Sustainability in Energy and Buildings: Research Advances:
Vol 3: Sustainability in Energy and Buildings (SEB-14) Short Papers pp.20-30 : seb14s-005

A Python-Modelica Interface for Co-Simulation

Jesús Febres, Raymond Sterling, Marcus Keane

Informatics Research Unit for Sustainable Engineering, Department of Civil Engineering and Ryan Institute, National University of Ireland, Galway, Ireland
j.febrespascual1@nuigalway.ie

Abstract

This paper presents a novel tool for data exchange between Modelica and a growing open source programming language, Python, especially for co-simulation purposes. This tool was developed using Python and targeted to the Dymola IDE. However, general concepts can be translated to any other language. A case study of a complex Modelica model is presented to illustrate how the tool can be used for solving an inverse problem (i.e. model calibration) that could not be possible using only Modelica.

Keywords: Python, Co-simulation, Interface, Inverse Problem

1 Introduction

Often, in modelling and simulation applications, manipulation of information is needed for data pre/post processing. For our modelling and simulation tool of choice, Modelica [1], it does not incorporate a suitable data pre-processing tool and the Modelica development environments target simple data visualisation rather than complex post-processing operations. This problem becomes much more complex if this information has to be manipulated in real-time (when the simulation is running), i.e. in co-simulation applications.

A solution for this problem is developing an interface between Modelica and any good data processing tool/programming language in existence. In this work Python was chosen as data processing environment which is widely accepted by the engineering community as one of the best scripting and programming languages and its use is widespread. The interface was implemented using the same language and C.

This paper presents a novel tool for data exchange to/from Modelica, especially designed for simulation-time (for this work we considered real-time equivalent to simulation-time) purposes. Our tool, PyMo, is developed as a library using a standard distribution of Python 2.7 [2] and targeted to the Dymola IDE [3]. However, general concepts can be translated to any other language. In order to ensure reusability, no additional modules/libraries were used but just the standard ones present in most python installations. A case study of a complex Modelica model is presented to illustrate the tool's usage.

ISSN 2054-3743 : http://nimbusvault.net/publications/koala/sebra

2 Python-Dymola interface

The idea behind the developed interface is to provide the modeller with the ability to manipulate and exchange data with Modelica models mainly during the simulation is running but also allowing data manipulation before and after simulations. The interface is composed by two main parts, namely python side and Modelica side, which are explained in the following sections.

3 Python side

The main elements of this library were developed based on necessities that aroused during our daily work with Modelica using Dymola as a development environment. Firstly, data obtained from Modelica simulations had to be processed for a deeper analysis. Secondly, tables in initialization file (typically 'dsin.txt'), used by Dymola to run Modelica simulations needed to be parsed for improving their usability in Python. Finally, values had to be sent (received) to (from) Modelica; in some cases in order to exchange data between Modelica and a fault detention engine in real-time; and in other cases in order to reduce the computational cost when inverse problems have to be solved by using iterative methods. An overview of the whole package is presented below.

3.1 Getting results from Modelica

Every time a simulation is run, Modelica generates a results file. In the particular case of Dymola, in its standard configuration, it produces a .mat file. For processing these .mat files, we developed a package based on an open source project named DyMat [4]. However, we provided modifications trying to get more simplicity and efficiency to satisfy our necessities.

The module developed, matFile.py, contains the class matFile that needs the .mat file path to be instantiated. The main methods are described in Table 1

3.2 Parsing dsin.txt file

With this package we can read and modify any parameter value in the dsin.txt file generated after Modelica translation (compilation) in Dymola.

The class for parsing is named dsinFile and needs the path to the dsin.txt to be instantiated. Its main attribute and method are summarized in Table 2.

3.3 Dymola object

dymolaModel is the main class of the tool. It needs three parameters to be instantiated. The 'moFile' parameter is the path of the Modelica file (.mo format) containing the model. The 'modelName' parameter is the name of the model to be simulated. And the 'dirResults' parameter is the path of the working directory on which the results will be stored.

