JModelica.org User Guide

Version 1.2.0

JModelica.org User Guide: Version 1.2.0

Publication date 2010-04-22

Acknowledgements

This document is produced with DocBook 5 using XMLMind XML Editor for authoring, Norman Walsh's XSL stylesheets and a GNOME xsltproc + Apache fop toolchain. Math contents is converted from LaTeX using the TeX/LaTeX to MathML Online Translator by the Ontario Research Centre for Computer Algebra and processed by JEuclid.

Chapter 1. Introduction

1. About JModelica.org

JModelica.org is an extensible Modelica-based open source platform for optimization, simulation and analysis of complex dynamic systems. The main objective of the project is to create an industrially viable open source platform for optimization of Modelica models, while offering a flexible platform serving as a virtual lab for algorithm development and research. As such, JModelica.org is intended to provide a platform for technology transfer where industrially relevant problems can inspire new research and where state of the art algorithms can be propagated form academia into industrial use. JModelica.org is a result of research at the Department of Automatic Control, Lund University, [Jak2007]and is now maintained and developed by Modelon AB in collaboration with academia.

2. Mission Statement

To offer a community-based, free, open source, accessible, user and application oriented Modelica environment for optimization and simulation of complex dynamic systems, built on well-recognized technology and supporting major platforms.

3. Technology

JModelica.org relies on the established modeling language Modelica. Modelica targets modeling of complex heterogeneous physical systems, and is becoming a de facto standard for dynamic model development and exchange. There are numerous model libraries for Modelica, both free and commercial, including the freely available Modelica Standard Library (MSL).

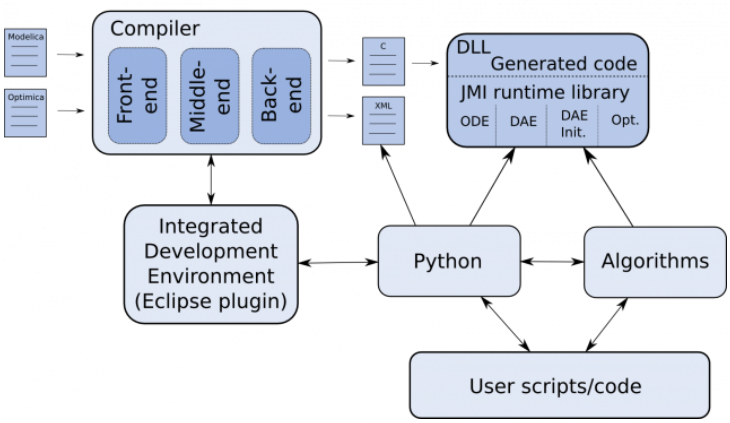
A unique feature of JModelica.org is the support for the innovative extension Optimica. Optimica enables you to conveniently formulate optimization problems based on Modelica models using simple but powerful constructs for encoding of optimization interval, cost function and constraints. Optimica also features annotations for choosing and tailoring the underlying numerical optimization algorithm a particular optimization problem.

The JModelica.org compilers are developed in the compiler construction framework JastAdd. JastAdd is based on established concepts, including object orientation, aspect orientation and reference attributed grammars. Compilers developed in JastAdd are specified in terms of declarative attributes and equations which together forms an executable specification of the language semantics. In addition, JastAdd targets extensible compiler development which makes it easy to experiment with language extensions.

For user interaction JModelica.org relies on the Python language. Python offers an interactive environment suitable for scripting, development of custom applications and prototype algorithm integration. The Python packages Numpy and Scipy provide support for numerical computation, including matrix and vector operations, basic linear algebra and plotting. The JModelica.org compilers as well as the model executables/dlls integrate seemlessly with Python and Numpy.

4. Architecture

Figure 1.1. JModelica platform architecture.



The JModelica.org platform consists of a number of different parts:

- The compiler front-ends (one for Modelica and one for Modelica/Optimica) transforms Modelica and Optimica code into a flat model representation. The compilers also check the correctness of model descriptions and reports errors.
- The compiler back-ends generates C code and XML code for Modelica and Optimica. The C code contains the model equations, cost functions and constraints whereas the XML code contains model meta data such as variable names and parameter values.
- The JModelica.org runtime library is written in C and contains supporting functions needed to compile the generated model C code. Also, the runtime library contains an integration with CppAD, a tool for computation of high accuracy derivatives by means of automatic differentiation.
- Currently, JModelica.org features one particular algorithm for solving dynamic optimization problems. The algorithm is based on collocation on finite elements and relies on the solver IPOPT for obtaining a solution of the resulting NLP.
- JModelica.org uses Python for scripting and prototyping. For this purpose, a Python package is under development with the objective of offering functions for driving the compilers and for accessing the (compiled) functions in the runtime library/generated C code.

5. Extensibility

The JModelica.org platform is extensible in a number of different ways:

- JModelica.org features a C interface for efficient evaluation of model equations, the cost function and the constraints: the JModelica Model Interface (JMI). JMI also contains functions for evaluation of derivatives and sparsity and is intended to offer a convenient interface for integration of numerical algorithms.
- In addition to the the C interface, model meta data can be exported in XML. In the future this feature is intended to be extended to include full model export in XML, which in turn enables use of XML techniques such as XPATH and XSLT.
- JastAdd produces compilers encoded in pure Java. As a result, the JModelica.org compilers are easily embedded in other applications aspiring to support Modelica and Optimica. In particular, a Java API for accessing the flat model representation and an extensible template-based code generation framework is offered.
- The JModelica.org compilers are developed using the compiler construction framework JastAdd. JastAdd features extensible compiler construction, both at the language level and at the implementation level. This feature

is explored in JModelica.org where the Optimica compiler is implemented as a fully modular extension of the core Modelica compiler. The JModelica.org platform is a suitable choice for experimental language design and research.

An overview of the JModelica.org platform is given [Jak2010]

Chapter 2. Installation

1. Supported platforms

JModelica.org can be installed on Linux, Mac OS X, and Windows (XP, Vista, 7) with 32-bit or 64-bit architectures. Most development work is carried out on 32-bit Mac OS X , Linux and Windows XP, so these platforms tend to be best tested.

2. Binary distribution

Pre-built binary distributions for Windows are available in the Download section of www.jmodelica.org.

2.1. Linux

Currently, no pre-built binary distributions are provided for Linux.

2.2. Mac OS X

Currently, no pre-built binary distributions are provided for Mac OS X.

2.3. Windows

The JModelica.org Windows installer contains a binary distribution of JModelica.org built using the JModelica.org-SDK, bundled with required third-party software components. The JModelica.org Windows installer sets up a pre-configured complete environment with convenient start menu shortcuts.

2.3.1. Prerequisites

Make sure to install the required software components listed in this section before installing JModelica.org.

2.3.1.1. Java

It is required to have a Java Runtime Environment (JRE) version 6 installed on your computer.

To install JRE

1. Get a JRE installer suitable for your platform here.
2. Run the installer.

2.3.1.2. Python

Python 2.6 with the following additional packages are required:

NumPy	The fundamental package needed for scientific computing with Python.
SciPy	A library of algorithms and mathematical tools for Python.
matplotlib	A plotting library for Python, with a MATLAB like interface.
PyReadline	A readline for Windows required by IPython.
IPython	An interactive shell for Python with additional shell syntax, code highlighting, tab completion, string completion, and rich history.

There are two options to install the necessary Python components, as described below.

Separate installation of Python prerequisites for Windows binary installer

1. Download and install Python 2.6.
2. Download and install the additional packages (be sure to select a version for Python 2.6 when applicable):
    1. NumPy version 1.2.0 or higher.
    2. SciPy
    3. matplotlib
    4. PyReadline version 1.5
    5. IPython version 0.10 or higher

Python(x,y) installation of Python prerequisites for Windows binary installer

Python(x,y) is a scientific-oriented Python distribution that includes all necessary dependencies (and a lot more!).

1. Download a Python(x,y) installer for Python 2.6.
2. Run the installer and select a full installation (or select the required packages manually for a smaller footprint).

2.3.2. Windows installer

To install JModelica.org from Windows installer

Make sure that all prerequisite components are installed before continuing.

1. Download a JModelica.org Windows binary installer.
2. Run the installer and follow the wizard.
    - Choose to install the additional Python packages when prompted, unless the are already installed.

Chapter 3. Getting started

The examples in JModelica.org uses the Python scientific library SciPy, which is dependent on NumPy and the mathematical plotting library matplotlib for plotting. These packages will thus also be used in the example below. Most Python functions have built in documentation, which can be accessed from the Python shell by invoking the help function, for example 'help(numpy.size)'. Use this feature frequently to learn more about the packages used in this tutorial, including the jmodelica package. It is possible do the tutorials without any Python knowledge (although it helps to know the basics). In most cases it is convenient to store the Python commands in a script file and run the script from the file by invoking the Python command run:

```
>>> run my_script.py
```

or

```
>>> run -i my_script.py
```

where the latter will render all variables to be accessible in the Python interpreter after termination of the script.

1. Compilation of models

This tutorial covers how to compile Modelica and Optimica models into C and XML and how to load the resulting DLLs into Python.

1.1. Compilation

There are two compilers available, `ModelicaCompiler` and `OptimicaCompiler`. Any model containing Optimica code has to be compiled with the `OptimicaCompiler`. This is the case with the CSTR model which will be used in the following examples.

1.1.1. Simple compilation example

Compiling a model file can be done with just a few lines of code.

