

# OpenModelica-Python Interoperability Applied to Monte Carlo Simulation

Mohsen Torabzadeh-Tari, Peter Fritzson, Martin Sjölund, Adrian Pop  
PELAB – Programming Environment Lab, Dept. Computer Science  
Linköping University, SE-581 83 Linköping, Sweden  
{mohto,petfr, marsj,adrpo}@ida.liu.se

## Abstract

During recent years the interest in computer aided modeling and simulation of complex multi-domain systems have increased significantly. The evolution of the declarative equation-based object-oriented (EEO) modeling language Modelica is a prime example of such a trend. The inherent acausal modeling possibilities and the hybrid modeling capabilities are two features that make Modelica superior. The Modelica community (Modelica 2009 [1], Fritzson 2004 [2]) has achieved considerable success through the development of libraries for a number of different technical domains.

OpenModelica is an open source platform for modeling and simulation of complex physical systems based on Modelica (OpenModelica 2009 [3]). It is intended for both commercial and academic use. The development is supported by the non-profit organization Open Source Modelica Consortium. The most important parts of the platform are: OpenModelica Compiler (OMC), the interactive session handler (OMShell), and DrModelica with the electronic notebook, and an Eclipse plug-in MDT for large-scale library and compiler development [4]. For interoperability with other tools and languages Modelica models can be represented in XML (The Extensible Markup Language), is in ModelicaXML (Pop 2003[5]), and a general Java-Modelica interface [7]. Furthermore, within this platform the MetaModelica extension of the Modelica language is available for modeling the meaning of languages, e.g. Modelica, and mechanisms for model transformations.

Often in a design process there is a need to make a model parameter sweep in order to find an optimal solution to a design problem, or as an improvement iteration of an existing design. For example, which design parameter values will give the minimum energy consumption in a car model. This often requires the interoperability of several tools. The aim of this paper is to illustrate the capabilities of the OpenModelica platform in this aspect. The parameter sweep is performed as a Monte Carlo simulation of a Modelica model (Käll Dahl 2007 [6]).

The Python language is used to set up the simulation parameters, the uncertainty distributions, as well as calling the OpenModelica compiler and simulator with the updated parameters in each step. The model variable of interest in the results can either be further analyzed in Python or exported to a Matlab file for further analyses.

*Keywords:* *OpenModelica, Python, interoperability, Modelica.*

## 1 The Modelica Language

In the late 90<sup>th</sup> people from both the academic community and industry joined forces to define and standardize a multi-domain modeling language, Modelica. Like other languages designed at the time the object-oriented approach for modeling the dynamic behavior of engineering systems was adopted. In December 1999, the first version of Modelica, version 1.3 was released. Since then, the Modelica design group has had many meetings, resulting new versions of the Modelica language. For example, the latest release, Modelica 3.1, introduces the `decouple` operator for parallel computing, among other new features (Modelica 2009 [1]).

The freely available Modelica Standard Library (MSL) contains many model components and examples from different application domains. Such components can be easily adopted by the modeler and integrated in his/her own application model. The inherent acausal capabilities built into the language lets the user express relationship between variables which gives more reusability compared to software based on assignment statements and related constructs as in conventional programming languages.

Furthermore, models in Modelica are described mathematically by Hybrid Differential, Algebraic Equations (HDAEs). The hybrid is referring to the fact that both continuous-time and discrete-time variables are handled, (Bachmann 2006 [9]).

## 2 OpenModelica Platform

In 2002 an initiative was taken by PELAB, [10] to develop an open source platform for the Modelica language based on more than 20 years of in-house experience of research in compiler construction. This effort was gradually expanded, and in 2007 an international consortium was formed to support the open source effort (OpenModelica 2009 [3]). Previously, the only option for the Modelica community, i.e. Modelica users, for simulating Modelica models was to use commercial tools.

The main goal for this open source platform is to create a complete environment for modeling, compiling and simulating Modelica models based on free software. Both the source code and the binaries are freely available and supported for a variety of intended uses in research, teaching as well as in industry.