The Dymola Object allows for the performing Modelica simulation in two different modes namely the non-real-time mode and the real-time mode.

The **non-real-time mode** is used when no communication between Python and Modelica is needed during simulation. For this mode, there are two important attributes: 'parameters' and 'results', and one main method, simulate. See Table 3.

In cases when we need to communicate with Modelica during simulations, the **real-time mode** must be used. In this mode the instantiated object will have four additional methods (described in Table 3) and the simulation method will change its behaviour.

The non-real-time mode can be used for any Modelica model without any additional Modelica package. However, when the real-time mode is required, an additional package is needed in order to establish and synchronise the communication between Python and Dymola/Modelica.

4 Modelica side

In addition to the Python library we developed a Modelica package in charge of establishing and synchronising communications with python when a simulation is running in the real-time mode. This package contains the main element PyCom and a few more functions that PyCom needs to work. The PyCom element can work as sender of information to python, receiver of information from python or controller to synchronise the data exchange. Its operation mode will depend on the value of the parameter named 'comType'. The behaviours of PyCom can be described, briefly, as follows:

- **sending/receiving function**: it has an output/input connector that can be used to send/receive values to/from Python;
- **controlling function**: when this function is selected there would be no connector, however, one block of this type has to be placed in the model to be guarantee synchronisation during data exchange.

5 Non-real-time mode example

In order to illustrate how the tool works in this mode, the heating coil model in Figure 1 taken from [5] is used. In this case, model inputs are taken from a data file (data). HX is a water-air heat exchanger without condensation and it needs six input variables. Three of them to define state of the incoming air: temperature (CCo_Temp), relative humidity (CCo_RH) and mass flow rate (supplyMflow). Another two for the state of the incoming water: temperature (HC_wTemp) and mass flow rate (HC_wMflow). And the valve opening signal (HC_openingSignal) as control signal. We assume that HC_wMflow is parameter (mflow0_w) since incoming water mass flow rate is controlled with HC_openingSignal.

In non-real-time mode, we can modify parameter values, simulate the Modelica model and collect results to be used inside Python (see code in Appendix II). However no data exchange during simulation is allowed.

Figure 2 and Figure 3 show the plots generated with the Python code described in Appendix II. To generate Figure 2, only experiment parameters

were modified. In Figure 3 both experiment and model parameters were changed. The same steps were followed in the python code for both cases.

