Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface

by Christian Andersson

LUND UNIVERSITY

ACADEMIC THESIS

which, by due permission of the Faculty of Engineering at Lund University, will be publicly defended on Wednesday 4th of May, 2016, at 10.15 in lecture hall MH:A, at the Centre for Mathematical Sciences, Sölvegatan 18, Lund, for the degree of Doctor of Philosophy in Engineering.

Faculty opponent: Dr. Carol Woodward, Lawrence Livermore National Laboratory, USA

Organization	Document name
LUND UNIVERSITY Centre for Mathematical Sciences Box 118 SE-221 00 Lund Sweden	**DOCTORAL DISSERTATION IN MATHEMATICAL SCIENCES** Date of disputation May 2016
Author(s) Christian Andersson	Sponsoring organization

Title and subtitle
Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface

Abstract

Simulation of coupled dynamical systems, where each subsystem is bundled with an internal solver, is an important industrial method to support model-based design workflows. This is due to that in many cases, with complex systems, this is the only viable option in heterogeneous simulation landscapes where different parts of a system are modeled in different simulation tools. In this setting, the dynamics of each system is hidden and information between subsystems is exchanged through sampled inputs and outputs. This is often denoted as a weakly coupled system. While a new industrial standard for exchanging models, the Functional Mock-up Interface (FMI), gains increasing acceptance, the numerical consequences of treating complex systems in this way are not completely understood.

In this thesis, stability questions of weakly coupled linear systems with feed-through are studied. New methods, within the scope of the FMI, are proposed which offer improved stability properties compared to the classical approaches.

A simulation of a weakly coupled system introduces discontinuities due to input changes for the internal solvers. If the internal solver is a multistep method, these discontinuities will result in performance degradation. To avoid the degradation, a modification of the predictor in a multistep method is proposed achieving increased performance.

Furthermore, two Python packages are presented. The package PyFMI is a high-level package for working with models compliant with the FMI standard. PyFMI also contains co-simulation masters for simulation of weakly coupled systems. The package Assimulo unifies different integrators under a common interface which, together with PyFMI, provides an environment for using and evaluating solvers on industrial models. The packages are demonstrated by various examples ranging from simple test cases to a more extensive industrial application. Additionally, they have been used to verify the proposed methods and predictor modification.

Key words
Functional Mock-up Interface, FMI, Co-Simulation, Coupled Systems, Master Algorithm, Stability, Assimulo, PyFMI

Classification system and/or index terms (if any)

Supplementary bibliographical information	Language English
ISSN and key title 1404-0034	ISBN 978-91-7623-697-0(print) 978-91-7623-698-7(digital)

Recipient's notes	Number of pages 180	Price
	Security classification	

Signature _____ Date 29 March 2016

Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface

Christian Andersson

LUND UNIVERSITY

Faculty of Engineering
Centre for Mathematical Sciences
Numerical Analysis

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

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

Doctoral Theses in Mathematical Sciences 2016:3
ISSN 1404-0034

ISBN 978-91-7623-697-0 (print)
ISBN 978-91-7623-698-7 (digital)
LUTFNA-1010-2016

Printed in Sweden by Media-Tryck, Lund 2016

Abstract

Simulation of coupled dynamical systems, where each subsystem is bundled with an internal solver, is an important industrial method to support model-based design workflows. This is due to that in many cases, with complex systems, this is the only viable option in heterogeneous simulation landscapes where different parts of a system are modeled in different simulation tools. In this setting, the dynamics of each system is hidden and information between subsystems is exchanged through sampled inputs and outputs. This is often denoted as a weakly coupled system. While a new industrial standard for exchanging models, the Functional Mock-up Interface (FMI), gains increasing acceptance, the numerical consequences of treating complex systems in this way are not completely understood.

In this thesis, stability questions of weakly coupled linear systems with feed-through are studied. New methods, within the scope of the FMI, are proposed which offer improved stability properties compared to the classical approaches.

A simulation of a weakly coupled system introduces discontinuities due to input changes for the internal solvers. If the internal solver is a multistep method, these discontinuities will result in performance degradation. To avoid the degradation, a modification of the predictor in a multistep method is proposed achieving increased performance.

Furthermore, two Python packages are presented. The package PyFMI is a high-level package for working with models compliant with the FMI standard. PyFMI also contains co-simulation masters for simulation of weakly coupled systems. The package Assimulo unifies different integrators under a common interface which, together with PyFMI, provides an environment for using and evaluating solvers on industrial models. The packages are demonstrated by various examples ranging from simple test cases to a more extensive industrial application. Additionally, they have been used to verify the proposed methods and predictor modification.

Populärvetenskaplig sammanfattning

Utveckling av nya industriella produkter sker allt oftare med hjälp av datormodeller. Dessa modeller beskriver fysiken för respektive produkt, vilket exempelvis kan vara en bil eller ett kraftverk. Med hjälp av modellerna kan en stor del av analyserna på de framtida produkterna ske med hjälp av datorer, varför färre prototyper behöver konstrueras, vilket medför kortare utvecklingstider samt kostnadsbesparingar. En vanlig analys som behöver genomföras på en bil är exempelvis att undersöka hur bilen kommer att bete sig på varierande underlag eller vid olika manövreringar, dvs. man måste simulerar dess beteende. Utvecklingen mot att i högre grad använda datormodeller har pågått under en längre tid, men har accelererat de senaste åren.

I en komplex datormodell, som av en bil, är det vanligt att olika fysikaliska domäner, t.ex. mekaniken och elektroniken, modelleras separat. Dessa separata modeller skapas vanligtvis med hjälp av olika verktyg. För att sedan kunna analysera den komplexa bilmodellen måste de separata modellerna kopplas samman, vilket leder till svårigheter eftersom modellerna behöver kunna utbytas mellan olika verktyg som representerar modellerna på olika vis.

Med hjälp av en ny standard har emellertid utbytet av modeller mellan olika verktyg blivit lättare - eller helt enkelt möjligt. Att koppla ihop dessa modeller har lett till att det behövs nya algoritmer och fördjupad kunskap i hur man på bästa sätt simulerar dem tillsammans. Risken är dock att resultatet man får inte går att lita på eller att algoritmen misslyckas med simuleringen.

Till detta behövs lättillgänglig mjukvara med tillgång till algoritmer som är fördelaktiga för olika typer av modeller. Det här är inte bara viktigt för forskning och industrin utan även i undervisningssyfte.

I denna avhandling behandlas simulering av sammankopplade modeller, där modellerna följer den nya standarden. Fokus i avhandlingen har varit att analysera olika algoritmer, föreslår nya algoritmer, samt utveckla en mjukvara för simulering. Mjukvaran som har tagits fram har gjorts publik.

Acknowledgements[1]

During the course of this journey, there have been a number of people who have assisted and supported me, and without their help this thesis would never have happened. I would like to express my sincere gratitude towards my supervisors, Claus Führer and Johan Åkesson, who have guided me during this journey. Without their guidance, I would never have reached this point.

Also, I would like to thank my colleagues and fellow Ph.D. students, current and past, at Numerical Analysis, who have made our corridor into a place for open discussions. Thank you for creating a sharing and pleasant working atmosphere.

To my colleagues at Modelon, thank you all for the many interesting discussions and for putting things into a larger perspective. To Johan Andreasson for providing the example model of a race car which has been used extensively throughout this thesis.

Further, I'd like to thank the Centre for Mathematical Sciences, the Department of Automatic Control and Modelon for making a joint position possible.

I would also like to thank Anders Holmqvist with colleagues, at the Department of Chemical Engineering, for interesting and fruitful collaborations and for letting a mathematician walk around in the laboratory.

Furthermore, I would like to thank my family and friends for their support and for believing in my abilities even when I myself had doubts. To my father, Stefan Andersson, who always hinted that pursuing a Ph.D. might be something to aim for. To my love, Helena Sjöblom, who have put up with me during this time when I have not always been my happy self.

Thank you all.

Christian Andersson
Lund, Sweden

[1]This work was supported in part by the Lund Center for Control of Complex Engineering Systems (LCCC), funded by the Swedish Research Council, which is gratefully acknowledged. Furthermore, the work was supported in part by Modelon, which is gratefully acknowledged.

Notation

x	The global state vector
x_i	The ith global state
$x^{[i]}$	The state vector of model i
$x_j^{[i]}$	The jth state from state vector of model i
u	The global input vector
y	The global output vector
H	The global step size
h	The local step size
$f(x, u)$	The global derivative function
$g(x, u)$	The global output function
$c(y)$	The coupling function
$\Phi(\cdot)$	General function for performing a step in time
$\Psi(H)$	Iteration matrix performing a step of the coupled system

Contents

Chapter 1

Introduction

Different simulation and modeling tools often use their own definition of how a model is represented and how model data is stored. Complications arise when trying to model parts in one tool and importing the resulting model in another tool, or when trying to verify a result by using a different simulation tool. The *Functional Mock-up Interface (FMI)* [50] is a standard to provide a unified model execution interface for exchanging dynamic system models between modeling tools and simulation tools. A model that follows the FMI is called a *Functional Mock-up Unit (FMU)*. The standard has gained widespread adoption among users and numerous commercial and open source tools[1] implement support for the standard. In the standard's footsteps, a great deal of attention has been on simulation of dynamic system models, and specifically simulation of coupled dynamic systems.

This thesis focuses on simulation of dynamic and coupled dynamical systems in relation to the FMI standard, either directly or indirectly. Emphasis has been on simulation of weakly-coupled systems (cf. Section 1.1). Furthermore, software has been developed for simulation and simulation of weakly-coupled systems (cf. Section 1.2) in relation to the FMI standard.

1.1 Coupled systems

There is a strong tradition among domain experts to use specialized modeling and simulation environments for component models. These tools are also favored as they usually offer larger libraries of components and domain specific features than a multi-domain tool. However, problems arise when trying to investigate a coupled system model, i.e. when component models from various tools need to be coupled, as there is no standard way of coupling the components. The problem is not only that of multi-domain modeling, it may also be that the know-how of a component model needs to be protected. Thus, the problem is still how to connect the component models into a monolithic simulation model and

[1] https://www.fmi-standard.org/tools [accessed: 2016-03-18]

perform the necessary evaluations to compute the solution profiles. In Figure 1.1, a typical situation for a system with different domains is shown where the components are coupled together.

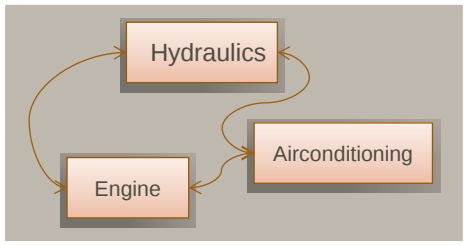

Figure 1.1: *A schematic figure of a multi-domain coupled system.*

Coupling the components together into a monolithic model can be performed via two different approaches, *strong*-coupling or *weak*-coupling. Given that the model exposes its internal dynamics, i.e. that it is possible to directly evaluate the model equations, one can easily assemble these equations which in turn can be solved by standard time integration algorithms. This approach is commonly known as strong-coupling. However, if the model equations are not exposed, but instead hidden behind an interface with the only options to set the inputs and retrieve the outputs, cf. Figure 1.2, the strong-coupling is no longer feasible. In these cases, one resorts to weak-coupling, where the separate models contain an internal integrator and the information exchange between the connected models, via inputs and outputs, is only performed at specified communication points. This approach has several benefits as it allows tailored integrators for component models. In addition, it allows the component models to be run in parallel. Moreover, the component models may have widely different time scales, which, by using the weak-coupling, can be exploited by the internal integrator. This becomes evident when comparing electrical components to multibody components. However, a weakly-coupled system also introduces difficulties, for example how the information exchange should be performed in order for a stable simulation.

Figure 1.2: *A schematic figure of an input / output model.*

This thesis focuses on simulation of weakly coupled systems. In Chapter 3, a background is given into the field of simulation of weakly coupled dynamical systems. Part II focuses on theory of simulation of weakly coupled systems, both regarding initialization and simulation as well as discuss novel results. Furthermore, the part focuses on the underlying solvers and discusses an efficient restart approach for multistep methods in a weakly

coupled systems context.

1.2 Simulation software

During the last three decades, a vast variety of methods to numerically solve ordinary differential equations and differential algebraic equations has been developed and investigated. The methods are mostly freely available in different programming languages and with different interfaces. Accessing them using an unified interface is a need not only of the research community and for education purposes, but also to make them available in industrial contexts.

An industrial model of a dynamic system is usually not just a set of differential equations. Instead, the models today may contain discrete controllers, impacts or friction, resulting in discontinuities that need to be handled by a modern solver in a correct and efficient way. In addition, the models may produce an enormous amount of data that puts strain on the simulation software.

Due to the above, and to the FMI standard, software was developed during the thesis. In Chapter 4, background of the developed simulation software is given. Part III, introduces the developed software, Assimulo and PyFMI. The first contains solvers for dynamic systems and the second is aimed at working with systems following the FMI standard, both regarding simulation and simulation of weakly coupled systems. Essentially, Assimulo provides the solvers, while PyFMI provides the problems. The software implements the algorithms discussed in Part II.

1.3 Contributions

The contributions of this thesis are two-fold. One part is the developed open-source software framework for simulation and simulation of weakly coupled systems together with software for working with FMUs. The software are,

- *Assimulo*
- *PyFMI*
    - *The Master Algorithm*

Assimulo is a simulation package for solving ordinary differential equations containing various different solvers. The primary aim of Assimulo is to provide a high-level interface for a wide variety of solvers rather than to develop new integration algorithms. PyFMI is a package for interacting with models adherent to the FMI standard. It is designed to provide a high-level, easy to use, interface for working with FMUs. Further, PyFMI contains the developed master algorithm for simulation of weakly coupled systems.

In addition to the software framework, co-simulation in general with focus on stability and on the underlying integrator is also discussed in this thesis. Specific contributions,

- *Analysis of the stability of the parallel co-simulation approach.*
- *Developed and implemented a linear correction algorithm.*
- *Efficient restart of multi-step methods at input changes.*

1.3.1 Publications

The thesis is based on the below mentioned publications.

> **C. Andersson**, C. Führer and J. Åkesson. "Efficient Predictor for Co-Simulation with Multistep Sub-System Solvers". *Preprint Math. Sci. Lund Uni. (2016)*.

> **C. Andersson**, J. Åkesson and C. Führer. "PyFMI: A Python package for simulation of coupled dynamic models with the Functional Mock-up Interface". *Preprint Math. Sci. Lund Uni. (2016)*.

> **C. Andersson**, C. Führer and J. Åkesson. "Assimulo: A unified framework for ODE solvers". *Math. Comput. Simul. (2015)*.

The first publication introduces a novel approach for restarting a multistep method at input changes. The second introduces the PyFMI package for working with and analyzing FMUs while the third publication introduces the Assimulo package. The first author has been the key contributor of the ideas, the case studies and the development, together with being primarily responsible for drafting the manuscript.

> **C. Andersson**. "A Software Framework for Implementation and Evaluation of Co-Simulation Algorithms". *Lic. Theses Math. Sci. (2013)*.

The licentiate thesis by the author describes and introduces many of the parts on which this thesis expands upon.

> A. Holmqvist, **C. Andersson**, F. Magnusson, J. Åkesson. "Methods and Tools for Robust Optimal Control of Batch Chromatographic Separation Processes". *Processes (2015)"*.

This publication extends and utilizes the developed software, PyFMI and Assimulo, with support for solving Lyapunov equations on a chromatographic separation process used in the chemical industry. The author has contributed ideas for solving the equations, implemented support for solving the equations and written part of the manuscript related to simulation and computation of the Lyapunov equations.

1.3.1.1 Related

The following publications are related to the work by the author, either by being building blocks to the above publications or by being within the scope of the developed software.

> P. Pannu, **C. Andersson**, C. Führer and J. Åkesson. "Coupling Model Exchange FMUs for Aggregated Simulation by Open Source Tools". *Proc. 11th Int. Modelica Conf. (2015)*.

This publication is related to simulation of strongly coupled systems, as opposed to weakly coupled systems, where models not only follow the FMI but also models implemented directly for Assimulo. For this publication the author has contributed ideas and assisted with the implementation and with the manuscript.

> E. Fredriksson, **C. Andersson**, J. Åkesson. "Discontinuities handled with events in Assimulo, a practical approach". *Proc. 10th Int. Modelica Conf. (2014)*.
>
> **C. Andersson**, J. Andreasson, C. Führer and J. Åkesson. "A Workbench for Multibody Systems ODE and DAE Solvers". *2nd Jnt. Int. Conf. Multibody Syst. Dyn. (2012)*.

The above publications are related to the developed Assimulo package where the second publication provides the first building blocks for Assimulo while the first publication investigates and improves on the event capabilities for a number of attached integrators. For the first publication, the author has contributed ideas and assisted with the implementation and with the manuscript. For the second publication, the author has contributed with software implementation of the workbench, and the evaluation on test models together with being primarily responsible for drafting the manuscript.

> S. Gedda, **C. Andersson**, J. Åkesson and S. Diehl. "Derivative-free Parameter Optimization of Functional Mock-up Units". *Proc. 9th Int. Modelica Conf. (2012)*.

This publication investigates parameter estimation using FMUs and implements support for the estimations within PyFMI. For this publication, the author has assisted in improving on the software implementation and in drafting the manuscript.

> **C. Andersson**, J. Åkesson, C. Führer and M. Gäfvert. "Import and Export of Functional Mock-up Units in JModelica.org". *Proc. 8th Int. Modelica Conf. (2011)"*.

This publication provides the first building blocks for the developed PyFMI package. The author has contributed with software implementation for import of FMUs, evaluation of the same, and has been primarily responsible for drafting the manuscript.

Part I:

Background

Overview

In this part we discuss the background to the thesis. In Chapter 2, the Functional Mock-up Interface standard is introduced, which this thesis is highly related to, either directly or indirectly. In Chapter 3, simulation of weakly coupled systems is introduced together with a background into the field of simulation of weakly coupled systems. In Chapter 4, a motivation and background is given for the developed software.

Chapter 2

Functional mock-up interface

The Functional Mock-up Interface (FMI) [50] is a standard designed to provide a unified model execution interface for dynamic system models between modeling tools and simulation tools. The idea is that tools generate and exchange models that adhere to the FMI specification. Such models are called Functional Mock-up Units (FMUs). This approach enables users to create models in one modeling environment, connect them in a second and finally simulate the complete system using a third simulation tool, cf. Figure 2.1.

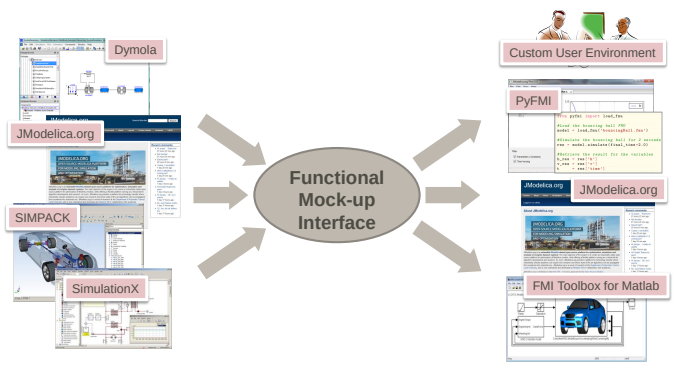

Figure 2.1: *Exchange of dynamical models following the Functional Mock-up Interface.*

The generated models, FMUs, are distributed and shared as compressed archives. They include either the source files for the model, allowing a user full access to the internals, or a shared object file containing the model information which is accessed through the FMI interface. Furthermore, both the source files and the shared object file can be included in the FMU. The archive additionally provides an XML file containing metadata of the model, such as the sizes of the dynamic system and the names of the variables, parameters, constants and inputs. In the archive, there can also be additional information, which does not impact a simulation of the model, but may be of interest to distribute with the FMU, for example documentation.

FMI was developed in a European project, MODELISAR[1], that was focused on improving the design of systems and of embedded software in vehicles. The standard is now maintained and developed by the Modelica Association[2].

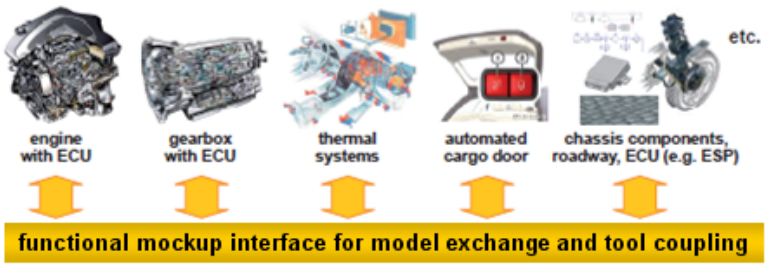

Figure 2.2: *Functional Mock-up Interface*[3].

Since its release, the standard has received a significant amount of attention among both tool vendors and users. There are currently over 70 tools[4] that support or plan to support FMI. Examples include the commercial products Dymola [20] and SIMPACK [67], as well as the open-source platform JModelica.org [62]. The large number of tool vendors that have adopted the standard shows that there is a real and pressing need to be able to export and import dynamic system models between existing tools, and also to be able to develop custom simulation environments.

FMI specifies two types of models, one named a model exchange FMU, cf. Section 2.1, and the other named a co-simulation FMU, cf. Section 2.2.

2.1 Model exchange

For model exchange, the standard describes an interface for discontinuous ordinary differential equations, with means to set the continuous states and time as well as evaluating the model equations, i.e. the right-hand-side, and specifying inputs.

The standard describes a model as,

$$\dot{x} = f(t, x, u; d, p) \tag{2.1a}$$
$$y = g(t, x, u; d, p) \tag{2.1b}$$

where t is the time, x are the continuous states, u are the inputs, d are the discrete variables that are kept constant between events and p are the parameters. Furthermore, y are the outputs. Additionally, the standard supports three kinds of events that can impact the model behavior. The three events are:

[1]https://itea3.org/project/modelisar.html [accessed: 2016-03-11]
[2]https://www.modelica.org/association [accessed: 2016-03-26]
[3]©Modelica Association
[4]https://www.fmi-standard.org/tools [accessed: 2016-03-18]

- *State Events* depends on the state solution profiles and are thus not known a priori. The model provides a set of event indicators, z, that the integrator monitors during the integration process,

$$z = h_{\text{state}}(t, x, u; d, p). \tag{2.2}$$

  If one of the event indicators, z_i, switches domain, there is a state event. The integrator is then responsible for finding the time when the event occurred.

- *Time Events* are known a priori, meaning that for each simulation segment it is known when the time event occurs and thus this time is set as the simulation end time for that segment. Given a previous time event, T_{pre} (or the initial time, T_0), the next time event is computed using,

$$T_{\text{next}} = h_{\text{time}}(t_{T_{\text{pre}}}, x_{T_{\text{pre}}}, u_{T_{\text{pre}}}; d_{T_{\text{pre}}}, p). \tag{2.3}$$

  An example is that after a certain elapsed time in the integration, a force is applied on the model.

- *Step Events* are events that typically do not influence the model behavior, instead they are events to ease the numerical integration. For instance they can be used for re-parameterization of a model. After each successful integrator step, T_{accepted}, the equation,

$$E_{\text{step}} = h_{\text{step}}(t_{T_{\text{accepted}}}, x_{T_{\text{accepted}}}, u_{T_{\text{accepted}}}; d_{T_{\text{accepted}}}, p) \tag{2.4}$$

  is evaluated and if E_{step} is True, a step event is triggered.

Further, Equation 2.1 is valid during continuous simulation and priori to this, the FMI specifies that the FMU need to be initialized. The simulation and initialization is separated as to allow a flexible definition of initial conditions. An example could be that the initial values for the states are computed using an initial equation, which is only active during the initialization. The initial equations are described by,

$$\hat{x}_0, \hat{d}_0, \hat{p} = f_{\text{init}}(t_0, \bar{x}_0, u_0; \bar{d}_0, \bar{p}), \tag{2.5}$$

where $\bar{x}_0$ are states with known initial values, $\bar{d}_0$ are the known discrete variables and $\bar{p}$ are known parameters. The complete initial states vector is $x_0 = [\bar{x}_0, \hat{x}_0]$, and for discrete variables, $d_0 = [\bar{d}_0, \hat{d}_0]$, while the full parameter vector is $p = [\bar{p}, \hat{p}]$.

For full details about the mathematical representation, cf. [50] for version 2.0 and [49] for version 1.0.

Simulating a model exchange FMU requires that an external integrator is connected to the FMI model, cf. Figure 2.3.

Figure 2.3: *A model exchange FMU and the connection to a tool for simulation. Note that the solver is outside of the FMU.*

2.2 Co-Simulation

For co-simulation, the standard rather describes a discrete interface to the underlying dynamic model, i.e. given the current internal state, input u_n and step size H of the model, return the outputs, y_{n+1}, at a time $T_n + H = T_{n+1}$,

$$y_{n+1} = \Phi(H, u_n; p), \tag{2.6}$$

where p are the parameters. The advancement of the states and time is completely hidden outside of the model and is also not specified by the standard, cf. Figure 2.4. A consequence of this is that, if there are events, these are also handled internally and are not visible from the outside. However, as the advancement is hidden, this allows for specialized solvers to

Figure 2.4: *A co-simulation FMU and the connection to a tool for simulation. Note that the solver is inside the FMU.*

be used for the particular subsystem at hand, which may give an increased performance and a more stable simulation. As in the model exchange case, Section 2.1, the initialization is done separately for co-simulation FMUs. The initialization is defined by,

$$\hat{p} = f_{\text{init}}(t_0, u_0; \bar{p}), \tag{2.7}$$

where $\bar{p}$ are known parameters. The full parameter vector is $p = [\bar{p}, \hat{p}]$.

For full details about the mathematical representation, cf. [50] for version 2.0 and [48] for version 1.0.

2.3 Features and restrictions

In the FMI standard there are a few key features that have been exploited and restrictions that have in some cases limited what co-simulation schemes can be realized. These are

explained in this section. Additionally, between version 1.0 and version 2.0 of the standard, changes have been made and features have been added. All of the features mentioned here are optionally available.

In FMI 2.0, a feature for saving and restoring the internal state of a FMU has been added, Feature 2.1.

Feature 2.1 (Save/ Get state). *Methods for retrieving and restoring the internal state of a FMU have been added.*

Support for allowing restoring and saving the FMU state makes it possible to restart a simulation from a previous point. Saving and restoring the internal state is, for example, important in a co-simulation master when simulating a coupled system using an error controlled algorithm as it allows for executing a step with different options and for rejecting steps.

Another added feature is directional derivatives.

Feature 2.2 (Directional derivatives). *Methods for computing the directional derivatives of Equation 2.1 and Equation 2.6 has been added.*

The addition of directional derivatives allow for computing the partial derivatives of the functions f and g in Equation 2.1 with respect to x and u, i.e.,

$$\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial u}, \quad \frac{\partial g}{\partial x}, \quad \frac{\partial g}{\partial g}, \tag{2.8}$$

and the partial derivatives of Φ in Equation 2.6 with respect to u,

$$\frac{\partial \Phi}{\partial u}. \tag{2.9}$$

With FMI 1.0, when simulating a FMU with a method requiring the Jacobian of Equation 2.1, the only option was to compute it using finite differences. With the directional derivatives available the Jacobian can be computed analytically, if the FMU supports the feature.

Furthermore, the dependency information was extended in FMI 2.0.

Feature 2.3 (Dependency information). *Information about which states and inputs directly impact the derivatives and outputs in Equation 2.1 has been added. Further, information about which inputs directly impact the outputs in Equation 2.6 has been added.*

With the dependency information, the sparsity pattern for the Jacobian can be defined and thus also allow for using sparse solvers. In FMI 1.0, only the dependency between inputs and outputs was available.

Changes have also been made to the initialization, cf. Feature 2.4.

Feature 2.4 (Separate initialization). *The initialization of an FMU has been separated into a state of the FMU instead of a single call to an initialization method.*

The below mentioned features and restrictions are related to co-simulation FMUs. Advancing the solution to the next communication point in a co-simulation FMU is performed using the `do_step` method. At a communication point there is a restriction for the update of outputs, given inputs.

Restriction 2.1. *The outputs, y, cannot be evaluated, during simulation, for different inputs, u, without advancing the solution time, i.e. performing a step with $H > 0$.*

Remark 2.3.1. *During initialization, i.e. at t_0, Restriction 2.1 does not apply.*

At a communication point, values inside the model can be retrieved and inputs can be set. In addition, there is a feature allowing higher order derivatives to be set.

Feature 2.5. *Higher order derivatives for the inputs, u, can be set to a co-simulation FMU at communication points.*

The input derivatives are represented by a vector,

$$\left[\frac{du}{dt}(T_n), \frac{d^2u}{dt^2}(T_n), \frac{d^3u}{dt^3}(T_n), \ldots, \frac{d^ku}{dt^k}(T_n)\right]. \tag{2.10}$$

The input is evaluated during the next global integration step by a truncated Taylor series expansions,

$$u(t) = u(T_n) + \sum_{i=1}^{k} \frac{1}{i!}\frac{d^iu}{dt^i}(t - T_n)^i, \quad t \in [T_n, T_{n+1}]. \tag{2.11}$$

Furthermore, the standard supports that higher order derivatives for the outputs, y, can be retrieved.

Feature 2.6. *Higher order derivatives for the outputs, y, can be retrieved from a co-simulation FMU at communication points.*

Chapter 3

Coupled systems

Simulation of weakly coupled systems, commonly co-simulation, is about how to simulate two or more dynamic systems that are connected. The systems are described as being discrete on the interface level, meaning that the transition from a time T_n to a time T_{n+1} is done internally for each system. The solver used to make this transition is usually unknown in a co-simulation scenario. This means that domain specific integrators can be used, which may have a superior performance when compared to a general purpose integrator.

One distinguishes between performing a *global* integration step (or *macro-step*) and a *local* integration step (*micro-step*). A global integration step is the transition of the system model from T_n to T_{n+1} while the local integration steps, $t_{n,m}$, are the steps taken by the internal solver in each subsystem, $T_n = t_{n,0} < t_{n,1} < \ldots < t_{n,m} = T_{n+1}$.

As an example of a co-simulation scenario, consider the following equation,

$$\dot{z} = Az \tag{3.1}$$

where A is a 2×2 matrix. Decoupling the system into two separate subsystems with the first being,

$$\dot{x}^{[1]} = a_{11}x^{[1]} + a_{12}u^{[1]} \tag{3.2a}$$

$$y^{[1]} = x^{[1]} \tag{3.2b}$$

where $x^{[1]}$ is the state, $u^{[1]}$ is the input and $y^{[1]}$ are the output. The superscript $[1]$ specifies the first subsystem. The second subsystem is similarly defined,

$$\dot{x}^{[2]} = a_{22}x^{[2]} + a_{21}u^{[2]} \tag{3.3a}$$

$$y^{[2]} = x^{[2]}. \tag{3.3b}$$

In a co-simulation approach these two systems use their own internal integrator for solving the differential equation. The first could for instance be solved with the Implicit Euler

method, while the second could be solved with the Explicit Euler method. However, this is usually unknown and the only interactions with other subsystems are done through the inputs and the outputs.

The interactions are specified via coupling equations which in this case are,

$$u^{[1]} = y^{[2]} \tag{3.4a}$$

$$u^{[2]} = y^{[1]}. \tag{3.4b}$$

These coupling equations are in a sense "outside" the individual subsystems. The following questions arise:

- How does the decoupling impacts the stability of the overall system?
- How should the exchange of information between the two systems be performed?
  - By extrapolation of the inputs?
  - By introducing an ordering so that some signals are interpolated while some are extrapolated?

The most commonly used co-simulation method is to simulate the models in parallel and at predefined global time points exchange information. The inputs in between the global steps are then kept constant, cf. Figure 3.1.

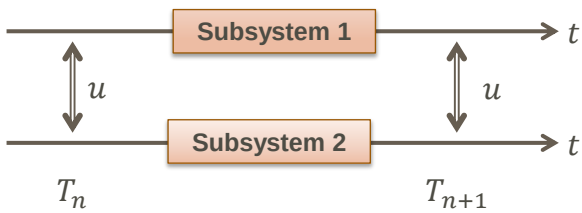

Figure 3.1: *Schematic figure of two subsystems that are executed in parallel with data exchange at predefined global time points.*

An algorithm for determining the exchange of information, the type of extrapolation/interpolation, the ordering and all information related to a simulation of the coupled system is called a *Master Algorithm*.

In this thesis, we consider N coupled systems of the type,

$$\dot{x}^{[i]} = f^{[i]}(t, x^{[i]}, u^{[i]}), \quad i = 1, \ldots, N \tag{3.5a}$$

$$y^{[i]} = g^{[i]}(t, x^{[i]}, u^{[i]}), \quad i = 1, \ldots, N \tag{3.5b}$$

$$u = c(y) \tag{3.5c}$$

if $g^{[i]}$ is depends on $u^{[i]}$, i.e. $(\frac{\partial g^{[i]}}{\partial u^{[i]}} \neq 0)$ we say that the subsystem is a feed-through system, i.e. the output $y^{[i]}$ directly depends on the input $u^{[i]}$. The function c determines the coupling between the systems.

Assumption 3.1. *A weakly coupled system of co-simulation FMUs, Equation 2.6, is described by Equation 3.5.*

Co-Simulation may also be referred to as *modular simulation* and the separate dynamic systems are sometimes refereed to as *slaves*. In this thesis, we reserve the name subsystem for a separate dynamic system and co-simulation for a simulation of coupled dynamic systems.

3.1 Previous work

Dividing a set of equations into subsystems and solving them separately has been discussed for decades and dates back to the late 1970s. In [8], solution of a set of equations divided into subsystems of "fast" and "slow" components is considered. The discussion centers on the benefits and performance gains of using different time scales in the integrator for the components. These type of methods are called multi-rate methods and a lot of research has been performed in this area. In [28], improvements for automatic step size selection of multi-rate methods for linear multistep methods is discussed.

The difference between multi-rate methods and methods for co-simulation is that in the latter case, the equations are not exposed directly and the integrator responsible for solving the system is hidden and unknown. In co-simulation, the subsystems are essentially black boxes with inputs and outputs.

In [23], an overview of co-simulation approaches is given. The parallel and staggered scheme is explained together with more sophisticated schemes. The parallel scheme is basically to let the subsystems simulate the same global time step, and once all subsystems are finished, exchange information between them. Using this approach, the multi-core nature of todays processors can easily be exploited for improving the simulation efficiency. The staggered scheme on the other hand, requires an ordering, i.e. the first subsystem is solved for a global time step. Once completed, the next subsystem is simulated over the same global time step.

In [40, 41], co-simulation is discussed from the point of view of block representation where the blocks contained the internal dynamics of a subsystem, inaccessible from the outside. Here, a block interacts with another block via its inputs and via its outputs through coupling equations outside of the blocks. The blocks are represented in a general state-space formulation, cf. Equation 3.5*a* and Equation 3.5*b*, which is widely used in control theory. The coupling equation, Equation 3.5*c*, is here assumed to be linear, $u = Ly$. The articles centers around stability issues when constant extrapolation for the inputs is used. In addition, the articles covers the cases when there is direct feed-through and when there is not. Constraints on the feed-through is highlighted in order to guarantee a stable simulation of the coupled system. The articles later served as a foundation for the definition of the FMI standard for co-simulation [48].

The release of the FMI standard triggered a renewed interest in co-simulation, especially in industry. The potential of coupling state of the art modeling tools using a standardized

format and being able to utilize each tools strength was met with much interest. In [1], it is shown that the multi-domain environment, SimulationX [38], and the multibody environment SIMPACK can be coupled together using FMI components. The application is to simulate a power-train of a heavy-duty truck where parts is modeled independently in the separate tools and then coupled together for analysis.

The interest from industry, regarding co-simulation, is not only triggered by the coupling of the environments but also by the potential efficiency gain of decoupling a large system model. This is exemplified in [34] where a model of an engine is decoupled into subsystems. By decoupling the chain drive into a subsystem, a decrease of the simulation time was achieved by an order of magnitude.

Research on the stability issues when using co-simulation has been active in recent years. In [12], stability of coupled differential algebraic equations are discussed and a contractivity condition is formulated that must be fulfilled in order to guarantee a stable error propagation. Stabilization of these systems is further discussed in [9, 10] where the Jacobian information is utilized for performing a stable integration when using a sequential algorithm. In [71], a stabilization technique was implemented that did not take into account loops on the coupling variables. An additional technique was proposed in [64], which was based on applying the constraint equations in a differential algebraic equation to more than one subsystem. In [66] a predictor - corrector approach was considered which required that the global steps had to be done twice.

The common approach used in industry is to use constant extrapolation and manually tune the global integration step size until the coupled system produces "satisfactory" results. This is a costly and time consuming approach, but has been known to work in practice. Improving the situation requires that an error estimation procedure is developed so that the step size can be automatically tuned according the local integration error. In [65, 63], an error estimation procedure for coupled systems was proposed. The error estimate was based on Richardson extrapolation and the assumption that the subsystems were integrated exactly. In an engineering setting, this can be achieved by requiring higher accuracy on the subsystems. The idea is that a global step is performed twice with step size H and $H/2$ following a comparison of the two results.

Another technique used for co-simulation is the *Transmission Line Modeling (TLM)* which introduces delays between the subsystems in order to decouple the problem. The delays that are introduced change the models and introduce errors, but on the other hand, this delay can usually be physically motivated and the error can be dealt with explicitly. In this thesis, TLM will not be covered, cf. [26].

Chapter 4

Simulation software

The simulation software developed in this thesis are the Python packages Assimulo and PyFMI.

Assimulo is a simulation package for solving ordinary differential equations containing various solvers, both state-of-the-art and more experimental ones. The primary aim of Assimulo is to provide a high-level interface for a wide variety of solvers rather than to develop new integration algorithms. Furthermore, the aim is to allow comparison of solvers for a given problem without the need to define the problem in different programming languages to accommodate the different solvers. The software is designed to satisfy the needs in research and education, as well as the requirements for solving industrial models with discontinuities and data handling. Activities on unifying interfaces of ODE software started already long before Assimulo and before object oriented programming became a paradigm. One such activity led to the early Fortran package ODEPACK, [32].

Assimulo was originally developed due to the need for an educational tool for students in numerical analysis, engineering and physics, which provided an easy and unified access to industrial quality ODE solvers, like Sundials [33]. Without Assimulo, students would need for this purpose knowledge in C or Fortran together with more advanced programming skills. A tool in a programming language which is used in the entire education together with a unified interface to many different integrators gives the students more time for numerical experiments which otherwise would be used for mere programming.

PyFMI grew from [2] where there was a need for working with the FMI standard in the open source tool JModelica.org. The software is designed to provide a high-level, easy to use, interface for working with FMUs. It connects the full set of methods in the FMI specification in an object-oriented approach. The package is not only a mapping of the FMI interface to Python, it provides much of the functionality needed to perform various experiments for both evaluating the complex dynamical system model by itself but also for evaluating the physics that the model represents. The evaluation of the model could be to verify the model dynamics by efficient simulation while evaluations of the physics could be to performing parameter estimations. These experimentations requires an extensive tool

beyond the low-level FMI interface which motivates the package. Furthermore, with the FMI, simulation of coupled models in a co-simulation setting is possible and within PyFMI a master algorithm has been implemented and made available.

In order to promote widespread use of the FMI standard and make it easily accessible, there is a need for an open package in an open platform for experimenting and working with FMUs which is what PyFMI offers. The package offers an open platform for working with FMUs and the algorithms that are included are open and accessible for modifications and further experimentations. In addition, it includes an open and available master algorithm for simulation of coupled FMUs. In a related work, PySimulator [60], there is also support for working with FMUs from Python. In their case they use a different approach for coupling the FMUs to Python and are more focused on post processing of simulation results. Furthermore, there is no included master algorithm.

PyFMI is commonly used together with Assimulo [4]. The packages complement each other as Assimulo provides the solvers for solving dynamical systems, such as those represented by FMUs, and PyFMI provides the problems. Further, this resulted in that Assimulo became a central part of the software project JModelica.org, together with PyFMI.

