Technische Universität Berlin

Fakultät für Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik

Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science

Dynamic Structure Modeling-Framework

User‘s Guide

Abstract

This document is the user's guide that illustrates how to install and run the DySMo Framework (Dynamic Structure Modeling-Framework) on Windows. The aim is to enable the user to create his own models and to simulate them using DySMo. The guide gives an introduction to DySMo and explains its functionalities by referring to examplary models.

Contents

List of Figures

List of Tables

Listings

1 Introduction

This section shows how to setup DySMo. If you have any questions or issues, please feel free to write an e-mail to *a.mehlhase@tu-berlin.de* or leave a comment at `https://bitbucket.org/amehlhase/dysmo`.

1.1 Getting started

First you install DySMo by executing the exe-file and following the instructions or by (extracting and) saving the install folder 'DySMo' to your desired destination path. The directory references are relative paths for the rest of the document. If there is a need to change some configurations, the pattern «yourDirectory» is used.

1.2 Installing Python

To install the Python interpreter on Windows you can use the following links. It is strongly recommended to use the given version of the Python interpreter and also the given versions of the needed packages, other versions are not tested and might not work. The Python interpreter and most of the packages can be downloaded both as 32-Bit version and as 64-bit version for Windows. If you have some issues using the 64-bit versions, please change to the 32-bit versions.

32-bit Windows

- python interpreter (version 3.4.2)
  Link: `https://www.python.org/ftp/python/3.4.2/python-3.4.2.msi`
  For further information about the python-interpreter, refer to the offcial Python project page `www.python.org`.

- pywin32 (version 219)
  Link `http://sourceforge.net/projects/pywin32/files/pywin32/Build%20219/pywin32-219.win32-py3.4.exe/download`

- numpy (version 1.9.1)
  Link: `http://sourceforge.net/projects/numpy/files/NumPy/1.9.1/numpy-1.9.1-win32-superpack-python3.4.exe/download`

- matplotlib (version 1.4.2)s
  Link: `https://downloads.sourceforge.net/project/matplotlib/matplotlib/matplotlib-1.4.2/windows/matplotlib-1.4.2.win32-py3.4.exe`

- pyparsing (version 2.0.3)
  Link `https://pypi.python.org/pypi/pyparsing/2.0.3`
  Please choose the win32 py3.4 version!

- dateutil (version 2.4.0)
  `https://pypi.python.org/pypi/python-dateutil/2.4.0`
  This package is only available as sources and needs to be compiled manually.

- six (version 1.9.0)
  `https://pypi.python.org/pypi/six/1.9.0`
  This package is only available as sources and needs to be compiled manually.

64-bit Windows

- python interpreter (version 3.4.2)
  Link: `https://www.python.org/ftp/python/3.4.2/python-3.4.2.amd64.msi`
  For further information about the python-interpreter, refer to the offcial Python project page `www.python.org`.

- pywin32 (version 219)
  Link `http://sourceforge.net/projects/pywin32/files/pywin32/Build%20219/pywin32-219.win-amd64-py3.4.exe/download`

- numpy (version 1.9.1)
  Unfortunately, you have to either find an unofficial 64-bit version or compile the sources by yourself.

- matplotlibversion 1.4.2
  Link: `http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-1.4.2/windows/matplotlib-1.4.2.win-amd64-py3.4.exe/download`

- pyparsing (version 2.0.3)
  Link `https://pypi.python.org/pypi/pyparsing/2.0.3`

- dateutil (version 2.4.0)
  `https://pypi.python.org/pypi/python-dateutil/2.4.0`
  This package is only available as sources and needs to be compiled manually.

- six (version 1.9.0) `https://pypi.python.org/pypi/six/1.9.0`
  This package is only available as sources and needs to be compiled manually.

First you install the Python interpreter by executing the exe-File. During the installation process you will probably have the possibility to set the *Python environment variable*, which is recommended. For the case that the possibilty is not given (or you want to set the environment variable after the installation process), the following section describes how to do that. For the case that you don't want to set the Python environment variable, you have to edit the run.bat file of DySMo. It will be explained later how to proceed.

The Python environment variable should be set to run the Python interpreter from any path. Therefore locate the file python.exe. If Python is installed in the default directory, you can find the file in *C : \ProgramFiles(x86) \Python34*. Copy this path and open the environment variables setup *Systemcontrol* → *System* → *advanced* → *environment variables* (or for a german version *Systemsteuerung* → *System* → *Erweiterte Systemeigenschaften* → *Umgebungsvariablen*). Find the variable path and add the copied path separated by a semi-colon. If everything works, you can now open the *command line* (*Eingabeaufforderung*) and start the Python interpreter. To test it, go to *Start* → *programs* → *accessories* → *execute* (or *Start* → *Programme* → *Zubehör* → *Ausführen*) and type in cmd. You can now type in Python, the interpreter should start and accept your commands.

To install the needed packages the easiest way is to execute the downloaded exe-files if the exe-files exist. If they do not, there are two ways to install the packages otherwise. The first option uses Python's *pip*. *pip* is a package management system used to install and manage software packages written in Python. It should be installed with the Python interpreter by default. (For further information about pip, refer to `https://pip.pypa.io/en/latest/reference/pip_install.html`). You have to type *python pip install SomePackage* into the command line for installing the latest version of the package and *python pip install SomePackage==X.X.X* for installing a specific version. The package will be automaticly downloaded and the links above are not needed in this case. For example use *python pip install python-dateutil==2.4.0* to install the package *dateutil*. In this case the package *six* should also have been installed.

If you do not want to use *pip*, you can compile the packages manually. In every downloaded package source folder exists a *setup.py* file. That file needs to be executed using the Python interpreter (change the directory in the command line to the source folder and type *python setup.py install* into the command line).