The platform was originally written in a language called RML (Relational Meta Language), which is a popular formalism for compiler semantics. This formalism allows efficient compilation combined with optimized C code. This was later (2006) replaced by an extension to Modelica itself, MetaModelica, and the whole compiler was migrated to MetaModelica

The OpenModelica environment compiler translates the Modelica model into a flat Modelica code first and then into C code after a couple of more steps. Also an interactive command handler, i.e., a shell, for executing Modelica scripts and functions etc., is also provided in the environment.

## 3 Interoperability

Although the OpenModelica environment is a powerful tool when it comes to modeling engineering problems, there are situations when you need interoperability with other platforms. A general trend in product development is the usage of distributed resources for tackling the increasing complexity of technical systems. In the OpenPROD project, [11] the OpenModelica platform is used as the basis for integrating hardware and software models from different platforms. There are four options for interoperating with the OpenModelica platform.

The first method is via the scripting interface of the platform, The Modelica scripting language (.mos files) which is used in this paper. The OpenModelica compiler is just called with this script as an argument in each iteration.

In the second method you just need the scripting interface once for building the model. The output executable file plus the initialization file for the parameter settings is used in each iteration.

The third option is to use the Corba interface for invoking the compiler and then just use the scripting interface to send commands to the compiler via this interface, [3].

The fourth variant is to use external function calls to directly communicate with the executing simulation process.

Often in a design process there are uncertainties associated with some of the design variables. One way of dealing with these uncertainties is to perform a parameter sweep, e.g. Monte Carlo simulations, in order to decide which parameter value that returns the optimal result. In this section an example of such an interoperability feature in OpenModelica is given in the form of a Monte Carlo simulation. The underlying Modelica model used in this paper for illustrating the platform's capabilities is based on ecohydrological feedbacks in arid regions, [12]. The rainfall is in this case the uncertain parameter that is swept with a uniform random density function. The result is exported to MATLAB, [13], for further analyses.

### 3.1 Monte Carlo Simulations

The Monte Carlo simulation is performed in such way that a random parameter is generated in a Python script and then passed to OpenModelica via the scripting interface. The Python scripts in this example for performing a parameter sweep on a Modelica model are divided in three scripts, see Appendix B, `MC_RootMoisture.py`, `make_OMScript.py`, and `sim.py`, see Fig. 1.

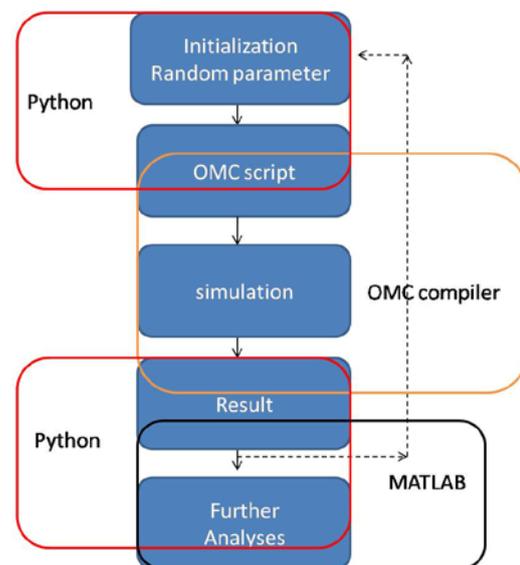
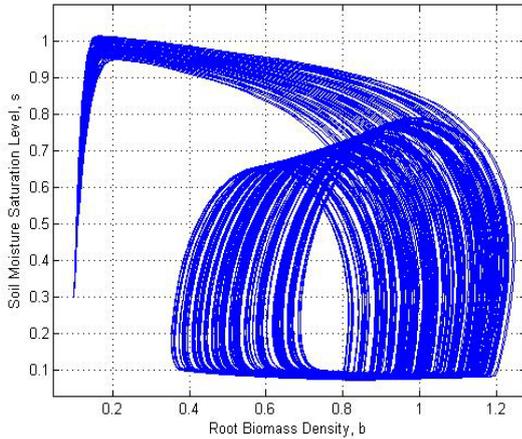


Figure 1. The execution sequence.