1.1.1.1. Instantiating the compiler

First the `OptimicaCompiler` must be imported and instantiated. The compiler instance can then be used multiple times on different model files.

```
# Import the compiler
from jmodelica.compiler import OptimicaCompiler

# Get an instance of OptimicaCompiler
oc = OptimicaCompiler()
```

1.1.1.2. Options

Compiler options are read from an XML file 'options.xml' which can be found in the JModelica.org installation folder. The options are loaded from the file as the compiler is instantiated. Options for a compiler instance can be modified and new options can be added interactively. There are four type categories: string, real, integer and boolean. The following example demonstrates how to get and set a string option.

```
# Get the string option default_msl_version
oc.get_string_option('default_msl_version')
>> '3.0.1'

# Set the default_msl_version to 3.0.1
oc.set_string_option('default_msl_version','3.0.1')
```

1.1.1.3. Compiling

A model can now be compiled with the compiler instance using the method `compile_model` which takes a model name and a model file name as arguments.

```
# Compile the model
oc.compile_model('CSTR.CSTR_Opt', 'CSTR.mo')
```

On a successful compilation a C file, some XML files and a shared object file (DLL) will have been generated. If the compilation has failed an exception will be raised.

1.1.2. Targets

The `compile_model` method takes an optional argument target which is '`model`' by default. There are two other options for this argument, 'algorithms' and 'ipopt'. It is necessary to compile with target '`ipopt`' to use the Ipopt algorithm interface for optimization.

```
# Compile the model with support for Ipopt
oc.compile_model('CSTR.CSTR_Opt', 'CSTR.mo', target='ipopt')
```

1.1.3. Compilation in more detail

Compiling with `compile_model` actually bundles a few steps required for the compilation which can be run one by one. These steps will be described briefly here, for more information on these steps, see the Architecture section in the Introduction.

1.1.3.1. Flattening

In the first step, the model is transformed into a flat representation which can be used to generate C and XML code. Before this can be done the model must be parsed and instantiated. If there are errors in the model, for example syntax or type errors, Python exceptions will be thrown during these steps.

```
# Parse the model and get a reference to the source root
source_root = oc.parse_model('CSTR.mo')

# Generate an instance tree representation and get a reference to the model instance
model_instance = oc.instantiate_model(source_root, 'CSTR.CSTR_Opt')

# Perform flattening and get a flat representation
flat_rep = oc.flatten_model(model_instance)
```

1.1.3.2. Code generation

The next step is the code generation which produces C code containing the model equations and a couple of XML files containing model meta data such as variable names and types.

```
# Generate code
oc.generate_code(flat_rep)
```

There several files are generated in this step.

1.1.3.3. Generate Shared Object file (DLL)

Finally, the DLL file is built where the C code is linked with the JModelica.org Model Interface (JMI) runtime library. The target argument must be set here if something other than the default 'model' is wanted.

```
# Compile DLL
oc.compile_dll('CSTR_CSTR_Opt', target='ipopt')
```

1.2. Loading the Shared Object file (DLL)

Once compilation has completed successfully a DLL file along with a few other files will have been created on the file system. The DLL file can then be loaded in Python using the class Model from which the JMI Model interface can be reached.

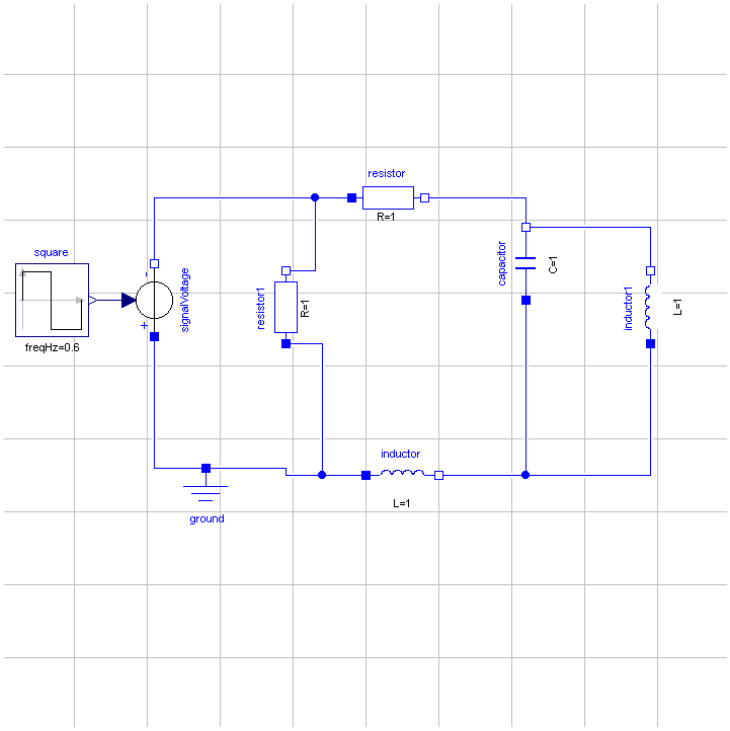
```
# Import Model
from jmodelica.jmi import Model
# Load dll file and create Model object
model = Model('CSTR_CSTR_Opt')
```

The model object can now be used to manipulate parameters and for optimization and simulation.

2. Simulation of models

This example focus on how to use the high-level simulation functionality on a model of an electric circuit. The model is depicted in Figure (RLC.png) and consists of resistances, inductors and a capacitor. The circuit is connected to a voltage source which generates a square-wave with an amplitude of 1.0 and a frequency of 0.6 Hz. This model is written in Modelica code and saved in the file RLC_Circuit.mo and is depicted in below.

Figure 3.1. Electric Circuit



To use the functionality provided by the JModelica.org platform they first have to be imported into the Python script. So we start by importing the following:

```
from jmodelica import simulate
import pylab as P
```

The method 'simulate' is the high-level simulation method and the 'pylab' package is used here for plotting.

Next, we need to provide the 'simulate' method with information about which model we would like to simulate and where it is stored. We also need information about the simulation interval. The information is then passed down in the following way,

```
(jmi_mod, res) = simulate(model='RLC_Circuit_Square', file_name='RLC_Circuit.mo',
                          alg_args={'start_time':0.0,'final_time':20.0,'num_communication_points':0})
```

The return arguments from 'simulate' are the compiled model and the simulation results. Here 'alg_args' are the arguments for the algorithm stored in a dictionary. The 'num_communication_points' represents the number of communication points stored by the algorithm. The default is 500 points and when set to zero (0), the internal steps calculated by the algorithm are stored. If the problem requires that the default options in the specific solver needs to be changed, they should be passed down in a dictionary called 'solver_args'. Typically these options can be the tolerances. Using the default simulation package, Assimulo, information regarding which algorithms are supported and the solver arguments can be found here, http://www.jmodelica.org/assimulo . The default solver is IDA.

After a successful simulation the statistics are printed in the prompt and the results are stored in the variable 'res'. To view the result, we have to retrieve information about the variables we are interested of. This is easily done in the following way,

```
square_y = res.get_variable_data('y')
```

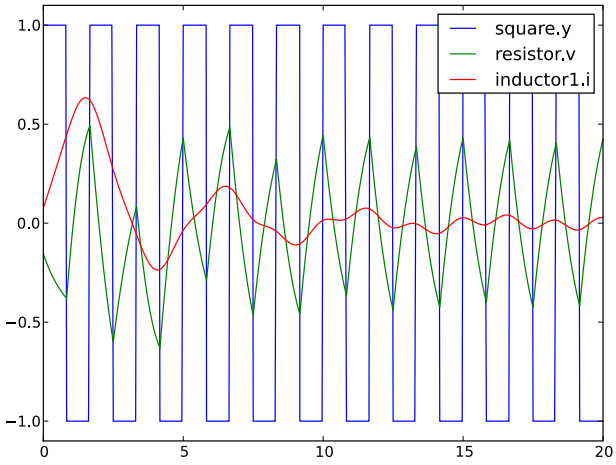
```
resistor_v = res.get_variable_data('resistor.v')
inductor1_i = res.get_variable_data('inductor1.i')
```

And then plotted with the help from pylab,

```
P.plot(square_y.t, square_y.x, resistor_v.t, resistor_v.x, inductor1_i.t, inductor1_i.x)
P.legend(('square.y','resistor.v','inductor1.i'))
P.show()
```

The simulation result is shown in the figure below:

Figure 3.2. Simulation result



3. Solving optimal control problems

3.1. The van der Pol Oscillator

We consider the following Optimica model:

```
optimization VDP_Opt (objective = cost(finalTime),
                      startTime = 0,
                      finalTime = 20)

  // The states
  Real x1(start=0,fixed=true);
  Real x2(start=1,fixed=true);

  // The control signal
  input Real u;

  Real cost(start=0,fixed=true);

equation
  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = x1;
  der(cost) = x1^2 + x2^2 + u^2;
constraint
   u<=0.75;
end VDP_Opt;
```

Create a new file named VDP_Opt.mo and save it in you working directory. Next, create a Python script file and a write (or copy paste) the following commands:

```
# Import the optimize function
from jmodelica import optimize

# Import the plotting library
```

```
import matplotlib.pyplot as plt
```

Next, we call the 'optimize' function which encapsulates operations for compiling, loading, executing the optimization algorithm, and loading the result from file:

```
(model, res) = optimize("VDP_Opt", VDP_Opt.mo")
```

In this case, we use the default settings for the optimization algorithm and provide only the name of the Optimica class (the first argument) and the name of the file (VDP.mo). The return argument 'model' is a reference to a jmodelica.Model object representing the compiled model. The 'res' return argument represents the optimization result and can be used to access the optimal trajectories:

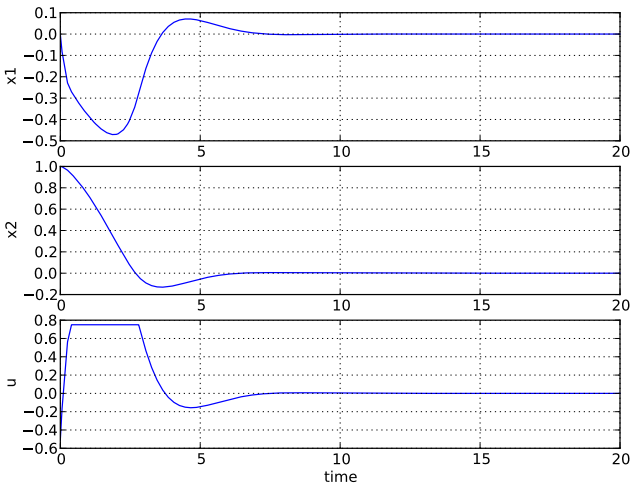
```
x1=res.get_variable_data('x1')
x2=res.get_variable_data('x2')
u=res.get_variable_data('u')
```

The return arguments are objects of the Python class jmodelica.io.Trajectory, which has two fields: 't' which represents the time vector and 'x' which represents the trajectory vector. t and x are both numpy arrays of the same length. Using the matplotlib package, we can visualize the optimization result:

```
plt.figure(1)
plt.clf()
plt.subplot(311)
plt.plot(x1.t,x1.x)
plt.grid()
plt.ylabel('x1')

plt.subplot(312)
plt.plot(x2.t,x2.x)
plt.grid()
plt.ylabel('x2')

plt.subplot(313)
plt.plot(u.t,u.x)
plt.grid()
plt.ylabel('u')plt.xlabel('time')
plt.show()
```

You should now see the optimization result as shown below.

Figure 3.3. Van der Pol optimization result.



3.2. The Hicks Ray Continuously Stirred Reactor (CSTR)

This example is based on the Hicks-Ray Continuously Stirred Tank Reactors (CSTR) system. The model was originally presented in [1]. The system has two states, the concentration, c, and the temperature, T. The control

input to the system is the temperature, Tc, of the cooling flow in the reactor jacket. The chemical reaction in the reactor is exothermic, and also temperature dependent; high temperature results in high reaction rate. The CSTR dynamics is given by:

$$\dot{c}(t) = \frac{F_0(c_0 - c(t))}{V} - k_0 c(t) e^{-\mathrm{EdivR}/T(t)}$$

$$\dot{T}(t) = \frac{F_0(T_0 - T(t))}{V} - \frac{dHk_0 c(t)}{\rho C_p} e^{-\mathrm{EdivR}/T(t)} + \frac{2U}{r\rho C_p}(Tc(t) - T(t))$$

This tutorial will cover the following topics:

- How to solve a DAE initialization problem. The initialization model have equations specifying that all derivatives should be identically zero, which implies that a stationary solution is obtained. Two stationary points, corresponding to different inputs, are computed. We call the stationary points A and B respectively. Point A corresponds to operating conditions where the reactor is cold and the reaction rate is low, whereas point B corresponds to a higher temperature where the reaction rate is high. For more information about the DAE initialization algorithm, see the JMI API documentation.
- An optimal control problem is solved where the objective is to transfer the state of the system from stationary point A to point B. The challenge is to ignite the reactor while avoiding uncontrolled temperature increase. It is also demonstrated how to set parameter and variable values in a model. More information about the simultaneous optimization algorithm can be found at JModelica.org API documentation.
- The optimization result is saved to file and then the important variables are plotted.

The Python commands in this tutorial may be copied and pasted directely into a Python shell, in some cases with minor modifications. Alternatively, you may copy the commands into a text file, e.g., cstr.py.

Start the tutorial by creating a working directory and copy the file $JMODELICA_HOME/Python/jmodelica/examples/files/CSTR.mo to your working directory. An on-line version of CSTR.mo is also available (depending on which browser you use, you may have to accept the site certificate by clicking through a few steps). If you choose to create Python script file, save it to the working directory.

3.2.1. Start the Python shell

Next, open a Python shell, preferably using the Pylab mode. If you are running Windows, select the menu option provided with the JModelica.org installation. If you are running Linux or Mac OS X, open a terminal and enter the command:

```
> /path/to/jmodelica_installation/Python/jm_ipython.sh -pylab
```

As your first action, go to the working directory you have created:

```
In [1]: cd '/path/to/working/directory'
```

In order to run the Python script, use the 'run' command:

```
in [2]: run cstr.py
```

3.2.2. Compile and instantiate a model object

The functions and classes used in the tutorial script need to be imported into the Python script. This is done by the following Python commands. Copy them and past them either directly into you Python shell or, preferably, into your Python script file.

```
import os.path
from jmodelica import initialize
from jmodelica import simulate
from jmodelica import optimize

import jmodelica.jmi as jmi
```

```
from jmodelica.compiler import OptimicaCompiler

import numpy as N
import matplotlib.pyplot as plt
```

Before we can do operations on the model, such as optimizing it, the model file must be compiled and the resulting DLL file loaded in Python. These steps are described in more detail in the tutorial on Compilation of models.

```
# Create a Modelica compiler instance
oc = OptimicaCompiler()

# Compile the stationary initialization model into a DLL and load it
init_model = oc.compile_model("CSTR.CSTR_Init", "CSTR.mo", target='ipopt')
```

At this point, you may open the file CSTR.mo, containing the CSTR model and the static initialization model used in this section. Study the classes CSTR.CSTR and CSTR.CSTR_Init and make sure you understand the models. Before proceeding, have a look at the interactive help for one of the functions you used:

```
In [8]: help(oc.compile_model)
```

3.2.3. Solve the DAE initialization problem

In the next step, we would like to specify the first operating point, A, by means of a constant input cooling temperature, and then solve the initialization problem assuming that all derivatives are zero.

```
# Set inputs for Stationary point A
Tc_0_A = 250
init_model.set_value('Tc',Tc_0_A)

# Solve the DAE initialization system with Ipopt
(init_model, init_result) = initialize(init_model)

# Store stationary point A
c_0_A = init_result.get_variable_data('c').x[0]
T_0_A = init_result.get_variable_data('T').x[0]

# Print some data for stationary point A
print(' *** Stationary point A ***')
print('input Tc = %f' % Tc_0_A)
print('state c = %f' % c_0_A)
print('state T = %f' % T_0_A)
```

Notice how the function set_value is used to set the value of the control input. The initialization algorithm is invoked by calling the function 'initialize', which returns the initialization result in the object 'init_result'. The 'initialize' function relies on the algorithm Ipopt for computing the solution of the initialization problem. The values of the states corresponding to grade A can then be extracted from the result object. Look carefully at the printouts in the Python shell to see a printout of the stationary values. Display the help text for the 'initialize' function and take a moment to look through it. The procedure is now repeated for operating point B:

```
# Set inputs for Stationary point B
Tc_0_B = 280
init_model.set_value('Tc',Tc_0_B)

# Solve the DAE initialization system with Ipopt
(init_model, init_result) = initialize(init_model)

# Store stationary point B
c_0_B = init_result.get_variable_data('c').x[0]
T_0_B = init_result.get_variable_data('T').x[0]

# Print some data for stationary point B
print(' *** Stationary point B ***')
print('input Tc = %f' % Tc_0_B)
print('state c = %f' % c_0_B)
print('state T = %f' % T_0_B)
```

We have now computed two stationary points for the system based on constant control inputs.

3.2.4. Solving an optimal control problem

The optimal control problem we are about to solve is given by:

$$\min_{u(t)} \int_0^{150} \left(c^{ref} - c(t)\right)^2 + \left(T^{ref} - T(t)\right)^2 + \left(T_c^{ref} - T_c(t)\right)^2 dt$$

$$\text{subject to}$$

$$230 \leq u(t) \leq 370$$

$$T(t) \leq 350$$

and is expressed in Optimica format in the class CSTR.CSTR_Opt in the CSTR.mo file above. Have a look at this class and make sure that you understand how the optimization problem is formulated and what the objective is.

Direct collocation methods often require good initial guesses in order to ensure robust convergence. Since initial guesses are needed for all discretized variables along the optimization interval, simulation provides a convenient mean to generate state and derivative profiles given an initial guess for the control input(s). It is then convenient to set up a dedicated model for computation of initial trajectories. In the model CSTR.CSTR_Init_Optimization in the CSTR.mo file, a step input is filtered through a first order filter in order to generate a smooth input for the CSTR system. The filtering is done in order not to excite unstable modes of the system, and in particular to avoid sudden ignition. Notice also that the variable names in the initialization model must match those in the optimal control model. Therefore, also the cost function is included in the initialization model.

Start by creating an input trajectory to be passed to the simulator:

```
# Create the time vector
t = N.linspace(1,150.,100)
# Create the input vector from the target input value. The
# target input value is here increased in order to get a
# better initial guess.
u = (Tc_0_B+35)*N.ones(N.size(t,0))
# Create a matrix where the first column is time and the second column represents
# the input trajectory.
u_traj = N.transpose(N.vstack((t,u)))
```

Next, compile the model and set model parameters:

```
# Compile the optimization initialization model and load the DLL
init_sim_model = oc.compile_model("CSTR.CSTR_Init_Optimization", "CSTR.mo", target='ipopt')

# Set model parameters
init_sim_model.set_value('cstr.c_init',c_0_A)
init_sim_model.set_value('cstr.T_init',T_0_A)
init_sim_model.set_value('Tc_0',Tc_0_A)
init_sim_model.set_value('c_ref',c_0_B)
init_sim_model.set_value('T_ref',T_0_B)
init_sim_model.set_value('Tc_ref',u[0])
```

Having initialized the model parameters, we can simulate the model using the 'simulate' function.

```
(init_sim_model,res) = simulate(init_sim_model,alg_args={'start_time':0.,'final_time':150.,
                                                         'input_trajectory':u_traj})
```

The function 'simulation' first computes consistent initial conditions and then simulates the model in the interval 0 to 150 seconds with the input trajectory specified by 'u_traj'. Notice that the arguments to the simulation function is specified in a Python dictionary. Take a moment to read the interactive help for the 'simulate' function.

