A Software Framework for Implementation and Evaluation of Co-Simulation Algorithms

Andersson, Christian

2013

A Software Framework for Implementation and Evaluation of Co-Simulation Algorithms

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/

Licentiate Theses in Mathematical Sciences 2013:1
ISSN 1404-028X

ISBN 978-91-7473-671-7
LUTFNA-2003-2013

Printed in Sweden by Media-Tryck, Lund 2013

Abstract

In this thesis, simulation of coupled dynamic models, denoted sub-systems, is analyzed and described in a co-simulation context. This means that the respective coupled systems contain their own internal integrator, hidden from the coupling interface. Co-Simulation is an interesting and active research field where industry is a driving force. The problems where co-simulation is an interesting approach is two-fold. On one-hand, there is the coupling of sub-systems between tools. Consider the case where tools use different representation of the sub-systems and the problem presented by the coupling of the two. On the other hand, there is the performance issue. There is a potential performance increase for the overall system simulation when using a tailored integrator for each sub-system compared to using a general integrator for the monolithic system. The aim of this thesis is to develop a testing framework for currently used co-simulation approaches and to describe the state of the art in co-simulation. Additionally, the aim is to be able to test the approaches on industrially relevant models and academic test models.

Using co-simulation for simulation of coupled systems may result in stability problems depending on the approach used, and the intention here is to describe when it occurs and how to handle it. The commonly used methods use fixed step-size for determining when information between the models are to be exchanged. A recent development for co-simulation of coupled systems using a variable step-size method is described together with the requirements for performing such a simulation.

Attaining the goals of the thesis has required a substantial effort in software development to create a foundation in terms of a testing framework. For gaining access to models from industry, the newly defined *Functional Mock-up Interface* has been used and a tool for working with these type of

models, called PyFMI, has been developed. Another part is access to integrators, necessary for evaluating the impact of the internal integrator in the sub-systems. A tool providing a unified high-level interface to various integrators has been developed and is called Assimulo. The key component is the algorithm for performing the co-simulation. It has been developed to be easily extensible and to support the currently used co-simulation methods.

The developed framework has been proven to be successful in evaluating co-simulation approaches on both academic test examples and on industrially relevant models as will be shown in the thesis.

Popular Scientific Description

In almost every field of science such as engineering, physics and economics there are processes that can be modeled mathematically. By a mathematical model of the process, further insight and a deeper understanding of the behavior of a process can be given. In the industry there are economic incentives for developing a mathematical model of a process and perform the evaluation of the design virtually instead of building prototypes. Modifying a virtual model is substantially easier and more cost effective than modifying a prototype.

Consider a ball that is dropped from a height h_0 under the influence of the gravity g. This system can be described by the single equation,

$$\ddot{h} = -g, \quad h(t_0) = h_0, \quad \dot{h}(t_0) = 0 \tag{1}$$

where $\ddot{h}$ is the acceleration of the ball. This type of equation is called an *Ordinary Differential Equation*. From this equation it is possible to calculate the height of the ball at a given time. It is even possible to solve the equation analytically,

$$h(t) = h_0 - \frac{gt^2}{2}. \tag{2}$$

Unfortunately, there are very few models that can be solved analytically leaving the only option to compute an approximation of the model solution using algorithms from scientific computing. For the industry, it is tremendously important to have access to reliable algorithms which can be used to calculate solutions for their models and to be able to trust the result.

As the models grow in complexity and as multiple domains need to be combined in the same model, problems arise. In a vehicle model there can be electrical parts together with mechanical parts which are typically modeled in different modeling software. To understand the vehicle properties these two parts need to be coupled together and evaluated as a single unit.

Due to restrictions posed by modeling programs or due to the fact that specialized algorithms are used, it may not always be possible to access the model equations directly. Instead, it may only be possible to set inputs to a model and after a certain time retrieve the outputs. In a co-simulation approach, this is exactly the case, a complex model that is comprised by a number of components without exposed equations. The question asked is how to calculate the solution of the coupled complex models, and how to develop reliable algorithms for which the result can be trusted.

Contents

Chapter 1

Introduction

Designing large complex engineering system models is difficult and the demand for these high fidelity models is ever increasing. A representative system is that of a vehicle, which involves several engineering domains. Typically a vehicle model involves mechanics for the dynamic behavior of the chassis and wheels, electrical components for the engine and the safety systems (brakes etc.) and also thermodynamics for the air conditioning.

The traditional approach where the component models were evaluated and tested separately is no longer a valid option for the complex system models. Instead, the fully coupled system model needs to be simulated and evaluated as a single unit in order to take into account the interaction between the components and fully investigate the complete dynamic behavior.

There is a strong tradition among domain experts, guided also by the availability, to use specialized modeling and simulation environments for component models. These tools are also favored due to that they usually offer larger libraries of components and features specific to the domain compared to a multi-domain tool. Problems arise when trying to investigate a coupled system model, which means that component models from various tools needs to be coupled which is usually problematic 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 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.

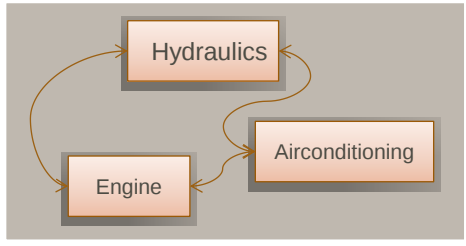

Figure 1.1: *A schematic figure of a multi-domain coupled system.*

Coupling the components together into a monolithic model can be performed via two different approaches, *strong*-coupling and *weak*-coupling. Given that the model exposes its internal dynamics, i.e. that it is possible to directly evaluate the model equations, one can straightforward assemble these equations which can in turn be solved by standard time integration algorithms. This approach is commonly called 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, see Figure 1.2, the strong-coupling is no longer possible. In these cases one resorts to the weak-coupling where the separate models contain an internal integrator and the information exchange, via inputs and outputs, to the connected models is only performed at specified communication points. This approach has several benefits as it allows tailored integrators for component models, and also it allows for them to be run in parallel. Moreover, the component models may have widely different time scales which can be exploited by the internal integrator. This is evident by comparing electrical components to multibody components. However, it also introduces difficulties in how the information exchange should be performed in order for a stable simulation of the weakly-coupled system.

In recent years, a new standard for exchanging dynamic system models between modeling and simulation tools has been developed. The standard was developed as part of the MODELISAR project with participants both from industry with tool vendors and users, and from academia. The standard is called the *Functional Mock-up Interface (FMI)*. The standard fills

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

the gap where before there has been costly custom solutions for coupling specific simulation environments, which was highlighted at the recent SAE World Congress [35] . What is particularly exciting is the amount of attention the standard has received and the number of vendors that has adopted the standard and implemented support for either importing models or exporting models.

This thesis discusses simulation of *weakly*-coupled systems, i.e. co–simulation, and is outlined as follows. In Chapter 1, an introduction to co-simulation is given together with an overview of previous research on co-simulation. In Chapter 2, the functional mock-up interface is introduced which serves as a basis for the discussion of co-simulation. Chapter 3 discusses co-simulation in more detail with regard to the stability requirements, i.e. what are the requirements on the algorithm and/or the model for a performing a stable simulation? Moreover, error estimation in a co-simulation context allowing for a variable step-size algorithm is discussed. Chapter 4 focus on the developed software and the developed software capabilities which is the main focus of the thesis. The software is a key contribution and necessary in order to evaluate, experiment and verify different co-simulation approaches. By supporting various approaches within the same environment, a fair comparison is possible. The chapter presents the ASSIMULO package for solving ordinary differential equations. PyFMI for working with models following the FMI and finally the master algorithm for simulating weakly-coupled systems. In Chapter 5, the developed software capabilities is shown on examples. Chapter 6 summarizes the thesis and presents future work.

1.1 What is Co-Simulation?

Co-Simulation is about how to simulate two or more dynamic systems which are connected. The systems are described as 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 separates 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 sub-system, $T_n = t_{n,0} < t_{n,1} < \ldots < t_{n,m} = T_{n+1}$.

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

$$\dot{z} = Az \tag{1.1}$$

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

$$\dot{x}^{[1]} = a_{11}x^{[1]} + a_{12}u^{[1]} \tag{1.2a}$$

$$y^{[1]} = x^{[1]} \tag{1.2b}$$

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

$$\dot{x}^{[2]} = a_{22}x^{[2]} + a_{21}u^{[2]} \tag{1.3a}$$

$$y^{[2]} = x^{[2]}. \tag{1.3b}$$

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 sub-systems are done through the inputs and the outputs.

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

$$u^{[1]} = y^{[2]} \tag{1.4a}$$

$$u^{[2]} = y^{[1]}. \tag{1.4b}$$

These coupling equations are in a sense "outside" the individual sub-systems. The question is now how the decoupling impact the result and the stability

of the overall system? Also, how should the exchange of information between the two systems be performed? By extrapolating the inputs using an appropriate polynomial? By introducing an ordering so that some signals are interpolated while some are extrapolated? Or something else entirely?

The most commonly used co-simulation method is to let the individual models run in parallel and at predefined global time points exchange information. The inputs in between the global steps are kept constant, see Figure 1.3.

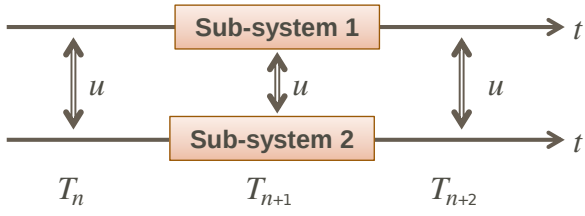

Figure 1.3: *Schematic figure of two sub-systems that are run in parallel with data exchange at predefined global time points.*

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*.

More generally we consider N coupled systems of the type,

$$\dot{x}^{[i]} = f^{[i]}(t, x^{[i]}, u^{[i]}), \quad i = 1, \ldots, N \tag{1.5a}$$

$$y^{[i]} = g^{[i]}(t, x^{[i]}, u^{[i]}), \quad i = 1, \ldots, N \tag{1.5b}$$

$$u = c(y) \tag{1.5c}$$

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

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

1.2 Previous Work

Dividing a set of equations into sub-systems and solving them separately has been discussed for decades and dates back to the late 1970s where Andrus [8] discussed solution of a set of equations divided into sub-systems of "fast" components and "slow" components. The discussion centered around the benefits and performance gains of using different time scales in the integrator for the components. These type of methods are called multi-rate and a lot of research has been performed in this area. Gear and Wells [19], for instance, discussed improvements for automatic step-size selection of multi-rate methods for linear multistep methods.

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

In [15], an overview of co-simulation approaches are discussed. The parallel and staggered scheme are explained together with more sophisticated schemes. The parallel scheme is basically to let the sub-systems simulate the same global time-step, once all sub-systems 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 sub-system is solved for a global time-step. Once completed, the next sub-system is simulated over the same global time-step.

In [30, 31], 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. A block interacted with another block via its inputs and via its outputs through coupling equations outside of the blocks. The blocks where represented in a general state-space formulation, see Equation 1.5a and 1.5b, which is widely used in control theory. The coupling equations, Equation 1.5c, was in their case assumed to be linear, $u = Ly$. The articles centered around stability issues and covered the cases where there is direct feed-through and when there is not. Constraints on the feed-through was highlighted in order to guarantee a stable simulation of the coupled system. The articles by Kübler et al later served as a foundation for the definition of the FMI standard for co-simulation [1].

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 [4], it was shown that the multi-domain environment SimulationX [27] and the multibody environment SIMPACK, [5], can be coupled together using FMI components. The application was to simulate a power-train of a heavy-duty truck where parts was 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 [25] where a model of an engine is decoupled into sub-systems. By decoupling the chain drive into a sub-system, an decrease of the simulation time by an order of magnitude was achieved.

Research on the stability issues when using co-simulation has been active in recent years. In [11], Arnold et al discussed stability of coupled differential algebraic equations and formulated a contractivity condition that must be fulfilled in order to guarantee a stable error propagation. Stabilization of these systems was further discussed in [9, 10] where the Jacobian information was utilized for performing a stable integration. Another technique proposed in [40] was based on applying the constraint equations in a differential algebraic equation to more than one sub-system.

The commonly used approach 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 [41, 39], an error estimation procedure for coupled systems was proposed. The error estimate was based on Richardson extrapolation and the assumption that the sub-systems were integrated exactly. In an engineering setting, this can be achieved by requiring higher accuracy on the sub-systems. 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 sub-systems in order to decouple the problem. The delays that are introduced changes the models and introduces errors, but on the other hand, this delay can usually be mo-

tivated by physics and the error can be dealt with explicitly. In this thesis TLM will not be covered, see [17].

1.3 Contributions

The contribution of this thesis are mainly that of building a solid foundation with a developed software framework for experimenting and evaluating co-simulation techniques. The software framework consists of three connected programs that has been developed. The programs are,

- *PyFMI*
- ASSIMULO
- *The Master Algorithm*

PyFMI, which is a joint project at Modelon AB based on the freely available FMI Library [3] and ASSIMULO are necessary tools for the evaluation of co-simulation approaches. The first give access to models following the FMI standard and thus models from a number of different tools. The second give access to ODE solvers which can be used together with FMUs following the model exchange standard to mimic a co-simulation FMU. This allows for the evaluation of various ODE solvers in a co-simulation setting. The master algorithm is the final tool that connects the mentioned programs and is responsible for performing the actual co-simulation. The framework developed will serve as an experimentation platform for further theoretical research.

Apart from the software framework, co-simulation in general with focus on stability, integration restart and error estimation is discussed. Specific contributions,

- *Analysis of the stability of the parallel co-simulation approach.*
- *Analysis of the requirements for an integration restart of linear multistep method.*
- *Analysis of an error estimation procedure.*

1.3.1 Publications

The thesis is based on the following publications.

- **C. Andersson**, J. Andreasson, C. Führer and J. Åkesson. "A Workbench for Multibody Systems ODE and DAE Solvers". *The Second Joint International Conference on Multibody System Dynamics (2012).*
- **C. Andersson**, J. Åkesson, C. Führer and M. Gäfvert. "Import and Export of Functional Mock-up Units in JModelica.org". *In 8th International Modelica Conference, 2011".*

For the first publication, the author has contributed with software implementation of the workbench, the evaluation on test models together with being primary responsible for drafting the manuscript. For the second publication, the author has contributed with software implementation for the import of functional mock-up units and the evaluation of the same together with being primary responsible for drafting the manuscript.

Other publications by the author.

- P. Grover and **C. Andersson**. "Optimized three-body gravity assists and manifold transfers in end-to-end lunar mission design". *22nd AAS/AIAA Space Flight Mechanics Meeting 2012.*
- S. Gedda, **C. Andersson**, J. Åkesson and S. Diehl. "Derivative-free Parameter Optimization of Functional Mock-up Units". *In 9th International Modelica Conference, 2012*

For the first publication, the author has contributed with software implementation of optimization methods, modeling and calculation of the final segment of the trajectory into the moon orbit. The author additionally aided in writing of the manuscript. For the second publication, the author has assisted in improving on the software implementation and in drafting the manuscript.

Chapter 2

The Functional Mock-up Interface

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. FMI is a standard to provide a unified model execution interface for exchanging dynamic system models between modeling tools and simulation tools. The idea is that tools generate and exchange models that adheres to the FMI specification. Such models are called Functional Mock-up Units (FMUs), see Figure 2.1. 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.

The generated models, FMUs, are distributed and shared as compressed archives. They include either or both 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. The archive additionally contains 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. There can also be additional information in the archive, which does not impact a simulation of the model, but which may be of interest to distribute with the FMU, for example documentation.

FMI was developed in a European project, MODELISAR, focused on

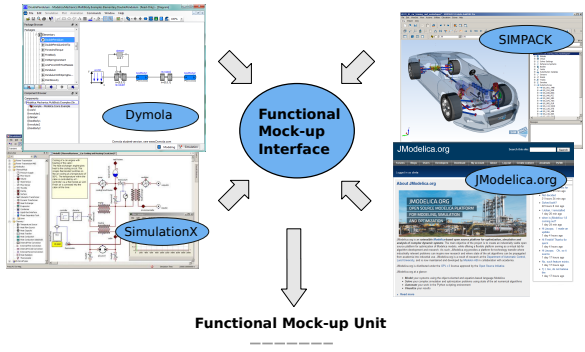

Figure 2.1: *Export of Functional Mock-up Units.*

improving the design of systems and of embedded software in vehicles. The standard is now maintained and developed by the Modelica Association.

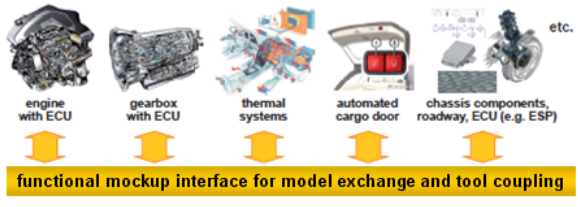

Figure 2.2: *Functional Mock-up Interface*[1].

2.1 FMI 1.0

FMI 1.0 consists of two specifications, one for model exchange [2] and one for co-simulation [1].

2.1.0.1 Model Exchange

For model exchange, the standard describes an interface for discontinuous ordinary differential equations, with means to set the continuous states and

[1] ©Modelica Association

time as well as evaluating the model equations, i.e. the right-hand-side, and specifying inputs.

The standard describes a model as,

$$\dot{x} = f(t, x, u; d) \tag{2.1}$$

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

- *State Events*
  These events are dependent on the state solution profiles and thus not known a prior. The model provides a set of event indicators that the integrator monitors during the integration process. If one of the event indicators switches domain, there is a state event. The integrator is then responsible for finding the time when the event occurred.

- *Time Events*
  These events on the other hand are known a prior, meaning that for each simulation segment it is known when the time event occur and thus this time is set as the simulation end time for that segment. Typically it can be that after a certain elapsed time in the integration, a force is applied on the model.

- *Step Events*
  These last type of events are events that typically do not influence the model behavior, instead they are events to ease the numerical integration. For instance it can be a change of the continuous states in the model as the current states are no longer appropriate numerically.

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

2.1.0.2 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 time T_n of the model, return the outputs, y_{n+1}, at a time $T_n + H = T_{n+1}$.

$$y_{n+1} = \Phi(T_n, u_n) \tag{2.2}$$

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

The advancement of the states and time are completely hidden for the master algorithm and is also not specified by the standard, see Figure 2.4. This

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

allows for specialized solvers to be used for the particular sub-system at hand which may give an increased performance and a more stable simulation. In FMI, advancing the solution to the next communcation point is performed using the `do_step` method. At a communcation point, values inside the model can be retrieved and inputs be set. There is additionally a capability flag that determines if higher order derivatives may be set for the inputs. These are represented by a vector,

$$\left[\frac{du}{dt}(T_n), \frac{d^2u}{dt^2}(T_n), \frac{d^3u}{dt^3}(T_n), \ldots, \frac{d^ku}{dt^k}(T_n)\right] \tag{2.3}$$

and the input is evaluated during the next global integration step as,

$$u(t) = u(T_n) + \sum_{i=1}^{k} \frac{1}{i!}\frac{d^iu}{dt^i}(t - T_n)^i, \quad t \in [T_n, T_{n+1}]. \tag{2.4}$$

2.1.1 Reception

FMI 1.0 for model exchange was the first release of the standard and was released in January 2010 and has since then received a significant amount of

attention among vendors and users. There are currently 34 tools that support or plan to support FMI. Examples include the commercial products Dymola [42] and Simpack [5] as well as the open-source platform JModelica.org [37]. 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.