The software developed is written in the programming language Python which is a powerful dynamic programming language with a clear and readable syntax. The choice of using Python is highly influenced by the ability and the ease of connecting software written in different programming languages, such as C or Fortran, and the ability of using Python as *glue*. Another aspect is that due to the clear and readable syntax of Python a user can easily create own scripts as the threshold is relatively low for learning the language compared with the low-level languages C and Fortran, especially if the user has a background in MATLAB [45]. Python is also a good choice for prototyping as there are specialized packages for scientific computing and for visualization, much like MATLAB. Moreover, in the area of scientific computing, Python has gained a great deal of momentum due to many freely available packages, notably NumPy and SciPy [55], but also due to the fact that the language is easy to learn. By providing an interface for working with FMUs from Python, the model is exposed to the full ecosystem that Python has to offer. Visualization and animations of simulation results can be done through matplotlib [37] and the flexibility that Python offers make it suitable for prototyping.

The core of the packages is implemented in Cython [14] which is a static compiler for Python. It allows mixing the programming languages C and Python interchangeably. The benefit of mixing the languages is that the main part of the package, where readability and scripting functionality matter, is based on Python, and performance critical parts are kept in C. In this way, computational performance is preserved, as opposed to if the package was solely relying on Python.

4.1 Integrator museum

Research on numerical methods for solving explicit ODEs and implicit ODEs, in particular differential algebraic equations (DAEs), produced dozens of academically approved algorithms and related codes. Most of them were never tested under industrial conditions and were soon forgotten. Together with the development of Assimulo, an activity, the ODE museum, started with the intention of providing access to these codes. Wrapping them by a unified interface makes it possible to apply such codes to highly sophisticated simulation problems and to gain insight in the performance of these approaches.

A starting point for the museum was DASP3 [69], one of the first DAE codes dating back to 1972.

Museum codes often do not have the same test level as the other codes. Primarily, this is due to the fact that these codes were developed as a proof-of-concept and tested on a limited number of problems. Also, recovering codes from a print medium by an OCR scan is a source of errors.

Current work is devoted on the incorporation of extrapolation methods, which were the focus of intense research activities in the 70ths and the 80ths, early DAE general purpose methods and those specialized on mechanical systems, such as MEXX [44], ODASSL [27] and DAESolve [61]. Also methods of the SDIRK type [42] and Rosenbrock methods are planned to be incorporated in near future.

Part II:

Coupled systems

Overview

In this part we consider the theoretical aspects related to simulation of weakly coupled systems. We cover both simulation and initialization of the coupled systems as well as discuss improvements to the underlying solvers in the subsystems, in case of a multistep method is used. Initialization and simulation of the coupled systems has been divided into two chapters, mainly due to Restriction 2.1. The initialization chapter, Chapter 5, focus on computing a consistent initialization via a structural analysis of the coupled system. The simulation chapter, Chapter 6, analyses different algorithms for simulation of the coupled systems and determine requirements for when we have a stable simulation. The chapter also introduces new approaches. In Chapter 7, a modification of multistep methods is proposed for cases when these methods are used internally in the subsystems which increase the performance of the methods.

Chapter 5

Initialization

Coupling models together in a co-simulation approach leads in general to an algebraic system of equations that needs to be solved in order to start the simulation from a consistent initial state, cf. Equation 5.1,

$$y = g(x, u) \tag{5.1a}$$
$$u = c(y) \tag{5.1b}$$

for u and y with x fixed. Any nonlinear solver, such as Newton's method, can be applied to the problem, although with the issue of finding a good initial guess. Without a good initial guess, there is a risk of a potential break down of the method.

In some cases, however, solving the arising equations may be unnecessary. Consider Figure 5.1. The illustration shows three coupled models where each has direct feed-through, shown in red, which indicates that a loop is present.

Definition 5.0.1 (Direct feed-through). *Let $y^{[i]} = g^{[i]}(x^{[i]}, u^{[i]})$ be the outputs from model i. If $y^{[i]}$ depends on the inputs $u^{[i]}$ then model i has direct feed-through.*

Definition 5.0.2 (Algebraic loop). *Consider Equation 5.1. If, no explicit evaluation sequence can be found to compute the outputs, y, then the coupled system has an algebraic loop.*

Tracking the evaluation from the external input to the external output, a path can be found that contains no loops. This means that the models can be evaluated in sequence in order to compute the external output and thus no equation system needs to be solved. This is referred to as a coupling induced loop.

Remark 5.0.1 (Coupling induced algebraic loop). *Consider Equation 5.1. If any model has direct feed-through and if there is no algebraic loop in the coupled system, then the system has a coupling induced algebraic loop.*

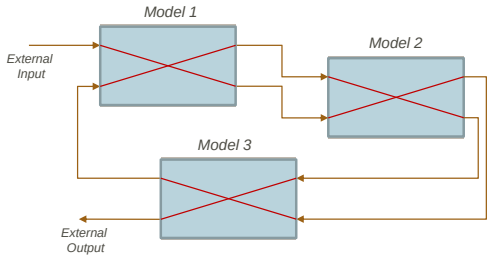

Figure 5.1: *A coupled system where it is possible to find a path from the external input (upper left) to the external output (lower left) without the need for solving the algebraic equation.*

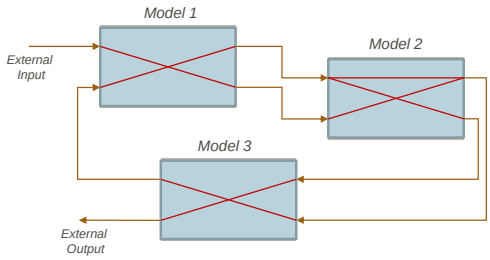

Figure 5.2: *A coupled system where it is not possible to find a direct path from the external input (upper left) to the external output (lower left) without the need for solving the algebraic equation. Notice the additional dependency on the outputs in Model 2.*

There are also situations where a subset of models introduces an algebraic loop and needs to be solved together while another subset can be evaluated in sequence. An illustration is shown in Figure 5.2. Consider Example 5.0.1, where a straightforward approach based on solving the resulting nonlinear system fails due to a poor initial guess.

Example 5.0.1 (Initialization failure). *Consider two coupled models defined by,*

$$\dot{x}^{[1]} = x^{[1]} + u^{[1]}, \qquad \dot{x}^{[2]} = x^{[2]} + u^{[2]} \tag{5.2a}$$

$$y^{[1]} = x^{[1]}, \qquad y^{[2]} = \frac{x^{[2]}}{u^{[2]}} \tag{5.2b}$$

with $x^{[1]}(t_0) = 1$ *and* $x^{[2]}(t_0) = -1$. *The coupling is defined by,*

$$u^{[2]} = y^{[1]}, \quad u^{[1]} = y^{[2]}. \tag{5.3a}$$

Assembling the coupled system and solving the resulting equations, Equation 5.1, using a straightforward approach by employing Newton's method will fail. The reason is that if no information is available for the starting values provided to Newton's method, commonly zero is used. Using zero as starting values, for the inputs $u^{[1]}$ *and* $u^{[2]}$*, will result in a division by zero in the expression* $\frac{x^{[2]}}{u^{[2]}}$*. Note that if other starting values are used, the example models can be trivially*

modified to highlight the same issue, without appropriate starting values the straightforward approach might fail.

The initial system can be analyzed by considering a graph relating the inputs to the outputs of the coupled system. In FMI, there is the possibility to provide this structural information (Feature 2.3). In Section 5.1, this is further discussed.

5.1 Structural analysis

The aim of structural analysis is to solve the initialization problem, Equation 5.1, by finding an evaluation order of the u_i's. The evaluation order can either be a direct sequence, i.e. the equations can be solved by a forward evaluation, or that a subset of equations needs to be solved together.

This section considers the initialization problem via graph theoretical concepts. In order to understand graphs and operations on graphs described in this thesis, only the necessary concepts are introduced. For a thorough introduction on graphs, cf. [16].

The initialization problem can be studied by considering a directed graph [11].

Definition 5.1.1 (Directed Graph). *A directed graph, or digraph, $\mathcal{G}(\mathcal{V}, \mathcal{E})$, is a set of nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$ where the edges are ordered pairs of nodes.*

Equation 5.1 is transformed into a directed graph where each component of the output vector y and the input vector u is a node. An edge is added between $y_j^{[i]}$ and $u_k^{[i]}$ if,

$$\exists u^{[i]}, x^{[i]} \quad \text{s.t} \quad \frac{\partial g_j^{[i]}(x^{[i]}, u^{[i]})}{\partial u_k^{[i]}} \neq 0. \tag{5.4}$$

Further, an edge is added between u_i and y_j if,

$$\exists y \quad \text{s.t} \quad \frac{\partial c_i(y)}{\partial y_j} \neq 0. \tag{5.5}$$

This structural dependency between u and y is assumed to be available, if not, an all-to-all dependency is assumed. It should be emphasized that finding an evaluation order and using that to solve the equations is mathematically equivalent to solving the equations directly.

Finding the evaluation order is equivalent of finding the strongly connected components in a directed graph.

Definition 5.1.2 (Strongly Connected Graph ([70])). *Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a directed graph. If, for every pair of nodes $v, w \in \mathcal{V}$, there exist paths $p_1 : v \Rightarrow w$ and $p_2 : w \Rightarrow v$. Then $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is said to be strongly connected.*

Definition 5.1.3 (Strongly Connected Component (SCC) ([70]). *Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a directed graph and let $\mathcal{G}(\bar{\mathcal{V}}, \bar{\mathcal{E}})$ where $\bar{\mathcal{E}} = \{(v, w) \in \mathcal{E} | v, w \in \bar{\mathcal{V}}\}$ be a subgraph of $\mathcal{G}(\mathcal{V}, \mathcal{E})$. If $\mathcal{G}(\bar{\mathcal{V}}, \bar{\mathcal{E}})$ is a strongly connected graph, then $\mathcal{G}(\bar{\mathcal{V}}, \bar{\mathcal{E}})$ is said to be a strongly connected component of $\mathcal{G}(\mathcal{V}, \mathcal{E})$.*

Note that the definition also considers single nodes as strongly connected components.

In general, the path and the strongly connected components can be found by employing Tarjan's algorithm [70] which is shown in Algorithm 1. The algorithm computes the strongly connected components together with the order in which the nodes need to be evaluated. This is worth emphasizing, Tarjan's algorithm both gives us the *evaluation order* in which nodes need to be evaluated which is exactly what is needed together with which nodes need to be evaluated together (i.e. the nodes in the *strongly connected components*). In Example 5.1.1 the system illustrated in Figure 5.2 is revisited while in Example 5.1.2, Example 5.0.1 is revisited using the structural approach.

Example 5.1.1 (Strongly connected components). *Consider the models from Figure 5.2 with the input-output relations,*

$$y^{[1]} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} u^{[1]} \quad y^{[2]} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} u^{[2]} \quad y^{[3]} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} u^{[3]} \tag{5.6a}$$

and the coupling,

$$u^{[2]} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} y^{[1]}, \quad u^{[3]} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} y^{[2]}, \quad y_1^{[3]} = u_2^{[1]}. \tag{5.7a}$$

The coupled models can be represented by a directed graph where the nodes are y and u and the edges are the relations between them. In Figure 5.3 the graph is shown. Analyzing the graph using Tarjan's algorithm a strongly connected component is found that contains more than one node, cf. Figure 5.4. The evaluation order is additionally given by the algorithm and shown in the figure. The variables included in the strongly connected component are necessary to solve for simultaneously.

Example 5.1.2 (Initialization failure, revisited). *Consider again Example 5.0.1. Now instead of using the straightforward approach for solving the initialization problem, structural analysis is performed to compute the evaluation order. In Figure 5.5 the graph of the connection is shown. As there are no algebraic loops in the graph, the evaluation order is clear. The evaluations become,*

1. *Get $y^{[1]}$:* $y^{[1]} = 1$
2. *Set $u^{[2]}$:* $u^{[2]} = 1$
3. *Get $y^{[2]}$:* $y^{[2]} = -1$
4. *Set $u^{[1]}$:* $u^{[1]} = -1$

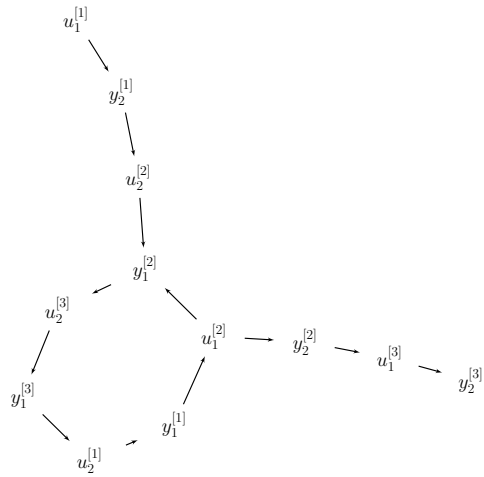

Figure 5.3: *The connections between the models from Example 5.1.1.*

and the initialization is successful.

An important observation is if Equation 5.1 contain models with direct-feed through and is free of strongly connected components then an ordering can be found for u and y such that the equations can be solved in an explicit sequence.This is equivalent to a coupling induced algebraic loop.

Consider the problem where g in Equation 5.1 is linear in u and c linear in y, i.e,

$$y = \hat{g}(x) + Du \tag{5.8a}$$
$$u = Ly \tag{5.8b}$$
$$\Rightarrow \tag{5.8c}$$
$$y = \hat{g}(x) + DLy. \tag{5.8d}$$

If there are no strongly connected components then an explicit sequence can be found to solve the above equation and specifically an evaluation order of the components of y can be found such that Equation 5.8d can be solved in sequence. In addition, this requires that DL is nilpotent and consequently the spectral radius, $\rho(DL) = 0$. This is an important aspect when considering stability in the simulation of coupled system, cf. Section 6.1.

In summary, using Tarjan's algorithm on the initialization problem, Equation 5.1, an evaluation order is given corresponding to the order in which the inputs, u, and outputs, y, should be computed for a successful initialization. In case of an algebraic loop, Definition 5.0.2, an explicit sequence for evaluating y and u cannot be found. For these cases, Tarjan's algorithm computes the strongly connected components, i.e. computes the inputs, u_j and outputs, y_i, which are included in the loop. The variables included in the loop need to be solved for simultaneously. The strongly connected components are included in the evaluation order.

Algorithm 1 Tarjan's Algorithm for finding SCCs and the evaluation order.

Require: A directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.

```
1: initialize
2:    lowlink = number = {v : not numbered |∀v ∈ V}
3:    i = 0
4: for v ∈ V do
5:    if v not numbered: call STRONGCONNECT(v)
6: end for
7: procedure STRONGCONNECT(x)
8:    lowlink(x) = number(x) = i
9:    stack.append(x)
10:   i = i + 1
11:    for w|(x, w) ∈ E do
12:       if w not numbered do
13:          call STRONGCONNECT(w)
14:          lowlink(x) = min(lowlink(x), lowlink(w))
15:       else if number(w) < number(x) and w ∈ stack do
16:          lowlink(x) = min(lowlink(x), number(w))
17:       end if
18:    end for
19:    if number(x) equal lowlink(x) do
20:       create new strongly connected component
21:       while stack and number(last in stack) >= number(x) do
22:          add last in stack to the component and remove from stack
23:       end while
24:    end if
25: end procedure
```

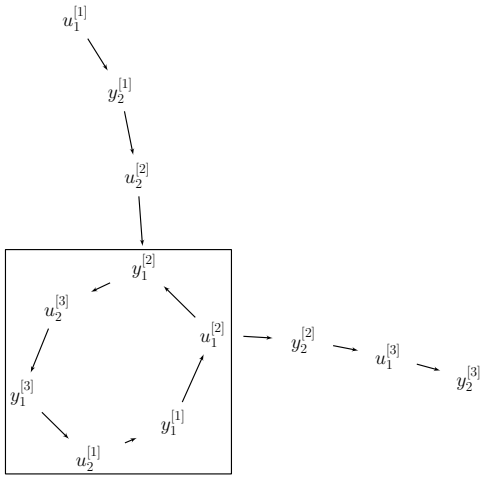

Figure 5.4: *The connections between the models from Example 5.1.1 with the strongly connected component of more than one node marked. The evaluation order is clearly shown, starting from* $u_1^{[1]}$.

Figure 5.5: *The connections between the models from Example 5.1.2. The evaluation order is obvious.*

5.2 Reducing model evaluations

Analyzing the initialization problem, Equation 5.1, using the structural analysis from Section 5.1, we find that the computed evaluation order is not necessarily unique. Depending on the relations between the connections, a reordering of the evaluation order may be performed, resulting in an evaluation order that is computationally beneficial as compared to the resulting order computed from Section 5.1. Consider Example 5.2.1, where the non-uniqueness of the evaluation order is highlighted. Further, in Section 5.3.2 the evaluation order and reordering is considered for an industrial model.

Example 5.2.1 (Non-unique evaluation order). *Consider two coupled models with the input-output relations,*

$$y_1^{[1]} = u_1^{[1]} \quad y^{[2]} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} u^{[2]} \tag{5.9a}$$

and the coupling,

$$u^{[2]} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} y^{[1]}. \tag{5.10a}$$

The resulting directed graph of the coupled system is shown in Figure 5.6. Analyzing the graph using Tarjan's algorithm results in that either the evaluation order is,

$$u_1^{[1]}, y_1^{[1]}, u_1^{[2]}, y_2^{[2]}, u_2^{[2]}, y_1^{[2]} \tag{5.11}$$

or

$$u_1^{[1]}, y_1^{[1]}, u_2^{[2]}, y_1^{[2]}, u_1^{[2]}, y_2^{[2]} \tag{5.12}$$

depending of the order of the edges in the directed graph. Both of the evaluation orders lead to the correct result.

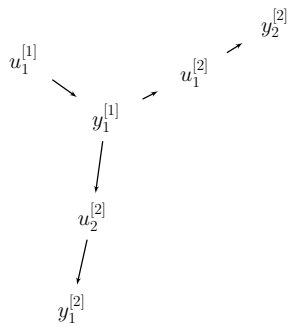

Figure 5.6: *The connections between the models from Example 5.2.1. As seen from the figure, either branch that emanates from $y_1^{[1]}$ can be chosen for evaluation prior to the other due to the non-uniqueness of the evaluation order.*

In order to understand the possible performance differences between evaluation orders it is necessary to revisit the model definition. The model considered here is a co-simulation FMU defined by Equation 2.6. Setting an input to the model and retrieving an output triggers an evaluation of the internal dynamics which is assumed expensive.

Assumption 5.1. *If any input $u^{[i]}$ has been set between retrieving outputs $y_j^{[i]}$, an internal evaluation of the dynamics in a co-simulation FMU, Equation 2.6, is triggered.*

Assumption 5.2. *Evaluation of a subsystems dynamics is expensive.*

In general, Assumption 5.1, can be relaxed if the internal subsystems only compute the relevant equations when an input has been set. However, this is not common which motivates the assumption. The goal here is to set as many inputs as possible before retrieving the outputs in order to minimize the number of evaluations of the internal dynamics in view of Assumption 5.2.

An outline of an algorithm for reducing the number of model evaluations is as follow. The graph of the initialization problem in Section 5.1, contains a node for each input and

output. To reduce the model evaluations, we first modify the graph before running Tarjan's algorithm on the graph. As a first step we group all output nodes from a model that are not included in a feed-through term. This is due to that these nodes cannot create an algebraic loop. Second, we group all output nodes from a model that are *connected* to inputs which are not included in a feed-through term. These can also not create algebraic loops, cf. Example 5.2.2. These steps are done to simplify the graph by removing nodes and connections which are not, in any way, involved in direct feed-through terms. After the two steps, Tarjan's algorithm is executed, as in the previous section, to identify the strongly connected components and consequently return an evaluation order.

To reduce the model evaluations we now modify the evaluation order. We consider each component in the sequence given by the evaluation order. If the component is a single output (a), we try to move it so that it is evaluated earlier. This is possible in cases when the evaluation order is not unique as shown in Example 5.2.1. If the component contains more than one node, we group these nodes in the graph. We test if a move is possible by considering again the evaluation order. Starting from the first component, we go through the components and check if the component (b) only contains output nodes from the same model as a belong to. If b and a are outputs from the same model, we try to group them into one component. If a is not a child of b then they can be joined, both in the component and in the graph. This is performed until all components in the evaluation order have been considered. The proposed algorithm is shown in Algorithm 2.

Example 5.2.2 (Outputs and direct feed-through). *Consider the initialization problem represented by the graph in Figure 5.7. There are three connected models but only one model has multiple outputs, the second model. From the figure we note that the outputs from the second*

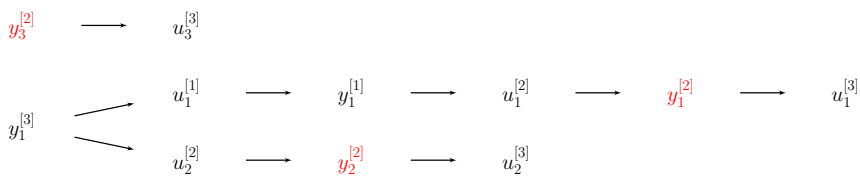

Figure 5.7: *Graph of the initialization problem in Example 5.2.2.*

model are connected to inputs which are not included in a feed-through term ($u^{[3]}$). Thus, the outputs from the second model can be grouped, without creating an algebraic loop. Furthermore, this results in a minimal number of model evaluations. The resulting graph is shown in Figure 5.8.

Algorithm 2 Computing a reduced initial evaluation order

Require: A directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.
Require: Information about which nodes belong to same model, M.
Require: Information about which nodes are output.
1: {Group all outputs, from a model, that are not included in a feed-through term}
2: {Group all outputs, from a model, that are connected to inputs which are not included in a feed-through term}
3: $\mathcal{F} =$ **Tarjan**$(\mathcal{G})$ {Compute the strongly connected components}
4: $i := 0$
5: **while** $i < \dim(\mathcal{F})$ **do**
6: $f_i \in \mathcal{F}$
7: $b := 0$
8: **if** $\dim(f_i) = 1$ and $v \in f_i | v$ output **then**
9: **for** $j = [0, \ldots, i-1]$ **and** $e_j \in \mathcal{F}$ **do**
10: **if** $\dim(e_j) = 1$ and $w \in e_j | w$ output, $w, v \in M^{[k]}$ **and** v not a child of w **then**
11: $f_i := \{f_i, e_i\}$ {Update the ith item in $\mathcal{F}$, by joining f_i and e_i into one}
12: $b := 1$ {Do not update the counter }
13: break
14: **end if**
15: **end for**
16: **end if**
17: **if** $b = 0$ **then**
18: $i := i + 1$ {Increment counter}
19: **end if**
20: $\mathcal{G} := \mathcal{G}/f_i$ {Group nodes in a strongly connected component}
21: **end while**

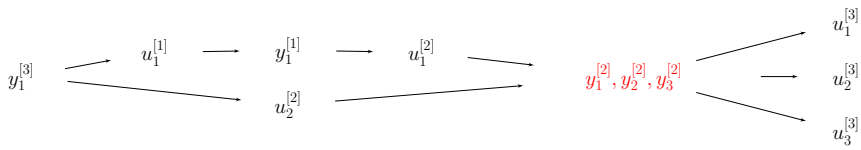

Figure 5.8: *Graph of the solution of the initialization problem from Example 5.2.2 where all the outputs from the second model, $y^{[2]}$, have been joined.*

5.3 Case studies

5.3.1 Academic test case

In this example five models with feed-through are connected. A graph of the couplings is shown in Figure 5.9. The example is intended to illustrate Algorithm 2 where the number

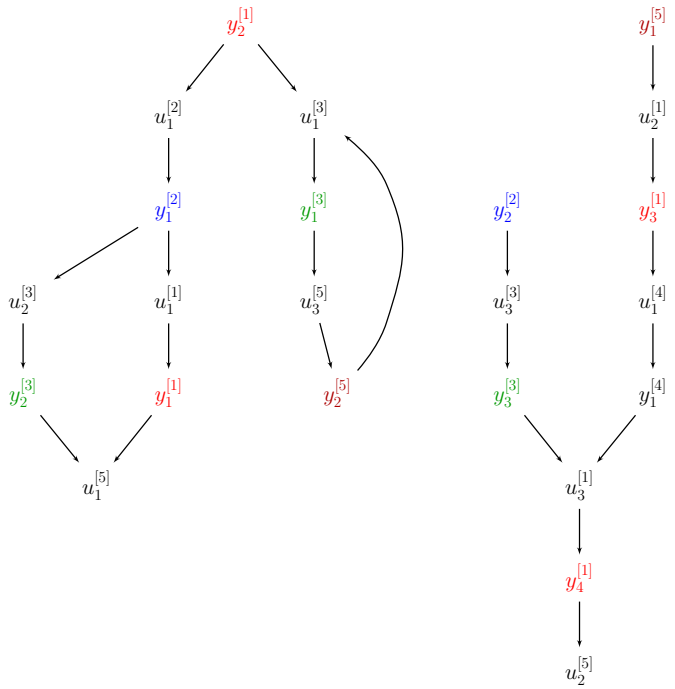

Figure 5.9: *Graph showing five coupled models where as many as possible of the outputs for each model should be joined.*

of model evaluations is reduced.

As a first step, we note that in the graph outputs $y_1^{[1]}$ and $y_4^{[1]}$ can directly be joined as they are not connected to inputs that are feed-through terms, cf. Figure 5.10. In the next step in Algorithm 2, Tarjan's algorithm (Algorithm 1) is executed in order to detect

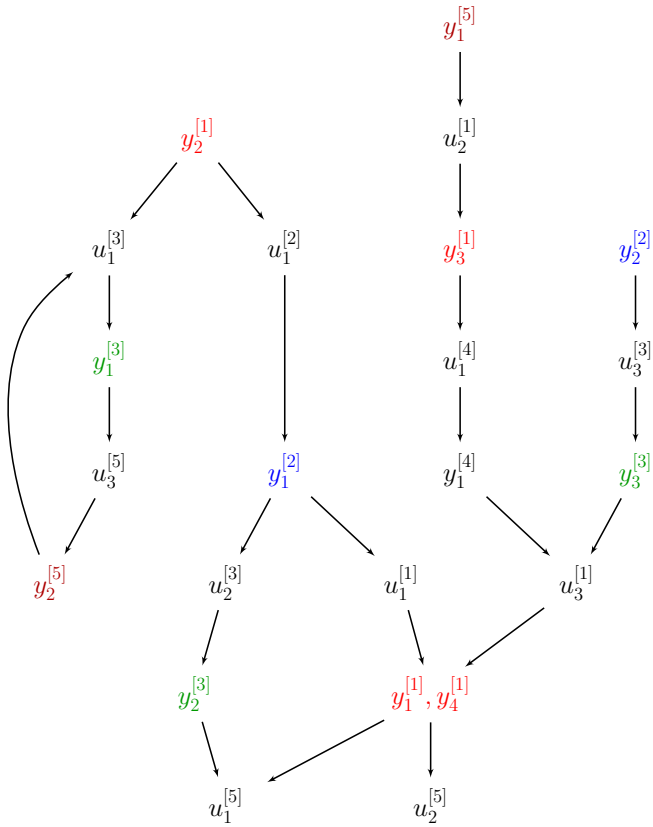

Figure 5.10: *Result after the first steps of Algorithm 2 for Section 5.3.1 where $y_1^{[1]}$ and $y_4^{[1]}$ has been joined.*

the strongly connected components and computing a first evaluation order. There is one strongly connected component with dimension greater than one ($u_1^{[3]}$, $y_1^{[3]}$, $u_3^{[5]}$, $y_2^{[5]}$). The outputs in this component, cannot be matched to any other output. However, looping through the outputs in the evaluation order, and trying to match the outputs higher up in the evaluation order, we find that $y_2^{[1]}$, $y_3^{[1]}$ and $y_1^{[2]}$, $y_1^{[2]}$ can be joined. Joining these nodes results in the graph shown in Figure 5.11 which gives also the evaluation order that minimizes the number of model evaluations.

5.3.2 Race car

For racing applications finding the maximal performance of the car is crucial. One method to quickly estimate the impact on performance of a change to the vehicle setup is to solve for the steady state limits under different driving conditions. Identifying a set of critical points along a race track and calculating the maximum achievable speed for each point can give a good indication on how the change will affect the lap time. Simulations can be

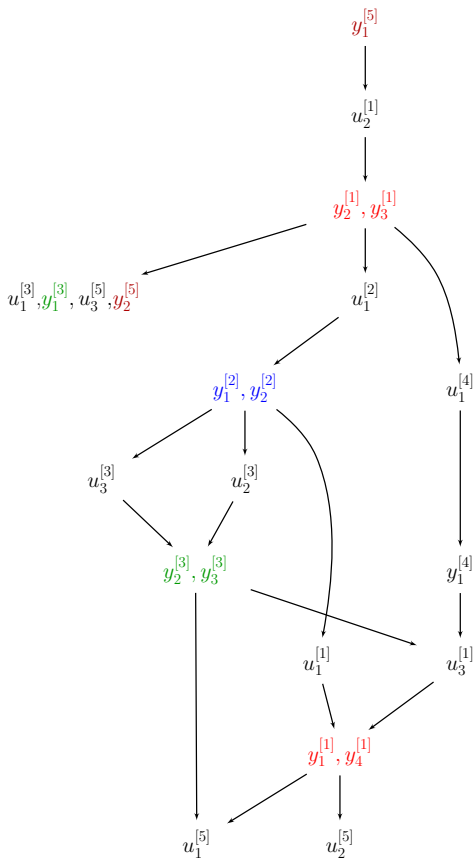

Figure 5.11: *Resulting graph after Algorithm 2 has been executed for Section 5.3.1. As many as possible of the outputs from the different models has been joined.*

carried out with predefined input or by a feedback loop using either a simulator or a virtual driver model to investigate the dynamic response.

In this example, a race car (cf. Figure 5.12) is modeled in Modelica [46] using the commercial Vehicle Dynamics Library [7]. The car is driven by a virtual driver that tries to stay onto an eight shaped course with increasing velocity in order to investigate the dynamic response of the car, especially when changing the turning direction. Here, we consider initialization of the race car in a co-simulation setting. There are five coupled models, the chassis and the four wheels. Between them, there are 172 connections. Additionally there is direct feed-through in the wheels, cf. Figure 5.13 and cf. Appendix A.4.

By employing Algorithm 1 on the coupled system, we find that there are no strongly connected components and thus no algebraic loops.

Analyzing the coupled system, given Assumption 5.1, we see that a worst case scenario for the number of model evaluations are 25, one evaluation of the chassis and 6 for each

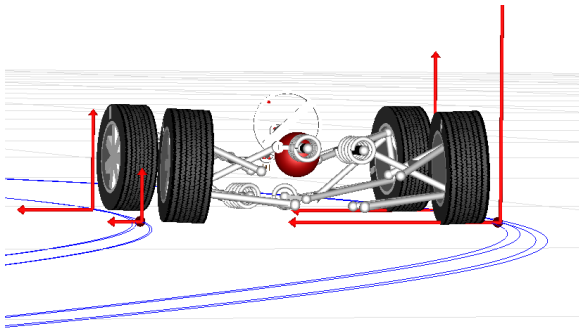

Figure 5.12: *Visualization of the race car from Section 5.3.2. © Modelon.*

wheel. This is due to that the torques, t_i and forces f_i, $i = 1 : 4$ in the couplings are spatial. The best scenario is 5 model evaluations, one evaluation of the chassis and one for each wheel.

Note though that we do not consider additional evaluations of the chassis as there is no direct feed-through.

Comparing the elapsed initialization time for the two scenarios we see that for the worst case, the initialization takes about 1 second while for the best case, about 0.1 seconds. For this coupled system, the optimal evaluation order for the initialization is about 10 times faster then the worst case. Furthermore, executing Algorithm 2 on the coupled system gives the optimal evaluation order.

5.4 Summary

In this chapter, the initialization problem has been analyzed. A structural approach has been taken in order to simplify the initialization by identifying equations that result in algebraic loops. The equations and variables resulting in an algebraic loop need to be solved together while the variables and equations that are not involved, can be solved in a sequence. Furthermore, due to Assumption 5.1 and Assumption 5.2 gains can be obtained, if the evaluation order is not unique, by reordering the evaluation order so that as many as possible of the outputs from a model are computed simultaneously. This resulted in Algorithm 2. No claims are made that the algorithm finds the minimal number of model evaluations. In our tests this reduced the number of model evaluations and increased performance.

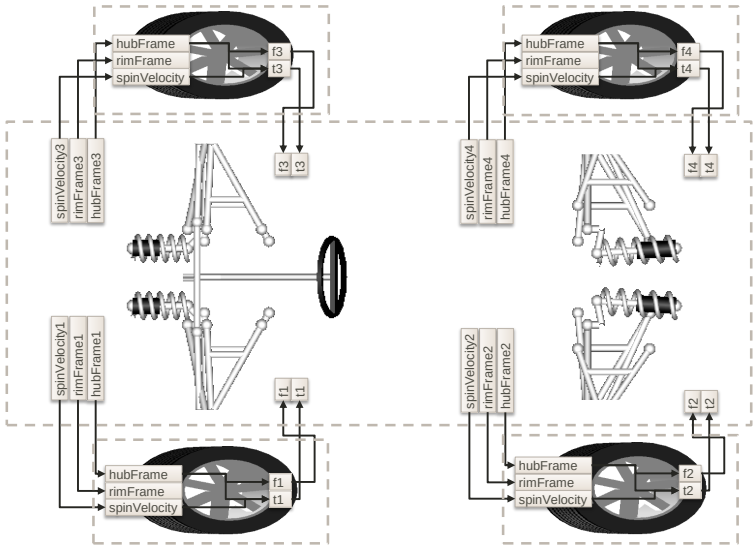

Figure 5.13: *Overview of the couplings between the wheels and chassis of the race car from Section 5.3.2. Shown in the figure is the direct feed-through in the wheels between the hubFrame and spinVelocity with t[1-4] and f[1-4]. Note that the connections are vector valued. © Modelon.*

Chapter 6

Simulation

Classically, simulation of weakly coupled system follows the parallel algorithm, Algorithm 3 (base algorithm), where the models are all treated in parallel and the inputs/outputs are extrapolated from the previous global time step. A major benefit of this approach is its obvious parallelism. This approach bears similarities with the Jacobi iteration for linear equations and is also referred to as a *Jacobi-like* approach.

Algorithm 3 Base Parallel Algorithm, ($T_n \rightarrow T_{n+1}$)

Require: M models and their connections

1: **for** $i = 1$ to M **do**
2: Set the input to the ith model, $u_n^{[i]}$.
3: Perform global time step, $T_n \rightarrow T_{n+1}$ for the ith model.
4: **end for**
5: **for** $i = 1$ to M **do**
6: Retrieve model outputs, $y_{n+1}^{[i]}$ ($y_{n+1}^{[i]} = g(x_{n+1}^{[i]}, u_n^{[i]})$).
7: **end for**
8: Compute $u_{n+1} = c(y_{n+1})$.

In this chapter, extensions to the base algorithm are discussed. In Section 6.1, stability is discussed and it is shown when and why stability problems may occur during co-simulation. In Section 6.2, algorithms for stabilizing the simulation is introduced and in Section 6.3, a smoothing approach for the inputs is introduced and discussed. The smoothing is done in order to preserve continuity over global time steps.

Other simulation approaches exist, such as the staggered approach where the models are solved in sequence, i.e. the global step is performed for one model at a time, cf. [23]. This and similar approaches will not be considered here due to that we only want to consider parallelizable algorithms.

6.1 Coupling stability

In the analysis of co-simulation one is interested in how the *coupling* effects the overall simulation with regard to stability and accuracy. In this section we focus on stability issues that may be influenced, depending on the simulation approach. In the analysis we focus on the parallel method for the overall simulation. As it is the coupling that is of interest we assume that the subsystems are solved exactly and only study how the coupling impacts the simulation. In practice this means that the subsystems are solved with high accuracy to not influence the stability or error estimation of the coupled system.

The aim in this section is to determine a propagation matrix $\Psi(H)$ which advances the solution using old values of the states and outputs and potentially old values of the output derivatives,

$$[x_{n+1}, y_{n+1}, \dots, y_{n+1}^{(k)}]^{\mathrm{T}} = \Psi(H)[x_n, y_n, \dot{y}_n, \dots, y_n^{(k)}]^{\mathrm{T}}. \tag{6.1}$$

Depending on the co-simulation approach, this matrix will have different properties which will influence the stability. The number of known values of the outputs used in advancing the solution is denoted by the parameter k, which in turn depends on the approach used. In order to prove stability we have to check the spectral radius of $\Psi(H)$,

$$\rho(\Psi(H)) \leq 1. \tag{6.2}$$

We have in case of multiple eigenvalues 1 to ensure that their algebraic and geometric multiplicity is the same. A minimal requirement is that Equation 6.2 holds at least for $H = 0$. This separates stability of the discretization method from the stability of the problem. This leads to the notion of zero stable methods [29]. The internal methods used in the subsystems are zero stable but in contrast to the classical case we will see, that in a co-simulation context, the problem and in particular the feed-through terms matter even in the case $H = 0$. Zero stability of the internal methods is not sufficient to guarantee stability of the coupled system for $H = 0$. We have to set up conditions on the feed-through term as well. In Definition 6.1.1, we define the *coupling stability*.

Definition 6.1.1 (Coupling stability). *A coupled system is coupling stable, if the spectral radius of $\Psi(H)$, in Equation 6.1, fulfills,*

$$\lim_{H \to 0} \rho(\Psi(H)) \leq 1,$$

and eigenvalues on the boundary are non defective, i.e. their algebraic and geometric multiplicity is the same.

In the analysis we consider N linearly coupled linear systems (linearization of Assumption 3.1),

$$\dot{x}^{[i]} = A^{[i]}x^{[i]} + B^{[i]}u^{[i]}, \quad i = 1, \dots, N \tag{6.3a}$$

$$y^{[i]} = C^{[i]}x^{[i]} + D^{[i]}u^{[i]}, \quad i = 1, \dots, N \tag{6.3b}$$

$$u = Ly \tag{6.3c}$$

the connections between the systems is determined by the coupling matrix L, which maps the outputs y to the inputs u. For convenience we write,

$$A = \begin{bmatrix} A^{[1]} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & A^{[N]} \end{bmatrix}, \quad B = \begin{bmatrix} B^{[1]} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B^{[N]} \end{bmatrix}$$
$$C = \begin{bmatrix} C^{[1]} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & C^{[N]} \end{bmatrix}, \quad D = \begin{bmatrix} D^{[1]} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & D^{[N]} \end{bmatrix}$$

where A, B, C and D are block diagonal matrices and Equation 6.3 simplifies to,

$$\dot{x} = Ax + Bu \tag{6.4a}$$
$$y = Cx + Du \tag{6.4b}$$
$$u = Ly. \tag{6.4c}$$

Moreover, the system can be reformulated as an ODE,

$$\dot{x} = (A + BL(I - DL)^{-1}C)x \tag{6.5}$$

where $(I - DL)$ is assumed to be non-singular so that Equation 6.4 is an index one system with y and u being the algebraic variables. Now, consider that each subsystem in Equation 6.3a is solved exactly for a global time step $[T_n, T_{n+1}]$, we get for Equation 6.4a,

$$\Phi(x_n, u_n) = \int_{T_n}^{T_{n+1}} \mathrm{e}^{A(T_{n+1}-\tau)} B u_n \mathrm{d}\tau + \mathrm{e}^{A(T_{n+1}-T_n)} x_n. \tag{6.6}$$

It can be expected that the stability in Algorithm 3 depends on the coupling due to that the outputs are explicit. Now we define this dependency. The algorithm is defined by,

$$x_{n+1} = \Phi(x_n, u_n) \tag{6.7a}$$
$$y_{n+1} = Cx_{n+1} + Du_n \tag{6.7b}$$
$$u_{n+1} = Ly_{n+1}. \tag{6.7c}$$

Note that a solution for the outputs, u, are directly given using Equation 6.7b,c. Furthermore, Equation 6.7b,c can be interpreted as constraints in a DAE and as the constraints are not solved in the algorithm, we will onwards call this the inconsistent approach.

