

An OpenModelica Python Interface and its use in PySimulator

Anand Kalaiarasi Ganeson¹, Peter Fritzson¹, Olena Rogovchenko¹, Adeel Asghar¹, Martin Sjölund¹
Andreas Pfeiffer²

¹PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, SE-581 83 Linköping, Sweden

²Institute of System Dynamics and Control, German Aerospace Center DLR, Oberpfaffenhofen
¹ganan642@student.liu.se, {peter.fritzson, olena.rogovchenko, adeel.asghar, martin.sjолund}@liu.se
²Andreas.Pfeiffer@dlr.de

Abstract

How can Python users be empowered with the robust simulation, compilation and scripting abilities of a non-proprietary object-oriented, equation based modeling language such as Modelica? The immediate objective of this work is to develop an application programming interface for the OpenModelica modeling and simulation environment that would bridge the gap between the two agile programming languages Python and Modelica.

The Python interface to OpenModelica – OMPython, is both a tool and a functional library that allows Python users to realize the full capabilities of OpenModelica's scripting and simulation environment requiring minimal setup actions. OMPython is designed to combine both the simulation and model building processes. Thus domain experts (people writing the models) and computational engineers (people writing the solver code) can work on one unified tool that is industrially viable for optimization of Modelica models, while offering a flexible platform for algorithm development and research.

Keywords: Python, OpenModelica, OMPython, Python, simulation, modeling, Modelica, Python simulator.

1 Introduction

Necessity is the mother of all inventions. Often in science and engineering, the insufficiency of available tools for researchers and developers creates difficulties in exploring and investigating a certain subject. This creates incentives to develop new infrastructures and tools to fill the void. The goal behind the creation of the Python interface to OpenModelica is to create a free, open source, highly portable, Python based interactive session handler for Modelica scripting and modeling, thus catering to the needs of the Python user community.

OMPython – the Python interface to OpenModelica is developed in Python using tool communication based on OmniORB and OmniORBpy - high performance CORBA ORBs for Python. It provides seamless support to the Modelica Standard Library and the Modelica Language Specification [3] supported by OpenModelica [2].

OMPython provides user-friendly features such as:

- Interactive session handling, parsing, interpretation of commands and Modelica expressions for evaluation, simulation, plotting, etc.
- Creating models, using pre-defined models, making component interfaces and annotations.
- Interface to all OpenModelica API calls.
- Optimized result parser that gives access to every element of the OpenModelica Compiler's (OMC) output.
- Helper functions to allow manipulation of nested dictionary data types.
- Easy access to the Modelica Standard library and calling of OpenModelica commands.
- Provides an extensible, deployable and distributable unit for developers.

Since OMPython is designed to function like a library, it can be used from within any Python application that requires the OpenModelica services. OMPython uses the CORBA implementation of OmniORB and OmniORBpy to communicate with the OpenModelica compiler.

2 Using OMPython

This section describes how to use OMPython and also demonstrates its use in creating a simple Modelica model callable from the Python interpreter. It also presents the two modes of operation specifically designed for testing OpenModelica commands and using the OMPython API as a Python library [1].

2.1 Installing OMPython

The two requirements for the operation of the API are installations of OpenModelica 1.8.1 and Python 2.6.

Since OMPython is supplied together with the OpenModelica installer, the standard source distribution of the API can be used to install it to the third party libraries of the installed Python version. Building and installing the module, for example in the Windows systems, is as simple as running one line of command from the terminal.

```
python setup.py install
```

Now OMPython can be imported into any Python application.

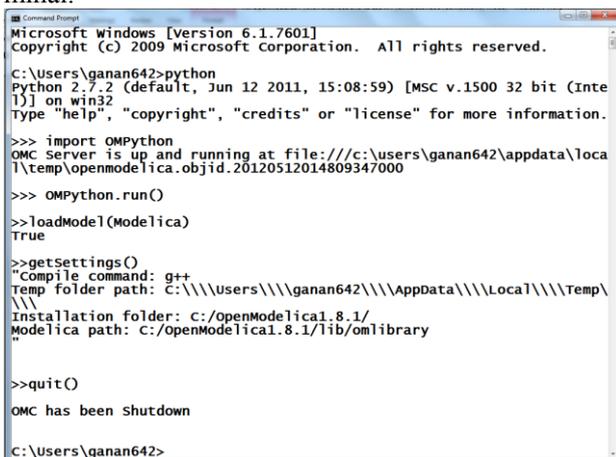
2.2 Executing OMPython

The API can be used in two modes, *Test* and *Library*, each designed for a specific purpose.

2.2.1 Test

Like any new tool, it is important to give its users the freedom to easily explore its capabilities, try its features and possibly suggest new improvements.

For this purpose, the API can be executed in the test mode by executing the `run()` method of the OMPython module. This mode allows users to interactively send OpenModelica commands to OMC via the CORBA interface. The Python types of the OpenModelica output are returned to the user. To illustrate this, in Figure 1 a few operations are presented from the Python terminal.



```
Microsoft windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\ganan642>python
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> import OMPython
OMC Server is up and running at file:///c:/users/ganan642/appdata/local/temp/openmodelica.objid.20120512014809347000

>>> OMPython.run()

>>> loadModel(Modelica)
True

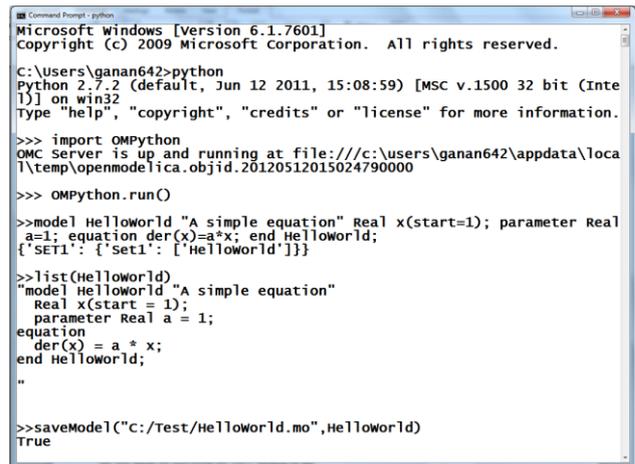
>>> getSettings()
"compile command: g++
temp folder path: C:\\\\Users\\\\ganan642\\\\AppData\\\\Local\\\\Temp\\\\
Installation folder: C:/OpenModelica1.8.1/
Modelica path: C:/OpenModelica1.8.1/lib/omlibrary

>>> quit()
OMC has been shutdown

C:\Users\ganan642>
```

Figure 1. OMPython executing OpenModelica commands in the Test mode.