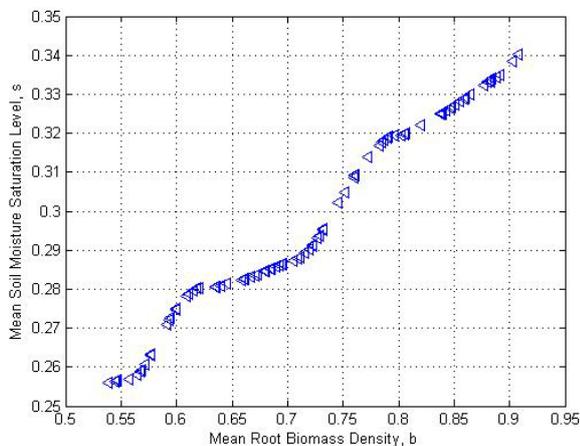
The first script is responsible for initiating the simulations by calling the other Python scripts and calling OpenModelica through the command interface.

In this example the result is passed to MATLAB for further calculations. The analyses performed here are the mean value and standard deviation calculations of the root biomass density against the soil moisture saturation level for each iteration.



**Figure 2.** The biomass (x-axis) vs. the moisture saturation level (y-axis) after each iteration

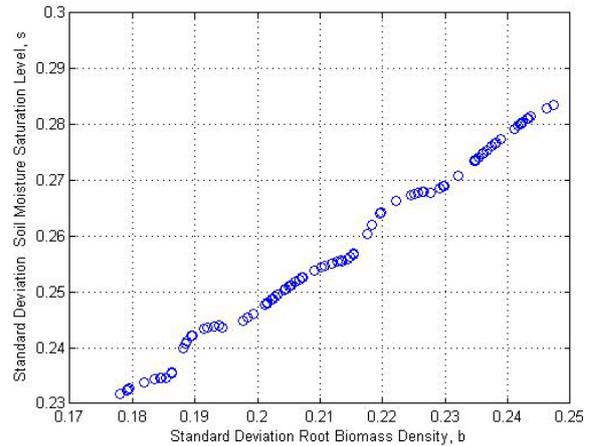
The root biomass density is plotted against the soil moisture saturation level in Fig 2.



**Figure 3:** Mean biomass vs. moisture saturation level after each loop

In Fig 3 the mean value of biomass is plotted against the moisture saturation level after each step.

The standard deviation is illustrated in Fig 4. for the two important variables in this problem.



**Figure 4:** Standard deviation of biomass vs. moisture after each iteration

## 4 Conclusions

The Modelica language has proven itself when it comes to modeling complex multi-domain engineering problems such as aircraft dynamics, industry robots, etc. In addition to the usefulness of the language the free open source OpenModelica platform opens the opportunities of interoperating with other tools and platforms.

The interoperability of the platform was demonstrated in this paper where the Modelica model, described in Appendix A, was simulated from Python. In this paper the OpenModelica command interface was used to start the simulations from Python. This strategy uses the Python language for setting up the uncertain parameter and pass it to the OpenModelica and repeat this step as in Monte Carlo simulations.

In addition to the parameter sweep initiated from Python the result is passed to MATLAB for further analyses. The result is then stored in a MATLAB file as a matrix after each step with additional commands for performing on these matrixes.

A better and faster way for performing parameter sweep is to update the parameter initialization file and call the executable file in every step instead of calling the openmodelica compiler,

## 5 Acknowledgements

This paper was supported by the EU project Lila, (Richter 2009 [8]), by the ITEA2 OPENPROD project,[11], and by VR.