Starting by eliminating u we get,

$$x_{n+1} = \Phi(x_n, Ly_n) \tag{6.8a}$$
$$y_{n+1} = Cx_{n+1} + DLy_n. \tag{6.8b}$$

The exact solution for the states in $[T_n, t]$ is,

$$x(t) = \int_{T_n}^{t} \mathrm{e}^{A(t-\tau)} BLy(\tau)\mathrm{d}\tau + \mathrm{e}^{A(t-T_n)} x(T_n). \tag{6.9}$$

In the next step, the solution is approximated using constant extrapolation for the inputs, i.e,

$$y(\tau) := y(T_n), \quad \tau \in [T_n, T_{n+1}]. \tag{6.10}$$

Denoting the numerical approximation of $x(T_n)$ by x_n we get the full approximation for a global step as,

$$\begin{aligned} x_{n+1} &= \Phi(x_n, Ly_n) && \text{(6.11a)} \\ &= \int_{T_n}^{T_{n+1}} \mathrm{e}^{A(T_{n+1}-\tau)}\mathrm{d}\tau BLy_n + \mathrm{e}^{A(T_{n+1}-T_n)} x_n && \text{(6.11b)} \\ &= A^{-1}(\mathrm{e}^{AH} - I)BLy_n + \mathrm{e}^{AH} x_n && \text{(6.11c)} \end{aligned}$$

where H is the global step size which we assume to be the same for all steps. For simplicity, we introduce $K_1(H)$,

$$K_1(H) = A^{-1}(\mathrm{e}^{AH} - I)BL \quad \text{with} \quad \lim_{H\to 0} K_1(H) = 0. \tag{6.12}$$

Inserting Equation 6.11 into Equation 6.8b gives

$$\begin{aligned} y_{n+1} &= CK_1(H)y_n + C\mathrm{e}^{AH}x_n + DLy_n \\ &= [CK_1(H) + DL]y_n + C\mathrm{e}^{AH}x_n. \end{aligned} \tag{6.13}$$

A global step is then calculated as,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \underbrace{\begin{bmatrix} \mathrm{e}^{AH} & K_1(H) \\ C\mathrm{e}^{AH} & CK_1(H) + DL \end{bmatrix}}_{\Psi(H)} \begin{bmatrix} x_n \\ y_n \end{bmatrix}. \tag{6.14}$$

Now, the interest is in the iteration matrix, $\Psi(H)$, in the limit as the step size, H, tends to zero,

$$\lim_{H\to 0} \Psi(H) = \begin{bmatrix} I & 0 \\ C & DL \end{bmatrix} \tag{6.15}$$

with eigenvalues,

$$\lambda_{1,\ldots,k} = 1, \quad \lambda_{k+1,\ldots,k+l} = \mathrm{eig}(DL). \tag{6.16}$$

where k are the total number of states and l are the total number of outputs. For stability, we *require* that the eigenvalues of DL are less or equal to one, $\rho(DL) \leq 1$ and those on

Algorithm 4 Parallel consistent, ($T_n \to T_{n+1}$)

Require: M models and their connections

1: **for** $i = 1$ to M **do**
2: Set the input to the ith model, $u_n^{[i]}$.
3: Perform global time step, $T_n \to T_{n+1}$ for the ith model.
4: **end for**
5: Solve $y_{n+1} = g(x_{n+1}, c(y_{n+1}))$, for y_{n+1}.
6: Compute $u_{n+1} = c(y_{n+1})$.

the boundary not being defective. Note that an eigenvalue of one is excluded due to the nonsingularity requirement of $(I - DL)^{-1}$.

A similar algorithm is Algorithm 4, which is implicit in the outputs instead of explicit. The question is, do we have the same stability requirements? The implicit algorithm proceeds by first solving for the states x_{n+1} and then solving the outputs y_{n+1} together with the inputs u_{n+1}, we get,

$$x_{n+1} = \Phi(x_n, u_n) \tag{6.17a}$$
$$y_{n+1} = Cx_{n+1} + Du_{n+1} \tag{6.17b}$$
$$u_{n+1} = Ly_{n+1}. \tag{6.17c}$$

For $D = 0$, the two algorithms are identical, which is the case if no subsystem has feedthrough. Note that the algorithm requires that we are able to solve Equation 6.17b and Equation 6.17c together and therefor it is an implicit method. This is in contrast to Equation 6.7b,c where a solution is directly given. Considering Equation 6.17b,c as constraints in a DAE, as in Equation 6.7b,c, we here solve the constraints and thus onwards call this the consistent approach. Note that in a FMI co-simulation framework, solving Equation 6.17b,c may not always be possible (Restriction 2.1). In the nonlinear case,

$$y_{n+1}^{[i]} = g^{[i]}(t, x_{n+1}^{[i]}, u_{n+1}^{[i]}), \quad i = 1, \ldots, N \tag{6.18a}$$
$$u_{n+1} = c(y_{n+1}) \tag{6.18b}$$

an iteration on the output and input variables should be performed in order to solve $y^{[i]}$ and u together.

Going back to Equation 6.17 and by eliminating u we get,

$$x_{n+1} = \Phi(x_n, Ly_n) \tag{6.19a}$$
$$y_{n+1} = (I - DL)^{-1}Cx_{n+1} \tag{6.19b}$$

which is the algorithm we will investigate.

Now, using Equation 6.11 and Equation 6.12 together with Equation 6.19b we get,

$$\begin{bmatrix} I & 0 \\ -C & I - DL \end{bmatrix} \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} \mathrm{e}^{AH} & K_1(H) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{6.20}$$

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \underbrace{\begin{bmatrix} \mathrm{e}^{AH} & K_1(H) \\ (I - DL)^{-1}C\mathrm{e}^{AH} & (I - DL)^{-1}CK_1(H) \end{bmatrix}}_{\Psi(H)} \begin{bmatrix} x_n \\ y_n \end{bmatrix}. \tag{6.21}$$

Now, considering the iteration, $\Psi(H)$, as before, with the step size, H, tending to zero,

$$\lim_{H \to 0} \Psi(H) = \begin{bmatrix} I & 0 \\ (I - DL)^{-1}C & 0 \end{bmatrix} \tag{6.22}$$

with eigenvalues,

$$\lambda_{1,\ldots,k} = 1, \quad \lambda_{k+1,\ldots,k+l} = 0 \tag{6.23}$$

where k is the total number of states and l are the total number of outputs. There is no restriction for coupling stability in this case, the approach is unconditionally coupling stable.

In Example 6.1.1, both methods are run on two coupled systems.

Example 6.1.1 (Linear example). *Consider a linear coupled system of two subsystems that both consist of a single state, x, a single input u and a single output y. The coupling is determined by the values of two parameters, $d^{[1]}$ and $d^{[2]}$. By setting both to zero we end up with a fully decoupled system. System one is determined by,*

$$\dot{x}^{[1]} = -x^{[1]} + u^{[1]} \tag{6.24a}$$
$$y^{[1]} = x^{[1]} + d^{[1]}u^{[1]} \tag{6.24b}$$

and system two defined by,

$$\dot{x}^{[2]} = -x^{[2]} + 3u^{[2]} \tag{6.25a}$$
$$y^{[2]} = -5x^{[2]} + d^{[2]}u^{[2]}. \tag{6.25b}$$

The coupling is determined by,

$$\begin{bmatrix} u^{[1]} \\ u^{[2]} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}}_{L} \begin{bmatrix} y^{[1]} \\ y^{[2]} \end{bmatrix}. \tag{6.26}$$

Discretizing the coupled system using constant extrapolation for the signals as described by Equation 6.8, and by letting $H \to 0$ we obtain,

$$\begin{bmatrix} y_{n+1}^{[1]} \\ y_{n+1}^{[2]} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{C} \begin{bmatrix} x_{n+1}^{[1]} \\ x_{n+1}^{[2]} \end{bmatrix} + \underbrace{\begin{bmatrix} d^{[1]} & 0 \\ 0 & d^{[2]} \end{bmatrix}}_{D} \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}}_{L} \begin{bmatrix} y_{n}^{[1]} \\ y_{n}^{[2]} \end{bmatrix} \tag{6.27}$$

The coupled system is stable, in case of the inconsistent approach is used, if the spectral radius $\rho(DL) \leq 1$. The eigenvalues are,

$$\lambda_{1,2} = \pm\sqrt{d^{[1]}d^{[2]}}. \tag{6.28}$$

In Figure 6.1, simulations using the inconsistent approach together with constant extrapolation for the inputs with different values on the parameters $d^{[1]}$ and $d^{[2]}$ are shown. As can be seen, the simulations are stable if $\rho(DL) \leq 1$ and unstable for $\rho(DL) > 1$. In Figure 6.2, the same coupled system is simulated using the consistent approach, and as can be seen from the figure, there is no problem related to the stability.

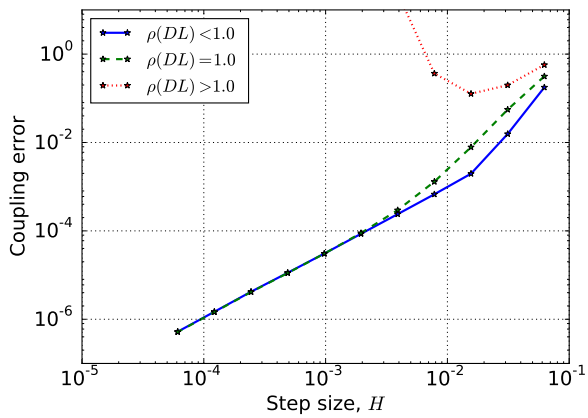

Figure 6.1: *Result of simulating Model 6.24 and Model 6.25 from Example 6.1.1 in parallel using the inconsistent approach, Equation 6.8, and for varying values on d_1 and d_2. As expected, the simulation become unstable for $\rho(DL) > 1$.*

In [41], the same coupling stability requirement, for both the consistent and the inconsistent method with constant extrapolation, was found. They called it *zero-stability*.

In this section, we have used constant extrapolation for the inputs and identified the cases where we have coupling stability. What if we increase the extrapolation order? When do we have coupling stability in these cases? In Figure 6.3, Example 6.1.1 is simulated using constant, linear,

$$u_{n+1} = u_n + H\Big(\frac{u_n - u_{n-1}}{H}\Big) \tag{6.29}$$

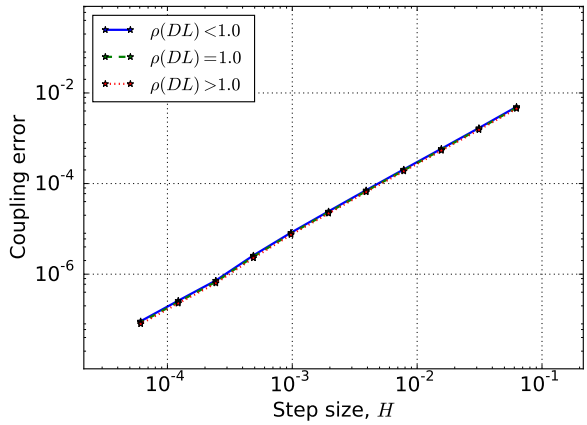

Figure 6.2: *Result of simulating Model 6.24 and Model 6.25 from Example 6.1.1 in parallel using the consistent approach, Equation 6.19, and for varying values on* d_1 *and* d_2*. As expected, there is no problem related to stability.*

and quadratic extrapolation,

$$u_{n+1} = u_n + H\Big(\frac{u_n - u_{n-1}}{H}\Big) + \frac{H^2}{2}\Big(\frac{u_n - 2u_{n-1} + u_{n-2}}{H^2}\Big), \tag{6.30}$$

where old values were used in the calculation of the output variables. As can be seen from the figure, the stability is affected by the extrapolation order. The question is how the requirement for a coupling stable integration changes with the extrapolation order.

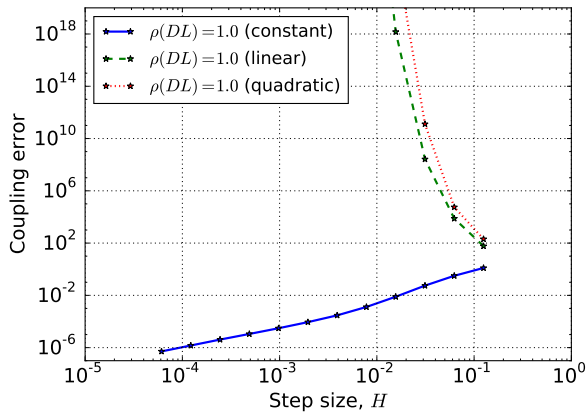

Figure 6.3: *Result of simulating Model 6.24 and Model 6.25 from Example 6.1.1 in parallel using constant, linear and quadratic extrapolation together with the inconsistent approach. As shown in the figure, the simulations become unstable for higher order extrapolations. The constant extrapolation case is equal to the case* $\rho(DL) = 1$ *in Figure 6.1.*

6.1.1 Linear extrapolation

Equation 6.7 is based on the assumption that constant extrapolation was used in-between the global time steps. If we instead assume linear extrapolation (Equation 6.29) for the inconsistent approach we get,

$$x_{n+1} = \Phi\Big(x_n, u_n, \frac{u_n - u_{n-1}}{H}\Big) \tag{6.31a}$$

$$y_{n+1} = Cx_{n+1} + D\Big(u_n + H\Big(\frac{u_n - u_{n-1}}{H}\Big)\Big) \tag{6.31b}$$

$$u_{n+1} = Ly_{n+1}. \tag{6.31c}$$

What is the impact on the coupling stability using this approach? Starting by eliminating u,

$$x_{n+1} = \Phi\Big(x_n, Ly_n, L\frac{y_n - y_{n-1}}{H}\Big) \tag{6.32a}$$

$$y_{n+1} = Cx_{n+1} + D(2Ly_n - Ly_{n-1}). \tag{6.32b}$$

Again assuming that the states are solved exactly in each subsystem,

$$x(t) = \int_{T_n}^{t} \mathrm{e}^{A(t-\tau)}BLy(\tau)\mathrm{d}\tau + \mathrm{e}^{A(T_n-t)}x(T_n) \tag{6.33}$$

and using the linear extrapolation we find for the states,

$$x_{n+1} = \Phi\Big(x_n, Ly_n, L\frac{y_n - y_{n-1}}{H}\Big) \tag{6.34a}$$

$$= \int_{T_n}^{T_{n+1}} \mathrm{e}^{A(T_{n+1}-\tau)}BL\Big(y_n + (\tau - T_n)\Big(\frac{y_n - y_{n-1}}{H}\Big)\Big)\mathrm{d}\tau \tag{6.34b}$$

$$+ \mathrm{e}^{A(T_{n+1}-T_n)}x_n \tag{6.34c}$$

$$= \big[\bar{\tau} = \tau - T_n\big] \tag{6.34d}$$

$$= \int_{0}^{H} \mathrm{e}^{A(H-\bar{\tau})}BL\Big(y_n + \bar{\tau}\Big(\frac{y_n - y_{n-1}}{H}\Big)\Big)\mathrm{d}\bar{\tau} \tag{6.34e}$$

$$+ \mathrm{e}^{A(T_{n+1}-T_n)}x_n. \tag{6.34f}$$

Again, for simplicity, we introduce $K_2(H)$,

$$K_2(H) = A^{-2}(\mathrm{e}^{AH} - I - AH)BLH^{-1} \quad \text{with} \quad \lim_{H\to 0} K_2(H) = 0. \tag{6.35}$$

Now, Equation 6.12 and Equation 6.35 together with Equation 6.34 result in,

$$\begin{aligned} x_{n+1} &= K_1(H)y_n + K_2(H)(y_n - y_{n-1}) + \mathrm{e}^{AH}x_n \\ &= (K_1(H) + K_2(H))y_n - K_2(H)y_{n-1} + \mathrm{e}^{AH}x_n. \end{aligned} \tag{6.36}$$

For the outputs we get,

$$
\begin{aligned}
y_{n+1} &= Cx_{n+1} + 2DLy_n - DLy_{n-1} && \text{(6.37a)}\\
&= C(K_1(H) + K_2(H))y_n - CK_2(H)y_{n-1} + C\mathrm{e}^{AH}x_n && \text{(6.37b)}\\
&\quad + 2DLy_n - DLy_{n-1} && \text{(6.37c)}\\
&= \big(CK_1(H) + CK_2(H) + 2DL\big)y_n && \text{(6.37d)}\\
&\quad + \big(- CK_2(H) - DL\big)y_{n-1} && \text{(6.37e)}\\
&\quad + C\mathrm{e}^{AH}x_n. && \text{(6.37f)}
\end{aligned}
$$

The resulting global step is performed as,

$$
\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ y_n \end{bmatrix} = \underbrace{\begin{bmatrix} \mathrm{e}^{AH} & K_1(H)+K_2(H) & -K_2(H) \\ C\mathrm{e}^{AH} & C(K_1(H)+K_2(H))+2DL & -CK_2(H)-DL \\ 0 & I & 0 \end{bmatrix}}_{\Psi(H)} \begin{bmatrix} x_n \\ y_n \\ y_{n-1} \end{bmatrix}. \tag{6.38}
$$

To determine the coupling stability we consider, $\Psi(H)$, in the limit as the step size, H, tends to zero, which result in,

$$
\lim_{H\to 0} \Psi(H) = \begin{bmatrix} I & 0 & 0 \\ C & 2DL & -DL \\ 0 & I & 0 \end{bmatrix}. \tag{6.39}
$$

The eigenvalues are,

$$
\lambda_{1,\ldots,k} = 1, \quad \lambda_{k+1,\ldots,k+2l} = \mathrm{eig}\left(\begin{bmatrix} 2DL & -DL \\ I & 0 \end{bmatrix} \right) \tag{6.40}
$$

where k are the total number of states and l are the total number of outputs. For stability, the requirement is that $|\lambda_{k+1,\ldots,k+2l}| \leq 1$ and those on the boundary not being defective. Note that an eigenvalue of one is excluded due to the nonsingularity requirement of $(I - DL)^{-1}$. If DL is diagonalizable, we find that,

$$
\lambda_{k+1,\ldots,k+2l} = \lambda_{DL} \pm \sqrt{\lambda_{DL}^2 - \lambda_{DL}} \tag{6.41}
$$

where λ_{DL} are the l eigenvalues of DL.

Alternatively to Equation 6.31 where a finite difference approach was used to approximate the derivative of inputs with respect to time, FMI allow for using the analytical

derivative (Feature 2.6). Using the analytical derivative we instead get,

$$x_{n+1} = \Phi(x_n, u_n, \dot{u}_n) \tag{6.42a}$$
$$y_{n+1} = Cx_{n+1} + D\Big(u_n + H\dot{u}_n\Big) \tag{6.42b}$$
$$u_{n+1} = Ly_{n+1} \tag{6.42c}$$
$$\dot{u}_{n+1} = L\dot{y}_{n+1} \tag{6.42d}$$

eliminating u,

$$x_{n+1} = \Phi(x_n, Ly_n, L\dot{y}_n) \tag{6.43a}$$
$$y_{n+1} = Cx_{n+1} + D\Big(Ly_n + HL\dot{y}_n\Big) \tag{6.43b}$$
$$\dot{y}_{n+1} = C\dot{x}_{n+1} + DL\dot{y}_n. \tag{6.43c}$$

Following the above line of calculations we find that a global step is performed as,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ \dot{y}_{n+1} \end{bmatrix} = \underbrace{\begin{bmatrix} \mathrm{e}^{AH} & K_1(H) & HK_2(H) \\ C\mathrm{e}^{AH} & CK_1(H)+DL & HCK_2(H)+HDL \\ CA\mathrm{e}^{AH} & C(AK_1(H)+BL) & HC(AK_2(H)+BL)+DL \end{bmatrix}}_{\Psi(H)} \begin{bmatrix} x_n \\ y_n \\ \dot{y}_n \end{bmatrix}. \tag{6.44}$$

Considering $\Psi(H)$ as before,

$$\lim_{H\to 0} \Psi(H) = \begin{bmatrix} I & 0 & 0 \\ C & DL & 0 \\ CA & CBL & DL \end{bmatrix} \tag{6.45}$$

with

$$HK_2(H) = A^{-2}(\mathrm{e}^{AH} - I - AH)BL \quad \text{and} \quad \lim_{H\to 0} HK_2(H) = 0. \tag{6.46}$$

Note that the requirement for stability is the same as in the inconsistent approach with constant extrapolation, $\rho(DL) \leq 1$. Using linear extrapolation with analytical derivatives do not reduce the region for coupling stability.

In Figure 6.4 the regions for which the inconsistent approaches are stable are shown for constant, linear and quadratic extrapolation where the higher order extrapolations used finite differences. The requirements for quadratic extrapolation can be found straightforward from above. In Figure 6.5 and in Figure 6.6, result is shown for simulating Example 6.1.1 using linear and quadratic extrapolation respectively with the inconsistent approach.

Considering again the consistent approach, Equation 6.19, where the output equations and the coupling equations are solved, but this time with linear extrapolation,

$$x_{n+1} = \Phi\Big(x_n, Ly_n, L\frac{y_n - y_{n-1}}{H}\Big) \tag{6.47a}$$
$$y_{n+1} = (I - DL)^{-1}Cx_{n+1}. \tag{6.47b}$$

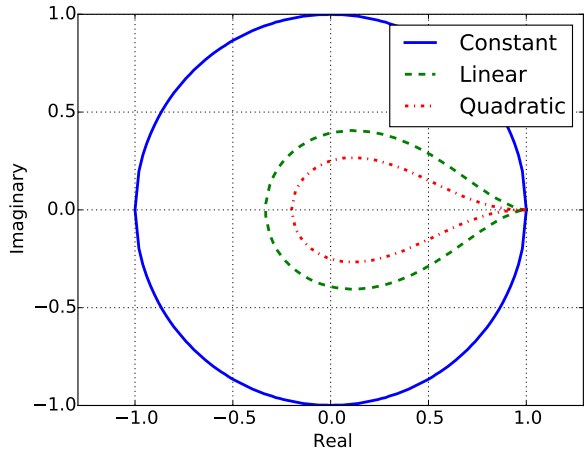

Figure 6.4: *Regions for which the inconsistent approaches are coupling stable in terms of $\rho(DL)$ using different extrapolation orders and approximated by finite differences.*

The states are calculated as in Equation 6.36 and inserting this into Equation 6.47b,

$$y_{n+1} = (I - DL)^{-1}C\big((K_1(H) + K_2(H))y_n - K_2(H)y_{n-1} + \mathrm{e}^{AH}x_n\big) \tag{6.48}$$

and a global step is calculated, with $[K_3 = (I - DL)^{-1}C]$, as,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ y_n \end{bmatrix} = \underbrace{\begin{bmatrix} \mathrm{e}^{AH} & K_1(H)+K_2(H) & -K_2(H) \\ K_3\mathrm{e}^{AH} & K_3(K_1(H)+K_2(H)) & -K_3K_2(H) \\ 0 & I & 0 \end{bmatrix}}_{\Psi(H)} \begin{bmatrix} x_n \\ y_n \\ y_{n-1} \end{bmatrix}. \tag{6.49}$$

Considering, $\Psi(H)$, as before, in the limit as the step size tends to zero,

$$\lim_{H\to 0} \Psi(H) = \begin{bmatrix} I & 0 & 0 \\ (I - DL)^{-1}C & 0 & 0 \\ 0 & I & 0 \end{bmatrix} \tag{6.50}$$

with the eigenvalues,

$$\lambda_{1,\ldots,k} = 1, \quad \lambda_{k+1,\ldots,k+2l} = 0 \tag{6.51}$$

where k are the total number of states and l are the total number of outputs. In this case we have unconditionally coupling stability, just as in the constant extrapolation case, for the consistent approach.

6.1.2 Summary

In this section, various methods for performing co-simulation based on the parallel base algorithm, Algorithm 3, have been introduced and their coupling stability properties have

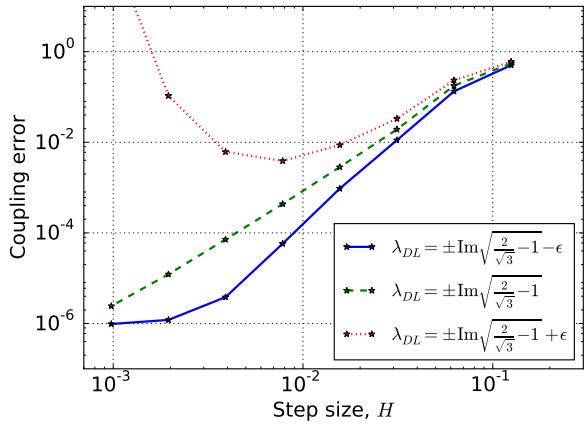

Figure 6.5: *Result of simulating Model 6.24 and Model 6.25 from Example 6.1.1 in parallel using the inconsistent method together with linear extrapolation and for varying values on* d_1 *and* d_2*.*

Table 6.1: *Coupling stability requirement for the different approaches, consistent and inconsistent. This together with extrapolation order and finally if the higher order derivative is approximated using finite differences or if it is available analytically.*

	Consistent		**Inconsistent**	
Order \ Type	Analytical	Finite Diff.	Analytical	Finite Diff.
Constant	Unconditionally coupling stable		$\rho(DL) \leq 1$	
Linear	Unconditionally coupling stable		$\rho(DL) \leq 1$	$\rho\begin{pmatrix} 2DL & -DL \\ I & 0 \end{pmatrix} \leq 1$, (Eq. 6.41)

been analyzed. There are two groups of method variations, one based on the consistent approach, Equation 6.17, where the outputs are solved for and one based on the inconsistent approach, Equation 6.7, where the outputs are computed using previous values in an explicit sequence. Additionally, coupled to both, there is a possibility to use higher order extrapolation for the inputs which are either available explicitly or approximated. In Table 6.1, the requirements for coupling stability is summarized for the different methods.

6.2 Linear correction

A simulation of coupled systems may become unstable if the algebraic constraints are not satisfied, recall the condition on $\rho(DL)$ in the inconsistent approaches from Section 6.1. In [41], Kübler et al. proposed to solve the algebraic constraints in each global step. However, this is not possible when using models following the FMI standard due to that the outputs, y, of a model cannot be recomputed with updated inputs without taking a step (Restriction 2.1). This is a restriction in the standard which results in that only the inconsistent

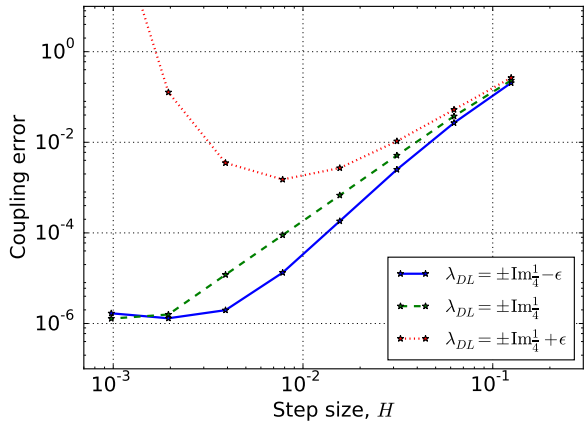

Figure 6.6: *Result of simulating Model 6.24 and Model 6.25 from Example 6.1.1 in parallel using the inconsistent method together with quadratic extrapolation and for varying values on* d_1 *and* d_2.

methods from Section 6.1 can be considered when performing co-simulation using FMI.

However, with the restriction in mind, a linear correction algorithm is proposed in Algorithm 5 which stabilizes the inconsistent approach. The idea can be viewed as performing

Algorithm 5 Parallel Linear Correction, $(T_n \to T_{n+1})$

Require: The models and their connections.

1: **for** $i = 1$ to N **do**
2: Set the input to the ith model, $u_n^{[i]}$.
3: Perform global time step, $T_n \to T_{n+1}$ for the ith model.
4: **end for**
5: **for** $i = 1$ to N **do**
6: Retrieve model outputs, $y_{n+1}^{[i]}$.
7: Retrieve feed-through matrix, $\frac{\partial g^{[i]}}{\partial u^{[i]}}$.
8: **end for**
9: Assemble $\frac{\partial g}{\partial u}$ and $\frac{\partial c}{\partial y}$.
10: Correct y_{n+1}, $z_{n+1} = y_{n+1} - \frac{\partial g}{\partial u} u_n$
11: Solve $\bar{y}_{n+1} = (I - \frac{\partial g}{\partial u}\frac{\partial c}{\partial y})^{-1} z_{n+1}$
12: Compute $u_{n+1} = c(\bar{y}_{n+1})$

a single step of Newton iteration in each global step. In the linear case, this corresponds to the consistent approach.

Consider a coupled system on the form,

$$\dot{x} = f(x, u) \tag{6.52a}$$
$$y = \hat{g}(x) + Du \tag{6.52b}$$
$$u = Ly. \tag{6.52c}$$

In the inconsistent approach, Equation 6.7, the instabilities occur due to the direct feed-through determined by D and L, Section 6.1. In order to stabilize the simulation, the algebraic equation need to be solved, i.e,

$$y = \hat{g}(x) + DLy \quad \rightarrow \quad y = (I - DL)^{-1}\hat{g}(x). \tag{6.53}$$

Using the proposed linear correction algorithm (Algorithm 5) for the coupled semilinear system, Equation 6.52, a step results in,

$$x_{n+1} = \Phi(x_n, u_n) \tag{6.54a}$$
$$y_{n+1} = \hat{g}(x_{n+1}) + Du_n \tag{6.54b}$$
$$z_{n+1} = y_{n+1} - Du_n \tag{6.54c}$$
$$\bar{y}_{n+1} = (I - DL)^{-1} z_{n+1} \tag{6.54d}$$
$$u_{n+1} = L(\bar{y}_{n+1}). \tag{6.54e}$$

It may seem redundant that first Du_n is added for y_{n+1} and then subtracted from z_{n+1}. However, this is due to the fact that y_{n+1} is retrieved from each individual subsystem in a co-simulation setup and that $\hat{g}(x_{n+1})$ is not directly available. But, as we know both D (Feature 2.2) and the input, u_n, $\hat{g}(x_{n+1})$ can be computed implicitly by subtracting Du_n from y_{n+1}. We call the above the linearly corrected inconsistent approach.

Now, simplifying the equations for $\bar{y}_{n+1}$,

$$\bar{y}_{n+1} = (I - DL)^{-1} z_{n+1} = (I - DL)^{-1}(y_{n+1} - Du_n)$$
$$= (I - DL)^{-1}(\hat{g}(x_{n+1}) + Du_n - Du_n) = (I - DL)^{-1}\hat{g}(x_{n+1}). \tag{6.55}$$

Comparing with Equation 6.53, we note that this approach solves the algebraic equation exactly as the system considered here was assumed to be semilinear. This approach results in, for linear coupled systems, equivalence with the consistent approach, Equation 6.17, which do not impose a restricting for coupling stability. In Example 6.2.1, linear correction is used to stabilize a simulation of a double pendulum.

Other stabilization techniques have been proposed. In [64], an overlapping approach was proposed which was based on applying the constraint equations in a differential algebraic equation to more than one subsystem. In [10] a stabilization based on derivative information was proposed for a staggered scheme while in [71] an implementation of a stabilization method was tested with the limitations that no algebraic loops should be present in the coupled system.

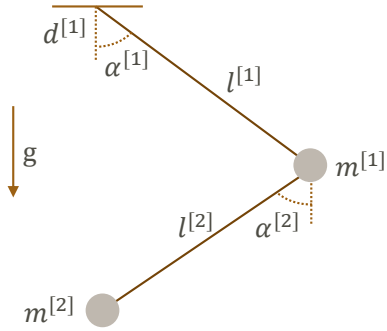

Figure 6.7: *The double pendulum from Example 6.2.1.*

Example 6.2.1 (Double pendulum). *In this example, a double pendulum (Figure 6.7) is simulated using a co-simulation approach. The double pendulum is divided into two systems. The top pendulum is defined by,*

$$\dot{\alpha}^{[1]} = \omega^{[1]} \tag{6.56a}$$

$$\dot{\omega}^{[1]} = \frac{1}{m^{[1]} {l^{[1]}}^2}\left[u_1^{[1]} l^{[1]} \cos\alpha^{[1]} + (u_2^{[1]} - m^{[1]}\mathrm{g}) l^{[1]} \sin\alpha^{[1]} - d^{[1]}\omega^{[1]}\right] \tag{6.56b}$$

with the outputs,

$$y_1^{[1]} = \frac{\cos\alpha^{[1]}}{m^{[1]}}\left[u_1^{[1]} \cos\alpha^{[1]} + (u_2^{[1]} - m^{[1]}\mathrm{g}) \sin\alpha^{[1]}\right] - {\omega^{[1]}}^2 l^{[1]} \sin\alpha^{[1]} \tag{6.57a}$$

$$y_2^{[1]} = \frac{\sin\alpha^{[1]}}{m^{[1]}}\left[u_1^{[1]} \cos\alpha^{[1]} + (u_2^{[1]} - m^{[1]}\mathrm{g}) \sin\alpha^{[1]}\right] + {\omega^{[1]}}^2 l^{[1]} \cos\alpha^{[1]}. \tag{6.57b}$$

The bottom pendulum is defined by,

$$\dot{\alpha}^{[2]} = \omega^{[2]} \tag{6.58a}$$

$$\dot{\omega}^{[2]} = \frac{1}{l^{[2]}}\left[-\mathrm{g}\sin\alpha^{[2]} - u_1^{[2]} \cos\alpha^{[2]} - u_2^{[2]} \sin\alpha^{[2]}\right] \tag{6.58b}$$

with the outputs,

$$y_1^{[2]} = -m^{[2]}\left[u_1^{[2]} \sin\alpha^{[2]} - u_2^{[2]} \cos\alpha^{[2]} - l^{[2]}{\omega^{[2]}}^2 - \mathrm{g}\cos\alpha^{[2]}\right] \sin\alpha^{[2]} \tag{6.59a}$$

$$y_2^{[2]} = m^{[2]}\left[u_1^{[2]} \sin\alpha^{[2]} - u_2^{[2]} \cos\alpha^{[2]} - l^{[2]}{\omega^{[2]}}^2 - \mathrm{g}\cos\alpha^{[2]}\right] \cos\alpha^{[2]}. \tag{6.59b}$$

In Table 6.2 the parameters and start values are shown.

Table 6.2: *Parameters used in Example 6.2.1 together with start values.*

$\alpha^{[1]}(t_0) = -1$ rad	$l^{[1]} = 1$ m	$m^{[1]} = 1$ kg	$d^{[1]} = 0.5$ Nms
$\alpha^{[2]}(t_0) = 0.3$ rad	$l^{[2]} = 1$ m	$m^{[2]} = 1.5$ kg	

In [41], it was shown that if $m^{[1]} < m^{[2]}$, the coupled system is not coupling stable in case of constant extrapolation and using the inconsistent approach.

The coupled system is simulated using $H = 0.01$ together with constant extrapolation and using the inconsistent approach, with and without correction. In Figure 6.8, the results are shown. As can been seen from the figures, the simulation is unstable when correction is not used and stable when it is used. In Figure 6.9, the coupling error is shown, when using linear correction, for decreasing step sizes.

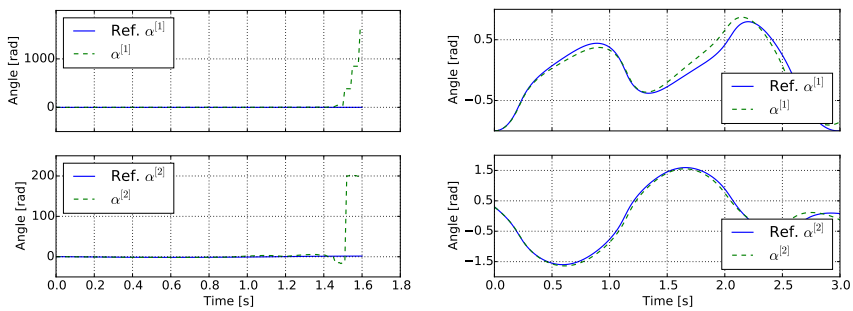

Figure 6.8: *Simulation result for Example 6.2.1 when using constant extrapolation and the inconsistent approach (left) and together with linear correction (right). As shown, the simulation is unstable for the non-corrected simulation and stable for the corrected.*

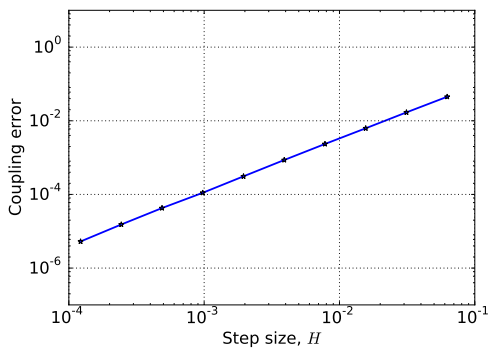

Figure 6.9: *Coupling error in Example 6.2.1 when using linear correction.*

6.2.1 Linear extrapolation

Following the same idea as discussed in the previous section, a linear corrected algorithm when using linear extrapolation of the inputs together with the inconsistent approach is given in Algorithm 6.

Algorithm 6 Parallel Linear Correction with Linear Extrapolation, ($T_n \to T_{n+1}$)

Require: The models and their connections.

1: **for** $i = 1$ to N **do**
2: Set the input to the ith model, $u_n^{[i]}$.
3: Set the input derivative to the ith model, $\frac{\mathrm{d}u_n^{[i]}}{\mathrm{d}t}$.
4: Perform global time step, $T_n \to T_{n+1}$ for the ith model.
5: **end for**
6: **for** $i = 1$ to N **do**
7: Retrieve model outputs, $y_{n+1}^{[i]}$.
8: Retrieve model output derivatives, $\frac{\mathrm{d}y_{n+1}^{[i]}}{\mathrm{d}t}$.
9: Retrieve feed-through matrix, $\frac{\partial g^{[i]}}{\partial u^{[i]}}$.
10: **end for**
11: Assemble $\frac{\partial g}{\partial u}$ and $\frac{\partial c}{\partial y}$.
12: Set, $u_{n,\text{est}} = u_n + H\frac{\mathrm{d}u_n}{\mathrm{d}t}$
13: Correct y_{n+1}, $z_{n+1} = y_{n+1} - \frac{\partial g}{\partial u}u_{n,\text{est}}$
14: Correct $\frac{\mathrm{d}y_{n+1}}{\mathrm{d}t}$, $\frac{\mathrm{d}z_{n+1}}{\mathrm{d}t} = \frac{\mathrm{d}y_{n+1}}{\mathrm{d}t} - \frac{\partial g}{\partial u}\frac{\mathrm{d}u_n}{\mathrm{d}t}$
15: Solve $\bar{y}_{n+1} = (I - \frac{\partial g}{\partial u}\frac{\partial c}{\partial y})^{-1}z_{n+1}$
16: Solve $\frac{\mathrm{d}\bar{y}_{n+1}}{\mathrm{d}t} = (I - \frac{\partial g}{\partial u}\frac{\partial c}{\partial y})^{-1}\frac{\mathrm{d}z_{n+1}}{\mathrm{d}t}$
17: Compute $u_{n+1} = c(\bar{y}_{n+1})$
18: Compute $\frac{\mathrm{d}u_{n+1}}{\mathrm{d}t} = \frac{\partial c}{\partial y}\frac{\mathrm{d}\bar{y}_{n+1}}{\mathrm{d}t}$

Consider the semilinear problem,

$$\dot{x} = f(x, u) \tag{6.60a}$$
$$y = \hat{g}(x) + Du \tag{6.60b}$$
$$u = Ly. \tag{6.60c}$$

In the linear extrapolation case, the state is calculated depending on the inputs at T_n and additionally on their derivatives at T_n, i.e.

$$x_{n+1} = \Phi(x_n, u_n, \dot{u}_n). \tag{6.61}$$

This in turn results in,

$$y_{n+1} = \hat{g}(x_{n+1}) + Du_{n,\text{est}} \tag{6.62}$$

where (H is the global step size),

$$u_{n,\text{est}} = u_n + H\dot{u}_n. \tag{6.63}$$

The proposed linear correction for the linear extrapolation case results in that a step is computed as,

$$\begin{aligned}
x_{n+1} &= \Phi(x_n, u_n, \dot{u}_n) && \text{(6.64a)}\\
y_{n+1} &= \hat{g}(x_{n+1}) + Du_{n,\text{est}} && \text{(6.64b)}\\
\dot{y}_{n+1} &= \dot{\hat{g}}(x_{n+1}) + D\dot{u}_n && \text{(6.64c)}\\
z_{n+1} &= y_{n+1} - Du_{n,\text{est}} && \text{(6.64d)}\\
\dot{z}_{n+1} &= \dot{y}_{n+1} - D\dot{u}_n && \text{(6.64e)}\\
\bar{y}_{n+1} &= (I - DL)^{-1} z_{n+1} && \text{(6.64f)}\\
\dot{\bar{y}}_{n+1} &= (I - DL)^{-1} \dot{z}_{n+1} && \text{(6.64g)}\\
u_{n+1} &= L(\bar{y}_{n+1}) && \text{(6.64h)}\\
\dot{u}_{n+1} &= L(\dot{\bar{y}}_{n+1}). && \text{(6.64i)}
\end{aligned}$$

Instead of, as in Algorithm 5, correcting the outputs, y, using u_n, the outputs is here corrected using the estimated inputs, $u_{n,\text{est}}$, resulting in,

$$\begin{aligned}
\bar{y}_{n+1} = (I - DL)^{-1} z_{n+1} = (I - DL)^{-1}(y_{n+1} - Du_{n,\text{est}}) = \\
(I - DL)^{-1}(\hat{g}(x_{n+1}) + Du_{n,\text{est}} - Du_{n,\text{est}}) = (I - DL)^{-1}\hat{g}(x_{n+1}).
\end{aligned} \tag{6.65}$$

Furthermore, a correction of the input derivatives is motivated due to that,

$$\dot{u}_{n+1} = L\dot{y}_{n+1} = L(I - DL)^{-1}\frac{\mathrm{d}\hat{g}(x)}{\mathrm{d}x}\dot{x}_{n+1}. \tag{6.66}$$

Considering now the corrected output derivatives in the proposed algorithm,

$$\begin{aligned}
\dot{\bar{y}}_{n+1} = (I - DL)^{-1}\dot{z}_{n+1} = (I - DL)^{-1}(\dot{y}_{n+1} - D\dot{u}_n) = \\
(I - DL)^{-1}(\frac{\mathrm{d}\hat{g}}{\mathrm{d}x}\dot{x}_{n+1} + D\dot{u}_n - D\dot{u}_n) = (I - DL)^{-1}\frac{\mathrm{d}\hat{g}}{\mathrm{d}x}\dot{x}_{n+1}
\end{aligned} \tag{6.67}$$

and thus,

$$\dot{u}_{n+1} = L(I - DL)^{-1}\frac{\mathrm{d}\hat{g}}{\mathrm{d}x}\dot{x}_{n+1}. \tag{6.68}$$

In Example 6.2.2, the double pendulum is revisited and simulated using the inconsistent approach with linear extrapolation, with and without correction.

