Design and Implementation of a User
Friendly OpenModelica - Python interface

Anand Kalaigarasi Ganeson

Reg Nr: LIU-IDA/LITH-EX-A12/037SE
Linköping 2012
Master’s Thesis

Design and Implementation of a User Friendly OpenModelica - Python interface

Anand Kalaiarasi Ganeson

Reg Nr: LIU-IDALITH-EX-A12/037SE
Linköping 2012

Supervisor: Olena Rogovchenko
IDA, Linköpings universitet

Examiner: Peter Fritzson
IDA, Linköpings universitet
Språk | Language
--- | ---
☐ Svenska/Swedish
☒ Engelska/English

Rapporttyp | Report category
--- | ---
☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport

ISBN
---
—

ISRN
---
LIU-IDA/LITH-EX-A12/037SE

Serietitel och serienummer | ISSN
--- | ---
Title of series, numbering | —

URL för elektronisk version
---
http://www.ida.liu.se/
http://www.liu.se

Titel | Design och implementation av ett användarvänligt OpenModelica-Python gränssnitt
--- | ---
Title | Design and Implementation of a User Friendly OpenModelica - Python interface

Författare | Anand Kalaiarasi Ganeson
--- | ---
Author

Sammanfattning | 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 thesis 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 simulation and model building. 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.

Nyckelord | Keywords
--- | ---
Python, OpenModelica, simulation, Modeling, Python Simulator, OpenSource
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 thesis 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 simulation and model building. 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.
I am grateful to the almighty God, my parents, teachers and friends for their continuous support and encouragement towards all my endeavors. I would like to thank Peter Fritzson for providing me the opportunity to work on a fun and challenging project with support to appropriate tools and facilities throughout the project, Olena Rogovchenko for introducing me to the project and quickly answering technical questions, Syed Adeel Asghar for his guidance and technical assistance during the integration process and Martin Sjölund for his help during the testing stages of the project. Your help and support for the success of this thesis project is much appreciated.
# Contents

1 Introduction ........................................... 3  
   1.1 Background ........................................ 3  
   1.2 Target Readers ................................... 4  
   1.3 Goals ............................................... 4  
   1.4 Method ............................................. 5  
   1.5 Overview of this thesis report ................... 5  

2 Background ........................................... 7  
   2.1 Modelica ............................................ 7  
   2.2 OpenModelica ..................................... 8  

3 Tools and Communication ............................... 11  
   3.1 OpenModelica Compiler - OMC ...................... 11  
   3.2 CORBA .............................................. 13  
   3.3 OmniORB ........................................... 15  
      3.3.1 Interface Definition Language (IDL) file ... 15  
      3.3.2 Stubs .......................................... 16  
      3.3.3 Object references and Servants .............. 16  
   3.4 OmniORBpy ......................................... 16  
   3.5 The Python Interpreter ............................ 17  

4 Design .................................................. 19  
   4.1 Requirements ...................................... 19  
   4.2 Design .............................................. 19  
      4.2.1 OMPython ..................................... 20  
      4.2.2 OMParser ..................................... 23  
      4.2.3 Understanding the parsed output .......... 24
5 Implementation
  5.1 Using OMC through CORBA .......................... 33
  5.2 Using the CORBA IOR file .......................... 34
  5.3 The OMC log file ................................. 34

6 Using OMPython ........................................ 35
  6.1 Installing OMPython ................................. 35
  6.2 Executing OMPython ................................. 35
     6.2.1 Test ........................................... 36
     6.2.2 Library ....................................... 36

7 Measurements .......................................... 39

8 Related work ........................................... 43
  8.1 PySimulator ........................................ 43
  8.2 PySCeS ............................................. 44
  8.3 SimPy .............................................. 44

9 Conclusion ............................................. 45

10 Future Work ........................................... 47

Bibliography ............................................. 49

A Setting-up omniORB ................................. 51
  A.1 Steps ............................................. 51

B Setting-up omniORBpy .............................. 54
  B.1 Steps ............................................. 54

C Testsuite Source ...................................... 55
List of Figures

2.1 Schematic example of Modelica ............................................. 8
3.1 OpenModelica Client-Server architecture ................................. 12
3.2 CORBA ORB architecture ...................................................... 13
4.1 Functions of the OMPython API ............................................. 20
4.2 Get/Set helper function ......................................................... 23
6.1 OMPython executing OpenModelica commands in the Test mode 36
6.2 Creating and saving a simple HelloWorld model file using OMPython 37
6.3 OMPython executing the Python script shown above ................... 38
7.1 Measuring response times of simulations for the BouncingBall model 40
7.2 Measuring response time for Simulations in OMPython ................ 41
7.3 Measuring response times of some OpenModelica commands in OMPython 42
8.1 PySimulator, using OpenModelica to simulate and plot a Modelica model .................................................................. 44
List of Tables

7.1 Response time comparisons for loading, simulating and plotting variables using OMPython .............................................. 40
7.2 Measuring response times of some OpenModelica commands in OMPython 41
Listings

3.1  omc__communication.idl ................................. 16
4.1  Code to illustrate the Get/Set helper function .............. 22
6.1  Simulating a simple Modelica model ........................ 37
List of Acronyms

