The 2nd Joint International Conference on Multibody System Dynamics
May 29–June 1, 2012, Stuttgart, Germany

A Workbench for Multibody Systems ODE and DAE Solvers

Christian Andersson[*#†], **Johan Andreasson**[†], **Claus Führer**[*], **Johan Åkesson**[#†]

[*] Department of Numerical Analysis
Lund University
Box 118, 221 00 Lund, Sweden
[chria, claus]@maths.lth.se

[#] Department of Automatic Control
Lund University
Box 118, 221 00 Lund, Sweden
johan.akesson@control.lth.se

[†]Modelon AB
Scheelevägen 17, 223 63 Lund, Sweden
johan.andreasson@modelon.com

ABSTRACT

During the last three decades, a vast variety of methods to numerically solve ordinary differential equations (ODEs) and differential algebraic equations (DAEs) has been developed and investigated. Few of them met industrial standards and even less are available within industrial multibody simulation software. Multibody Systems (MBS) offer a challenging class [5] of applications for these methods, since the resulting system equations are in the unconstrained case ODEs which are often stiff or highly oscillatory. In the constrained case the equations are DAEs of index-3 or less. Friction and impact in the MBS model introduce discontinuities into these equations while coupling to discrete controllers and hardware-in-the-loop components couple these equations to additional time discrete descriptions. Many of the developed numerical methods have promising qualities for these types of problems, but rarely got the chance to be tested on large scale problems. One reason is the closed software concept of most of the leading multibody system simulation tools or interface concepts with a high threshold to overcome. Thus, these ideas never left the academic environment with their perhaps complex but dimensionally low scale test problems. In this paper we will present a workbench, ASSIMULO, which allows easy and direct incorporation of new methods for solving ODEs or DAEs written in FORTRAN, *C*, Python or even MATLAB and which indirectly interfaces to multibody programs such as Dymola and Simpack, via a standardized interface, the *functional mock-up interface*. The paper is concluded with industrial relevant examples evaluated using industrial and academic solvers.

1 INTRODUCTION

In an attempt to provide a unified interface for model interaction between simulation tools and modeling environments, the MODELISAR project defined the *Functional Mock-up Interface (FMI)*. The intention is that a tool generates a model following the FMI specification which then in turn can be exchanged via the standardized interface. A model following the specification is called a *Functional Mock-up Unit (FMU)*. Our intention is to describe a workflow where it is possible to create a multibody system using a commercially available tool, such as Dymola, and by exporting the system as an FMU, enables it to be interfaced into the simulation package ASSIMULO and made available for the solvers connected.

ASSIMULO is a simulation package for solving ordinary differential equations containing various different solvers, both state-of-the art and more experimental. ASSIMULO is written in the programming language Python which is a powerful dynamic programming language with a clear and readable syntax. The choice of Python is highly influenced by the ability and the ease of connecting different software written in different programming languages, such as *C* or FORTRAN, and the ability of using Python as *glue*. Another aspect is due to the clear and readable syntax of Python a user can easily create their own scripts as the threshold is relatively low for learning the language compared with the low-level languages *C* and FORTRAN, especially if the user has a background in MATLAB. Python is also a good choice for prototyping as there are specialized packages for scientific computing and for visualization, much like MATLAB.

Another key part of the workflow is the Python package PyFMI which enables a FMU to be accessed from Python and thus also be able to be interfaced into ASSIMULO.

This paper is outlined as follows. In Section 2, an overview of the Functional Mock-up Interface (FMI) is given together with a description of its capabilities and a package, PyFMI, for interaction with models using the high-level language Python. Next, the simulation package ASSIMULO is described together with its available solvers and problem formulations. In Section 4, an example is given demonstrating the concept of using the interface to gain access to models written in commercial multibody software. Finally, in Section 5, conclusions and limitations are discussed as well as future work.

2 FUNCTIONAL MOCK-UP INTERFACE

The Functional Mock-up Interface (FMI) [3] defines a standard for model exchange with a small set of functions for model interaction while at the same time it provides the necessary means for simulation of complex hybrid dynamic models. FMI is a result of the ITEA2 project MODELISAR where the idea is that a modeling and / or simulation environment generates a dynamic library or source files, written in *C*, which then can be exchanged between different tools and connected to other models. Additionally, FMI specifies an XML format for describing the variables and parameters contained in the model. The idea is that these two, self-contained files are compressed into a Functional Mock-up Unit (FMU) which can be exchanged between different environments, see Figure 1 (Simpack plan to support export of Functional Mock-up Units in 2012[1]).