1.3 Installing Dymola

To install the modeling environment Dymola, please refer to the Dymola user's manual (*«Your- Directory»\Documentation\Dymola5Manual.pdf*). Make sure you have a fully licensed version, DySMo does not work with the demo version. For more information about Dymola visit `www.dymola.com`.

1.4 Installing OpenModelica

You can also use OpenModelica to simulate your models. DySMo works with OpenModelica Version 1.9.2. For further information about OpenModelica, refer to the offcial project page `www.openmodelica.org`.

1.5 Installing Matlab/Simulink

DySMo does also support Matlab/Simulink. To use Matlab/Simulink in the Framework Matlab/Simulink has to be installed on your computer. The Framework was tested with Matlab 2011a and was not fully tested with other versions, so the Framework might not work with other versions.

1.6 Editing DySMo's Config-Files

During the installation process of DySMo (in case you used the installer) you will have the possibility to set the paths shown in Listing 2, which is recommended. If you did not, some brief editing of DySMo's config-files must be done.

First you have to edit the Config.cfg file (Listing 1).

```
[Dymola]
AlistDir =
PathExe =

[OpenModelica]
PathExe =
```

Listing 1: Config.cfg

For Dymola, the path to the directory containing the *alist.exe* file and the path to the *dymola.exe* file need to be set. For Open Modelica, just the path to *omc.exe* file needs to be set. If Dymola and OpenModelica have been installed into their default directories, the Config.cfg should look like Listing 2.

```
[Dymola]
AlistDir = C:\Program Files (x86)\Dymola 2014\bin
PathExe = C:\Program Files (x86)\Dymola 2014\bin\Dymola.exe

[OpenModelica]
PathExe = C:\OpenModelica1.9.2\bin\omc.exe
```

Listing 2: Config.cfg edited

In general DySMo must be started like this from command line:

1. *«PathToPython»\python.exe*
2. *«yourDirectory» \src \DySMo.py*
3. *«PathToModel» \ModelConfig.py*

Consider that you might need to execute this with administrator rights in certain circumstances. For easy execution, a Windows-batch file called *run.bat is included* in DySMo that easisly starts DySMo through drag and drop.

If you have not set the Python environment variable as described above, you further have to edit the *run.bat* file (see Listing 3).

```
@echo off
cd \%~dp0\src
python DySMo.py \%1
pause
```

Listing 3: run.bat

The path to the *python.exe* has to be set. **It is important that the path does not contain any space characters.** Otherwise the run.bat will not work. If you have installed python into the default directory, the edited *run.bat* file should look like Listing 4.

```
@echo off
cd \%~dp0\src
C:\Python34\python.exe DySMo.py \%1
pause
```

Listing 4: run.bat edited

2 Getting started with DySMo

DySMo supports different modeling tools and an arbitrary number of mode switches. This chapter gives you a design overview and shows you how your models have to be prepared to use them as variable-structure models in the given framework.

2.1 DySMo's general design

The basic idea of DySMo consists of creating variable-structure models by using integrated standard simulation tools and thus using already implemented models, too. Further the framework should always disclose what happens at any point during simulation. This section gives a brief introduction into the structural design of DySMo.

A variable-structure model consists of different modes whereby every mode is a separately executable model. At any point during simulation one of the modes is active and determines the dynamic behavior of the whole variable-structure model (see Figure 1). The mode switches can occur during simulation using transitions. Therefore, a control is needed which manages the mode switches, and this control is provided by the framework. The framework at the current state only allows modes in the highest level of hierarchy, modes consisting of further modes cannot be implemented (directly).

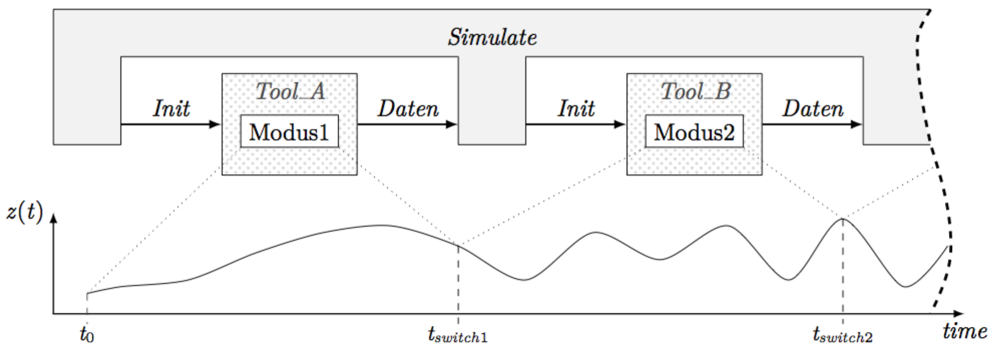


Figure 1: General simulation procedure

At the beginning the first mode must be identified and initialized. At every mode switch the simulation data of the just simulated mode will be read, saved and used to identify the next mode and to initialize that next mode.

The framework features an easily understandable object-oriented structure (see Figure 2). There are three main classes which are important (and can be found in *«yourDirectory»\src*).