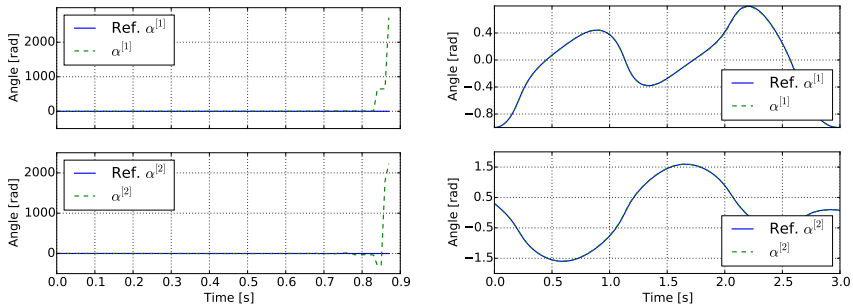

Figure 6.10: *Simulation result for Example 6.2.2 when using linear extrapolation and the inconsistent approach (left) and together with linear correction (right). As shown, the simulation is unstable for the non-corrected simulation and stable for the corrected.*

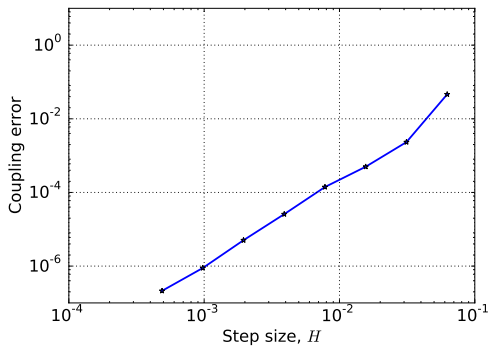

Figure 6.11: *Shown is the coupling error in Example 6.2.2 when using linear correction for the inconsistent approach with linear extrapolation.*

Example 6.2.2 (Double pendulum (revisited)). *In this example, we revisit Example 6.2.1 and run the same experiments but instead of using constant extrapolation, we now use linear extrapolation.*

In Figure 6.10, the results are shown. As can been seen from the figures, the simulation is unstable, as before, when correction is not used and stable when it is used. In Figure 6.11, the coupling error is shown.

Higher order extrapolation schemes using linear correction is currently not possible due to limitations in the FMI standard. For higher order extrapolation, derivatives of the states are necessary for the correction which are not available as they are hidden inside the model and not exposed to the simulation tools.

6.3 Smoothing of connections

Using the inconsistent approach with linear extrapolation for co-simulation, with and without linear correction, discontinuities are introduced in the coupling inputs, u, and thus propagated to the underlying subsystems due to $u_{n,\text{est}} \neq u_{n+1}$, cf. Figure 6.12. These discontinuities may result in a degradation of the performance of the local solver. Consider the case when a multistep method [29] is used in the subsystems, then the introduction of a discontinuity in the inputs, $u_n^{[i]}$, may result in order reductions and in worst case simulation failure.

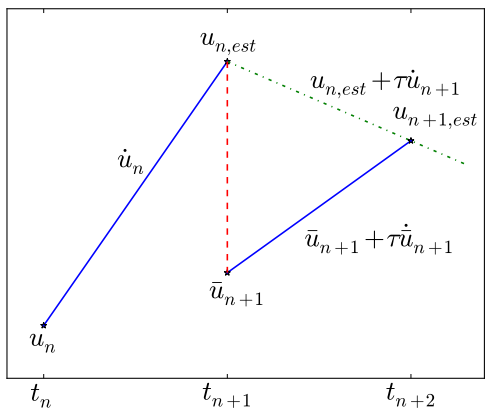

Figure 6.12: *Illustration of smoothing. The red dashed line is the jump in the inputs if no smoothing is used while the dashed dotted green line is the smooth input.*

Instead of directly propagating u_{n+1} and $\dot{u}_{n+1}$ in the subsequent step, we propose using $u_{n,\text{est}}$ instead and modify the derivatives so that still $u_{n+2} = u_{n+1} + H\dot{u}_{n+1}$ is reached. This results in that we preserve continuity ($\mathcal{C}^0$) of the inputs and thus do not introduce jump discontinuities ($\mathcal{C}^{-1}$) to the underlying solvers. In Example 6.3.1, an motivating example is shown.

Another smoothing approach can be found in [63].

Example 6.3.1 (Motivating example - Linear extrapolation with and without smoothing). *Consider the two models,*

$$\dot{x}^{[1]} = -x^{[1]}{u^{[1]}}^2 \qquad \dot{x}^{[2]} = -x^{[2]} + u^{[2]} \tag{6.69a}$$

$$y^{[1]} = x^{[1]} \qquad y^{[2]} = x^{[2]} + \sin(u^{[2]}) \tag{6.69b}$$

with the coupling,

$$u^{[2]} = y^{[1]} \quad u^{[1]} = y^{[2]}. \tag{6.70}$$

The coupled system is simulated using the inconsistent approach with linear extrapolation, with and without smoothing, where the underlying solver is, in both subsystems, CVode.

Table 6.3: *Statistics of the underlying solver for both subsystems from Example 6.3.1, with and without smoothing, using the same global step size,* $H = 0.02$.

	#steps	#fevals	#steps (with)	#fevals (with)
Subsystem [1]	423	677	258	350
Subsystem [2]	291	494	185	262

Investigating the impact of the smoothing on the underlying solvers we see a big decrease in the number of local steps and local function evaluations compared to the case without smoothing. In Table 6.3 the statistics is shown for the local models. We see that the number of steps has nearly been decreased by a factor of two. In Figure 6.13, the inputs for subsystem [1] *are shown and compared to the reference, highlighting the difference of using smoothing versus not using it. In Figure 6.14 the error is shown in the states for the two simulations. We see that the error is*

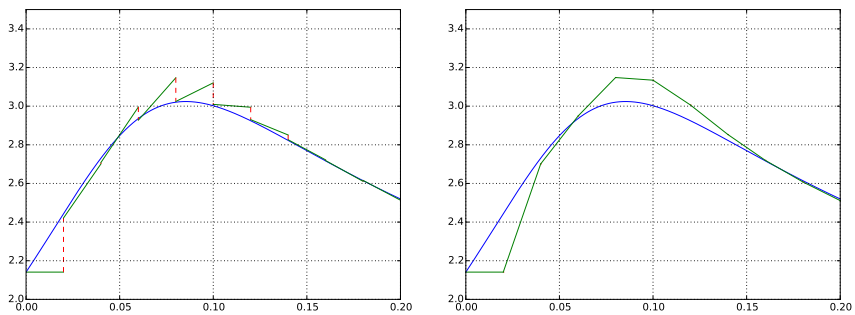

Figure 6.13: *The input for subsystem* [1] *in Example 6.3.1 when using smoothing (right) vs only using linear extrapolation (left). The figure clearly shows the introduced discontinuities in the inputs in the case when only linear extrapolation is used (left).*

larger for the simulation using smoothing. However, as the amount of work for the subsystems was substantially decreased the global step size can be decreased while still doing less work than in the non smoothing case. The question arises, can we decrease it enough for the error to be equal while still doing less work?

In Figure 6.15 the error is shown when the global step size has been adapted so that the error of both simulations are equal (step size 0.0145 *vs* 0.02*). In Table 6.4, the local simulation statistics is shown. Noted from the table is that the number of steps and function evaluations in the smoothing case remains significantly lower.*

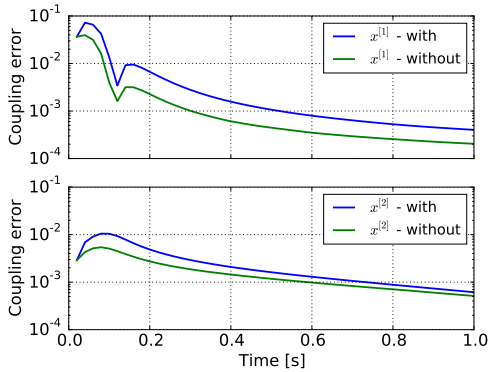

Figure 6.14: *Error in the states when simulating the coupled system from Example 6.3.1 with and without smoothing. Here, the step size is set to* 0.02 *in both cases.*

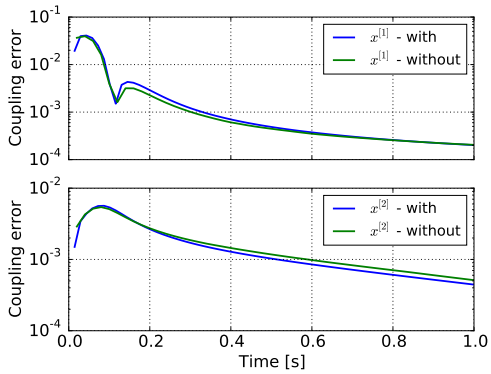

Figure 6.15: *Error in the states when simulating the coupled system from Example 6.3.1 using smoothing with a global step size set to* 0.0145 *and without smoothing with a global step size set to* 0.02.

Smoothing for the inconsistent approach, with linear extrapolation, is defined by,

$$x_{n+1} = \Phi(x_n, u_n, \dot{u}_n) \tag{6.71a}$$

$$\bar{y}_{n+1} = Cx_{n+1} + D\Big(u_n + H\dot{u}_n\Big) \tag{6.71b}$$

$$\dot{\bar{y}}_{n+1} = C\dot{x}_{n+1} + D\dot{u}_n \tag{6.71c}$$

$$u_{n+1} = u_n + H\dot{u}_n \tag{6.71d}$$

$$\dot{u}_{n+1} = \frac{L\bar{y}_{n+1} + LH\dot{\bar{y}}_{n+1} - u_{n+1}}{H}. \tag{6.71e}$$

Note that the updated inputs, u_{n+1}, are set depending on the previous inputs. The input derivatives, $\dot{u}_{n+1}$ are modified so that,

$$u_{n+2} = u_{n+1} + H\dot{u}_{n+1} = L\bar{y}_{n+1} + LH\dot{\bar{y}}_{n+1}. \tag{6.72}$$

Table 6.4: *Statistics of the underlying solver for both subsystems from Example 6.3.1, with and without smoothing. The step size for the simulation with smoothing is chosen so that both simulations give the same solution ($H = 0.0145$ vs $H = 0.02$).*

	#steps	#fevals	#steps (with)	#fevals (with)
Subsystem [1]	423	677	279	376
Subsystem [2]	291	494	191	259

Eliminating u, as before, we get,

$$x_{n+1} = \Phi(x_n, Ly_n, L\dot{y}_n) \tag{6.73a}$$
$$y_{n+1} = y_n + H\dot{y}_n \tag{6.73b}$$
$$\dot{y}_{n+1} = \frac{Cx_{n+1} + DLy_n - y_{n+1}}{H} + C\dot{x}_{n+1} + DL\dot{y}_n. \tag{6.73c}$$

Furthermore, we need $\dot{x}_{n+1}$ which, from Equation 6.4a and due to linear extrapolation of the inputs, is given by,

$$\dot{x}_{n+1} = Ax_{n+1} + BL(y_n + H\dot{y}_n). \tag{6.74}$$

Additionally, as the output derivatives, $\dot{y}_{n+1}$, only enters the equations together with the step size, H, we introduce,

$$\dot{z}_{n+1} = H\dot{y}_{n+1}. \tag{6.75}$$

As in Section 6.1, we are interested in an iteration matrix, $\Psi(H)$, such that,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ \dot{z}_{n+1} \end{bmatrix} = \Psi(H) \begin{bmatrix} x_n \\ y_n \\ \dot{z}_n \end{bmatrix}. \tag{6.76}$$

In this case, the iteration matrix is defined by,

$$\Psi(H) = \begin{bmatrix} \mathrm{e}^{AH} & K_1(H) & K_2(H) \\ 0 & I & I \\ C(I+AH)\mathrm{e}^{AH} & C(I+AH)K_1(H)+K_4(H) & C(I+AH)K_2(H)+K_4(H) \end{bmatrix} \tag{6.77}$$

with

$$K_4(H) = DL - I + HCBL \quad \text{and} \quad \lim_{H\to 0} K_4(H) = DL - I. \tag{6.78}$$

We consider, $\Psi(H)$, in the limit as the step size, H, tends to zero,

$$\lim_{H\to 0} \Psi(H) = \begin{bmatrix} I & 0 & 0 \\ 0 & I & I \\ C & DL - I & DL - I \end{bmatrix}. \tag{6.79}$$

The iteration matrix above leads to coupling stability as long as $\rho(DL) \leq 1$. Note that this is the same requirement as in the approach defined by Equation 6.42, i.e. the inconsistent approach with linear extrapolation using analytical derivatives.

In the same way as above, we define smoothing for the consistent algorithm with linear extrapolation. The algorithm is defined by,

$$x_{n+1} = \Phi(x_n, u_n, \dot{u}_n) \tag{6.80a}$$
$$\bar{y}_{n+1} = (I - DL)^{-1} C x_{n+1} \tag{6.80b}$$
$$\dot{\bar{y}}_{n+1} = (I - DL)^{-1} C \dot{x}_{n+1} \tag{6.80c}$$
$$u_{n+1} = u_n + H\dot{u}_n \tag{6.80d}$$
$$\dot{u}_{n+1} = \frac{L\bar{y}_{n+1} + LH\dot{\bar{y}}_{n+1} - u_{n+1}}{H}. \tag{6.80e}$$

Note that the inputs, u_{n+1}, are not set to the consistent values, $\bar{y}_{n+1}$. The input derivatives, $\dot{u}_{n+1}$, are set according to Equation 6.72. By eliminating the inputs as before we get,

$$x_{n+1} = \Phi(x_n, Ly_n, L\dot{y}_n) \tag{6.81a}$$
$$y_{n+1} = y_n + H\dot{y}_n \tag{6.81b}$$
$$\dot{y}_{n+1} = \frac{(I - DL)^{-1} C x_{n+1} - y_{n+1}}{H} + (I - DL)^{-1} \dot{x}_{n+1}. \tag{6.81c}$$

With Equation 6.75, the iteration matrix, $\Psi(H)$, is defined by,

$$\Psi(H) = \begin{bmatrix} \mathrm{e}^{AH} & K_1(H) & K_2(H) \\ 0 & I & I \\ K_3[I+AH]\mathrm{e}^{AH} & K_3[(I+AH)K_1(H)+HBL)-I & K_3[(I+AH)K_2(H)+HBL]-I \end{bmatrix}. \tag{6.82}$$

We consider, $\Psi(H)$, in the limit as the step size, H, tends to zero and we get,

$$\lim_{H \to 0} \Psi(H) = \begin{bmatrix} I & 0 & 0 \\ 0 & I & I \\ K_3 & -I & -I \end{bmatrix}. \tag{6.83}$$

Here, the approach is unconditionally coupling stable, as in the consistent approaches.

The above smoothing leads to a proposed linear correction algorithm with linear extrapolation and smoothing which is shown in Algorithm 7.

Algorithm 7 Parallel Linear Correction with Linear Extrapolation and Smoothing, ($T_n \rightarrow T_{n+1}$)

Require: The models and their connections.

1: **for** $i = 1$ to N **do**
2: Set the input to the ith model, $u_n^{[i]}$.
3: Set the input derivative to the ith model, $\frac{\mathrm{d}u_n^{[i]}}{\mathrm{d}t}$.
4: Perform global time step, $T_n \rightarrow T_{n+1}$ for the ith model.
5: **end for**
6: **for** $i = 1$ to N **do**
7: Retrieve model outputs, $y_{n+1}^{[i]}$.
8: Retrieve model output derivatives, $\frac{\mathrm{d}y_{n+1}^{[i]}}{\mathrm{d}t}$.
9: Retrieve feed-through matrix, $\frac{\partial g^{[i]}}{\partial u^{[i]}}$.
10: **end for**
11: Assemble $\frac{\partial g}{\partial u}$ and $\frac{\partial c}{\partial y}$.
12: Set, $u_{n,\text{est}} = u_n + H\frac{\mathrm{d}u_n}{\mathrm{d}t}$
13: Correct y_{n+1}, $z_{n+1} = y_{n+1} - \frac{\partial g}{\partial u}u_{n,\text{est}}$
14: Correct $\frac{\mathrm{d}y_{n+1}}{\mathrm{d}t}$, $\frac{\mathrm{d}z_{n+1}}{\mathrm{d}t} = \frac{\mathrm{d}y_{n+1}}{\mathrm{d}t} - \frac{\partial g}{\partial u}\frac{\mathrm{d}u_n}{\mathrm{d}t}$
15: Solve $\bar{y}_{n+1} = (I - \frac{\partial g}{\partial u}\frac{\partial c}{\partial y})^{-1} z_{n+1}$
16: Solve $\frac{\mathrm{d}\bar{y}_{n+1}}{\mathrm{d}t} = (I - \frac{\partial g}{\partial u}\frac{\partial c}{\partial y})^{-1}\frac{\mathrm{d}z_{n+1}}{\mathrm{d}t}$
17: Compute $u_{n+1} = u_{n,\text{est}}$
18: Compute $\frac{\mathrm{d}u_{n+1}}{\mathrm{d}t} = c(\bar{y}_{n+1}) - u_{n,\text{est}} + \frac{\partial c}{\partial y}\frac{\mathrm{d}\bar{y}_{n+1}}{\mathrm{d}t}$

6.4 Case studies

6.4.1 Mass-spring-damper

Consider a mass-spring-damper problem with three coupled springs and three coupled dampers which are connected between two fixed points and two masses, cf. Figure 6.16. The problem is governed by the equations,

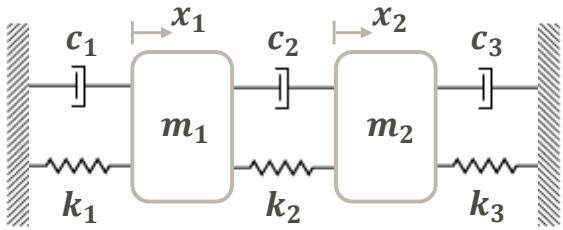

Figure 6.16: *Mass-spring-damper problem.*

$$m_1\ddot{x}_1 = -(k_1 + k_2)x_1 - (c_1 + c_2)\dot{x}_1 + k_2x_2 + c_2\dot{x}_2 \tag{6.84a}$$
$$m_2\ddot{x}_2 = -(k_3 + k_2)x_2 - (c_3 + c_2)\dot{x}_2 + k_2x_1 + c_2\dot{x}_1. \tag{6.84b}$$

The problem is divided into two subsystems. The first is defined by,

$$m_1\ddot{x}_1 = -k_1x_1 - c_1\dot{x}_1 + k_2u_{11} + c_2u_{12} \tag{6.85a}$$
$$y_1 = [x_1, \dot{x}_1]^{\mathrm{T}}. \tag{6.85b}$$

The input is the difference in position and velocity between the two subsystems and the output is the position and velocity. In the second subsystem, the output is the difference between the position and the velocity,

$$m_2\ddot{x}_2 = -(k_3 + k_2)x_2 - (c_3 + c_2)\dot{x}_2 + k_2u_{21} + c_2u_{22} \tag{6.86a}$$
$$y_2 = [x_2, \dot{x}_2]^{\mathrm{T}} - [u_{21}, u_{22}]^{\mathrm{T}}. \tag{6.86b}$$

The input for this second subsystem is the position and velocity of the first subsystem. This leads to the coupling equation,

$$u_{11} = y_{21}, \qquad u_{21} = y_{11} \tag{6.87a}$$
$$u_{12} = y_{22}, \qquad u_{22} = y_{12}. \tag{6.87b}$$

The coupled system is first simulated with three different approaches, all using extrapolation order one and a fixed step size of $H = 0.085$. The intention is to highlight different stability properties of the algorithms for $H > 0$. A reference solution was computed from an FMU

of the monolithic system, Equation 6.84, exported using JModelica.org. Using PyFMI and Assimulo, the reference was computed with CVode [33] using relative and absolute tolerance set to 10^{-10}.

In Figure 6.17 the coupled system is simulated using the inconsistent approach while in Figure 6.18, the system is simulated with smoothing, with and without linear correction. These figures highlights that there are stability differences between using the different approaches. However, in Figure 6.19 the same approaches as above are used in simulations as $H \to 0$ and the expected results are achieved.

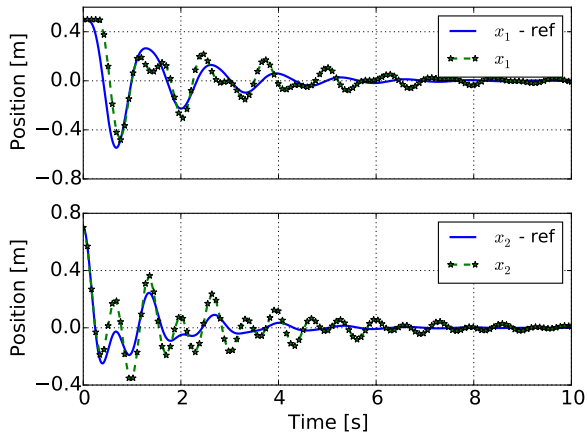

Figure 6.17: *Simulation result for Section 6.4.1 when using the inconsistent approach with step size* $H = 0.085$.

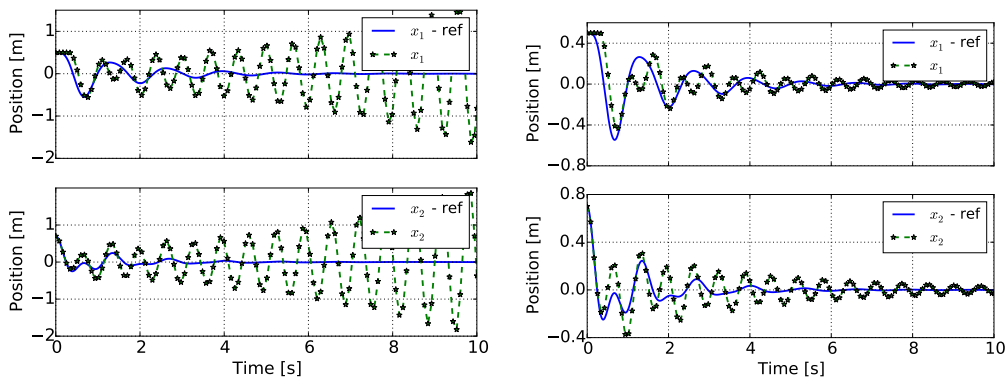

Figure 6.18: *Simulation result for Section 6.4.1 when using the inconsistent approach with smoothing (left) and when using smoothing together with linear correction (right). The step size used:* $H = 0.085$.

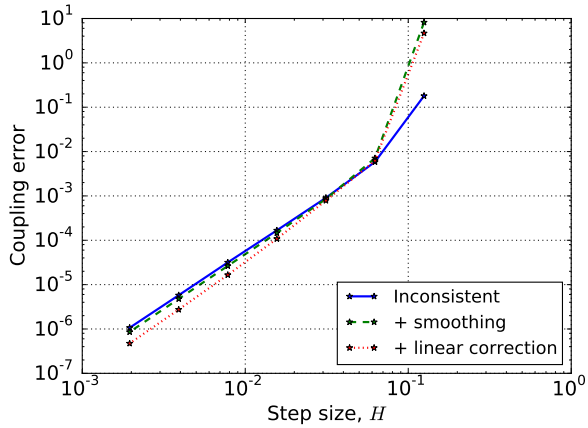

Figure 6.19: *Coupling error in Section 6.4.1 when simulated using the inconsistent method, the inconsistent method with smoothing and the inconsistent method with smoothing and linear correction. All simulations were carried out using linear extrapolation.*

6.4.2 Race car

Revisiting the race car from Section 5.3.2 in a co-simulation setting and now considering, instead of initialization, simulation of the coupled systems. In Section 5.3.2, it was found that the couplings did not introduce any algebraic loops and thus $\rho(DL) = 0$ which means that we can expect coupling stability for the inconsistent approaches.

A reference solution was computed from an FMU of the monolithic system, Appendix A.4, exported using Dymola 2016 [20]. Using PyFMI and Assimulo, the reference was computed with CVode using relative and absolute tolerance set to 10^{-8}.

In Figure 6.20, result of the simulated system is shown when the inconsistent approach was used together with constant extrapolation and step size $H = 0.005$. In Figure 6.21, the coupling error is shown for different orders of extrapolation together with the inconsistent approach, with and without linear correction and in Figure 6.22 with smoothing.

6.5 Summary

In this chapter, algorithms for simulation of weakly coupled systems has been considered. The algorithms are variants of the consistent approach, where the constraints are solved (Equation 6.4b,c), and of the inconsistent approach, where the constraints are not solved. Focus has been on stability in the limit as $H \to 0$. In Table 6.1, the conditions for stability for the different algorithms are shown. Furthermore, in Section 6.3, smoothing of the coupling variables was considered. In Table 6.5, the conditions for stability is shown for these cases.

Using the FMI, the consistent approach is not realizable due to Restriction 2.1. To overcome this restriction, a modification of the inconsistent approach was considered in

Section 6.2 and a linearly corrected inconsistent approach was proposed.

Table 6.5: *Coupling stability requirement for the different approaches, consistent and inconsistent, when using smoothing on the coupling variables.*

	Consistent w. smoothing	**Inconsistent w. smoothing**
Order \ Type	Analytical	Analytical
Linear	Unconditionally coupling stable	$\rho(DL) \leq 1$

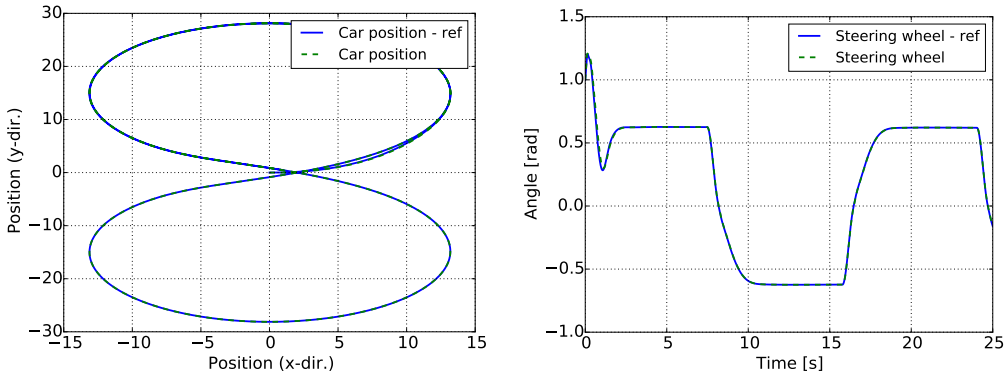

Figure 6.20: *Simulation result for Section 6.4.2 when using the inconsistent algorithm with constant extrapolation and a step size of* $H = 0.005$.

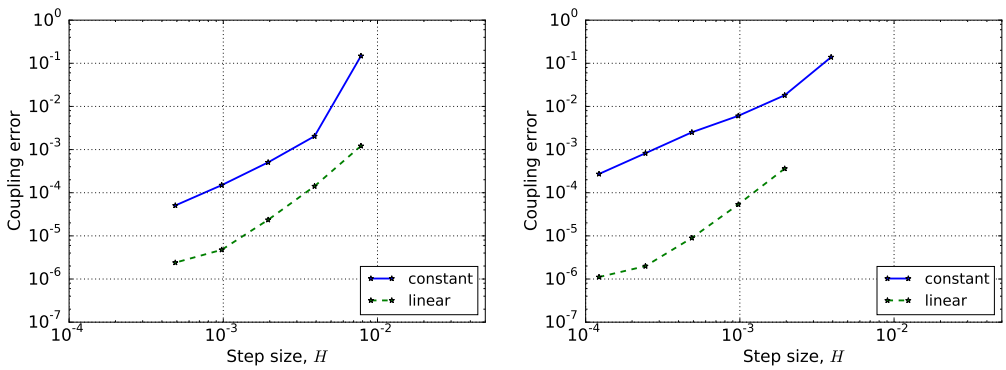

Figure 6.21: *Coupling error result for Section 6.4.2 when using the inconsistent algorithm (left) and when using the inconsistent algorithm with linear correction (right).*

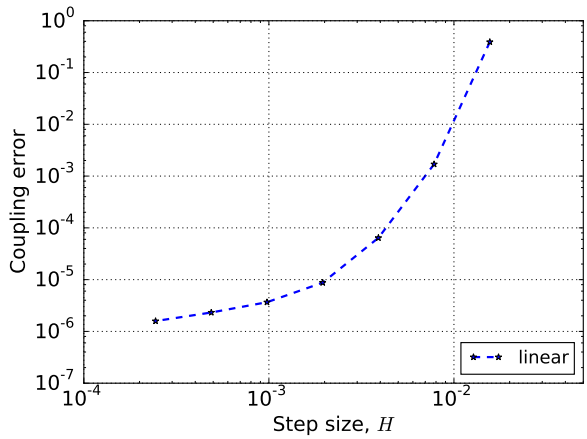

Figure 6.22: *Coupling error result for Section 6.4.2 when using the inconsistent algorithm together with smoothing.*

Chapter 7

Modification of multistep predictor

A common underlying integrator, in a subsystem, is a multistep method [29]. A multistep method uses the solution from previous steps to predict future solutions. It is usually implemented using a variable time grid, which means that the step size of the method adapts to the problem during the simulation. Additionally this is complemented with adapting the order of the method.

Restarting a multistep method used for solving initial value problem is expensive. A restart resets the current integrator order to one and discards the information from previous steps. In certain situations this is necessary, such as when a discontinuity in the states are detected. Here, we intend to use the knowledge of what has changed in the problem and modify the method so that a restart becomes unnecessary.

Instead of considering the fully coupled system, Equation 3.5, we consider in this chapter an initial value problem (IVP) together with an external input $u(t)$,

$$\dot{x}(t) = f(x(t), u(t)), \quad x(T_0) = x_0, \quad t \in [T_0, T_M]. \tag{7.1}$$

The input is assumed to be a given piecewise constant signal, on a global grid, defined as,

$$u(t) = u_i, \quad t \in [T_i, T_{i+1}) \tag{7.2}$$

with $i = 0, \ldots, M-1$. Alternatively, we consider a piecewise linear input signal, possibly discontinues,

$$u(t) = u_i + (t - T_i)\dot{u}_i, \quad t \in [T_i, T_{i+1}]. \tag{7.3}$$

The IVP (7.1), is solved using a variable-step, variable-order multistep method. The assumption is that during a global step, i.e., $t \in [T_i, T_{i+1}]$, the method requires many local steps to satisfy the tolerance requirements.

The above setting is exactly the case when performing simulations of weakly coupled systems. Another situation where the above is relevant is the case when the IVP is coupled

to an external process which is responsible for providing the inputs. Consider a hardware-in-the-loop simulation [17], here the inputs are not known for the complete simulation horizon at initial time (due to that they do not exist), but rather at known time-points where an update occur.

There are two approaches that first come to mind when simulating the IVP using the discussed inputs. Either the method is restarted at each segment or the method proceeds using values computed from the previous segment. In Figure 7.1, a typical plot of the step size history and order history is shown for the two cases. As can be seen, neither are efficient. Both approaches experience order and step size reductions at the global input changes.

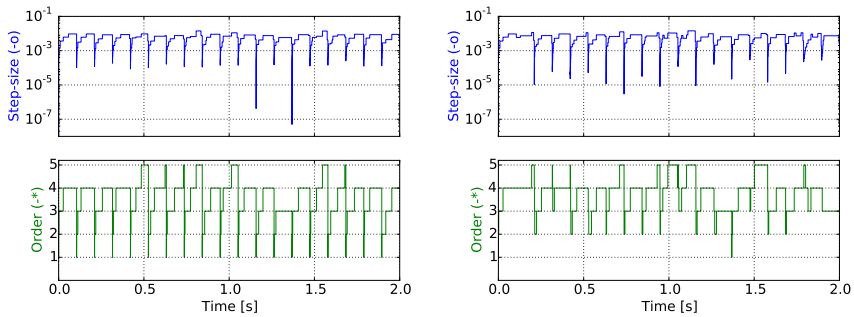

Figure 7.1: *Step size and order for a simulation of an example model using linear input segments. In the left figure, the method is restarted at every global step while in the right figure, it proceeds from stored values.*

In this chapter, an approach is presented that modifies the predictor in a multistep method at the start of each global step. The modification is shown to significantly improve the simulation performance.

7.1 Multistep methods

A general multistep method for solving an IVP is defined as,

$$\sum_{i=0}^{q} \alpha_{n,i} x_{n-i} + h_n \sum_{i=0}^{q} \beta_{n,i} f(x_{n-i}) = 0 \tag{7.4}$$

where q determines the number of steps, i.e. the number of previous solution points used, in the formula. The coefficients α and β determine the method. They are dependent on the step size history and order.

A multistep method needs during the integration access to the previous solutions points, i.e. the solution history. The representation of the history varies between implementations of a multistep method. Most commonly used history representation is via a Nordsieck array or an array of modified divided differences. For a detailed description cf. [68].

In a Nordsieck history array, the history is represented as,

$$z_{n-1} = \left[x_{n-1}, h_n \dot{x}_{n-1}, \ldots, \frac{h_n^q x_{n-1}^{(q)}}{q!}\right]. \tag{7.5}$$

From the solution history z_{n-1}, a prediction to z_n, denoted $z_{n(0)}$, and also x_n ($x_{n(0)}$), is computed. This prediction is used as an initial guess to the nonlinear equation system, resulting from Equation 7.4,

$$G(x_n) = 0. \tag{7.6}$$

In order to accept the computed solution x_n, an error test must be passed,

$$||e_n|| = ||x_n - x_{n(0)}|| \leq \text{TOL}. \tag{7.7}$$

If the error test succeeds for a given tolerance TOL the step is accepted. The above steps are independent of how the solution history is represented.

7.2 Problem formulation

Here we consider the IVP, Equation 7.1, with either a piecewise constant input or a piecewise linear input resulting from a co-simulation. At the start of a global segment $[T_{i+1}, T_{i+2}]$ the solution history of the multistep method is based on,

$$\dot{x} = f(x, u_i), \quad t \in [T_i, T_{i+1}]. \tag{7.8}$$

Once the inputs are updated, at T_{i+1}, the equation,

$$\dot{x} = f(x, u_{i+1}) \tag{7.9}$$

are no longer consistent,

$$\dot{x}_{i+1}^{-} := f(x_{i+1}, u_i,) \quad \neq \quad f(x_{i+1}, u_{i+1}) =: \dot{x}_{i+1}^{+}. \tag{7.10}$$

This means that the solution history is no longer valid and that the the error test (Equation 7.7) is likely to fail due to a poor prediction. Additionally, there is also the problem that the predicted step delivers a bad initial guess for the nonlinear system (Equation 7.6), resulting in convergence failures.

7.3 Modifying the predictor

As previously mentioned, most of the order reductions and step size reductions seen are due to error test failures caused by a poor prediction.

The predicted next step is computed by extrapolating the polynomial defined with the solution history array. By modifying the history array, a better prediction is achieved resulting in a reduced risk for error test failures and convergence failures.

Considering specifically the Nordsieck representation, Equation 7.5, the second term in the array is,

$$z_{i+1,1} = h\dot{x}_{i+1}. \tag{7.11}$$

Here, a correction is computed as,

$$\Delta\dot{x}_{i+1} = \dot{x}^{-}_{i+1} - \dot{x}^{+}_{i+1} \Rightarrow z_{i+1,1} = z_{i+1,1} - h\Delta\dot{x}_{i+1} \Rightarrow z_{i+1,1} = \dot{x}^{+}_{i+1}. \tag{7.12}$$

A single additional function evaluation is required in order to correct the first derivative. For the second derivative,

$$\Delta\ddot{x}_{i+1} = \ddot{x}^{-}_{i+1} - \ddot{x}^{+}_{i+1} = \ddot{x}^{-}_{i+1} - \Big[\frac{\partial f(x_{i+1}, u_{i+1})}{\partial x}\dot{x}^{+}_{i+1} + \frac{\partial f(x_{i+1}, u_{i+1})}{\partial u}\dot{u}_{i+1}\Big] \tag{7.13}$$

we correct by,

$$z_{i+1,2} = z_{i+1,2} - \frac{h^2}{2}\Delta\ddot{x}_{i+1} \Rightarrow \tag{7.14}$$

$$z_{i+1,2} = \Big[\frac{\partial f(x_{i+1}, u_{n+1})}{\partial x}\dot{x}^{+}_{i+1} + \frac{\partial f(x_{i+1}, u_{i+1})}{\partial u}\dot{u}_{i+1}\Big]. \tag{7.15}$$

Correcting the second derivative requires a new Jacobian evaluation. Additionally, if the inputs are linear segments, an evaluation of the partial derivatives with respect to the inputs is necessary.

Higher order corrections are considered too costly and therefore not considered here.

7.4 Case studies

In the experiments below, the models were modeled in the modeling language Modelica and compiled into FMUs [50] using the open-source tool JModelica.org. The multistep method used was CVode to which the correction, discussed in Section 7.3, has been implemented. No changes were made inside CVode. Further, the correction to the predictor does not disable any of the features in CVode for step size reductions or order reductions for cases when the error test still fails or when the nonlinear solver fail to converge.

7.4.1 Mass-spring-damper

Consider a linear spring-damper problem with three coupled springs and three coupled dampers which are connected between two fixed points and two masses, cf. Figure 7.2. The problem is governed by the equations,

$$m_1\ddot{x}_1 = -(k_1 + k_2)x_1 - (c_1 + c_2)\dot{x}_1 + k_2x_2 + c_2\dot{x}_2 \tag{7.16a}$$

$$m_2\ddot{x}_2 = -(k_3 + k_2)x_2 - (c_3 + c_2)\dot{x}_2 + k_2x_1 + c_2\dot{x}_1. \tag{7.16b}$$

The parameters are listed in Table 7.1.