2.2 FMI 2.0

The demand for accomplishing more with the standard triggered the development of FMI 2.0. One feature requested was the ability to get the directional derivatives and another an improved interface for restoring the model state completely to a previous time point. Both of these features are included in FMI 2.0.

The added feature of providing the directional derivatives give the user the ability to calculate, for model exchange, the directional derivatives at a specific time point. For co-simulation it give the user the ability to calculate the directional derivatives at a specific communication point. For model exchange it is particularly useful when using an implicit method for simulating the FMU. Implicit methods require the Jacobian in order to advance the solution,

$$J = \frac{\partial f}{\partial x}. \tag{2.5}$$

The Jacobian can be approximated using finite differences, however, the ability to provide an analytical or one generated by automatic differentiation may give an increased performance. For co-simulation, understanding how, if any, the direct feed-through terms influence the outputs is important. Using the directional derivatives, we can calculate the partial derivative of the outputs, g, with respect to the inputs and use this information in a master algorithm,

$$D = \frac{\partial g(t, x, u)}{\partial u}. \tag{2.6}$$

The directional derivatives is an optional feature in the standard.

The other important change to the standard is the ability to store a complete model state, for co-simulation it includes the internal solver, and the ability to restore a previous stored state. This is useful for both model exchange and co-simulation. In co-simulation there is a demand for variable

step-size algorithms when simulating coupled co-simulation FMUs. In order to adapt the step-size, an error estimate is needed which requires in one way or another that the same global step is performed more than once. This in turn requires that we are able to go back in time and with this feature, it is possible. Another use-case, for both model exchange and co-simulation, is when a model contains a transient requiring a substantial amount of work and the interest is after the transient has settled. In this case, the model can be simulated past the transient once and the model state be stored. For each experiment, the model is simply restored to the stored state.

The specification for co-simulation and model exchange is also intended to be merged into a uniformed specification. This will simplify both the implementation for export of FMUs and also import of FMUs. The type of the FMU is instead defined by a capability flag in the XML description.

In addition there has been numerous changes and clarifications to the specification for problems and inconsistencies detected by vendors and users alike. An overview of the specification can be found in [12].

Chapter 3

Co-Simulation

This chapter gives an overview of co-simulation and the different approaches used. The intention is to discuss the stability in regards to the parallel approach and to show when and why stability problems may occur when performing co-simulation. The final section is dedicated to discussing error estimation and integration restart which are necessary in order to be able to perform a variable step-size integration of the coupled system.

3.1 Approaches

The classical approaches in co-simulation is the parallel setup, Section 3.1.1, and the serial setup, Section 3.1.2. More complex setups have been discussed, for example in [15], but they will not be considered here.

3.1.1 Parallel

In a parallel approach, the sub-systems are all treated in parallel and the inputs/outputs are extrapolated from the previous global time-steps. A major benefit of this approach is its obvious parallelism where a global time-step is performed simultaneously for all the sub-systems. This approach bears similarities with the Jacobi iteration for linear equations and is also referred to as a *Jacobi-like* approach. In Figure 3.1 an overview of the scheme is shown.

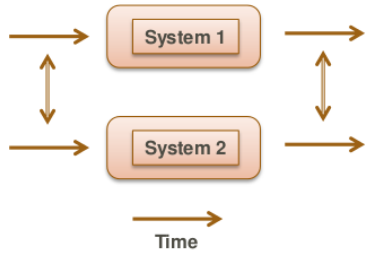

Figure 3.1: *Overview of Jacobi.*

3.1.2 Staggered

In a staggered approach, the sub-systems are ordered and for a global time-step simulated sequentially. If an input to sub-system i belongs to a sub-system that has already been solved for the current global time-step, this input is interpolated. If it does not belong to a sub-system that has already been solved, the input is instead extrapolated from the previous global time-step. A drawback of this approach is that the sub-systems are solved sequentially and additionally the question of how the sub-systems should be ordered arises. However, this approach may give a more stable simulation if the ordering is done in an appropriate way. As the parallel approach bears similarities with the Jacobi iteration, the staggered approach is similar to the Gauss-Seidel iteration for linear equations and is in the same way referred to as a *Gauss-Seidel-like* approach. In Figure 3.2 an overview of the scheme is shown.

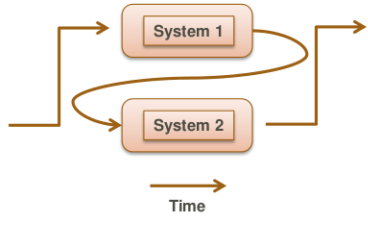

Figure 3.2: *Overview of Gauss-Seidel.*

3.2 Stability Analysis

In the analysis of co-simulation one is interested in how the *coupling* effect the overall simulation with regard to stability and accuracy. In this section we focus on the stability issues that may be influenced, depending on the simulation approach. In the analysis we focus on the Jacobi method for the overall simulation. As it is the coupling that is of interest we assume that the sub-systems are solved exactly and only study how the coupling impacts the simulation. In practice this means that the sub-systems are solved with higher accuracy as 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 known values of the states and outputs,

$$[x_{n+1}, y_{n+1}, \ldots, y_{n+1-k}]^{\mathrm{T}} = \Psi(H)[x_n, y_n, \ldots, y_{n-k}]^{\mathrm{T}}. \tag{3.1}$$

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

$$\rho(\Psi(H)) \leq 1 \tag{3.2}$$

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 3.2 holds at least for $H = 0$. Classically this separates stability of the discretization method from the stability of the problem. This leads to the notion of zero stable methods. All methods we use in this thesis are zero stable but in contrast to the classical case we will see, that in the co-simulation context the problem and in particular the feed-trough terms matter even in the case $H = 0$. So zero stability of a method is not sufficient to guarantee stability for $H = 0$. We have to set up conditions on the feed through term as well. We follow the lines of Kübler [31] in the derivation of the conditions.

In the analysis we consider N coupled linear systems with a linear cou-

pling,

$$\dot{x}^{[i]} = A^{[i]}x^{[i]} + B^{[i]}u^{[i]}, \quad i = 1, \ldots, N \tag{3.3a}$$
$$y^{[i]} = C^{[i]}x^{[i]} + D^{[i]}u^{[i]}, \quad i = 1, \ldots, N \tag{3.3b}$$
$$u = Ly \tag{3.3c}$$

the connection between the system 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 3.3 simplifies to,

$$\dot{x} = Ax + Bu \tag{3.4a}$$
$$y = Cx + Du \tag{3.4b}$$
$$u = Ly \tag{3.4c}$$

The monolithic system can additionally be simplified to

$$\dot{x} = (A + BL(I - DL)^{-1}C)x \tag{3.5}$$

where $(I - DL)$ is assumed to be non-singular so that Equation 3.4 is an index one system with y and u being the algebraic variables.

Now, consider that each sub-system in Equation 3.3a is solved exactly, we get for Equation 3.4a,

$$\Phi(x_n, u_n) = \int_{T_n}^{T_{n+1}} e^{A(T_{n+1}-\tau)}Bu(\tau)d\tau + e^{A(T_{n+1}-T_n)}x(T_n) \tag{3.6}$$

where we have used the definition for the matrix exponential.

Definition 3.2.1 (Matrix exponential)**.** *The exponential of a square matrix is defined as,*

$$e^A = \sum_{i=0}^{\infty} \frac{1}{i!} A^i.$$

The 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,

$$x_{n+1} = \Phi(x_n, u_n) \tag{3.7a}$$
$$y_{n+1} = Cx_{n+1} + Du_{n+1} \tag{3.7b}$$
$$u_{n+1} = Ly_{n+1} \tag{3.7c}$$

Note that this requires that we are able to solve Equation 3.7b and Equation 3.7c together and therefor it is an implicit method. However, in a co-simulation framework this may not always be possible. In the general case,

$$y^{[i]} = g^{[i]}(t, x^{[i]}, u^{[i]}), \quad i = 1, \ldots, N \tag{3.8a}$$
$$u = c(y) \tag{3.8b}$$

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

Going back to Equation 3.7 and by eliminating u we get,

$$x_{n+1} = \Phi(x_n, Ly_n) \tag{3.9a}$$
$$y_{n+1} = (I - DL)^{-1} Cx_{n+1} \tag{3.9b}$$

which is the algorithm we will investigate.

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

$$x(t) = \int_{T_n}^{t} e^{A(t-\tau)} BLy(\tau) d\tau + e^{A(t-T_n)} x(T_n). \tag{3.10}$$

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

$$y(\tau) := y(T_n), \quad \tau \in [T_n, T_{n+1}]. \tag{3.11}$$

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

$$x_{n+1} = \Phi(x_n, Ly_n) \tag{3.12a}$$
$$= \int_{T_n}^{T_{n+1}} e^{A(T_{n+1}-\tau)} d\tau BLy_n + e^{A(T_{n+1}-T_n)} x_n \tag{3.12b}$$
$$= A^{-1}(e^{AH} - I)BLy_n + e^{AH} x_n \tag{3.12c}$$

where H is the global step-size which we assume to be the same for all steps. Together with $y_{n+1} = (I - DL)^{-1} C x_{n+1}$ we finally get,

$$\begin{bmatrix} I & 0 \\ -C & I - DL \end{bmatrix} \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} e^{AH} & A^{-1}(e^{AH} - I)BL \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{3.13}$$

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} e^{AH} & A^{-1}(e^{AH} - I)BL \\ (I - DL)^{-1} C e^{AH} & (I - DL)^{-1} C A^{-1}(e^{AH} - I)BL \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix}. \tag{3.14}$$

Fixing $T = nH$,

$$H \to 0 \quad \Longrightarrow \quad \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} I & 0 \\ (I - DL)^{-1} C & 0 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{3.15}$$

and the eigenvalues are,

$$\lambda_{1,\ldots,k} = 1, \quad \lambda_{k+1,\ldots,k+l} = 0 \tag{3.16}$$

where k are the total number of states and l are the total number of outputs.

Alternatively to Equation 3.7, often a formulation explicit in u is used for co-simulation,

$$x_{n+1} = \Phi(x_n, u_n) \tag{3.17a}$$
$$y_{n+1} = C x_{n+1} + D u_n \tag{3.17b}$$
$$u_{n+1} = L y_{n+1}. \tag{3.17c}$$

Note that while Equation 3.7b and 3.7c are only solvable for y_{n+1} in case of $\det(I - DL) \neq 0$, the Equations 3.17b,c has always a solution. For $D = 0$, the two algorithms are identical, however when $D \neq 0$ which is the case if any sub-system has feed-through, the stability and convergence depends on this coupling.

As before, eliminating the inputs u,

$$x_{n+1} = \Phi(x_n, Ly_n) \tag{3.18a}$$
$$y_{n+1} = Cx_{n+1} + DLy_n. \tag{3.18b}$$

A global step for the states is determined by the same approach as in Equation 3.12,

$$x_{n+1} = A^{-1}(e^{AH} - I)BLy_n + e^{AH}x_n \tag{3.19}$$

inserting this into Equation 3.18b,

$$y_{n+1} = CA^{-1}(e^{AH} - I)BLy_n + Ce^{AH}x_n + DLy_n \tag{3.20}$$
$$= (CA^{-1}(e^{AH} - I)BL + DL)y_n + Ce^{AH}x_n. \tag{3.21}$$

A global step is then calculated as,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} e^{AH} & A^{-1}(e^{AH} - I)BL \\ Ce^{AH} & CA^{-1}(e^{AH} - I)BL + DL \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix}. \tag{3.22}$$

Now, fixing $T = nH$,

$$H \to 0 \quad \Longrightarrow \quad \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} I & 0 \\ C & DL \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{3.23}$$

and the eigenvalues are,

$$\lambda_{1,\ldots,k} = 1, \quad \lambda_{k+1,\ldots,k+l} = \text{eig}(DL). \tag{3.24}$$

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 is less or equal to one, $\rho(DL) \leq 1$ and those equal to one being simple.

Example 3.2.1 (Linear Stability Example)**.** *Consider a linear coupled system of two sub-systems which both consists of one state, x, one input u and one 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,*

$$\dot{x}^{[1]} = -x^{[1]} + u^{[1]} \tag{3.25a}$$
$$y^{[1]} = x^{[1]} + d^{[1]}u^{[1]} \tag{3.25b}$$

and system two defined by,

$$\dot{x}^{[2]} = -x^{[2]} + u^{[2]} \tag{3.26a}$$

$$y^{[2]} = x^{[2]} + d^{[2]}u^{[2]}. \tag{3.26b}$$

The coupling is determined by,

$$\begin{bmatrix} u^{[1]} \\ u^{[2]} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}}_{L} \begin{bmatrix} y^{[1]} \\ y^{[2]} \end{bmatrix}. \tag{3.27}$$

Discretizing the coupled system using constant extrapolation for the signals as described by Equation 3.18, and by letting $H \to 0$ we obtain,

$$\begin{bmatrix} y^{[1]}_{n+1} \\ y^{[2]}_{n+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{C} \begin{bmatrix} x^{[1]}_{n+1} \\ x^{[2]}_{n+1} \end{bmatrix} + \underbrace{\begin{bmatrix} d^{[1]} & 0 \\ 0 & d^{[2]} \end{bmatrix}}_{D} \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}}_{L} \begin{bmatrix} y^{[1]}_{n} \\ y^{[2]}_{n} \end{bmatrix} \tag{3.28}$$

The coupled system is stable if the spectral radius $\rho(DL) \leq 1$. The eigenvalues are,

$$\lambda_{1,2} = \pm\sqrt{d^{[1]}d^{[2]}} \tag{3.29}$$

In Figure 3.3 simulations using the Jacobi method and using 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 3.4, the same coupled system is simulated using Equation 3.9, and as can be seen from the figure, there is no problem related to the stability.

In this section, we have used constant extrapolation for the inputs and defined the cases where we have zero-stability. Changing the extrapolation to use a higher order polynomial, when do we have zero-stability in these cases? In Figure 3.5, Example 3.2.1 where simulated using constant, linear,

$$u_{n+1} = u_n + H\Big(\frac{u_n - u_{n-1}}{H}\Big) \tag{3.30}$$

and quadratic extrapolation,

$$u_{n+1} = u_n + H\Big(\frac{u_n - u_{n-1}}{H}\Big) + \frac{H^2}{2}\Big(\frac{u_n - 2u_{n-1} + u_{n-2}}{H^2}\Big). \tag{3.31}$$

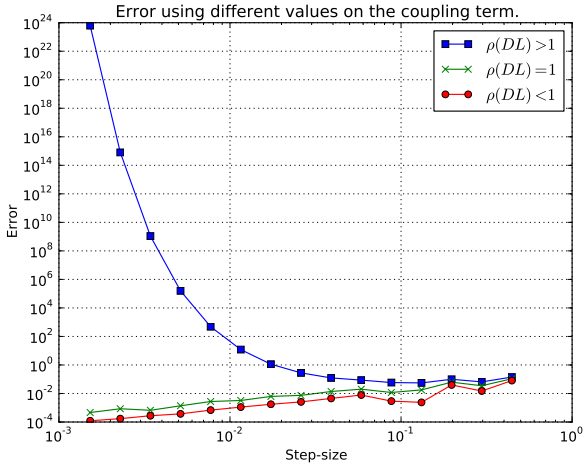

Figure 3.3: *Result of simulating Model 3.25 and Model 3.26 in parallel using Equation 3.18 and for varying values on d_1 and d_2.*

where old values was 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 zero-stable integration changes with the extrapolation order.

3.2.1 Linear Extrapolation

Equation 3.17 is based on the assumption that constant extrapolation was used in-between the global time-steps. If we instead assume linear extrapolation (Equation 3.30) we find that,

$$x_{n+1} = \Phi(x_n, u_n, u_{n-1}) \tag{3.32a}$$

$$y_{n+1} = Cx_{n+1} + D\Big(u_n + H\Big(\frac{u_n - u_{n-1}}{H}\Big)\Big) \tag{3.32b}$$

$$u_{n+1} = Ly_{n+1} \tag{3.32c}$$

eliminating u,

$$x_{n+1} = \Phi(x_n, Ly_n, Ly_{n-1}) \tag{3.33a}$$

$$y_{n+1} = Cx_{n+1} + D\big(2Ly_n - Ly_{n-1}\big). \tag{3.33b}$$

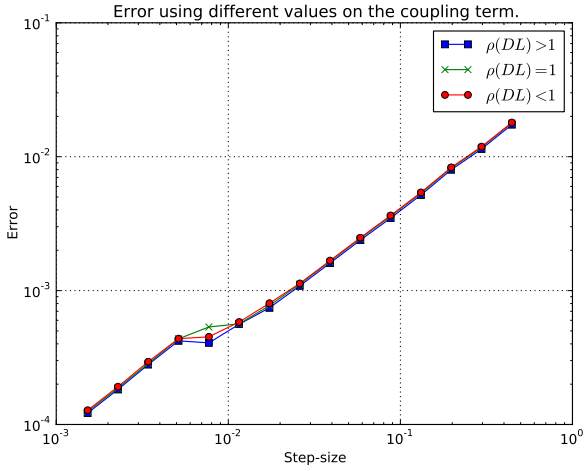

Figure 3.4: *Result of simulating Model 3.25 and Model 3.26 in parallel using Equation 3.9 and for varying values on d_1 and d_2.*

Still assuming that the states are solved exactly in each sub-system,

$$x(t) = \int_{T_n}^{t} e^{A(t-\tau)} BLy(\tau) d\tau + e^{A(T_n - t)} x(T_n) \tag{3.34}$$

and using the linear extrapolation we find for the states,

$$\begin{aligned}
x_{n+1} &= \Phi(x_n, Ly_n, Ly_{n-1}) && \text{(3.35a)}\\
&= \int_{T_n}^{T_{n+1}} e^{A(T_{n+1}-\tau)} BL\Big(y_n + (\tau - T_n)\Big(\frac{y_n - y_{n-1}}{H}\Big)\Big) d\tau && \text{(3.35b)}\\
&\quad + e^{A(T_{n+1}-T_n)} x_n && \text{(3.35c)}\\
&= \big[\bar{\tau} = \tau - T_n\big] && \text{(3.35d)}\\
&= \int_{0}^{H} e^{A(H-\bar{\tau})} BL\Big(y_n + \bar{\tau}\Big(\frac{y_n - y_{n-1}}{H}\Big)\Big) d\bar{\tau} && \text{(3.35e)}\\
&\quad + e^{A(T_{n+1}-T_n)} x_n && \text{(3.35f)}\\
&= c_1 y_n + c_2\big(y_n - y_{n-1}\big) + e^{AH} x_n && \text{(3.35g)}\\
&= (c_1 + c_2) y_n - c_2 y_{n-1} + e^{AH} x_n && \text{(3.35h)}
\end{aligned}$$