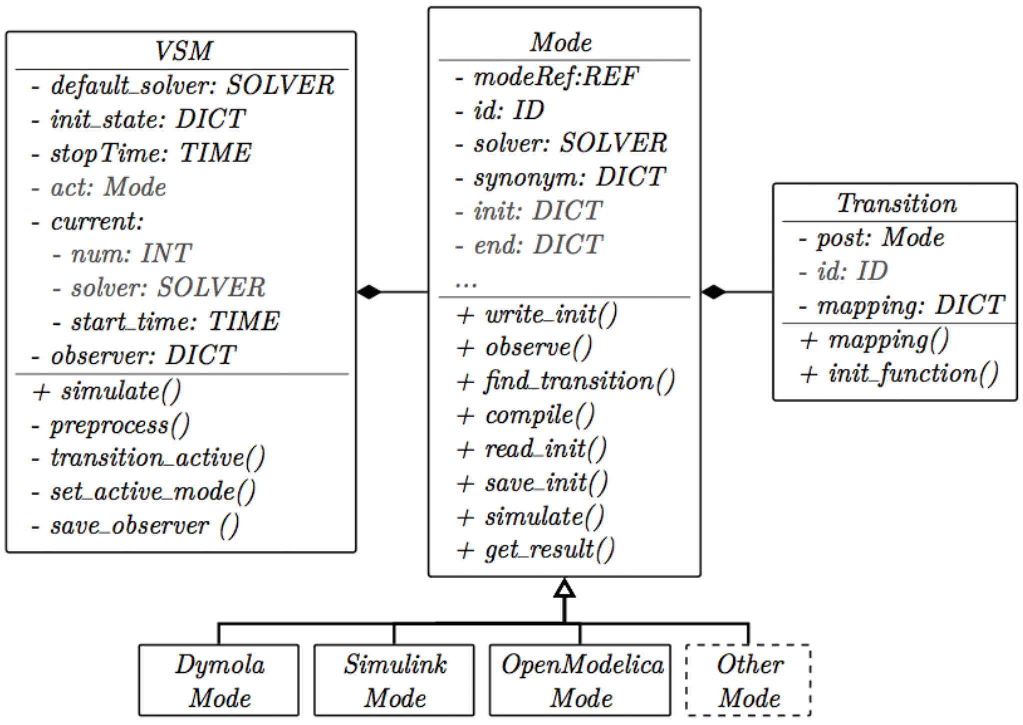


Figure 2: Object-Oriented Design of DySMo

First you have the *VSM* class (short for variable-structure model class). For every simulation of a variable-structure model exactly one instance of this class will be created. This instance provides some general attributes for the simulation (that can be set by the user) and the implementation of the basic functions which are needed to simulate the variable-structure model. In the *VSM* class an *observe* array containing the variables which should be observed and saved during the simulation, is provided and can be set.
Further the *VSM* class provides attributes for the start and stop time of the simulation, for a default solver, for the initial state of the simulation (or the initial values of the first mode's variables) and attributes for the current simulating mode such as the simulation number, the used solver and the start time.

A general mode class is given for the modes. The mode class is an abstract class ensuring that every mode fulfills basic requirements. Further for every integrated simulation tool a subclass is implemented which provides the tool-specific attributes and functions. Every mode is created as an instance of one of this subclasses. In case that a new tool should be integrated into DySMo the main task would be to implement such a subclass suitable to the tool-specific requirements (and also DySMo‘s requirements). The modes are directly assigned to the *VSM*.
Every mode provides a dictionary *synonym* which maps the observed variables' names

of the *VSM* onto the corresponding mode's variables. Furthermore the modes have attributes for the solver, the init and the end state (or the initial and end values of the variables), a mode id and a *modeRef* which references a mode uniquely. The simulation results of a mode's simulation are saved into a file, named due to the simulation number and the mode ID. The simulation number increases by one, everytime the current simulated mode switches.

The transitions are assigned to the modes out of which they lead. That means for a mode switch, that just the relevant subset of transitions has to be checked for the right one instead of the whole set. This is important for describing the model's structure.
The transition also provides a dictionary which maps the variables of the mode out of which it leads, onto the variables of the next mode. The initial state of the next mode can be modified by an *init_function* which can be specified in the transition, too.

For creating a variable-structure model you need the mode's model files on the one hand and a *Config*-File on the other hand. The *Config*-File defines the concrete structure of the variable-structure model. It describes loosely speaking the general properties, the number and properties of the modes, the needed transitions and the approach of visualizing the simulation results through plots. How this is done, is illustrated in the following section.

2.2 Creating a variable-structure model

In this section a case study is designed to explain how you create and simulate a variable-structure model with DySMo. You can find the implemented model in *«yourDirectory»\sample\pendel*.

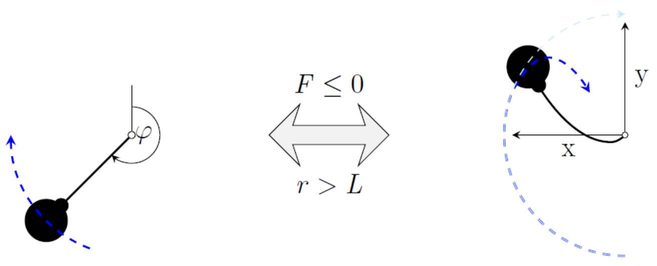
2.2.1 The cable pendulum

Consider a cable pendulum which becomes a falling mass, once the centrifugal force is no longer sufficient to keep the pendulum on its circular path (see Figure 3).

Once the mass falls into the cable again, the pendulum changes to its normal swing. As you can see in Figure 3, cartesian and polar coordinates are used in two different systems of equations to describe the normal swing and the falling mass. Thus, two modes are supposed to be implemented.

2.2.2 The Config-File

For every variable-structure model you need to create a Config-File which defines the parameters for the variable-structure model itself, the single modes, the transitions



$$\ddot{\varphi} = -\frac{g}{L} \cdot sin(\varphi) - D \cdot \frac{g}{L} \cdot \ddot{\varphi}$$

$$F = m \cdot g \cdot cos(\varphi) + m \cdot L \cdot \dot{\varphi}^2$$

$$m \cdot \dot{vx} = 0$$

$$m \cdot \dot{vy} = -g \cdot m$$

$$r = \sqrt{x^2 + y^2}$$

Figure 3: Physical representation of the cable pendulum

and maybe some initial functions for initializing new modes. The Config-File has to be written in Python and uses the classes that are already implemented in DySMo. The complete Config-File of the pendulum can be found in *«yourDirectory»\sample\pendel*.