Creating new models in the text based Python terminal is rather straightforward using OMPython. Figure 2 illustrates this and shows how a model can be saved with a simple command.



```
Microsoft windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\ganan642>python
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> import OMPython
OMC server is up and running at file:///c:/users/ganan642/appdata/local/temp/openmodelica.objid.20120512015024790000

>>> OMPython.run()

>>> model Helloworld "A simple equation" Real x(start=1); parameter Real a=1; equation der(x)=a*x; end Helloworld;
{"set1": {"set1": ["Helloworld"]}}

>>> list(Helloworld)
"model Helloworld "A simple equation"
  Real x(start = 1);
  parameter Real a = 1;
  equation
    der(x) = a * x;
  end Helloworld;
"

>>> saveModel("c:/Test/Helloworld.mo",Helloworld)
True
```

Figure 2. Creating and saving a simple HelloWorld model file using OMPython.

2.2.2 Library

Once modelers are familiar with the interface they know what type of responses can be expected and can use the module as a library to programmatically design, simulate, plot, and do more with the models.

This can be done by executing the `execute()` method of the OMPython module. The `execute` method forms the essence of the OMPython API. It encapsulates the OMC operations, CORBA functionalities, parses the results to native Python data types and exposes the API as a simple string processing method. Each instance of the `execute` method returns a result that the modeler can make use of. Additionally, complicated data structures such as deeply nested dictionaries are constructed, strictly typed, and are made available to the user using this method.

The Code Listing 1 shown below provides a simple Python script that uses OMPython as a library to perform a few tasks like loading Modelica libraries to simulating pre-defined Modelica models. Figure 3 depicts the output of the program generated by OMPython on a standard Python terminal.

Code Listing 1

```
import OMPython
OMPython.execute("loadFile(\"c:/OpenModelica1.8.1/testmodels/BouncingBall.mo\")")

result=OMPython.execute("simulate(BouncingBall, stopTime=2, method='Euler')")

print result
OMPython.execute("quit()")
```

```

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\ganan642>python test_execute_mode.py
OMC Server is up and running at file:///c:/users/ganan642/appdata/local
temp/openmodelica.objid.20120512015521889000

{'simulationOptions': {'options': '***', 'storeInTemp': False, 'cflags':
***, 'simflags': '***', 'variableFilter': '***', 'noClean': False,
'outputFormat': 'mat', 'method': 'dassl', 'measureTime': False, 's
topTime': 2.0, 'startTime': 0.0, 'numberOfIntervals': 500, 'tolerance'
: 1e-06, 'fileNamePrefix': 'BouncingBall'}, 'simulationResults': {'t
imeCompile': 2.69126048770347, 'timeBackend': 0.0219980482622027, 'mes
sages': '***', 'timeFrontend': 0.0262914864581348, 'timeSimulation': 0.
149678656188106, 'timeTemplates': 0.0174994273747587, 'timesimCode': 0
.00998658420863192, 'timeTotal': 2.9168286378063, 'resultFile': 'c:/u
sers/ganan642/BouncingBall_res.mat'}}

OMC has been shutdown

C:\Users\ganan642>_

```

Figure 3. OMPython executing the Python script shown above.

3 Deploying OMPython in PySimulator

PySimulator is a Python-based Simulation and Analysis tool that is developed by the German Aerospace Center (DLR) in Germany. The tool uses plugins for simulators based on Dymola [10], FMUs [11], and OpenModelica [2]. It also provides analysis tools for some applications particularly in physics and engineering.

This section shows the integration of the new OpenModelica simulator plugin for PySimulator using OMPython.

3.1 The OpenModelica Plugin

The plugin for the OpenModelica simulator integrates easily and well into the PySimulator package by using the OMPython library. PySimulator's template for the plugins provides convenient methods to implement simulation routines, parameter settings, retrieve and use simulation variables and more. Figure 4 shows a part of the development package of PySimulator that includes the OpenModelica plugin.

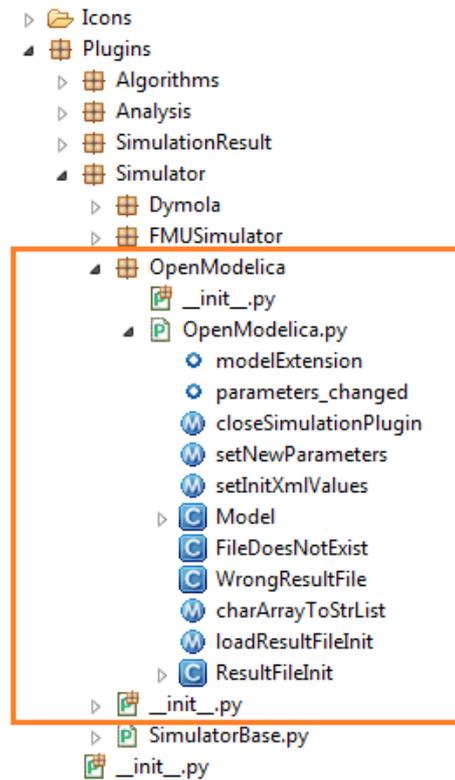


Figure 4. OpenModelica plugin using OMPython within PySimulator.