**OMC** OpenModelica Compiler - The compiler that OpenModelica features

**CORBA** Common Object Request Broker Architecture - uses an interface definition file to implement distributed applications

**IDL** Interface Definition Language - is a language independent interface through which remote objects can communicate

**IOR** Interoperable Object Reference - is the contact details that a client application uses to communicate with a CORBA object

**ORB** Object Request Broker - is a piece of middleware software that allows programs to make program calls

**API** Application Programming Interface - is a specification intended to be used as an interface by software components to communicate with each other

**LSODA** Ordinary Differential Equation Solver for Stiff or Non-Stiff System

**ODE** Ordinary Differential Equation

**DAE** Differential-Algebraic Equation

**DLR** Deutschen Zentrums für Luft- und Raumfahrt - German Aerospace Center

**FMU** Functional Mockup Interface

**GUI** Graphical User Interface
Chapter 1

Introduction

1.1 Background

Modeling and Simulation is a discipline, it is also very much a form of an art. It is possible to learn about riding a bicycle from reading a book but to really learn to ride a bicycle one must become practically engaged with a bicycle. Modeling and Simulation follows much the same reality. It’s very much a learn as you go process. From the interaction of the developer and the models emerges an understanding of what makes sense and what doesn’t [1].

Modelica is a non-proprietary, object-oriented, equation based language to conveniently model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents [2].

OpenModelica is an open-source Modelica-based modeling and simulation environment intended for industrial and academic usage. Its long-term development is supported by a non-profit organization the Open Source Modelica Consortium (OSMC)[4].

Python is a remarkably powerful dynamic programming language that is used in a wide variety of application domains. Python is often compared to Tcl, Perl, Ruby, Scheme or Java. Some of its key distinguishing features include:

- very clear, readable syntax
- strong introspection capabilities
- intuitive object orientation
• natural expression of procedural code
• full modularity, supporting hierarchical packages
• exception-based error handling
• very high level dynamic data types
• extensive standard libraries and third party modules for virtually every task
• extensions and modules easily written in C, C++ (or Java for Jython, or .NET languages for IronPython)
• embeddable within applications as a scripting interface [5].

omniORB is a robust high performance CORBA ORB for C++ and Python. It is freely available under the terms of the GNU Lesser General Public License (for the libraries), and GNU General Public License (for the tools). omniORB is largely CORBA 2.6 compliant. omniORB is a CORBA Object Request Broker (ORB) for C++ and Python [6].

The implementation of the Python interface to OpenModelica is made possible by implementing a CORBA communication to the OpenModelica Compiler and the Python interpreter. The Python interpreter acts as the front-end of this interface and it is used to send-receive queries and results to and from the OpenModelica compiler. This enables the Python interface to be used as an interactive session handler for Modelica scripting and command evaluation.

1.2 Target Readers

The main group of readers that this document focuses on are Python users who would like to get benefited from the robust simulation, compilation and scripting abilities of a non-proprietary object-oriented, equation based modeling language such as Modelica. However, OpenModelica developers are also benefited by using this document to get a deeper understanding of how this interface is developed and what could be added to this interface in the future to further enhance its support to its users.

1.3 Goals

• Interactive session handling, parsing, interpretation of commands and Modelica expressions for evaluation, simulation, plotting, etc.
1.4 Method

- Interface to the latest OpenModelica API calls.
- Optimized parser results that give control over every element of the output.
- Helper functions to allow manipulation on Nested dictionaries.
- Easy access to the library and testing of OpenModelica commands.

1.4 Method

Since the focus of the project was to develop a working implementation of a non-existent feature to OpenModelica, some amount of reading on the existing architecture, specifications and tools had to be done. To gain the basic understanding of the OpenModelica environment and Modelica language, I used the exercises and contents provided in the "OpenModelica OMSHELL and OMNotebook Introduction and Exercises" course designed by Peter Fritzson, Adrian Pop and Peter Aronsson. The OpenModelica User Guide and System Documentation provided a great deal of information in studying the existing components of the system. Professor Peter Fritzson’s book [8] gave me an introduction to the Modelica language, the exercises given in the book were productive and easy to understand.

The first working implementations of the OpenModelica - Python interface were built as external Python applications that called upon OpenModelica’s services and did not feature the Parser module. However, subsequent module developments and integration with test suites helped to develop more advanced and complete modules. The development was done using Windows XP and Windows 7 versions, Python 2.7 and IDLE (Python editor). Integration testing phases were conducted on both Windows and Linux based machines.

1.5 Overview of this thesis report

This thesis report starts by introducing the Modelica language and OpenModelica. It then elaborates upon the tools that were used to build the application programming interface. Later, the different parts of the implementation are detailed and finally the project is assessed.
Chapter 2

Background

2.1 Modelica

Modelica is a freely available, object-oriented language for modeling of large, complex and heterogeneous physical systems, designed to support effective library development and model exchange. It is a modern language built on acausal modeling with mathematical equations and object oriented constructs to facilitate reuse of modeling knowledge. It is suited for multi-domain modeling, for example, mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, distribution of electric power [2].