The simulation result is returned in the output argument 'res', from which you may now retrieve trajectories for plotting:

```
# Extract variable profiles
c_init_sim=res.get_variable_data('cstr.c')
T_init_sim=res.get_variable_data('cstr.T')
```

```
Tc_init_sim=res.get_variable_data('cstr.Tc')

# Plot the results
plt.figure(1)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(c_init_sim.t,c_init_sim.x)
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(T_init_sim.t,T_init_sim.x)
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(Tc_init_sim.t,Tc_init_sim.x)
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

Look at the plots and make sure you understand the effect of the filter. Think about alternative, better ways to chose the input profile. Also, try to increase the value 35 that was added to the target input: how much can you increase this value without experiencing sudden ignition of the reactor?

Once the initial guess is generated, we compile the model containing the optimal control problem:

```
cstr = oc.compile_model("CSTR.CSTR_Opt", "CSTR.mo", target='ipopt')
```

We will now initialize the parameters of the model so that their values correspond to the optimization objective of transferring the system state from operating point A to operating point B. Accordingly, we set the parameters representing the initial values of the states to point A and the reference values in the cost function to point B:

```
cstr.set_value('Tc_ref',Tc_0_B)
cstr.set_value('c_ref',c_0_B)
cstr.set_value('T_ref',T_0_B)

cstr.set_value('cstr.c_init',c_0_A)
cstr.set_value('cstr.T_init',T_0_A)
```

In order to solve the optimization problem, we need to specify the mesh on which the optimization is performed. The simultaneous optimization algorithm is based on a collocation method that corresponds to a fixed step implicit Runge-Kutta scheme, where the mesh defines the length of each step. Also, the number of collocation points in each element, or step, needs to be provided. This number corresponds to the stage order of the Runge-Kutta scheme. The selection of mesh is analogous to the choice of step length in a one-step algorithm for solving differential equations. Accordingly, the mesh needs to be fine-grained enough to ensure sufficiently accurate approximation of the differential constraint. For an overview of simultaneous optimization algorithms, see [2].

Collocation-based optimization algorithms often require a good initial guess in order to achieve fast convergence. Also, if the problem is non-convex, initialization is even more critical. Initial guesses can be provided in Optimica by the 'initialGuess' attribute, see the CSTR.mo file for an example for this. Notice that initialization in the case of collocation-based optimization methods means initialization of all the control and state profiles as a function of time. In some cases, it is sufficient to use constant profiles. For this purpose, the 'initialGuess' attribute works well. In more difficult cases, however, it may be necessary to initialize the profiles using simulation data, where an initial guess for the input(s) has been used to generate the profiles for the dependent variables. This approach for initializing the optimization problem is used in this tutorial.

We are now ready to solve the actual optimization problem. This is done by invoking the method optimize:

```
# Initialize the mesh
n_e = 100 # Number of elements
hs = N.ones(n_e)*1./n_e # Equidistant points
n_cp = 3; # Number of collocation points in each element
```

```
(cstr,res) = optimize(cstr,alg_args={'n_e':n_e,'hs':hs,'n_cp':n_cp,'init_traj':res})
```

You should see the output of Ipopt in the Python shell as the algorithm iterates to find the optimal solution. Ipopt should terminate with a message like 'Optimal solution found' or 'Solved to an acceptable level' in order for an optimum to be found. Again, the arguments to the algorithm (number of elements, number of collocation points, element length vector and initial guess object) are given in a Python dictionary. The optimization result is returned in the output argument 'res'.

We can now retrieve the trajectories of the variables that we intend to plot:

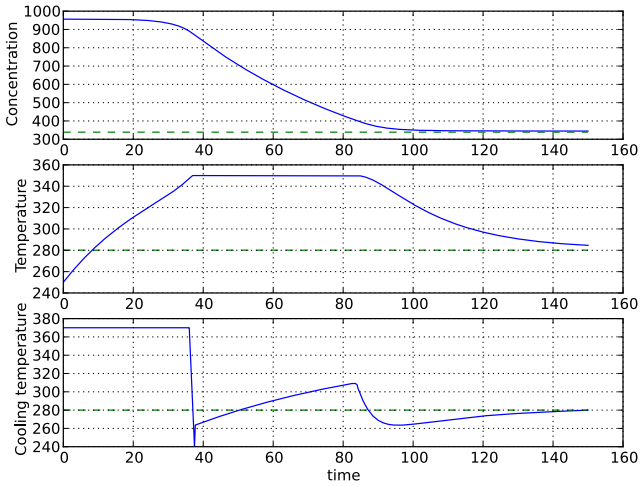
```
# Extract variable profiles
c_res=res.get_variable_data('cstr.c')
T_res=res.get_variable_data('cstr.T')
Tc_res=res.get_variable_data('cstr.Tc')

c_ref=res.get_variable_data('c_ref')
T_ref=res.get_variable_data('T_ref')
Tc_ref=res.get_variable_data('Tc_ref')
```

Finally, we plot the result using the functions available in matplotlib:

```
plt.figure(1)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(c_res.t,c_res.x)
plt.plot(c_ref.t,c_ref.x,'--')
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(T_res.t,T_res.x)
plt.plot(T_ref.t,T_ref.x,'--')
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(Tc_res.t,Tc_res.x)
plt.plot(Tc_ref.t,Tc_ref.x,'--')
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

Your should now see a plot as the one below:

Figure 3.4. Optimization result



Take a minute to analyze the optimal profiles and to answer the following questions:

1. Why is the concentration high in the beginning of the interval?

2. Why is the input cooling temperature high in the beginning of the interval?

3.2.5. Verify optimal control solution

Solving optimal control problems by means of direct collocation implies that the differential equation is approximated by a discrete time counterpart. The accuracy of the solution is dependent on the method of collocation and the number of elements. In order to assess the accuracy of the discretization, we may simulate the system using a DAE solver using the optimal control profile as input. With this approach, the state profiles are computed with high accuracy and the result may then be compared with the profiles resulting from optimization. Notice that this procedure does not verify the optimality of the resulting optimal control profiles, but only the accuracy of the discretization of the dynamics.

The procedure for setting up and executing this simulation is similar to above:

```
# Simulate to verify the optimal solution
# Set up input trajectory
t = Tc_res.t
u = Tc_res.x
u_traj = N.transpose(N.vstack((t,u)))

# Comile the Modelica model first to C code and
# then to a dynamic library
sim_model = oc.compile_model("CSTR.CSTR","CSTR.mo",target='ipopt')

sim_model.set_value('c_init',c_0_A)
sim_model.set_value('T_init',T_0_A)
sim_model.set_value('Tc',u[0])

(sim_model,res) = simulate(sim_model,compiler='optimica',
                           alg_args={'start_time':0.,'final_time':150.,
                                     'input_trajectory':u_traj})
```

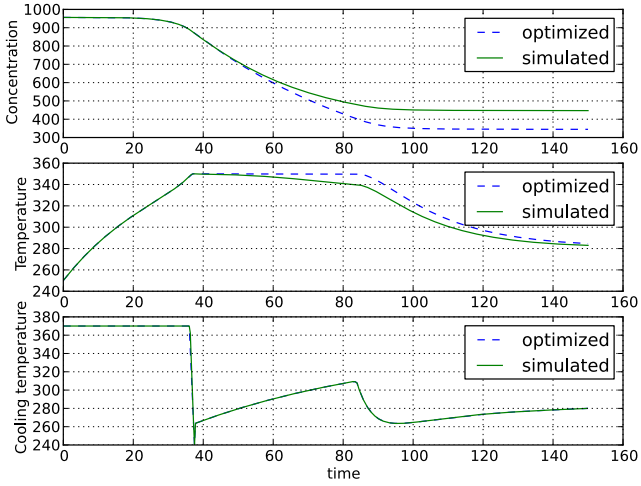
Finally, we load the simulated data and plot it to compare with the optimized trajectories:

```
# Extract variable profiles
c_sim=res.get_variable_data('c')
T_sim=res.get_variable_data('T')
Tc_sim=res.get_variable_data('Tc')

# Plot the results
plt.figure(3)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(c_res.t,c_res.x,'--')
plt.plot(c_sim.t,c_sim.x)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(T_res.t,T_res.x,'--')
plt.plot(T_sim.t,T_sim.x)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(Tc_res.t,Tc_res.x,'--')
plt.plot(Tc_sim.t,Tc_sim.x)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Cooling temperature')
```

```
plt.xlabel('time')
plt.show()
```

You should now see a plot similar to:

Figure 3.5. Simulated system response



Discuss why the simulated trajectories differs from the optimized counterparts.

3.2.6. Exercises

After completing the tutorial you may continue to modify the optimization problem and study the results.

1. Remove the constraint on cstr.T. What is then the maximum temperature?

2. Play around with weights in the cost function. What happens if you penalize the control variable with a larger weight? Do a parameter sweep for the control variable weight and plot the optimal profiles in the same figure.

3. Add terminal constraints ('cstr.T(finalTime)=someParameter') for the states so that they are equal to point B at the end of the optimization interval. Now reduce the length of the optimization interval. How short can you make the interval?