At first you should set the general model parameters (see Figure 5).

```
1 model.default_solver = Solver("dassl");
2 model.translate = True;
3 model.init = {};
4 model.startTime = 0;
5 model.stopTime = 10;
6 model.observe = ['x', 'y'];
```

Listing 5: Pendulum Config.py - Model Parameters

Consider that the variable *model* is always available in a Config-File, which is the single instance of the VSM class. There are different general model parameters for the variable-structure model you can set. You can choose the default solver (line 1), start and stop time of the simulation (line 4 and 5). If the parameter *model.translate* is *True*, your model and modes will be compiled before simulation (see line 2). If the models of the single modes have already been compiled, you can set this parameter to *False* to save compilation time. It is recommended to leave this option *True* to avoid mistakes.. In line 9 an array is created for the variables you want to observe (and maybe plot later). At this point you just list the names of the observed variables under which they will be saved after simulation.

Next you should define the single modes of the variable-structure model (see Listing 6).

```
 1 #First mode
 2 mode1 = DymolaMode();
 3 mode1.solver.tolerance = 1e-4;
 4 mode1.modeRef = "pendulum.Pendulum_struc";
 5 mode1.files = ["pendulum.mo"];
 6 mode1.synonym = {'x' : 'x', 'y' : 'y'};
 7 
 8 #Second mode
 9 mode2 = OpenModelicaMode();
10 mode2.solver.tolerance = 1e-4;
11 mode2.modeRef = "pendulum.Ball_struc";
12 mode2.files = ["pendulum.mo"];
13 mode2.synonym = {'x' : 'x', 'y' : 'y'};
```

Listing 6: Pendulum Config.py - Modes

First a mode has to be declared with a constructor referring to the simulation tool it will use (see line 2 or 9). After that you can set parameters of the mode like the solver tolerance (see line 3) or the solver that should be used. If you do not specify a solver, the default solver that is set in the general model parameters, is used. After that you have to state which model (see line 4 or 11) of which package file (see line 5 or 12) should be used for the mode. In lines 5 and 12 you specify in which files to look for models, this is modelica specific, as models may need data from different files. Consider that these files must be located in the same directory as the Config-File. At last the variables of the model which should be observed, must be mapped to the variables you listed for the hole model (see line 6 and 13) in a *dictionary*. You form tupels of the observed variables and the variables that represent them in the mode, in that order using colons.

The next step is to declare the transitions from one mode to another. In this case you need a transition from the first mode to the second one and a transition the other way round (see Listing 7). Consider that you cannot refer to any modes in the transitions that have not been declared yet.

```
 1 #Transition from mode 1 to mode 2
 2 trans1_2 = Transition();
 3 trans1_2.post = mode2;
 4 trans1_2.mapping = {'x' : 'x', 'y' : 'y',
 5                   'vx': 'der(x)' , 'vy':'der(y)'};
 6 
 7 #Transition from mode 2 to mode 1
 8 def speed(actMode, oldMode):
 9     actMode.set_initialValue('dphi', 0.0);
10 
11 trans2_1 = Transition();
```

```
12 trans2_1.post = mode1;
13 trans2_1.mapping = {'x' : 'x', 'phi' : 'phi'};
14 trans2_1.init_function = speed;
```

Listing 7: Pendulum Config.py - Transitions

First, you have to declare a transition (see line 2 or 11). Next you have to set the mode the transition leads to (see line 3 or 12). After that the variables which must be initialized, have to be specified. Therefore, you can specify a mapping dictionary or define an own function. If you use the mapping dictionary, tuples of variables must be formed (see line 4 or 13). The first variable is the one of the new mode that should be initialized. The second one is the variable of the old mode which should be used to initialize the new one. The variables are separated by colons and the tuples by commas again. Further you can initialize variables by defining an own function (see line 8-9). After defining an own function you can set the transition's init-function as that function (see line 14).

Finally the transitions have to be added to the modes and the modes to the (variable-structure) model (see Listing 8).

```
1 #Set the transitions
2 mode1.transitions = [trans1_2];
3 mode2.transitions = [trans2_1];
4
5 #Set the modes
6 model.modes = [mode1, mode2];
```

Listing 8: Pendulum Config.py - Set Transitions and Modes

For every mode the transitions which lead out of it, have to be set in an array (see line 2 and 3). Consider that the transition's ID refers to its position in this array, starting with one. That is important for designing the models of the modes. The same applies to the modes and the variable-structure model (see line 6).

Now you have the possibility to create plots of the observed variables after simulation (see Listing 9).

```
#Create plots
plot = VariablePlot();
plot.vars = {'y' : Color.MAGENTA};
plot.xAxisVar = 'x';
plot.drawGrid = 1;
plot.labelXAxis = "x";
plot.labelYAxis = "y";
plot.fileName = 'pendulumplot.png';

#Set the plots
model.plots = [plot];
```

Listing 9: Pendulum Config.py - Set Plots

After simulation you get the plot of Figure 4.