The OpenModelica plugin defines and uses some features of PySimulator for performing simulations, reading result files, and displaying variables etc. The plugins use PySimulator's plugin templates; this allows other simulation packages to be integrated easily.

The deployment of the OpenModelica plugin within the PySimulator project allows the project to benefit from the full scripting capabilities of the latest OpenModelica API.

3.2 Loading a Modelica Model

The integration of the OMPython module within the OpenModelica plugin for PySimulator makes it possible for the modeler to quickly load Modelica files such as models (.mo) or load a simulated model's executable file.

The user can open these files from the menu bar by selecting `File > Open Model > OpenModelica`.

In this introductory example we will use a pre-defined model named `Influenza` to demonstrate the use of OMPython in PySimulator. Figure 5 depicts the graphical user interface of PySimulator when opening a model file. Once the model file is selected, the model is loaded into the variables browser and is ready to be configured for simulations.

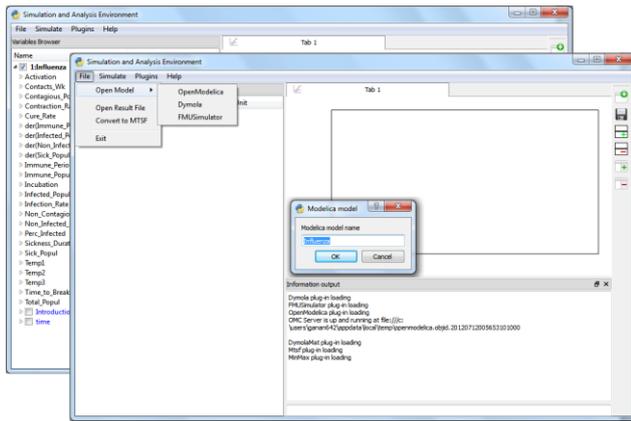


Figure 5. Loading Modelica models or model executables in PySimulator

3.3 Using the OpenModelica plugin

The loaded Modelica model can be simulated from PySimulator using the default simulation options or by setting the simulation options before simulating from the Integrator Control dialog box. The OpenModelica plugin defines the simulation routine for the Modelica models by using the execute method of the OMPython API.

Figure 6 shows how the simulation options can be set using PySimulator's Integrator control feature.

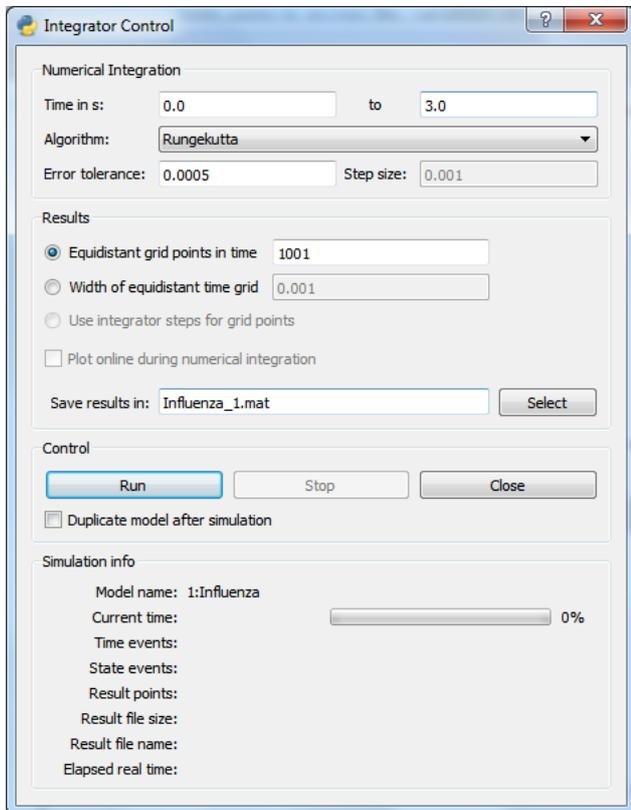


Figure 6. Preparing the simulation settings using the Integrator Control.

3.4 Simulating the model

The initial simulation parameters and settings are provided as inputs to the OMC via the front-end of PySimulator. The Run button of the Integrator control triggers the simulate command of the OMC with the supplied simulation options. The simulate command has the following parameters,

- Simulation Interval
 - Start Time
 - Stop Time
- Algorithm
- Error Tolerance
- Step size

The user has the option to choose from a range of Numerical integration algorithms from the Algorithm selection box. The Integrator control dialog box also filters some parameters that are not available for some integration solvers by disabling the field; avoiding error and providing more accuracy in the results.

The Variables browser builds a tree structure of the instance variables and highlights time-continuous variables in blue. The user can select these variables and plot them in the Plot window by checking the check box near the highlighted variables.

Figure 7 illustrates the Variables browser that allows users to access the variables after the Influenza model has been simulated with some simulation parameters set.

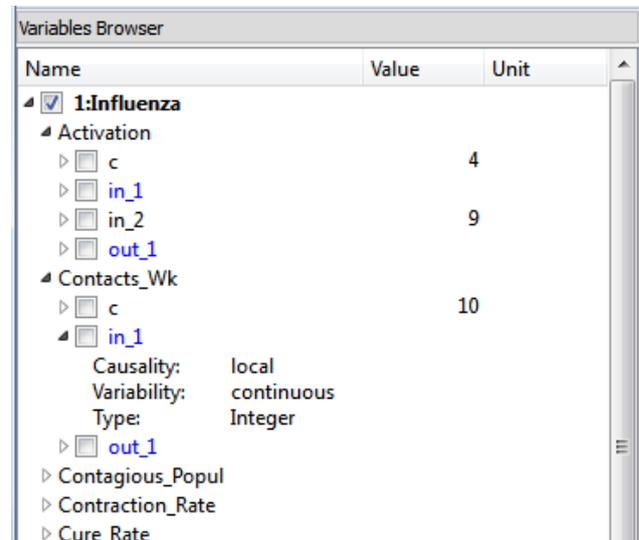


Figure 7. Variables browser of the simulated model.