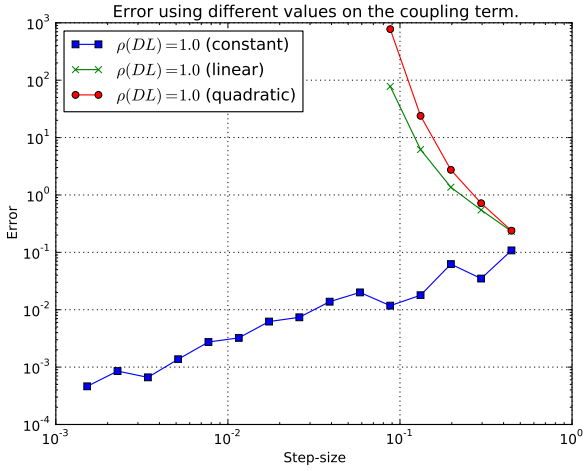

Figure 3.5: *Result of simulating Model 3.25 and Model 3.26 in parallel using constant, linear and quadratic extrapolation and using old values in the calculation of the output variables.*

where,

$$c_1 = A^{-1}(e^{AH} - I)BL \tag{3.36}$$

$$c_2 = A^{-2}(e^{AH} - I - Ah)BLH^{-1}. \tag{3.37}$$

For the outputs we get,

$$\begin{aligned} y_{n+1} &= Cx_{n+1} + 2DLy_n - DLy_{n-1} && \text{(3.38a)}\\ &= C(c_1 + c_2)y_n - Cc_2y_{n-1} + Ce^{AH}x_n && \text{(3.38b)}\\ &\quad + 2DLy_n - DLy_{n-1} && \text{(3.38c)}\\ &= (Cc_1 + Cc_2 + 2DL)y_n && \text{(3.38d)}\\ &\quad + (-Cc_2 - DL)y_{n-1} && \text{(3.38e)}\\ &\quad + Ce^{AH}x_n. && \text{(3.38f)} \end{aligned}$$

The resulting global step is performed as,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ y_n \end{bmatrix} = \begin{bmatrix} e^{AH} & c_1 + c_2 & -c_2 \\ Ce^{AH} & C(c_1 + c_2) + 2DL & -Cc_2 - DL \\ 0 & I & 0 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \\ y_{n-1} \end{bmatrix} \tag{3.39}$$

Using this approach, what is the requirements for the method to be stable? Fixing $T = nH$,

$$H \to 0 \quad \Longrightarrow \quad \begin{bmatrix} c_1 = 0 \\ c_2 = 0 \end{bmatrix} \tag{3.40}$$

we get,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ y_n \end{bmatrix} = \begin{bmatrix} I & 0 & 0 \\ C & 2DL & -DL \\ 0 & I & 0 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \\ y_{n-1} \end{bmatrix}. \tag{3.41}$$

The eigenvalues are,

$$\lambda_{1,\ldots,k} = 1, \quad \lambda_{k+1,\ldots,k+2l} = \text{eig}\left(\begin{bmatrix} 2DL & -DL \\ I & 0 \end{bmatrix} \right) \tag{3.42}$$

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 being equal to one are simple.

If we look again at the consistent approach, Equation 3.9, where the output equations and the coupling equations are solved, but this time with linear extrapolation,

$$x_{n+1} = \Phi(x_n, Ly_n, Ly_{n-1}) \tag{3.43a}$$

$$y_{n+1} = (I - DL)^{-1} C x_{n+1}. \tag{3.43b}$$

The states are calculated as Equation 3.35 and inserting this into Equation 3.43b,

$$y_{n+1} = (I - DL)^{-1} C((c_1 + c_2)y_n - c_2 y_{n-1} + e^{Ah} x_n) \tag{3.44}$$

and a global step is calculated with $[c_3 = (I - DL)^{-1} C]$ as,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ y_n \end{bmatrix} = \begin{bmatrix} e^{AH} & c_1 + c_2 & -c_2 \\ c_3 e^{Ah} & c_3(c_1 + c_2) & -c_3 c_2 \\ 0 & I & 0 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \\ y_{n-1} \end{bmatrix} \tag{3.45}$$

Fixing $T = nH$,

$$H \to 0 \quad \Longrightarrow \quad \begin{bmatrix} c_1 = 0 \\ c_2 = 0 \end{bmatrix} \tag{3.46}$$

and,

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ y_n \end{bmatrix} = \begin{bmatrix} I & 0 & 0 \\ (I - DL)^{-1}C & 0 & 0 \\ 0 & I & 0 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \\ y_{n-1} \end{bmatrix} \tag{3.47}$$

with the eigenvalues,

$$\lambda_{1,\ldots,k} = 1, \quad \lambda_{k+1,\ldots,k+2l} = 0 \tag{3.48}$$

where k are the total number of states and l are the total number of outputs. In this case we have stability.

In Figure 3.6 the regions for which the inconsistent approaches are stable are shown for constant, linear and quadratic extrapolation. The requirements for quadratic extrapolation can be found straightforward from above. In Figure 3.7 and in Figure 3.8, result is shown for simulating Example 3.2.1

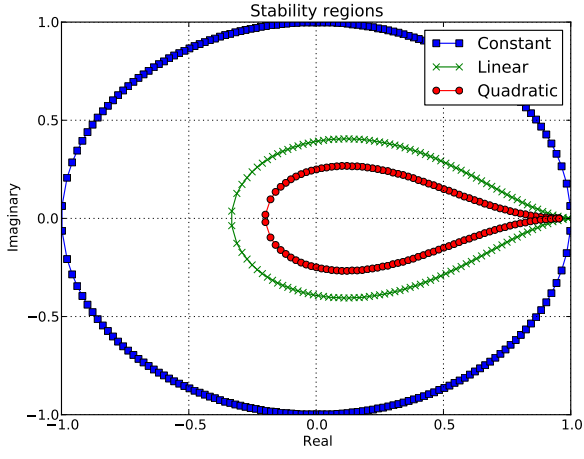

Figure 3.6: *Regions for which the inconsistent approaches are convergent in terms of $\rho(DL)$ using different extrapolation approaches.*

using linear and quadratic extrapolation respectively with the inconsistent approach.

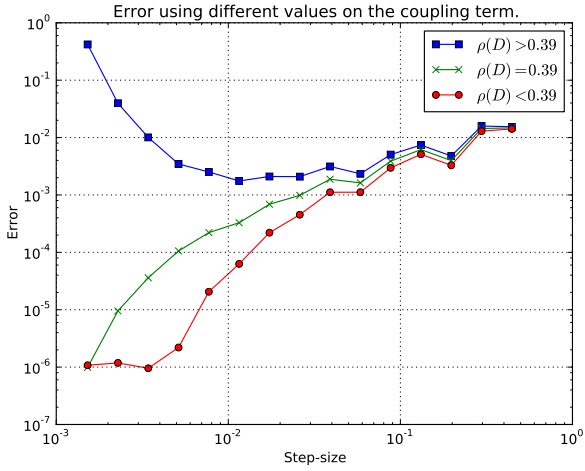

Figure 3.7: *Result of simulating Model 3.25 and Model 3.26 in parallel using linear extrapolation and for varying values on d_1 and d_2.*

3.3 Variable Step-Size Integrators

In order to develop an efficient and robust simulation method, a key component is the ability to vary the step-size depending on the local behavior. If the local behavior is "slowly changing" we may use a larger step-size as the simulation method better approximates the solution. If the local behavior is "rapidly changing" we want to use a smaller step-size. A requirement for varying the step-size is the ability to estimate the error. How this is done depends on the simulation method. Another important component is the ability to reject an integration step and redo the step with different options which is typically done in the case when an estimated error is larger then a user defined tolerance.

3.3.1 Restart of Integration

Creating error estimators in a co-simulation environment requires that the global time-step $T_n + H \rightarrow T_{n+1}$ is performed more than once with different options. In order to be able to perform a global time-step multiple times the possibility to store the current solver state is required. Note, that for a global time-step, multiple local time-steps may have been performed,

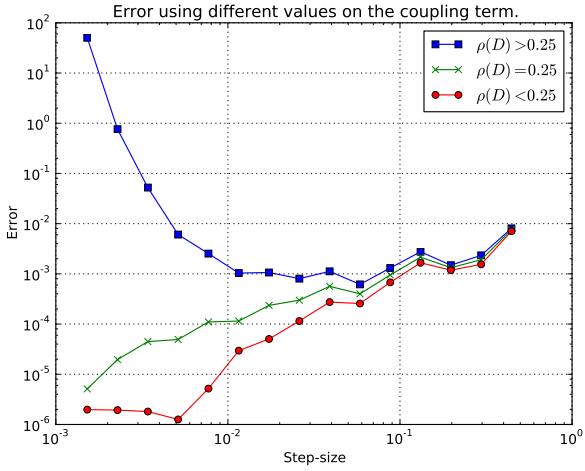

Figure 3.8: *Result of simulating Model 3.25 and Model 3.26 in parallel using quadratic extrapolation and for varying values on d_1 and d_2.*

discarding possible history of the internal solver already. It should also be noted that applying the stored state to the solver again is not the same as re-initializing the solver. The aim is here that, in the case for variable step-size, variable-order multistep methods to store the complete set of step-size history and order history etc, which will not result in the same simulation as if one simply re-initialize the solver with a blank history.

Example 3.3.1 (The Van der Pol Oscillator)**.** *The Van der Pol oscillator is given by the equations,*

$$\dot{x}_1 = x_2, \qquad x_1(t_0) = 2 \tag{3.49a}$$

$$\dot{x}_2 = 10^6((1 - x_1^2)x_2 - x_1), \qquad x_2(t_0) = -0.6. \tag{3.49b}$$

The intention is to simulate the problem using CVODE from $t_0 = 0 \rightarrow t_1 = 0.5$. At t_1, the simulation is interrupted and the solver state is stored. The simulation is then continued until $t_2 = 0.75$. The calculated solution is compared with a simulation where the solver state is restored at t_1 and with a simulation where the solver is re-initialized at t_1. In Figure 3.9, the result is shown where one clearly sees that re-initializing a solver produces different result as compared to applying a stored solver state. Comparing

the number of function evaluations, we find that for the restored and the continued simulation the number of function evaluations was 268 *while for the re-initialized simulation,* 330*.*

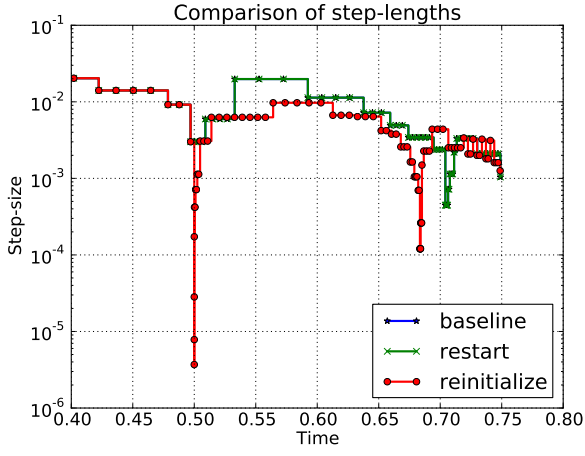

Figure 3.9: *Step-size history comparison for when applying a stored solver state at* $t_1 = 0.5$ *with re-initializing the same solver at* t_1*. Note that the baseline overlaps the restart line.*

For explicit fixed step-size one-step methods such as Explicit Euler, storing the solver state is trivial. Explicit Euler does not have any history or other internal settings so the solver state is simply the problem states x_i. However, simply looking at Implicit Euler things become more tricky, as it involves solving a nonlinear system with possibly Jacobian information that may or may not need to be stored depending on the approach. In a state-of-the art BDF code, such as CVODE [23], storing a solver state becomes increasingly complex.

In CVODE which is a variable-order variable step-size integrator, there are heuristics for changing the order, updating the Jacobian and changing the step-size. Throughout the integration, the order of the method is continuously evaluated for possible order increase or order decrease with heuristics for determining if it is allowed or not. For instance, have enough steps been performed on the current order for an order increase? All this needs to be taken into account when storing the solver status.

None of the currently most used solvers such as CVODE, DASSL [36] and RADAU5 [21] provide this feature by default. The information necessary is usually located in an internal memory, not accessible to the user.

In case of discontinuous problems, storing only the solver state is not enough. For a discontinuous problem, information about the discrete variables that determine the mode of "operation" or which influence the behavior of the problem need also to be stored. These variables are typically not part of the continuous solution found by the solver but they influence the behavior through the events.

3.3.1.1 CVODE

Storing the solver state of an advanced simulation code like CVODE is a complex task. This section is intended as a guide through the algorithmic parts of CVODE where the necessary information for storing the solver state is highlighted. In addition to the algorithmic information needed there are also heuristics that need to be taken into consideration.

CVODE implements multistep methods for solving an ordinary differential equation,

$$\sum_{i=0}^{q} \alpha_{n,i} y_{n-i} + h_n \sum_{i=0}^{q} \beta_{n,i} f(t_{n-i}, y_{n-i}) = 0 \tag{3.50}$$

where q determines the number of steps in the formula. CVODE supports both an Adams-Moulton method for non-stiff problems with q varying between 1 and 12. For stiff problems, a BDF method in fixed leading coefficient form is implemented with q varying between 1 and 5. The coefficients α and β determine the method and they are additionally dependent on the step-size history and order.

As CVODE implements multistep methods, access to the solution history is needed during the integration. The solution history is stored as a Nordsieck history array,

$$z_{n-1} = \left[y_{n-1}, h\dot{y}_{n-1}, \ldots, \frac{h^q y_{n-1}^{(q)}}{q!} \right]. \tag{3.51}$$

From the solution history z_{n-1}, an prediction to z_n is calculated as,

$$z_{n(0)} = z_{n-1} A(q) \tag{3.52}$$

where $A(q)$ is a matrix with Pascal's triangle in its lower left part. The correction to z_n is performed as,

$$z_n = z_{n(0)} + le_n, \quad l = [l_0, \ldots, l_q] \tag{3.53}$$

where l is depends on the step-size history and the order. The correction vector e is,

$$e_n = y_n - y_{n(0)}. \tag{3.54}$$

What is left is calculating the solution y_n. Using Equation 3.50 an equation for y_n is found, rearranging, the resulting equation is,

$$G(y_n) = 0 \tag{3.55}$$

which can be solved by either functional iteration or Newton iteration. In case of Newton iteration,

$$My_{n(m+1)} - My_{n(m)} = G(y_{n(m)}), \quad M \approx I - \gamma \frac{\partial f}{\partial y} \quad \gamma = h_n \beta_{n,0}. \tag{3.56}$$

where m is the iteration counter and y_n is the finally accepted iteration result.

In a variable step-size method, the error is estimated at every step and used for determining step-size changes and order changes. The local truncation error estimate (LTE) is calculated as,

$$\text{LTE}_q = C_q e_n \tag{3.57}$$

where C_q depends on the step-size history. The error is additionally estimated for the corresponding $q-1$ and $q+1$ step methods as,

$$\text{LTE}_{q-1} = C_{q-1} e_n \tag{3.58}$$

$$\text{LTE}_{q+1} = C_{q+1} e_n \tag{3.59}$$

where C_{q-1} depends on the step-size history while C_{q+1} depends on the step-size history and the previous correction vector e_{n+1}

When performing order changes, the history array needs to be updated accordingly. When considering an order change, either $q-1$ or $q+1$, the history array z_n is updated as,

$$z_n^* = \begin{cases} \bar{z}_n + \frac{h^q y_n^{(q)}}{q!} d & \text{for } q-1 \\ \tilde{z}_n + e_n c & \text{for } q+1 \end{cases} \tag{3.60}$$

where $\bar{z}_n$ is the history array without the last column and $\tilde{z}_n$ is the history array appended by a zero valued column. The coefficient vector c is dependent on the step-size history and the correction vector e_n while d is only dependent on the step-size history. If the step-size is changed by a factor η, $h' = \eta h$ the solution history is updated as,

$$z_n^* = z_n \text{diag}[1, \eta, \ldots, \eta^q] \tag{3.61}$$

The new order and step-size is set according to maximize the length of the next step.

Summarizing the algorithmic requirements for storing the solver state,

- The Nordsieck history vector, z_{n-1}.
- The step-size history τ from which l, c and d can be calculated.
- The current order q.
- The step-size to be tried on the next step, h.
- The calculated correction vector, e_n

In an advanced solver such as CVODE there are control parameters included for improving the robustness. An order change for instance, is only considered if $q+1$ steps have been successfully been computed, without any convergence failure or error test failure, since the last change. An update of M in Equation 3.56, occurs when 20 steps have been taken since the last update or if an error test or convergence failure occurred. Additionally an update occurs if $|\gamma/\gamma_{old} - 1| > 0.3$. Similarly, the Jacobian in Equation 3.56 is updated after 50 steps since the last update, after a convergence failure which forced a step-size reduction and after a convergence failure with an outdated M together with $|\gamma/\gamma_{old} - 1| < 0.2$.

Summarizing the control parameters requirements for storing the solver state,

- Number of steps to wait before considering an order change, q_{wait}
- The Jacobian factor, γ_{old}
- Number of steps since last M update
- Number of steps since last Jacobian update

As a final step, the run-time statistics needs to be stored.

Regarding the practical considerations, this information is all stored in an internal data structure which is not accessible in a default installation of CVODE. In order to store the solver state, the installation needs modification so that this data structure can be accessed. Additionally, method for performing duplication and restoration are needed.

For a detailed description of CVODE, see [13, 23, 24, 14, 28].

3.3.1.2 Restart of an FMU

In the FMI version 1.0 for co-simulation, storing the complete model state, including the solver state, is not possible. This severely limits the options for implementing an advanced general master algorithm. As discussed in Section 2.2, the FMI version 2.0 includes an API for just saving and restoring the full state, which significantly improves the interface in case of co-simulation. The model support for this is determined by an optional capability flag.

3.3.2 Error Estimation

Estimating the local error for a method is essential in order to be able to adapt the step-size. For co-simulation, a method based on using Richardson extrapolation was proposed in [41] where the idea is to compare two simulations using different step-sizes. The first simulation with a global integration step of $2H$ and the another by performing two steps with a global integration step of H, see Figure 3.10. Between the two steps in the second simulation, input and output data is exchanged between the sub-models. Richardson extrapolation is a reliable and robust algorithm for producing higher order approximations using low-order methods. The idea is to combine the two approximations in such a way that their largest error terms cancel. See [20] for information about Richardson extrapolation.