Models in Modelica are mathematically described by differential, algebraic and discrete equations. No particular variable needs to be solved for manually. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, specialized algorithms can be utilized to enable efficient handling of large models having more than hundred thousand equations. It is suited (and used) for hardware-in-the-loop simulations and for embedded control systems [3].

From a user's point of view, models are described by schematics, also called object diagrams. Figure 2.1 shows an example Schematic from Modelica.
A schematic consists of connected components, like a resistor, or a hydraulic cylinder. A component has "connectors" (often also called "ports") that describe the interaction possibilities, e.g., an electrical pin, a mechanical flange, or an input signal. By drawing connection lines between connectors a physical system or block diagram model is constructed. Internally a component is defined by another schematic or on "bottom" level, by an equation based description of the model in Modelica syntax.

The Modelica language is a textual description to define all parts of a model and to structure model components in libraries, called packages. An appropriate Modelica simulation environment is needed to graphically edit and browse a Modelica model (by interpreting the information defining a Modelica model) and to perform model simulations and other analysis [2].

2.2 OpenModelica

OpenModelica is an open-source Modelica-based modeling and simulation environment intended for industrial and academic usage. The purpose of OpenModelica is to create a comprehensive Open Source Modelica modeling, compilation and simulation environment based on free software distributed in binary and source code form for research, teaching, and industrial use.

The primary language that OpenModelica operates on is Modelica; however,
it can compile C, Fortran and UML code also. To completely support the Modelica environment, OpenModelica has several components. For the purpose of this thesis, the OpenModelica compiler/interpreter subsystem (OMC) is the main component. The OMC is invoked using the CORBA client-server interface to implement the Python interface to OpenModelica due to its simplicity and scalability of the approach.
Chapter 3

Tools and Communication

3.1 OpenModelica Compiler - OMC

OMC is a Modelica compiler, translating Modelica to C code, with a symbol table containing definitions of classes, functions, and variables. Such definitions can be predefined, user-defined, or obtained from libraries. The compiler also includes a Modelica interpreter for interactive usage and constant expression evaluation. The subsystem also includes facilities for building simulation executables linked with selected numerical ODE or DAE solvers [4].

The OpenModelica Compiler/Interpreter subsystem (OMC) can be invoked in two ways,

- As a whole program, called at the operating-system level, e.g. as a command.

- As a server, called via a CORBA client-server interface from client applications.

The second method, using the CORBA client-server interface is used to implement the API due to the simplicity and scalability of the approach. The OpenModelica client-server architecture is schematically depicted in Figure 3.1 and highlights the integrated Python interface module in the architecture with the blue outline.
The CORBA interface is used to send-receive commands and results from the OMC via the Python interpreter. Since the output from the CORBA interface is usually a stream of strings, the output needs to be parsed in-order to be usable in the Python environment.

The OMC can be used in several modes. The interactive mode can be used to execute or process scripts (mos files) or to invoke sub-processes that can be communicated using CORBA. This mode is usually used to manipulate models, run simulations, etc, but its usage in this thesis focuses on its functions. When calling Modelica functions in the interactive mode, the OMC translates the function into C-code, it is then compiled and executed.

The SCode module simplifies the Absyn representation, public components are collected together, protected ones together, etc. The Interactive module serves the untyped API, updates, searches, and keeps the abstract syntax representation. An environment structure is not kept/cached, but is built by Inst at each call. Call Inst for more exact instantiation lookup in certain cases. The whole Absyn AST is converted into Scode when something is compiled, e.g. converting the whole standard library if something.

Commands or Modelica expressions are sent as text from the clients via the Corba interface, parsed, and divided into two groups by the main program:

- All kinds of declarations of classes, types, functions, constants, etc., as well as equations and assignment statements. Moreover, function calls to the untyped API also belong to this group a function name is checked if it belongs to the API names. The Interactive module handles this group of
3.2 CORBA

The following section aims to give the readers an idea about the standard CORBA architecture that is used in the implementation, taken from the chapter on Overview of CORBA [9]. Figure 3.2 illustrates the primary components in the CORBA ORB architecture. Descriptions of these components are available below the figure.

- **Object** – This is a CORBA programming entity that consists of an identity, an interface, and an implementation, which is known as a Servant.

- **Servant** – This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.

---

**Figure 3.2: CORBA ORB architecture**

- Object – This is a CORBA programming entity that consists of an identity, an interface, and an implementation, which is known as a Servant.

- Servant – This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.
• Client – This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., obj->op(args). The remaining components in Figure ?? help to support this level of transparency.

• Object Request Broker (ORB) – The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.

• ORB Interface – An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

• CORBA IDL stubs and skeletons – CORBA IDL stubs and skeletons serve as the “glue” between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

• Dynamic Invocation Interface (DII) – This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking deferred synchronous (separate send and receive operations) and oneway (send-only) calls.
• Dynamic Skeleton Interface (DSI) – This is the server side’s analogue to the client side’s DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

• Object Adapter – This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).