3.5 Plotting variables from the simulated models

The Plot window of the PySimulator GUI provides additional user interface controls for comparing different

plots side-by-side, adding and removing plots and also to save the plots.

Figure 8 shows the plotted variables in the plot window and the list of simulation variables in the Variables browser along with the variables selected for plotting.

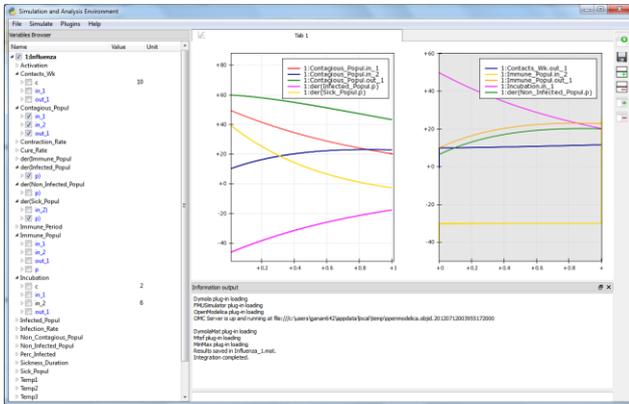


Figure 8. Plotted variables using PySimulator.

3.6 Using Simulated results

It is desirable to avoid simulating the model again every time the user needs the simulation results. It is instead preferable to use an existing simulation result file for the calculations, this saves resources and time. Py-Simulator supports opening the OpenModelica simulation result files (.mat) and the model's executable file to build the variable tree in the variables browser. The user can then adjust some parameters from the variable tree or the Integrator control to achieve the desired results.

4 The OMPython API

The Python interface to OpenModelica addresses its functional requirements through the implementation of two interrelated modules, OMPython and OMParser [1]. This section introduces the two modules and demonstrates their functionalities with some examples.

The following Figure 9 illustrates the functions of the OMPython API with its components.

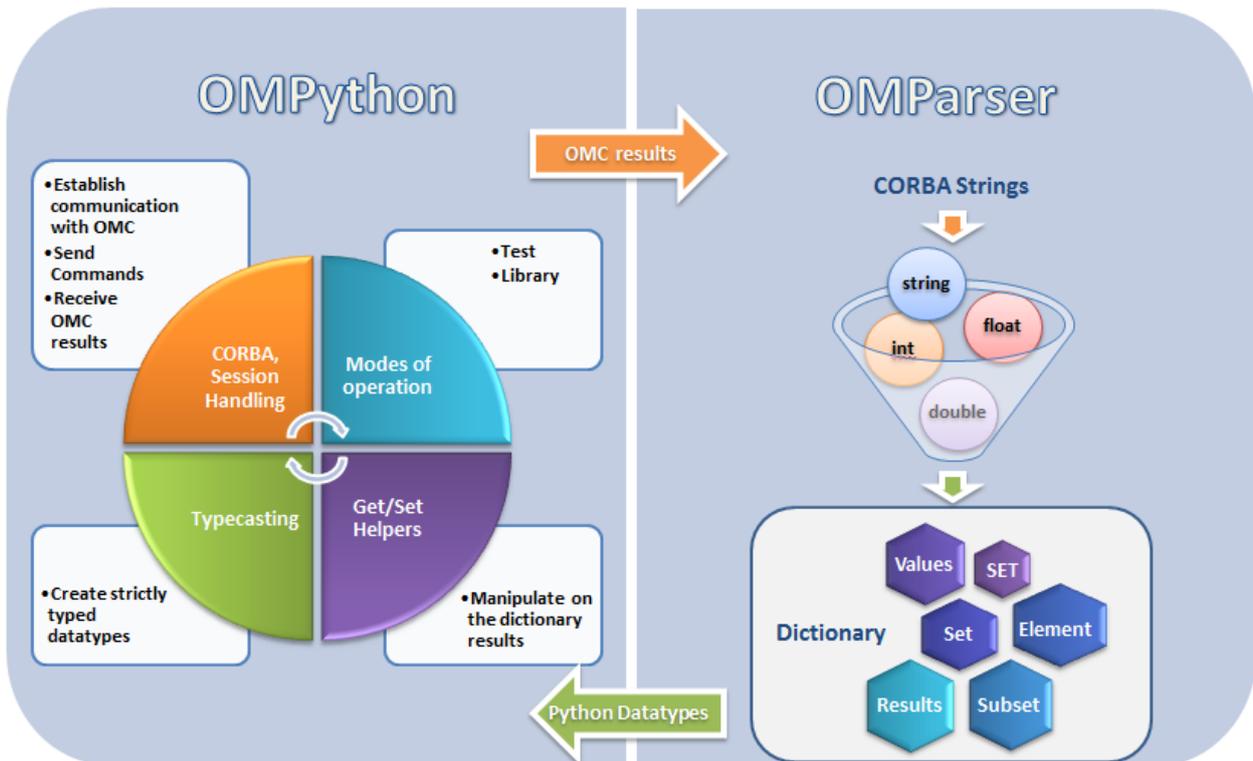


Figure 9. Functions of the OMPython API

4.1 OMPython module

The OMPython module is the main interfacing module of the OMPython API which is responsible for providing the API as a tool and a Python library.

The following are its components:

4.1.1 Interactive Session handler

Each instance of the module creates an interactive session between the user and the OMC. The session handler uses the CORBA Interoperable Object Reference (IOR) file to maintain the user's session activ-

ities and log files for standard output and errors. The session is closed down when the user issues the `quit()` command to the OMC. This also removes the temporary IOR file from the user's machine. The log files facilitate the user with some straight forward debugging and trace-backing purposes.