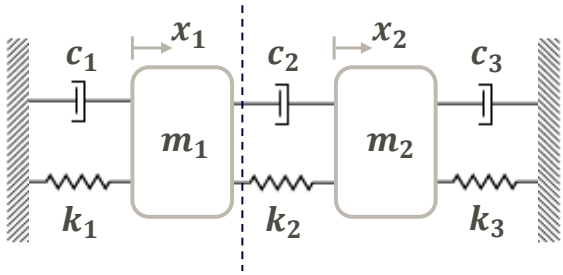

Figure 7.2: *A triple spring, triple damper example.*

Table 7.1: *Parameters used in Section 7.4.1.*

$k_1 = 10\ \mathrm{Nm}^{-1}$	$k_2 = 25\ \mathrm{Nm}^{-1}$	$k_3 = 50\ \mathrm{Nm}^{-1}$	$m_1 = 1\ \mathrm{kg}$
$c_1 = 1\ \mathrm{Nsm}^{-1}$	$c_2 = 0.1\ \mathrm{Nsm}^{-1}$	$c_3 = 2\ \mathrm{Nsm}^{-1}$	$m_2 = 1\ \mathrm{kg}$

Dividing the system into two subsystems along the dotted line in Figure 7.2 results in that the dynamics for the right subsystem is described by.

$$m_2\ddot{x}_2 = -(k_3 + k_2)x_2 - (c_3 + c_2)\dot{x}_2 + k_2u_{21} + c_2u_{22} \tag{7.17a}$$

where u_{21} and u_{22} are inputs due to coupling.

In the following experiments, Equation 7.16, was solved with high accuracy and used as the reference solution. Furthermore, the inputs to the right subsystem were computed from the reference solution. In Figure 7.3 the inputs are shown when using piecewise linear segments.

The right subsystem, Equation 7.17, was simulated for two seconds using twenty piecewise linear segments. The tolerances were set to 10^{-6} for both the relative and absolute tolerance. In Figure 7.4 and Figure 7.5, the step size history and order history is shown when using different approaches for handling the crossing of segments. In Table 7.2, the simulation statistics is shown.

From the statistics we draw the conclusion that a correction in both the first and second derivative is beneficial as it reduced the number of function evaluations and number of steps taken as compared with the restart case. Additionally, the worst choice is to proceed with old values. Considering the figures, we note that CVode, in the corrected case, persistently remain at high order. The global error is on the same magnitude and are nearly equal in all four cases.

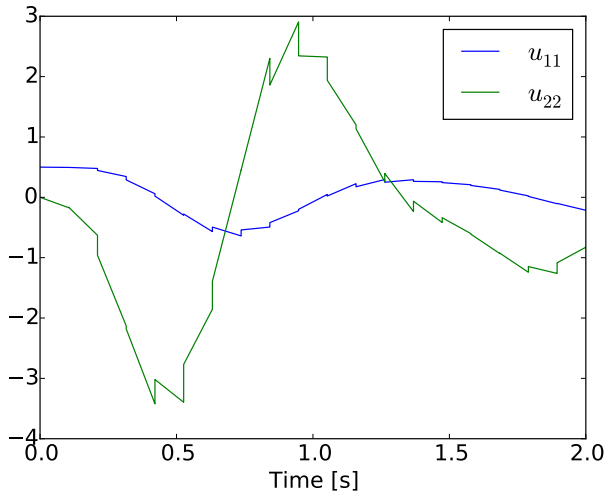

Figure 7.3: *The piecewise linear inputs to Equation 7.17.*

Table 7.2: *Statistics from simulating Equation 7.17 with different options for crossing the global segments. The elapsed time is normalized with respect to the restart option.*

	Restart	Proceed	Pred. Corr. in $\dot{x}$	Pred. Corr. in $\dot{x}, \ddot{x}$
# steps	598	876	584	343
# fevals	847	1257	833	466
# jacs	19	17	11	19
# errfails	22	87	47	18
time	1.00	1.22	0.85	0.59

7.4.2 Coupled pendula

Consider two pendula coupled via a spring, cf. Figure 7.6. For a detailed description, cf. [57]. The pendula are described in polar coordinates as,

$$\dot{x}_1^{[i]} = x_2^{[i]} \tag{7.18a}$$

$$\dot{x}_2^{[i]} = (-g + u_3^{[i]}) \sin(x_1^{[i]}) + (u_1^{[i]} + u_2^{[i]}) \cos(x_1^{[i]}) \tag{7.18b}$$

for $i = 1, 2$. The inputs to the pendula are external excitation forces acting on the pivot $u_1^{[i]}$ and the inputs $u_2^{[i]}$ and $u_3^{[i]}$ are computed through the spring coupling with the coupled pendula. As in the previous example, we consider only part of the full system, the left pendulum ($i = 1$), and regard the inputs as known. The known inputs are computed from a simulation of the fully coupled system.

The left pendulum was simulated for two seconds with forty constant segments for the three inputs. The tolerances were set to 10^{-6} for both the relative and absolute tolerance. In Table 7.3 the simulation statistics is shown and in Figure 7.7 and Figure 7.8 the step size

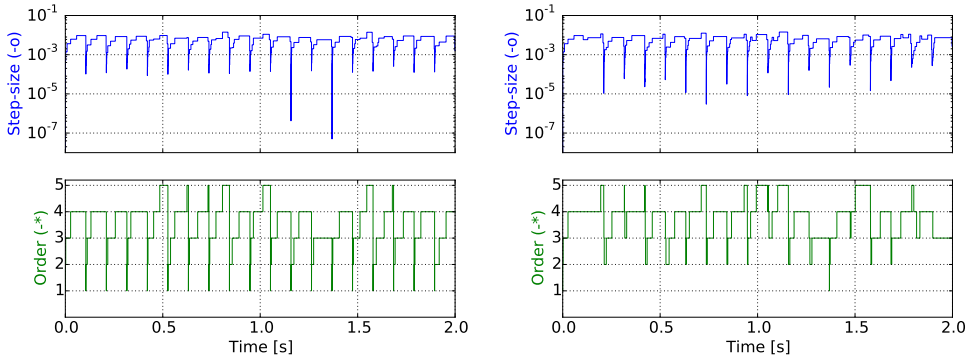

Figure 7.4: *Step size history and order history when simulating Equation 7.17 using linear input segments. The figure to the left show a simulation when CVode is restarted at each global step while the right show a simulation when CVode proceeds without any modifications.*

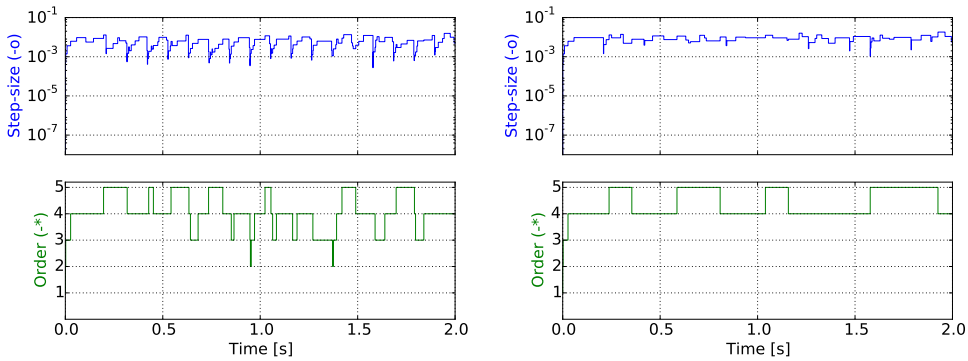

Figure 7.5: *Step size history and order history when simulating Equation 7.17 using linear input segments. The figure to the left show a simulation when the predictor is corrected in the first derivative while the right figure show when the predictor is corrected in the first and second derivative.*

histories and order histories are shown. In Figure 7.9, the predictor polynomial for a time step is shown to illustrate the impact of the correction.

Again, as in the previous example, we draw the conclusions, that a correction in both the first and second derivative is beneficial as it reduced the number of function evaluations and number of steps taken as compared with the restart case. Additionally, the worst choice is to proceed with old values. Considering the figures, we note that CVode, in the corrected case, persistently remain at high order. The global errors are on the same magnitude and are nearly equal in all four cases.

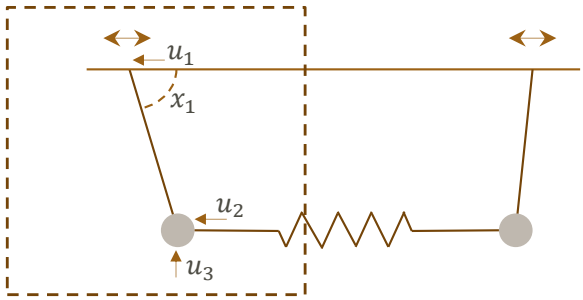

Figure 7.6: *Two coupled pendula with a spring. The dashed square is the model considered in Equation 7.18.*

Table 7.3: *Statistics from simulating Equation 7.18 with different options for crossing the global segments. The elapsed time is normalized with respect to the restart option.*

	Restart	Proceed	Pred. Corr. in $\dot{x}$	Pred. Corr. in $\dot{x}, \ddot{x}$
# steps	1377	1962	1275	850
# fevals	1989	2858	1792	1144
# jacs	55	38	24	39
# errfails	69	210	102	42
time	1.00	1.33	0.88	0.67

7.4.3 Race car

In this example, we consider a race car model previously seen in Section 5.3.2 and in Section 6.4.2. The race car is modeled in Modelica and exported as an FMU using Dymola 2016. Additionally, a separate Modelica model of the wheels used in the race car is exported from JModelica.org. The interest, in this example, is in the right front wheel, cf. Figure 7.10, and the impact of the corrections to the predictor on the simulation. In each wheel, there are 37 inputs. As before, CVode is used, in the wheel, with a relative and absolute tolerance set to 10^{-6} and 200 global segments. The reference trajectories was computed using Assimulo with the solver Radau5 [30] together with a relative and absolute tolerance set to 10^{-10}. The input trajectories for the wheel were computed with a simulation of the full race car.

The wheel is simulated using the different approaches for when crossing global segments. In Figure 7.11, the state trajectories are shown together with the error in the states when using the different approaches. The figure show that the error is on the same magnitude for all approaches, i.e. at the requested accuracy, and thus the error is not impacted by the correction.

In Table 7.4, the simulation statistics are shown. The statistics show a clear decrease in the number of steps taken when using the corrections for the predictor. However, for the second order correction, the cost of computing a new Jacobian has to be weighed against

the reduction in the number of steps. For this case, a second order correction is beneficial.

Table 7.4: *Statistics from simulating the right front wheel in the race car example, Section 7.4.3, with different options for crossing the global segments. The elapsed time is normalized with respect to the restart option.*

	Restart	Proceed	Pred. Corr. in $\dot{x}$	Pred. Corr. in $\dot{x}, \ddot{x}$
# steps	3500	5278	3043	1598
# fevals	6670	8843	5048	2970
# jacs	199	105	59	195
# errfails	323	844	384	70
time	1.00	1.29	0.77	0.47

7.5 Summary

In this chapter, we presented efficient restart of the multistep method CVode in the context of the FMI and co-simulation FMUs. Modifications to the predictor is computed when inputs are set instead of restarting the multistep method. The approach show a significant reduction of work necessary for computing the solution trajectories. The method has been implemented in the open-source tool JModelica.org.

If the structure of a given problem is available, further improvements to the correction can be made, for instance if the problem is linear in the states. In the tool JModelica.org, this information can be made available and should be considered for future improvements. Also, if the dependency information between state derivatives and inputs are given, then only a partial update of the Jacobian might be necessary. Finally, further investigations into whether or not an update of the Jacobian is required at a given global segment is needed.

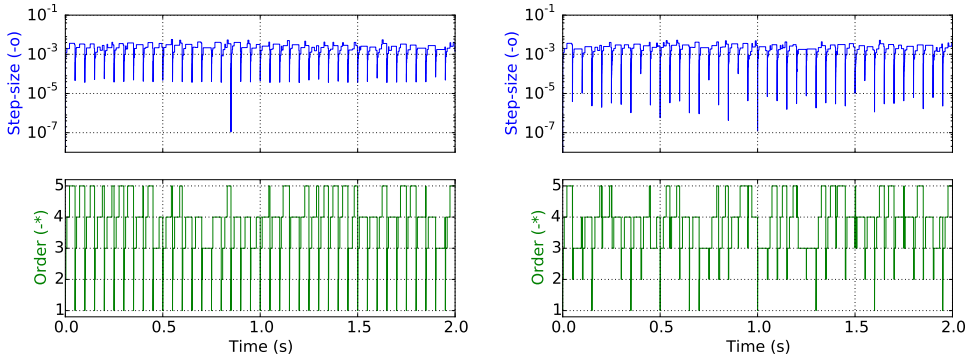

Figure 7.7: *Step size history and order history when simulating Equation 7.18 using constant input segments. The figure to the left show a simulation when CVode is restarted at each global step while the right show a simulation when CVode proceeds without any modifications.*

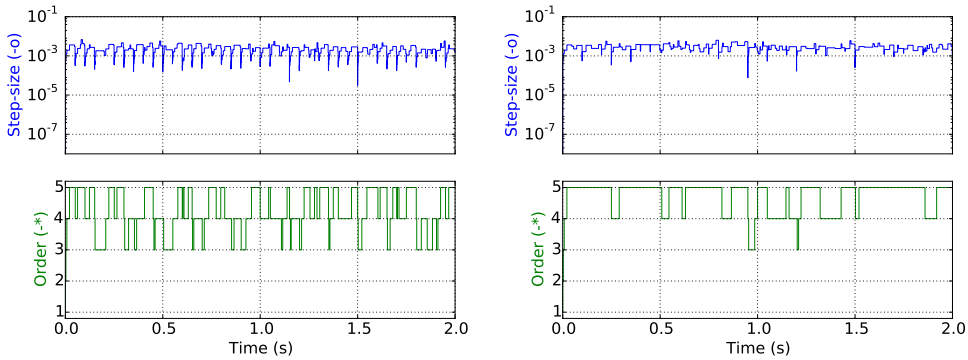

Figure 7.8: *Step size history and order history when simulating Equation 7.18 using constant input segments. The figure to the left show a simulation when the predictor is corrected in the first derivative while the right figure show when the predictor is corrected in the first and second derivative.*

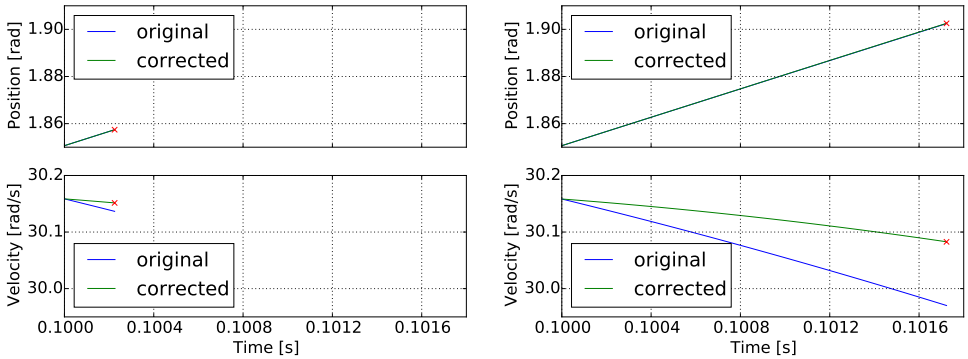

Figure 7.9: *Predictor trajectories at $t = 0.1s$ extrapolated to the next accepted solution when simulating Equation 7.18 using constant input segments. The figures show the original (non-corrected) predictor polynomial together with the corrected predictor polynomial. The figure to the left uses first order correction while the second uses first and second order correction.*

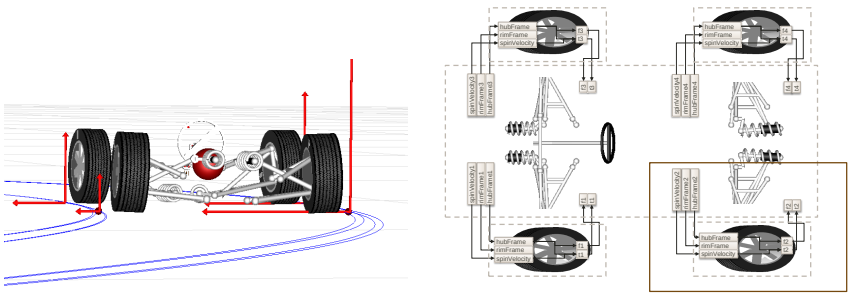

Figure 7.10: *Illustration of the couplings in the race car model from Section 7.4.3. © Modelon.*

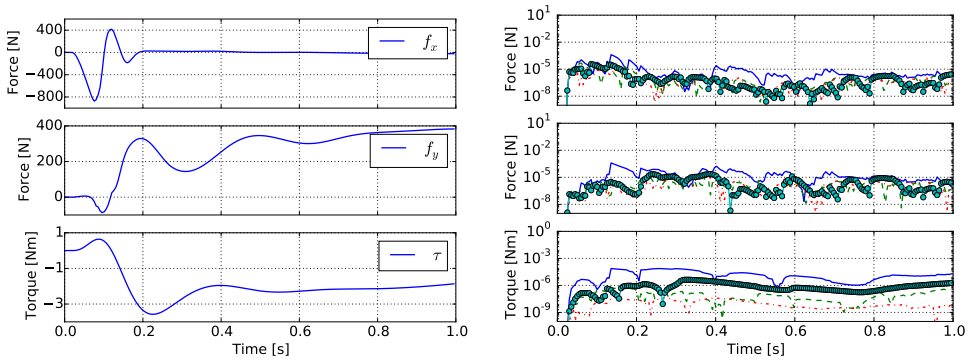

Figure 7.11: *In the left figure, the reference state trajectories when simulating the right front wheel in the race car example, Section 7.4.3. In the right figure, the state errors when using the different approaches for crossing the global segments. The solid line represents a proceeded simulation, the dashed line represents a correction in the first derivative, the dashed dot in the second while the circled represents a restarted simulation.*

Part III:

Software

Overview

In this part we introduce the developed Python packages, PyFMI and Assimulo. In Chapter 8, Assimulo is introduced which is package that unifies different integrators for solving ordinary differential and differential algebraic equations under a common interface. In Chapter 9, PyFMI is introduced which is a package for working with models compliant with the FMI and which connects the models to the solvers in Assimulo. Furthermore, the package contain a master algorithm which implements the algorithms discussed in Part II.

Chapter 8

Assimulo

Assimulo is a simulation package for solving ordinary differential equations containing various different solvers, both state-of-the-art and more experimental ones. The primary aim of Assimulo is to provide a high-level interface for a wide variety of solvers rather than to develop new integration algorithms. Furthermore, the aim is to allow comparison of solvers for a given problem without the need to define the problem in a number of different programming languages to accommodate the different solvers.

Assimulo distinguishes between a problem (or model), which contains the problem equations and the actual solver used for integrating the problem. For instance, the tolerances, which are important quantities to control the solver, are attributes of the solver class and are kept separate from the problem description. The problem definitions are not only limited to the so-called *right-hand-side function* of the problem, but they may also contain event functions in order to support systems with state, step and time events - so-called *hybrid systems*. Additionally, a problem definition can specify options related to the problem such as which states are actually algebraic variables.

Assimulo is designed so that it can easily be incorporated and used as a simulation engine in other modeling or simulation tools. The idea is that the problem class is extended and adapted for the specific tool at hand which makes all the integrators available in Assimulo to the tool, cf. Figure 8.1.

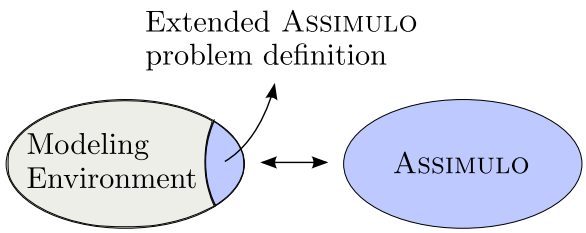

Figure 8.1: *Overview of how Assimulo is extended and integrated into other tools.*

Assimulo is currently used as the simulation engine in JModelica.org and has been successfully applied to a number of applications such as in simulation of dynamical systems with interacting fluid flow [13], simulation of networked control systems [36], simulation of a polyethylene plant [6] and in conjunction with derivative-free optimization [5] to cite a few. Apart from its use in research and in industry, Assimulo is also used as a teaching tool at the Centre for Mathematical Sciences at Lund University [1,2]. Due to its use as a teaching tool, a great effort has been put into the documentation which contains detailed information about all the solvers, a tutorial and an extensive set of examples showing how the various solvers are used and can serve as a basis for further studies.

8.1 Integrators

In the present state, Assimulo provides interfaces to production quality solvers like CVode and IDA from the Sundials suite [33] developed at Lawrence Livermore National Laboratory (LLNL). Sundials is a further development of the codes VODE and DASPK dating back to the 1980s. CVode solves problems defined by explicit ordinary differential equations, $\dot{y} = f(t, y)$. A method flag allows to use BDF methods for stiff problems and Adams–Moulton methods for non-stiff problems. CVode uses a variable order and variable step size implementation. IDA, on the other hand solves the more general problem described as implicit ODEs (differential algebraic equations, DAEs). It uses BDF methods of variable order and variable step size. While primarily intended to solve index-1 problems (in mechanics, problems with constraints on acceleration level), it allows to exclude certain solution components from the step size selection procedure and thus at least technically enables the possibility to solve higher index systems, e.g. mechanical systems with constraints on position or velocity level. As the method tolerances are used to control both step size selection and the corrector iteration process even the tolerances on the algebraic components have to be raised when dealing with higher index problems in order to avoid corrector convergence failures.

One important purpose of the Assimulo project is to give the simulation and modeling engineer access to the wide spread flora of research codes. A typical representative for this class of codes is GLIMDA [72] which is now accessible by Assimulo. GLIMDA is an implementation of general linear multistep methods to solve lower index DAEs. These methods can be viewed as a blend of collocation type implicit Runge–Kutta methods with interpolation-based linear multistep methods. These techniques allow to adapt the method coefficients to the special stability characteristics of the problem at hand. Assimulo's concepts expose this method class to a wide range of mechanical problems and help in this way to gain experience of this relatively new method class when applied to large and industrial models.

[1] http://ctr.maths.lu.se/na/courses/FMNN05/ [accessed: 2016-03-11]
[2] http://ctr.maths.lu.se/na/courses/FMNN25/ [accessed: 2016-03-11]

The implicit Runge–Kutta method RADAU5 [30] shares stability properties with the implicit Euler method but promises higher efficiency due to its larger order. A classical implementation of this method by Hairer is included in Assimulo. The solvers DOPRI5 [29] and RODAS [30] are additionally available for problems on the form $\dot{y} = f(t, y)$. The solvers are various Runge–Kutta and Rosenbrock type methods with variable step size. RADAU5 and RODAS are suitable for stiff problems while DOPRI5 is a non stiff integrator.

The codes wrapped into Assimulo are kept in their original form, only I/O parts and user communication is lifted to the Python level in order to guarantee a homogeneous handling. But Assimulo also supports experimental code directly written in Python. A constant step size Runge–Kutta method and an explicit Euler method, both implemented in Python, are included in Assimulo. They serve as templates for adding own method prototypes. Assimulo also aims to expose even historically interesting codes together with modern industrial codes and more unknown research codes. Among the classical codes we name the solver LSODAR from ODEPACK[31] which is a multistep method that depending on the stiffness of the problem switches between variable order Adams–Moulton and a BDF methods. Also ODASSL [27] is provided as a code specialized on mechanical systems described by the problem class of overdetermined DAEs. It is a variant of DASSL [59] with the linear algebra part of the corrector iteration replaced by a special pseudo-inverse reflecting a transformation to state space form. Other classical MBS simulation codes are planned to be included.

8.2 Problem formulations

Assimulo can solve problems formulated in a number of different ways with the most common being problems formulated by explicit ordinary differential equations,

$$\dot{x} = f(t, x), \quad x(t_0) = x_0 \tag{8.1a}$$

$$\text{with} \quad t \in \mathbb{R}, \quad x \in \mathbb{R}^n, \quad f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n. \tag{8.1b}$$

Another common problem formulation is based on implicit ordinary differential equations including differential-algebraic equations (DAEs),

$$0 = F(t, x, \dot{x}), \quad x(t_0) = x_0, \quad \dot{x}(t_0) = \dot{x}_0 \tag{8.2a}$$

$$\text{with} \quad t \in \mathbb{R}, \quad x \in \mathbb{R}^n, \quad \dot{x} \in \mathbb{R}^n, \quad F : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n. \tag{8.2b}$$

These problem formulations can be extended in order to be able to describe problems with discontinuities. The extension consists of allowing the differential equation to depend on a set of switches, s, which determines the active problem branch. The extended formulations then take the form,

$$\dot{x} = f(t, x, s), \quad x(t_0) = x_0, \quad s(t_0) = s_0 \qquad \text{(explicit problem)}$$

$$0 = F(t, x, \dot{x}, s), \quad x(t_0) = x_0, \quad \dot{x}(t_0) = \dot{x}_0, \quad s(t_0) = s_0 \qquad \text{(implicit problem).}$$

Other supported formulations include singular perturbed problems,

$$\dot{x} = f(t, x, z), \quad x(t_0) = x_0 \tag{8.3a}$$
$$\varepsilon \dot{z} = h(t, x, z), \quad z(t_0) = z_0 \tag{8.3b}$$
$$\text{with} \quad t \in \mathbb{R}, \quad x \in \mathbb{R}^n, \quad z \in \mathbb{R}^m \tag{8.3c}$$
$$f : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n \quad h : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m. \tag{8.3d}$$

Additionally, a further generalization of the problems described by DAEs, Equation 8.2, is supported. The generalization consists of allowing the number of equations or components of F, be greater than the number of states, x, resulting in an overdetermined problem

$$0 = F(t, x, \dot{x}), \quad x(t_0) = y_0, \quad \dot{x}(t_0) = \dot{x}_0 \tag{8.4a}$$
$$\text{with} \quad t \in \mathbb{R}, \quad x \in \mathbb{R}^n, \quad \dot{x} \in \mathbb{R}^n, \quad F : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \quad \text{and} \quad m > n. \tag{8.4b}$$

These type of problems are commonly occurring in mechanical systems which is given its own problem class.

In Figure 8.2 an overview is given over the various problem formulations and the available integrators.

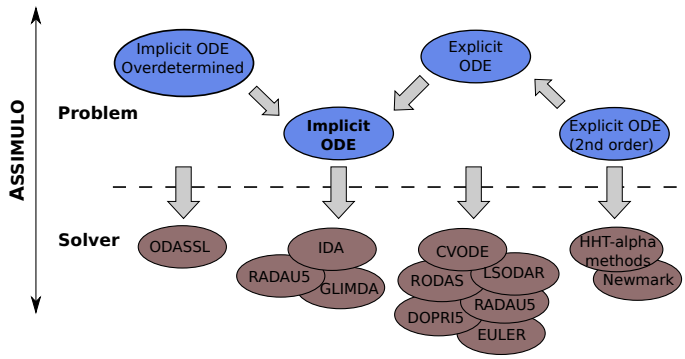

Figure 8.2: *Connection between problem formulations and solvers.*

8.3 Support for discontinuous systems

Assimulo supports general discontinuous systems. Three types of discontinuities are supported: discontinuities dependent on state variables, discontinuities dependent on time and discontinuities dependent on the current integrator state.

The different events are commonly called,

- State Events

- Time Events
- Step Events

State events depend on the solution trajectory and thus their location is not known a priori. This means that the integrator needs to use a monitoring algorithm to detect when an event has triggered. This is commonly done via user provided *event indicator functions* that change their sign when there is an event. The integrator thus needs to check for sign changes of these functions, and if detected, to locate the time for the sign change. The event indicator function has to be provided as part of the problem description. A typical scenario of these kind of events are bounces in mechanical multibody systems, where the time of the impact depends on overall dynamics.

The location of *time events* on the other hand is known a priori, meaning that for each simulation segment it is known when the time event occurs. Consequently, this time can be set as the simulation end time for that segment. A typical scenario is a change of a force at a given time point.

Step events are similar to state events but the localization of the event can be done with less precision. It is sufficient to react on the event after a completed integrator step instead of locating a time point within a step. Often, step events are used when a model has to be re-parametrized during integration due to that the current parametrization approaches a singularity. A typical scenario is coordinate partitioning [73]. Step events differ from the state and time events in that they do not change the solution trajectory.

The following example combines state and step events to clarify their conceptual differences: Consider a bell modeled as a pendulum with a stop. It can be described by the equations,

$$\dot{\theta} = \omega \tag{8.5}$$

$$\dot{\omega} = -\frac{\mathrm{g}_0}{l}\sin(\theta) \tag{8.6}$$

where θ and ω is the angle and angular velocity respectively. The parameter g_0 is the gravity while l is the length of the pendulum. The stop is a wall situated at an angle $-3\pi/4$ refraining the pendulum from swinging freely, cf. Figure 8.3. The impact can be modeled

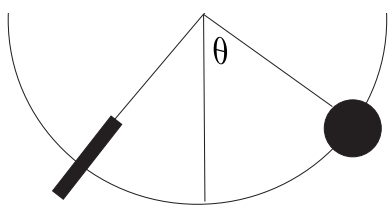

Figure 8.3: *A blocked pendulum.*

by a state event dependent on the angle, θ,

$$g(\theta) = \theta + \frac{3\pi}{4} \tag{8.7}$$

where g is the event indicator function. At the event, t_e, the sign of the current angular velocity is inverted, i.e:

$$\omega(t_e^+) = -\omega(t_e^-). \tag{8.8}$$

As it turns out, this model of the discontinuity causes the integrator to get stuck at the wall. This is due to that the event is detected when $g(\theta) < -\epsilon$ and once the velocity is inverted an event will be detected again due to that now $g(\theta)$ will be increasing. At this point the angular velocity is again inverted and we are stuck in an infinite loop. One idea for rectifying this situation is by introducing another event indicator that deactivates and reactivates the primary event indicator at appropriate times. However, this introduces additional unnecessary cost in the integration. A better alternative is to introduce a step event in the model. The reactivation time can be at the end of the first step where the velocity changed its sign again. The exact time for the sign change is not required. In Figure 8.4, simulation result is shown for the blocked pendulum when using the solver CVode.

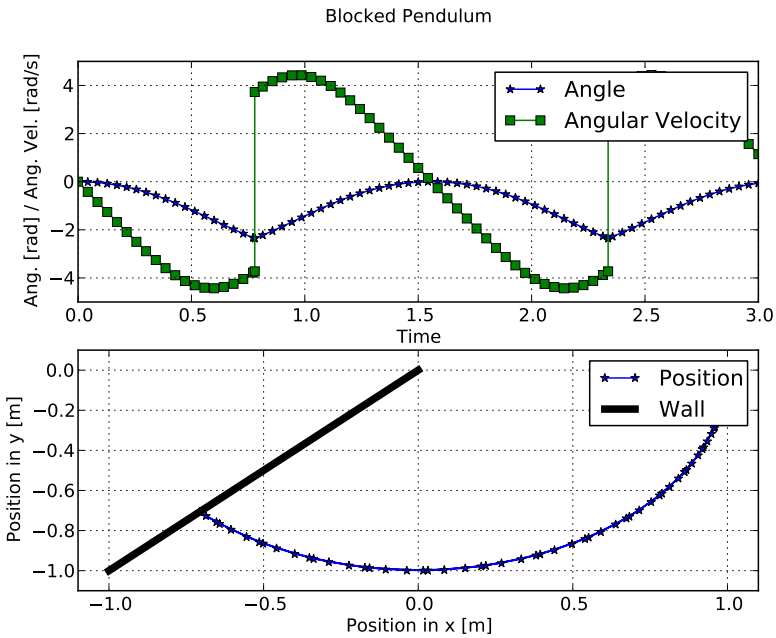

Figure 8.4: *Simulation result of the blocked pendulum.*

As default, there are only the Sundials solvers CVode and IDA together with LSODAR that supports state events. This is unfortunate as many industrial simulation problems

exhibit state discontinuities. It is desirable to be able to use the other solvers as well for these problems. This led to a general implementation of an algorithm for locating state events in Assimulo [24] that has been combined with most of the other available solvers.

8.4 Simulation strategy

In Figure 8.5, steps for performing an integration are shown. Here, filled boxes represent methods in the problem description giving the user a significant amount of flexibility for controlling different aspects of the simulation process. This is specifically important for cases where Assimulo is integrated into other tools which may for instance use specialized algorithms for initialization or storing the result over the network. The flowchart largely represents what is common simulation practice. The addition is the step events which have been influenced by the work done with the FMI [15].

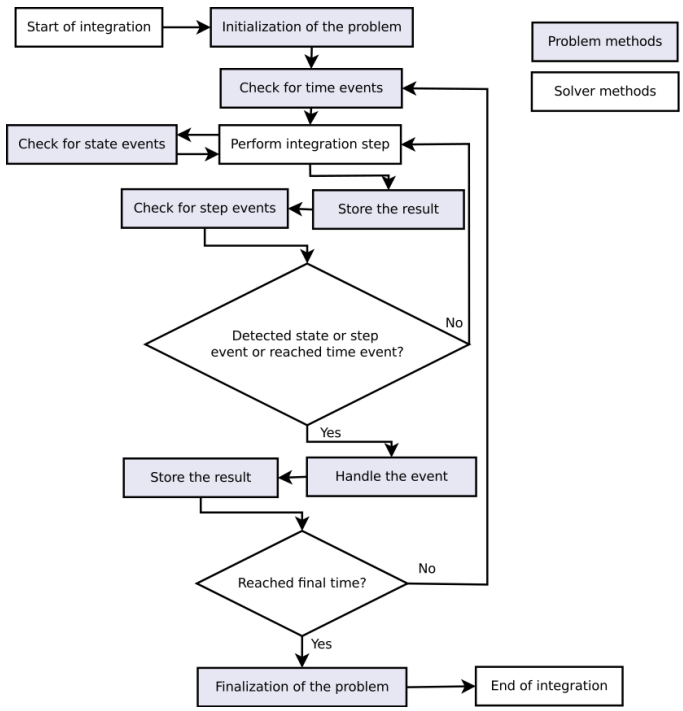

Figure 8.5: *Flowchart over how the integration is performed. The filled boxes represent methods that can be defined in the problem class. The other boxes are internal.*

8.5 Implementation overview

The core of Assimulo is implemented in Cython which is a static compiler for Python. It allows to mix the programming languages C and Python interchangeably. The added benefit of mixing the languages is that the main part of the package, where readability and scripting functionality matter, is based on Python and performance critical parts are kept in C. In this way computational performance is preserved as opposed to relying solely on Python.

In Assimulo, each type of problem formulation, as explained in Section 8.2, has its own Cython class. In addition to the Cython class, there is an inherited Python class. In Figure 8.6, the structure of the problem classes is shown. The classes correspond to,

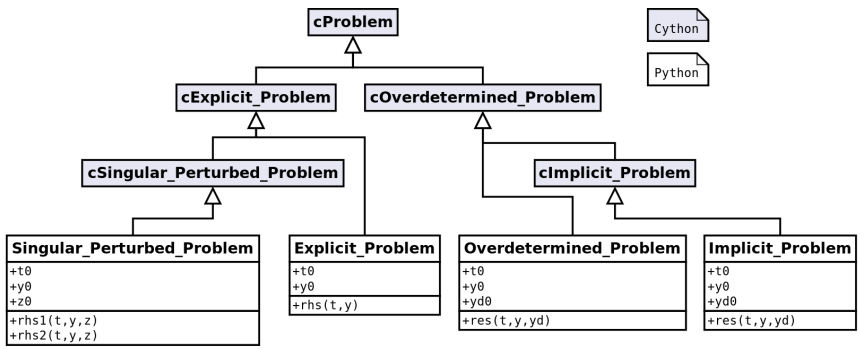

Figure 8.6: *Structure of the problem classes.*

- `Explicit_Problem` , Equation 8.1.
- `Implicit_Problem` , Equation 8.2.
- `Singular_Perturbed_Problem` , Equation 8.3.
- `Overdetermined_Problem` , Equation 8.4.

The reason for having a corresponding Cython class is the obvious benefit of allowing parts of the implementation to be optimized and allowing a problem to be fully implemented in Cython. In this regard it may seem redundant to have Python classes on top. However, this is due to Cython restrictions which might cause inconveniences to a non specialized user.

A problem, in its basic form, accepts a method containing the differential equations and the initial conditions in its constructor. In the case of an explicit problem, this corresponds to providing the right-hand-side, $f(t, y)$, and t_0 together with y_0. However, in order to facilitate a broader use of a problem, additional methods can be provided that do not directly belong to the strictly mathematical problem of solving a set of differential equations. One of the additional methods is a method for handling the solution. This method

is called at each requested solution point to enable the user to control how the solution is handled. An example where it is vital to be able to modify *data handling* is the case when simulating large models over a long time horizon which results in a large amount of result data.

Other methods are provided for initialization and finalization. They are called prior and respectively after the simulation process. Finally there is a reset method, which allows for a custom reset of a problem, cf. Figure 8.7.

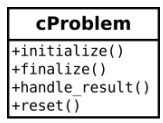

Figure 8.7: *Additional methods available in a problem class that are not strictly related to the mathematical problem.*

The core of Assimulo is the general solver class ODE together with the explicit and implicit ODE classes shown in Figure 8.8. Theses classes contain the logic for performing

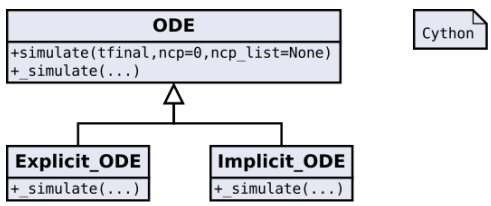

Figure 8.8: *Structure of the simulation core and the general solver classes.*

a simulation including the logic for following the calling sequence explained by Figure 8.5. Apart from the simulation logic, they contain the common implementation for the logging and the options.

Assimulo connects integrators that are implemented in different programming languages. For this task some additional tools are needed. For integrators that are implemented in C, for example CVode, Cython is used as it is already used in the core and as it provides a fast interface to external C code. For other integrators, for example those implemented in Fortran, the connection between Fortran code and Python code is done by F2PY [58] which allows to directly invoke a Fortran subroutine from Python.

In Figure 8.9 and Figure 8.10, the class structure for the available solvers and its connection to one of the general solver classes is shown. The figures depict also the connection to the external codes containing the solvers mentioned in Section 8.1. What is not shown in the figures however, is which problem type each solver accepts. For solvers connected to the general explicit ODE class, all solvers except DASP3 accept a problem on the form described by Equation 8.1 (explicit problem) while DASP3 accepts singular perturbed problems described by Equation 8.3. For the solvers connected to the general implicit ODE class,

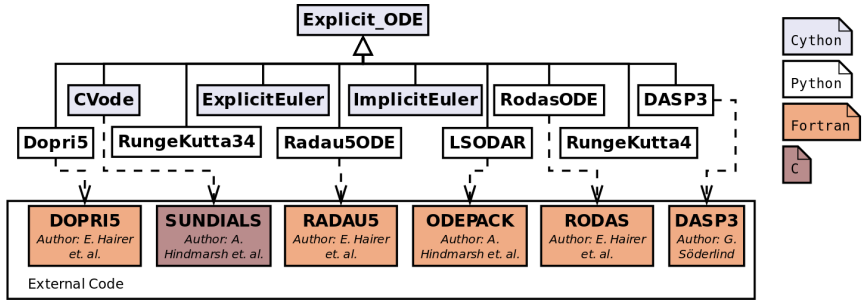

Figure 8.9: *Structure of the solvers for explicit ODEs together with the connection to external codes.*

ODASSL accepts overdetermined problems (Equation 8.4) while the other accept implicit problems (Equation 8.2). As mentioned before, the only solvers that by default support state events are IDA, CVode and LSODAR but with the addition of the algorithm described in [24], the number of solvers that do support state events has been increased. The only solvers that do not currently support state events are, ODASSL, GLIMDA, DASP3 and RungeKutta4.