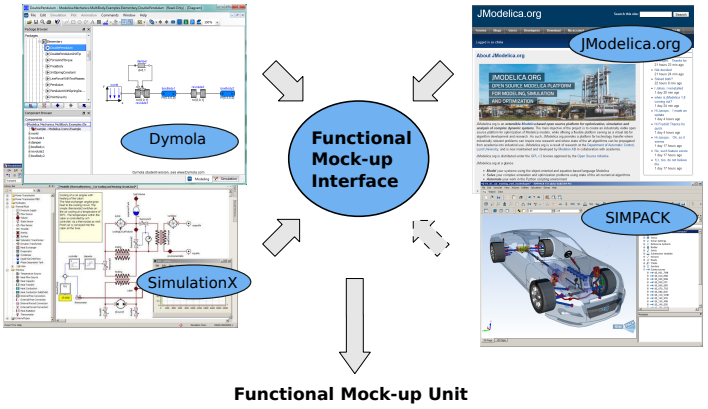

Figure 1. Model exchange using the Functional Mock-up Interface (FMI).

The standard describes models as ordinary differential equations with time, state and step events. The tool that export the FMU is responsible for providing the *right-hand-side* together with the methods for monitoring the different events. How the events are intended to be handled depends on the type of event. Time events are simply handled by asking the provided method for the next time event and if it is less than the user asked final time, integrate up to that time instant and ask for the next time event. State events are a bit more difficult as they are dependent on a set of functions where a sign change in any of the functions indicates that a state event has occurred. This sets a constraint on the integrator that it has to continuously monitor these sets of functions during the integration so that it can detect sign changes and if so, halt the integration. Step events also sets a constraint on the integrator as they should be monitored after each successful internal step taken by the integrator.

FMI 1.0 for model exchange was released in January 2010 and has received a significant amount of attention among vendors. There are currently 34 tools that support or plan to support the FMI. Example of tools includes the commercial products Dymola [1] and Simpack [2] as well as the open-source platform JModelica.org [11].