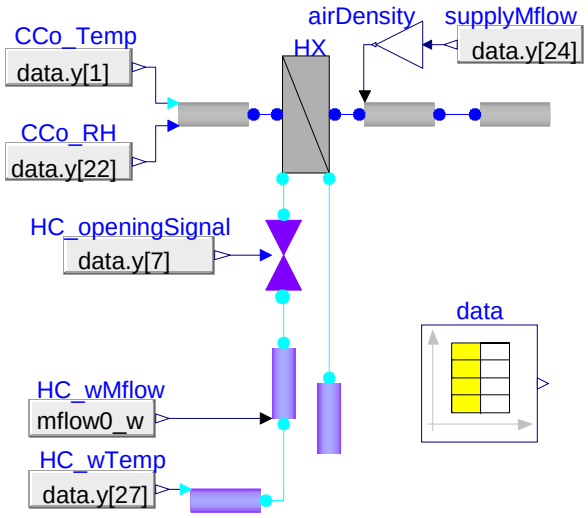

Figure 1. Non-real-time mode. Modelica model

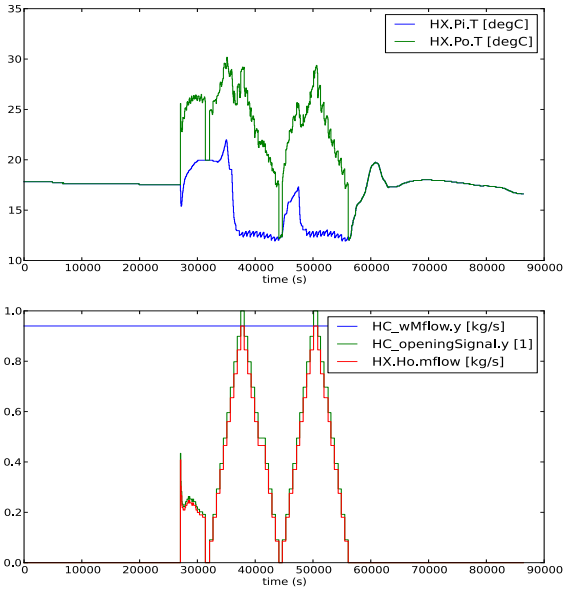

Figure 2. Non-real-time. Results plot with only 'experiment' modified

Initially, the Dymola model object was created then a simulation is run using the desired parameters. Once the simulation is finished, the corresponding plotting method is called and two plots are presented in the resulting plotting window. Input ('HX.Pi.T') and output ('HX.Po.T') values of air temperature

are shown on top and the three variables directly related with ‘mlow0_w’ in the bottom plot.

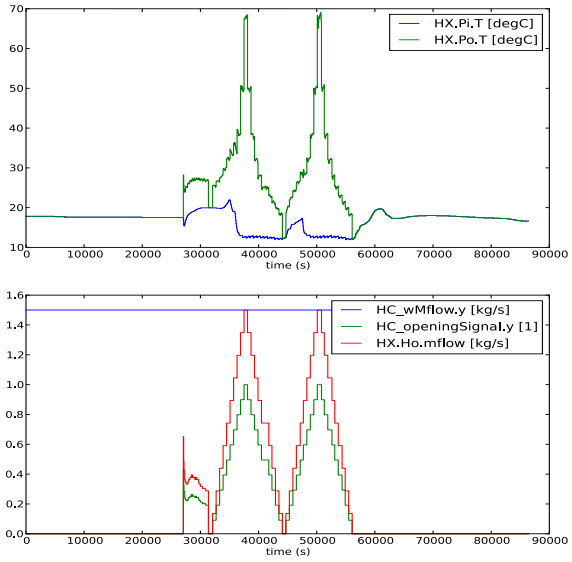

Figure 3. Non-real-time mode. Results plot with 'initialValue' modified

Plots show how parameter values were passed successfully and how they modified the outputs of the model simulation. One day was simulated in both cases and ‘mflow0_w’ value changes from first simulation (Figure 2) to the second one (Figure 3).

6 Real-time mode example

For this mode (Figure 4), the same model was considered but model inputs and outputs are read from Python directly.

In order to link Python with the Modelica model, PyCom blocks should be placed in the model. Six receiving blocks are needed. They represent the model inputs: incoming air temperature (Ti_a), incoming air relative humidity (RHi), air mass flow rate (mflow_a), opening valve signal (opening), incoming water temperature (Ti_w) and water mass flow rate (mflow_w). One sending block (To_a) is used to send outgoing air temperature values (HX.Po.T) to Python. In addition, as mentioned above, a control block (Control) is placed to guarantee the communication synchronization. All these blocks can be seen in Figure 4.

Outgoing air temperature and all inputs are known except opening valve signal.

To illustrate the usefulness of our tool, a real example (solving the inverse problem) with real data is presented. The idea is to adjust (and readjust) the unknown input ‘opening’ by using the feedback error between the current value of the study variable ‘HX.Po_T’ and its desired value ‘ref[t]’ taken from the data in order to decrease the error below the tolerance ‘errorMax’

(Appendix III). Although it can be done considering a step-by-step approach by running a Modelica simulation for every single iteration point, the computational cost is reduced substantially if the whole iteration process is done with just one Modelica simulation. Only data samples where air mass flow rate is higher than 1.0 kg/s are considered because they represent when the heat exchanger is working. Figure 5 shows how simulated values (HX.Po.T) match with desired values (ref.x) for those samples where air mass flow (mflow_a.x) rate is higher than 1.0 kg/s.