Theorem 3.3.1 (Local error without feed-through)**.** *The error in the outputs, y, in Equation 3.74, calculated using one step of step-size $2H$ (y_{2H}) and using two steps of step-size H (y_H) where coupling data is exchanged satisfies,*

$$y(T_{n+2}) - y_{2H}(T_{n+2}) = c_1(2H)^{k+2} + \mathcal{O}(H^{k+3}) \tag{3.62}$$

$$y(T_{n+2}) - y_H(T_{n+2}) = 2c_1H^{k+2} + \mathcal{O}(H^{k+3}) \tag{3.63}$$

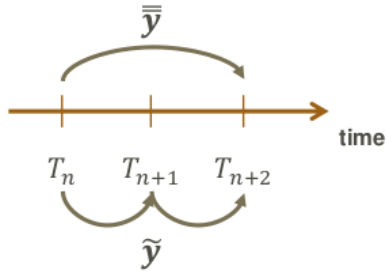

Figure 3.10: *Error estimation procedure using Richardson extrapolation.*

whenever there is no direct feed-through and where k is the order of the extrapolation of the inputs u and c_1 is a factor independent on H.

Proof. See Section 3.3.2.1. □

The resulting solutions for the two simulations are then compared at $T_{n+2} = T_n + 2H$ and an estimate for the error is calculated using Theorem 3.3.1 and neglecting the higher order terms as,

$$y_H(T_{n+2}) - y_{2H}(T_{n+2}) = c_1(2H)^{k+2} - 2c_1H^{k+2} \tag{3.64}$$

$$2c_1H^{k+2} = \frac{y_H(T_{n+2}) - y_{2H}(T_{n+2})}{2^{k+1} - 1} \tag{3.65}$$

and finally the error estimate,

$$EST = \frac{y_H(T_{n+2}) - y_{2H}(T_{n+2})}{2^{k+1} - 1} \tag{3.66}$$

Theorem 3.3.2 (Local error with feed-through)**.** *The error in the outputs, y, in Equation 3.74, calculated using one step of step-size $2H$ (y_{2H}) and using two steps of step-size H (y_H) where coupling data is exchanged satisfies,*

$$y(T_{n+2}) - y_{2H}(T_{n+2}) = c_2(2H)^{k+1} + \mathcal{O}(H^{k+2}) \tag{3.67}$$

$$y(T_{n+2}) - y_H(T_{n+2}) = \underbrace{(I + DL)}_{c_3} c_2H^{k+1} + \mathcal{O}(H^{k+2}) \tag{3.68}$$

when there is direct feed-through and where k is the order of the extrapolation of the inputs u and c_2,c_3 are factors independent on H.

As with Theorem 3.3.1, we compare the two simulations at $T_{n+2} = T_n + 2H$ from Theorem 3.3.2,

$$y_H(T_{n+2}) - y_{2H}(T_{n+2}) = c_2(2H)^{k+1} - c_3c_2H^{k+1} \tag{3.69}$$

$$c_3c_2H^{k+1} = (2^{k+1}c_3^{-1} - I)^{-1}(y_H(T_{n+2}) - y_{2H}(T_{n+2})) \tag{3.70}$$

and the estimate,

$$EST = (2^{k+1}c_3^{-1} - I)^{-1}(y_H(T_{n+2}) - y_{2H}(T_{n+2})) \tag{3.71}$$

A modified approach, compared to performing a step of size $2H$ and two steps of size H, was proposed in [41] to mitigate the high cost of the error estimation. The modified approach was to perform a step of size H for both simulations and then for the first simulation, continue with another step of size H. For the second simulation, information was exchanged among the sub-systems before performing the second step of H.

In a simulation code, the error estimate, Equation 3.66 and Equation 3.71, is usually normalized using user supplied tolerances, both an absolute tolerance and an relative tolerance,

$$ERR = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\frac{EST_i}{ATOL_i + RTOL\|y_H(T_{n+2})\|}}, \tag{3.72}$$

$ERR \leq 1$ results in an accepted step, otherwise the step has to be rejected and repeated with a smaller global step-size.

The normalized error is then used to adapt the global step-size such that it meets the user supplied tolerances while performing the largest possible step [20],

$$h_{new} = h_{prev}\frac{1}{ERR}^{\frac{1}{k+2}}. \tag{3.73}$$

3.3.2.1 Error Analysis

In our error analysis we start from the nonlinear coupled system,

$$\dot{x}^{[i]} = f^{[i]}(t, x^{[i]}, u^{[i]}), \quad i = 1, \ldots, N \tag{3.74a}$$

$$y^{[i]} = g^{[i]}(t, x^{[i]}, u^{[i]}), \quad i = 1, \ldots, N \tag{3.74b}$$

$$u = c(y) \tag{3.74c}$$

and we will investigate the error propagation when using an approximation for the inputs $\bar{u}^{[i]}(t) \approx u^{[i]}(t)$. In the sequel, the solutions resulting from the approximation will be denoted with bars and the superscript is dropped for the individual sub-systems. Comparing the solution with the approximation, gives

$$\dot{x} - \dot{\bar{x}} = f(x,u) - f(\bar{x},\bar{u}) \tag{3.75a}$$
$$y - \bar{y} = g(x,u) - g(\bar{x},\bar{u}) \tag{3.75b}$$
$$u - \bar{u} = c(y) - c(\bar{y}). \tag{3.75c}$$

Expanding the functions on the right-hand side in a Taylor series, we get for the states,

$$\dot{x} - \dot{\bar{x}} = \underbrace{\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x(T_n),u(T_n)}}_{A}(x-\bar{x}) + \underbrace{\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{x(T_n),u(T_n)}}_{B}(u-\bar{u}) \tag{3.76}$$

and for the outputs,

$$y - \bar{y} = \underbrace{\left.\frac{\mathrm{d}g}{\mathrm{d}x}\right|_{x(T_n),u(T_n)}}_{C}(x-\bar{x}) + \underbrace{\left.\frac{\mathrm{d}g}{\mathrm{d}u}\right|_{x(T_n),u(T_n)}}_{D}(u-\bar{u}) \tag{3.77}$$

and finally for the inputs,

$$u - \bar{u} = \underbrace{\left.\frac{\mathrm{d}c}{\mathrm{d}y}\right|_{y(T_n)}}_{L}(y-\bar{y}). \tag{3.78}$$

Note that here we are assuming that higher order terms are negligible, i.e. products and powers of the difference between the solution and the approximation. From the following calculations this assumption will be made clear as other terms contribute to the error significantly more.

The approximation for the inputs is an extrapolation from previous known values and the extrapolation error is defined for $t \in [T_n, T_{n+1}]$ as,

$$u(t) - \bar{u}(t) = \frac{u^{(k+1)}(\xi)}{(k+1)!}(t-T_n)^{k+1}, \quad \xi \in [T_n, T_{n+1}] \tag{3.79}$$

$$u(t) - \bar{u}(t) = \frac{u^{(k+1)}(T_n)}{(k+1)!}(t-T_n)^{k+1} + \mathcal{O}((\underbrace{t-T_n}_{H})^{k+2}) \tag{3.80}$$

where $\bar{u}$ is the kth degree Taylor polynomial and assuming that $u \in \mathcal{C}^{k+2}$.

$$x(t) - \bar{x}(t) = \int_{T_n}^{t} e^{A(t-\tau)} B(u(\tau) - \bar{u}(\tau)) d\tau + e^{At}(x(T_n) - \bar{x}(T_n)) \quad (3.81)$$

$$y(t) - \bar{y}(t) = C(x(t) - \bar{x}(t)) + D(u(t) - \bar{u}(t)) \quad (3.82)$$

Considering first the error in a step $T_n \to T_{n+1}$ of step-size H for the states,

$$x(T_{n+1}) - \bar{x}(T_{n+1}) = \int_{T_n}^{T_{n+1}} e^{A(T_{n+1}-\tau)} B(u(\tau) - \bar{u}(\tau)) d\tau \quad (3.83a)$$

$$+ e^{AT_{n+1}}(x(T_n) - \bar{x}(T_n)). \quad (3.83b)$$

Using Equation 3.80 together with Equation 3.83 and using that in the first step $x(T_n) = \bar{x}(T_n)$, we find,

$$x(T_{n+1}) - \bar{x}(T_{n+1}) = \int_{T_n}^{T_{n+1}} e^{A(T_{n+1}-\tau)} B\Big(\frac{u^{(k+1)}(T_n)}{(k+1)!}(\tau - T_n)^{k+1} + \dots \quad (3.84a)$$

$$+ \mathcal{O}((\tau - T_n)^{k+2}) \Big) \mathrm{d}\tau. \quad (3.84b)$$

Changing the integration limits using $\bar{\tau} = \tau - T_n$,

$$x(T_{n+1}) - \bar{x}(T_{n+1}) = \int_{0}^{H} e^{A(H-\bar{\tau})} B\Big(\frac{u^{(k+1)}(T_n)}{(k+1)!}\bar{\tau}^{k+1} + \mathcal{O}(\bar{\tau}^{k+2}) \Big) \mathrm{d}\bar{\tau} \quad (3.85a)$$

$$= \int_{0}^{H} e^{A(H-\bar{\tau})} \bar{\tau}^{k+1} \mathrm{d}\bar{\tau} B \frac{u^{(k+1)}(T_n)}{(k+1)!} \quad (3.85b)$$

$$+ \int_{0}^{H} e^{A(H-\bar{\tau})} \mathcal{O}(\bar{\tau}^{k+2}) \mathrm{d}\bar{\tau} B. \quad (3.85c)$$

Using the definition for the matrix exponential we get a first order approximation,

$$e^{A} = \sum_{i=0}^{\infty} \frac{1}{i!} A^i \quad \Longrightarrow \quad e^{A(H-\tau)} \approx I + \mathcal{O}(H), \quad \tau \in [0, H] \quad (3.86)$$

which is appropriate as the first term will dominate the error. Finally for the states, we get,

$$x(T_{n+1}) - \bar{x}(T_{n+1}) = \int_0^H (I + \mathcal{O}(\bar{\tau}))\bar{\tau}^{k+1}\mathrm{d}\bar{\tau}B\frac{u^{k+1}(T_n)}{(k+1)!} \tag{3.87a}$$

$$+ \int_0^H (I + \mathcal{O}(\bar{\tau}))\mathcal{O}(\bar{\tau}^{k+2})\mathrm{d}\bar{\tau}B \tag{3.87b}$$

$$= B\frac{u^{(k+1)}(T_n)}{(k+2)!}H^{k+2} + \mathcal{O}(H^{k+3}) \tag{3.87c}$$

and for the outputs,

$$y(T_{n+1}) - \bar{y}(T_{n+1}) = C(x(T_{n+1}) - \bar{x}(T_{n+1})) \tag{3.88a}$$

$$+ D(u(T_{n+1}) - \bar{u}(T_{n+1})) \tag{3.88b}$$

$$= CB\frac{u^{(k+1)}(T_n)}{(k+2)!}H^{k+2} + \mathcal{O}(H^{k+3}) \tag{3.88c}$$

$$+ D\frac{u^{(k+1)}(T_n)}{(k+1)!}H^{k+1} + D\mathcal{O}(H^{k+2}). \tag{3.88d}$$

In the next step, $T_{n+1} \to T_{n+2}$, we denote the resulting solution and the approximation with double bars ($\bar{\bar{u}}$). The approximation error for the inputs in this second step includes the error accumulated in the first step,

$$u(t) - \bar{\bar{u}}(t) = \frac{u^{(k+1)}(T_{n+1})}{(k+1)!}(t - T_{n+1})^{k+1} + \mathcal{O}((t - T_{n+1})^{k+2}) \tag{3.89}$$
$$+ L(y(T_{n+1}) - \bar{y}(T_{n+1})).$$

Additionally, we have that $\bar{x}(T_{n+1}) = \bar{\bar{x}}(T_{n+1})$ and we get for the states,

$$x(T_{n+2}) - \bar{\bar{x}}(T_{n+2}) = \int_{T_{n+1}}^{T_{n+2}} e^{A(T_{n+2}-\tau)}B(u(\tau) - \bar{\bar{u}}(\tau))\mathrm{d}\tau \tag{3.90a}$$

$$+ e^{AH}(x(T_{n+1}) - \bar{x}(T_{n+1})). \tag{3.90b}$$

Following the same approach as in the first step and using Equation 3.89,

Equation 3.86 and Equation 3.87c,

$$x(T_{n+2}) - \bar{\bar{x}}(T_{n+2}) = \int_0^H (I + \mathcal{O}(\bar{\tau}))B\Big(\frac{u^{(k+1)}(T_{n+1})}{(k+1)!}\bar{\tau}^{k+1} + \mathcal{O}(\bar{\tau}^{k+2}) \tag{3.91a}$$

$$+ L(y(T_{n+1}) - \bar{y}(T_{n+1}))\Big)\mathrm{d}\bar{\tau} \tag{3.91b}$$

$$+ (I + \mathcal{O}(H))\Big(B\frac{u^{(k+1)}(T_n)}{(k+2)!}H^{k+2} + \mathcal{O}(H^{k+3})\Big) \tag{3.91c}$$

which simplifies to,

$$x(T_{n+2}) - \bar{\bar{x}}(T_{n+2}) = B\frac{u^{(k+1)}(T_{n+1})}{(k+2)!}H^{k+2} + BLD\frac{u^{(k+1)}(T_n)}{(k+1)!}H^{k+2} \tag{3.92a}$$

$$+ B\frac{u^{(k+1)}(T_n)}{(k+2)!}H^{k+2} + \mathcal{O}(H^{k+3}). \tag{3.92b}$$

For the outputs at T_{n+2},

$$y(T_{n+2}) - \bar{\bar{y}}(T_{n+2}) = C(x(T_{n+2}) - \bar{\bar{x}}(T_{n+2})) + D((u(T_{n+2}) - \bar{\bar{u}}(T_{n+2})) \tag{3.93a}$$

$$= CB\frac{u^{(k+1)}(T_{n+1})}{(k+2)!}H^{k+2} + CB\frac{u^{(k+1)}(T_n)}{(k+2)!}H^{k+2} \tag{3.93b}$$

$$+ D\frac{u^{(k+1)}(T_{n+1})}{(k+1)!}H^{k+1} + DLD\frac{u^{(k+1)}(T_n)}{(k+1)!}H^{k+1} \tag{3.93c}$$

$$+ D\mathcal{O}(H^{k+2}) + \mathcal{O}(H^{k+3}). \tag{3.93d}$$

Similarly as the error in the first step ($T_n \to T_{n+1}$), the error in a step of size $2H$, $T_n \to T_{n+2}$ is,

$$y(T_{n+2}) - \tilde{y}(T_{n+2}) = CB\frac{u^{(k+1)}(T_n)}{(k+2)!}(2H)^{k+2} + D\frac{u^{(k+1)}(T_n)}{(k+1)!}(2H)^{k+1} \tag{3.94a}$$

$$+ D\mathcal{O}(H^{k+2}) + \mathcal{O}(H^{k+3}) \tag{3.94b}$$

So if there is no direct feed-through in the systems, i.e. $D = 0$ we get for the two approximations,

$$y(T_{n+2}) - \tilde{y}(T_{n+2}) = CB\frac{u^{(k+1)}(T_n)}{(k+2)!}(2H)^{k+2} + \mathcal{O}(H^{k+3}) \tag{3.95}$$

$$y(T_{n+2}) - \bar{\tilde{y}}(T_{n+2}) = CB\frac{u^{(k+1)}(T_n)}{(k+2)!}2H^{k+2} + \mathcal{O}(H^{k+3}) \tag{3.96}$$

which corresponds to Theorem 3.3.1. While if $D \neq 0$ we have instead,

$$y(T_{n+2}) - \tilde{y}(T_{n+2}) = D\frac{u^{(k+1)}(T_n)}{(k+1)!}(2H)^{k+1} + \mathcal{O}(H^{k+2}) \tag{3.97}$$

$$y(T_{n+2}) - \bar{\tilde{y}}(T_{n+2}) = \Big(I + DL\Big)D\frac{u^{(k+1)}(T_n)}{(k+1)!}H^{k+1} + \mathcal{O}(H^{k+2}) \tag{3.98}$$

which corresponds to Theorem 3.3.1 with $c_3 = I + DL$.

Example 3.3.2 (Without feed-through)**.** *Consider a mass-spring-damper problem with three coupled springs and three coupled dampers which are connected between two fixed points and two masses, see Figure 3.11. The*

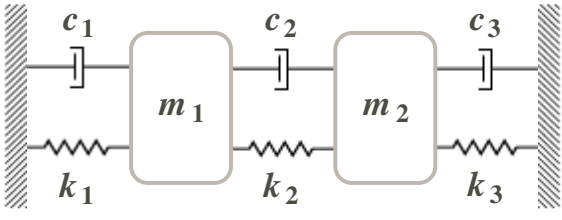

Figure 3.11: *Mass-spring-damper problem.*

problem is governed by the equations,

$$m_1\ddot{x}_1 = -(k_1 + k_2)x_1 - (c_1 + c_2)\dot{x}_1 + k_2x_2 + c_2\dot{x}_2 \tag{3.99a}$$

$$m_2\ddot{x}_2 = -(k_3 + k_2)x_2 - (c_3 + c_2)\dot{x}_2 + k_2x_1 + c_2\dot{x}_1. \tag{3.99b}$$

Dividing the problem into two sub-systems where the position and velocity

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

Table 3.1: *Parameters used in the Example 3.3.2.*

are both the inputs and outputs from the models, i.e,

$$m_1\ddot{x}_1 = -(k_1 + k_2)x_1 - (c_1 + c_2)\dot{x}_1 + k_2u_{11} + c_2u_{12} \tag{3.100a}$$
$$y_1 = [x_1, \dot{x}_1]^{\mathrm{T}} \tag{3.100b}$$
$$m_2\ddot{x}_2 = -(k_3 + k_2)x_2 - (c_3 + c_2)\dot{x}_2 + k_2u_{21} + c_2u_{22} \tag{3.100c}$$
$$y_2 = [x_2, \dot{x}_2]^{\mathrm{T}}. \tag{3.100d}$$

The coupling equations are defined by,

$$u_{11} = y_{21}, \qquad u_{21} = y_{11} \tag{3.101a}$$
$$u_{12} = y_{22}, \qquad u_{22} = y_{12}. \tag{3.101b}$$

The problem is solved using a parallel co-simulation approach with different extrapolations and the estimated local error is compared against the analytical local error. The problem parameters are listed in Table 3.1.

In Figure 3.12 the estimated local error using Richardson extrapolation for constant extrapolation is shown and in Figure 3.13 it is shown for linear extrapolation. Note the difference in the step-sizes that is due to that in the linear case, the internal tolerance of the sub-systems impacted the error estimation when $H = 0.0001$ was used. The internal tolerances of the sub-systems was set to 10^{-11}.

