FMU Export of a memoryless Python-driven Simulator

Release 0.1.0

LBNL - Building Technology and Urban Systems Division

May 15, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION

This user manual explains how to install and use SimulatorToFMU.

SimulatorToFMU is a software package written in Python which allows users to export a memoryless Python-driven simulation program or script as a *Functional Mock-up Unit* (FMU) for model exchange or co-simulation using the *Functional Mock-up Interface* (FMI) standard version 1.0 or 2.0. This FMU can then be imported into a variety of simulation programs that support the import of Functional Mock-up Units.

A memoryless Python-driven simulation program/script is a simulation program which meets following requirements:

- The simulation program/script can be invoked through a Python script.
- The invocation of the simulation program/script is memoryless. That is, the output of the simulation program at any invocation time `t` depends only on the inputs at the time `t`.
- The inputs and the outputs of the simulation program/script must be `real` numbers.

Note: The Python-driven script could invoke scripts written in languages such as MATLAB using the `subprocess` or `os.system()` module of Python or specifically for MATLAB using the MATLAB engine API for Python.

CHAPTER

TWO

INSTALLATION AND CONFIGURATION

This chapter describes how to install, configure, and uninstall SimulatorToFMU.

Software requirements

To export a Simulator as an FMU, SimulatorToFMU needs:

1. Python and following dependencies:
    - jinja2
    - lxml
2. Modelica parser
3. C-Compiler

SimulatorToFMU has been tested with:

- Python 2.7.13
- Python 3.5.0
- Dymola 2017 FD01 (Modelica parser, Windows and Linux)
- OpenModelica 1.11.0 (Modelica parser, Windows)
- Microsoft Visual Studio 10 Professional (C-Compiler)

Note: SimulatorToFMU can use OpenModelica and Dymola to export a Simulator as an FMU. However OpenModelica 1.11.0 (on Windows) and Dymola 2017 FD01 (on Linux) do not copy all required libraries dependencies to the FMU. As a workaround, SimulatorToFMU checks if there are missing libraries dependencies and copies the dependencies to the FMU.

Installation

To install SimulatorToFMU, proceed as follows:

1. Add following folders to your system path:
    - Python installation folder (e.g. `C:\Python35`)
    - Python scripts folder (e.g. `C:\Python35\Scripts`),
    - Dymola executable folder (e.g. `C:\Program Files(x86)\Dymola2017 FD01\bin`)

- OpenModelica executable folder (e.g. `C:\OpenModelica1.11.0-32bit\`)

You can add folders to your system path by performing following steps on Windows 8 or 10:

- In Search, search for and then select: System (Control Panel)
- Click the Advanced system settings link.
- Click Environment Variables. In the section System Variables, find the PATH environment variable and select it. Click Edit.
- In the Edit System Variable (or New System Variable) window, specify the value of the PATH environment variable (e.g. `C:\Python35`, `C:\Python35\Scripts`). Click OK. Close all remaining windows by clicking OK.
- Reopen Command prompt window for your changes to be active.

To check if the variables have been correctly added to the system path, type `python`, `dymola`, or `omc` into a command prompt to see if the right version of Python, Dymola or OpenModelica starts up.

2. To install SimulatorToFMU, run

```
> pip install SimulatorToFMU
```

The installation directory should contain the following subdirectories:

- `bin/` (Python scripts for running unit tests)
- `doc/` (Documentation)
- `fmus/` (FMUs folder)
- `parser/` (Python scripts, Modelica templates and XML validator files)

Uninstallation

To uninstall SimulatorToFMU, run

```
> pip uninstall SimulatorToFMU
```

CHAPTER

THREE

BEST PRACTICE

This section explains to users the best practice in configuring a Simulator XML input file, and implementing the Python wrapper which will interface with the Simulator.

Configuring the Simulator XML input file

To export Simulator as an FMU, the user needs to write an XML file which contains the list of inputs, outputs and parameters of the FMU. The XML snippet below shows how a user has to write such an input file. A template named `SimulatorModeldescritpion.xml` which shows such a file is provided in the `parser/utilities` installation folder of SimulatorToFMU. This template should be adapted to create new XML input file.