3.3 OmniORB

OmniORB is an Object Request Broker (ORB) that implements the 2.6 specification of the Common Object Request Broker Architecture (CORBA). It is a robust high performance CORBA ORB for C++ and Python. It is freely available under the terms of the GNU Lesser General Public License [6]. The CORBA interface needed for the OMC communication is developed using omniORB.

The OmniORB IDL (Interface Definition Language) compiler uses OpenModelica’s IDL file to create the Stub files.

3.3.1 Interface Definition Language (IDL) file

CORBA utilizes the Interface Definition Language (IDL) to specify the interface for the distributed objects. In-order to enable distribution in the CORBA architecture, the interfaces need to be defined such that they are understandable by both the client and the server objects. The interface file should specify the operations, their type and the input/output parameters. The interface should be specified on a programming language that can enforce this architecture, object orientation and exception handling which form the basis for the distributed system.

Every CORBA object has a type name, which is the same as the name of the interface that is assigned in its IDL declaration. In order to communicate with the OpenModelica server, OpenModelica’s IDL file omc_ communication.idl is used. Listing 3.1 shows the omc_ communication.idl file.
Listing 3.1: omc_ communication.idl

// As simple as can be omc communication, sending and
// receiving of strings

interface OmcCommunication {
    string sendExpression( in string expr);
    string sendClass( in string model);
};

The interface needs only one function namely sendExpression which takes one
input string parameter and returns a string to serve the purpose of sending string
commands to the OMC via the CORBA interface.

3.3.2 Stubs

Stubs contain the language based declarations for the methods in the IDL files and
provide the language specific mappings for the interfacing language. OmniORB’s
IDL compiler, omniidl, can be used to create the stubs for the IDL file.

3.3.3 Object references and Servants

CORBA objects can be accessed by object references. The implementation of a
CORBA object is called as a Servant. Object references and servants should not
be confused as they are separate entities. The client code deals with the object
references whereas the object implementation code must deal with both object
references and servants. The Python interface to OpenModelica will represent the
object implementation code that will communicate with the OMC via the CORBA
Server.

3.4 OmniORBpy

OmniORBpy is an Object Request Broker (ORB) that implements the CORBA
2.6 Python mapping. It works in conjunction with omniORB for C++, version
4.1. Using this package allows users to have omniORBpy with omniORB binding
using Python, this can be used to create the interface and object implementation
files in Python [7].

For example, the Stubs written in Python can be generated using the command,

omniidl bpython Wbstubs=stubs OmcCommunication.idl
This creates the stub files in the directory stubs.

3.5 The Python Interpreter

The implementation aims at developing a Python API for the OpenModelica environment, therefore the Python interpreter is a necessary tool in this development project. One of the goals of the API is to function as both an application and a library that can be imported by other applications.
Chapter 4

Design

This chapter details about the design features and implementation modules of the Python interface to OpenModelica, the API is called OMPython.

4.1 Requirements

The requirements can be summarized as follows,

- Setup the CORBA communication environment.
- Interface OpenModelica with Python to create an interactive session.
- Communicate with the OMC from the Python API.
- Perform the Modelica based operations on the OMC.
- Obtain the results via the Python API.
- Format the results to be usable by the Python API.
- Enable manipulation on the results.

4.2 Design

OMPython - the Python interface to OpenModelica is implemented using the CORBA interface and hence plays the role of a Client to the OMC. The OMPython API has two modules namely, OMPython and OMParser.

The following Figure 4.1 gives an overview of the OMPython API, shows its submodules and gives an idea about their functions.
4.2.1 OMPython

The OMPython module is the main module that is responsible for establishing the OMC communication. The OMPython module is designed to perform the following functions:

- Initialize and maintain the interactive CORBA communication
- Send commands to the OMC via the CORBA interface
- Receive the CORBA string results
- Use the OMParse module to format the results
- Typecast the results to their appropriate forms
- Define the Get and Set functions for nested dictionaries
- Display the results

Interactive Session handler

Each time the OMPython module is instantiated, it creates an interactive session between the user and the OMC. The OMPython module’s session handler rou-
tine uses the CORBA Interoperable Object Reference (IOR) file to establish and maintain the user’s session activities including the log files via the standard output and standard error streams. The user can close the connection by issuing the quit() command to the OMC. Once the connection close command is requested, OMC removes the temporary IOR file from the working directory of the user’s machine also. The log files allow users to take a look at their session, perform some debugging and also provide the ability to traceback the commands used.

**CORBA Communication**

The Client-Server architecture of the CORBA mechanism is used to enable the interaction of the OMPython module with the OMC. The module implements the Client side of the architecture.

**Modes of Operation**

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

- Test
- Library

The Test mode allows users to test or learn about the OMPython API while the Library mode allows users to use the results of OMPython.

**Using the interface definition**

The interface definition specifies the operations to be performed, including the input and output parameters along with their types, allowing the client and the server to encode and decode values for their travel over the network. The important 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 [10].

**Get/Set helper functions**

OMC produces complicated result strings such as Component Annotations, Diagram Annotations, etc. To make these results accessible and usable, 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. Listing 4.1 is a sample program that uses the Get and Set helper functions to manipulate on the Simulation results of a model.