In Example 3.3.2, it was shown that the error estimate is in good agreement with the analytical error. However, the error estimate is based on the assumption that the sub-systems are solved exactly, which is not entirely true. In the example, the sub-systems were solved using a very low tolerance 10^{-11} in order to mimic the assumption that the sub-systems are solved exactly. By changing the tolerance on the sub-systems however, the estimate becomes overly optimistic, see Figure 3.14. The figure shows the importance of using an appropriate tolerance on the sub-systems.

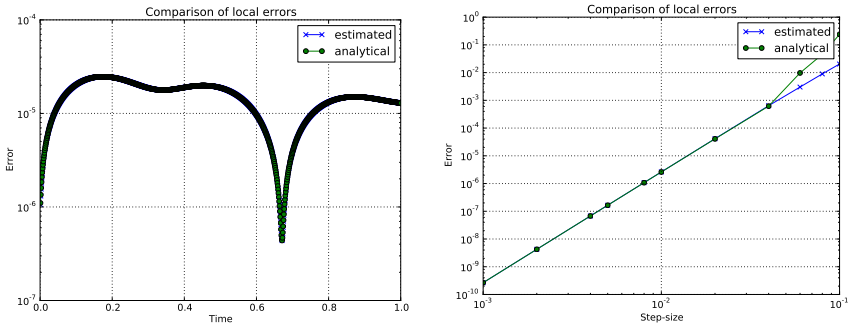

Figure 3.12: *Estimated local errors together with the analytical local error for constant extrapolation of Example 3.3.2. In the left figure, the local errors are shown for a simulation over one second for a given step-size of $H = 0.001$. In the right figure, the local errors are shown for various step-sizes.*

Example 3.3.3 (With feed-through)**.** *Looking again at Equation 3.99 in Example 3.3.2,*

$$m_1\ddot{x}_1 = -(k_1 + k_2)x_1 - (c_1 + c_2)\dot{x}_1 + k_2x_2 + c_2\dot{x}_2 \tag{3.102a}$$
$$m_2\ddot{x}_2 = -(k_3 + k_2)x_2 - (c_3 + c_2)\dot{x}_2 + k_2x_1 + c_2\dot{x}_1. \tag{3.102b}$$

Using, as before, the position and the velocity as output from the first sub-system,

$$m_1\ddot{x}_1 = -k_1x_1 - c_1\dot{x}_1 + k_2u_{11} + c_2u_{12} \tag{3.103a}$$
$$y_1 = [x_1, \dot{x}_1]^{\mathrm{T}}. \tag{3.103b}$$

However, the input is instead the difference in position and velocity between the two sub-systems. In the second sub-system, the output is the difference between the position and the velocity,

$$m_2\ddot{x}_2 = -(k_3 + k_2)x_2 - (c_3 + c_2)\dot{x}_2 + k_2u_{21} + c_2u_{22} \tag{3.104a}$$
$$y_2 = [x_2, \dot{x}_2]^{\mathrm{T}} - [u_{21}, u_{22}]^{\mathrm{T}} \tag{3.104b}$$

and the input for this second sub-system is the position and velocity of the first sub-system. So the coupling equations are the same as before, and thus the coupling equations are defined by,

$$u_{11} = y_{21}, \qquad u_{21} = y_{11} \tag{3.105a}$$
$$u_{12} = y_{22}, \qquad u_{22} = y_{12}. \tag{3.105b}$$

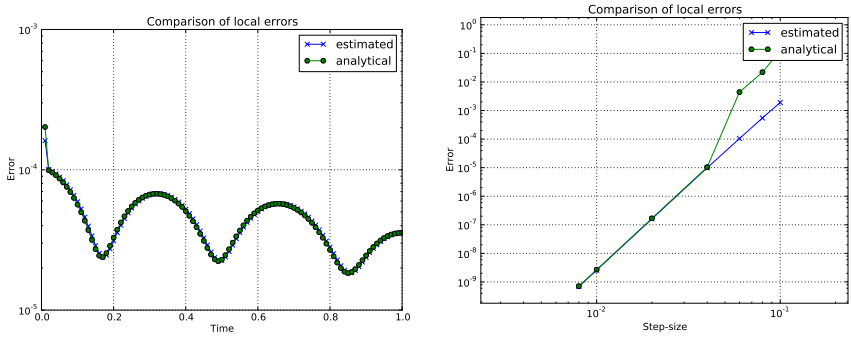

Figure 3.13: *Estimated local errors together with the analytical local error for linear extrapolation of Example 3.3.2. In the left figure, the local errors are shown for a simulation over one second for a given step-size of* $H = 0.01$*. In the right figure, the local errors are shown for various step-sizes.*

In Figure 3.15 the error estimate is shown together with the analytical local error when using Richardson extrapolation. A step-size of $H = 0.01$ *was used together with linear extrapolation. As can be seen from the figure, the estimate is slightly overestimating the error. Further investigation is needed as to find the cause of the discrepancy.*

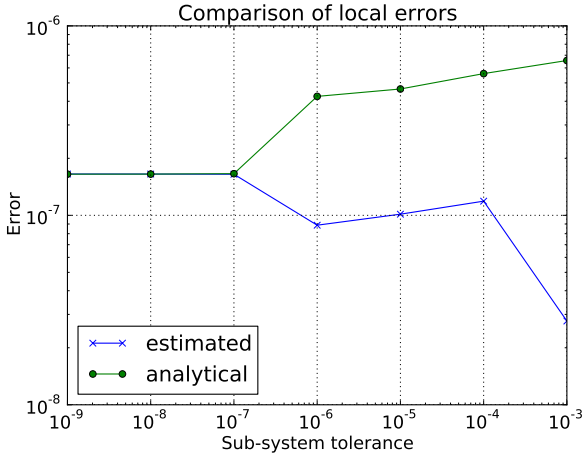

Figure 3.14: *Result of simulating Example 3.3.2 with constant extrapolation and using a step-size of $H = 0.05$ together with changing the tolerance of the sub-systems.*

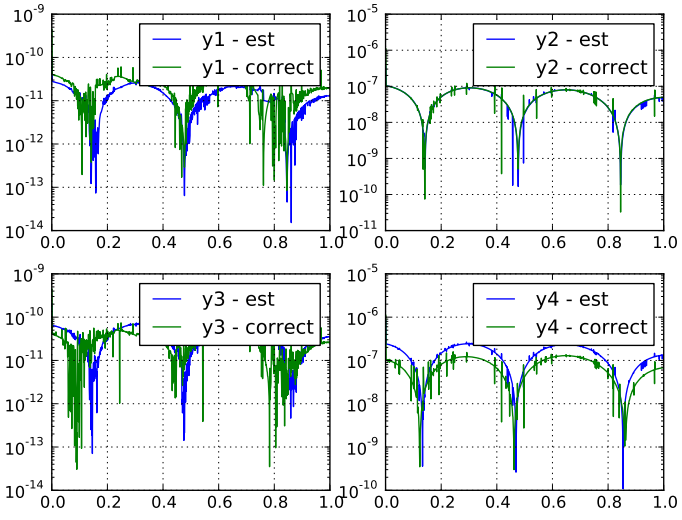

Figure 3.15: *Estimated local errors together with the analytical local error for linear extrapolation of Example 3.3.3 for all components.*

Chapter 4

Software Framework

This chapter describes the main contributions of this thesis. It describes a developed framework used for testing purposes and evaluations of co-simulation approaches. It describes a developed framework for evaluating ordinary differential equation (ODE) solvers, Assimulo and it describes a framework for working with models following the FMI standard, PyFMI.

The purpose of the framework is to evaluate various co-simulation approaches on both academic test examples and on models with industrial relevance. For the latter, we want to achieve the ability to import industrial models developed within closed industrial software tools into PyFMI and Assimulo for experimentation. Possibly together with the former class of models.

The following prerequisites is necessary for attaining our goals,

- *Access to industrial examples*
  By using the FMI standard, access to models from other tools supporting the standard is achieved as FMI is a project that aims at serving as a bridge between different tools and for exchanging models. The developed software PyFMI allows interaction and manipulation of FMUs.

- *Ability to investigate solver impact*
  The Assimulo package has been developed to provide a unified high-level interface to a number of state-of-the art solvers. Switching between them is an easy operation allowing investigation of the impact of the solver in sub-systems.

- *Access to an extensible master algorithm*
  A master algorithm has been developed with the ability to evaluate and test the various co-simulation approaches discussed in Chapter 3.

Looking at the requirements from an algorithmic point of view and from the discussion of co-simulation in Chapter 3, we find that there are a number of features that need to be supported both by the sub-systems and the master algorithm to be able to test and validate the co-simulation approaches. For the sub-systems, the ability to perform repeated evaluation of the outputs at a global time-step is necessary in order to find a consistent set of input signals in case of feed-through. Additionally the ability of storing the model and solver state is necessary in case a step is rejected by the master algorithm. For the master algorithm, the ability to handle an arbitrary number of sub-systems simulated using the parallel or serial approach is necessary together with handling signal interpolation and extrapolation of various degrees and additionally to perform error estimations. Support for these features has been implemented in the master algorithm.

Assimulo and PyFMI have been implemented in the Python programming language. Python is a high-level versatile programming language where readability and ease of use is of high importance. It is an object-oriented, powerful and dynamically typed language with a large supporting community. There are packages available for performing a large number of tasks and analyses. For scientific computing, the most notable are NumPy [34], SciPy [29] and Matplotlib [26], which combined offer similar functionality as MATLAB. An aspect that has contributed to Python's popularity is the ability to easily use external code written in, for example, FORTRAN or C. This is important as many algorithms are written in either of these languages. The execution speed of code written in Python is not comparable to code written in low-level languages. However, the ability to connect algorithms from external code and perform the bulk of the operations in the these algorithms reduce the performance hit. The performance hit is usually outweighed by the ease of which a program is created in Python.

PyFMI and Assimulo are necessary tools for the evaluation of co-simulation approaches as the former gives access to models following the FMI standard and thus models from a number of different tools. The latter gives access to ODE solvers which can be used together with FMUs following the model exchange standard to mimic a co-simulation FMU. The advantage with this approach is that the influence of the solver and solver options can be investigated even though the tool providing the FMU does

not support the particular solver. These two packages are also an integral part of the JModelica.org platform [37] and are both freely available at `http://www.jmodelica.org` as open source software. In Section 4.1, Assimulo is described and in Section 4.2, PyFMI is described. Section 4.3 describes the implemented master algorithm which combines PyFMI and Assimulo and allows co-simulation of FMUs using various approaches and options.

4.1 Assimulo

Assimulo [7] is a package for solving ordinary differential equations. The primary aim of Assimulo is not to develop new integration algorithms but to provide a high-level interface for a wide variety of solvers, both new and old and solvers of industrial standards as well as experimental solvers. 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 separates between a problem (or model), which contains the problem equations (in most cases the physics) 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, for instance the *right-hand-side* of the problem, but they may also contain event functions in order to support hybrid systems with state, step and time events. Additionally, a problem definition can specify options related to the problem such as which states are actually algebraic variables.

4.1.1 Problem Formulations

Various problem types are supported by Assimulo and can be used together with appropriate solvers. The most general formulation available is a fully implicit ordinary differential equation, also referred to as a differential–algebraic equation, DAE,

$$F(t, y, \dot{y}) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0. \tag{4.1}$$

Additionally, there is support for systems of explicit ordinary differential equations (ODE),

$$\dot{y} = f(t, y), \quad y(t_0) = y_0. \tag{4.2}$$

and mechanical systems in overdetermined implicit DAE form,

$$\dot{p} = v \tag{4.3a}$$
$$M(p)\dot{v} = f(t, p, v) - G^T(p)\lambda \tag{4.3b}$$
$$0 = g_{\text{constr}}(p) \tag{4.3c}$$
$$0 = G(p)v \tag{4.3d}$$

where p is the position and v the velocity. The constraints are given by g_{constr} and noticing that $\frac{\mathrm{d}g_{constr}}{\mathrm{d}p} = G$.

Problem formulations can also be transformed into a more general description, for instance, an explicit ODE can always be transformed into a fully implicit DAE. This is useful when comparing solvers written for DAEs with solvers written for ODEs.

ASSIMULO focuses on hybrid problems in such a way, that it helps to express them in a general sense and that it simplifies the handling of the events once they have been detected by the solver. Consider a problem that is defined as follows,

$$\dot{y} = f(t, y, sw), \quad y(t_0) = y_0 \tag{4.4a}$$
$$0 = g(t, y, sw) \tag{4.4b}$$

where f is the *right-hand-side* of an explicit ODE and g are the state event functions (root functions). Apart from time, t, and states, y, there is an extra argument, sw, which is a set of switches. They are input variables, which are not altered by the solver. Their purpose is to indicate which internal branch of f and g is to be evaluated. These switches are (re-)set once the integration has been stopped, due to an event detection, triggered by a sign change in one of the components of the vector valued function g. How these switches have to be set depends on the problem and has to be defined in the problem definition by providing a handle-event function.

4.1.2 Solvers

In the present state, ASSIMULO provides interfaces to production quality solvers like CVODE and IDA from the SUNDIALS [23] suite developed

at the 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 ODEs, $\dot{y} = f(t, y)$ or hybrid explicit ODEs, $\dot{y} = f(t, y, sw)$. A method flag allows to use for stiff problems BDF methods and for non-stiff problems Adams-Moulton methods. CVODE uses a variable-order and variable step-size algorithm. IDA, on the other hand solves the more general problem described as hybrid 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 in order to avoid corrector convergence failures.

One important purpose of Assimulo is to give the simulation and modeling engineer access to the wide spread flora of research integrators. A typical representative for this class of codes is GLIMDA [43] which is now accessible in Assimulo. GLIMDA is an implementation of general linear multistep methods to solve low index DAEs. These methods can be viewed as a blend of the collocation approach of implicit Runge–Kutta methods with the interpolation-based linear multistep methods. These techniques allows to adapt the methods coefficients to the special stability characteristics of the problem at hand. Assimulo's concept exposes this method class to a wide range of mechanical problems and helps this way to gain experience of this relatively new method class based on large and industrially relevant models.

The implicit Runge–Kutta method RADAU5 [21], shares stability properties with the implicit Euler method but promises higher accuracy due to its larger order. A classical implementation of this method by Hairer is interfaced with Assimulo. The solvers DOPRI5 [20] and RODAS [21], are additionally available for problems on the form $\dot{y} = f(t, y)$. The solvers are different Runge–Kutta type methods with variable step-size and where RADAU5 and RODAS are suitable for stiff problems.

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 provide a homogeneous handling of accessing a solver and using the

computed result. But Assimulo also supports contribution of experimental code directly written in Python. A constant step-size Runge–Kutta method, an explicit Euler method and an implicit Euler method, all implemented in Python, are included in Assimulo. Assimulo also aims to expose historically interesting codes together with modern industrial codes and more unknown research codes. Among the classical codes we name the solver LSODAR from ODEPACK [22] which is a multistep method that depending on the stiffness of the problem switches between an Adams method and a BDF method. Also ODASSL [18] is provided as a code specialized on mechanical systems described by the problem class of overdetermined DAEs. It is a variant of DASSL with the linear algebra part of the corrector iteration replaced by a special pseudo-inverse reflecting a transformation to state space form.

Most of the solvers mentioned do not natively support problems with discontinuities. Exceptions are CVode, IDA and LSODAR. This triggered the work in [16], where a general event localization scheme was implemented in Assimulo. The algorithm requires a continuous representation of the solution which in addition has been implemented for the solvers that do not provide one by default. The algorithm has been added to all solvers except GLIMDA and ODASSL and shown to locate the events accurately, despite the fact of order drops due to the use of the continuous representation.

An overview of the available solvers in Assimulo together with the problem formulations and their connections as-well as the connection to FMUs is given in Figure 4.1. Additionally in Table 4.1, a list of the solvers are given together with if they support hybrid problems or not. Solvers supporting hybrid problems and either ODE or DAE problems can be used together with FMUs.

4.1.3 Example - The Van der Pol Oscillator

This example is intended to show how Assimulo can be used to solve the Van der Pol oscillator, defined by Equation 3.49. The oscillator is again given here for convenience,

$$\dot{y}_1 = y_2 \tag{4.5a}$$

$$\dot{y}_2 = \mu((1 - y_1^2)y_2 - y_1) \tag{4.5b}$$

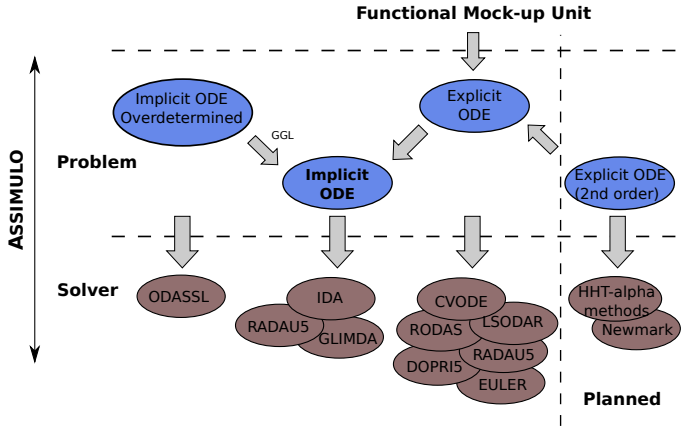

Figure 4.1: *The available solvers and problem formulations in* ASSIMULO *together with the planned additions. The arrows between the problem formulations indicate how the problem can be reformulated. The arrows between the problems and the solvers indicate which problem formulation the particular solver supports.*

with the initial conditions, $y_1(t_0) = 2$, $y_2(t_0) = -0.6$ together with $\mu = 200$.

In order to solve this problem, the *right-hand-side* needs to be defined as a Python function,

```
#Define the rhs
def rhs(t,y):
    my = 1.0/5.0e-3

    yd_0 = y[1]
    yd_1 = my*((1.0-y[0]**2)*y[1]-y[0])

    return array([yd_0,yd_1])
```

Once the *right-hand-side* is defined, the problem can be specified by creating an *Explicit Problem* using the *right-hand-side* function together with a set of initial conditions. The `Explicit_Problem` class needs additionally be imported from ASSIMULO.

```
from assimulo.problem import Explicit_Problem

y0 = [2.0,-0.6] #Initial conditions

exp_mod = Explicit_Problem(rhs,y0) #Define an Assimulo problem
```

Solver	Problem type	Hybrid support	Language
CVODE	ODE	Yes	C
IDA	DAE	Yes	C
DOPRI5	ODE	Yes	Fortran
LSODAR	ODE	Yes	Fortran
GLIMDA	DAE	No	Fortran
ODASSL	ODAE	No	Fortran
Explicit Euler	ODE	Yes	Python
Implicit Euler	ODE	Yes	Python
Runge-Kutta 34	ODE	Yes	Python
Runge-Kutta 4	ODE	No	Python
RODAS	ODE	Yes	Fortran
RADAU5	ODE/DAE	Yes	Fortran

Table 4.1: *The currently available solvers in* ASSIMULO *and the type of problems a solver support. Also shown is the language in which the solver is written together with if it supports hybrid models or not. Solvers supporting hybrid problems and either ODE or DAE problems can be used together with FMUs.*

Now that we have an explicit problem, a solver object can be created by using a solver from ASSIMULO. The various solvers have different options to be set. Here, a change of the relative tolerance is used to show how they are set.

```
from assimulo.solvers import CVode #Import the solver

#Define an explicit solver
exp_sim = CVode(exp_mod) #Create a CVode solver

#Sets the options
exp_sim.rtol = 1e-4 #The relative tolerance
```

The final step is the actual simulation which is performed by invoking the *simulate* method.

```
#Simulate
t, y = exp_sim.simulate(2.0) #Simulate 2 seconds
```

The returned vectors t and y contains the calculated result. In Figure 4.2, the result is visualized.

