doi.org/10.3384/ECP14096827`.

Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, September 2005. ISSN 0098-3500. doi:10.1145/1089014.1089020.

Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Comput. Chem. Eng.*, 34(11):1737–1749, November 2010. doi:http://dx.doi.org/10.1016/j.compchemeng.2009.11.011.

Master's Theses in Mathematical Sciences 2015:E49
ISSN 1404-6342

LUTFNA-3036-2015

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

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