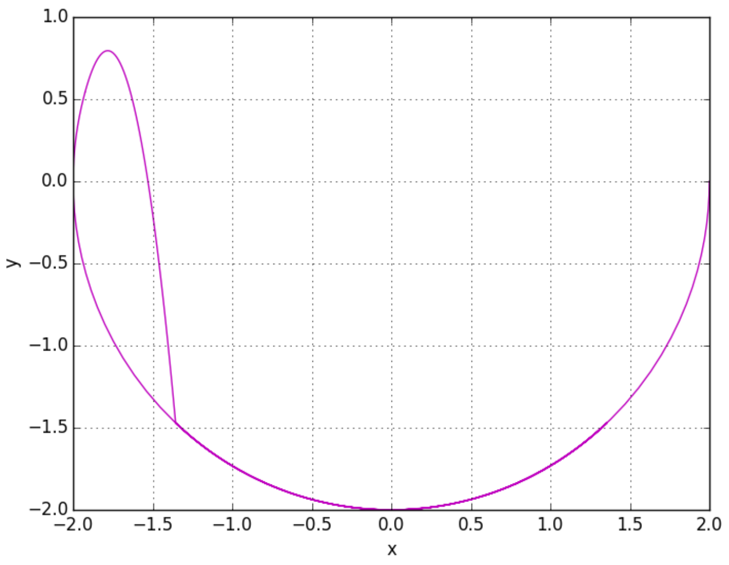


Figure 4: Pendulum - Plot

The plot shown in Listing 9 is of type *VariablePlot*. The framework provides two types of plots that can be used. The *VariablePlot* is a plot for creating a curve for every plotted (*observed*) variable. Thus, the variables are indicated in a dictionary as tuples of the variable name itself and the color of their curve (see line 3, Listing 9). The second type is the *ModePlot*. For the *ModePlot* the same attributes can be set as for the *VariablePlot*. The only difference is the *vars* attribute which is not a dictionary anymore, but a simple array of variables. The color of the curve will automatically change in accordance with the mode switches during simulation.

2.2.3 Transitions in the model file

For a variable-structure model the modes need termination conditions which are specified in the corresponding models (see «yourDirectory» \samples \pendel \pendulum.mo). For each mode the termination condition and the transition that leads to the next mode, has to be specified. Thus, every mode must have a variable named *transitionId* (exactly in this spelling, code characters are case sensitive). Before the simulation of the active mode terminates, an integer value has to be assigned to this variable referring to the corresponding transition. As already mentioned the transition will be taken from the position in the mode's transition array according to this *transitionID*.

For the cable pendulum example every mode has just one transition which leads out of it. Thus the *transitionId* is set to 1 before terminating. In Listing 10 the mode of the normal swing is shown (the second mode). It is implemented in Modelica.

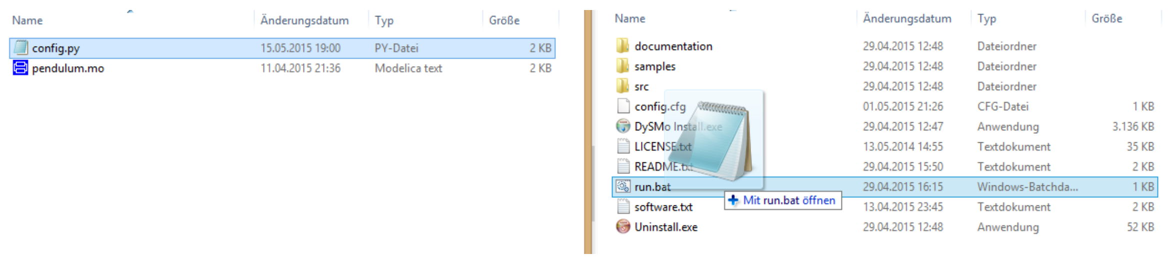
```
1 model Pendulum_struc
2   extends Pendulum_phi;
3   Integer transitionId(start = 0);
4 equation
5     when F <= 0 or terminal() then
6     transitionId = 1;
7     terminate("Pendulum to ball");
8     end when;
9 end Pendulum_struc;
```

Listing 10: Pendulum Model File - pendulum.mo

The *transitionId* is set to 0 by default (see line 3). Once the termination condition is met (see line 5), the *transitionId* is set to 1 before terminating this mode. According to Listing 8 the first transition will be chosen out of the array in line 3 which is *trans2_1*.

2.3 Simulation

The easiest way to start the simulation is to drag the Config.py file an drop it on the bat.run file (see Figure 5).