There are many more options and supported features than what this

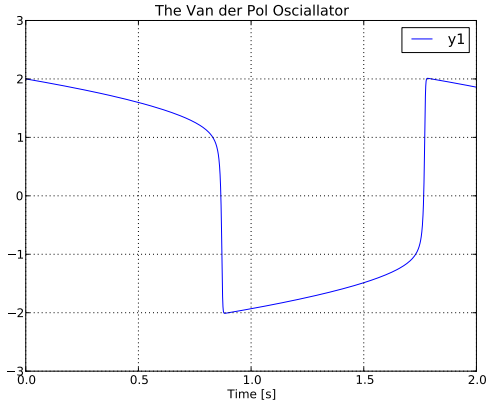

Figure 4.2: *Result of a simulation of the Van der Pol Oscillator, from Section 4.1.3, using the solver CVode.*

example shows. In Assimulo there are about 25 examples demonstrating the various features and may serve as starting points or reference for solving a particular problem.

4.2 PyFMI

PyFMI [6] is a Python package for interacting with models adherent with the FMI standard based on the FMI library. 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. There is support for both model exchange and co-simulation FMUs. In preparation for the new version of the FMI standard, support for the beta version has been implemented in PyFMI [33].

A model is imported into Python using just a few lines of code,

```
from pyfmi import load_fmu

model = load_fmu("Pendulum.fmu")
```

A simulation of an FMU, either a model exchange or a co-simulation FMU, is performed using the high-level `simulate` method,

```
res = model.simulate(final_time=10)
```

where the returned object, `res`, contains the calculated result. Additionally there are convenience functions available for accessing and working with the model data.

A key feature is the connection to Assimulo, Section 4.1, which provides capabilities for performing simulations of model exchange FMUs using ODE solvers interfaced with Assimulo. Another feature is the capability of writing result files and visualize the result. See Figure 4.3 for an overview. The result file format used is readable by multiple tools such as the leading Modelica tool, Dymola [42].

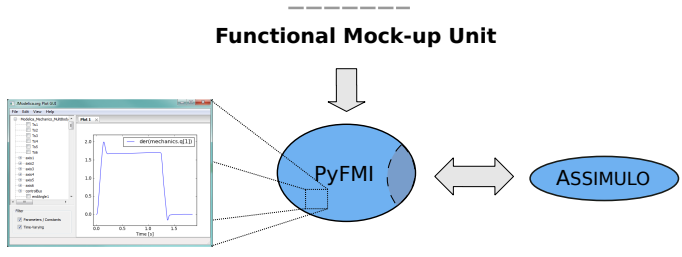

Figure 4.3: *The connection between PyFMI, the Functional Mock-up Interface and* Assimulo.

4.2.1 Interacting with a Model

The starting point for interacting with an FMU in PyFMI is the `load_fmu` method.

```
def load_fmu(fmu, path = '.', enable_logging = True,
                  log_file_name = "", kind = 'auto'):
```

The method is a factory method meaning that depending on the provided input, i.e. the FMU, it returns an instance of an appropriate class. The class returned is one of four possible,

- `FMUModelME1` - For models following FMI 1.0 for model exchange
- `FMUModelCS1` - For models following FMI 1.0 for co-simulation
- `FMUModelME2` - For models following FMI 2.0 for model exchange
- `FMUModelCS2` - For models following FMI 2.0 for co-simulation

The other arguments are for specifying where the FMU is located, if logging should be turn on or not, and if true, the name of log file. In FMI 1.0, an FMU is either a model exchange or a co-simulation, however in FMI 2.0 an FMU can be both and for those cases the final argument to `load_fmu` specifies the type of model class to be returned. In Figure 4.4, the hierarchy of the model classes are shown together with their parents which collects common functionality.

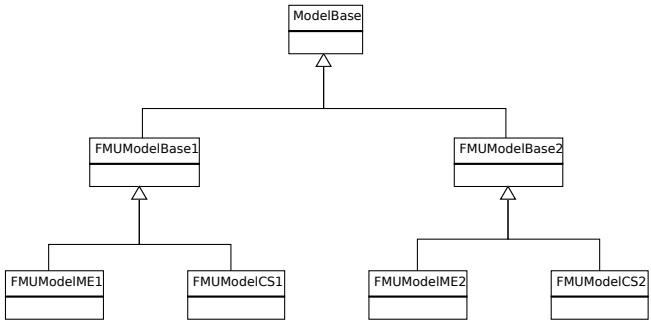

Figure 4.4: *The model class hierarchy in PyFMI.*

During the loading process, the object file is connected to the model class and the model metadata is read from the XML file. Once the process has finished, the model is instantiated and ready for initialization and simulation. Prior to the initialization, variable and parameter values can be set to the model for those with a defined start value in the metadata. Values are set using the `set` method.

```
model.set("gravity", 10)
```

The name of the variables are not defined in the object file, however the information can be retrieved from the metadata. A useful method in PyFMI is the `get_model_variables` which retrieves these names and information about them.

```
vars = model.get_model_variables()
```

The returned dictionary, `vars`, contain all the variables, parameters and constants defined in the model. The method additionally allows for filtering on which information should be returned. For instance, only variables that

are not an aliased variable, only variables that are inputs, only variables that are of type real and only variables whose name contains the string chassis. There are additionally methods for retrieving all information provided in the metadata.

In addition to the high-level methods, the low-level methods defined in the FMI standard is provided. In certain cases it might be more efficient to directly use these methods. For example, the `set_real` method is provided.

```
model.set_real([432,423,122], [-1.0, 2.0, 4.0])
```

The first list specifies the value references, i.e. the identification of the variables to be set. The second list are the actual values to be set. Using these methods directly one need to be particularly careful in case of aliased variables.

4.2.2 Simulating a Model

Simulation of a loaded model, be it a model exchange or a co-simulation, is performed using the `simulate` method.

```
res = model.simulate(start_time='Default', final_time='Default',
            input=(), algorithm='AssimuloFMIAlg', options={})
```

The only difference between simulating a model exchange FMU or a co-simulation FMU using this method is the default algorithm. For model exchange it is `AssimuloFMIAlg` and for co-simulation it is `FMICSAlg`. By default, the method initializes the model by calling the underlying FMI method for initialization and then performs a simulation using default values. The default values are retrieved from the metadata, i.e. information about the start time and the final time together with the tolerance.

Inputs can be provided to the model using the `input` argument in the `simulate` method. The input defines the input trajectories to the model and can be specified in one of two ways. Either as a data matrix where the values are interpolated linearly or as a general function. In both cases, a tuple is provided where the first index contain the variable names of the input variables and the second index the data or a function, see below.

```
#In case of input data
input_object = (['u1','u2'],input_data)
#In case of an input function
input_object = (['u1','u2'],input_function)
```

For the data matrix, the requirement is that the first column contain the time vector and the following columns correspond to the input variables. In case of a function, the argument is required to be the time and the output correspond to the input variables.

The `option` argument in the `simulate` method allows for specifying general options for the simulation, such as how the result is to be handled, specifying the solver and solver options in case of a model exchange FMU. Options for an algorithm is retrieved using the `simulate_options` method.

```
opts = model.simulate_options(algorithm='AssimuloFMIAlg')
```

The returned argument, `opts`, is a modified dictionary containing the available options for that particular algorithm. The modification allows for providing a description of the object which is helpful when using Python interactively. Changing the options, for example the solver, in case of a model exchange FMU, and the result file name is performed using the below code.

```
#Change from the default solver to Explicit Euler
opts["solver"] = "ExplicitEuler"
#Change the result file name
opts["result_file_name"] = "vdp_result.txt"
```

In Figure 4.5, the class hierarchy for both the algorithms the options are shown.

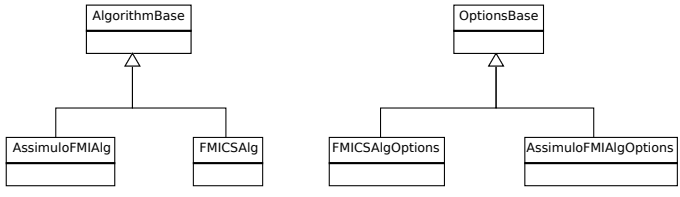

Figure 4.5: *In the left figure, the class hierarchy is shown for the available algorithms in PyFMI. In the right figure, the class hierarchy for the algorithm options are shown.*

By default, during a simulation, the result is continuously written to file. After a simulation has been successful, the result is read and returned.

```
res = model.simulate()
```

The object `res` contain primarily the resulting solution trajectories which are easily accessible for further computations or visualizations.

```
t = res["time"] #Retrieve the time vector
x1 = res["x1"]  #Retrieve the trajectory for the variable x1
```

Using the options, the variables of interest can be specified such that result are only stored for those variables and additionally if they should be stored directly in memory.

4.3 Master Algorithm

This section describes the implemented master algorithm. The algorithm implements the various approaches used for co-simulation and will serve as a basis for future evaluation of new approaches and new ideas.

The implementation supports simulation of a coupled system via a parallel approach and a staggered approach. Interpolation or extrapolation using constant polynomials and up to cubic polynomials may be used together with an error estimation based on Richardson extrapolation. Iteration on the output variables is also implemented. In other words, the approaches discussed in Chapter 3, are implemented.

4.3.1 Extended Model Definition

In order to evaluate the influence of the underlying ODE solver in an co-simulation FMU and the impact of the options provided to the solver, such as the tolerances, we need access to the solver. We need also the ability to change the underlying solver. In the FMI specification, there is currently no way of providing options to the underlying solver or changing the solver as these are statically compiled in when it is created. A way of *mimicking* a co-simulation FMU is instead to use an FMU for model exchange and dynamically bundling it together with an integrator. A way of achieving

Figure 4.6: *By bundling* Assimulo *together with an FMU for model exchange an co-simulation FMU is created.*

this is to extend PyFMI's interface for an model exchange FMU and implemented the necessary methods, i.e. the coupling to an integrator, using ASSIMULO to provide the API of a co-simulation FMU, see Figure 4.6. This allows full access to the solver with the ability to investigate the impact of different solvers and options. In addition, as the solver is exposed, the ability to store the solver state is available for those solvers that support the feature as discussed in Section 3.3.1.1.

The crucial methods to implement is the initialization method, where an ASSIMULO integrator is setup, and the method for performing a global step, using the integrator. The methods are called `fmi_initialize_slave` and `fmi_do_step` in the FMI specification. Once these has been defined one can easily pick a model exchange FMU, load it into the framework, and the output is a co-simulation FMU with the added benefit of allowing solver options to be set and the solver to be changed. An additional benefit is that the interpolation/extrapolation is done outside of the FMU allowing a user the freedom to specify a different representation than what is defined in the specification for co-simulation. The default behavior is although the same as in the specification,

$$u(t) = u(T_n) + \sum_{i=1}^{k} \frac{1}{i!} \frac{d^i u}{dt^i} (t - T_n)^i, \quad t \in [T_n, T_{n+1}] \tag{4.6}$$

where the derivatives are approximated using backward differences. For the kth derivative at T_n we have,

$$\frac{d^k u_n}{dt^k} = \frac{1}{\prod_{m=1}^{k} (t_{n-m+1} - t_{n-m})} \sum_{m=0}^{k} \binom{k}{m} (-1)^m u_{n-m}. \tag{4.7}$$

4.3.2 Approaches

The approaches supported in the master algorithm is the parallel, i.e where all the sub-systems are simulated in parallel over the same global integration step and all input signals are extrapolated, see Algorithm 1. The extrapolation is possible using either constant, linear, quadratic or cubic polynomials. Additionally a staggered approach may be used where the sub-systems are integrated in the order provided to the master algorithm. The signals are either interpolated, if the sub-system has already been solved for the current global integration step, or extrapolated if it has not been solved, see 2. The degree of the polynomial can be set to the same as in the Jacobi case.

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

Require: The models, the number of models N, the extrapolation order K.

for $i = 1$ to N **do**
  Set the ith model input, $u_n^{[i]}$.
  if $K > 0$, i.e. if using higher order extrapolation **then**
    for $k = 1$ to K **do**
      Set the kth input derivative to the ith model, $\frac{\mathrm{d}^k u_n^{[i]}}{\mathrm{d}t^k}$.
    end for
  end if
  Perform global time step, $T_n \rightarrow T_{n+1}$ for the ith model.
end for
for $i = 1$ to N **do**
  Retrieve model outputs, $y_{n+1}^{[i]}$.
end for

4.3.3 Error Estimations

Based on the method presented in [41], an error estimation procedure has been implemented. The estimate is based on performing a global integration step twice using different input, see Section 3.3.2. A first step is performed using a step-size H. This step is compared with two steps of a step-size $H/2$ where inputs and outputs between the sub-systems are updated before taking the second step of step-size $H/2$. Error rejection is however difficult as FMI 1.0 does not support a method for storing the internal state at a previous time point. Using the extended model definition as described in Section 4.3.1 one can however store the *solver* state, which for a model exchange FMU without discontinuities is enough and step rejections are possible. This is however not the entire truth as an model exchange FMU may contain internal iteration variables that are not exposed and may cause problems, but for "simple" FMUs, it is enough. In FMI 2.0, there is capabilities for storing the full model state, including the solver, which will remove this limitation.

4.3.4 Interface

The master algorithm is implemented in the Python programming language and contained in a package called `master`. The main implementation and

Algorithm 2 Staggered step, $(T_n \to T_{n+1})$

Require: The models, the number of models N, the extrapolation order K.

for $i = 1$ to N **do**
  if inputs are from a model $< i$ **then**
    Set the ith model input, $u_{n+1}^{[i]}$.
    if $K > 0$, i.e. if using higher order extrapolation **then**
      for $k = 1$ to K **do**
        Set the kth input derivative to the ith model, $\frac{\mathrm{d}^k u_{n+1}^{[i]}}{\mathrm{d}t^k}$.
      end for
    end if
  else
    Set the ith model input, $u_n^{[i]}$.
    if $K > 0$, i.e. if using higher order extrapolation **then**
      for $k = 1$ to K **do**
        Set the kth input derivative to the ith model, $\frac{\mathrm{d}^k u_n^{[i]}}{\mathrm{d}t^k}$.
      end for
    end if
  end if
  Perform global time step, $T_n \to T_{n+1}$ for the ith model.
  Retrieve model outputs, $y_{n+1}^{[i]}$.
end for

the user entry-point is the `Master` class which needs to be imported from the package.

```
from master import Master
```

The interface supports both models described by the FMI CS standard, see Section 2.1, and models described by the FMI ME standard with the additional coupling to Assimulo, see Section 2.1 and 4.3.1. The CS FMUs can be loaded into Python via PyFMI, see Section 4.2, and the ME FMUs are loaded using the extended model definition available in the package.

```
from pyfmi.fmi import FMUModelCS1 #For CS FMUs
from master import FMUModelME1Extended #For ME FMUs
```

For use in the `Master` class, the FMUs need to be loaded into a list.

```
models = [FMUModelCS1("Subsystem1"), FMUModelCS1("Subsystem2")]
```

This list may contain an arbitrary number of FMUs and a mix between the variants. If the Jacobi approach is used for the overall simulation, the ordering in the list is irrelevant. However, in the case of the Gauss-Seidel approach, the ordering influence which model is simulated first, second and so forth.

In order to simulate a coupled system, the coupling needs to specified. Here the following convention is used. First, from which model is the variable data coming from? It should be an index mapping to the model list, starting from zero. Second, the name of the variable in the model where data is coming from. Thirdly, the index of the receiving model and finally the name of the receiving variable.

```
connections = [(var_from_ind,"x_chassi",var_to_ind,"x_chassi"),
               (var_from_ind,"v_chassi",var_to_ind,"v_chassi")]
```

The connection list can contain an arbitrary number of connections.

The models together with the their connections can then be loaded into the `Master` class,

```
master_simulator = Master(models, connections)
```

and simulated using the `simulate` method.

```
res = master_simulator.simulate(start_time=0.0, final_time=1.0)
```

The available options to the master algorithm is listed below.

set_method
Specifies which approach to use in the simulation. Available methods are the Jacobi and the Gauss-Seidel.

```
master_simulator.set_method("JACOBI")
```

set_step_size
Specifies the global step-size to be used in the simulation in case that a fixed step-size method is used.

```
master_simulator.set_step_size(0.01)
```

set_extrapolation_order
Specifies the extrapolation order to be used in the simulation.

```
master_simulator.set_extrapolation_order(2)
```

use_richardson
Specifies if Richardson extrapolation, see Section 3.3.2, should be used to produce an error estimation for which to base the step-size on.

```
master_simulator.use_richardson(True)
```

set_atol
Specifies the absolute tolerance to be used in case of step-size control, i.e if error estimation has been activated. The tolerance is used to control the step-size according to Equation 3.72.

```
master_simulator.set_atol(1e-4)
```

set_rtol
Specifies the relative tolerance to be used in case of step-size control, i.e if error estimation has been activated. The tolerance is used to control the step-size according to Equation 3.72.

```
master_simulator.set_rtol(1e-4)
```

Chapter 5

Experiments

This chapter applies the software described in Chapter 4 on examples. The intention is to demonstrate the capabilities of the software. The elapsed simulation time is not shown in the examples as the focus, in this thesis, is not on the performance. The focus is rather on the feature set and the capabilities allowing for experimentation regarding co-simulation.

5.1 The Woodpecker

This example is intended to show how Assimulo can be used to solve hybrid systems. The problem is that of a toy woodpecker, originally presented in [32]. The difference is that in our case we consider impacts without friction. 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, see Figure 5.1.