The following snippet shows an input file where the user defines 1 input and 1 output variable.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <SimulatorModelDescription
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    fmiVersion="2.0"
5    modelName="Simulator"
6    description="Input data for a Simulator FMU"
7    generationTool="SimulatorToFMU">
8    <ModelVariables>
9      <ScalarVariable
10       name="v"
11       description="Voltage"
12       causality="input">
13       <Real
14         unit="V"
15         start="0.0"/>
16     </ScalarVariable>
17     <ScalarVariable
18       name="i"
19       description="Current"
20       causality="output">
21       <Real
22         unit="A"/>
23     </ScalarVariable>
24   </ModelVariables>
25 </SimulatorModelDescription>
```

To create such an input file, the user needs to specify the name of the FMU (Line 5). This is the `modelName` which should be unique. The user then needs to define the inputs and outputs of the FMUs. This is done by adding `ScalarVariable` into the list of `ModelVariables`.

To parametrize the `ScalarVariable` as an input variable, the user needs to

- define the name of the variable (Line 10),
- give a brief description of the variable (Line 11)
- give the causality of the variable (`input` for inputs, `output` for outputs) (Line 12)
- define the type of variable (Currently only `Real` variables are supported) (Line 13)
- give the unit of the variable (Currently only valid *Modelica* units are supported) (Line 14)
- give a start value for the input variable (This is optional) (Line 15)

To parametrize the `ScalarVariable` as an output variable, the user needs to

- define the name of the variable (Line 18),
- give a brief description of the variable (Line 19)
- give the causality of the variable (`input` for inputs, `output` for outputs) (Line 20)
- define the type of variable (Currently only `Real` variables are supported) (Line 21)
- give the unit of the variable (Currently only valid *Modelica* units are supported) (Line 22)

Configuring the Python Wrapper Simulator

To export Simulator as an FMU, the user needs to write a Python wrapper which will interface with the Simulator. The wrapper will be embedded in the FMU when the Simulator is exported and used at runtime on the target machine.

The user needs to extend the Python wrapper provided in `parser/utilities/simulator_wrapper.py` and implements the function `exchange`.

The following snippet shows the Simulator function.

```
1  # Main Python function to be modified to interface with the main simulator.
2
3  def exchange(configuration_file, time, input_names,
4               input_values, output_names, write_results):
5      """
6      Return  a list of output values from the Python-based Simulator.
7      The order of the output values must match the order of the output names.
8
9      :param configuration_file (String): Model configuration name
10     :param time (Float): Simulation time
11     :param input_names (Strings): Input names
12     :param input_values (Floats): Input values (same length as input_names)
13     :param output_names (Strings): Output names
14     :param write_results (Float): Store results to file (1 to store, 0 else)
15
16     Example:
17         >>> configuration_file = 'config.json'
18         >>> time = 0
19         >>> input_names = ['v']
20         >>> input_values = [220.0]
21         >>> output_names = ['i']
22         >>> write_results = 0
23         >>> output_values = simulator(configuration_file, time, input_names,
24                           input_values, output_names, write_results)
25     """
26
27     ######################################################################
```

