2018 Building Performance Analysis Conference and
SimBuild co-organized by ASHRAE and IBPSA-USA
Chicago, IL
September 26-28, 2018

SIMULATORTOFMU: A PYTHON UTILITY TO SUPPORT BUILDING SIMULATION TOOL INTEROPERABILITY

Thierry S. Nouidui, Michael Wetter
Lawrence Berkeley National Laboratory, Berkeley, CA

ABSTRACT

The Functional Mock-up Interface (FMI) standard is an open standard for an application programming interface that allows run-time interoperability with other simulation tools, such as for co-simulation. As of today, more than 100 simulation tools use the FMI standard. However only few building simulation tools are exported as FMUs because of a lack of export utilities.

To address this limitation, we developed SimulatorToFMU, a Python utility which exports Python-driven simulation tools and Python scripts as FMUs.

First, we describe SimulatorToFMU. Second, we demonstrate its application to successfully export OPAL-RT, a real-time simulation tool as an FMU. Third, we conclude and discuss potential extensions to SimulatorToFMU.

INTRODUCTION

As buildings are becoming increasingly integrated and complex, it is often not possible to simulate in one single tool all building domains including HVAC, controls, envelope, and daylighting. Rather, it may be required to couple during runtime, for example, a simulator for the envelop with a simulator for the HVAC and control system.

Since different simulation tools are written in different languages with different Application Programming Interfaces (API), it is challenging to couple different simulation tools. To address this problem, the automotive industry has developed an open standard called Functional Mock-up Interface (Blochwitz et al. 2012) (FMI[1]). A tool which implements the FMI standard exposes itself in a way that makes it run-time interoperable with other simulators or tools that use the FMI standard. In this paper, we present a software package which leverages the FMI standard to support interoperability of simulation tools. The next section introduces the FMI standard. We then discuss the application of FMI in the building simulation community. This is followed by a description of SimulatorToFMU, the tool we implemented to support simulation tools interoperability. We show an application where SimulatorToFMU is used, and discuss other potential applications of SimulatorToFMU. We conclude the paper with an outlook for SimulatorToFMU.

FUNCTIONAL MOCK-UP INTERFACE

The FMI standard originated from the ITEA2 MODELISAR[2] project, a multi-million European project. FMI standardizes an API to be implemented by a simulation model to facilitate its interoperability with other models. The FMI standard provides two APIs, called FMI for model exchange and FMI for co-simulation. A simulation model which implements the FMI for model exchange API exposes its differential equations, whereas a model which implements the FMI for co-simulation standard exposes a function that returns the new value of the state variables, hence it includes a solver for its system of differential equations. A simulation model which implements the FMI standard is called a Functional Mock-up Unit (FMU).

An FMU is an input/output representation of the model which is distributed as a zip file with the extension *.fmu*. This zip file contains an XML[3] model description file with metadata about the model, the model itself, and a set of standardized C functions to interact with the model.

Figure 1 shows a model described by a first-order ordinary differential equation which has been exported as an FMU for model exchange (top right), and an FMU

[1] https//fmi-standard.org
[2] https://itea3.org/project/modelisar.html
[3] https://www.w3.org/XML/

for co-simulation (bottom right). The model has one continuous state *x*, an input *u*, and an output *y*.

When the model is exported as an FMU for model exchange, the FMU exposes as external inputs the time *t*, the state *x*, and the input *u* of the model. As outputs, the model exposes the state derivative *dx/dt* and the output *y*. The FMU does not provide a time integration algorithm. Hence, a time integration algorithm needs to be provided outside the FMU in order to compute the state trajectory.

When the model is exported for co-simulation, the FMU exposes its external inputs, the integration step size *Δt* and the input *u*. As outputs, it exposes, the integrated state *x* and the output *y*. In this situation the FMU contains an integrator which computes the state trajectory from a time *t* to a time *t+Δt*, with *Δt* being specified by the tool which imports and runs the FMU.

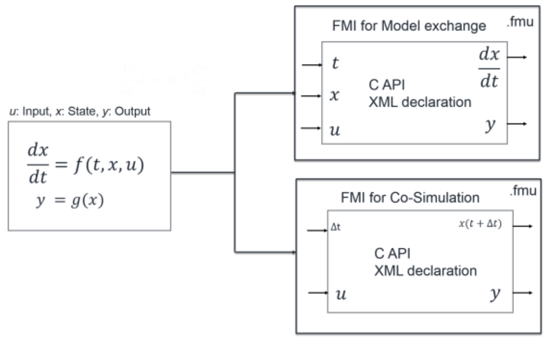

Figure 1 First-order ordinary differential equation exported as an FMU for model exchange (top right) and co-simulation (bottom right).

FUNCTIONAL MOCK-UP INTERFACE IN BUILDING SIMULATION

Although there are more than 100 tools which support the FMI standard, only few building simulation tools implement an FMI interface. EnergyPlus provides an FMI import (Nouidui et al. 2014) and export[4] interface for FMI 1.0 for co-simulation. TRNSYS models can be exported as FMUs for co-simulation 1.0 and 2.0 using the FMI++ library[5]. The Building Controls Virtual Test Bed[6] supports the import of FMUs for co-simulation and model exchange 1.0 and 2.0. WUFI+ supports the import of FMUs for co-simulation 1.0 (Pazold M. et al. 2012). This lists shows the limited number of tools which support FMI and motivates the need for FMU export facilities. The next sections describe SimulatorToFMU, a tool which allows exporting a certain class of simulation tools as FMUs.