4. Try varying the number of elements in the mesh and the number of collocation points in each interval. 2-10 collocation points are supported.

3.2.7. References

[1] G.A. Hicks and W.H. Ray. Approximation Methods for Optimal Control Synthesis. *Can. J. Chem. Eng.*, 40:522–529, 1971.

[2] Bieger, L., A. Cervantes, and A. Wächter (2002): "Advances in simultaneous strategies for dynamic optimization." *Chemical Engineering Science*, **57**, pp. 575-593.

4. Working with file I/O

In this tutorial you will learn how to load simulation/optimization results.

4.1. I/O functionality

The module jmodelica.io provides useful functions for exporting and loading simulation or optimization results from Dymola. The result files can be in Dymola textural or Dymola binary format. The variable data is saved

together with the variable names which makes it possible to load the result files and match result data with a specific variable.

4.2. Loading result data

To load a result data file saved using the export functionality jmodelica.io.export_result_dymola the class ResultDymolaTextual in the same module, jmodelica.io, can be used. The result object can then be used to retrieve data for a specific variable.

```
# Load the CSTR results
res = jmodelica.io.ResultDymolaTextual('CSTR_CSTR_Opt_result.txt')

# Get variable data for T_ref
res.get_variable_data('T_ref').x
>> array([ 320., 320.])
```

There is a similar function for retrieving results from a file in Dymola binary format.

5. Setting and saving model parameters

This tutorial shows how to set model parameters and how to load and save parameter data from/to XML files.

5.1. Model parameter XML files

The model parameter meta data and values are saved in XML files which are generated during the compilation. They follow the name convention:

- <model class name>.xml
- <model class name>_values.xml

The parameter meta data is saved in <model class name>.xml and the parameter values in <model class name>_values.xml. The name of the parameter is used to map a parameter value in the values file to a parameter specification.

5.2. Get and set value

The model parameters can be accessed with via the jmi.Model interface. It is possible to look at the whole vector of, for example, all real parameters in the model or one specific parameter. Accessing one specific parameter requires that the parameter name is known.

The following code example assumes the CSTR model has been compiled and the DLL file loaded in jmi.Model.

```
# Get independent real parameter vector
cstr.get_real_pi()
>> array([ 1.66666667e-03, 1.00000000e+03, 1.66666667e-03,
           3.50000000e+02, 2.19000000e-01, 1.20000000e+09,
           8.75000000e+03, 9.15600000e+02, 1.00000000e+03,
           2.39000000e+02, -5.00000000e+04, 1.00000000e+02,
           1.00000000e+03, 3.50000000e+02, 5.00000000e+02,
           3.20000000e+02, 3.00000000e+02, 1.00000000e+00,
           1.00000000e+00, 1.00000000e+00])

# Get independent parameter c_ref
cstr.get_value('c_ref')
>> 500.0

# Set independent parameter
cstr.set_value('c_ref', 450)
# c_ref has now changed
cstr.get_value('c_ref')
>> 450.0
```

5.3. Loading from and saving to XML

5.3.1. Loading XML values file

It is possible to load the values from an XML file as is done automatically when the jmi.Model object was first created. If, for example, there were many local changes to parameters it could be desirable to reset everything as it was from the beginning.

```
# Load values XML file
cstr.load_parameters_from_XML()
```

Default behaviour is to load the same file as was created during compilation. If another file should be used this must be passed to the method.

```
# Load other XML file
cstr.load_parameters_from_XML('new_values.xml')
```

5.3.2. Writing to XML values file

Setting a parameter value with Model.set_value only changes the value in the vector loaded when jmi.Model was created, which means that they will not be saved. To save all changes made to parameters in a model, the values have to be written to the XML values file.

```
# Save parameters to values XML
cstr.write_parameters_to_XML()
```

If this method is called without arguments the values will be written to the XML file which was created when the model was compiled (following the name conventions mentioned above). It is also possible to save the changes in a new XML file. This is quite convenient since different parameter value settings can easily be saved and loaded in the model.

```
# Save to specific XML file
cstr.write_parameters_to_XML('test_values.xml')
```

Chapter 4. FMI Interface

FMI (Functional Mock-up Interface) is a standard for exchanging models between different modelling and simulation environments. FMI defines a C-interface and a set of C-functions for which the communication between the model and the simulation environment is to be handled. Models are currently described as ODEs with time-, state and step-events with the intention to extend the standard to also cover DAEs. A model is distributed in a zip-file with the extension '.fmu' and consists of several files. The important ones are mainly the '.xml'-file, which contains the definition of all variables and then the C-functions which can either be provided in source and/or binary form. For more information regarding the FMI standard, please visit http://www.functional-mockup-interface.org/.

1. Overview of JModelica.org FMI package

Our FMI-interface is written in Python and is intended to be a close copy of the defined C-interface for an FMU. The interface is located in "jmodelica.fmi" and consist of the class FMIModel together with methods for unzipping the FMU and for writing the result. Connected to this interface is a wrapper for JModelica.org's simulation package to enable an easy simulation of the FMUs. The simulation wrapper is located in "jmodelica.simulation.assimulo", FMIODE.

Below is a list of the defined FMI C-interface and its counterpart in our Python package. There is also one-to-one map to the C-functions. These functions are also located in FMIModel and is named with a leading underscore together with the same name as specified in the standard.

FMI C-Interface	FMI Python Interface
const char* fmiGetModelTypesPlatform()	string FMIModel.get_validplatforms()
const char* fmiGetVersion()	string FMIModel.get_version()
fmiComponent fmiInstantiateModel(...)	FMIModel.__init__
struct fmiCallbackFunctions	...
void fmiFreeModelInstance(fmiComponent c)	FMIModel.__del__
fmiStatus fmiSetDebugLogging(...)	...
fmiStatus fmiSetTime(...)	FMIModel.t (property)
fmiStatus fmi(Get/Set)ContinuousStates(...)	FMIModel.real_x (property)
fmiStatus fmiCompletedIntegratorStep(...)	boolean FMIModel.step_event()
fmiStatus fmiSetReal/Integer/Boolean/String(...)	none FMIModel.set_real/integer/boolean/string(valueref,values)
fmiStatus fmiInitialize(...)	none FMIModel.initialize() (also sets the start attributes)
struct fmiEventInfo	FMIModel.event_info (get property)
fmiStatus fmiGetDerivatives(...)	numpy.array FMIModel.real_dx (get property)
fmiStatus fmiGetEventIndicators(...)	numpy.array FMIModel.event_ind (get property)
fmiStatus fmiGetReal/Integer/Boolean/String(...)	numpy.array FMIModel.get_real/integer/boolean/string(valueref)
fmiStatus fmiEventUpdate(...)	none FMIModel.update_event()
fmiStatus fmiGetNominalContinuousStates(...)	FMIModel.real_x_nominal (get property)
fmiStatus fmiGetStateValueReferences(...)	numpy.array FMIModel.get_continuous_value_reference(self):
fmiStatus fmiTerminate(...)	FMIModel.__del__

It is also recommended to have a look at the interactive help from IPython by using for instance,

```
FMIModel.get_real?
```

2. Examples

In the next two sections it will be shown how to use the JModelica.org platform both for simulation of a FMU using the raw Python interface and also to simulate a FMU using JModelica.org's simulation package.

The Python commands in these examples may be copied and pasted directly into a Python shell, in some cases with minor modifications. Alternatively, they may be copied into a text file, which also is the recommended way.

2.1. Example using the raw interface

This example shows how to use the raw (JModelica.org) FMI interface for simulation of an FMU. The FMU that is to be simulated is the bouncing ball example from Qtronics FMU SDK. This example is written similar to the example in the documentation of the 'Functional Mock-up Interface for Model Exchange' version 1.0 (http://www.functional-mockup-interface.org/). The example is to be simulated using explicit Euler.

The bouncing ball consists of two equations,

der(h) = v

der(v) = -g

and one event function, h < 0 where we turn,

v = -e*v.

Here 'h' is the height, g the gravity and v the velocity. The starting values are, h=1, v=0, e=0.7 and g = 9.81.

2.1.1. Implementation

We first start by importing the necessary modules,

```
import numpy as N
import pylab as P #Used for plotting
from jmodelica.fmi import FMIModel #The FMI Interface
```

Next, we have to load the FMU and initialize the model,

```
#Load the FMU by specifying the fmu and the directory
bouncing_fmu = FMIModel('bouncingBall.fmu', '/path/to/FMU/')

Tstart = 0.5 #The start time.
Tend   = 3.0 #The final simulation time.

bouncing_fmu.t = Tstart #Set the start time before the initialization.

bouncing_fmu.initialize() #Initialize the model. Also sets all the start attributes
                          #defined in the XML file.
```

The first line loads the FMU and connects the C-functions of the model to Python together with loading the information from the '.xml'-file. Then we define the start-time which is default to the start-time in the '.xml'-file and if there is no start-time in the '.xml' it defaults to zero. The model is also initialized, which is needed before we start our simulation.

```
#Get Continuous States
```

```
x = bouncing_fmu.real_x
#Get the Nominal Values
x_nominal = bouncing_fmu.real_x_nominal
#Get the Event Indicators
event_ind = bouncing_fmu.event_ind

#Values for the solution
t_sol = [Tstart]
h_sol = [bouncing_fmu.get_real([0])]
```

Here we have retrieved the continuous states together with the nominal values and the event indicators. In our case the nominal values are all equal to one. This information is available in the '.xml'-file. We also create lists which is used for storing the result. What's left before we can start the integration loop is to define the step-size.

```
time = Tstart
Tnext = Tend #Used for time events
dt = 0.01 #Step-size
```

We are now ready to create our main integration loop.

```
#Main integration loop.
while time < Tend and not bouncing_fmu.event_info.terminateSimulation:
    #Compute the derivative
    dx = bouncing_fmu.real_dx

    #Advance
    time = time + h

    #Set the time
    bouncing_fmu.t = time

    #Set the inputs at the current time (if any)
    #bouncing_fmu.set_real,set_integer,set_boolean,set_string (valueref, values)

    #Set the states at t = time (Perform the step)
    x = x + h*dx
    bouncing_fmu.real_x = x
```

This is the integration loop for advancing the solution one step using the explicit Euler method. We loop until we have reached the final time or if the FMU reported that the simulation is to be terminated (event_info.terminateSimulation). In the start of the loop the derivatives of the continuous states are retrieved and then the simulation time is just incremented by the step-size and set to the model. It could also be the case that the model is depended on inputs which can be set using the set_(real/...) methods.