4.1.2 CORBA Communication

OMPpython uses the Client-Server architecture of the CORBA mechanism to interact with the OMC. OMPython implements the client side of the architecture.

4.1.3 Modes of Operation

The module defines two modes of operation, each designed for specific purposes.

- Test
- Library

The Test mode allows users to test OMPython while the Library mode gives the user the ability to use the results of OMPython.

4.1.4 Using the interface definition

The vital link between the client and the server processes in this distributed implementation is the Interface Definition Language (IDL) file. OMC defines the `omc_communication.idl` file that it uses to implement the Remote Procedure Calls (RPCs), OMPython mirrors this IDL file to establish the RPC from the client machine.

4.1.5 Get/Set helper functions

Due to the nature of the complicated string outputs generated by the OMC such as Component Annotations, the parser module of the OMPython module generates nested dictionaries. Deeply nested dictionaries in Python require cumbersome operations to retrieve and set values inside dictionaries at various levels. To simplify the multiple steps necessary to perform a get or set operation within a dictionary, OMPython defines the dot-notation `get/set` methods. Figure 10 shows how the user can get and set the values of any nested dictionary data type.

```

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\ganan642>python test_get_set.py
OMC Server is up and running at file:///c:/Users/ganan642/appdata/local
temp/openmodelica.objid.20120512014034211000

{'SET1': {'Set1': [-100.0, -100.0, 100.0, 100.0, True, 0.1, 2.0, 2.0]}
, 'SET2': {'Elements': {'Line1': {'Properties': {'Values': [True, 0,
LinePattern.Solid', 0.25, 3, 'Smooth.None']}, 'Subset1': {'Set1': [-96,
0], 'Set2': [-70, 0]}, 'Set1': [0, 0, 0, 0], 'Set2': [0, 0, 255]}, 'Set3
': ['Arrow.None', 'Arrow.None']}}, 'Rectangle1': {'Properties': {'Valu
es': [True, 0, 'LinePattern.Solid', 'FillPattern.None', 0.25, 'BorderP
attern.None', 0], 'Subset1': {'Set1': [-70, 30], 'Set2': [70, -30]}, '
Set1': [0, 0, 0, 0], 'Set2': [0, 0, 255]}, 'Set3': [0, 0, 0]}, 'Line2':
{'Properties': {'Values': [True, 0, 'LinePattern.Solid', 0.25, 3, 'Smo
oth.None']}, 'Subset1': {'Set1': [70, 0], 'Set2': [96, 0]}, 'Set1': [0,
0, 0, 0], 'Set2': [0, 0, 255]}, 'Set3': ['Arrow.None', 'Arrow.None']}}]}

Values of Line 1 are:
[True, 0, 'LinePattern.Solid', 0.25, 3, 'Smooth.None']

The new values of Line 1 are:
[0, 1, 2, 3]

OMC has been shutdown

C:\Users\ganan642>_

```

Figure 10. Get/Set helper function

4.1.6 Universal Typecaster

Since the variables in Python are dynamically typed, the interpretation of the data types needs to be strictly controlled during runtime. For this purpose, the OMPython module defines a universal typecasting function that typecasts the data to the correct types before building the results.

4.1.7 Imports OMParser

Although the OMC outputs the results to the OMPython module via its CORBA interface, the results are still in the String-to-String CORBA output format which cannot be used intelligibly. So the OMPython module uses its own built-in parser module the OMParser to generate appropriate data structures for the OMC retrieved results.

4.2 OMParser module

Since the results of the OMC are retrieved in a String format over CORBA, some data treatment must be done to ensure that the results are usable correctly in Python.

The OMParser module is designed to do the following,

- Analyze the result string for categorical data.
- Group each category under a category name
- Typecast the data within these categories
- Build suitable data structure to hold these data so that the results are easily accessible.

4.2.1 Understanding the Parsed output

Each command in OpenModelica produces a result that can be categorized according to the statistics of the pattern of data presented in the text. Grammar based parsers were found to be tedious to use be-

cause of the complexity of the patterns of data. This is also the case because the OpenModelica implementation has two types of APIs. One is typed, which could use grammar and the other is untyped, which cannot.

OMPParser follows a few simple rules to parse the OMC output:

- Result strings that do not contain a pair of curly braces "{}" are simply typecasted to their respective types.

For example:

```
>>getVectorizationLimit()
20
>>getNthInheritedClass(Modelica.Electrical.Analog.Basic.Resistor,1)
Modelica.Electrical.Analog.Interfaces.OnePort
```

- Result strings that include one or more pairs of curly braces "{}" are categorized for making dictionary types.

For example:

```
>>getClassNames()
{'SET1':{'Set1': ['ModelicaServices', 'Modelica']}}
```

- Data contained within double quotes "" are formatted to string types; removing the escape sequences in-order to keep the semantics.

For example:

```
>>getModelicaPath()
"C:/OpenModelica1.8.0/lib/omlibrary"
```

4.2.2 The Dictionary data type in Python

Dictionaries are useful as they allow to group data with different data types under one root dictionary name. Dictionaries in Python are indexed by keys unlike sequences, which are indexed by a range of numbers.

It is best to think of dictionaries as an unordered set of *key:value* pairs, with the requirement that the keys are always unique. The common operation on dictionaries is to store a value associate with a key and retrieve the value using the key. This provides us the flexibility of creating keys at runtime and accessing these values using their keys later. All data within the dictionary are stored in a named dictionary. An empty dictionary is represented by a pair of braces {}.

In the result returned by the OMC, the complicated result strings are usually the ones found within the curly braces. In order to make a meaningful categorization of the data within these brackets and to avoid the potential complexities linked to creating dynamic variables, we introduce the following nota-

tions that are used within the dictionaries to categorize the OMC results,

- SET
- Set
- Subset
- Element
- Results
- Values

In this section, to explain these categories, we use the parsed output of OMPython obtained using the Test mode.