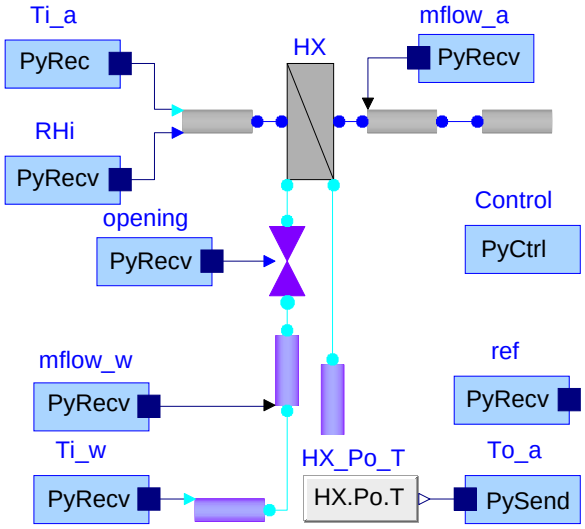

Figure 4. Real-time mode. Modelica model

7 Discussion

One important use of the real-time mode comes from the fact that sometimes Modelica not necessarily can solve inverse problems where model outputs are known (i.e. measurements or observations on the plant) and inputs are required (i.e. actuator signals), this is particularly true if there exist a high nonlinearity in the system, for example when limiters, if-then-else statements, or comparators are included in the model. Nonetheless, this problem can be seen as a classic control problem. In this case, the error between known outputs and actual outputs can be used as feedback to modify the inputs until this error decreases below some fixed tolerance. In our approach, instead of starting and stopping simulations for each iteration step, we can just run just one simulation for the whole process by using the tool and the values are adjusted iteratively, in simulation time during this one simulation. This process was shown in the examples presented.

Another advantage of the tool is that it allows for use of the executable generated by Dymola. That means that if there are several modellers developing several different components, all of them can be used together without disclose the source code. Finally, the tool can be used in cases where

an interaction between Modelica and any other M&S application is need, provided that it can communicate with Python.

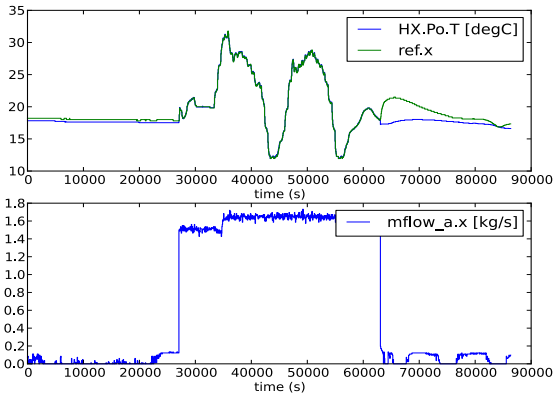

Figure 5. Real-time mode. Calculated and desired values (above). Air mass flow rate (below)

8 Future work

Next step in this research is to generate a useful documentation of the whole tool and improve the error handling. Once a stable version is developed, the idea will be to translate it to an open source Modelica environment, as such OpenModelica. At the end, Functional Mock-up Interface concepts will be integrated in order to avoid the Modelica model modification.

9 References

[1] M. Association, "Modelica ® - A Unified Object-Oriented Language for Systems Modeling Language Specification," 2012.

[2] M. Pilgrim, *Dive Into Python*. 2004.

[3] Dassault Systèmes AB, *Dymola - User Manual*, vol. 1, no. September. Lund - Sweden, 2012.

[4] J. Rädler, "DyMat," 2013. .

[5] J. Febres, R. Sterling, I. Torrens, and M. M. Keane, "Heat Ventilation and Air Conditioning Modelling for Model-Based Fault Detection and Diagnosis," in *Building Simulation 2013*, 2013, pp. 3513 – 3520.