Listing 4.1: Code to illustrate the Get/Set helper function

```python
import OMPython

OMPython.execute("loadModel(Modelica)")
result = OMPython.execute("getDiagramAnnotation(Modelica.Electrical.Analog.Basic.Resistor)")
print result

Line1_values = OMPython.get(result, "SET2.Elements.Line1.Properties.Values")
print "\nValues of Line1 are: \n%s" %Line1_values

result = OMPython.set(result, "SET2.Elements.Line1.Properties.Values", [0, 1, 2, 3])
Line1_values = OMPython.get(result, "SET2.Elements.Line1.Properties.Values")
print "\nThe new values of Line1 are: \n%s" %Line1_values

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

Figure 4.2 shows the output on the console that shows how the user can get and set the values of any nested dictionary data type.
4.2 Design

**Figure 4.2: Get/Set helper function**

**Typcasting**

Since Python is a dynamically typed language, 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.

**Imports the OMParser module**

Although the results of the OMC are available 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, OMParser, to generate appropriate data structures for the OMC retrieved results.

**4.2.2 OMParser**

The OMParser module is the submodule, OMPython’s Parser module, that is used by the OMPython module to construct datatypes from the CORBA string outputs and generate strictly typed datatypes that can be used in Python. The OMParser module plays a significant role of the API because the OMC output from the CORBA interface contains string results from both the typed and untyped API of
OpenModelica. This makes grammar based parsers unsuitable for the task since only the typed API can be handled with such parsers. The OMParse module is designed to perform the following operations:

- Analyze the CORBA strings for categorical data.
- Group each category under a category name
- Typecast the data within these categories
- Build suitable data structures to hold these data so that the results can be easily accessed and used.
- Return the results to the OMPython module.

### 4.2.3 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 CORBA string outputs. The strings without the 'curly brackets' are simply typecasted into the appropriate datatypes, removing the escape sequences and displayed as the result.

---

**Example 4.1**

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

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

---

**Example 4.2**

```plaintext
»getModelicaPath()
"C:/OpenModelica.8.0/lib/omlibrary"
```
4.2 Design

The Dictionary data type in Python

Dictionaries are a useful data type in Python which are indexed by Keys unlike the 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 values associated with a key and retrieve the value using the key. This provides us with the flexibility of creating keys at runtime and access the values using these keys later. All data within the dictionary are stored inside 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 due to creating Dynamic variables, the following notations are introduced which are used within dictionaries,

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

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 in an increasing index starting from 1. This feature was planned to eliminate the need for dynamic variable creation and duplicate Keys. The SET belongs within the dictionary, result.

Example 4.3

OMShell
›strtok("abcabcdef","b")
"a","c","def"
**OMPython**

```python
»strtok("abcbdef","b")
'SET1': 'Values': ['"a","c","def"]
```

In the example shown above, the command `strtok` is used to tokenize the string "abcbdef" at every occurrence of b and produces a SET with values 'a', 'c', 'def'.

**Set**

A set is used to group all data within the 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 a SET, the inner brackets are the Set within the SET.

**Subset**

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

---

**Example 4.4**

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

---

**Element**

Elements are the data which are grouped within a pair of Parenthesis ( ). As observed from the results string, elements have an element name that describes the data within them, so elements can be associated and grouped by their names. Also, 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, Values and Subsets within them. However, they are in turn enclosed within the SET.

---

**Example 4.5**
In the above example, the result contains, a SET with a Element named rec1 which has Properties which are Results 4.2.3 of the element.

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 an Name = Value relationship. So, they form the sub dictionary called Results which are placed within the Element (for example). Then these values can be related by the key : value pair relationship.
```java
»getClassAttributes(test)
{'SET1': {'Elements': {'rec1': {'Properties': {'Results':
{'comment': '""', 'restriction': 'MODEL',
'startLine': 1, 'partial': False, 'name': '"test"',
'encapsulated': False, 'startColumn': 1, 'readonly':
'"writable"', 'endColumn': 19, 'file': '"<interactive>"',
'endLine': 1, 'final': False}}}}}}
```

### Values

Data within SETs, Sets, Elements and Subsets that are not assignments and are separated by commas are grouped together into the list called Values. The Values list may also be empty.

---

**Example 4.7**

*OMShell*

```bash
G.n, Nr.P, ""
```

*OMPython*

```python
{'SET1': {'Set1': ['G.n', 'Nr.P', '""]}}
```

---

### 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 dictionary contains two sub dictionaries namely, SimulationResults and SimulationOptions.

---

**Example 4.8**

*OMShell*

```bash
» simulate(BouncingBall)
```
record SimulationResult resultFile = "C:/Users/ganan642/BouncingBall_res.mat",
simulationOptions = "startTime=0.0, stopTime = 1.0,
numberOfIntervals = 500, tolerance = 1e-006, method= 'dassl', fileNamePrefix = 'BouncingBall', storeInTemp = false,
noClean = false, options = '', outputFormat = 'mat',
variableFilter = '.*', measureTime = false, cflags = '',
simflags = '', messages = '',
timeFrontend = 0.89961754408252,
timeBackend = 0.0204728440900263,
timeSimCode = 0.00981912741537176,
timeTemplates = 0.0227772035485442,
timeCompile = 3.48699707624289,
timeSimulation = 0.153226430456349,
timeTotal = 4.59302417344674
end SimulationResult;

OMPython

»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': 0.013996795849597, 'timeTotal': 6.3452533928534,
'resultFile': '"C:/Users/ganan642/ BouncingBall_res.mat"'}
The Record types

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

--- Example 4.9 ---

*OMShell*