## References

- [1] Modelica Association (2009) "Modelica - A Unified Object-Oriented Language for Physical

- Systems Modeling - Language Specification Version 3.1," May, 2009.
- [2] Fritzon Peter (2004). Principles of Object-Oriented Modeling and Simulation - with Modelica 2.1. Wiley - IEEE Press, 2004.
  - [3] Open Source Modelica Consortium (2009) "OpenModelica System Documentation - version 1.5 Beta", www.openmodelica.org, September 2009.
  - [4] <http://www.eclipse.org>
  - [5] Pop Adrian (2003) "ModelicaXML: A Modelica XML Representation with Applications", Modelica conference 2003, Linköping, Sweden.
  - [6] Källdahl Malin (2007), Separation Analysis with OpenModelica. Linköping, Sweden, Master thesis, Dept. of Electrical Engineering, 2007.
  - [7] Martin Sjölund and Peter Fritzon. An OpenModelica Java External Function Interface Supporting MetaProgramming. In Proc. of (Modelica'2009), Como, Italy, September.20-22, 2009.
  - [8] Thomas Richter, David Boehringer and Sabina Jeschke. Lila: A European Project on Networked Experiments, In Proc. of International Association of Online Engineering 2009, Bridgeport, USA, 2009.
  - [9] Bernhard Bachmann, Peter Aronsson, Peter Fritzon. Robust Initialization of Differential Algebraic Equations, In Proc. of (Modelica'2006), Vienna, Austria, 2006.
  - [10] [www.ida.liu.se/pelab](http://www.ida.liu.se/pelab)
  - [11] [www.openprod.org](http://www.openprod.org)
  - [12] Private communications with Willis Gwenzi from University of Western Australia
  - [13] [www.mathworks.com](http://www.mathworks.com)
  - [14] [www.python.org](http://www.python.org)

## Appendix A The Studied Model

The used Modelica example, above, that the Monte Carlo simulations are performed on in this paper is a model for feedback interactions between root biomass and soil moisture. In this model the state variables are the root biomass density,  $b$ , and the soil moisture saturation level,  $s$ .

```
// RootMoistureModel.mo
model RootMoistureModel
  Real s(start=0.3), b(start=0.1),
    T, L, Es;
  parameter Real c=0.43, B=12, u=0.125,
```

```
Ke=0.01, I0=0.7, Kd=0.8,
rainfall=0.4, Ks=0.4, Ki=0.6,
Ku=0.3, Kc=1.0, sc=0.1, sh=0.03,
sw=0.05, sfc=0.3, Ep=1, g=1, f=0.02,
K3=0.1, r=0.2, Kb=1.2;
equation
  der(s)=rainfall*(b+Ki*I0)/(b+Ki)-
    T*b-L*(b+Ks*Kd)/(b+Kd)-Es*(1-
    Kb*b/(b+Ke))+r*s*b;
  //Soil water ODE //
  der(b)=c*s/(s+Kc)*T*b*(1-b)-u*(1-
    s/(s+Ku))*b7+r*s*b;
  //Root biomass balance ODE//
  Es = if s<=sh then 0
        else if (sh<s and s<=sfc) then
          Ep*(s-sh)/(sfc-sh)
        else Ep*1;
  T = if s<=sw then 0
        else if sw<s and s<=sc then
          g*(s-sw)/(sc-sw)
        else g*1;
  L = if s<sfc then 0
        else (1/(2.718^(B*(1-sfc))-
          1))*(2.718^(B*(s-sfc))-1);
end RootMoistureModel;
```

The parameter  $c$  is the maximum water usage efficiency converting the transpiration to root biomass. The  $B$  is a drainage parameter and  $u$  the maximum intrinsic death rate depending on soil moisture. The parameters  $K_e$  and  $K_d$  are saturation levels corresponding to the dependence of the bare soil evaporation,  $E_s$ , on vegetation and drainage dependence on root biomass respectively. The infiltration level in an unvegetated soil is set in  $K_0$ . The parameters  $K_i$ ,  $K_u$ , and  $K_c$  are saturation levels corresponding to the infiltration dependence on vegetation, intrinsic rate dependence on soil moisture, and root water usage efficiency on soil moisture. The fraction of precipitation available for infiltration is set with  $s_0$  and scaled saturated hydraulic conductivity with  $K_s$  respectively. The  $sh$  and  $sw$  are responsible for the soil moisture saturation at hygroscopic point and permanent wilting point respectively. The parameter  $sc$  is the threshold of soil moisture saturation between which root mortality exceeds births. The  $Ep$  is maximum bare soil evaporation under non-limiting soil moisture and  $sfc$  the soil moisture saturation at field capacity. The  $g$  is maximum transpiration parameter under non-limiting soil moisture.