10 Appendix I

Table 1. Main methods of matFile

Method	Inputs	Output
names()	Not required	Sorted list with the simulation variables names
description(name1, name2,…)	At least one variable name	Dictionary containing the description and the unit assigned inside Modelica for each input variable
get(name)	A variable name	Dictionary containing a time series with the resulting values for the input variable
plot([name11, name12, …], [name21 ,name22, …], …)	At least one names list is required	Plot window which contains a plot for each list. The x-axis corresponds to the simulation time in second

Table 2 Main attribute and method of dsinFile

Attribute/Method	Inputs	Output
parameters	(Attribute)	Dictionary with the four tables (‘experiment’, ‘method’, ‘settings’ and ‘initialValues’) from dsin.txt
modifyParameters({name1: value1, name2: value2, …})	Dictionary with the names of the parameters that will be modified and their new values	Modified dsin.txt with the new values

Table 3. Main attributes and methods of dymolaModel

Attribute/Method	Inputs	Output
parameters	(Attribute)	Dictionary containing the four tables from the dsin.txt file.
results	(Attribute)	It is a matFile object and is created when the simulation finishes
simulate({name1: value1, name2: value2, …})	Dictionary with the simulation parameters	Non-real-time mode: runs a Modelica simulation with passed parameters. Real-time mode: starts the simulation which will wait for a run() or finish() method call
run(step)	Real value. Default = 1.0	Move ‘step’ seconds in the simulation
setInputs({name1: value1, name2: value2, …})	Dictionary containing the variable names and the values to be sent	Sends values from Python to Modelica
getOutputs()	Not required	Dictionary with variable names and their values
finish()	Not required	Finishes an active simulation

11 Appendix II

11.1 Python code for non-real-time mode example

```
1  # -*- coding: utf-8 -*-
2  from dymola import *
3
4  packageName = 'C:/ModelicaConference.mo'
5  modelName = 'ModelicaConference.offLineMode'
6  dirWork = 'C:/Work'
7  model = dymolaModel(moFile=packageName,
8                      modelName=modelName,
9                      dirResults=dirWork)
10
11 parameters = {'StartTime':0.0,
12              'StopTime':60.0*60.0*24.0}
13 model.simulate(parameters)
14 model.results.plot(['HX.Pi.T','HX.Po.T'],
15                    ['HC_wMflow.y',
16                     'HC_openingSignal.y',
17                     'HX.Ho.mflow'])
18
19 HX_mflow0_w = model.parameters['initialValue']['HX.mflow0_w']
20 mflow0_w = 2.0*HX_mflow0_w
21 parameters = {'StartTime':0.0,
22              'StopTime':60.0*60.0*24.0,
23              'mflow0_w':mflow0_w}
24 model.simulate(parameters)
25 model.results.plot(['HX.Pi.T','HX.Po.T'],
26                    ['HC_wMflow.y',
27                     'HC_openingSignal.y',
28                     'HX.Ho.mflow'])
29
30 del model
```

Code 1. Non-real-time mode. Python code

The python code starts calling our library and defining dymolaModel class parameters in order to instantiate it (line 2 to line 9).

From line 11 to 17 a simulation is run just changing ‘experiment’ parameters, such as ‘StartTime’ and ‘StopTime’. In lines 11 and 12 new parameter values are passed. ‘StopTime’ is set in 86400 seconds, that means one day of simulation. Next, a simulation with those values is executed. Once the simulation is finished, the ‘results’ attribute/object is created and used to generate a plot window with two plots, one for each list passed as argument (see Figure 2).

In the code described above, just experiment parameters have been changed; however, modifying model parameters is the point of this tool. Now, we are going to modify ‘mflow0_w’ which was fixed in 1.5 kg/s, its new value shall be two times the nominal mass flow rate of water in the heat exchanger (‘HX.mflow0_w’). ‘HX.mflow0_w’ is read (line 19) and used to modify

'mflow0_w' parameter in lines 20-23. A new simulation is done using values defined above. Same variables are plotted (see Figure 3).

Last line is used to delete the 'model' object thus removing any temporary files used in simulations.