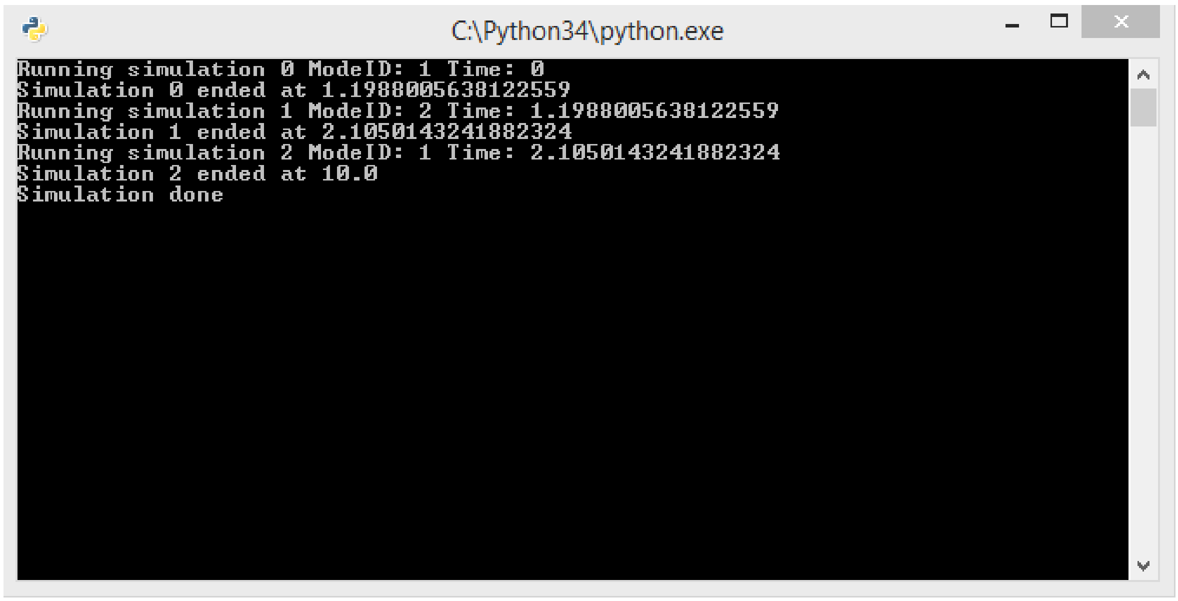


Name	Änderungsdatum	Typ	Größe
config.py	15.05.2015 19:00	PY-Datei	2 KB
pendulum.mo	11.04.2015 21:36	Modelica text	2 KB

Name	Änderungsdatum	Typ	Größe
documentation	29.04.2015 12:48	Dateiordner	
samples	29.04.2015 12:48	Dateiordner	
src	29.04.2015 12:48	Dateiordner	
config.cfg	01.05.2015 21:26	CFG-Datei	1 KB
DySMo Install.exe	29.04.2015 12:47	Anwendung	3.136 KB
LICENSE.txt	13.05.2014 14:55	Textdokument	35 KB
README.txt	29.04.2015 15:50	Textdokument	2 KB
run.bat	29.04.2015 16:15	Windows-Batchda...	1 KB
software.txt	13.04.2015 23:45	Textdokument	2 KB
Uninstall.exe	29.04.2015 12:48	Anwendung	52 KB

Figure 5: Starting the simulation

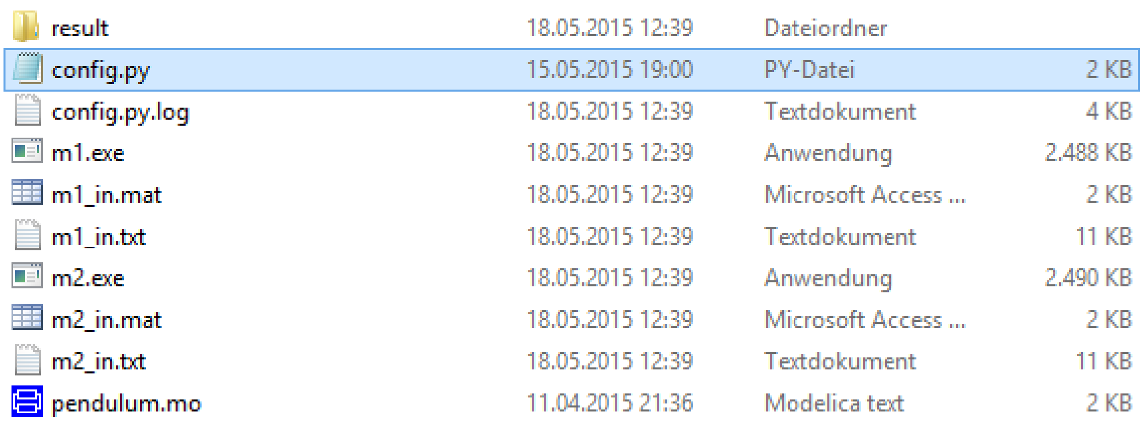
The simulation will start and simulate the single modes according to the transitions, always starting with the first mode (you have set in line 6, Listing 8). Sometimes the first time you start a simulation after booting your computer, causes an error. If this happens, just try again. It should work the second time. After simulation the Python console should look like Figure 6.



```
Running simulation 0 ModeID: 1 Time: 0
Simulation 0 ended at 1.1988005638122559
Running simulation 1 ModeID: 2 Time: 1.1988005638122559
Simulation 1 ended at 2.1050143241882324
Running simulation 2 ModeID: 1 Time: 2.1050143241882324
Simulation 2 ended at 10.0
Simulation done
```

Figure 6: The python console after simulation

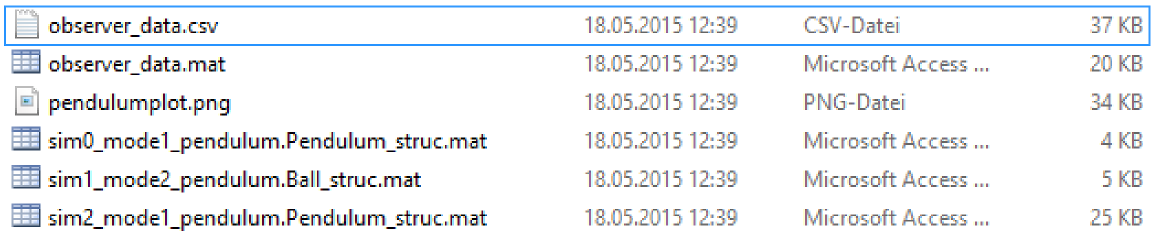
Now lets take a closer look to the folder that contains your model (in this case *«yourDirectory»\sample\pendel*). Now it contains some more files and an additional folder named *result* (see Figure 7).



result	18.05.2015 12:39	Dateiordner	
config.py	15.05.2015 19:00	PY-Datei	2 KB
config.py.log	18.05.2015 12:39	Textdokument	4 KB
m1.exe	18.05.2015 12:39	Anwendung	2.488 KB
m1_in.mat	18.05.2015 12:39	Microsoft Access ...	2 KB
m1_in.txt	18.05.2015 12:39	Textdokument	11 KB
m2.exe	18.05.2015 12:39	Anwendung	2.490 KB
m2_in.mat	18.05.2015 12:39	Microsoft Access ...	2 KB
m2_in.txt	18.05.2015 12:39	Textdokument	11 KB
pendulum.mo	11.04.2015 21:36	Modelica text	2 KB