Note that only variables defined in the '.xml'-file to be inputs can be set using the set_(real/...) methods.

The step is performed by calculating the new states (x+h*dx) and setting the values into the model.

```
    #Get the event indicators at t = time
    event_ind_new = bouncing_fmu.event_ind

    #Inform the model about an accepted step and check for step event.
    step_event = bouncing_fmu.step_event()

    #Check for time and state events
    time_event  = abs(time-Tnext) <= 1.e-10
    state_event = True if bool(event_ind_new[0]>0.0) != bool(event_ind[0]>0.0) else False
```

We also have to monitor and check for events during the simulation. Events can be of either type, time-, state- or step events. The time events are check by continuously monitor the current time and the time where the next time

event (Tnext) occurs towards a small epsilon. State events are basically checked against sign changes of the event functions. Step events are monitored in the FMU, in the method fmiCompletedIntegratorStep and return True if any event handling is necessary. If an event have occurred, it needs to handled and an example is shown below,

```
        #Event handling
        if step_event or time_event or state_event:

            eInfo = bouncing_fmu.event_info
            eInfo.iterationConverged = False

            #Event iteration
            while eInfo.iterationConverged == False:
                bouncing_fmu.update_event()
                eInfo = bouncing_fmu.event_info

                #Retrieve solutions (if needed)
                if eInfo.iterationConverged == False:
                    #bouncing_fmu.get_real,get_integer,get_boolean,get_string (valueref)
                    pass

            #Check if the event affected the state values and if so sets them
            if eInfo.stateValuesChanged:
                x = bouncing_fmu.real_x

            #Get new nominal values.
            if eInfo.stateValueReferencesChanged:
                #atol = 0.01*rtol*bouncing_fmu.real_x_nominal
                pass

            #Check for new time event
            if eInfo.upcomingTimeEvent:
                Tnext = min(eInfo.nextEventTime, Tend)
            else:
                Tnext = Tend
```

So if an event occurred, we enter the iteration loop where we loop until the solution of the new states have converged. During this iteration we can also retrieve the intermediate values with the normal 'get' methods. At this point 'eInfo' contains information about the changes made in the iteration. If the state values have changed, they are retrieved and if the state references have changed for example in the case with dynamic state selection, new absolute tolerances are calculated with the new nominal values. Finally the model is checked for a new time event.

```
        event_ind = event_ind_new

        #Retrieve solutions at t=time for outputs
        #bouncing_fmu.get_real,get_integer,get_boolean,get_string (valueref)

        t_sol += [time]
        h_sol += [bouncing_fmu.get_real([0])]
```

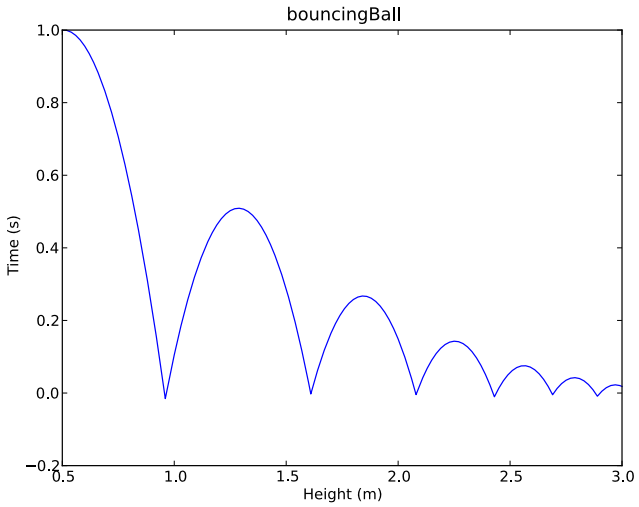
In the end of the loop, the solution is stored and the old event indicators are stored for use in the next loop.

Finally we can plot the solution,

```
    #Plot the solution
    P.plot(t_sol,h_sol)
    P.title(bouncing_fmu.get_name())
    P.xlabel('Height (m)')
    P.ylabel('Time (s)')
    P.show()
```

and the figure below shows the results.

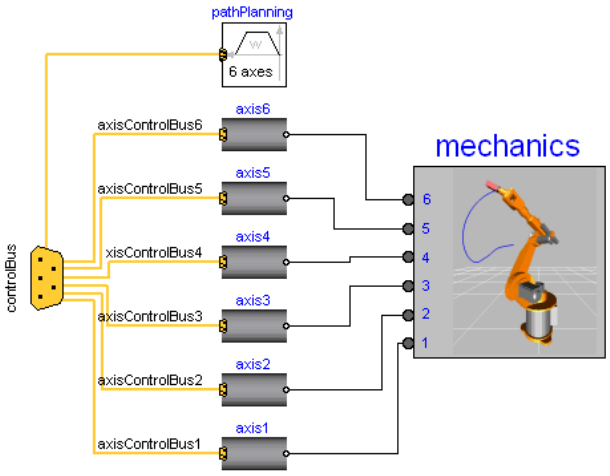
Figure 4.1. Simulation result



2.2. Example using a compiled FMU

This example will show how to use the JModelica.org's FMI-interface together with its simulation package. The FMU to be simulated is the full Robot from the Modelica standard library examples. It consists of brakes, motors, gears and path planning. The model consists of 36 continuous states together with 98 event functions and also a few thousand constants/parameters. The FMU was generated using Dymola.

Figure 4.2. Full Robot



2.2.1. Implementation

We first start by importing the necessary modules,

```
import pylab as P
from jmodelica.fmi import FMIModel
from jmodelica.simulation.assimulo import FMIODE, write_data
```

```
from jmodelica.io import ResultDymolaTextual
from Assimulo.Explicit_ODE import CVode
```

Then we load and initialize the FMU.

```
model = FMIModel('fullRobot.fmu')
model.initialize()
```

We can also retrieve the tolerances specified in the '.xml'-file, shown below. If the tolerances are not specified in the '.xml'-file it defaults to 1.0e-4. The absolute tolerances are defined to be, according to the FMI specification, 0.01*rtol*x_nominal.

```
rtol, atol = model.get_tolerances()

modRobot = FMIODE(model)
simRobot = CVode(modRobot)
```

Here we also created our problem from the model, FMIODE and then also created the solver from the problem, CVode. Then we set the attributes to the solver.

```
simRobot.rtol  = rtol #Set the relative tolerance
simRobot.atol  = atol #Set the absolute tolerances
simRobot.iter  = 'Newton' #Set the iteration to Newton, default functional iteration
simRobot.discr = 'BDF' #Set the method to BDF, default Adams
```

Here we set the relative and absolute tolerances retrieved from the models default experiment. We also changed from the default solver, Adams to BDF and the iteration to Newton.For more information about the solver attributes see, http://www.jmodelica.org/assimulo. Now we are ready to simulate the problem, we want to simulate from time=0.0 to 1.8 with 1000 communication points so we give just that information to the simulate method (default starting time is 0.0).

```
simRobot.simulate(1.8,1000)
```

The simulation statistics will be printed in the prompt. To store the result data on file, simply use the method write_result in the jmodelica.simulation.assimulo package.

```
write_result(simRobot)
```

The result can then be loaded using the method ResultDymolaTextual from the I/O module.

```
res = ResultDymolaTextual('fullRobot_result.txt')
```

To retrieve data about a variable from the result object (res), just use the method get_variable_data,

```
dq1 = res.get_variable_data('der(mechanics.q[1])')
dq6 = res.get_variable_data('der(mechanics.q[6])')
```

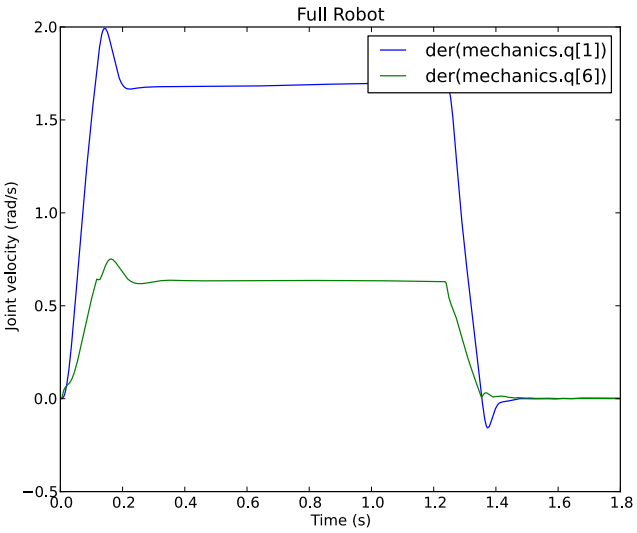
Now we have loaded and retrieved the variables of interest. So lets plot them.

```
P.plot(dq1.t,dq1.x,dq6.t,dq6.x)
P.legend(['der(mechanics.q[1])','der(mechanics.q[6])'])
P.xlabel('Time (s)')
```

```
    P.ylabel('Joint Velocity (rad/s)')
    P.title('Full Robot')
    P.show()
```

Below is the resulting figure.