The model can be in either one of three states. In the first state, there is no blocking and the sleeve is free to move along the bar. In the second state, the sleeve is fixed with the bar at the upper left corner and the lower right corner of the sleeve. Finally in the third state, the sleeve is also blocked against the bar at the upper right corner and the lower left corner. The

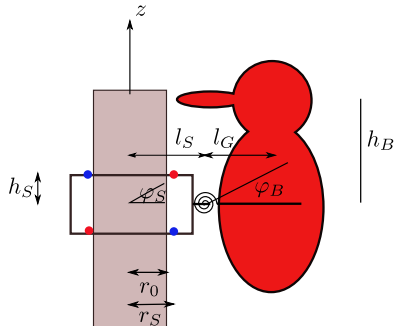

Figure 5.1: *Schematic figure of the woodpecker.*

linearized equations of motion are for the first state given by,

$$f_1(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = -(m_S + m_B)g \tag{5.1a}$$
$$f_2(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_B - \varphi_S) - m_B l_S g - \lambda_1 \tag{5.1b}$$
$$f_3(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_S - \varphi_B) - m_B l_G g - \lambda_2 \tag{5.1c}$$
$$0 = \lambda_1 \tag{5.1d}$$
$$0 = \lambda_2 \tag{5.1e}$$

where,

$$f_1(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = (m_S + m_B)\ddot{z} + m_B l_S \ddot{\varphi}_S + m_B l_G \ddot{\varphi}_B \tag{5.2a}$$
$$f_2(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = (m_B l_S)\ddot{z} + (J_S + m_B l_S^2)\ddot{\varphi}_S + (m_B l_S l_G)\ddot{\varphi}_B \tag{5.2b}$$
$$f_3(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = (m_B l_G)\ddot{z} + (m_B l_S l_G)\ddot{\varphi}_S + (J_B + m_B l_G^2)\ddot{\varphi}_B. \tag{5.2c}$$

For the second state, the equations of motion are given by,

$$f_1(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = -(m_S + m_B)g \tag{5.3a}$$
$$f_2(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_B - \varphi_S) - m_B l_S g - h_S \lambda_1 - r_S \lambda_2 \tag{5.3b}$$
$$f_3(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_S - \varphi_B) - m_B l_G g \tag{5.3c}$$
$$0 = h_S \ddot{\varphi}_S \tag{5.3d}$$
$$0 = \dot{z} + r_S \dot{\varphi}_S. \tag{5.3e}$$

$m_S = 3.0\text{E-}4$ 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			

Table 5.1: *Parameters used in the woodpecker example, see Section 5.1.*

Finally for the third state, the equations of motion are given by,

$$f_1(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = -(m_S + m_B)g - \lambda_2 \tag{5.4a}$$
$$f_2(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_B - \varphi_S) - m_B l_S g + h_S\lambda_1 - r_S\lambda_2 \tag{5.4b}$$
$$f_3(\ddot{z}, \ddot{\varphi}_S, \ddot{\varphi}_B) = c_p(\varphi_S - \varphi_B) - m_B l_G g \tag{5.4c}$$
$$0 = -h_S\ddot{\varphi}_S \tag{5.4d}$$
$$0 = \dot{z} + r_S\dot{\varphi}_S. \tag{5.4e}$$

The parameter values are defined in Table 5.1.

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

$$h_S\varphi_S > r_S - r_0 \quad \text{and} \quad \dot{\varphi}_B > 0 \tag{5.5}$$

or when

$$h_S\varphi_S < -(r_S - r_0) \quad \text{and} \quad \dot{\varphi}_B < 0. \tag{5.6}$$

In the first case, the state is changed to the third state and in the second case the state is changed to the second state. Changing to the first state from the second or the third state occurs when,

$$\lambda = 0. \tag{5.7}$$

There is one additional state change that can occur, namely when the woodpecker hits the bar. However, after the impact, the state is directly changed to the previous state. The woodpecker hits the bar when,

$$h_B\varphi_B > l_S + l_G - l_B - r_0. \tag{5.8}$$

This leads to four event indicators (or root functions) that needs to be monitored during the integration. The event indicators are given by,

$$g_1 = h_S\varphi_S + (r_S - r_0) \tag{5.9a}$$
$$g_2 = h_S\varphi_S - (r_S - r_0) \tag{5.9b}$$
$$g_3 = \lambda_1 \tag{5.9c}$$
$$g_4 = h_B\varphi_B - l_S - l_G + l_B + r_0. \tag{5.9d}$$

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,

$$I^- = m_B l_G \dot{z}^- + (m_B l_S l_G)\dot{\varphi}_S^- + (J_B + m_B l_G)\dot{\varphi}_B^- \tag{5.10}$$

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 $\dot{\varphi}_B^+$ using that,

$$I^+ = m_B l_G \dot{z}^+ + (m_B l_S l_G)\dot{\varphi}_S^+ + (J_B + m_B l_G)\dot{\varphi}_B^+. \tag{5.11}$$

For the case when the woodpecker hits the bar, the angular velocity of the woodpecker, $\dot{\varphi}_B$, changes sign.

Simulating the model using ASSIMULO requires that first the problem class and the solver class is imported into Python.

```
from assimulo.problem import Implicit_Problem
from assimulo.solvers import IDA
```

the model is a hybrid DAE and in this case we imported the solver IDA for performing the simulation. The residual is defined in a method `res` using Equation 5.1, 5.3 and 5.4,

```
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

    res_01 = y[3]-yd[0]
    res_02 = y[4]-yd[1]
    res_03 = y[5]-yd[2]
```

```
    if sw[0]: #State 1
        res_1 = pre1+(mS+mB)*g
        res_2 = pre2-cP*(phiB-phiS)+mB*lS*g+lam1
        res_3 = pre3-cP*(phiS-phiB)+mB*lS*g+lam2
        res_4 = lam1
        res_5 = lam2

    if sw[1]: #State 2
        res_1 = pre1+(mS+mB)*g+lam2
        res_2 = pre2-cP*(phiB-phiS)+mB*lS*g+hS*lam1+rS*lam2
        res_3 = pre3-cP*(phiS-phiB)+mB*lG*g
        res_4 = hS*phiSpp
        res_5 = zp+rS*phiSp

    if sw[2]: #State 3
        res_1 = pre1+(mS+mB)*g+lam2
        res_2 = pre2-cP*(phiB-phiS)+mB*lS*g-hS*lam1+rS*lam2
        res_3 = pre3-cP*(phiS-phiB)+mB*lG*g
        res_4 = -hS*phiSpp
        res_5 = zp+rS*phiSp

    return N.array([res_01,res_02,res_03,res_1,res_2,res_3,res_4
        ,res_5])
```

The argument *sw* are the switches which are kept constant during the integration and only changed at an event. The event indicators, Equations 5.9, 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 handle 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 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 start in the second state.

```
y0  = [0.0, -0.1, -0.65, 0.0 ,0.0 ,0.0 ,0.0 , 0.0]
yd0 = [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,y0,yd0,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 which variables are the states and which are the 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 state (1) and the algebraic (0) variables
woody.algvar = [1]*6+[0]*2
```

A simulation is finally performed using the `simulate` method, where we have additionally specified that the algebraic variables is to be excluded from the error test.

```
sim = IDA(woody)

#Specify simulation options
sim.suppress_alg = True #Suppress the algebraic variables
                        #from the error test.

t,y,yd = sim.simulate(0.16)
```

The computed solution trajectories are returned and stored in t, y and yd.

In Figure 5.2, the result is shown for when simulating the woodpecker 0.16 seconds and 1.0 second using the solver IDA. The simulations were performed with the default tolerance 10^{-6} for the absolute and the relative tolerance. For the first simulation, the woodpecker hit the bar two times and for the second simulation 11 times.

In this example is has been shown that Assimulo is able to handle hybrid DAEs. The flexibility in defining events and how to handle them are additionally highlighted.

5.2 The Monolithic 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

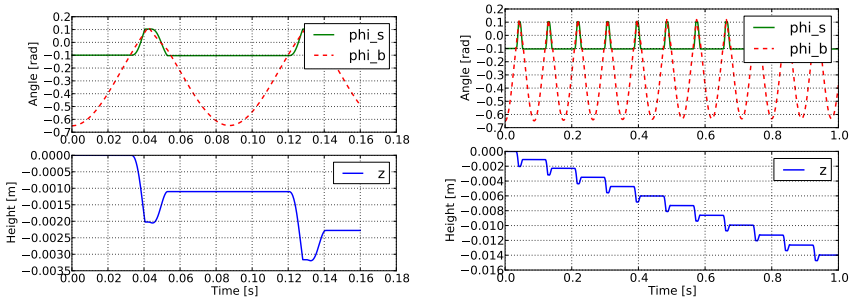

Figure 5.2: *Simulation results for when simulating the woodpecker from Section 5.1. The model was simulated using the solver IDA for* 0.16 *seconds (left figure) and* 1.0 *seconds (right figure). The tolerances was set to* 10^{-6} *for both the absolute and the relative tolerance.*

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. To investigate the dynamic response, simulations can be carried out with predefined input or by a feedback loop using either a simulator or a virtual driver model.

In this example, a race car is modeled in Dymola and exported to an model exchange FMU. 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.

This example is intended as a demonstration that PyFMI together with Assimulo is able to simulate large complex models. The model contains about 90k parameters, constants and variables resulting in that the XML data file is 700k lines. There are 47 continuous states and 44 event indicators.

The FMU is imported into Python, by means of PyFMI, and made available for simulation using the integrators in Assimulo. A simulation is performed by creating a Python script which imports the necessary packages and then load the FMU into a Python object. Parameters and solver options are in turn changed with the available high-level interface. Once the options have been specified, if any, a call to `simulate` performs the simulation.

Finally, the result can be retrieved and visualized using either the graphical user interface or directly using the Matplotlib package. Below, a Python script for a simulation of the race car is shown.

```
from pyfmi import load_fmu

model = load_fmu("FormulaStudent_Eight.fmu")

#Change the initial position of the steering wheel
model.set("steeringInEight.left_turn", -1.0)

#Change the number of result points (ncp)
opts = model.simulate_options()
opts["ncp"] = 1000

#Simulate the model with the specified options
res = model.simulate(final_time=30, options=opts)
```

The model is simulated using the solver CVODE in Assimulo for 30 seconds. In Figure 5.3, the resulting angle of the steering wheel is shown together with the position of the race car.

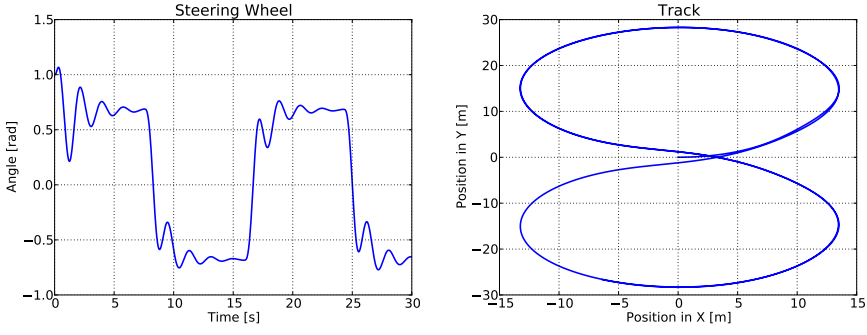

Figure 5.3: *Results from a simulation of the race car, in Section 5.2, driving on an eight shaped course. The left figure shows the angle of the steering wheel while the right figure shows the position of the race car. The model was generated as an FMU from Dymola and simulated in* Assimulo *using the solver CVODE.*

The result can easily be compared with simulations of the original model in Dymola and with different integrators connected in Assimulo. In Figure 5.4, a comparison is made of the angle of the steering wheel when using two different integrators in Assimulo, CVODE and LSODAR, versus a

simulation with the integrator DASSL on the original model in Dymola. The tolerances was set to 10^{-6}, both for the relative and the absolute tolerance for the ASSIMULO integrators while in DASSL, 10^{-10}. As can be seen from the figures, both CVODE and LSODAR produces comparable results with DASSL.

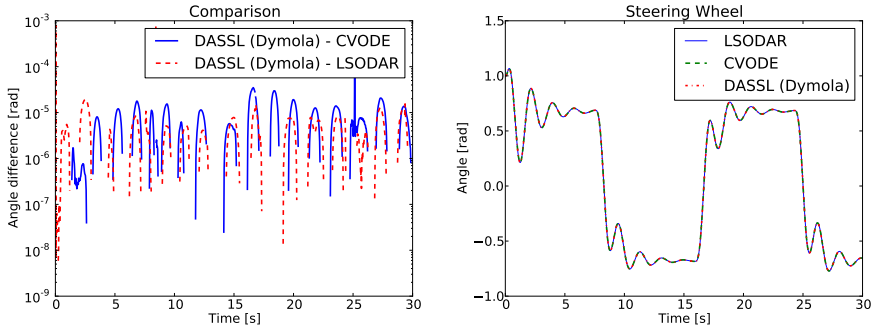

Figure 5.4: *Comparison between a simulation of the original model of the race car, in Section 5.2, in Dymola using DASSL with simulations of an FMU of the race car using the integrators LSODAR and CVODE from* ASSIMULO. *The left figure shows the difference of the steering wheel angle between the different integrators while the right figure shows the actual angle calculated using the different integrators.*

The example demonstrates the potential of the presented software to connect industrial models from acknowledged modeling software to a wide range of ODE integrators.

5.3 Quarter Car

In this example, a quarter car, see Figure 5.5, is simulated with step-size control. In a co-simulation setup, this example was discussed in [41]. The quarter car is governed by the equations,

$$m_c\ddot{x}_c = k_c(x_w - x_c) + d_c(\dot{x}_w - \dot{x}_c) \tag{5.12a}$$

$$m_w\ddot{x}_w = k_w(0.1 - x_w) + k_c(x_w - x_c) + d_c(\dot{x}_w - \dot{x}_c) \tag{5.12b}$$

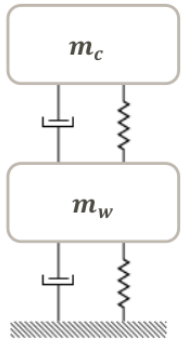

Figure 5.5: *The quarter car from Section 5.3.*

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 sub-system and the wheel another. The coupling is given by

$$y = I \begin{bmatrix} x_c \\ \dot{x}_c \\ x_w \\ \dot{x}_w \end{bmatrix} \tag{5.13a}$$

$$u = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} y \tag{5.13b}$$

where there is no direct feed-through.

FMUs of the sub-systems was generated using JModelica.org 1.10 as model exchange FMUs. The intention is to use the extended model definition of a model exchange FMU, as discussed in Section 4.3.1. This due to that we need to store the solver state in order to repeat a global step which is needed in the step-size control.

To use the implemented master algorithm described in Section 4.3 to simulate the coupled system the extended model definition and the algorithm itself needs to be imported.

```
from master import Master
from master import FMUModelME1Extended
```

The FMUs are then loaded into Python.

```
#Load the FMUs using the extended model definition
model_wheel = FMUModelME1Extended(fmu_wheel)
model_chassi = FMUModelME1Extended(fmu_chassi)
```

The coupling is specified by a connection matrix where the first number specifies the index in the models list from where the output should be retrieved from. The second part specifies to what sub-system the values should be provided and to which variable.

```
#Specify the coupling
connections = [(0,"x_chassi",1,"x_chassi"),
               (0,"v_chassi",1,"v_chassi"),
               (1,"x_wheel",0,"x_wheel"),
               (1,"v_wheel",0,"v_wheel")]
```

The next step is to load the master algorithm and specify optional options.

```
models = [model_chassi, model_wheel]

#Load the models into the master algorithm
master_simulator = Master(models, connections)

#Set the option to use error estimation based
#on Richardson extrapolation
master_simulator.use_richardson(True)
master_simulator.set_atol(1e-4)
master_simulator.set_rtol(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 5.6 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 the same version of JModelica.org and simulated using PyFMI/Assimulo. In Figure

Extrapolation order	0	1	2
Number of global steps	300	92	71
Number of error test failures	4	1	5

Table 5.2: *Simulation statistics for when simulating the Quarter Car in Section 5.3 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}.

5.7 the estimated error is shown together with the global step-size and the time points where a step rejection occurred.

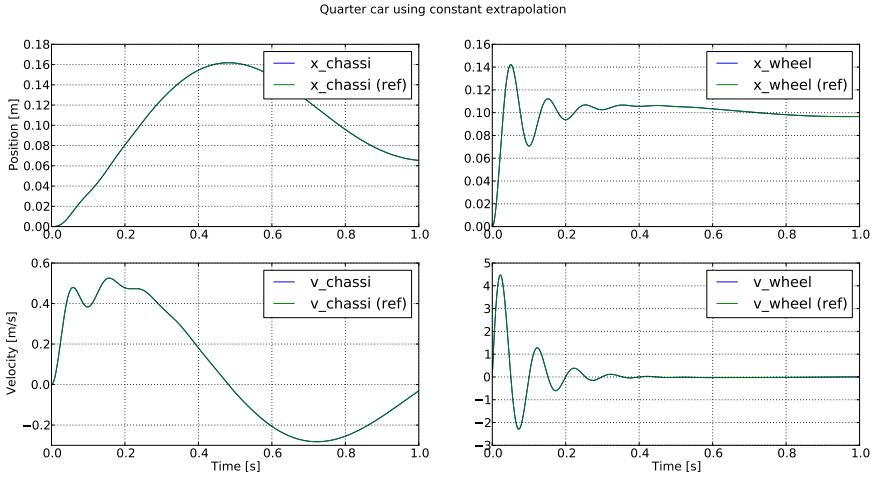

Figure 5.6: *The velocity and the position of the quarter car from Section 5.3 together with the reference solution.*

Simulations using higher order extrapolation was additionally carried out to investigate the influence of the extrapolation order on the number of steps. In Table 5.2, 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.

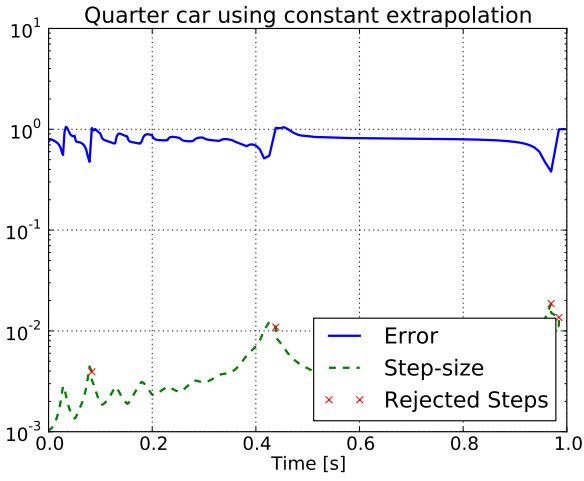

Figure 5.7: *The normalized estimated error together with the global step-size and the time points for where a step was rejected when simulating the quarter car from Section 5.3. The simulation was carried out using constant extrapolation together with a relative and an absolute tolerance of* 10^{-4}.

The example shows that we are able to reproduce the results in [41] using the developed tools.

5.4 Race Car

Revisiting the race car from Section 5.2, where we had a model of a race car driven by a virtual driver, see Figure 5.8. The aim of the simulation is to investigate the dynamic response of the car when the driver tries to stay onto an eight shaped course with increasing velocity. In Section 5.2, simulations was carried out on the monolithic model, in this section, the model will be simulated in a co-simulation setting. The race car and the driver is divided into five separate sub-systems, the chassis and the driver consisting of one sub-system and each of the wheels as one sub-system. The chassis and the driver sub-system is from now on referred to as the chassis sub-system. The separation of the system is a valid approach considering that it is not uncommon that the wheel-models are provided externally.

The chassis does not contain any direct feed-through terms. However,

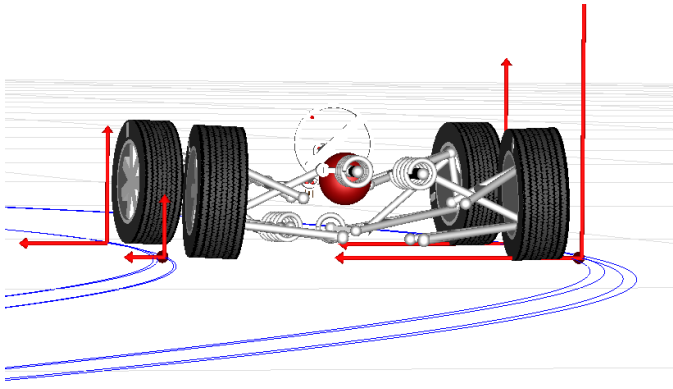

Figure 5.8: *Visualization of the race car in Section 5.4.*

there is direct feed-through in the wheels. The chassis contain 37 continuous states and the wheels 3 each.

The full system model was modeled in Dymola 2014 and exported as an model exchange FMU. The reference solution was calculated using the solver CVode from Assimulo with a relative tolerance of 10^{-8}. The sub-systems was additionally generated from Dymola, both as co-simulation FMUs and model exchange FMUs, where the wheels are represented by the same FMU.

Simulating the coupled system requires that the models are loaded into Python. This is performed by the following statements,

```
from pyfmi import load_fmu

model_chassi = load_fmu("Chassis.fmu")
model_wheel_rf = load_fmu("TyreForcesSlick.fmu")
model_wheel_lf = load_fmu("TyreForcesSlick.fmu")
model_wheel_rb = load_fmu("TyreForcesSlick.fmu")
model_wheel_lb = load_fmu("TyreForcesSlick.fmu")
```

This loads the co-simulation FMUs into Python. If instead, an extended model exchange FMU is to be used, the `load_fmu` method should be replaced by `FMUModelME1Extended`. The second step is to specify the connection between the sub-systems, there is a total of 172 connections. For each wheel, there are six outputs connecting to the chassis and 37 inputs from the chassis. Using the method `get_model_variables` and filtering on the

inputs and outputs together with a name filter, the connections can easily be created. The method defined below is used as a helper method as the connections are the same for all wheels, except for a number representing where the wheel is connected to the chassis.