```omshell
» checkSettings()
record OpenModelica.Scripting.CheckSettingsResult
OPENMODELICAHOME = "C:/OpenModelica1.8.1/",
OPENMODELICALIBRARY="C:/OpenModelica1.8.1/lib/omlibrary", OMC_PATH="C:/OpenModelica1.8.1//bin/omc.exe",
OMC_FOUND = true, MODELICAUSERCFLAGS = ",", WORKING_DIRECTORY = "C:/Users/ganan642/AppData/Local/Temp/OpenModelica",
CREATE_FILE_WORKS=false,REMOVE_FILE_WORKS = true,
OS ="Windows_NT",SYSTEM_INFO = ",",RTLIBS = " -static-libgcc -luuid -lole32 -lws2_32", C_COMPILER = "gcc",
C_COMPILER_RESPONDING=false,CONFIGURE_CMDLINE = "Manually created Makefiles for OMDev"
end OpenModelica.Scripting.CheckSettingsResult;
```

*OMPython*

```ompython
» checkSettings()
{'RecordResults': {'RTLIBS': ' -static-libgcc -luuid -lole32 -lws2_32', 'OMC_FOUND': 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_COMPILER': '"gcc"'}

Chapter 5

Implementation

The OMPython API is implemented using the ClientServer architecture of CORBA to communicate to the OMC. OMPython acts as the Client that requests the services from 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 perform its tasks.

5.1 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 $OPENMODELICA HOME/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 [10].

5.2 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. This IOR file is the session information of the User, when the user sends the `quit()` command to OMC via OMPython, the CORBA connection is closed and the IOR file is destroyed and OMPython stops executing.

5.3 The OMC log file

The OMPython module is designed to maintain a log file for the standard output and error logs that are written to the CORBA process. These files are created when the OMC is invoked and the IOR file is written to the disk. This feature helps Users to trace back their operations and possibly improve some solutions, it also helps the developers to debug the application.
Chapter 6

Using OMPython

This chapter describes how OMPython can be used and also demonstrates how a simple Modelica model can be created and used from within the Python interpreter. It also presents the two modes of operation that the API supports.

6.1 Installing OMPython

The two minimum 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 [10].

6.2 Executing OMPython

The API can be used in two modes, Test and Library, each designed for specific purposes.
6.2.1 Test

Like any new tool, it is important to give its users the freedom to easily learn and 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 iteratively 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 6.1 a few operations are presented from the Python terminal.

![Figure 6.1: OMPython executing OpenModelica commands in the Test mode](image)

Creating new models in the text based Python terminal is rather straightforward using OMPython. Figure 6.2 illustrates this and shows how a model can be created and saved with a simple command.

6.2.2 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,
6.2 Executing OMPython

Figure 6.2: Creating and saving a simple HelloWorld model file using OMPython 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.

Listing 6.1 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 6.3 depicts the output of the program generated by OMPython on a standard Python terminal [10].