[1] http://www.modelisar.com/tools.html

2.1 PyFMI

PyFMI [2] is a Python package for interacting with FMUs using Python. PyFMI takes care of unzipping the FMU, connecting the binary for interaction from Python and loads the model information. The package provides both a connection to the low-level native FMI functions and convenient high-level functions for accessing model parameters and setting attributes. A model can be imported into Python using just a few lines:

```
from pyfmi import FMUModel

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

PyFMI also provides natively a connection to ASSIMULO and thus enables a FMU to be simulated using the different solvers available in ASSIMULO. Additionally, a graphical user interface for visualization of simulation results is available. An overview of the interactions are shown in Figure 2.

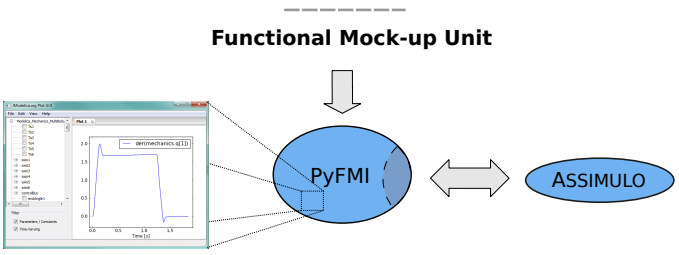

Figure 2. Interaction between a Functional Mock-up Unit (FMU), PyFMI and ASSIMULO.

3 ASSIMULO

ASSIMULO [4] is a workbench for solving ordinary differential equations formulated as first or second order explicit ordinary differential equations, (1) and implicit ordinary differential equations (differential–algebraic equations, DAEs), (2).

$$\dot{y} = f(t, y), \quad y(t_0) = y_0 \tag{1}$$

$$F(t, y, \dot{y}) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0. \tag{2}$$

ASSIMULO is written in the high-level programming language Python and combines a variety of different solvers written in FORTRAN, *C* and even Python via a common high-level interface. ASSIMULO consists of mainly two parts, problem definitions and solvers. 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. The idea is to keep information related to a problem separate from the solver. For instance, the tolerances, which are important quantities to control the solver, are attributes of the solver class rather and are kept separate from the problem description.

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

$$\dot{y} = f(t, y, sw), \quad y(t_0) = y_0 \tag{3a}$$

$$0 = g(t, y, sw) \tag{3b}$$

[2] http://www.pyfmi.org

where f is the *right-hand-side* of an explicit ordinary differential equation 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.

In Section 3.1, the available problem formulations are presented and in Section 3.2 follows a brief description of the solvers available and those planned to be included.

3.1 Problem formulations

ASSIMULO provides at its present stage four problem classes:

- Explicit hybrid ODEs

$$\dot{y} = f(t, y, sw), \quad y(t_0) = y_0, \quad sw(t_0) = sw_0 \tag{4}$$

- Implicit hybrid ODEs (also called DAEs)

$$F(t, y, \dot{y}, sw) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0, \quad sw(t_0) = sw_0 \tag{5}$$

- Mechanical systems in second order explicit ODE form

$$\ddot{p} = M(p)^{-1} f(t, p, \dot{p}) \tag{6}$$

- Mechanical systems in (overdetermined) implicit ODE form

$$\dot{p} = v \tag{7}$$
$$M(p)\dot{v} = f(t, p, v) - G^T(p)\lambda \tag{8}$$
$$0 = g_{\text{constr}}(p) \tag{9}$$
$$0 = G(p)v \tag{10}$$

For mechanical systems we denoted position and velocity states by p and v respectively. Lagrange multipliers are denoted by λ, constraint matrix by G and the mass matrix by M. The applied forces are described by f and the constraints by g_{constr}. Note, $\frac{\mathrm{d}}{\mathrm{d}p} g_{\text{constr}}(p) =: G$.

3.2 Solvers

The solvers interfaced to ASSIMULO consist of a variety of different codes written in both FORTRAN and *C*. The codes include explicit and implicit Runge–Kutta methods, Rosenbrock-Wanner methods, multistep methods of Adams and BDF type, Newmark and HHT-α methods. It is planned to include even extrapolation methods for DAEs. Some methods come with a continuous solution representation which enables root-finding for event detection to correctly handle hybrid systems.

The solver and problem classes are related to each other corresponding to Figure 3. The most universal solver class handles implicit ODEs. Explict ODEs are thus transformed to their implicit counterpart (residual formulation), when solved with an implicit solver. Correspondingly overdetermined mechanical DAEs are in this case transformed to fully determined DAEs by using a GGL-stabilization technique [7]. Also, mechanical problems given as second order ODEs are automatically transcribed to implicit ODEs, when exposed to a solver of that class.

The advantage of specially designed problem classes is to allow special purpose solvers. Thus, ASSIMULO allows to solve most of the problems with specialized methods as well as with general purpose solvers. This

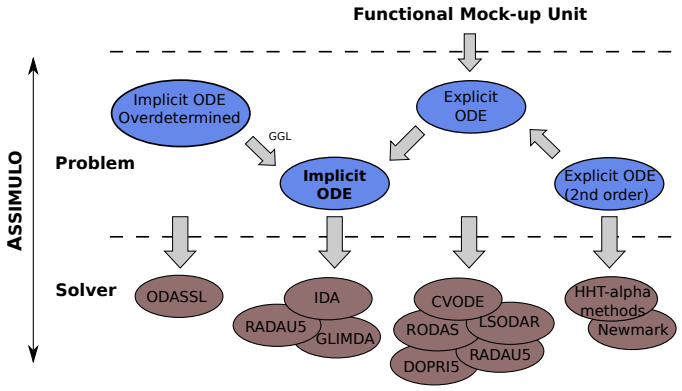

Figure 3. Connection between the different problem formulations and the different solvers available in ASSIMULO. The connection of the Functional Mock-up Interface to ASSIMULO is also shown.

makes the tuning of code control parameters as well as performance and accuracy comparisons easy and documentable.

In the present state, ASSIMULO provides interfaces to production quality solvers like CVODE and IDA from the SUNDIALS [10] 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 ordinary differential equations, $\dot{y} = f(t, y)$. 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 implementation. IDA, on the other hand solves the more general problem described as implicit ODEs (differential algebraic equations). 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 the ASSIMULO project is to give the simulation and modeling engineer access to the wide spread flora of research codes. A typical representative for this class of codes is Glimda [12] which is now accessible by ASSIMULO. Glimda is an implementation of general linear multistep methods to solve lower index DAEs. These methods can be viewed as a blend of the collocation approach of implict 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 concepts 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 [9] 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 included in ASSIMULO. The solvers DOPRI5 [8] and RODAS [9] 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 guarantee a homogeneous handling. But ASSIMULO also provides to contribute with experimental code directly written in Python. A constant step size Runge–Kutta method and an explicit Euler method, both implemented in Python, are included in ASSIMULO. ASSIMULO also aims to expose even historically interesting codes together with modern industrial codes and more unknown

research codes. Among the classical codes we name the solver LSODAR from ODEPACK[3] which is a multistep method that depending on the stiffness of the problem switches between a Adams method and a BDF method. Also ODASSL [6] 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. Other classical MBS simulation codes are planned to be included.

4 EXAMPLES AND RESULTS

For racing applications, finding the maximal performance of the car is crucial. One method to quickly estimate the impact on performance of a change to the vehicle setup is to solve for the steady state limits under different driving conditions. Identifying a set of critical points along a race track and calculating the maximum achievable speed for each point can give a good indication on how the change will affect the lap time. 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, Figure 4, and exported to an 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.

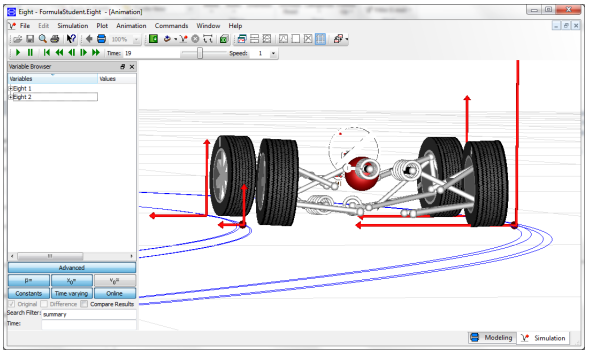

Figure 4. Dymola model of a student formula race car with 47 states.

The FMU is then imported into Python, by means of the PyFMI package, 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 an 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 a specialized package. Below, a Python script for a simulation of the race car is shown.