Figure 7: Folder of the variable-structure model after simulation

In the *config.py.log* file you can find general information about the simulation such as the overall simulation time, the simulation and compilation time of each mode. The exe-files *m1.exe* and *m2.exe* are compiled versions of the modes. The files *m1_in.txt* and *m2_in.txt* contain further information for the modes' initializations such as initial values of the variables, solver informations and so on. The files *m1_in.mat* and *m2_in.mat* contain the initial values of the variables that are loaded into the simulation tool before simulating the according mode. The simulation results are saved in the folder *result* (see Figure 8).



observer_data.csv	18.05.2015 12:39	CSV-Datei	37 KB
observer_data.mat	18.05.2015 12:39	Microsoft Access ...	20 KB
pendulumplot.png	18.05.2015 12:39	PNG-Datei	34 KB
sim0_mode1_pendulum.Pendulum_struc.mat	18.05.2015 12:39	Microsoft Access ...	4 KB
sim1_mode2_pendulum.Ball_struc.mat	18.05.2015 12:39	Microsoft Access ...	5 KB
sim2_mode1_pendulum.Pendulum_struc.mat	18.05.2015 12:39	Microsoft Access ...	25 KB

Figure 8: Simulation results

In this folder you can find a mat-File containing the observer variables' values over the complete simulation and you can find mat-files referring to the single modes. Finally also the plots which are set in the Config.py file, can be found here.

3 Overview of DySMo's Functionalites

This chapter will give a brief overview of the most important attributes of DySMo's variable-structure model (*VSM*) class, of the *Mode* classes, the *Transition* class and the *Plot* classes.

3.1 Variable-Structure Model Class

For every simulation one instance of the *VSM* class will be created by the framework. It represents the behavior of the variable-structure model over the whole simulation time. Thus it does not have to be instantiated by the user in the *Config*-File. The attributes of the variable-structure model class (*VSM*) and the values they can assume, are shown in the following Table 1.

Parameter	Values	Description
translate	*Boolean:* True, False	indicates, wether the modes are compiled
default_solver	*Solver Object*	indicates the used (default) solver for the modes
startTime	*Double*	indicates the the start time of simulation (in s)
stopTime	*Double*	indicates the stop time of simulation (in s)
init	*Dictionary:* "«variable»" : «initial value»	indicates the initial values for the mode's variables
observe	*String Array*	indicates the variables that are observed
modes	*Array of Mode Objects*	contains the modes the variable-structured model will use
plots	*Array of Plot Objects*	contains the plots wich will be created after simulation

Table 1: Variable-Structure Model

3.2 Solver

The needed solver can be set in the *VSM* as an default solver for all modes (consider that the tools have to support this solver and recognize it by the indicated name) and in any *mode*. To initialize a solver (class *Solver.py*) you can use the constructor *Solver("«name»")*. The attributes of the solver class and the values they can assume, are shown in the following Table 2.

Parameter	Values	Description
tolerance	*Double*	indicates the used solver for the mode
name	*String*	indicates the concretely used (default) solver for the *VSM/Mode*
stepSize	*Double*	indicates the step size of the solver

Table 2: Solver

Names

The *name* of the solver you can set in the constructor or afterwards, depends on the simulation tool you use. For the tools Dymola, OpenModelica and Matlab/Simulink they can be set with:

- *Dymola:* deabm, lsode1, lsode2, lsodar, dopri5, dopri8, grk4t, dassl, odassl, mexx, euler, rkfix2, rkfix3, rkfix4
- *OpenModelica:* dassl, euler, rungekutta, inline-euler, inline-rungekutta, dasslwort, dasslSymJac **TODO: Solver suchen, die stimmen hier so vielleicht nicht ...**
- *Matlab/Simulink:*

3.3 Modes

Mode Initialization

You can initialize a mode by using their constructors. Possible constructors are *DymolaMode()* for the Dymola tool and *OpenModelicaMode()* for the OpenModelica tool.

Setting Modes

Every mode must be set in the model with *model.modes = [«mode1», «mode2», ...]*. The position in the array indicates the ID of the mode (This is only important for the simulation's data storage).

Attributes

The attributes of the general mode class (*Mode.py*) and the values they can assume, are shown in the following Table 3.

Parameter	Values	Description
modeRef	*String:* «model identifier»	indicates the identifier of the mode's model
solver	*Solver Object*	indicates the used solver for the mode
synonym	*Dictionary:* {"«observer variable»" : "«mode variable»"}	maps the variables of a mode onto the variables which should be observed

Table 3: Mode - General

Specific Attributes

The specific mode classes such as *DymolaMode.py* for the Dymola mode and *OpenModelicaMode.py* for the OpenModelica mode derive from the general mode class. Thus the attributes shown in Table 3 are provided for every type of mode. Further every specific mode class provides specific attributes listed in the following Tables 4, 5 and 6.

Parameter	Values	Description
files	*String array*	indicates the paths to one or more model files/packages which contain your mode's models (and submodels)
numberOfIntervals	*Integer*	indicates the number of output intervals
intervalLength	*Double*	indicates the length of output intervals
params	*Dictionary:* {"«parameter»" : *Double* }	indicates the start values of the model's parameters (if not set, the default start values in the model are used)

Table 4: Mode - Dymola

3.4 Transitions

Initialization

To initialize a transition you use the constructor *Transition()*.

Setting Transitions

Every transition must be set in the mode where it leads out, with *«mode».transitions = [«transition1», «transition2», ...]*. The position in the array indicates the ID of the transition (which is important for the *transitionId* you set in the mode's model).