4.2.3 SET

A SET (*note the capital letters*) is used to group data that belong to the first set of balanced curly brackets. According to the needed semantics of the results, a SET can contain *Sets, Subsets, Elements, Values and Results*.

A SET can also be empty, denoted by {}. The SETs are named with an increasing index starting from 1 (one). This feature was planned to eliminate the need for dynamic variable creation and having duplicate Keys. The SET belongs within the dictionary called "result".

For example:

```
>>strtok("abcbdef","b")
{'SET1':{'Values': ['"a","c","def"]}}
```

The command `strtok` tokenizes the string "abcbdef" at every occurrence of `b` and produces a SET with values "a", "c", "def". Each value of the SET is then usable in Python.

4.2.4 Set

A set is used to group all data within a SET that is enclosed within a pair of balanced {}s. A Set can contain only Values and Elements. A set can also be empty, it can be depicted as {}, the outer brackets compose the SET, the inner brackets are the Set within the SET.

4.2.5 Subset

A Subset is a two-level deep set that is found within a SET. A subset can contain multiple Sets within its enclosure.

For example:

```
{SET1 {Subset1 {Set1},{Set2},{Set3}}}
```

4.2.6 Element

Elements are the data which are grouped within a pair of Parentheses (). As observed from the OMC result strings, elements have an element name that describes the data within them, so elements can be grouped by their names.

In some cases such as when using the untyped OpenModelica API calls, element structures do not have a name, in these cases the data contained within the parenthesis is parsed into outputs generated by the typed API calls, such as set, values, etc. Also, in some cases many elements have the same names, so they are indexed by increasing numbers starting from 1 (one). Elements have the special property of having one or more Sets and Subsets within them. However, they are still enclosed within the SET.

For example:

```
>>getClassAttributes(test.mymodel)
{'SET1': {'Elements': {'recl':
{'Properties': {'Results': {'comment':
None, 'restriction': 'MODEL',
'startLine': 1, 'partial': False,
'name': "mymodel", 'encapsulated':
False, 'startColumn': 14, 'readonly':
"writable", 'endColumn': 69,
'file': "<interactive>", 'endLine': 1,
'final': False}}}}}}}
```

In this example, the result contains a SET with an Element named `recl` which has Properties which are Results (see section 4.2.7) of the element.

4.2.7 Results

Data that is related by the assignment operator "=", within the SETs are denoted as Results. These assignments cannot be assigned to their actual values unless they are related by a Name = Value relationship. So, they form the sub-dictionary called Results within the Element (for example). These values can then be related and stored using the *key:value* pair relationship.

For example:

```
>>getClassAttributes(test.mymodel)
{'SET1': {'Elements': {'recl':
{'Properties': {'Results': {'comment':
None, 'restriction': 'MODEL',
'startLine': 1, 'partial': False,
'name': "mymodel", 'encapsulated':
False, 'startColumn': 14, 'readonly':
"writable", 'endColumn': 69, 'file':
"<interactive>", 'endLine': 1,
'final': False}}}}}}}
```

4.2.8 Values

Data within any or all of SETs, Sets, Elements and Subsets that are not assignments and separated by commas are grouped together into a list called "Values". The Values list may also contain empty dictionaries, due to Python's representation of a null string "" as {} - an empty dictionary. Although a null string is still a null value, sometimes it is possible to observe data grouped into Values to look like Sets within the Values list.

For example:

```
>>getNthConnection(Modelica.Electrical.Analog.Examples.ChuaCircuit,2)
{'SET1': {'Set1': ['G.n', 'Nr.p', {}]}}
```

4.2.9 The Simulation results

The `simulate()` command produces output that has no SET or Set data in it. Instead, for the sake of simplicity, the result contains two dictionaries namely, SimulationResults and SimulationOptions.

For example:

```
>>simulate(BouncingBall)
{'SimulationOptions': {'options': "",
'storeInTemp': False, 'cflags': "",
'simflags': "", 'variableFilter':
".*'", 'noClean': False,
'outputFormat': "mat", 'method':
"dassl", 'measureTime': False,
'stopTime': 1.0, 'startTime': 0.0,
'numberOfIntervals': 500, 'tolerance':
1e-06, 'fileNamePrefix': "BouncingBall"},
'SimulationResults': {'timeCompile': 4.75231650258347,
'timeBackend': 0.016026309771926,
'messages': None, 'timeFrontend': 1.42004668806536,
'timeSimulation': 0.119703995817784,
'timeTemplates': 0.0230460728977474,
'timeSimCode': 0.0139967955849597,
'timeTotal': 6.3452533928534, 'resultFile': "C:/Users/ganan642/BouncingBall_res.mat"}}
```

4.2.10 The Record types

Some commands produce result strings with Record constructs, these data are categorized for making dictionaries too. To keep the uniformity and simplicity, the data of Record types are grouped into the dictionary `RecordResults`.

For example:

```
>>checkSettings()
{'RecordResults': {'RTLIBS': " -static-libgcc -luuid -lole32 -lws2_32", 'OMC_F
```

```

OUND': True, 'MODELICAUSERCFLAGS': None,
'C_COMPILER_RESPONDING': False, 'OPENMODELICAHOME': '"C:/OpenModelica1.8.1/"',
'CREATE_FILE_WORKS': False, 'SYSTEM_INFO': None, 'CONFIGURE_CMDLINE': '"Manually
created Makefiles for OMDev',
'RecordName':
'OpenModelica.Scripting.CheckSettingsResult', 'OMC_PATH': '"C:/OpenModelica1.8.1//
bin/omc.exe"', 'WORKING_DIRECTORY': '"C:/Users/ganan642"', 'REMOVE_FILE_WORKS':
True, 'OS':
'"Windows_NT"', 'OPENMODELICALIBRARY': '"C:/OpenModelica1.8.1/lib/omlibrary"', 'C_C
OMPILER': '"gcc"'}}