12 Appendix III

12.1 Python code for real-time mode example

```
from dymola import *

data = dataFile('C:/AHU09_Data.csv')
times = data.times()
labels = data.labels()
Ti_a = data[labels[1]]
RHi = data[labels[22]]
ref = data[labels[8]]
mflow_a = data[labels[24]]
Ti_w = data[labels[27]]
```

Code 2. Loading data

```
packageName = 'C:/ModelicaConference.mo'
modelName = 'ModelicaConference.inLineMode'
dirWork = 'C:/Work'
model = dymolaModel(moFile=packageName,
                    modelName=modelName,
                    dirResults=dirWork)
```

Code 3. Model Instantiation

After importing our library the data is loaded (see Code 2). This is the same data used in previous example (data block), however, in this case it is loaded directly to python using 'dataFile' class which works with Modelica tables or .csv files. As in the non-real-time mode the model is (Code 3).

```
n_mflow_w = 2.0
d_a = 1.22
parameters = {'HX.c_a':1006.0,
              'HX.c_w':4186.0,
              'HX.Ti0_a':6.3,
              'HX.To0_a':18.8,
              'HX.mflow0_a':d_a*(1.40+1.30/2.0),
              'HX.Ti0_w':82.0,
              'HX.To0_w':71.0,
              'HX.mflow0_w':0.47,
              'HX.r':0.9}
iMax = 50
errorMax = 0.1
alpha = 1e-2
starTime = 0.0
stopTime = iMax*(len(times)-1)
parameters.update({'StartTime':starTime,
                   'StopTime':stopTime})
model.simulate(parameters)
```

Code 4. Starting the model

```
HX_openingPP = {}
mflow_w                                    =
n_mflow_w*parameters['HX.mflow0_w']
for t in times:
  if mflow_a[t]>1.0:
    model.setInputs({'Ti_a':Ti_a[t],
                     'RHi':RHi[t]/100.0,
                     'mflow_a':d_a*mflow_a[t],
                     'Ti_w':Ti_w[t],
                     'mflow_w':mflow_w})
    openingAux = 0.5
    for i in range(iMax):
      model.setInputs({'opening':openingAux})
      model.run()
      To_a = model.getOutputs()['To_a']
      error = ref[t] - To_a
      if abs(error) < errorMax: break
      else: openingAux += error*alpha
    HX_openingPP[t] = 100.0*openingAux
  else: HX_openingPP[t] = 0.0
model.finish()
```

Code 5. Solving inverse problem

Now, the model parameters are defined and then the 'experiment' ones. After that, the model is started with the 'simulate()' method (see Code 4). The model will not run a whole simulation like non-real-time example but will wait for a 'run()' method call.

The idea is to adjust (and readjust) unknown input 'opening' by using the feedback error between current value 'To_a' and desired values 'ref[t]' in order to decrease the error below the tolerance 'errorMax' (Code 5). Although it can be done by using a step-by-step Modelica simulation for every single iteration point, the computational cost is reduced substantially if the whole iteration process is done with just one Modelica simulation. The first *for* loop swaps all data points, however, just points where air mass flow

rate is higher than 1.0 kg/s are considered for calculating 'opening' values, that corresponds when the heat exchanger is working. Initially, all inputs are set for current time 't' by using 'setInputs()' method. For every iteration in the second *for* loop, a new 'opening' value is used to run the model with actual inputs values, then 'To_a' is extracted in order to compare it with the desired value 'ref[t]'. Next, the 'opening' value is calculated using the resulting error. Best values of 'opening' are stored in 'HX_openingPP'. At the end the model is stopped with the 'finish()' method.

```
step = 60
parameters['StopTime'] = (len(times)-1)*step
model.simulate(parameters)
for t in times:
  model.setInputs({'Ti_a':Ti_a[t],
                   'RHi':RHi[t]/100.0,
                   'mflow_a':d_a*mflow_a[t],
                   Ti_w':Ti_w[t],
                   'mflow_w':mflow_w,
                   'ref':ref[t],
                   'opening':HX_openingPP[t]/100.0})
  model.run(step)
model.finish()
model.results.plot(['HX.Po.T','ref.x'], [mflow_a.x'])
del model
```

Code 6. Simulation with resulting values

In addition, a new simulation is run using 'opening' vales saved in 'HX_openingPP' (Code 6). In this part of the code, 'run()' method is called using a 60 seconds step which is the sample time in the data file.