```
from pyfmi import FMUModel
import pylab as P #Used for plotting

model = FMUModel("FormulaStudent_Eight.fmu", enable_logging=True)
model.set("steeringInEight.left_turn", -1.0) #Change the initial position
                                             #of the steering wheel

#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)

#Get the results
t = res["time"] #Time
```

[3]https://computation.llnl.gov/casc/odepack/odepack_home.html

```
steering_wheel = res["chassis.summary_p_sw"]
p_x = res["chassis.summary_r_0[1]"]
p_y = res["chassis.summary_r_0[2]"]

#Plot the results
P.figure(1)
P.plot(t, steering_wheel)
P.title("Steering Wheel")
P.xlabel("Time [s]");P.ylabel("Angle [rad]")

P.figure(2)
P.plot(p_x, p_y)
P.title("Track")
P.xlabel("Position in X [m]");P.ylabel("Position in Y [m]")

P.show()
```

The model is simulated using the solver CVODE in ASSIMULO for 30 seconds. In Figure 5, the resulting angle of the steering wheel is shown together with the position of the race car. In Figure 6, the solver statistics is shown.

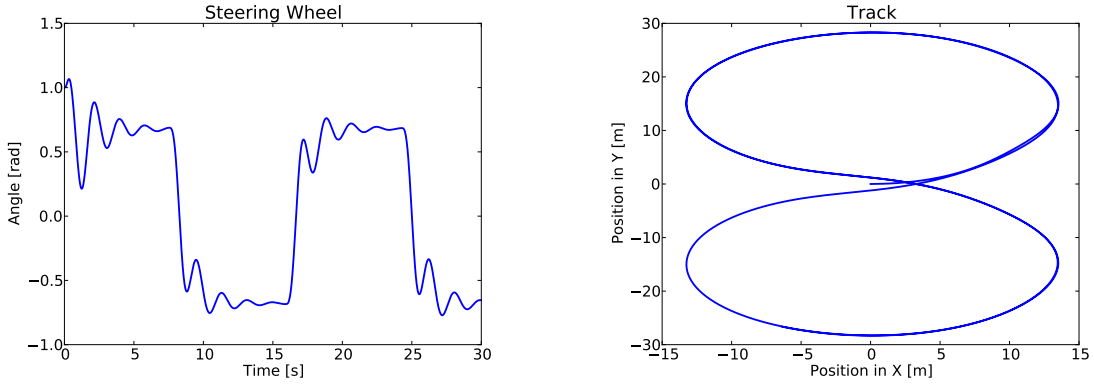

Figure 5. Results from a simulation of the race car driving on an eight shaped course. The left figure shows the angle of the steering wheel while the right figure shows the position of the race car. The model was generated as an FMU from Dymola and simulated in ASSIMULO using the solver CVODE.