```
def create_connections(m_c, m_w, wheel_number):

    #Get the inputs and outputs from the wheel
    vars_in_w = m_w.get_model_variables(causality=FMI_INPUT,
        include_alias=False)
    vars_out_w = m_w.get_model_variables(causality=FMI_OUTPUT,
        include_alias=False)

    #Get the inputs and outputs from the chassi
    vars_in_c = m_c.get_model_variables(causality=FMI_INPUT,
        include_alias=False, filter="?"+str(wheel_number)+"???")
    vars_out_c = m_c.get_model_variables(causality=FMI_OUTPUT,
        include_alias=False, filter=["*Frame"+str(wheel_number)+
            "*","*Velocity"+str(wheel_number)])

    #Create the connection list
    connection_list = []
    for var in vars_out_w.keys():
        connection_list.append((wheel_number,var,0,var.replace("
            1",str(wheel_number),1)))
    for var in vars_out_c:
        connection_list.append((0,var,wheel_number,var.replace(
            str(wheel_number),"",1)))

    return connection_list
```

The actual connection list is created calling this method for each wheel.

```
connections = []
connections.extend(create_connections(model_chassi,
    model_wheel_rf,1))
connections.extend(create_connections(model_chassi,
    model_wheel_lf,2))
connections.extend(create_connections(model_chassi,
    model_wheel_rb,3))
connections.extend(create_connections(model_chassi,
    model_wheel_lb,4))
```

The number represents the position of the wheel, number 1 represents right front, 2 left front, 3 right back and number 4 represents the left back wheel.

Initializing the coupled system requires that the sub-systems are initialized in a specific order. First, the chassis sub-system is initialized and when

finished, the values computed in the chassis for the connection variables is set to the wheels. The wheels are in turn initialized.

```
model_chassi.initialize(0.0, tf, True)

#Exchange data
for conn in connections:
    #Connection from model (chassi) 0 to model (wheel rf) 1
    if conn[0] == 0 and conn[2]==1:
        model_wheel_rf.set(conn[3],model_chassi.get(conn[1]))

    #Connection from model (chassi) 0 to model (wheel lf) 2
    if conn[0] == 0 and conn[2]==2:
        model_wheel_lf.set(conn[3],model_chassi.get(conn[1]))

    #Connection from model (chassi) 0 to model (wheel rb) 3
    if conn[0] == 0 and conn[2]==3:
        model_wheel_rb.set(conn[3],model_chassi.get(conn[1]))

     #Connection from model (chassi) 0 to model (wheel lb) 4
    if conn[0] == 0 and conn[2]==4:
        model_wheel_lb.set(conn[3],model_chassi.get(conn[1]))

model_wheel_rf.initialize(0.0, tf, True)
model_wheel_lf.initialize(0.0, tf, True)
model_wheel_rb.initialize(0.0, tf, True)
model_wheel_lb.initialize(0.0, tf, True)
```

The arguments to the initialization method is the start time of the simulation, the final time of the simulation and if the final simulation time is known.

The sub-systems together with the connections is next provided to the master class.

```
models = [model_chassi, model_wheel_rf, model_wheel_lf,
    model_wheel_rb, model_wheel_lb]
master_simulator = Master(models, connections)
```

The options in the master class is modified using the methods below.

```
master_simulator.set_step_size(step_size)
master_simulator.set_extrapolation_order(extrapolation_order)
master_simulator.use_output_iteration(output_iteration)
```

The coupled system was simulated both as a native co-simulation FMU and as an extended model exchange FMU. The `simulate` method was invoked as shown below.

```
res = master_simulator.simulate(final_time=5, initialize_slaves=
    False)
```

As the sub-systems already are initialized, the initialization flag to the `simulate` method is set to false.

In Figure 5.9 the result is shown for a simulation of both the native FMUs and the extended FMUs, with the solver CVode, using a global step-size of $H = 0.001$ and using the parallel approach. In Figure 5.10, the result is shown for the cases where iteration on the output variables is used. As the results show, the impact of using iteration on the output variables is negligible in this case.

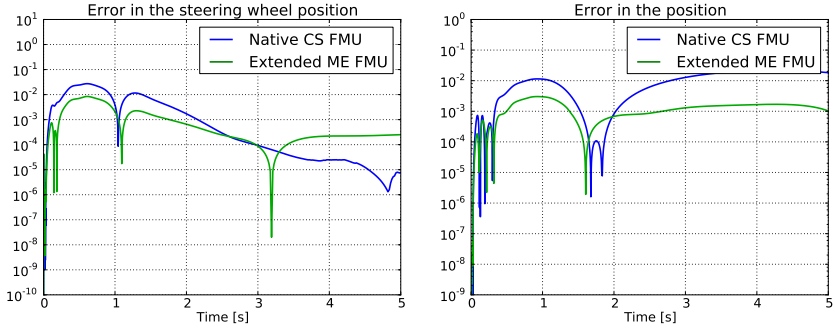

Figure 5.9: *Comparison between a simulation of the monolithic model of the race car, in Section 5.4, as an model exchange FMU using the solver CVode from* Assimulo *with a relative tolerance of* 10^{-8} *with simulations of the coupled system, both using native FMUs and extended FMUs. The global step-size was* $H = 0.001$*. The left figure shows the difference of the position of the race car while the right figure shows the difference of the steering wheel angle.*

In this example, it has been shown that co-simulating a race car using the developed sofware is possible.

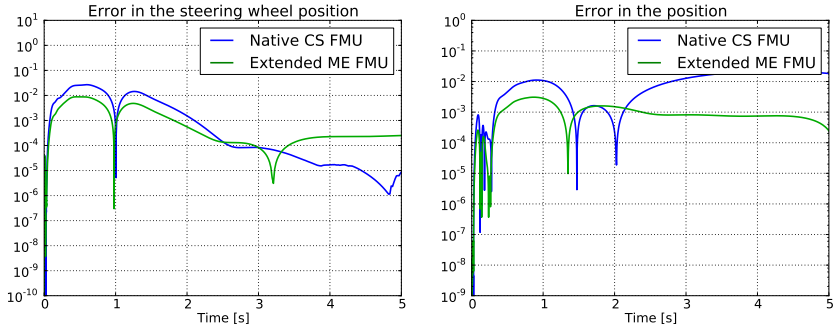

Figure 5.10: *Comparison between a simulation of the monolithic model of the race car, in Section 5.4, as an model exchange FMU using the solver CVode from* ASSIMULO *with a relative tolerance of* 10^{-8} *with simulations of the coupled system, both using native FMUs and extended FMUs. The global step-size was* $H = 0.001$ *and iteration on the output variables was used. The left figure shows the difference of the position of the race car while the right figure shows the difference of the steering wheel angle.*

Chapter 6

Discussion

6.1 Summary

Co-Simulation is an interesting and active research field where industry is a driving force, especially after the release of the Functional Mock-up Interface. 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 this thesis simulation of coupled systems in a co-simulation approach has been presented from the view of the Functional Mock-up Interface and block representation of the sub-systems. Different methods for co-simulation has been presented and tested in a developed framework. The stability problems that may occur when simulating coupled systems has been described and additionally how it can be handled. An approach for error estimation has been discussed and tested.

The thesis has presented an open framework for evaluating and testing co-simulation approaches. It has described the developed software, PyFMI and Assimulo together with the master algorithm. The framework developed has been successful in being able to evaluate co-simulation approaches on both academic test examples and on industrially relevant models.

The main contribution has been the developed software which will serve as a foundation for future research.

6.2 Future Work

Co-Simulation is an active research field with areas still unaddressed. There is however, a growing community targeting them, especially around the *Functional Mock-up Interface*. To name a few areas,

Discontinuous Systems

An area which is largely unexplored is co-simulation of discontinuous systems. Consider when there is a state event in one of the sub-systems, which leads to discontinuities in the solution. How should this information be propagated and handled by the master algorithm? What if the discontinuity triggers an event in another sub-system? In FMI 1.0 for instance, there is no way of knowing if an event has occurred or not which could lead to unsatisfactory behavior.

Sub-System Errors

Another area worth exploring is whether or not the knowledge of the subsystem errors, i.e. the error estimate of the local solvers, can be used to improve the master algorithm. Especially in the case when trying to adapt the global step-size to a meet a user defined tolerance. Currently the error estimation procedure discussed in Section 3.3.2 assumes that the subsystems are solved exactly which is not the case. Knowledge of the errors may be used to create an even more accurate estimation of the error in the master algorithm.

Impact of the solver

Choosing an appropriate solver to be used in export of FMUs is an area also worth further exploration. For example, using one of the acknowledged and popular multistep methods, for instance CVode, might not be the best choice depending on the use of the FMU. If using the FMU together with constant extrapolation, resulting in a piece-wise constant input trajectory, which in turn is similar to integrating over a discontinuity. This results in problem for the integrator where it has to reduce the step-size significantly in order to possibly be able to pass the discontinuity. If on the other hand a one-step method was used in the FMU, the problem is no longer present and a possible performance increase is achieved.

Investigating the options provided to the underlying solver is also of interest as the options may impact the performance. An example is the size of the initial step.

Acknowledgements

First and foremost I want to extend a special thank you to my supervisors, Claus Führer and Johan Åkesson, for their continued support and belief in my success. Without their guidance, this thesis would not have been possible.

To everyone at Modelon[1] and especially to the people in the Simulation & Optimization R&D division, thank you for creating a productive environment and for the friendly after-works. To Johan Andreasson for providing test models. I also want to thank the colleagues at the Centre for Mathematical Sciences, and especially the colleagues at Numerical Analysis, for the many fruitful discussions and for providing an open and pleasant environment.

Moreover, I would like to thank the executives at Modelon and the Faculty of Engineering at Lund University for making a joint position between Lund University and Modelon possible. I also want to thank LCCC[2], Lund Center for Control of Complex Engineering Systems, for the interdisciplinary environment and for providing a forum for discussions across subjects.

To my family and friends, a big thank you for believing in me and finally a thank you to my love Helena Sjöblom for your love and support.

[1] I gratefully acknowledge the financial support from Modelon.
[2] I gratefully acknowledge the financial support from LCCC.

Bibliography

[1] Functional mock-up interface for co-simulation. Interface specification, MODELISAR, October 2010.

[2] Functional mock-up interface for model exchange. Interface specification, MODELISAR, January 2010.

[3] Modelon AB. Fmi library. `http://www.jmodelica.org/FMILibrary`, 2013. [Online; accessed 30-July-2013].

[4] Andreas Abel, Torsten Blochwitz, Alexander Eichberger, Peter Hamann, and Udo Rein. Functional mock-up interface in mechatronic gearshift simulation for commercial vehicles. In *9th International Modelica Conference. Munich*, 2012.

[5] SIMPACK AG. Simpack - multi-body simulation software. http://www.simpack.com/.

[6] C. Andersson, J. Åkesson, C. Führer, and M. Gäfvert. Import and export of functional mock-up units in jmodelica.org. In *In 8th International Modelica Conference 2011*. Modelica Association, 2011.

[7] C. Andersson, J. Andreasson, C. Führer, and J. Åkesson. A workbench for multibody systems ode and dae solvers. In *The Second Joint International Conference on Multibody System Dynamics*, 2012.

[8] J. F. Andrus. Numerical solution of systems of ordinary differential equations separated into subsystems. *SIAM Journal on Numerical Analysis*, 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*. Martin Luther University Halle-Wittenberg, Institute of Mathematics, 2007.

[10] Martin Arnold. Stability of sequential modular time integration methods for coupled multibody system models. *Journal of Computational and Nonlinear Dynamics*, 5(3):031003, 2010.

[11] Martin Arnold and Michael Günther. Preconditioned dynamic iteration for coupled differential-algebraic systems. *BIT Numerical Mathematics*, 41(1):1–25, 2001.

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

[13] Peter N. Brown, G. D. Byrne, and A. C. Hindmarsh. Vode: a variable-coefficient ode solver. *SIAM J. Sci. Stat. Comput.*, 10(5):1038–1051, September 1989.

[14] G. D. Byrne and A. C. Hindmarsh. A polyalgorithm for the numerical solution of ordinary differential equations. *ACM Trans. Math. Softw.*, 1(1):71–96, March 1975.

[15] Carlos A. Felippa, K.C. Park, and Charbel Farhat. Partitioned analysis of coupled mechanical systems. *Computer Methods in Applied Mechanics and Engineering*, 190(24–25):3247 – 3270, 2001.

[16] Emil Fredriksson. Discontinuities handled with events in assimulo, a practical approach. Master's thesis, Lund University, June 2013.

[17] Dag Fritzson, Jonas Ståhl, and Iakov Nakhimovski. Transmission line co-simulation of rolling bearing applications. In *Proceedings of the 48th Conference of Simulation and Modeling*, 2007.

[18] C. Führer and B. J. Leimkuhler. Numerical solution of differential-algebraic equations for constrained mechanical motion. *Numerische Mathematik*, 59:55–69, 1991.

[19] C. W. Gear and D. R. Wells. Multirate linear multistep methods. *BIT Numerical Mathematics*, 24:484–502, 1984. 10.1007/BF01934907.

[20] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations: Nonstiff problems*. Springer series in computational mathematics. Springer-Verlag, 1991.

[21] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations: Stiff and differential-algebraic problems*. Springer series in computational mathematics. Springer-Verlag, 1993.

[22] Alan C. Hindmarsh. Serial fortran solvers for ode initial value problems. `https://computation.llnl.gov/casc/odepack/odepack_home.html`, 2013. [Online; accessed 30-July-2013].

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

[24] Alan C. Hindmarsh and Radu Serban. User Documentation for CVODE v2.7.0. Technical report, Center for Applied Scientific Computing Lawrence Livermore National Laboratory, 2012.

[25] Gerhard Hippmann, Martin Arnold, and Marcus Schittenhelm. Efficient simulation of bush and roller chain drives. In *Proceedings of Multibody Dynamics, ECCOMAS Thematic Conference*, 2005.

[26] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[27] ITI. Simulationx - multi-domain system simulation and modeling. http://www.itisim.com/.

[28] K. R. Jackson and R. Sacks-Davis. An alternative implementation of variable step-size multistep formulas for stiff odes. *ACM Trans. Math. Softw.*, 6(3):295–318, September 1980.

[29] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[30] R. Kübler and W. Schiehlen. Modular simulation in multibody system dynamics. *Multibody System Dynamics*, 4:107–127, 2000. 10.1023/A:1009810318420.

[31] R Kübler and W Schiehlen. Two methods of simulator coupling. *Mathematical and Computer Modelling of Dynamical Systems*, 6(2):93–113, 2000.

[32] R. I. Leine, D. H. Van Campen, and C. H. Glocker. Nonlinear Dynamics and Modeling of Various Wooden Toys with Impact and Friction. *Journal of Vibration and Control*, 9(1-2):25, 2003.

[33] Teo Nilsson. A simulation environment for coupled systems of discontinuous ode:s. Master's thesis, Lund University, June 2013.

[34] Travis E. Oliphant. *Guide to NumPy*. Provo, UT, March 2006.

[35] Automotive Engineering Online. Sae 2013 world congress. `http://www.sae.org/mags/aei/12036?PC=130520NWTS`, 2013. [Online; accessed 30-July-2013].

[36] L.R. Petzold. Description of dassl: a differential/algebraic system solver. Technical report, September 1982.

[37] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749, November 2010.

[38] Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2007.

[39] Tom Schierz. *Modulare Zeitintegration gekoppelter Differentialgleichungssysteme in der technischen Simulation*. PhD thesis, Naturwissenschaftlichen Fakultät II Chemie, Physik und Mathematik, Martin Luther Universität Halle Wittenberg, Germany, Mai 2013.

[40] Tom Schierz and Martin Arnold. Stabilized overlapping modular time integration of coupled differential-algebraic equations. *Appl. Numer. Math.*, 62(10):1491–1502, October 2012.

[41] Tom Schierz, Martin Arnold, and Christoph Clauss. Co-simulation with communication step size control in an fmi compatible master algorithm. In *In 9th International Modelica Conference 2012*. Modelica Association, 2012.

[42] Dassault Systèmes. Dymola - multi-engineering modeling and simulation. http://www.dymola.com/.

[43] S. Voigtmann. *General Linear Methods for Integrated Circuit Design*. Logos Verlag Berlin, 2006.