```
    # EDIT AND INCLUDE CUSTOM CODE FOR TARGET SIMULATOR
    # Include body of the function used to compute the output values
    # based on the inputs received by the simulator function.
    # This function currently returns dummy output values.
    # This will need to be adapted so it returns the correct output_values.
    # If the list of output names has only one name, then only a scalar
    # must be returned.
    if (len(output_names) > 1):
        output_values = [1.0] * len(output_names)
    else:
        output_values = 1.0
    ########################################################################
    return output_values

```

The arguments of the functions are in the next table

Arguments	Description
`configuration_file`	The Simulator model path or configuration file
`time`	The current simulation model time
`input_values`	The list of input values of the FMU
`input_names`	The list of input names of the FMU
`output_values`	The list of output values of the FMU
`output_names`	The list of output names of the FMU
`write_results`	A flag for writing results to a file

Note:

- The function `exchange` must return a list of output values which matchs the order of the output names.
- The function `exchange` can be used to invoke external programs/scripts which do not ship with the FMU. In this situation, the FMU will must be exported with the option `<-n>` set to `true`. The external programs/scripts will have to be installed on the target machine where the FMU is run. See *Creating an FMU* for details on command line options.
- Once `simulation_wrapper.py` is implemented, it must be saved under the same name and used as required argument for `SimulatorToFMU.py`.

CHAPTER

FOUR

CREATING AN FMU

This chapter describes how to create a Functional Mockup Unit. It assumes you have followed the *Installation and Configuration* instructions, and that you have created the Simulator model description file as well as the Python script required to interface the Simulator following the *Best Practice* guidelines.

Command-line use

To create an FMU, open a command-line window (see *Notation*). The standard invocation of the SimulatorToFMU tool is:

```
> python  <scriptDir>SimulatorToFMU.py -s <python-scripts-path>
```

where `scriptDir` is the path to the scripts directory of SimulatorToFMU. This is the `parser` subdirectory of the installation directory. See *Installation and Configuration* for details.

An example of invoking `SimulatorToFMU.py` on Windows is

```
# Windows:
> python parser\SimulatorToFMU.py -s parser\utilities\simulator_wrapper.py, d:\calc.py
```

Following requirements must be met hen using SimulatorToFMU

- All file paths can be absolute or relative.
- If any file path contains spaces, then it must be surrounded with double quotes.

`SimulatorToFMU.py` supports the following command-line switches:

Options	Purpose
-s	Paths to python scripts required to run the Simulator. The main Python script must be an extension of the `simulator_wrapper.py` script which is provided in `parser/utilities/simulator_wrapper.py`. The name of the main Python script must be `simulator_wrapper.py`.
-c	Path to the Simulator model file.
-i	Path to the XML input file with the inputs/outputs of the FMU. Default is `parser/utilities/SimulatorModelDescription.xml`
-v	FMI version. Options are `1.0` and `2.0`. Default is `2.0`
-a	FMI API version. Options are `cs` (co-simulation) and `me` (model exchange). Default is `me`.
-t	Modelica compiler. Options are `dymola` (Dymola) and `omc` (OpenModelica). Default is `dymola`.

The main functions of SimulatorToFMU are

- reading, validating, and parsing the Simulator XML input file. This includes removing and replacing invalid characters in variable names such as $*+-$ with _,
- writing Modelica code with valid inputs and outputs names,
- invoking a Modelica compiler to compile the *Modelica* code as an FMU for model exchange or co-simulation `1.0` or `2.0`.

Outputs of SimulatorToFMU

The main output from running `SimulatorToFMU.py` consists of an FMU named after the `modelName` specified in the input file, and a zip file called `"modelname" + ".scripts.zip"`. That is, if the `modelName` is called `Simulator`, then the outputs of `SimulatorToFMU` will be `Simulator.fmu` and `Simulator.scripts.zip`.

The FMU and the zip file are written to the current working directory, that is, in the directory from which you entered the command.

`"modelname" + ".scripts.zip"` contains the Python scripts that are needed to interface with the Simulator. The unzipped folder must be added to the `PYTHONPATH` of the target machine where the FMU will be used.

Any secondary output from running the SimulatorToFMU tools can be deleted safely.

Note that the FMU itself is a zip file. This means you can open and inspect its contents. To do so, it may help to change the "`.fmu`" extension to "`.zip`".

Note:

- FMUs exported using OpenModelica 1.11.0 needs almost `10` times more compilation/simulation time compared to Dymola 2017 FD01.
- FMUs exported using Dymola 2017 FD01 needs a Dymola runtime license to run. A Dymola runtime license is not be needed if the FMU is exported with a version of Dymola which has the `Binary Model Export` license.

CHAPTER

FIVE

DEVELOPMENT

The development site of this software is at https://github.com/LBNL-ETA/SimulatorToFMU.

To clone the master branch, type

git clone https://github.com/LBNL-ETA/SimulatorToFMU.git

CHAPTER

SIX

HELP

This chapter lists potential issues encountered when using SimulatorToFMU.

Compilation failed with Dymola

If the export of the Simulator failed when compiling the model with Dymola, comment out `"exit()"` in `parser/utilities/SimulatorModelica_Template_Dymola.mos` with `"//exit()"`, and re-run `SimulatorToFMU.py` to see why the complation has failed.

Compilation failed with OpenModelica

If the export of the Simulator failed when compiling the model with OpenModelica, check if the variable `OPENMODELICALIBRARY` is defined in the Windows `Environment Variables`.

`OPENMODELICALIBRARY` is the path to the libraries which are required by OpenModelica to compile Modelica models.

Simulation failed in OpenModelica and Dymola FMUs

If the simulation failed with the exported FMU, check if the `"modelname" + ".scripts.zip"` was added to the `PYTHONPATH` as described in *Outputs of SimulatorToFMU*. Please note that any software which is required to run the exported FMu will need to be installed on the target machine where teh FMU is run.

Simulation failed with Dymola FMUs

If an FMU exported using Dymola fails to run, check if the version of Dymola which exported the FMU had the `Binary Model Export` license. The `Binary Model Export` license is required to export FMUs which can be run without requiring a Dymola runtime license. You can also inspect the model description of the FMU to see if a Dymola runtime license is required to run the FMU.

CHAPTER

SEVEN

NOTATION

This chapter shows the formatting conventions used throughout the User Guide.

The command-line is an interactive session for issuing commands to the operating system. Examples include a DOS prompt on Windows, a command shell on Linux, and a Terminal window on MacOS.

The User Guide represents a command window like this:

```
# This is a comment.
> (This is the command prompt, where you enter a command)
(If shown, this is sample output in response to the command)
```

Note that your system may use a different symbol than ">" as the command prompt (for example, "`$`"). Furthermore, the prompt may include information such as the name of your system, or the name of the current subdirectory.

CHAPTER

EIGHT

GLOSSARY

Dymola Dymola, Dynamic Modeling Laboratory, is a modeling and simulation environment for the Modelica language.

Functional Mock-up Interface The Functional Mock-up Interface (FMI) is the result of the Information Technology for European Advancement (ITEA2) project *MODELISAR*. The FMI standard is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of XML-files, C-header files, C-code or binaries.

Functional Mock-up Unit A simulation model or program which implements the FMI standard is called Functional Mock-up Unit (FMU). An FMU comes along with a small set of C-functions (FMI functions) whose input and return arguments are defined by the FMI standard. These C-functions can be provided in source and/or binary form. The FMI functions are called by a simulator to create one or more instances of the FMU. The functions are also used to run the FMUs, typically together with other models. An FMU may either require the importing tool to perform numerical integration (model-exchange) or be self-integrating (co-simulation). An FMU is distributed in the form of a zip-file that contains shared libraries, which contain the implementation of the FMI functions and/or source code of the FMI functions, an XML-file, also called the model description file, which contains the variable definitions as well as meta-information of the model,additional files such as tables, images or documentation that might be relevant for the model.

Modelica 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.

MODELISAR MODELISAR is an ITEA 2 (Information Technology for European Advancement) European project aiming to improve the design of systems and of embedded software in vehicles.

PyFMI PyFMI is a package for loading and interacting with Functional Mock-Up Units (FMUs), which are compiled dynamic models compliant with the Functional Mock-Up Interface (FMI).

Python Python is a dynamic programming language that is used in a wide variety of application domains.

CHAPTER

NINE

ACKNOWLEDGMENTS

The development of this documentation was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies of the U.S. Department of Energy, under contract No. xxx.

The following people contributed to the development of this program:

- Thierry Stephane Nouidui, Lawrence Berkeley National Laboratory
- Michael Wetter, Lawrence Berkeley National Laboratory

CHAPTER

TEN

DISCLAIMERS

CHAPTER

ELEVEN

COPYRIGHT AND LICENSE

Copyright

xxxxx.

PYTHON MODULE INDEX

p

INDEX