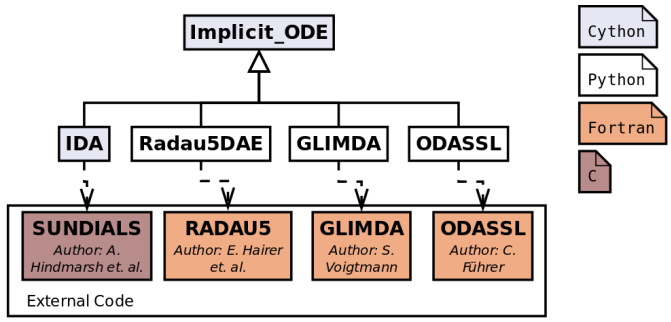

Figure 8.10: *Structure of the solvers for implicit problems together with the connection to external codes.*

8.6 Case studies

In this section, the use of Assimulo for solving various problems is demonstrated. First, the classical van der Pol oscillator is used as an example and a detailed explanation of the steps involved for solving the problem using Assimulo is given. Next, a problem with state discontinuities is discussed with emphasis on how the necessary methods are provided to the problem class. Finally, a model of an industrial robot is simulated together with a discussion of the performance of Assimulo along with comparisons with other simulation environments.

The intention of these demonstrations is to highlight the use of these solvers within Assimulo and the possibilities of the package rather than to evaluate the solvers on the

particular problems.

8.6.1 Van der Pol oscillator

The van der Pol oscillator is given by the two differential equations,

$$\dot{x}_1 = x_2 \tag{8.9a}$$

$$\dot{x}_2 = \mu[(1 - x_1^2)x_2 - x_1], \quad \mu = 2 \cdot 10^3 \tag{8.9b}$$

together with the initial conditions, $x_1 = 2.0$ and $x_2 = -0.6$.

Simulating the oscillator using Assimulo requires that first the differential equations, the dynamics of the problem, are defined in a Python method:

```
def van_der_pol(t, x):
    x1, x2 = x
    xd1 = x2
    xd2 = 2e3 * ((1. - x1**2) * x2 - x1)
    return [xd1, xd2]
```

This method together with the initial conditions is used to create an instance of an explicit problem. The initial time defaults to $t_0 = 0$.

```
x0 = [2.0, -0.6] #Initial conditions
oscillator_model = Explicit_Problem(van_der_pol, x0)
```

The problem, `oscillator_model`, can now be used to create any instance of a solver, Figure 8.9, for example Dopri5, by providing it to the solver constructor.

```
oscillator_dopri5 = Dopri5(oscillator_model) #Creates the solver
```

Finally, a simulation is performed by invoking the simulate method of the solver which returns the simulation results. The arguments are the final time, 2, and the number of result points, 1000.

```
td, xd = oscillator_dopri5.simulate(2, 1000)
```

The results can easily be used for visualization, cf. Figure 8.11. The figure displays the simulation results for the oscillator obtained by DOPRI5 along with results from CVode. Using CVode instead of Dopri5 only requires that the above code, where the solver is created, `Dopri5(...)` is changed to `CVode(...)`. These two simulations are additionally compared with a reference solution calculated by RADAU5 using extreme tolerance requirements, atol = rtol = 10^{-12}.

8.6.2 The woodpecker

This example is intended to show how Assimulo can handle hybrid systems illustrated by a toy woodpecker, [43]. The model consists of a vertical bar attached to the ground, a sleeve

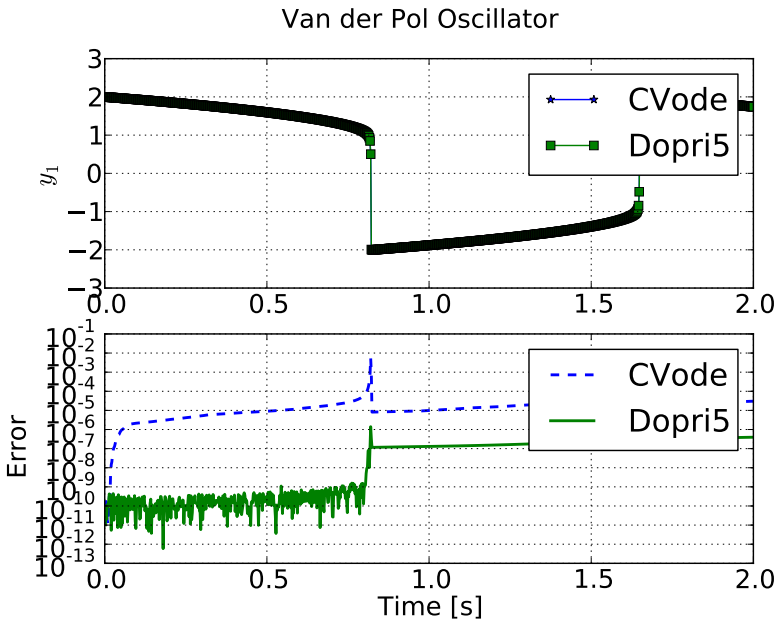

Figure 8.11: *Simulation result of the van der Pol oscillator together with a comparison of the error between a reference solution and solutions obtained by both CVode and Dopri5.*

able to slide along the bar and the woodpecker which is attached to the sleeve via a spring, cf. Figure 8.12. Impact is modeled without friction for simplicity.

The model can be in one of three states. In State I, there is no blocking and the sleeve is free to move along the bar. In State II, the sleeve is blocked by a contact with the bar at the sleeve's upper left and lower right corner. Finally in State III, the sleeve is blocked by the opposite corner pair. The equations of motion are given for the three states, I,II,III, by three different functions on the same form,

$$0 = F_i(t, y, \dot{y}) \ \texttt{with} \ i \in \{\texttt{I},\texttt{II},\texttt{III}\} \tag{8.10}$$

A change of state is determined by switching conditions. Changing from State I, where the sleeve is free to slide, occurs when,

$$h_S \varphi_S > r_S - r_0 \quad \text{and} \quad \dot{\varphi}_B > 0 \tag{8.11}$$

or when

$$h_S \varphi_S < -(r_S - r_0) \quad \text{and} \quad \dot{\varphi}_B < 0. \tag{8.12}$$

In the first case, the state is changed from I to III and in the second case the state is changed from I II. Changing back to State I occurs when,

$$\lambda_1 = 0. \tag{8.13}$$

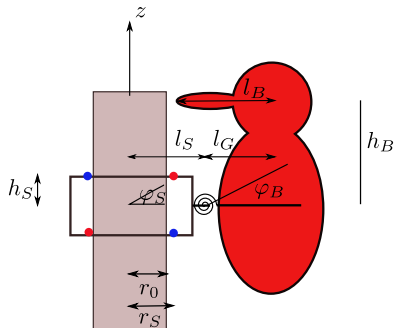

Figure 8.12: *Schematic figure of the woodpecker.*

where λ_1 is the lagrange multiplier taking care of the constraints. Additionally, another state change occurs when the woodpecker hits the bar. However, after this impact, the state is directly changed back to the previous state and only state values are reset. The woodpecker hits the bar when,

$$h_B\varphi_B > l_S + l_G - l_B - r_0. \tag{8.14}$$

This leads to four event indicators that are monitored during the integration. The event indicators are given by,

$$g_1 = h_S\varphi_S + (r_S - r_0) \tag{8.15a}$$
$$g_2 = h_S\varphi_S - (r_S - r_0) \tag{8.15b}$$
$$g_3 = \lambda_1 \tag{8.15c}$$
$$g_4 = h_B\varphi_B - l_S - l_G + l_B + r_0. \tag{8.15d}$$

When the model changes to either of the blocking states, the momentum is preserved, i.e $I^- = I^+$. The momentum prior to the impact, I^-, is given by,

$$I^- = m_B l_G \dot{z}^- + (m_B l_S l_G)\dot{\varphi}_S^- + (J_B + m_B l_G)\dot{\varphi}_B^- \tag{8.16}$$

where the superscript $\dot{z}^-$ indicates the value prior to the impact. Post impact, the sleeve is in a blocking state and thus $\dot{z}^+ = 0$ and $\dot{\varphi}_S^+ = 0$, allowing computation of $\dot{\varphi}_B^+$ due to,

$$I^+ = m_B l_G \dot{z}^+ + (m_B l_S l_G)\dot{\varphi}_S^+ + (J_B + m_B l_G)\dot{\varphi}_B^+. \tag{8.17}$$

For the case when the woodpecker hits the bar, the angular velocity of the woodpecker, $\dot{\varphi}_B$, is reset by changing its sign.

In Appendix A.1, the details of the differential equations are given and the Python code for simulating the woodpecker is described.

The results of simulating the woodpecker 0.16s and 1.0s with IDA are depicted in Figure 8.13. The simulations were performed with the absolut and relative tolerance being 10^{-6} (default values). For the first simulation, the woodpecker hits the bar twice and for the second simulation 11 times.

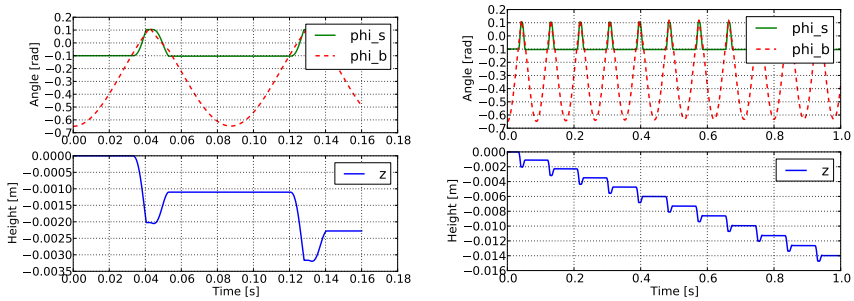

Figure 8.13: *Simulation results for when simulating the woodpecker from Section 8.6.2. The model was simulated using the solver IDA for* 0.16s *(left) and* 1.0s *(right). The absolute and relative tolerances were set to* 10^{-6}.

8.6.3 Performance considerations

In order to demonstrate the capabilities of Assimulo and discuss the performance of the package as compared to using similar solvers from commercial tools, a robot model was chosen as a test example. The model is described in [56] and depicted in Figure 8.14. In

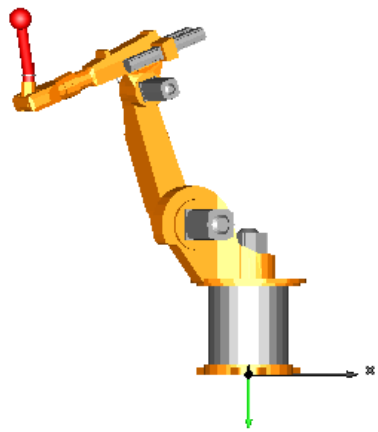

Figure 8.14: *Robot model. © Modelica Association.*

order to perform a fair comparison, we use the robot modeled in the Modelica language [22] from the Modelica Standard Library [47] and export the model using the FMI from the modeling and simulation tool Dymola 2014 [19]. Using PyFMI (cf. Chapter 9) this model can be simulated in Assimulo. In this way we have a model that we can simulate under same conditions, i.e. we have decoupled the modeling part including event handling from the actual integrator part as exactly the same modeling description is used in Dymola and Assimulo due to the FMI concept. Moreover, the model can be imported into Mat-

Table 8.1: *Simulation statistics for the robot model using adaptive solvers from Assimulo.*

Solver	Steps	F-Evals	J-Evals	Elapsed Time
CVode	1904	4787	65	0.433s
Dopri5	4480	27602	-	3.512s
LSODAR	2301	10526	179	0.550s
Radau5	397	7417	127	0.907s
Rodas	480	21325	480	1.709s

Table 8.2: *Simulation statistics for the robot model using two similar BDF methods.*

Solver	Steps	F-Evals	J-Evals	Elapsed Time
CVode (Assimulo)	1904	4787	65	0.433s
ode15s (MATLAB)	1660	4404	56	2.67s

LAB using the FMI Toolbox for MATLAB [52] and simulated using the available solvers in MATLAB.

The robot has 36 states and 98 event indicators and is simulated from $t_0 = 0$ to $t_f = 2$ seconds using a relative tolerance of 10^{-4} and a absolute tolerance of 10^{-6}. Additionally, generation of a result file has been disabled in all cases in order to not influence the simulation time.

Before a performance comparison is made, the robot is simulated using the adaptive solvers from Assimulo in order to highlight that it is indeed possible to use the various solvers from the package to solve complex problems, including problems with events. In Table 8.1 the simulation statistics are shown.

In a first performance test, the integrator ODE15S in MATLAB, is compared with a similar solver, CVode within Assimulo. In Table 8.2 the simulation statistics are shown. From the statistics it is clear that CVode is a competitive solver for this particular problem and even superior in terms of computation time. In Figure 8.15 the simulation error over time is depicted which shows that the achieved accuracy is approximately the same. This motivates the comparison of the simulation statistics. However, it should be mentioned again that these two solvers are similar, both are BDF-type methods of variable order intended for the same class of differential equations.

In the second performance test LSODAR is used as it is available both in Dymola and Assimulo. The same settings as above are used. In Table 8.3 and in Figure 8.16 simulation statistics and the error are shown. It is worth mentioning that even though Assimulo is written in Python the solver LSODAR is written in Fortran and the model is in C so the

Table 8.3: *Simulation statistics for the robot model when using LSODAR in Assimulo and in Dymola.*

Solver	Steps	F-Evals	J-Evals	Elapsed Time
LSODAR (Dymola)	1940	8871	146	0.469s
LSODAR (Assimulo)	2002	9000	151	0.490

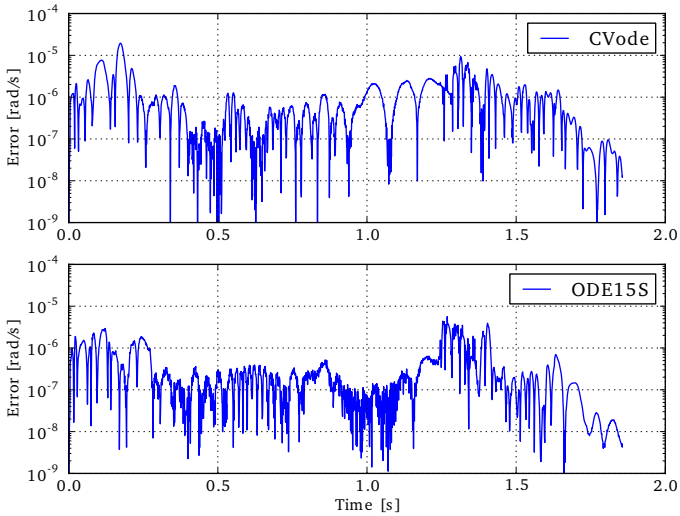

Figure 8.15: *Simulation error when simulating the robot model using both CVode and ODE15S.*

effect of the overhead of using Python will decrease with increasing size of the models.

The conclusion is that even though Assimulo is used from Python, the performance is on par with other simulation environments and even in some cases superior.

8.7 Summary

In this chapter, we presented the background and aim of the simulation framework Assimulo. Different problem classes and related solvers were described and examples served to highlight various aspects of the package - in particular the use of Assimulo coupled to modeling tools by the FMI standard was illustrated. Assimulo[3] is freely available under the LGPL [25] license and the future plan is to include an increasing variety of original codes and make them available through the framework presented.

[3]http://www.assimulo.org [accessed: 2016-03-11]

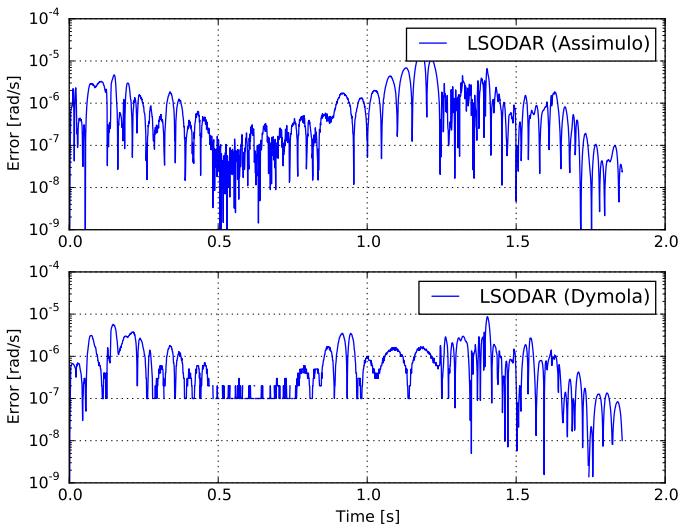

Figure 8.16: *Simulation error when simulating the robot model using LSODAR both in Assimulo and in Dymola.*

Chapter 9

PyFMI

PyFMI is a Python package for working with models compliant with the FMI standard and based on the open source package FMI Library [51]. It is designed to provide a high-level, easy to use, interface for working with FMUs. It connects the full set of methods in the FMI specification in an object-oriented approach. The package is not only a mapping of the FMI interface to Python, it provides much of the functionality needed to perform various experiments for both evaluating the complex dynamical system model by itself but also for evaluating the physics that the model represents. The evaluation of the model could be to verify the model dynamics by efficient simulation while evaluations of the physics could be to performing parameter estimations. These experimentations requires an extensive tool beyond the low-level FMI interface which motivates the package. Furthermore, with the FMI, simulation of coupled models in a co-simulation setting is possible. In this setting, the dynamics of each system is hidden and exchanging information between systems is done through inputs and outputs. This is important as in many cases, with complex systems, this is the only viable option due to that parts of the model is modeled in different tools. An algorithm for performing a co-simulation is called a *Master Algorithm* and within PyFMI a master algorithm has been implemented and made available.

PyFMI has been used successfully in a number of different applications such as in [35] and [3] as well as in [74]. It is additionally an integral part of the open source JModelica.org platform.

9.1 Overview and analyses

The FMI standard describes a light-weight interface for interacting with a model which by itself does not include any analyses of the dynamic model. In this section, the available capabilities of PyFMI is described and shown how they can be used. The major features of the package is linearization of an FMU, Section 9.1.1, simulation of an FMU, Section 9.1.2, simulation of coupled FMUs, Section 9.1.3, and estimation of parameters within an FMU, Section 9.1.4.

These analyses are necessary in order to support model-based design workflows. Linearization of a model is useful when, for instance, designing control systems using classical approaches and when analyzing stability of the model. Simulation and simulation of coupled systems are vital to understanding the dynamics and how the dynamics behave over time. With efficient simulation and access to solvers that are appropriate for a given problem, the return time is reduced resulting in more time for experimentations. Furthermore, a common situation in a model of a system is that not all parameters are given, only an approximation is known due to that the parameter can be hard to measure on a physical system that the model represents. In these cases, parameter estimation is key to make the model more representative of the physical system.

For illustration purposes, we consider a model of the van der Pol oscillator given by,

$$\dot{x}_1 = \mu[(1 - x_2^2)x_1 - x_2] + u, \quad x_1(t_0) = -0.6 \tag{9.1a}$$

$$\dot{x}_2 = x_1, \quad x_2(t_0) = 2 \tag{9.1b}$$

where $\mu = 20$ and u is the input signal. The Modelica code of the example is shown in Appendix A.2 and the model is compiled into an FMU named `VDP.fmu`.

As a first step for using PyFMI, the FMU needs to loading into Python. In Example 9.1.1 this is explained. In Example 9.1.2, it is shown how to interact with the FMU.

Example 9.1.1 (Loading an FMU). *The first step for working with FMUs is to load the model into Python, i.e. couple the binary from the FMU and read the model description containing information about the variables etc.*

```
#Convenience function for loading a general FMU
from pyfmi import load_fmu

#Loads the FMU and return a model object
model = load_fmu("VDP.fmu")
```

The FMU is automatically extracted and the metadata is read together with coupling of memory handling. If the model is discarded, memory is automatically handled and deallocated if necessary. No manual handling of memory is necessary.

Example 9.1.2 (Interacting with a model). *Once the model is loaded into Python, values can be retrieved from the model using the high-level get/set methods.*

```
#Get the value of the variable 'mu'
mu = model.get("mu") #.set for setting values
print mu
>>>20
```

For variable attributes, these can be obtained similarly,

```
#Get the start value of variable 'x1'
start_x1 = model.get_variable_start("x1")
print start_x1
>>>-0.6
```

All attributes such as min, max, nominal and start can just as easily be retrieved.

9.1.1 Linearization

For analyzing the dynamics of the system, linearizing the model (Equation 2.1) is usually the first step. The linearized state space form for a model exchange FMU is,

$$\dot{x} = Ax + Bu \tag{9.2a}$$
$$y = Cx + Du. \tag{9.2b}$$

In FMI there is no direct way of computing the matrices in the linearized state space form (Equation 9.2). There are however, methods for computing the directional derivatives, in FMI 2.0, with respect to a set of variables (here either x, u or a subset of them) and a set of functions (f or g or a subset) together with a seed vector. The definition of the directional derivatives are,

$$g_z = \frac{\mathrm{d}g(z)}{\mathrm{d}z} v \tag{9.3}$$

where $g(z)$ is a vector-valued function, z is the vector of variables and v is the seed vector. From the directional derivatives, the partial derivatives, the matrices A, B, C, D in Equation 9.2, can directly be computed by a sequence of calls with v replaced by unit vectors.

If structural information is available, e.g. if the structural dependency between x_i, u_i and $\dot{x}_j$ is known and between x_i,u_i and y_j, compression can be employed such that the number of evaluations of either the directional derivatives or evaluations of f (in case a finite difference approximations is used) is reduced.

Consider the ODE in Equation 9.4,

$$\dot{x}_1 = x_1 \tag{9.4a}$$
$$\dot{x}_2 = x_2 + x_3 \tag{9.4b}$$
$$\dot{x}_3 = x_1 + x_3. \tag{9.4c}$$

Now consider that the Jacobian, $\frac{\mathrm{d}\dot{x}}{\mathrm{d}x}$, is computed using a first order finite difference scheme requiring $3 + 1$ evaluations of the derivatives. However, due to the structure of the ODE, the partial derivatives $\frac{\partial \dot{x}}{\partial x_1}$ and $\frac{\partial \dot{x}}{\partial x_2}$ can be computed simultaneously as $\dot{x}_1$ and $\dot{x}_3$ are independent of x_2 and $\dot{x}_2$ are independent of x_1. This leads to that the construction of the Jacobian only need $2 + 1$ evaluations of the derivatives. In general, an adjacency matrix, determining the structural relation between the variables (states or inputs) and the functions (derivatives or outputs), can be constructed, Equation 9.5 for the given ODE.

$$A_{\text{adj}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}. \tag{9.5}$$

Given the adjacency matrix, a compression can be computed aimed at reducing the number of evaluations of the derivatives or outputs. In PyFMI, the algorithm proposed in [18] is used.

With the growing size of models and due to that in general the state space matrices are sparse, utilizing this information is essential in order for efficient handling of the system. Using SciPy, the ability to represent these matrices is available and supported by PyFMI.

For co-simulation FMUs, the derivatives are not exposed, cf. Equation 2.6. However, the above applies to the outputs which are available.

9.1.2 Simulation of single models

A key feature of the package is the connection to Assimulo, which provides capabilities for performing simulations of model exchange FMUs using ODE solvers interfaced with Assimulo. The coupling is made possible by extending the definition of the problem classes

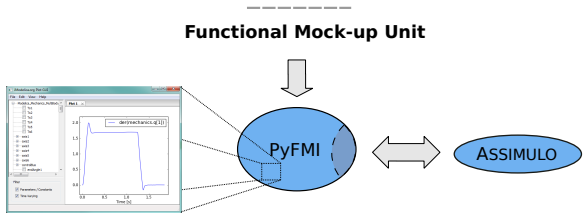

Figure 9.1: *Coupling between PyFMI and Assimulo.*

accepted by Assimulo, Figure 9.1. With the extension, customizations related to FMI is made possible, such as exposing the different events in FMI to the solver.

Assimulo separates between a problem, which contains the problem equations, and the actual solver used for the integration. The problem object is not only limited to the derivatives equation, but it may also contain event functions which is necessary in the FMI case. Furthermore, the problem object can be used to define specific event handling and user defined result handling. All of these features are necessary in order to couple a model exchange FMU to a simulation environment. In Example 9.1.3, a simulation of an FMU is shown.

Example 9.1.3 (Simulating an FMU). *A simulation of an FMU, either a model exchange or a co-simulation FMU, follows the same steps. First, the model is loaded.*

```
#Convenience function for loading a general FMU
from pyfmi import load_fmu

#Loads the FMU and return a model object
model = load_fmu("VDP.fmu")
```

Then, a simulation is performed by invoking the simulate method on the model object.

```
#Simulate the model
res = model.simulate(final_time=2)
```

The simulation results are returned in the `res` *object. In Figure 9.2, the simulation result for*

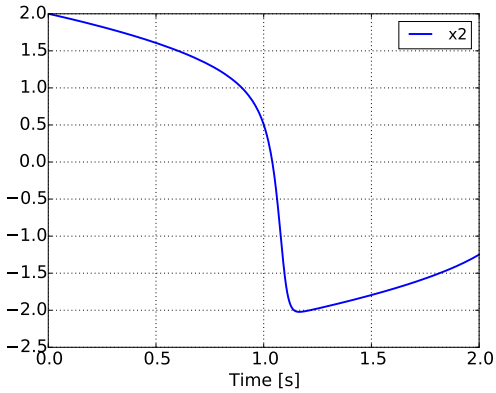

Figure 9.2: *Simulation result of the van der Pol oscillator from Example 9.1.3.*

x_2 *is shown.*

A dynamical model typically has control signals or external forces acting on the model during a simulation. An example is the road profile for a vehicle. In general, this data is a list of points connected to a point in time,

$$(t_i, u_i), \quad i = 0, \ldots, N. \tag{9.6}$$

Within PyFMI, this data can be provided to the simulation setup and will be evaluated during the simulation using linear interpolation between the data points,

$$u(t) = u_i + (t - t_i)\frac{u_{i+1} - u_i}{t_{i+1} - t_i}, \quad t \geq t_i \wedge t < t_{i+1}, \quad i \in [0, N]. \tag{9.7}$$

Another option is that the expression for the control signals are known and for these cases providing a function, instead of data points,

$$u(t) = h(t) \tag{9.8}$$

is beneficial. The reason is that in the first case, discontinuities in higher derivatives are introduced which may degrade the performance of the simulation. In Example 9.1.4 an example using an input function is shown.

Example 9.1.4 (Inputs). *In this example, an input function,* $f(t) = 100\sin(30t)$*, is defined which provides the input to the variable* u *in the van der Pol oscillator model, Equation 9.1.*

```
import numpy as np #Import numpy

#Define the function, need to be dependent on time
def f(time):
    return 100*np.sin(30*time)

#Specify the input variable together with the function
input = ('u', f)

#Provide the input object to the simulate method
res = model.simulate(final_time=2, input=input)
```

During the integration, the input will be invoked during each evaluation of the model equations, for model exchange. For co-simulation FMUs, the input function will be evaluated at every step. In Figure 9.3, the simulation result for x_2 and the input, u, is shown.

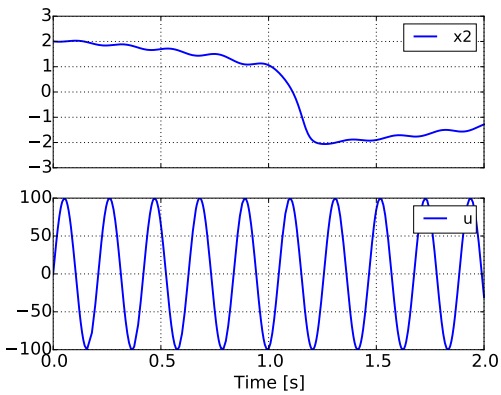

Figure 9.3: *Simulation result of the van der Pol oscillator from Example 9.1.4 together with the input function, u.*

Furthermore, general options for controlling the simulation are available. In Example 9.1.5, changing options are shown and in Table 9.1, the options for controlling a simulation of a model exchange FMU is shown.

A simulation of a co-simulation FMU follow the syntax as shown above. However, as the simulation do not require an external solver, the options available are limited, cf. Table 9.2.

Example 9.1.5 (Providing options to the simulation). *Setting options for controlling the simulation can be done in two steps. First the available options are retrieved from the model object.*

```
#Retrieves the options dictionary
opts = model.simulate_options()
```

The `opts` *object is a Python dictionary which contains the available options. Setting an option is done using the normal Python dictionary syntax.*

```
opts["ncp"] = 500 #Change the number of output points
```

For the simulation to use the options, these need to be provided when invoking the simulation.

```
#Provide the options object to the simulate method
res = model.simulate(options=opts)
```

Table 9.1: *Description of the options available in the default algorithm in order to control the simulation of an model exchange FMU.*

Option	Description
solver	*The ODE solver to use. By default, all solvers from Assimulo can be used.*
ncp	*The number of communication points, i.e. the number of requested (equally spaced) points to store the result.*
initialize	*If the model should be initialized or not.*
result_handling	*Determines how the result should be stored, either on file or directly in memory.*
result_handler	*Ability to specify a custom result handler.*
filter	*A filter for choosing which variables to actually store result for.*
extra_equations	*Determines if additional equations should be solved together with the model.*
{solver}_options	*Specifies additional solver specific options.*

Table 9.2: *Description of the options available in the default algorithm for simulation of a co-simulation FMU.*

Option	Description
ncp	*The number of communication points, i.e. the number of requested (equally spaced) points to store the result.*
initialize	*If the model should be initialized or not.*
result_handling	*Determines how the result should be stored, either on file or directly in memory.*
result_handler	*Ability to specify a custom result handler.*
filter	*A filter for choosing which variables to actually store result for.*

9.1.3 Simulation of coupled models

A second key feature of the package is the ability to simulate coupled systems, i.e. coupled FMUs. PyFMI implements a master algorithm which includes the approaches used for co-simulation discussed and analyzed in Chapter 5 and in Chapter 6. The implementation supports simulation of a coupled system via a parallel approach, i.e. Jacobi-like, defined as (for M models),

$$y_{n+1}^{[i]} = \Phi^{[i]}(H, u_n^{[i]}; p), \quad i = 1, \dots M \tag{9.9}$$

$$u_{n+1} = c(y_{n+1}). \tag{9.10}$$

The algorithm proceeds by first providing inputs to a model and then performing a global time step, for the ith model: $y_{n+1}^{[i]} = \Phi^{[i]}(H, u_n^{[i]}; p)$. This can be done for all models

simultaneously and once all models have performed the step, information is exchanged between the models and inputs for the next step are computed using the coupling equations, $c(\cdot)$. This is the commonly used approach for simulation of coupled systems. In Example 9.1.6, a simulation of a coupled system using PyFMI is explained.

Example 9.1.6 (Coupled system simulation). *In order to simulate a coupled system, first of all the models needs to be loaded and collected together,*

```
sub_system1 = load_fmu("Subsystem1.fmu") #First model
sub_system2 = load_fmu("Subsystem2.fmu") #Second model
models = [sub_system1, sub_system2] #List the models
```

This list may contain an arbitrary number of models and the ordering in the list is irrelevant.

Secondly, the coupling needs to be specified. Here the following convention is used. First, from which model is the variable data coming from? It should be an reference to a model. Second, the name of the variable in the model where data is coming from. Thirdly, the reference to the receiving model and finally the name of the receiving variable.

```
#Connecting inputs / outputs from two models
connections = [(sub_system1,"x_chassi",sub_system2,"x_chassi"),
               (sub_system2,"v_chassi",sub_system1,"v_chassi")]
```

The connection list can contain an arbitrary number of connections.

The main implementation and the user entry-point is the `Master` *class for a simulation of a coupled systems. This class needs to be imported from the package.*

```
#Import of the Master object
from pyfmi import Master
```

The models together with their connections can then be loaded into the `Master` *class.*

```
#Create the simulator object
master_simulator = Master(models, connections)
```

Once the simulator object is created, a simulation is performed using the simulate method.

```
master_simulator.simulate(start_time=0.0, final_time=1.0)
```

The simulation statistics are printed and the simulation result are returned in the `res` *object, just as in the case for a simulation of a single system.*

Included in the master algorithm are variants of the above algorithm. Higher order extrapolation is possible for the inputs, from using constant polynomials, as is shown, up-to using quadratic polynomials for the inputs. Additionally, the update of inputs between time steps introduces discontinuities in the input signals due to the coupling equations. Using a smoothing approach on the inputs, continuity is preserved, cf. Section 6.3. Another issue is the stability of the algorithm, depending on the couplings between the models the algorithm may become unstable. Using the directional derivatives, a stabilization can

be performed. For details, cf. Section 6.2. Both the smoothing and the stabilization is implemented in the master algorithm.

Furthermore, the master algorithm may used together with an error estimation based on Richardson extrapolation [65]. The estimate is based on performing a global integration step twice using different input. A first step is performed using a step size H. This step is compared with two steps of step size $H/2$ where inputs and outputs between the subsystems are updated before taking the second step of step size $H/2$.

For initialization, the master algorithm supports initialization based on graph cycle detection, cf. Chapter 5. The idea is that the dependency information between inputs and outputs are used to detect cycles and in so doing, computing an evaluation order of the input / output variables of the separate models. This is done in order to simplify the initialization problem.

Simulation of coupled systems are restricted to models following the co-simulation interface for FMI 2 and as in the case of simulation of a single system, options are available to control the mater algorithm and are shown in Table 9.3.

Table 9.3: *Description of the options available in the master algorithm in order to control the simulation of the coupled system.*

Option	Description
step_size	*The global step size to be used when using the fixed step approach.*
extrapolation_order	*The order of the extrapolation for the coupling variables.*
linear_correction	*Defines if linear correction for the coupling variables should be used during the simulation.*
execution	*Defines if the models are to be evaluated in parallel or in serial.*
smooth_coupling	*Defines if the extrapolation should be smoothen, i.e. the coupling variables are adapted so that they are $\mathcal{C}^0$ instead of $\mathcal{C}^{-1}$ in case the extrapolation order is > 0.*
num_threads	*Specifies the number of threads to be used when the execution is set to parallel.*
error_controlled	*Defines if the algorithm should adapt the step size during the simulation.*
atol	*The absolute tolerance in case an error controlled simulation is performed.*
rtol	*The relative tolerance in case an error controlled simulation is performed.*
result_handling	*Specifies how the result should be handled. Either stored to file or stored in memory.*
filter	*A filter for choosing which variables to actually store result for.*

9.1.4 Parameter estimation

Verifying the dynamics of a model usually requires that parameters are validated against experimental data due to that not all parameters are known. The parameters can either be tuned manually or by an optimization aimed at minimizing the difference between experimental data and the model response. In [5], PyFMI were extended with parameter estimation using derivative free methods and which has since been further extended by coupling to SciPy's minimization algorithms and with an improved user interface. In Example 9.1.7, an example on how to perform parameter estimation is shown.

Example 9.1.7 (Parameter estimation). *This example illustrates how parameter estimation can be performed. As before, the model is loaded into Python using the* `load_fmu` *method.*

```
from pyfmi import load_fmu

# Load model
model = load_fmu("MyModel.fmu")
```

Second, the measurment data need to be stacked into a matrix. Here it assumed that the data is stored in the arrays, `t_meas`, `x1_meas` *and in* `x2_meas`.

```
#Stack the measurement data into a matrix
#The measurments, x1_meas,x2_meas, are 1-dim arrays
meas_data = np.vstack((t_meas,x1_meas,x2_meas)).transpose()
```

Following the same approach as in the simulation case, the estimation is performed by invoking the estimation method on the model object.

```
#Invoke the estimation for the parameters k1 and k2
res_est = model.estimate(parameters=['k1','k2'], measurements=(['x1'
    ,'x2'],meas_data))
```

Here, the parameters of interests, $k1$ and $k2$, are specified together with the measurement data. The estimation is performed, by default, with SciPy's Nelder-Mead routine. The resulting parameters are returned in the `res` *object.*

The parameter estimation is coupled to SciPy's optimization routine and the default algorithm used is the Nelder-Mead method [54]. The method is a derivative free method. The parameter estimation is available for all FMU types using the same syntax as shown in the above example.

9.2 Implementation overview

The core of PyFMI is implemented in Cython which is a static compiler for Python. It allows to mix the programming languages C and Python interchangeably. The added benefit of mixing the languages is that the main part of the package, where readability and scripting functionality matter, is based on Python and performance critical parts are kept in C. In this way computational performance is preserved as opposed to relying solely on Python. Not only does Cython allow to mix the languages, it also allow to connect to external C code which is imperative due to the dependency on the FMI Library.

As shown in Example 9.1.2, the high-level methods commonly uses the names of the variables instead of the value references which is an identifier for a variable, used by the FMI interface. Using the names results in a convenient way for working with variables although it introduces an overhead, cf. Figure 9.4.

The methods in the specification are connected via a high-level interface as well as access to the metadata. For specific use cases, direct access to the low-level methods are necessary and they have additionally been made available.

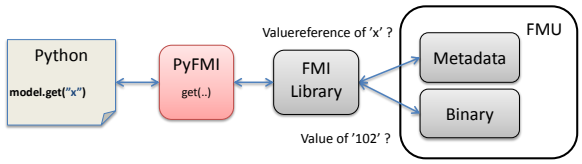

Figure 9.4: *Overview of the functionality of the high-level method* get. *The variable name for which the value is requested is sent to PyFMI. The variable name is translated into a value reference by help of the metadata using FMI Library. Using the value reference, the value is retrieved from the binary, also through FMI Library, and passed to the user.*

The algorithms implemented in the master algorithm are all based on a Jacobi-like scheme where the individual models perform a global time step and then exchange information. A global time step can be performed simultaneously for all models and thus also be straightforwardly parallelized. In PyFMI this is implemented, cf. Figure 9.5.

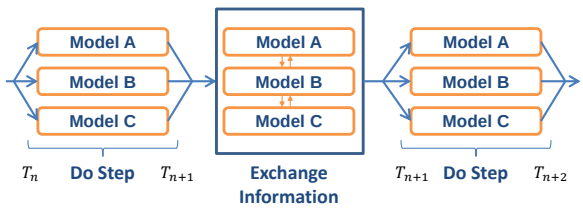

Figure 9.5: *Performing the global time steps in parallel when simulating a coupled system using the implemented master algorithm. The exchange of information is done in serial.*

9.2.1 Architecture

In PyFMI, each version and type of model defined in the standard is represented by its own Cython class, cf. Figure 9.6. The versions and model types all contain their specific func-

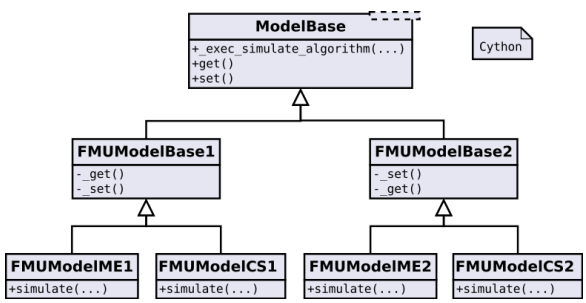

Figure 9.6: *Overview of the class diagram for the classes holding the FMUs.*

tionality and extensions. However, much of the functionality between them are common motivating the structure.

A simulation, as previously shown, is performed by invoking the simulate method on the model object.

```
#Definition of the simulate method
model.simulate( start_time='Default', final_time='Default',
                input=(), algorithm='AssimuloFMIAlg', options={} )
```

The method allows a number of arguments such as defining the start and final time of the simulation, inputs and specifying an algorithm. There are two algorithms available in PyFMI, one with the coupling to Assimulo in order to gain access to solvers and one for simulation of co-simulation FMUs. Additionally, user defined algorithms may be used. In Figure 9.7, the relation between the model objects and the algorithms together with algorithmic options are shown.

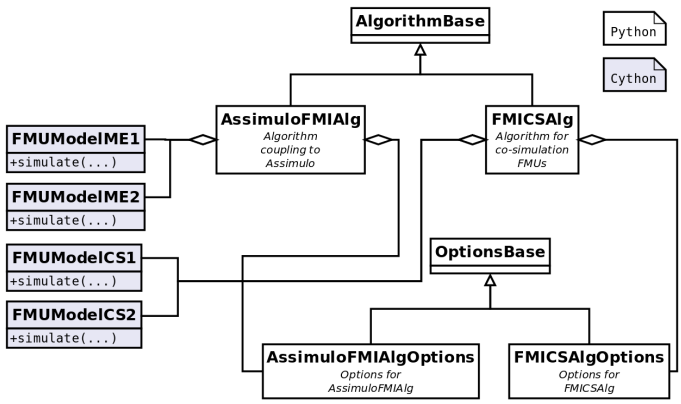

Figure 9.7: *Overview of the class diagram and the coupling between model objects and algorithms for simulation together with the algorithms options.*

A simulation of a coupled system is different from simulating a single system. This is due to that the coupled system needs to be defined. Specifically the coupling between the models needs to be specified. An algorithm for simulating a coupled system is usually called a master algorithm and here the implementation is contained in a `Master` class. In Figure 9.8, the relations between the classes are shown.