```

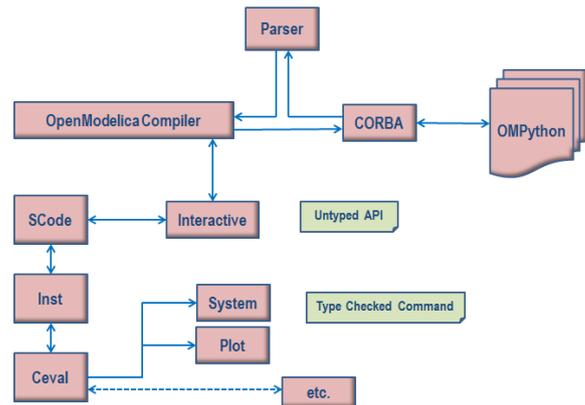


Figure 11. Client-Server of OpenModelica with some interactive tool interfaces

5 OMPython Implementation

The implementation of the OMPython API relies on the Client–Server architecture of CORBA to communicate to the OMC [2]. OMPython acts as the client that requests the services of OMC and OMC behaves like the server and replies to the Python module using the OmniORB and OmniORBpy – Object Request Brokers (ORBs) of CORBA as the communication platform.

This section briefly describes how the API uses CORBA and its other features to achieve its requirements.

5.1 The OMC CORBA interface

The OpenModelica Compiler – OMC can be invoked using two methods:

- Executed at the operating system level, like a program.
- Invoked as a server from a client application using a CORBA client-server interface.

OMPython uses the second method to start OMC since this allows the API to interactively query the compiler/interpreter for its services.

5.2 OMC Client Server architecture

Figure 11 gives an overview of the OpenModelica client server architecture. OMPython plays the role of the client in this architecture. It sends queries and receives replies from the OMC via the CORBA interface. The messages and expressions from the CORBA interface are processed in two groups. The first group consists of the commands which are evaluated by the `Ceval` module and the second group contains the expressions that are handled by the `Interactive` module.

Messages in the CORBA interface are classified into two groups. The first group consists of the user commands or expressions; these are evaluated by the `Ceval` module. The second group contains the declaration of variables, classes, assignments, etc. The client-server API calls are processed by the `Interactive` module.

5.3 Using OMC through CORBA

The OMC process can be invoked from CORBA by executing the OMC executable file using special parameters passed to it. The default location of the OMC executable file is in the `$OPENMODELICAHOME/bin` directory. OMPython invokes OMC with some special flags `+d=interactiveCorba` `+c=random_string` which instructs OMC to start and enable the interactive CORBA communication and also use a timestamp to name the CORBA Interoperable Object Reference (IOR) file that will be created. The timestamp is needed to differentiate the different instances of OMC that have been started by different client processes simultaneously.

The default location where the IOR file is created is in the temp directory. Normally, when OMC is started with the `+d=interactiveCorba` flag, it will create a file named `openmodelica.objid`. On Windows (for example), if the `+c` flag was given, the file name is suffixed with the random string to avoid name conflicts between the simultaneously running OMC server processes. This file contains the CORBA IOR.

5.4 Using the CORBA IOR file

The IOR file contains the CORBA object reference in string format. The CORBA object is created by reading the strings written inside the IOR file.

6 Measurements

In this section, we present some performance measurements for the OMPython API.

The measurements shown are based on the response time of the Python interpreter/compiler that performs the various functions of establishing the CORBA communication, sending commands to the OMC, receiving CORBA outputs, parsing the CORBA outputs and finally displaying the results to the user.

Figure 12 illustrates a simple script that simulates a Modelica model and plots a variable using the Plot generated by OpenModelica. It also shows the received response times of each command that was executed to perform the simulation. Table 1 and Table 2 show the time statistics collected from five unique runs of two simple scripts using the OMPython API. The time is measured in Seconds. Figure 13 and Figure 14 illustrate the overhead between the average output and the unparsed output's response times.

These measurements aim to give an idea about the overhead of the OMPython API in addition to the CORBA overhead that is needed for OMC communication.

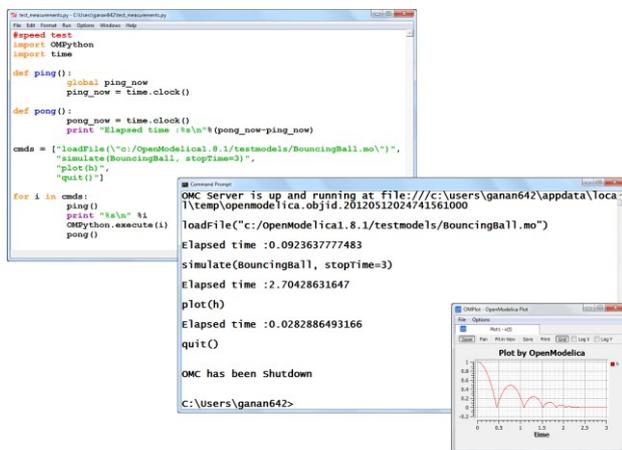


Figure 12. Measuring response times of simulations for the BouncingBall model.

Command	Average response time (s)	Average unparsed response time (s)
loadFile("c:/OpenModelica1.8.1/models/BouncingBall.mo")	0.09223065	0.0421344389

simulate(BouncingBall)	2.60921512	1.8922307169
plot(h)	0.03251472	0.0183359414

Table 1. Response time comparisons for loading, simulating and plotting variables using OMPython.