Listing 6.1: Simulating a simple Modelica model

```python
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()")

Figure 6.3: OMPython executing the Python script shown above
Chapter 7

Measurements

In this section, some performance measurements from the OMPython API are presented.

The measurements shown are based on the response times obtained from 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. The measurements give an idea about the overhead that the Python interpreter has over processing the OMC results using its own parser in addition to the overhead of the CORBA connection.

The Figure 7.1 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 7.1 shows the statistics of the various time values collected from five unique runs of the simple simulation script using the OMPython API. The time is measured in Seconds.
Figure 7.1: Measuring response times of simulations for the BouncingBall model

<table>
<thead>
<tr>
<th>Command</th>
<th>Average response time (s)</th>
<th>Average unparsed response time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadFile('c:/OpenModelica1.8.1/models/BouncingBall.mo')</td>
<td>0.09223065</td>
<td>0.0421344389</td>
</tr>
<tr>
<td>simulate(BouncingBall)</td>
<td>2.60921512</td>
<td>1.8922307169</td>
</tr>
<tr>
<td>plot(h)</td>
<td>0.03251472</td>
<td>0.0183359414</td>
</tr>
</tbody>
</table>

Table 7.1

Figure 7.2 illustrates the overhead between the average outputs and the unparsed outputs. Similarly, Table 7.2 details about the time measurements obtained by executing a few OpenModelica commands over five different executions of the OMPython API.
Figure 7.2: Measuring response time for Simulations in OMPython

<table>
<thead>
<tr>
<th>Command</th>
<th>Average response time (s)</th>
<th>Average unparsed response time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>getVersion()</td>
<td>0.0680588293</td>
<td>0.0590995445</td>
</tr>
<tr>
<td>loadModel(Modelica)</td>
<td>5.971103887</td>
<td>4.4708573210</td>
</tr>
<tr>
<td>getElementsInfo(Modelica.Electrical.Analog.Basic.Resistor)</td>
<td>0.0264064349</td>
<td>0.0190346404</td>
</tr>
<tr>
<td>getClassNames()</td>
<td>0.3907942649</td>
<td>0.2707218157</td>
</tr>
<tr>
<td>getTempDirectoryPath()</td>
<td>0.0244359882</td>
<td>0.0193691690</td>
</tr>
<tr>
<td>getSettings()</td>
<td>0.0327650196</td>
<td>0.0234227783</td>
</tr>
</tbody>
</table>

Table 7.2

Figure 7.3 shows the timing values obtained from some of the OpenModelica commands and compares the mean and the unparsed response times in a graph.
Figure 7.3: Measuring response times of some OpenModelica commands in OMPython
Chapter 8

Related work

8.1 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, FMUs and OpenModelica. It also provides analysis tools for some applications particularly in physics and engineering.

The OpenModelica plugin for PySimulator uses OMPython to implement the features needed for its simulation and analysis package. The OpenModelica plugin that is developed for PySimulator defines routines for loading, simulating, adjusting simulation parameters, etc [10].

Figure 8.1 illustrates a screen from the PySimulator package that uses OMPython to simulate a Modelica model and plot some candidate variables from the simulation.
Figure 8.1: PySimulator, using OpenModelica to simulate and plot a Modelica model

8.2 PySCeS

The Python Simulator for Cellular Systems. It uses the model description language to define its models. Supports solvers like LSODA, provides 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 [12].

8.3 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 [13].
Chapter 9

Conclusion

OMPython is a versatile Python library for OpenModelica. It invites 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 Modelica libraries that can assist in performing complex simulation and analysis tasks.

The OMPython API places minimal requirements on the user while offering an industry viable standard modeling and simulation environment.
Chapter 10

Future Work

Some future work that can be done to enrich the usage of the OMPython API are suggested here.

The API is not a standalone package for Python, it is dependent on the OpenModelica installation. The API can therefore be expanded to implement most if not all features of OpenModelica. User interfaces can be easily built on top of OMPython to implement additional graphical usability.

Further interesting efforts can be made if the OpenModelica API can be designed to expose its command definitions as interface definitions in the omc_communication.idl file.
Bibliography


Appendix A

Setting-up omniORB

This section contains information on installing, building and using omniORB4.1.6 on Win32 platforms.

We need to compile omniORB first then we can compile omniORBpy against that.

A.1 Steps

1. First, we need to create a directory to put our source files into. For example,
   \texttt{mkdir C:\Work}

2. The omniORB source files can be downloaded from SourceForge at the following link,
   \url{http://sourceforge.net/projects/omniorb/files/} Download the latest omniORB (omniORB-4.1.6.tar.bz2) and omniORBpy (omniORBpy-3.6.tar.bz2)

3. We also need the OPENSSL implementation which can be downloaded from:
   \url{http://code.google.com/p/openssl-for-windows/downloads/list} The latest one is openssl-0.9.8k\_WIN32 (at the time of writing).

4. Unpack them all into \texttt{C:\Work} so that we have three folders
   \texttt{C:\Work\omniORB-4.1.6},
   \texttt{C:\Work\omniORBpy-3.6} and
   \texttt{C:\Work\openssl-0.9.8k\_WIN32}.
5. omniORBpy-3.6 needs to be in a special place so move it to
   C:\Work\omniORB-4.1.6\src\lib\omniORBpy