Figure 4.3. Full Robot Results



3. Limitations

Cannot compile FMUs distributed in source code.

Chapter 5. Advanced topics

1. Tutorial on Abstract Syntax Trees (ASTs)

1.1. About Abstract Syntax Trees

A fundamental data structure in most compilers is the Abstract Syntax Tree (AST). An AST serves as an abstract representation of a computer program and is often used in a compiler to perform analyses (e.g., binding names to declarations and checking type correctness of a program) and as a basis for code generation.

Three different ASTs are used in the JModelica.org front-ends.

- The source AST results from parsing of the Modelica or Optimica source code. This AST shares the structure of the source code, and consists of a hierarchy consisting of Java objects corresponding to class and component declarations, equations and algorithms. The source AST can also be used for unparsing, i.e., pretty printing of the source code.

- The instance AST represents a particular model instance. Typically, the user selects a class to instantiate, and the compiler then computes the corresponding instance AST. The instance AST differs from the source AST in that in the former case, all components are expanded down to variables of primitive type. An important feature of the instance AST is that it is used to represent modification environments; merging of modifications takes place in the instance AST. As a consequence, all analysis, such as name and type analysis takes is done based on the instance AST.

- The flat AST represents the flat Modelica model. Once the instance AST has been computed, the flat AST is computed simply by traversing the instance AST and collecting all variables of primitive type, all equations and all algorithms. The flat AST is then used, after some transformations, as a basis for code generation.

For more information on how the JModelica.org compiler transforms these ASTs, see the paper "Implementation of a Modelica compiler using JastAdd attribute grammars" by J.Åkesson et. al.

This tutorial demonstrates how the Python interface to the three different ASTs in the compiler can be used. The JPype package is used to create Java objects in a Java Virtual Machine which is seamlessly integrated with the Python shell. The Java objects can be accessed interactively and methods of the object can be invoked.

For more information about the Java classes and their methods used in this example, please consult the API documentation for the Modelica compiler. Notice however that the documentation for the compiler front-ends is still very rudimentary. Also, the interfaces to the source and instance AST will be made more user friendly in upcoming versions.

Three different usages of ASTs are shown:

- Count the number of classes in the Modelica standard library. In this example, a Python function is defined to traverse the source AST which results from parsing of the Modelica standard library.

- Instantiate the CauerLowPassAnalog model. The instance AST for this model is dumped and it is demonstrated how the merged modification environments can be accessed. Also, it is shown how a component redeclaration affects the instance tree.

- Flatten the CauerLowPassAnalog model instance and print some statistics of the flattened Model.

The Python commands in this tutorial may be copied and pasted directely into a Python shell, in some cases with minor modifications. You are, however, strongly encouraged to copy the commands into a text file, e.g., ast_example.py.

Start the tutorial by creating a working directory and copy the file $JMODELICA_HOME/Python/jmodelica/examples/files/CauerLowPassAnalog.mo to your working directory. An on-line version of

CauerLowPassAnalog.mo is also available (depending on which browser you use, you may have to accept the site certificate by clicking through a few steps). If you choose to create Python script file, save it to the working directory. The tutorial is based on a model from the Modelica Standard Library: Modelica.Electrical.Analog.Basic.Examples.CauerLowPassAnalog.

1.2. Start the Python shell

Next, open a Python shell, preferably using the Pylab mode. If you are running Windows, select the menu option provided with the JModelica.org installation. If you are running Linux or Mac OS X, open a terminal and enter the command:

```
> /path/to/jmodelica_installation/Python/jm_ipython.sh -pylab
```

As your first action, go to the working directory you have created:

```
In [1]: cd '/path/to/working/directory'
```

In order to run the Python script, use the 'run' command:

```
In [2]: run -i ast_example.py
```

Notice the '-i' switch which is used in this tutorial in order to avoid loading the Modelica standard library multiple times and thereby preventing the Python shell from running out of memory.

1.3. Load the Modelica standard library

Before we can start working with the ASTs, we need to import the Python packages that will be used

```
# Import library for path manipulations
import os.path

# Import the JModelica.org Python packages
import jmodelica
import jmodelica.jmi as jmi
from jmodelica.compiler import ModelicaCompiler

# Import numerical libraries
import numpy as N
import ctypes as ct
import matplotlib.pyplot as plt

# Import JPype
import jpype

# Create a reference to the java package 'org'
org = jpype.JPackage('org')
```

Also, we need to create an instance of a Modelica compiler in order to compile models:

```
# Create a compiler
mc = ModelicaCompiler()
```

In order to avoid parsing the same file multiple times (we will not change the Modelica file in this tutorial), we will check the variable 'source_root' exists in the shell before we parse the file CauerLowPassAnalog.mo:

```
# Don't parse the file if it har already been parsed.
try:
    source_root.getProgramRoot()
except:
    # Parse the file CauerLowPassAnalog.mo and get the root node
    # of the source AST
    source_root = mc.parse_model("CauerLowPassAnalog.mo")
```

At this point, try the built-in help feature of Python by typing the following command in the shell to see the help text for the function you just used.

```
In [2]: help(mc.parse_model)
```

In the first part of the tutorial, we will not work with the filter model, but rather load the Modelica standard library. Again, we check if the library has already been loaded:

```
# Don't load the standard library if it is already loaded
try:
    modelica.getName().getID()
except NameError, e:
    # Load the Modelica standard library and get the class
    # declaration AST node corresponding to the Modelica
    # package.
    modelica = source_root.getProgram().getLibNode(1). \
               getStoredDefinition().getElement(0)
```

The means to access the node in the source AST corresponding to the class (package) declaration of the Modelica library is somewhat cumbersome; the source AST interface will be improved in later versions.

1.4. Count the number of classes in the Modelica standard library

Having accessed a node in the source AST, we may now perform analysis by traversing the tree. Say that we are interested in counting the number of classes (packages, models, blocks, functions etc.) in the Modelica standard library. As the basis for traversing the AST, we may use the method ClassDecl.classes() that returns a list of local classes contained in a class. Based on this method, a Python function for traversing the class hierarchy of the source AST can be defined:

```
def count_classes(class_decl,depth):
    """ Count the number of classes hierarchically contained
    in a class declaration."""

    # Get a list of local classes using the method ClassDecl.classes()
    # which returns a Java ArrayList object containing ClassDecl objects.
    local_classes = class_decl.classes()

    # Get the number of local classes.
    num_classes = local_classes.size()

    # Loop over all local classes
    for i in range(local_classes.size()):
        # Call count_classes recursively for all local classes
        num_classes = num_classes + \
                      count_classes(local_classes.get(i),depth + 1)

    # If the class declaration corresponds to a package, print
    # the number of hierarchically contained classes
    if class_decl.getRestriction().getNodeName() == 'MPackage' \
           and depth <= 1:
        print("The package %s has %d hierachically contained classes" \
              %(class_decl.qualifiedName(),num_classes))

    # Return the number of hierachically contained classes
    return num_classes
```

We then call the function:

```
# Call count_classes for 'Modelica'
num_classes = count_classes(modelica,0)
```

Now run the script and study the printouts in the Python shell. The first time the script is run, you will se printouts corresponding also to the compiler accessing individual files of the Modelica standard library; the loading of the library is done on demand as the library classes are actually accessed. Run the script once again (using the '-i' switch), to get a cleaner output, which should now look similar to:

```
The package Modelica.UsersGuide has 16 hierachically contained classes
The package Modelica.Constants has 0 hierachically contained classes
The package Modelica.Icons has 16 hierachically contained classes
The package Modelica.SIunits has 532 hierachically contained classes
```

```
The package Modelica.StateGraph has 64 hierachically contained classes
The package Modelica.Blocks has 258 hierachically contained classes
The package Modelica.Electrical has 361 hierachically contained classes
The package Modelica.Math has 74 hierachically contained classes
The package Modelica.Mechanics has 474 hierachically contained classes
The package Modelica.Media has 1064 hierachically contained classes
The package Modelica.Thermal has 88 hierachically contained classes
The package Modelica.Utilities has 86 hierachically contained classes
The package Modelica has 3045 hierachically contained classes
```

Take some time to ponder the results and make sure that you understand how the Python function 'count_classes' works and which Python variables corresponds to references into the source AST.

1.5. Dump the instance AST

We shall now turn our attention to the CauerLowPassAnalog model. Specifically, we would like to analyze the instance hierarchy of the model by dumping the tree structure to the Python shell. In addition, we will look at the merged modification environment of each instance AST node. Again, we will use methods defined for the Java objects representing the AST.

First we create an instance of the CauerLowPassAnalog filter. Again we only create the instance if it has not already been created:

```
# Don't instantiate if instance has been computed already
try:
    filter_instance.components()
except:
    # Retrieve the node in the instance tree corresponding to the class
    # Modelica.Electrical.Analog.Examples.CauerLowPassAnalog
    filter_instance = mc.instantiate_model(source_root,"CauerLowPassAnalog")
```

Next we define a Python function for traversing the instance AST and printing each node in the shell. We also print the merged modification environment for each instance node. In order to traverse the AST, we use the methods InstNode.instComponentDeclList() and InstNode.instExtendsList(), which both return an object of the class List, which in turn contain instantiated component declarations and instantiated extends clauses. By invoking the 'dump_inst_ast' function recursively for each element in these lists, the instance AST is in effect traversed. Due to the internal representation of the instance AST, nodes of type InstPrimitive, corresponding to primitive variables, are not leaves in the AST as would be expected. To overcome this complication, we simply check if a node is of type InstPrimitive, and if this is the case, the recursion stops.