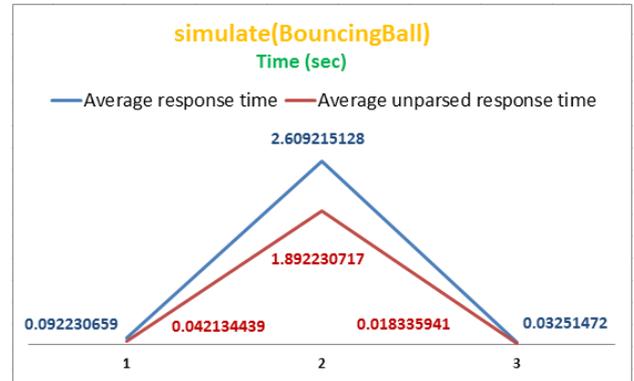


Figure 13. Measuring response time for Simulations in OMPython

Command	Average response time (s)	Average unparsed response time (s)
getVersion()	0.0680588293	0.0590995445
loadModel(Modelica)	5.971103887	4.4708573210
getElementInfo(Modelica.Electrical.Analog.Basic.Resistor)	0.0264064349	0.0190346404
getClassNames()	0.3907942649	0.2707218157
getTempDirectoryPath()	0.0244359882	0.0193691690
getSettings()	0.0327650196	0.0234227783

Table 2. Measuring response times of some OpenModelica commands in OMPython

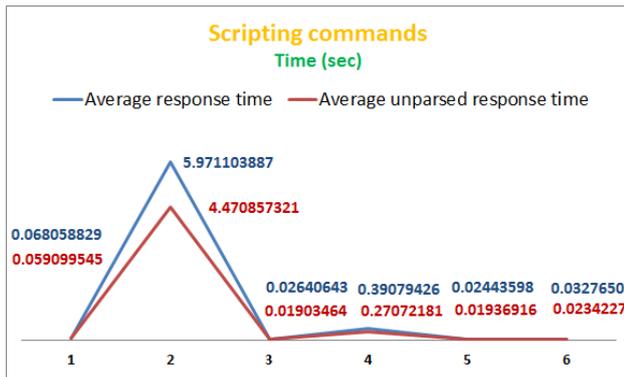


Figure 14. Measuring response times of some OpenModelica commands in OMPython

7 Related Work

Some Simulation packages are available for Python but these packages do not implement an equation-based solving system. Also, they do not provide a Modelica based modeling and simulation environment, but rather present their custom model types.

- *PySCeS* – The Python Simulator for Cellular Systems. It uses the model description language to define its models. Supports solvers like LSO-DA, sections for non-linear root finding algorithms, Metabolic control analysis, Matplotlib/Gnuplot plotting interfaces, etc. It is released under a new BSD style license and is open source software [4].
- *SimPy* – Simulation in Python, is an object-oriented, process-based discrete-event simulation language for Python. It is released under the GNU Lesser GPL (LGPL) license version 2.1. It features data collection capabilities, GUI and plotting packages. It provides the modeler with the active and passive components of a simulation model and monitor variables for gathering statistics [5]
- *JModelica.org* [12], *MWORKS* [13], and *Amesim* [14] are other Modelica tools providing a Python scripting API.

8 Conclusion

OMPpython is a versatile Python library for OpenModelica that can be used by engineers, scientists, researchers and interested architects to explore and develop Modelica based modeling and simulation efforts. It is free, open source and is distributed with the OpenModelica installation which gives the user the potential to use the full collection of Model-

ica libraries that can assist in performing complex simulations and analyses.

The OMPython API places minimal requirements on the user while offering an industry viable standard modeling and simulation environment.

We suggest some future work that can be done to enrich the usage of the OMPython API. The API can be expanded to provide access to the GUI based features of other OpenModelica tools such as OMEdit. User interfaces can be easily built on top of OMPython to implement additional graphic features. Further interesting efforts can be made if the OpenModelica API can be designed to expose its commands as interface definitions in the `omc_communication.idl` file.

9 Acknowledgments

This work has been supported by Serc, by the Swedish Strategic Research Foundation in the EDOp and HIPo projects and Vinnova in the RTSIM and ITEA2 OPENPROD projects. The Open Source Modelica Consortium supports the OpenModelica work.

References

- [1] Anand Kalaiarasi Ganeson. *Design and Implementation of a User Friendly OpenModelica – Python interface*, Master thesis LIU-IDA/LITH-EX-A12/037SE, Linköping University, Sweden, 2012
- [2] Open Source Modelica Consortium. *OpenModelica System Documentation Version 1.8.1*, April 2012. <http://www.openmodelica.org>
- [3] Modelica Association. *The Modelica Language Specification Version 3.2*, March 24th 2010. <http://www.modelica.org>. Modelica Association. *Modelica Standard Library 3.1*. Aug. 2009. <http://www.modelica.org>.
- [4] PySCeS. <http://pysces.sourceforge.net/index.html>
- [5] SimPy. <http://simpy.sourceforge.net/>
- [6] Mark Lutz. *Programming Python*. ISBN 9781449302856, O'Reilly, 2011.
- [7] omniORB 4.1.6 and omniORBpy 3.6. The omni-ORB version 4.1 User's guide, the omniORBpy version 3 User's guide.
- [8] <http://omniorb.sourceforge.net/>
- [9] Andreas Pfeiffer, M. Hellerer, S. Hartweg, Martin Otter, and M. Reiner. *PySimulator – A Simulation and Analysis Environment in Py-*

- thon with Plugin Infrastructure. Submitted to the 9th International Modelica Conference, Munich, Germany, September. 2012.
- [10] Dassault Systèmes AB: la, www.dymola.com.
- [11] MODELISAR consortium: Functional Mock-up Interface for Model Exchange, Version 1.0, 2010. www.functional-mockup-interface.org
- [12] JModelica.org. <http://JModelica.org>. Accessed May 20, 2012.
- [13] MWORKS. <http://en.tongyuan.cc/>. Accessed May 20, 2012.
- [14] LMS Inc. Amesim tool suite. <http://www.lmsintl.com/Imagine-Amesim-suite>. Accessed May 20, 2012.