   (i.e. rename the omniORBpy-3.6 directory to omniORBpy after copying it
   to the lib directory).

6. The build process needs Cygwin but its make just hangs during the build,
   so the cut-down version of cygwin which has just what is needed to build
   omniORB, called gnuwin32 can be downloaded from,
   http://omniorb.sourceforge.net/att_packages/
   gnu-win32-lite.zip
   Download it and unpack it to C:\gnuwin32.

7. Now that we have all the software downloaded, we can start configuring the
   build. Edit C:\Work\omniORB-4.1.6\config\config.mk and uncom-
   ment the platform you are using. In my case it is x86_win32_vs_10

8. Next, load the platform specific file from
   C:\Work\omniORB-4.1.6\mk\platforms\x86_win32_vs_10.mk. As
   we will be building in a Cygwin environment, we need to specify the paths
   in Cygwin format. So change the variable PYTHON to point to the correct
   location of the Python executable.
   i.e; PYTHON = /cygdrive/c/Python27/python

9. OpenSSL transport support is disabled by default. Since we will need the
   Open SSL libraries, uncomment the OPEN_SSL_ROOT location. The
   path has to be a relative path. Set the ssl root to: OPEN_SSL_ROOT =
   ../../../../../../openssl-0.9.8k_WIN32 which is a relative ref-
   erence to C:\Work\openssl-0.9.8k_WIN32 from
   C:\Work\omniORB-4.1.6\src\lib\omniORB\orbcore\ssl.

10. Now that we are configured we can actually start the build. Open a command
    window and add C:\gnuwin32 to the start of your path:
    set PATH=c:\gnuwin32\bin;%PATH%

11. Run the visual studio batch file to set up the build environment: C:\Program
    Files\Microsoft Visual Studio 10.0\VC\vcvarsall.bat

12. Now you have to run a small script *once* to setup in your registry the
    necessary 'mount' points. Basically, it tells the gnuwin32 runtime how
    to translate a path such as /bin/sh to the real path name Win32 (e.g.
C:\gnuwin32\bin\sh.exe) understands. Just execute this command in a command prompt window:

C:\gnuwin32\bin\checkmounts C:\gnuwin32

If all goes well, this is what you see:

C:\> C:\gnuwin32\bin\checkmounts C:\gnuwin32
no /bin/sh.exe, mounting c:\gnuwin32\bin as /bin
Completed successfully.

13. Change to the omniORB source directory and run the build:

cd C:\Work\omniORB-4.1.6\src make export The build of omniORB will take about 30 minutes. omniORB has a large number of parameters that can be configured. The files sample.cfg and sample.reg contain sample configurations and registry entries. Open the sample.reg file to register omniORB settings. Next up is to compile the omniORBpy.
Appendix B

Setting-up omniORBpy

Once omniORB is setup, we can proceed with setting-up omniORBpy.

B.1 Steps

Change to the omniORBpy source directory and run the build from there:

\texttt{cd C:\Work\omniORB-4.1.6\src\lib\omniORBpy}
\texttt{make export}

The build of omniORBpy will take about 3 minutes.
import OMPython

cmds = ["loadModel(Modelica)",
"appendEnvironmentVar (\"var\",\"C:/\")",
"basename(\"C:/\")",
"cd("")",
"checkAllModelsRecursive()",
"checkSettings()",
"dirname(\"C:/\")",
"model test end test;",
"getAlgorithmCount(test)",
"getAlgorithmItemsCount(test)",
"getAnnotationCount(test)",
"getAnnotationVersion()",
"getClassComment(test)",
"getClassesInModelicaPath()",
"getCompileCommand()",
"getEnvironmentVar(\"var\")",
"getInstallationDirectoryPath()",
"getLanguageStandard()",
"getMessagesString()",
"getMessagesStringInternal()",
"getModelicaPath()",
"getSettings()",
"getTempDirectoryPath()",
"getVectorizationLimit()",
"help()",
"loadFileInteractive("C:/OpenModelica1.8.1/testmodels/Circle.mo")",
"instantiateModel(Circle)",
"getNthAlgorithm(Circle,1)",
"list(Modelica.Math.sin,interfaceOnly=true)",
"listVariables()",
"loadFileInteractive("C:/OpenModelica1.8.1/testmodels/BouncingBall.mo")",
"loadFileInteractiveQualified("C:/OpenModelica1.8.1/testmodels/
BouncingBall.mo")",
"parseFile("C:/OpenModelica1.8.1/testmodels/BouncingBall.mo")",
"parseString("data",BouncingBall)"
"simulate(BouncingBall)",
"readFile("C:/OpenModelica1.8.1/testmodels/BouncingBall.mo")",
"readFileNoNumeric("C:/OpenModelica1.8.1/testmodels/BouncingBall.mo")",
"readFilePostprocessLineDirective("C:/OpenModelica1.8.1/testmodels/
BouncingBall.mo")",
"readFileShowLineNumbers("C:/OpenModelica1.8.1/testmodels/
BouncingBall.mo")",
"readSimulationResult("BouncingBall.res.mat",h,0)",
"runScript("C:/OpenModelica1.8.1/testmodels/simBouncingBall.mos")",
"basePlotFunction("BouncingBall",F,linear,", Plotted mess
",true,true,false,false,"time",,false,0.0,0.0,0.0,0.0)",
"codeToString(BouncingBall)",
"compareSimulationResults("C:/OpenModelica1.8.1/testmodels/
BouncingBall.mo","C:/OpenModelica1.8.1/Circle.mo",
"C:/OpenModelica1.8.1/BouncingBall.log",1e-006,1e-006,0.0,0.0,0.0,0.0)",
"loadFile("C:/OpenModelica1.8.1/testmodels/BouncingBall.mo")",
"dumpXMLDAE(BouncingBall,"flat",false,false,false,",false")",
"getAstAsCorbaString("BouncingBall")",
"getIconAnnotation(Modelica.Electrical.Analog.Basic.Resistor)"
'strtok("abcdef","b")*
'translateGraphics(BouncingBall)*,
'typeOf(h)*,
'val(currentSimulationResult, 0.5)*,
'strictRMLCheck()*
'regex("abc", "a")*
'regex("lorem", ".*\([A-Za-z]*\) \([A-Za-z]*\) ",3,true,false)*
'getSourceFile(BouncingBall)*,
'setClassComment(BouncingBall, "binggg")*
'getClassNames()*,
'list(*)

for i in cmds:
    print » %s\n" %i
answer = OMPython.execute(i)
print answer
OMPython.execute('quit()')
Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare under 25 från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovanbeskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned
when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© Anand Kalaiarasi Ganeson