```
Final Run Statistics: FormulaStudent_Eight

 Number of Steps                           : 1454
 Number of Function Evaluations            : 2070
 Number of Jacobian Evaluations            : 39
 Number of F-Eval During Jac-Eval          : 1833
 Number of Root Evaluations                : 1710
 Number of Error Test Failures             : 39
 Number of Newton Iterations               : 2022
 Number of Newton Convergence Failures     : 0

Solver options:

 Solver                  : CVode
 Linear Multistep Method : BDF
 Nonlinear Solver        : Newton
 Maxord                  : 5
 Tolerances (absolute)   : [  1.0e-06   ...   1.0e-06]
 Tolerances (relative)   : 0.0001

Simulation interval    : 0.0 - 30.0 seconds.
Elapsed simulation time: 54.3402747302 seconds.
```

Figure 6. Solver statistics reported by ASSIMULO from a simulation of the race car using the solver CVODE.

The result can easily be compared against simulations of the original model in Dymola and with different

integrators connected in ASSIMULO. In Figure 7, a comparison is made of the angle of the steering wheel using two different integrators in ASSIMULO against a simulation with the integrator DASSL on the original model in Dymola.

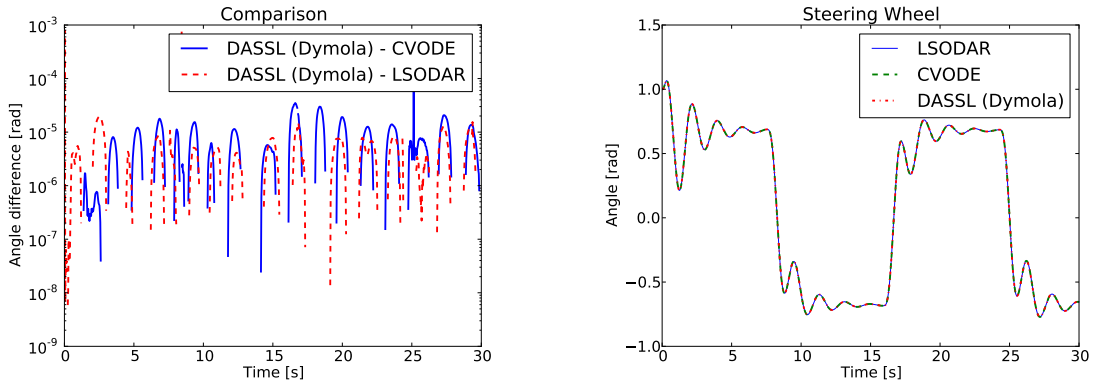

Figure 7. Comparison between a simulation of the original model in Dymola using DASSL with simulations of the FMU using the integrators LSODAR and CVODE from ASSIMULO. The left figure shows the difference of the steering wheel angle between the different integrators while the right figure shows the actual angle calculated using the different integrators.

5 CONCLUSIONS AND FUTURE WORK

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

ASSIMULO currently interfaces to a number of different solvers suitable for different problems and different problem formulations. The intention is to continue to include solvers and a growing number of problem formulations, such as delay differential equations, to provide a solid foundations for developers to evaluate their solver on industrial relevant examples and for users to test their problems on a variety of different solvers. We believe that providing a open-source platform where industrial and academic solvers can be tested on industrial relevant examples will benefit both academia and the industry.

A current limitation is that the Functional Mock-up Interface only specifies models described as explicit ordinary differential equations. With this article we want to stimulate to open the standard for a wider range of problem formulations, such as overdetermined problems commonly in MBS, which would enable evaluations of a greater selection of solvers.

ASSIMULO has been proven to be a powerful code and concept. We hope that in the near future an increasingly variety of original codes will be included, thanks of software like Cython and F2PY.

REFERENCES

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

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

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

[4] Christian Andersson, Claus Führer, Johan Åkesson, and Magnus Gäfvert. Assimulo. http://www.assimulo.org.

[5] M. Arnold, B. Burgermeister, C. Führer, G. Hippmann, and G. Rill. Numerical methods in vehicle system dynamics: state of the art and current developments. *Vehicle System Dynamics*, 49(7):1159–1207, 2011.

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

[7] C.W. Gear, B. Leimkuhler, and G.K. Gupta. Automatic integration of euler-lagrange equations with constraints. *Journal of Computational and Applied Mathematics*, 12-13(0):77 – 90, 1985.

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

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

[10] 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.

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

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