The environment of an instance node is accessed by calling the method InstNode.getMergedEnvrionment(), which returns a list of modifications. According to the Modelica specification, outer modifications overrides inner modifications, and accordingly, modifications in the beginning of the list has precedence over later modifications.

```
def dump_inst_ast(inst_node, indent):
    """Pretty print an instance node, including its merged enviroment."""

    # Get the merged environment of an instance node
    env = inst_node.getMergedEnvironment()

    # Create a string containing the type and name of the instance node
    str = indent + inst_node.prettyPrint("")
    str = str + " {"

    # Loop over all elements in the merged modification environment
    for i in range(env.size()):
        str = str + env.get(i).toString()
        if i<env.size()-1:
            str = str + ", "
        str = str + "}"

    # Print
    print(str)

    # Get all components and dump them recursively
    components = inst_node.instComponentDeclList
```

```
    for i in range(components.getNumChild()):
        # Assume that primitive variables are leafs in the instance AST
        if (inst_node.getClass() is \
            org.jmodelica.modelica.compiler.InstPrimitive) is False:
            dump_inst_ast(components.getChild(i),indent + "  ")

    # Get all extends clauses and dump them recursively
    extends= inst_node.instExtendsList
    for i in range(extends.getNumChild()):
        # Assume that primitive variables are leafs in the instance AST
        if (inst_node.getClass() is \
            org.jmodelica.modelica.compiler.InstPrimitive) is False:
            dump_inst_ast(extends.getChild(i),indent + "  ")
```

Take a minute and make sure that you understand the essential parts of the function.

Having defined the function 'dump_inst_ast', we call it with the CauerLowPassAnalog instance as an argument.

```
# Dump the filter instance
dump_inst_ast(filter_instance,"")
```

You should now see a rather lengthy printout in your shell window. Let us have a closer look at a few of the instances in the model. First look at the printouts for a resistor in the model:

```
    InstComposite: Modelica.Electrical.Analog.Basic.Resistor R1 {R=1}
      InstPrimitive: SI.Resistance R {=1, start=1, final quantity="Resistance", \
                                      final unit="Ohm"}
      InstExtends: Interfaces.OnePort {R=1}
        InstPrimitive: SI.Voltage v {final quantity="ElectricPotential", final unit="V"}
        InstPrimitive: SI.Current i {final quantity="ElectricCurrent", final unit="A"}
        InstComposite: PositivePin p {}
          InstPrimitive: SI.Voltage v {final quantity="ElectricPotential", final unit="V"}
          InstPrimitive: SI.Current i {final quantity="ElectricCurrent", final unit="A"}
        InstComposite: NegativePin n {}
          InstPrimitive: SI.Voltage v {final quantity="ElectricPotential", final unit="V"}
          InstPrimitive: SI.Current i {final quantity="ElectricCurrent", final unit="A"}
```

The model instance if of type InstComposite, and contains two elements, one primitive variable, R, and one extends clause. The modification environment for the resistor contains a value modification '=1' and some modifications of the built in attributes for the type Real. The InstExtends node contains a number of child nodes, which corresponds to the content of the class Interfaces.OnePort. Notice the difference between the source AST, where an extends node is essentially a leaf in the tree, whereas in the instance tree, the extends clause is expanded.

Let us have a look at the effects of redeclarations in the instance AST. In the CauerLowPassAnalog model, a step voltage signal source is used, which in turn relies on redeclaration of a generic signal source to a step. The instance node for the step voltage source 'V' is given below:

```
    InstComposite: Modelica.Electrical.Analog.Sources.StepVoltage V {V=0, startTime=1, \
                                                                     offset=0}
      InstPrimitive: SI.Voltage V {=0, start=1, final quantity="ElectricPotential", \
                                   final unit="V"}
      InstExtends: Interfaces.VoltageSource {V=0, startTime=1, offset=0,
           redeclare Modelica.Blocks.Sources.Step signalSource(height=V)}
        InstPrimitive: SI.Voltage offset {=0, =0, final quantity="ElectricPotential", \
                                          final unit="V"}
        InstPrimitive: SI.Time startTime {=1, =0, final quantity="Time", final unit="s"}
        InstReplacingComposite: Modelica.Blocks.Sources.Step signalSource {height=V, \
                                          final offset=offset, final startTime=startTime}
          InstPrimitive: Real height {=V, =1}
          InstExtends: Interfaces.SignalSource {height=V, final offset=offset, \
                                                final startTime=startTime}
            InstPrimitive: Real offset {=offset, =0}
            InstPrimitive: SIunits.Time startTime {=startTime, =0, final quantity="Time", \
                                                   final unit="s"}
            InstExtends: SO {height=V, final offset=offset, final startTime=startTime}
              InstPrimitive: RealOutput y {}
              InstExtends: BlockIcon {height=V, final offset=offset,
```

```
                                        final startTime=startTime}
```

Here we see how the modification "redeclare Modelica.Blocks.Sources.Step signalSource(height=V)" affects the instance AST. The node InstReplacingComposite represents the component instance, instantiated from the class Modelica.Blocks.Sources.Step, resulting from the redeclaration. As a consequence, this branch of the instance AST is significantly altered by the redeclare modification.

Now look at the modification environment for the component instance startTime. The environment contains two value modifications: '=1' and '=0'. As noted above, the first modification in the list corresponds to the outermost modification and have precedence over the following modifications. Take a minute to figure out the origin of the modifications by looking upwards in the instance AST.

1.6. Flattening of the filter model

Having computed the instance, we can now flatten the model:

```
# Don't flatten model if it already exists
try:
    filter_flat_model.name()
except:
    # Flatten the model instance filter_instance
    filter_flat_model = mc.flatten_model(filter_instance)
```

During flattening, the instance tree is traversed and all primitive declarations and equations are collected. In addition, such as scalarization and elimination of alias variables are performed.

Let us have a look at the flattened model:

```
print(filter_flat_model.prettyPrint(""))
```

We may also retrieve some model statistics:

```
print("*** Model statistics for CauerLowPassAnalog *** ")
print("Number of differentiated variables: %d" \
      % filter_flat_model.numDifferentiatedRealVariables())
print("Number of algebraic variables:      %d" \
      % filter_flat_model.numAlgebraicRealVariables())
print("Number of equations:                %d" \
      % filter_flat_model.numEquations())
print("Number of initial equations:        %d" \
      % filter_flat_model.numInitialEquations())
```

How many variables and equations is the model composed of? Does the model seem to be well posed?

At this point, take some time to explore the 'filter_flat_model' object by typing 'filter_flat_model.<tab>' in the Python shell to see what methods are available. You may also have a look in the Modelica compiler API.

Chapter 6. Optimica

The Optimica extension in described in detail in [Jak2008a].

Chapter 7. Limitations

This page lists the current limitations of the JModelica.org platform, as of version 1.2.0. The development of the platform can be followed at the Trac site, where future releases and associated features are planned. In order to get an idea of the current Modelica compliance of the compiler front-end, you may look at the associated test suite. All models with a test annotation can be flattened.

- The Modelica compliance of the front-end is limited; the following features are currently not supported:
  - If expressions are supported, but not:
    - When clauses
    - If equations
  - Parsing of full Modelica 3.2 (Modelica 3.0 is supported)
  - Integer and boolean variables (integer and boolean parameters and constants are supported)
  - Strings
  - Enumerations
  - Generics (redeclare constructs) is only partially supported
  - Limitations apply to the use of functions with arguments with unknown array sizes
  - External functions
  - The following built-in functions are not supported: sign(v), Integer(e), String(...), div(x,y), mod(x,y), rem(x,y), ceil(x), floor(x), integer(x), delay(...), cardinality(), semiLinear(...), Subtask.decouple(v), initial(), terminal(), smooth(p, expr), sample(start, interval), pre(y), edge(b), reinit(x, expr), scalar(A), vector(A), matrix(A), diagonal(v), product(...), outerProduct(v1, v2), symmetric(A), skew(x).
  - Overloaded operators (chapter 14)
  - Stream connectors (chapter 15)
  - Mapping of models to execution environments (chapter 16)
- In the Optimica front-end the following constructs are not supported:
  - Annotations for transcription information
  - Minimum time problems
- The JModelica.org Model Interface (JMI) has the following Limitations:
  - The ODE interface requires the Modelica model to be written on explicit ODE form in order to work.
  - Second order derivatives (Hessians) are not provided
  - The interface does not yet comply with FMI specification

Bibliography

[Jak2007] Johan Åkesson. *Tools and Languages for Optimization of Large-Scale Systems*. LUTFD2/TFRT--1081--SE. Lund University. Sweden. 2007.

[Jak2008b] Johan Åkesson, Görel Hedin, and Torbjörn Ekman. *Tools and Languages for Optimization of Large-Scale Systems*. 117-131. *Electronic Notes in Theoretical Computer Science*. 203:2. April 2008.

[Jak2008a] Johan Åkesson. *Optimica—An Extension of Modelica Supporting Dynamic Optimization*. *Proc. 6th International Modelica Conference 2008*. Modelica Association. March 2008.

[Jak2010] Johan Åkesson, Karl-Erik Årén, Magnus Gäfvert , , and . *Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problem*. *Computers and Chemical Engineering*. 203:2. 2010.

Index