The interface for the parameter estimation follow that of the simulate method where the method is invoked on a model object.

```
#Definition of the estimate method
model.estimate( parameters, measurements,
    input=(), algorithm='SciEstAlg', options={} )
```

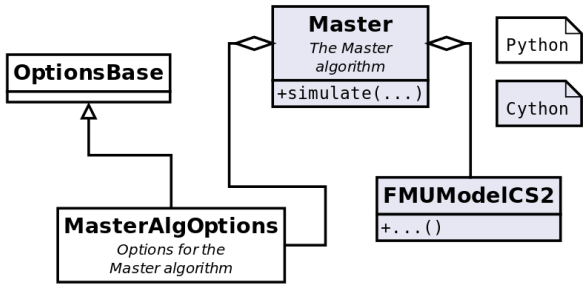

Figure 9.8: *Coupling between the FMU model class, options and the master algorithm.*

The `parameters` are the parameters of interest to tune while the `measurements` is the experimental data. Additionally, inputs can be set. In Figure 9.9, the relation between the model objects and the algorithm for parameter estimation is shown together with algorithmic options.

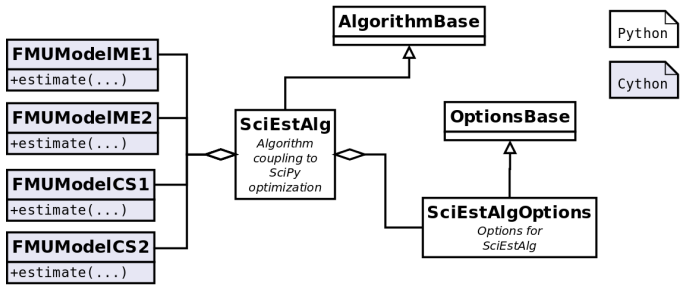

Figure 9.9: *Overview of the class diagram and the coupling between model objects and algorithms for parameter estimations together with the algorithms options.*

9.2.2 Result handling

Within the package, simulation results are handled through a base class, `ResultHandler`, that determines the interface for the underlying specific storage types, cf. Figure 9.10. The possible options are to store the result to a specific file format supported by the Modelica tool Dymola, store the result in a CSV file or store the result directly in memory. Additionally, a custom result handler can be provided to the simulation so that the result is handled in a user defined way.

The simulation result is returned to the user after a successful simulation in a general format, independent on how the result was actually stored. In Example 9.2.1, accessing the simulation results are shown.

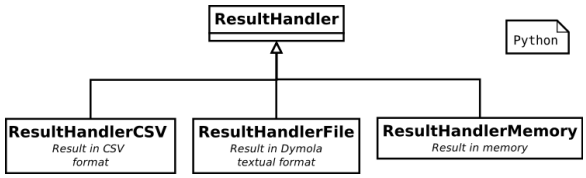

Figure 9.10: *Overview of the class diagram for the classes storing the result.*

Example 9.2.1 (Result handling). *Invoking the simulate method on a model object result in that the computed simulation result is returned.*

```
res = model.simulate()
```

Trajectories for specific variables are easily retrieved by operations on the result object.

```
res["x"] #Result trajectory the variable x
res["time"] #Result trajectory for the time
```

In case of a simulation of a coupled system, the result is returned as above.

```
res = master_simulator.simulate()
```

However, accessing the individual variable trajectories, both the model from which the variable is defined and the variable itself is needed.

```
res[sub_system1]["x"] #Result trajectory the variable x
                      #from the model object "sub_system1"
res[sub_system1]["time"] #Result trajectory for the time
                         #from the model object "sub_system1"
```

Visualization of the trajectories can easily by done using the matplotlib [37] package.

For large industrial models, the stored result can easily be gigabytes of data and the data handling can have a significant impact on the simulation performance. Coupling the result handling with the filter option in Table 9.1,Table 9.2 and Table 9.3, i.e. storing only the variables of interest, reduces both.

9.3 Case studies

9.3.1 Simulation of a woodpecker

This example is intended to show how PyFMI can handle hybrid systems, as model exchange FMUs from different sources, illustrated by a toy woodpecker, [43]. The model consists of a vertical bar attached to the ground, a sleeve able to slide along the bar and the woodpecker which is attached to the sleeve via a spring, cf. Figure 9.11. Impact is modeled without friction for simplicity. In [4], the woodpecker was defined in Python and sim-

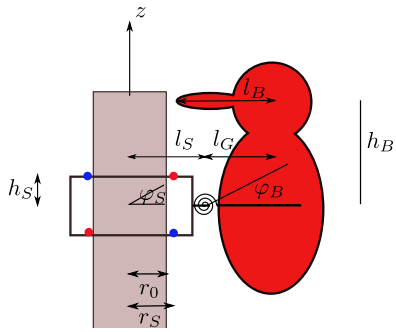

Figure 9.11: *Schematic figure of the woodpecker.*

ulated using Assimulo. Here, the model is modeled in Modelica and exported as model exchange FMUs from Dymola 2016 and JModelica.org. The Modelica code is shown in Appendix A.1.

The woodpecker is loaded into PyFMI and simulated with the solver CVode connected through Assimulo with absolute and relative tolerance set to 10^{-6}.

```
model = load_fmu("Woody.fmu")

#Get the options
opts = model.simulate_options()

#Specify tolerances
opts["CVode_options"]["atol"] = 1e-6
opts["CVode_options"]["rtol"] = 1e-6

#Simulate
res = model.simulate(final_time=tf, options=opts)
```

This was performed for the FMUs from the different tools. In Figure 9.12 the simulation results are shown for the Dymola FMU. In Figure 9.13, a comparison is made between an FMU generated from JModelica.org and Dymola, simulated using CVode and tolerances set to 10^{-6}. The reference used was computed using the JModelica.org generated FMU with the Radau5 solver connected through Assimulo together with absolute and relative tolerance set to 10^{-10}.

9.3.2 Co-simulation of a quarter car

In this example, a quarter car, cf. Figure 9.14, is simulated with step size control. In a co-simulation setup, this example was discussed in [65] and the intention with the example is to show that PyFMI are able to replicate the results shown in that article. The quarter car is governed by the equations,

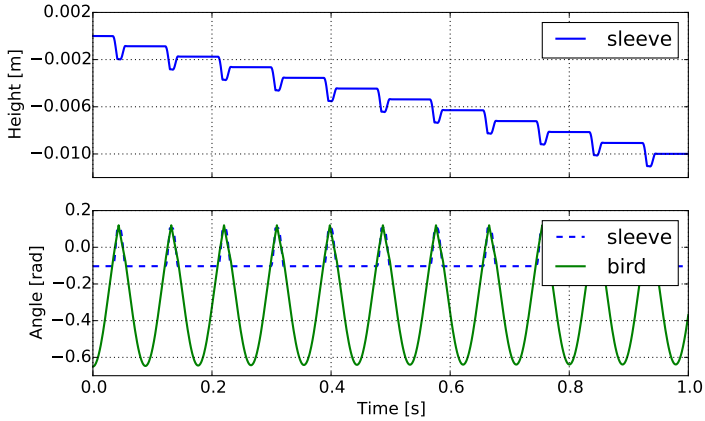

Figure 9.12: *The height of the sleeve and the angle of both the sleeve and the bird of the woodpecker from Section 9.3.1.*

$$m_c\ddot{x}_c = k_c(x_w - x_c) + d_c(\dot{x}_w - \dot{x}_c) \tag{9.11a}$$
$$m_w\ddot{x}_w = k_w(0.1 - x_w) + k_c(x_w - x_c) + d_c(\dot{x}_w - \dot{x}_c) \tag{9.11b}$$

with the constants, $m_w = 40kg$, $m_c = 400kg$, $k_w = 150000N/m$, $k_c = 15000N/m$ and $d_c = 1000Ns/m$.

The system is decoupled with the chassis being one subsystem and the wheel another. The coupling is given by

$$y = I \begin{bmatrix} x_c \\ \dot{x}_c \\ x_w \\ \dot{x}_w \end{bmatrix} \tag{9.12a}$$
$$u = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} y \tag{9.12b}$$

where there is no direct feed-through.

FMUs of the subsystems were generated using Dymola 2016 as co-simulation FMUs with support for saving the internal state and setting the internal state which allows for re-computation of a global step (Feature 2.1).

Using the implemented master algorithm, cf. Section 9.1.3, to simulate the coupled system the algorithm itself needs to be imported together with methods for loading the FMU into Python.

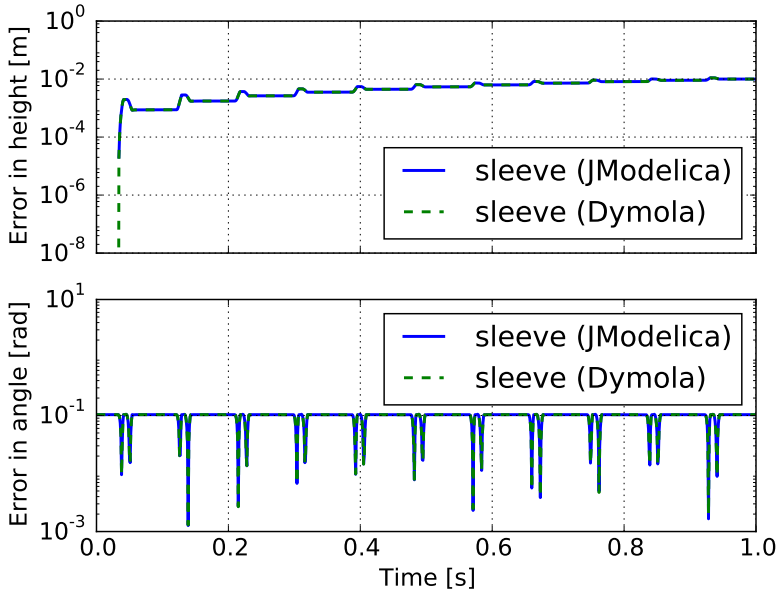

Figure 9.13: *Comparison between a JModelica.org and a Dymola generated FMU of the toy woodpecker in Section 9.3.1.*

```
from pyfmi import load_fmu
from pyfmi.master import Master
```

The FMUs are then loaded into Python.

```
#Load the FMUs
model_wheel  = load_fmu(fmu_wheel)
model_chassi = load_fmu(fmu_chassi)
```

The coupling is specified by a connection matrix where the first object specifies the model from where the output should be retrieved from. The second part specifies to what subsystem the values should be provided and to which variable.

```
#Specify the coupling
connections = [(model_chassi,"x_chassi",model_wheel,"x_chassi"),
               (model_chassi,"v_chassi",model_wheel,"v_chassi"),
               (model_wheel,"x_wheel",model_chassi,"x_wheel"),
               (model_wheel,"v_wheel",model_chassi,"v_wheel")]
```

The next step is to load the master algorithm with the models and the couplings.

```
models = [model_chassi, model_wheel]

#Load the models into the master algorithm
master_simulator = Master(models, connections)
```

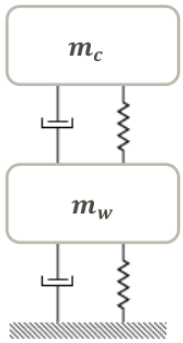

Figure 9.14: *The quarter car model from Section 9.3.2.*

Specifying the options is done through the options dictionary.

```
opts = master_simulator.simulate_options()

#(0 = Constant, 1 = Linear)
master_opts["extrapolation_order"] = 0
master_opts["error_controlled"] = True
master_opts["rtol"] = 1e-4
master_opts["atol"] = 1e-4
```

The use of Richardson for the error estimation is specified as well as both the absolute and the relative tolerance. The tolerances was set to 10^{-4}. Finally the coupled system can be simulated using the `simulate` method.

```
#Simulate the coupled system
res = master_simulator.simulate(final_time=1)
```

In Figure 9.15 the result is shown for both the position and the velocity. The figures also show the reference trajectory which was calculated by simulating the monolithic system using the solver CVode with a tolerance of 10^{-12}. The monolithic system was exported as an model exchange FMU using JModelica.org and simulated using PyFMI together with Assimulo. In Figure 9.16 the estimated error is shown together with the global step size and the time points where a step rejection occurred.

Simulations using higher order extrapolation was additionally carried out to investigate the influence of the extrapolation order on the number of steps. In Table 9.4, simulation statistics is shown for when using various order on the extrapolation. As can be seen from the table, using a higher order extrapolation polynomial results in a decrease of the number of steps.

The example show that we are able to reproduce the results in [65] using the developed tools.

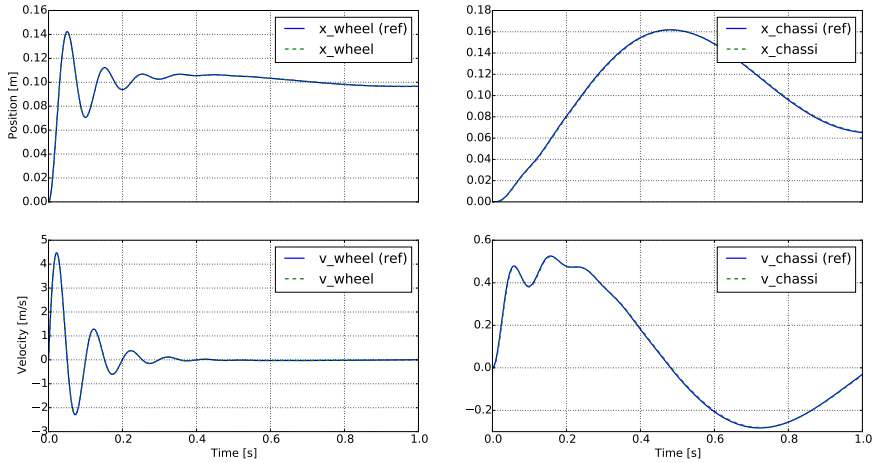

Figure 9.15: *The velocity and the position of the quarter car from Section 9.3.2 simulated using constant extrapolation and a step size of* 0.001 *together with the reference solution.*

Table 9.4: *Simulation statistics for when simulating the Quarter Car in Section 9.3.2 using various order on the extrapolation. The simulation was performed using the parallel approach together with variable step size and an absolute and a relative tolerance of* 10^{-4}.

Extrapolation order	0	1	2
Number of global steps	300	92	71
Number of error test failures	4	1	5

9.3.3 Parameter estimation in a quadruple tank

In this example, the parameter estimation capabilities within PyFMI is demonstrated on a quadruple tank model [39]. The example is inspired by the tank example in JModelica.org. The model consists of four coupled tanks, stacked two by two, and coupled so that the third tank deposits water into the first and the fourth tank deposits water into the second. The amount of water deposited is dependent on the size of the tube, connecting the tanks. Furthermore, tank one and two also have runoff dependent on the size of a tube, although they are not connected any other tank. The input to the model are voltages, controlling two pumps which pumps water into the system. Pump one pumps water into the first and fourth tank, while the second pump pumps water into the second and third, cf. Figure 9.17. The goal of the parameter estimation is to estimate the size of the tubs for the water runoff. The quadruple tank was modeled in Modelica and given in Appendix A.3, and compiled into an FMU using JModelica.org.

The input trajectories and measurement used in this example was recorded on a experimental setup of the tank system[1]. In Figure 9.18, the input voltages are shown. Furthermore, an initial estimate for the parameters controlling the runoff ($a_{[1-4]}$) are shown in

[1]The data was recorded at the Department of Automatic Control, Lund, Sweden by Kristian Soltesz

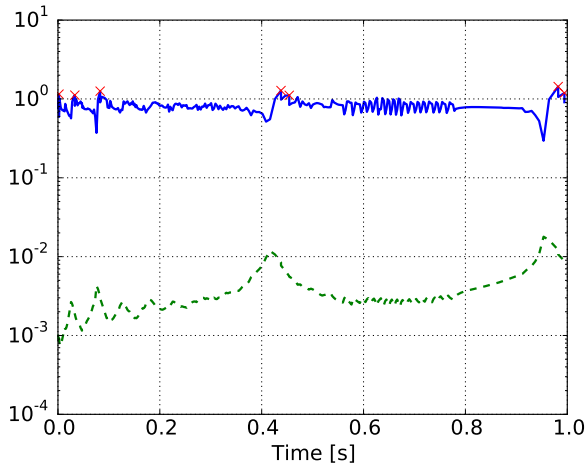

Figure 9.16: *The normalized estimated error (solid) together with the global step size (dashed) and the time points for where a step was rejected (cross) when simulating the quarter car from Section 9.3.2. The simulation was carried out using constant extrapolation together with a relative and an absolute tolerance of* 10^{-4}.

Table 9.5.

Table 9.5: *Initial parameters in Section 9.3.3.*

$a_1 = 0.03$ cm^2	$a_2 = 0.03$ cm^2
$a_3 = 0.03$ cm^2	$a_4 = 0.03$ cm^2

Now, as a first step, the data needs to be imported into Python. The data is stored in a MATLAB format and using SciPy, this can be read.

```
from scipy.io.matlab.mio import loadmat
data = loadmat('quadtank_measurements')
```

The measurement and input signals are extracted from the loaded data.

```
#Time vector
t_meas = data['t'][6000::100,0]-60
#Tank levels
x1_meas = data['y1_f'][6000::100,0]/100
x2_meas = data['y2_f'][6000::100,0]/100
x3_meas = data['y3_d'][6000::100,0]/100
x4_meas = data['y4_d'][6000::100,0]/100
#Input signals
u1 = data['u1_d'][6000::100,0]
u2 = data['u2_d'][6000::100,0]
```

With the loaded input signals and the initial parameter values, a simulation is performed as below.

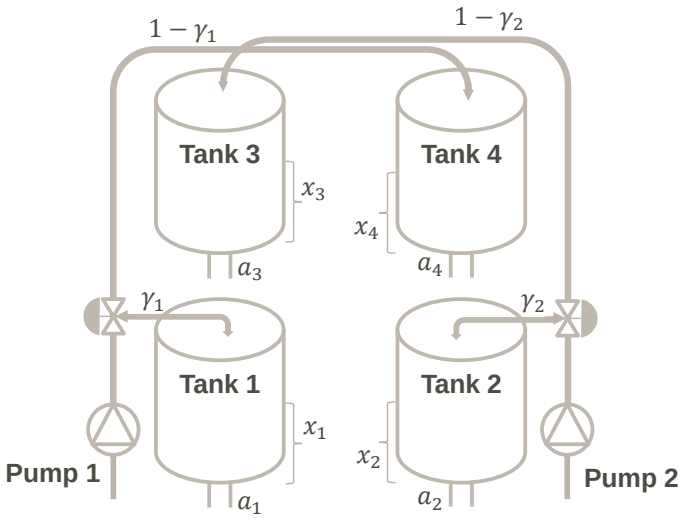

Figure 9.17: *Visualization of the quadruple tank in Section 9.3.3. The tanks all have runoff determined by a_{1-4} and the top tank deposits water into the bottom tanks. Water enters the tanks and is controlled via pumps one and two. The water level in the tanks is the variables, x_{1-4}.*

```
model = load_fmu("Quadtank.fmu") #Load the FMU

# Create the input matrix
u = N.transpose(N.vstack((t_meas,u1,u2)))

# Simulate the model response, given the initial parameters
res = model.simulate(final_time=60, input=(['u1','u2'],u))
```

The model response, for the simulation, is shown in Figure 9.19. As seen in the figure, there is a discrepancy between the simulated response and the measurement. By performing the parameter estimation, the hope is that this discrepancy will be decreased.

Performing the parameter estimation requires that the interested parameters are specified, here $a_{[1-4]}$. Furthermore, which variables that have measurements need to be specified together with the measurement data. As in the simulation case, the inputs need also be provided.

```
meas_data = N.vstack((t_meas,N.vstack((y1_meas,y2_meas,y3_meas,
    y4_meas)))).transpose()
res_est = model.estimate(parameters=['a1','a2','a3','a4'],
                         measurements=(['x1','x2','x3','x4'],
                             meas_data),
                         input=(['u1','u2'],u))
```

Using the default algorithm, the call to the estimate method will invoke the Nelder-Mead algorithm [54], which is a derivative free optimization method, included in SciPy. The

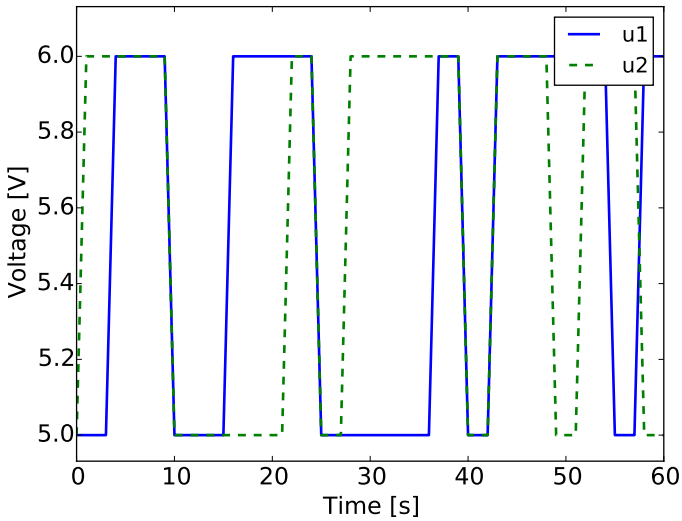

Figure 9.18: *Input signals in Section 9.3.3*

returned object contains the estimated parameters which are shown in Table 9.6. In order to verify the model response, the estimated parameter values are set to the model and the model is simulated once more.

```
model = load_fmu("Quadtank.fmu") #Load the FMU

# Setting the estimated parameter values into the model
model.set(['a1','a2','a3','a4'],
          [res_est["a1"], res_est["a2"],res_est["a3"],res_est["a4"]]
              )

# Simulate the model response, given the estimated parameters
res = model.simulate(final_time=60, input=(['u1','u2'],u))
```

The simulated response, given the estimated parameter values, are shown in Figure 9.20. As seen in the figure, using the estimated parameters, the model response has substantially been improved when compared to the simulation with the initial parameter values, Figure 9.19.

Table 9.6: *Estimated parameters in Section 9.3.3.*

$a_1 = 0.02660115\ \text{cm}^2$	$a_2 = 0.0270179\ \text{cm}^2$
$a_3 = 0.03008687\ \text{cm}^2$	$a_4 = 0.02929907\ \text{cm}^2$

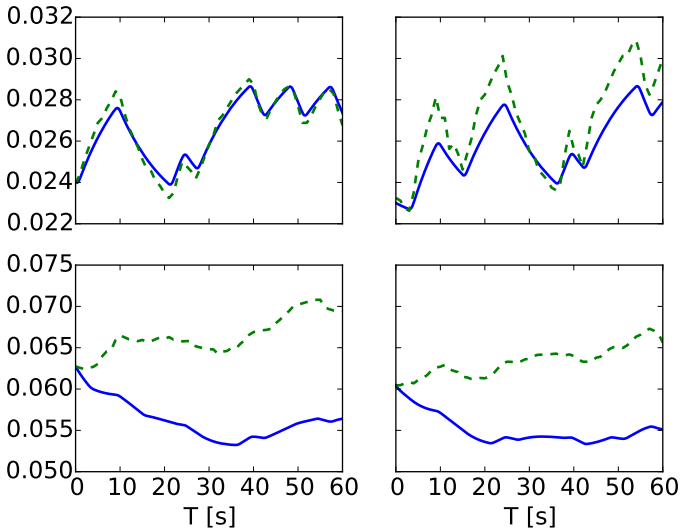

Figure 9.19: *A comparison of the measured data (dashed) together with the simulated response (solid) given the initial values of the parameters. The trajectories are the tank levels. The left top figure represents the third tank and the top right, the fourth tank. The left bottom figure is the first tank while the right figure is the second.*

9.3.4 Parallel co-simulation of a race car

In this example, a race car is modeled in Modelica using the commercial Vehicle Dynamics Library. The race car model is the same as previously seen in Part II. In the example, the car is driven by a virtual driver that tries to stay onto an eight shaped course with increasing velocity in order to investigate the dynamic response of the car, especially when changing the turning direction. The model is simulated as a coupled system in a co-simulation setup where the model has been separated into wheels and chassis, Figure 9.21. The intention of the example is to highlight the parallelization feature in the implemented master algorithm. The model of the chassis was compiled into a co-simulation FMU using Dymola 2016 while the model of a wheel was exported using JModelica.org. The models contain about 90k parameters, constants and variables in total.

An increase in performance using the parallelization can only be expected if the majority of the simulation time is not spent in a single model. In this example, more time is spent in the simulation of the chassis than for a wheel, cf. Table 9.7. However, when considering the total simulation time and the chassis part of it, a speedup is expected when using the parallelization.

In order to specify that global steps, in the master algorithm, should be performed in parallel, the options need to be set.

```
# Retrieve the simulation options
master_options = master_simulator.simulate_options()
```

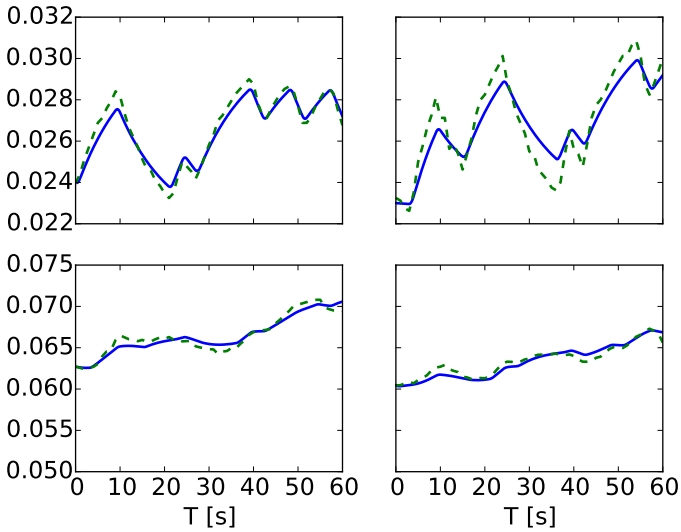

Figure 9.20: *A comparison of the measured data (dashed) together with the simulated response (solid) given the estimated values of the parameters. The trajectories are the tank levels. The left top figure represents the third tank and the top right, the fourth tank. The left bottom figure is the first tank while the right figure is the second.*

Table 9.7: *Normalized elapsed time, for each model, for a simulation of the race car from Section 9.3.4. The overhead is time not spent in the separate models, storing the results for example.*

Total	Chassis	Wheels (each)	Overhead
1.0	0.31	0.16	0.05

```
master_options["execution"] = "parallel"
master_options["num_threads"] = 1
```

Furthermore, due to the amount of variables and parameters in the models, the filter is set so that only the interesting variables are stored.

```
master_options["filter"] = {model_chassi:"*summary*" ,
                model_wheel_lf: "forces.f_*",
                model_wheel_lb: "forces.f_*",
                model_wheel_rf: "forces.f_*",
                model_wheel_rb: "forces.f_*"}
```

The test was run on laptop with two cores. Using the two cores, the simulation time was reduced by 34%. While this is not optimal, this is still a substantial decrease of the simulation time.

The full Python script can be found in Appendix A.4.

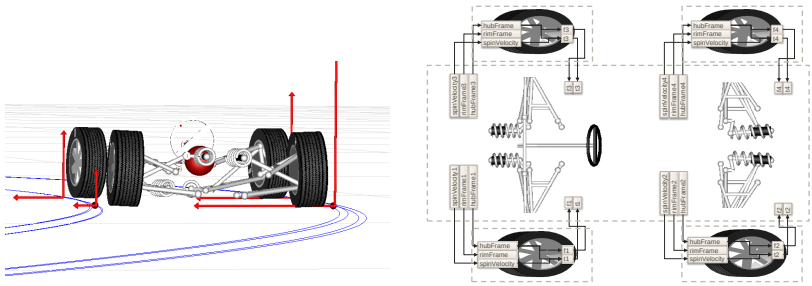

Figure 9.21: *Visualization of the race car (left) and visualization of the couplings in the race car from Section 9.3.4 in a co-simulation setup where the wheel and chassis has been divided into separate models (right). © Modelon.*

9.3.5 Sparsity exploitation in a chromatography separation process

In [35], the robustness of a high-pressure liquid chromatographic process (Figure 9.22) was investigated. Given a nominal input trajectory, the aim was to quantify the robustness of the process with regards to disturbances in the input. The process is described by an ODE with a scalar input,

$$\dot{x} = f(x, u), \quad x \in \mathbb{R}^{142}, u \in \mathbb{R}. \tag{9.13}$$

An FMU for the process was generated from Dymola 2016.

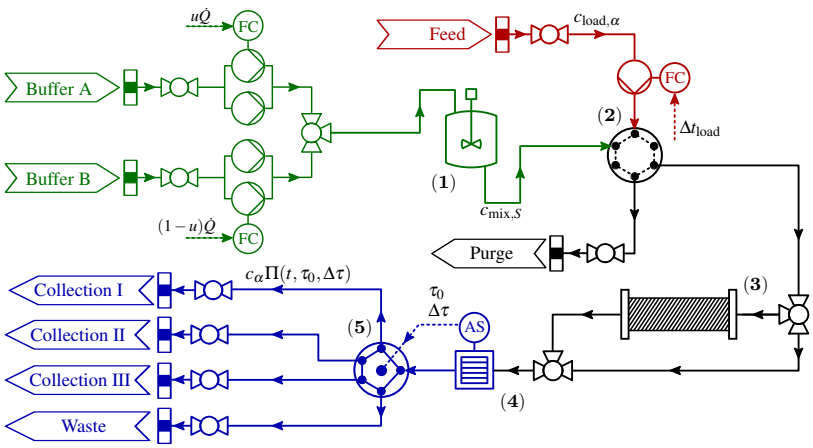

Figure 9.22: *Visualization of the chromatography separation process used in Section 9.3.5. The input, u, controls how much of buffer A and B enters the process through the mixing tank (1). The feed enters the system through (2), where also the buffers passes. The separation takes place in the separation column (3). The output from the process is collected in either I, II, III or dumped as waste. The collection is determined using the detector (4) and the valve (5).*

In order to quantify robustness towards disturbances, a Lyapunov equation needs to be solved,

$$\dot{P} = AP + PA^T + BB^T, \quad P(t_0) = B(t_0)B^T(t_0) \tag{9.14}$$

where, $A = \frac{\partial f}{\partial x}$ and $B = \frac{\partial f}{\partial u}$. The primary focus of this section is to demonstrate how sparsity information in FMUs can be used to significantly decrease simulation times. For a full problem statement and results, cf. [35]. In Figure 9.23, the structure of A is shown. As Equation 9.14 is matrix valued, we first vectorize the equation which results in,

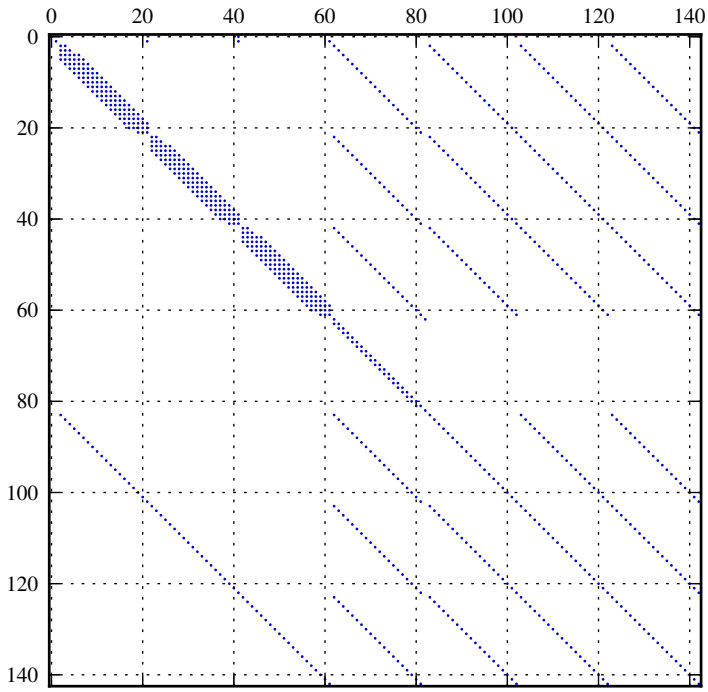

Figure 9.23: *The structure of the matrix A from Section 9.3.5.*

$$\text{vec}(\dot{P}) = (I \otimes A)\text{vec}(P) + (A \otimes I)\text{vec}(P) + \text{vec}(B^T B), \tag{9.15}$$

where $\otimes$ is the Kronecker product. The full system can then be formed as,

$$\begin{bmatrix} \dot{x} \\ \text{vec}(\dot{P}) \end{bmatrix} = \begin{bmatrix} f(t, x, u) \\ (I \otimes A)\text{vec}(P) + (A \otimes I)\text{vec}(P) + \text{vec}(B^T B) \end{bmatrix}. \tag{9.16}$$

Furthermore, the process model results in a stiff problem which requires an implicit solver.

Thus, we need the Jacobian of Equation 9.16 which is defined as,

$$J = \begin{bmatrix} A & 0 \\ 0 & I \otimes A + A \otimes I \end{bmatrix}. \tag{9.17}$$

In Figure 9.24, the structure of the Jacobian is shown.

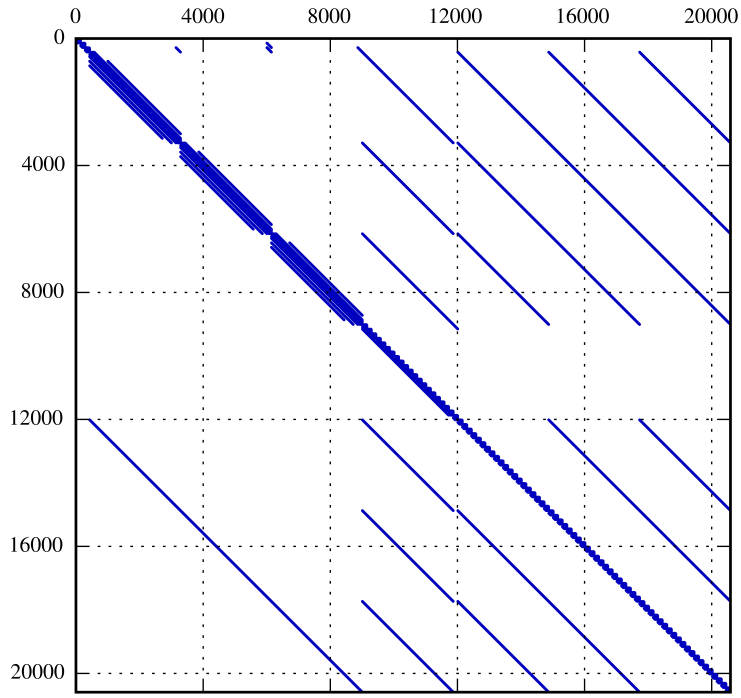

Figure 9.24: *The structure of the Jacobian for the augmented system, Equation 9.17, used in Section 9.3.5.*

In order to solve the augmented system, Equation 9.16, PyFMI was extended to be able to add equations that are solved together with the FMU. Furthermore, in order to efficiently solve the above problem, the structure of A needs to be taking into account. By using the compression discussed in Section 9.1.1, the number of calls to the directional derivatives was reduced by 90%. Furthermore, with the connection to SuperLU [21] from CVode, the Jacobian can be provided as a sparse matrix.

By using both the compression for computing A and by providing the Jacobian (Equation 9.17) as a sparse matrix, the simulation time was reduced to only 4% of the original time where a dense representation of the Jacobian was used and no compression.

In Appendix A.5, the full Python script for adding the Lyapunov equations to a simulation of an FMU is shown.

9.4 Summary

In this chapter, we presented PyFMI, a software for working with models following the Functional Mock-up Interface. The package support models following version 1.0 and 2.0 of the standard as well as the different model types, model exchange and co-simulation. Interactions with the models are conveniently performed using high-level methods and if needed, access to the low-level methods is additionally available.

With a connection to the simulation package Assimulo, simulation of model exchange FMUs can be performed using state of the art integrators. For coupled systems, PyFMI implements a master algorithm for simulation of coupled co-simulation FMUs. Furthermore, the simulation analyses are complemented with support for parameter estimation. Having these analyses easily available in an open tool, we hope that the standard will continue to grow and spread even further.

The package is demonstrated on a number of problems and show promising results. PyFMI[2] is freely available under the LGPL [25] license.

[2]http://www.pyfmi.org [accessed 2016-03-24]

Conclusions and future work

Chapter 10

Conclusions and future work

10.1 Coupled systems

In this thesis, stability properties of different approaches for simulation of weakly coupled linear systems with feed-through have been studied. We considered the consistent approach, where the constraint equations are solved at each global time step, and the inconsistent approach, where the constraint equations are not solved at the global time steps. Furthermore, higher order extrapolations for the inputs between the global time steps were considered for both approaches.

In Table 6.1, the coupling stability requirements are summarized. Using higher order extrapolation in the inconsistent approach leads to a smaller stability region. It could be shown that using analytical derivatives, instead of using approximated derivatives, to compute the linear extrapolation of the inputs, stability properties from the constant extrapolation case are recovered.

In a co-simulation setting, the update of the inputs at each global time step introduces discontinuities which can lead to a general performance degradation. In Section 6.3, a smoothing approach was considered that preserves the continuity of the inputs and that does not affect stability, cf. Table 6.1.

Using the FMI standard, only the inconsistent approaches can be used for simulation of the coupled system. This is due to Restriction 2.1 which does not allow recomputing the outputs based on new input values at global time steps. This is a problem if there is an algebraic loop in the coupled system as the inconsistent approaches are not unconditionally coupling stable. In order to overcome this limitation, a linear correction to the inconsistent approaches was proposed in Section 6.2. For linearly coupled linear subsystems, the correction to the inconsistent approach is equivalent to the consistent approach. The correction has been tested on a number of linear and nonlinear test examples and shown to stabilize the simulations.

In Chapter 5, a structural approach for initializing coupled systems is presented. With the structural approach, an evaluation order of subsystem inputs and outputs is computed,

and algebraic loops are detected. The inputs and outputs involved in algebraic loops need to be solved simultaneously while the others are computed sequentially. Performance gains can be obtained by exploiting the non-uniqueness of the evaluation order of the outputs. Reordering the outputs leads to fewer subsystem evaluations and an algorithm for computing the reordering is proposed in Algorithm 2. In our tests this approach reduced the number of subsystem evaluations and increased performance. Further studies are necessary in order for a general claim about the algorithm's performance.

Due to its unconditionally coupling stability, a master algorithm implementing the consistent approach is a significant improvement compared to the inconsistent approach. Using the structural approach for detecting algebraic loops together with the proposed algorithm for reducing subsystem evaluations during simulation is another source of improvement. Consequently we propose to remove Restriction 2.1 to allow these improvements in future versions of the standard.

With the current versions of the FMI standard (version 1.0 and version 2.0) there is no means to rigorously handle subsystems with events in a co-simulation setting. Events are handled internally in co-simulation FMUs and they are not exposed externally. There is a discussion within the FMI community to introduce a new kind of FMU, a hybrid co-simulation FMU, with the possibility of signaling events. This direction is important to enable robust and accurate co-simulation of FMUs with events and should be pursued.

10.2 Subsystem solvers

In Chapter 7, subsystem solvers were discussed. In a co-simulation context, the inputs to a subsystem are typically discontinuous at communication points. Such discontinuities pose challenges for internal integrators, in particular in the case of multistep methods, as they may lead to order reductions and step size reductions in the method. A modification to the predictor for multistep methods was proposed and it could be shown to reduce simulation times by up to 50% for the examples considered. A formal analysis of the impact on the method, with regards to stability and accuracy, is left for future work.

Furthermore, as the considered case here is with an internal solver in a subsystem, potential performance improvements can be made to the proposed modification. This is due to that with an internal solver, more information can potentially be shared between the subsystem and the solver allowing for tailored modifications. If information about how an input impacts the subsystem equations is available a full update of the Jacobian might be unnecessary. This is left for future studies.

10.3 Software

Two Python packages have been developed and are presented in this thesis: Assimulo and PyFMI. The packages have been successfully applied to a number of industrial problems and in teaching. Both packages are open source and freely available and the number of

downloads of the packages currently counts in the thousands. As both packages are open and are publicly available, they support the further adoption of the FMI standard and add to its success.