## Appendix B The Python Code

The Python scripts used in this example are described below. The execution flow, also shown in Fig 1. is:

1. Initiating the uncertain parameter through a random generator.
2. Calling the `make_OMScript.py` for setting up the `*.mos` script (input argument to the OpenModelica compiler)

3. Calling the OpenModelica compiler (starting a simulation)
  4. Retrieving the simulation result (sim.py)
- Iteration and passing the result to MATLAB.

The file MC\_RootMoisture.py sets up the uncertain parameter, makes the mos script, calls the OpenModelica compiler with the mos script as argument and finally calling the MATLAB.

```
# MC_RootMoisture.py
import sys,os, random
global rain # The uncertain variable
global max_iter
max_iter = 100
random.seed() # random generator
for k in range(1,max_iter):
rain=str(random.uniform(0.1,0.8))
execfile('make_OMscript.py')
os.popen(r"OMCBIN\omc.exe
RootMoistureModel.mos").read()
execfile('sim.py')
os.system(r'MATLABBIN\matlab.exe -r
OutputMatlab') # Calling matlab
```

The make\_make\_script.py makes the OpenModelica script file, mos-file with a new value of the uncertain parameter.

```
# make_OMscript.py
mos_file=open('RootMoistureModel.mos',
'w',1)
mos_file.write("loadFile(\\"
RootMoistureModel.mo\");\n")
# Writing mos-commands
mos_file.write("setComponentModifierValue
(RootMoistureModel,rainfall,Code
(="+str(rain)+") );\n")
mos_file.write("simulate(RootMoistureModel
, stopTime=150);\n")
mos_file.close()
```

The sim.py is called after each simulation for retrieving the result and pass it to MATLAB.

```
#sim.py
def zeros(n): # vector initialization
vec = [0.0]
for i in range(int(n)-1):
vec = vec + [0.0]
return vec
```

```
res_file=open("RootMoistureModel_res.plt",
'r',1) # Opening the result file
line=res_file.readline()
# skip first line
size=int(res_file.readline().split('=')
[1])
# Read simulation interval size
time= zeros(size)
b = zeros(size)
# Read the result to variables
while line != ['DataSet: time\n']:
line=res_file.readline().split(',')
for j in range(int(size)):
time[j]=float(res_file.readline().split
(','))[0]
while line != ['DataSet: b\n']:
line=res_file.readline().split(',')[0:1]
for j in range(int(size)):
b[j]=float(res_file.readline().split
(','))[1]
res_file.close()
# pass results to MATLAB
mat_file=open('OutputMatlab.m','a+',1)
mat_file.write("%time b" + "\n")
mat_file.write("b = [ \n")
for i in range(int(size)) :
mat_file.write(str(time[i])+", "+
str(b[i])+";\n")
mat_file.write("]; \n")
mat_file.write("rain = "+str(rain)+ " ;
\n")
mat_file.write("mean_b = mean(b(:,2)) \n")
mat_file.write("std_b = std(b(:,2)) \n")
mat_file.write("figure(1); \n")
mat_file.write("plot(b(:,1),b(:,2)); \n")
mat_file.write("grid; \n")
mat_file.write("hold on; \n")
mat_file.write.close()
```

It should be perhaps noted that there are other options for using the OpenModelica compiler interface. In this small example the \*.mos file script was set up in every iteration. A faster way is to use the command buildModel once outside the iteration loop instead of the simulate command and set up the uncertainty parameter in the initial parameter file, RootMoistureModel\_init.txt instead of using the command setComponentModifierValue. Then the file RootMoistureModel.exe is just called in each step. The reason why this method will be faster is that there is no need to restart the OMC in the iteration sequence. You only invoke one OMC. This can be improved one step further by using the Corba interface of OpenModelica, [3].