Parameter	Values	Description
files	*String array*	indicates the paths to one or more model files/packages which contain your mode's models (and submodels)
params	*Dictionary:* {"«parameter»" : *Double* }	indicates the start values of the model's parameters (if not set, the default start values in the model are used)

Table 5: Mode - OpenModelica

Parameter	Values	Description

Table 6: Mode - Matlab/Simulink

Attributes

The attributes of the transition class (*Transition.py*) and the values they can assume, are shown in the following Table 7.

Parameter	Values	Description
post	*Mode Object*	indicates the mode the transition leads to
mapping	*Dictionary:* {"«new variable»":"«old variable»"}	maps the values of the old mode's variables to the new mode's variables, if you want to map all old mode's variables onto the new mode's variables with the same name, you can use ** : **
init_function	*function* («new mode», «old mode»)	indicates a function you can implement to set the initial values of the new mode's variables

Table 7: Transition

3.5 Plots

Initialization

To initialize a plot you can use two different constructors. *VariablePlot()* creates a plot for your observed variables (see *observe* in Table 1) which describes the value-over-(simulation)time. *ModePlot()* creates also a plot for the observed variables, but additionally colors the parts of the plotted curve dependent on the mode to which they belong.

Setting plots

Every plot must be set in the model with *model.plots = [«plot1», «plot2», ...]*.

Attributes

The attributes of the two types of plots and the values they can assume, are shown in the following table 8.

Parameter	Values	Description
vars	*Dictionary* or *String Array*	indicates the variables which should be plotted, the variables have to be defined in the *observe* attribute of the *VSM*
drawGrid	*Integer:* 0, 1	indicates wether a grid should be plotted (1) or not (0)
labelXAxis	*String*	indicates the label of the x-axis
labelYAxis	*String*	indicates the label of the y-axis
xAxisVar	*String*	indicates wether the variables should be plotted over another variable than the time and over which one (the default variable for the x-axis is always the *time*)
show	*Boolean*	indicates wether a plot window opens after simulation
fileName	*String*	indicates whether and under which name a plot is saved in the result folder

Table 8: Plot

The both plot classes (*ModePlot* and *VariablePlot*) derive from the class *Plot* and provide the attributes shown in Table 8. But they differ on the attribute *vars*.

The *Variable Plot* is for creating a plot with one curve for every plotted variable. Thus the attribute *vars* is a dictionary with *{"«variable name»":Color}* which indicates the plotted variables and the colors of their curves. Possible colors are: BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, YELLOW.

For *Mode Plots* the *vars* attribute is a simple *array of Strings*, containig the (*observed*) variables wich should be plotted. The curve for the variables will change its color automatically in accordance with the mode switches.

4 Integrating a new tool

This section shall briefly explain how to integrate a new simulation tool into *DySMo*. DySMo is designed in a way which should easily enable the user to integrate new tools. Thus there should not be any needs to change or extend DySMo's existing classes. The communication between DySMo and a simulation tool is realized by exactly one special mode class (such as *DymolaMode* for Dymola). This special mode class has to be designed and implemented for the tool you want to integrate. The following short step-by-step instructions shall give a little assistance for achieving this goal (this section should not give an introduction into programming with *Python*, thus general understanding of *Python* is assumed):

1. Create a new file in the folder *«yourDirectory»\src\modes* and define your special mode class. The mode class has to derive from the general mode class (see *Mode.py* which you can find in *«yourDirectory»\src*), to fit DySMo's requirements.

2. Import the new mode class into the file *__modes__.py* which is located in the folder *«yourDirectory»\src*, with: *from modes.«package name» import «new mode class»*.

3. Then the four essential methods of every mode class have to be implemented in your new mode. The methods are *compile()*, *get_result()*, *save_init()* and *simulate()*. By further questions first read the comments in the (abstract) general mode class you find in *Mode.py*.

   - *compile()*: This method will be called by the framework and compile the mode. Hints: Every mode belongs to a model which can be accessed by the method *get_model()*. The model again has a method *get_path()* to get the path to the folder which contains the mode's model file(s). This *compile* method will only be called once when the framework calls the mode for the first time.

   - *get_result(simNum)*: This method reads the simulation results which are created by this mode for the simulation number *simNum*. Every time (and not only the first time) a mode is activated by the framework for simulation, the *simNum* increases by one. The result has to be an object of type *SimulationResult*. A *SimulationResult* (see *SimulationResult.py* in *«yourDirectory»\src*) expects a dictionary which maps variables (or variable names) onto its values. The dictionary should look like:

```
{  'x'     :  [t1(x), t2(x), ...],
   'y'     :  [t1(y), t2(y), ...],
              ...
   'time'  :  [t1(time), t2(time), ...]}
```

Consider that one line of the dictionary indicates the time vector for the simulation. It must exist and be named *time* (be aware of case sensitivity).

- *save_init()*: This method saves the initial simulation configuration into a file wich can be used for initialization and simulation later.
- *simulate()*: This method simulates the mode and saves the simulation results into the *result* folder. To save the simulation results in an useful manner, it is recommended to use the method *_getResultFileName()* of the *Mode* class and then to add a file extension.

Further Hint: It can be necessary to configure the new mode. Therefore the *config.cfg* file may be used which can be found in *«yourDirectory»*. For example the paths to the .exe-files of your simulation tool can be saved there. The values which are set in the *config.cfg* can be read by the *Config* class (can be found in *«yourDirectory»\src*) or by its global instance *g_cfg* in the package *globals* (can also be found in *«yourDirectory»\src*).

As already mentioned in the beginning, please take a look at the comments in the source code. A lot of further or specific questions will be answered there!

References