The co-simulation master algorithm, which implements the approaches presented in this thesis is available as part of PyFMI. It is intended for the master algorithm available in PyFMI to serve as a basis for further research, as well as offer a capable means of solving industrial co-simulation problems.

The packages have been successfully used in research, such as in [35] and [6]. Furthermore, they are integrated as the simulation engine into the open source tool JModelica.org and the commercial OPTIMICA Compiler Toolkit [53]. Within these tools, the packages are used to successfully simulate complex industrial models from commercial Modelica libraries such as the Vehicle Dynamics Library as well as models from the Modelica Standard Library.

Apart from its use in research and in industry, Assimulo is also used as a teaching tool at the Centre for Mathematical Sciences at Lund University[1,2]. Due to its use as a teaching tool, a great effort has been put into the documentation which contains detailed information about all the solvers, tutorials and an extensive set of examples showing how the various solvers are used and can serve as a basis for further studies. Additionally, its use as a teaching tool contributed to its development.

There are discussions within the FMI community to also extend the directional derivatives with respect to parameters. This would allow to compute the sensitivity equations of a model, with respect to parameters. As the solver CVodes [33] supports the addition of the sensitivity equations to the original problem, and due to its availability through Assimulo adding support within PyFMI is straightforward.

[1] http://ctr.maths.lu.se/na/courses/FMNN05/ [accessed: 2016-03-11]
[2] http://ctr.maths.lu.se/na/courses/FMNN25/ [accessed: 2016-03-11]

Appendix A

Reference system models

A.1 The woodpecker

The equations of motion of the toy woodpecker stated here are a slight simplification of the model given in [43]. The equations are given for the three model states described in Equation 8.10 separately. The dependent variables are $z, \varphi_S, \varphi_B, \lambda_1, \lambda_2$, where z denotes the vertical translatory degree of freedom, φ the inclination of the sleeve S and the bird B, $\lambda_i, i = 1, 2$ are the Lagrange multipliers if constraints are active (State II, III) or zero otherwise (State I). The linearized equations of motion for State I are,

$$f_1(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = -(m_S + m_B)g \tag{A.1a}$$
$$f_2(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_B - \varphi_S) - m_B l_S g - \lambda_1 \tag{A.1b}$$
$$f_3(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_S - \varphi_B) - m_B l_G g - \lambda_2 \tag{A.1c}$$
$$0 = \lambda_1 \tag{A.1d}$$
$$0 = \lambda_2 \tag{A.1e}$$

where,

$$f_1(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = (m_S + m_B)\ddot{z} + m_B l_S \ddot{\varphi}_S + m_B l_G \ddot{\varphi}_B \tag{A.2a}$$
$$f_2(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = (m_B l_S)\ddot{z} + (J_S + m_B l_S^2)\ddot{\varphi}_S + (m_B l_S l_G)\ddot{\varphi}_B \tag{A.2b}$$
$$f_3(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = (m_B l_G)\ddot{z} + (m_B l_S l_G)\ddot{\varphi}_S + (J_B + m_B l_G^2)\ddot{\varphi}_B. \tag{A.2c}$$

For State II, the equations of motion read,

$$f_1(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = -(m_S + m_B)g \tag{A.3a}$$
$$f_2(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_B - \varphi_S) - m_B l_S g - h_S \lambda_1 - r_S \lambda_2 \tag{A.3b}$$
$$f_3(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_S - \varphi_B) - m_B l_G g \tag{A.3c}$$
$$0 = h_S \ddot{\varphi}_S \tag{A.3d}$$
$$0 = \dot{z} + r_S \dot{\varphi}_S. \tag{A.3e}$$

Table A.1: *Parameters used in the woodpecker example, cf. Appendix A.1. B stands for parameters related to the bird, while S denotes the sleeve.*

$m_S = 3.0\text{E-}4$ kg	$m_B = 4.5\text{E-}3$ kg	$J_S = 5.0\text{E-}9$ kgm	$J_B = 7.0\text{E-}7$ kgm
$r_0 = 2.5\text{E-}3$ m	$r_S = 3.1\text{E-}3$ m	$h_S = 5.8\text{E-}3$ m	$l_S = 1.0\text{E-}2$ m
$l_G = 1.5\text{E-}2$ m	$l_B = 2.01\text{E-}2$ m	$h_B = 2.0\text{E-}2$ m	$c_P = 5.6\text{E-}3$ N/rad
$g = 9.81$ m/s^2			

Finally for State III, the equations of motion are given by,

$$f_1(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = -(m_S + m_B)g - \lambda_2 \tag{A.4a}$$
$$f_2(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_B - \varphi_S) - m_B l_S g + h_S \lambda_1 - r_S \lambda_2 \tag{A.4b}$$
$$f_3(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_S - \varphi_B) - m_B l_G g \tag{A.4c}$$
$$0 = -h_S \ddot{\varphi}_S \tag{A.4d}$$
$$0 = \dot{z} + r_S \dot{\varphi}_S. \tag{A.4e}$$

The parameter values are defined in Table A.1.

Assimulo description

Simulating the model using Assimulo first requires that the problem class and the solver class are imported into Python.

```
from assimulo.problem import Implicit_Problem
from assimulo.solvers import IDA
```

The model is a hybrid DAE and we imported the solver IDA for performing the simulation. The residual is defined in a method `res` using Equation A.1, A.3 and A.4 together with a switch parameter *sw*, which indicates the model state. This parameter is kept constant during the integration and only changed at an event.

```
def res(t,y,yd,sw):

    z,phiS,phiB,zp,phiSp,phiBp,lam1,lam2 = y
    zpp,phiSpp,phiBpp = yd[3:6]

    pre1 = (mS+mB)*zpp+mB*lS*phiSpp+mB*lG*phiBpp
    pre2 = mB*lS*zpp+(JS+mB*lS**2)*phiSpp+mB*lS*lG*phiBpp
    pre3 = mB*lG*zpp+mB*lS*lG*phiSpp+(JB+mB*lG**2)*phiBpp

    res01 = y[3]-yd[0]
    res02 = y[4]-yd[1]
    res03 = y[5]-yd[2]

    if sw[0]: #State I
        res1 = pre1+(mS+mB)*g
        res2 = pre2-cP*(phiB-phiS)+mB*lS*g+lam1
```

```
        res3 = pre3-cP*(phiS-phiB)+mB*lS*g+lam2
        res4 = lam1
        res5 = lam2

    if sw[1]: #State II
        res1 = pre1+(mS+mB)*g+lam2
        res2 = pre2-cP*(phiB-phiS)+mB*lS*g+hS*lam1+rS*lam2
        res3 = pre3-cP*(phiS-phiB)+mB*lG*g
        res4 = hS*phiSpp
        res5 = zp+rS*phiSp

    if sw[2]: #State III
        res1 = pre1+(mS+mB)*g+lam2
        res2 = pre2-cP*(phiB-phiS)+mB*lS*g-hS*lam1+rS*lam2
        res3 = pre3-cP*(phiS-phiB)+mB*lG*g
        res4 = -hS*phiSpp
        res5 = zp+rS*phiSp

    return N.array([res01,res02,res03,res1,res2,res3,res4,res5])
```

The event indicators, Equation 8.15, are defined in a `state_events` method.

```
def state_events(t,y,yd,sw):

    z,phiS,phiB,zp,phiSp,phiBp,lam1,lam2 = y
    zpp,phiSpp,phiBpp = yd[3:6]

    event_1 = hS*phiS+(rS-r0)
    event_2 = hS*phiS-(rS-r0)
    event_3 = lam1
    event_4 = hB*phiB-lS-lG+lB+r0

    return N.array([event_1,event_2,event_3,event_4])
```

The third method that is defined handles the events once they have been detected. This is the method responsible for the actual transition between the states and it is called once an event indicator has indicated the occurrence of an event.

```
def handle_event(solver, event_info):

    events = event_info[0]

    if solver.sw[0]: #We are in the first state
        if events[0] and solver.y[5] < 0: #Switch from 1 to 2
            solver.sw[0] = False
            solver.sw[1] = True

            solver.y[5] = 1./(JB+mB*lG**2)*(mB*lG*solver.y[3]+ \
                mB*lG*lS*solver.y[4]+(JB+mB*lG**2)*solver.y[5])
            solver.y[3] = 0.0
            solver.y[4] = 0.0
```

```
            return

        if events[1] and solver.y[5] > 0: #Switch from 1 to 3
            solver.sw[0] = False
            solver.sw[2] = True

            solver.y[5] = 1./(JB+mB*lG**2)*(mB*lG*solver.y[3]+ \
                mB*lG*lS*solver.y[4]+(JB+mB*lG**2)*solver.y[5])
            solver.y[3] = 0.0
            solver.y[4] = 0.0

            return

    if solver.sw[1]: #We are in the second state
        if events[2]: #Switch from 2 to 1
            solver.sw[1] = False
            solver.sw[0] = True
            return

    if solver.sw[2]: #We are in the third state
        if events[2] and solver.y[5] < 0: #Switch from 3 to 1
            solver.sw[2] = False
            solver.sw[0] = True
            return
        if events[3] and solver.y[5] > 0: #Woodpecker hit
            solver.y[5] = -solver.y[5]

            return
```

Prior to starting the simulation, initial conditions are specified. They are specified such that the woodpecker starts in the second state.

```
x0  = [0.0, -0.1, -0.65, 0.0 ,0.0 ,0.0 ,0.0 , 0.0]
xd0 = [0.0, 0.0, 0.0, 0.0 ,0.0 ,0.0 ,0.0, 0.0]
switches0 = [False,True,False]
```

Using the initial conditions together with the residual method, an implicit problem is created.

```
woody = Implicit_Problem(res,x0,xd0,sw0=switches0)
```

Additional information is provided to the problem, such as the event indicators, how to handle an event once it has been detected and also the name of the problem together with information about which variables are dynamic and which are algebraic.

```
woody.state_events = state_events #Provide the event indicators
woody.handle_event = handle_event #How to handle an event
woody.name = "Woodpecker w/o friction"

#Specify the dynamic variables (1) and the algebraic variables (0)
```

```
woody.algvar = [1]*6+[0]*2
```

A simulation is then performed using the `simulate` method, where we have additionally specified that the algebraic variables have to be excluded from the error test. Excluding algebraic variables from the error test is required when dealing with DAEs of index larger than one, as the error model in most error estimators does not apply to algebraic variables and would overestimate the error. Sundials provides special control parameters to exclude these variables from the error test, which are also accessible in Assimulo.

```
sim = IDA(woody)

#Specify simulation options
sim.suppress_alg = True #Suppress the algebraic variables
                        #from the error test.

t,x,xd = sim.simulate(0.16)
```

The computed solution trajectories are returned and stored in `t`, `x` and `xd`

Modelica description

Modelica code representing the toy woodpecker is shown below.

```
model Woody
  //Constants
  constant Real g = 9.81;
  //Parameters
  parameter Real mS = 3.0e-4, mB = 4.5e-3;
  parameter Real r0 = 2.5e-3, rS = 3.1e-3;
  parameter Real JS = 5.0e-9, JB = 7.0e-7;
  parameter Real hS = 5.8e-3, hB = 2.0e-2;
  parameter Real lS = 1.0e-2, lB = 2.01e-2;

  parameter Real masstotal = mS + mB;
  parameter Real rM = rS - r0;
  parameter Real cp = 5.6e-3, lG = 1.5e-2;

  //Continuous variables
  Real z(start = 0.0),          zp(start = 0.0);
  Real phiS(start = -0.10344), phiSp(start = 0.0);
  Real phiB(start = -0.65),     phiBp(start = 0.0);
  Real phiBpp(start = 1.40059e2);
  Real lam1(start = -0.6911), lam2(start = -0.1416);
  Integer state(start = 2, fixed = true);
  discrete Real last_update(start=0);

equation
  der(z) = zp;
  der(phiS) = phiSp;
  der(phiB) = phiBp;
```

```
  phiBpp = der(phiBp);
  if state == 1 then
    masstotal * der(zp) + mB * lS * der(phiSp) + mB * lG * der(phiBp
        ) + masstotal * g = 0;
    mB * lS * der(zp) + (JS + mB * lS * lS) * der(phiSp) + mB * lS *
         lG * der(phiBp) - cp * (phiB - phiS) + mB * lS * g = -lam1;
    mB * lG * der(zp) + mB * lS * lG * der(phiSp) + (JB + mB * lG *
        lG) * der(phiBp) - cp * (phiS - phiB) + mB * lG * g = -lam2;
    lam1 = 0;
    lam2 = 0;
  elseif state == 2 then
    masstotal * der(zp) + mB * lS * der(phiSp) + mB * lG * der(phiBp
        ) + masstotal * g = -lam2;
    mB * lS * der(zp) + (JS + mB * lS * lS) * der(phiSp) + mB * lS *
         lG * der(phiBp) - cp * (phiB - phiS) + mB * lS * g = (-hS *
         lam1) - rS * lam2;
    mB * lG * der(zp) + mB * lS * lG * der(phiSp) + (JB + mB * lG *
        lG) * der(phiBp) - cp * (phiS - phiB) + mB * lG * g = 0;
    //Index 3
    //0 = (rS-r0)+hS*phiS;
    //0 = der(z)+rS*phiSp;
    //Index 1
    0 = hS * der(phiSp);
    0 = der(zp) + rS * der(phiSp);
  else
    masstotal * der(zp) + mB * lS * der(phiSp) + mB * lG * der(phiBp
        ) + masstotal * g = -lam2;
    mB * lS * der(zp) + (JS + mB * lS * lS) * der(phiSp) + mB * lS *
         lG * der(phiBp) - cp * (phiB - phiS) + mB * lS * g = hS *
        lam1 - rS * lam2;
    mB * lG * der(zp) + mB * lS * lG * der(phiSp) + (JB + mB * lG *
        lG) * der(phiBp) - cp * (phiS - phiB) + mB * lG * g = 0;
    //Index 3
    //0 = (rS-r0)-hS*phiS;
    //0 = der(z)+rS*phiSp;
    //Index 1
    0 = -hS * der(phiSp);
    0 = der(zp) + rS * der(phiSp);
  end if;
algorithm
  when {rM + hS * phiS < 0.0, rM - hS * phiS < 0.0} then
    if state == 1 and phiBp < 0 then
      state := 2;
      last_update := time;
    end if;
    if state == 1 and phiBp > 0 then
      state := 3;
      last_update := time;
    end if;
  elsewhen {lam1 > 1e-8, lam1 < -1e-8} then
```

```
    if state == 2 and time - last_update > 0 then
        last_update := time;
      state := 1;
    end if;
    if state == 3 and time - last_update > 0 then
      state := 1;
      last_update := time;
    end if;
  end when;
equation
  when {hB * phiB - (lS + lG - lB - r0) > 0 and phiBp > 0} then
    reinit(phiBp, -pre(phiBp));
  elsewhen state == 2 then
    reinit(phiBp, (mB * lG * pre(zp) + mB * lS * lG * pre(phiSp) + (
        JB + mB * lG * lG) * pre(phiBp)) / (JB + mB * lG * lG));
    reinit(phiSp, 0.0);
    reinit(zp, 0.0);
  elsewhen state == 3 then
    reinit(phiBp, (mB * lG * pre(zp) + mB * lS * lG * pre(phiSp) + (
        JB + mB * lG * lG) * pre(phiBp)) / (JB + mB * lG * lG));
    reinit(phiSp, 0.0);
    reinit(zp, 0.0);
  end when;
end Woody;
```

A.2 The van der Pol oscillator

Modelica code representing the van der Pol oscillator used in the examples in Section 9.1.

```
model VDP
    // Parameters
    parameter Real mu = 2e1;

    // The states
    Real x1(start=-0.6);
    Real x2(start=2);

    // The control signal
    input Real u;

  equation
    der(x1) = mu*((1 - x2^2) * x1 - x2) + u;
    der(x2) = x1;
end VDP;
```

A.3 Quadruple tank

Modelica code representing the quadruple tank model from Section 9.3.3 is shown below. Courtesy of JModelica.org.

```
model QuadTank
// Process parameters
parameter Modelica.SIunits.Area A1=4.9e-4, A2=4.9e-4, A3=4.9e-4, A4
    =4.9e-4;
parameter Modelica.SIunits.Area a1(min=1e-6,nominal=1e-6)=0.03e-4,
    a2(nominal=1e-6)=0.03e-4;
parameter Modelica.SIunits.Area a3(nominal=1e-6)=0.03e-4, a4(nominal
    =1e-6)=0.03e-4;
parameter Modelica.SIunits.Acceleration g=9.81;
parameter Real k1_nmp(unit="m^3/s/V") = 0.56e-6, k2_nmp(unit="m^3/s/
    V") = 0.56e-6;
parameter Real g1_nmp=0.30, g2_nmp=0.30;

// Tank levels
Modelica.SIunits.Length x1(start=0.0627);
Modelica.SIunits.Length x2(start=0.06044);
Modelica.SIunits.Length x3(start=0.024);
Modelica.SIunits.Length x4(start=0.023);

// Inputs
input Modelica.SIunits.Voltage u1;
input Modelica.SIunits.Voltage u2;

equation
der(x1) = -a1/A1*sqrt(2*g*x1) + a3/A1*sqrt(2*g*x3) +
                g1_nmp*k1_nmp/A1*u1;
der(x2) = -a2/A2*sqrt(2*g*x2) + a4/A2*sqrt(2*g*x4) +
                g2_nmp*k2_nmp/A2*u2;
der(x3) = -a3/A3*sqrt(2*g*x3) + (1-g2_nmp)*k2_nmp/A3*u2;
der(x4) = -a4/A4*sqrt(2*g*x4) + (1-g1_nmp)*k1_nmp/A4*u1;

end QuadTank;
```

A.4 Race car

For racing applications, finding the maximal performance of the car is crucial. One method to quickly estimate the impact on performance of a change to the vehicle setup is to solve for the steady state limits under different driving conditions. Identifying a set of critical points along a race track and calculating the maximum achievable speed for each point can give a good indication on how the change will affect the lap time. Simulations can be carried out with predefined input or by a feedback loop using either a simulator or a virtual driver model to investigate the dynamic response.

Here a race car is modeled in Modelica using the commercial Vehicle Dynamics Library [7], Figure A.1. In the model, the car is driven by a virtual driver that tries to stay onto an eight shaped course with increasing velocity in order to investigate the dynamic response of the car, especially when changing the turning direction. The model contains about 90k

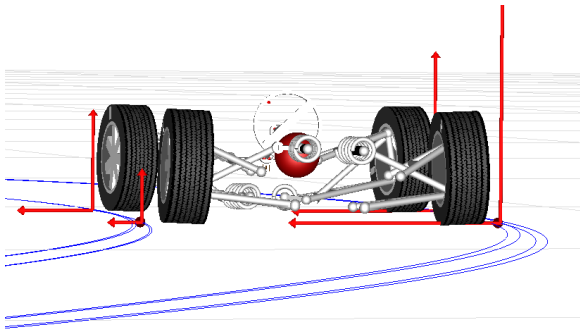

Figure A.1: *Visualization of the race car from Section A.4. © Modelon.*

parameters, constants and variables resulting in that the XML data file is 700k lines when compiled into an FMU. There are 47 continuous states and 44 event indicators.

The race car model is used in a number of different settings within this thesis. The model is simulated as a monolithic model and in a co-simulation setup where the model has been separated into wheels and chassis, Figure A.2. In a co-simulation setup, there are 172 connections between the separate models. The example is considered in Section 5.3.2 where initialization of the model is discussed. In Section 6.4.2, the different approaches for co-simulation are tested on the model. Further, in Section 7.4.3, a wheel from the model is considered when evaluating the modification to the predictor in a multistep method. Finally, in Section 9.3.4, the model is used to evaluate parallelization in PyFMI.

PyFMI description

This section describes how the race car can be simulated, as a coupled system, using PyFMI. Before a simulation can be started, the methods for loading an FMU and the master algorithm is imported into Python.

```
from pyfmi import load_fmu
from pyfmi.master import Master
```

The models are loaded into Python using the method from PyFMI.

```
#Load the corresponding FMUs
model_chassi    = load_fmu("Chassis.fmu")
model_wheel_lf = load_fmu("TyreForcesSlick.fmu")
model_wheel_lb = load_fmu("TyreForcesSlick.fmu")
model_wheel_rf = load_fmu("TyreForcesSlick.fmu")
model_wheel_rb = load_fmu("TyreForcesSlick.fmu")
```

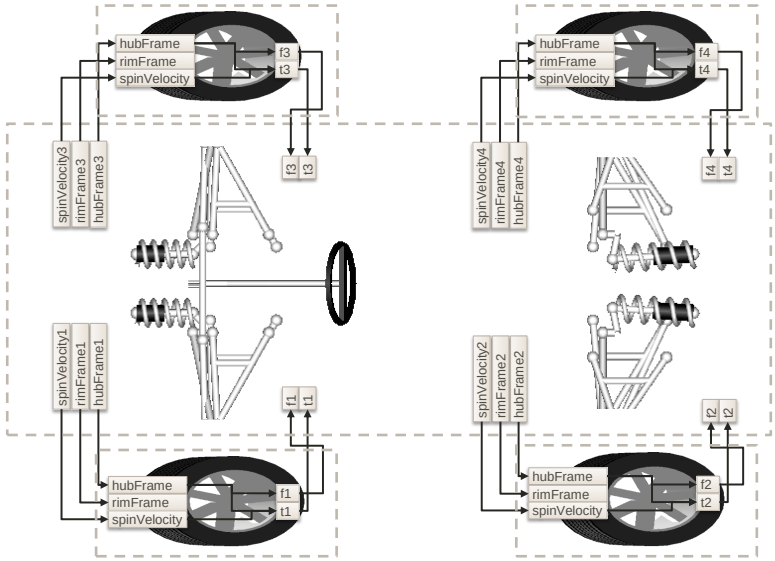

Figure A.2: *Visualization of the couplings in the race car from Section A.4 in a co-simulation setup where the wheel and chassis have been divided into separate models. © Modelon.*

Next, we list the models that are in the coupled system.

```
#Define a list of loaded FMUs
models = [model_chassi, model_wheel_lf, model_wheel_lb,
                        model_wheel_rf, model_wheel_rb]
```

The couplings between the systems are specified next.

```
#Specify the connections
connections = []
for i,wheel_number in enumerate([1,2,3,4]):
    connections.extend(
        [(models[i+1], out, model_chassi,
         out.replace("1","%d"%wheel_number,1))
         for out in models[i+1].get_output_list().keys()])
    for out in model_wheel_lf.get_input_list().keys():
        if out != "spinVelocity":
            connections.append(
    (model_chassi, out.replace(".", "%d."%wheel_number, 1),
     models[i+1], out))
        else:
            connections.append(
    (model_chassi, "spinVelocity%d"%wheel_number, models[i+1], out))
```

Before the simulation starts, we specify the initial position of the steering in the chassis.

```
#Specify start steering
model_chassi.set("steeringInEight.left_turn", 1)
```

In order to create the master algorithm object, the models and their connections need to be provided.

```
#Create the Master simulator
master_simulator = Master(models, connections)
```

To control the simulation, options need to be specified. Here we specify a few options, such as the step size.

```
#Specify the simulation options
master_options = master_simulator.simulate_options()
master_options["local_rtol"] = 1e-9 #Local tolerance in the models
master_options["step_size"]  = 0.01 #Global step size
master_options["linear_correction"]    = False #Correction
master_options["extrapolation_order"] = 0     #Order
master_options["smooth_coupling"]     = False #Smoothing
master_options["execution"] = "serial" #or parallel
master_options["num_threads"] = 2 #Set the number of threads (if
    parallel)
master_options["block_initialization"] = True
master_options["filter"] = {model_chassi:"*summary*" ,
                            model_wheel_lf: "forces.f_*",
                            model_wheel_lb: "forces.f_*",
                            model_wheel_rf: "forces.f_*",
                            model_wheel_rb: "forces.f_*"}
```

Finally, we are ready to simulate the coupled system by invoking the simulation method.

```
res = master_simulator.simulate(final_time=25,
                        options = master_options)
```

The computed solution is stored in the `res` object and the individual trajectory for the variables are retrieved using the Python directory syntax.

```
t = res[model_chassi]["time"]
stearing_wheel = res[model_chassi]["chassis.summary_p_sw"]
p_x = res[model_chassi]["chassis.summary_r_0[1]"]
p_y = res[model_chassi]["chassis.summary_r_0[2]"]
```

Now, these trajectories can be plotted by using for instance the package matplotlib.

```
import pylab as plt
plt.rcParams["lines.linewidth"] = 2
plt.rcParams["font.size"] = 18
plt.rc('legend',**{'fontsize':16})
plt.figure()
plt.plot(t, stearing_wheel, '--', label="Steering wheel")
plt.legend()
plt.grid()
plt.xlabel("Time [s]");plt.ylabel("Angle [rad]")
plt.figure()
```

```
plt.plot(p_x,p_y, '--', label="Car position")
plt.xlabel("Position (x-dir.)");plt.ylabel("Position (y-dir.)")
plt.grid()
plt.legend()
plt.show()
```

A.5 Chromatography separation process

The Python script used for adding the Lyapunov equations in Section 9.3.5 to a simulation using PyFMI.

```
import numpy as np
import scipy.sparse as sp
from assimulo.problem import Explicit_Problem

class AppendedODEs(Explicit_Problem):

    def __init__(self, model):

        assert model.get_version() == "2.0" #Assert the FMI version
            is 2.0
        assert model.get_capability_flags()["
            providesDirectionalDerivatives"] == True #Assert
            directional derivatives are provided

        self._model = model
        self.setup()

        self._res = []
        self._res_C = []
        self._res_CPCT = []
        self._order = "F"
        self._sparse_representation = True

        self.f_nbr = self._nbr_states*self._nbr_states
        self.y0 = np.zeros(self.f_nbr)

        [derv_state_dep, derv_input_dep] = model.
            get_derivatives_dependencies()
        self.jac_nnz = 2*self._nbr_states*np.sum([len(derv_state_dep
            [key]) for key in derv_state_dep.keys()])+self.
            _nbr_states*self._nbr_states

    def get_size(self):
        return self.f_nbr

    def setup(self):
        self._nbr_states = len(self._model.get_states_list())
        self._nbr_inputs = len(self._model.get_input_list())
```

```
    #User defined extra right-hand-side
    def rhs(self, P):
        #A = df/dx, B = df/du
        A,B,C,D = self._model.get_state_space_representation(C=False
            , D=False)
        A = A.toarray(order=self._order)
        B = B.toarray(order=self._order)

        P = P.reshape(self._nbr_states,self._nbr_states, order=self.
            _order)

        #dP = A P + P A^T + B B^T
        dP = A.dot(P)+P.dot(A.transpose())+B.dot(B.transpose())

        return dP.flatten(order=self._order)

    def jac(self, P):

        [A,B,C,D] = self._model.get_state_space_representation(A=
            True, B=False, C=False, D=False)

        data = []
        row_ind = []
        col_ind = []

        Aco = A.tocoo()
        AjFull = [A[:,j] for j in range(self._nbr_states)]

        for i in range(self._nbr_states):
            data.extend(Aco.data)
            row_ind.extend(i*self._nbr_states+Aco.row)
            col_ind.extend(i*self._nbr_states+Aco.col)

            col_ind_i = range(i*self._nbr_states,(i+1)*self.
                _nbr_states)

            for j,val in enumerate(AjFull[i].data):
                data.extend([val]*self._nbr_states)
                row_ind.extend(range(AjFull[i].indices[j]*self.
                    _nbr_states,(AjFull[i].indices[j]+1)*self.
                    _nbr_states))
                col_ind.extend(col_ind_i)

        PJac = sp.coo_matrix((data, (row_ind, col_ind)))

        return PJac

    #User defined handle result for the extra equations
    def handle_result(self, export, P):
```

```
            [A,B,C,D] = self._model.get_state_space_representation(A=
                False, B=False, C=True, D=False)
            C = C.toarray(order=self._order)

            P = P.reshape(self._nbr_states,self._nbr_states, order=self.
                _order)
            self._res_CPCT.append(np.dot(np.dot(C,P),np.transpose(C)).
                flatten(order=self._order))
```

Bibliography

[1] A. Abel, T. Blochwitz, A. Eichberger, P. Hamann, and U. Rein. Functional mock-up interface in mechatronic gearshift simulation for commercial vehicles. In *Proc. 9th Int. Modelica Conf.* LiU E-Press, 2012.

[2] C. Andersson, J. Åkesson, C. Führer, and M. Gäfvert. Import and export of functional mock-up units in jmodelica.org. In *Proc. 8th Int. Modelica Conf.*, pages 329–338. LiU E-Press, 2011.

[3] C. Andersson, J. Andreasson, C. Führer, and J. Åkesson. A workbench for multibody systems ode and dae solvers. In *2nd Jnt. Int. Conf. Multibody Syst. Dyn.*, 2012.

[4] C. Andersson, C. Führer, and J. Åkesson. Assimulo: A unified framework for ODE solvers. *Math. Comput. Simul.*, 116:26 – 43, 2015.

[5] C. Andersson, S. Gedda, J. Åkesson, and S. Diehl. Derivative-free parameter optimization of functional mock-up units. In *Proc. 9th Int. Modelica Conf*, pages 819–828. LiU E-Press, 2012.

[6] N. Andersson, P.-O. Larsson, J. Åkesson, S. Skålén, N. Carlsson, and B. Nilsson. Parameter estimation of dynamic grade transitions in a polyethylene plant. In *Symposium on Computer Aided Process Engineering*, volume 17, page 20, 2012.

[7] J. Andreasson and M. Gäfvert. The vehicledynamics library - overview and application. In *Proc. 5th Int. Modelica Conf.* Modelica Association, 2006.

[8] J. F. Andrus. Numerical solution of systems of ordinary differential equations separated into subsystems. *SIAM J. Numer. Anal.*, 16(4):pp. 605–611, 1979.

[9] M. Arnold. *Multi-rate time integration for large scale multibody system models.*, volume 1 of *Solid Mechanics and its Applications*. Springer Netherlands, Martin Luther University Halle-Wittenberg, Institute of Mathematics, 2007.

[10] M. Arnold. Stability of sequential modular time integration methods for coupled multibody system models. *J. Comput. Nonlinear Dynam.*, 5(3):031003, 2010.

[11] M. Arnold, C. Clauss, and T. Schierz. Numerical aspects of fmi for model exchange and co-simulation v2.0. In *2nd Jnt. Int. Conf. Multibody Syst. Dyn.*, 2012.

[12] M. Arnold and M. Günther. Preconditioned dynamic iteration for coupled differential-algebraic systems. *BIT Numer. Math.*, 41(1):1–25, 2001.

[13] J. H. Baayen. Vortexje - an open-source panel method for co-simulation. *CoRR*, abs/1210.6956, 2012.

[14] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith. Cython: The best of both worlds. *IEEE Comput. Sci. Eng.*, 13(2):31 –39, March 2011.

[15] T. Blochwitz et al. The functional mockup interface for tool independent exchange of simulation models. In C. Clauss, editor, *Proc. 8th Int. Modelica Conf*, pages 105–114. LiU E-Press, Mar. 2011.

[16] R. A. Brualdi and H. J. Ryser. *Combinatorial Matrix Theory*. Cambridge University Press, 1991. Cambridge Books Online.

[17] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer US, 2006.

[18] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse jacobian matrices. *J. Inst. Math. Appl*, 13(1):117–120, 1974.

[19] Dassault Systèmes. Dymola - multi-engineering modeling and simulation - version 2014. `http://www.dymola.com/`, 2013. [accessed: 2014-05-07].

[20] Dassault Systèmes. Dymola - multi-engineering modeling and simulation - version 2016. `http://www.dymola.com/`, 2015. [accessed: 2016-02-03].

[21] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[22] H. Elmqvist and S. Mattsson. Modelica - the next generation modeling language an international design effort. In *Proc. 1th World Congr. Syst. Simul.*, pages 1–3. SCS, 1997.

[23] C. A. Felippa, K. C. Park, and C. Farhat. Partitioned analysis of coupled mechanical systems. *Comput. Methods Appl. Mech. Eng.*, 190(24–25):3247 – 3270, 2001.

[24] E. Fredriksson, C. Andersson, and J. Åkesson. Discontinuities handled with events in assimulo, a practical approach. In *Proc. 10th Int. Modelica Conf.* LiU E-Press, 2014.

[25] Free Software Foundation. GNU Lesser General Public License, version 3. `http://www.gnu.org/licenses/lgpl.html`, 2007. [accessed: 2014-05-07].

[26] D. Fritzson, J. Ståhl, and I. Nakhimovski. Transmission line co-simulation of rolling bearing applications. In *Proc. 48th Conf. Simul. Model."*, 2007.

[27] C. Führer and B. J. Leimkuhler. Numerical solution of differential-algebraic equations for constrained mechanical motion. *Numer. Math.*, 59:55–69, 1991.

[28] C. W. Gear and D. R. Wells. Multirate linear multistep methods. *BIT Numer. Math.*, 24:484–502, 1984.

[29] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations. I. Nonstiff Problems*. Springer Ser. Comput. Math. Springer-Verlag, Berlin, 1991.

[30] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations. II. Stiff and Differential-Algebraic Problems*. Springer Ser. Comput. Math. Springer-Verlag, Berlin, 2nd edition, 1996.

[31] A. C. Hindmarsh. *ODEPACK, a systematized collection of ODE solvers*. Lawrence Livermore National Laboratory, 1982.

[32] A. C. Hindmarsh. Toward a systematized collection of ODE solvers. In *10th IMACS World Congr. Syst. Simul. Sci. Comput.*, pages 427 – 429. ESP, 1982.

[33] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, Sept. 2005.

[34] G. Hippmann, M. Arnold, and M. Schittenhelm. Efficient simulation of bush and roller chain drives. In *Proc. Multibody Dyn., ECCOMAS Conf.*, 2005.

[35] A. Holmqvist, C. Andersson, F. Magnusson, and J. Åkesson. Methods and tools for robust optimal control of batch chromatographic separation processes. *Processes*, 3(3):568, 2015.

[36] P. Horvath, M. Yampolskiy, Y. Xue, X. D. Koutsoukos, and J. Sztipanovits. An integrated system simulation approach for wireless networked control systems. In *5th ISRCS*, pages 118–123. IEEE, 2012.

[37] J. D. Hunter. Matplotlib: A 2d graphics environment. *Comput. Sci. Eng.*, 9(3):90–95, 2007.

[38] ITI GmbH. SimulationX - multi-domain system simulation and modeling. `http://www.itisim.com/`, 2016. [accessed 2016-03-18].

[39] K. H. Johansson. The quadruple-tank process: a multivariable laboratory process with an adjustable zero. *IEEE Trans. Contr. Syst. Technol.*, 8(3):456–465, May 2000.

[40] R. Kübler and W. Schiehlen. Modular simulation in multibody system dynamics. *Multibody Syst. Dyn.*, 4:107–127, 2000.

[41] R. Kübler and W. Schiehlen. Two methods of simulator coupling. *Math. Comp. Model. Dyn.*, 6(2):93–113, 2000.

[42] A. Kværnø. *Runge-Kutta Methods for the Numerical Solution of Differential-Algebraic Equations of Index 1*. PhD thesis, NTH, Trondheim, Norway, 1988.

[43] R. I. Leine, D. H. Van Campen, and C. H. Glocker. Nonlinear Dynamics and Modeling of Various Wooden Toys with Impact and Friction. *J. Vib. Control*, 9(1-2):25, 2003.

[44] C. Lubich, U. Nowak, U. Pohle, and C. Engstler. *MEXX-numerical software for the integration of constrained mechanical multibody systems*. ZIB, Berlin, 1992.

[45] MATLAB. *Version - R2012b*. The MathWorks Inc., Natick, Massachusetts, 2012.

[46] S. E. Mattsson, H. Elmqvist, and J. F. Broenink. Modelica: An international effort to design the next generation modelling language. In *Spec. CACSD*, pages 16–19, 1997.

[47] Modelica Association. Modelica Standard Library - Version 3.2.1 (Build 2). `http://modelica.github.io/Modelica/help/Modelica.html`, 2013. [accessed: 2014-05-07].

[48] MODELISAR. Functional mock-up interface for co-simulation. Interface specification, MODELISAR, October 2010.

[49] MODELISAR. Functional mock-up interface for model exchange. Interface specification, MODELISAR, January 2010.

[50] MODELISAR. Functional Mock-up Interface, Version 2.0. Interface specification, MODELISAR, July 2014.

[51] Modelon AB. FMI Library. `http://www.jmodelica.org/FMILibrary`, 2014. [accessed 2016-02-03].

[52] Modelon AB. FMI Toolbox for MATLAB - Version 1.7.2. `http://www.modelon.com/products/fmi-toolbox-for-matlab`, 2014. [accessed: 2014-05-07].

[53] Modelon AB. OPTIMICA Compiler Toolkit. `http://www.modelon.com/products/optimica-compiler-toolkit/`, 2016. [accessed: 2016-03-29].

[54] J. A. Nelder and R. Mead. A simplex method for function minimization. *Comput. J.*, 7(4):308–313, 1965.

[55] T. E. Oliphant. Python for scientific computing. *IEEE Comput. Sci. Eng.*, 9(3):10–20, May 2007.

[56] M. Otter. *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter*. PhD thesis, Fakultät für Maschinenbau der Ruhr-Universität Bochum, Bochum, Germany, 1995.

[57] P. Pannu, C. Andersson, C. Führer, and J. Åkesson. Coupling model exchange fmus for aggregated simulation by open source tools. In *Proc. 11th Int. Modelica Conf.*, pages 903–909. LiU E-Press, 2015.

[58] P. Peterson. F2py: a tool for connecting fortran and python programs. *Int. J. Comput. Sci. Eng.*, 4(4):296–305, Nov. 2009.

[59] L. R. Petzold. Description of DASSL: a differential/algebraic system solver. Technical report, Applied Mathematics Division, Sandia National Laboratories, September 1982.

[60] A. Pfeiffer, M. Hellerer, S. Hartweg, M. Otter, and M. Reiner. Pysimulator - a simulation and analysis environment in python with plugin infrastructure. In *Proc. 9th Int. Modelica Conf*, pages 523–536. LiU E-Press, 2012.

[61] P. J. Rabier and W. C. Rheinboldt. *Nonholonomic Motion of Rigid Mechanical Systems from a DAE Viewpoint*. SIAM, Philadelphia, PA, USA, 2000.

[62] J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. 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.

[63] T. Schierz. *Modulare Zeitintegration gekoppelter Differentialgleichungssysteme in der technischen Simulation*. PhD thesis, Martin Luther Universität Halle Wittenberg, Germany, May 2013.

[64] T. Schierz and M. Arnold. Stabilized overlapping modular time integration of coupled differential-algebraic equations. *Appl. Numer. Math.*, 62(10):1491–1502, Oct. 2012.

[65] T. Schierz, M. Arnold, and C. Clauss. Co-simulation with communication step size control in an fmi compatible master algorithm. In *Proc. 9th Int. Modelica Conf*, pages 205–214. LiU E-Press, 2012.

[66] B. Schweizer and D. Lu. Predictor/corrector co-simulation approaches for solver coupling with algebraic constraints. *ZAMM*, 95(9):911–938, 2015.

[67] SIMPACK GmbH. SIMPACK - multi-body simulation software. `http://www.simpack.com/`, 2016. [accessed 2016-03-18].

[68] A. Sjö. *Analysis of Computational Algorithms for Linear Multistep Methods*. PhD thesis, Lund University, Lund, Sweden, 1999.

[69] G. Söderlind. *DASP3*. KTH, Stockholm, 1980.

[70] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[71] A. Viel. Implementing stabilized co-simulation of strongly coupled systems using the functional mock-up interface 2.0. In *Proc. 10th Int. Modelica Conf.* LiU E-Press, 2014.

[72] S. Voigtmann. *General Linear Methods for Integrated Circuit Design*. Logos Verlag Berlin, 2006.

[73] R. A. Wehage and E. J. Haug. Generalized coordinate partitioning for dimension reduction in analysis of constrained dynamic systems. *J. Mech. Des.*, 104(1):247–255, 1982.

[74] Xogeny. FMQ. `http://www.xogeny.com/products/`, 2016. [accessed: 2016-03-18].