SIMULATORTOFMU

SimulatorToFMU is a Python utility which allows to export Python modules that either implement an object with memory, or simply call a Python function. By a Python object with memory, we mean an application that can for example step forward in time and use at each time step data that it stored previously in memory as well as current inputs. This allows for example to interface certain simulators or data processing applications which may be written in Python or C (through the use of Python's C API). In contrast, Python functions do not require their memory to be preserved between their invocations.
Although not widespread, more and more simulation tools provide high-level APIs (CYME[7], PowerFactory[8]) which allow the tool to be driven by external tools. These APIs are often written in Python. The approach of SimulatorToFMU is to wrap such tool API around an FMI interface. This standardizes the interface to be used to communicate with the tool.
In the remainder of this document, we call for simplicity a Python-driven simulation tool or a Python function a simulator. The objective of SimulatorToFMU is to allow using such simulators with other tools that require run-time data exchange, using a standardized way.

To export the Python API of a simulator as an FMU, SimulatorToFMU requires the user to provide an input file which lists the names of the inputs and outputs of the FMU. SimulatorToFMU gets as additional arguments the path to the simulator Python API wrapper, and a flag which indicates if the tool is a Python object with memory, or a call to a Python function that does not require storing the memory between invocations. SimulatorToFMU uses these arguments to create an FMU with libraries and resources which are necessary to interface with the simulator.

The following sections describe the key components of SimulatorToFMU which include the input configuration file, the Python API wrapper, and the libraries developed to export the Python API wrapper as an FMU.

[4]https://github.com/lbl-srg/EnergyplusToFMU EnergyPlusToFMU is a tool developed to export building models developed in EnergyPlus as FMUs.
[5] https://sourceforge.net/projects/trnsys-fmu/.
[6] https://github.com/lbl-srg/bcvtb
[7] http:// http://www.cyme.com/
[8] https://www.digsilent.de/en/powerfactory.html

Configuration Input File

To export a simulator as an FMU, the user has to define the names of the inputs and outputs of the simulator to be exposed through the FMI interface. This is done in a XML configuration file. The path to the file is passed to SimulatorToFMU. In addition, the XML file contains the start values of the input variables.

Figure 2 shows a snippet of an input file where one input and one output variable are defined. To create the file, the user needs to specify the name of the FMU (Line 5). The user needs to define the inputs and outputs of the FMUs. This is done by adding ScalarVariable into the list of ModelVariables.

To parameterize the ScalarVariable as an input variable, the user needs to do the following:

- provide the name of the variable (Line 10),
- provide a description of the variable (Line 11),
- declare the causality of the variable ("input" for inputs, "output" for outputs) (Line 12),
- define the type of variable (only *Real* variables are supported) (Line 13),
- provide the unit of the variable (units as used by Modelica[9] [3] are supported) (Line 14),
- optionally give a start value for the input variable (Line 15).

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

- provide the name of the variable (Line 18),
- provide a description of the variable (Line 19),
- declare the causality of the variable ("input" for inputs, "output" for outputs) (Line 20),
- define the type of variable (only *Real* variables are supported) (Line 21),
- give the unit of the variable (units as used by Modelica are supported) (Line 22).

```
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"/>
25   </ScalarVariable>
26  </ModelVariables>
27 </SimulatorModelDescription>
```

Figure 2 XML configuration input file of a simulator exported by SimulatorToFMU.

Python API Wrapper

To export the Python API of a simulator, the user needs to write a Python API wrapper which will interface with the simulator. SimulatorToFMU comes with a Python template wrapper which needs to be modified by the user. Figure 3 shows a snippet of the Python API wrapper of a simulator which has memory.

Line 2 defines a dummy simulator which will be called by the Python API wrapper.

Line 21 declares the main function of the Python API wrapper which needs to be implemented to call the simulator using inputs obtained through the FMI interface. The arguments of the function are:

- The *configuration_file* which is the path to an input file needed to execute the model.
- The *time* which is the current simulation time.
- The *input_names* which is a list of FMU input names.
- The *input_values* which is a list with values of the FMU inputs.
- The *output_names* which is a list of FMU output names to be retrieved.
- The *write_results* which is a flag to indicate whether the simulator needs to write the simulation results to a file which is saved in the working directory of the import tool.
- The *memory* which is a variable which holds the memory of a Python object used in the simulator between invocation. If the simulator

[9] http://www.modelica.org. Modelica is an open source object-oriented equation-based Modelling language to conveniently model physical systems.

doesn't have memory, then this argument must be removed from the *exchange* function signature.

The function *exchange* gets these arguments and returns *output_values* which is a list with output values corresponding to the list of output names. If the simulator has memory, then the *exchange* function must also return the *memory*.

```
# Dummy simulator
Class Simulator():
    """
    Dummy Python-driven simulator
   """
  def __init__(self,
             configuration_file,
             time,
             input_names,
             input_values,
             output_names,
             write_results,
             memory):
          self.input_values = input_values

   def doTimeStep(self):
        return self.input_values + 1

# Main function to be implemented to interface
# with the simulator through FMI
def exchange(configuration_file,
              time,
              input_names,
              input_values,
              output_names,
              write_results,
              memory):
    """
    Return  a list of output values from
    the Python-driven simulation tool.

    """
    ########################################
     #EDIT AND INCLUDE CUSTOM CODE FOR TARGET
    # SIMULATOR
    if memory == None:
      s = Simulator(…) # Initialize Simulator
      memory =
              {'a':input_values,
                 'simulator': s}
    else:
      memory['a'] =
              memory['a'] + 1
      s = memory['s']
    output_values = s.doTimeStep ()
    if (…):
        raise ("")
    # Store the new state of the simulator
    memory['s'] = s
    ########################################
    return [output_values, memory]
```

Figure 3 Python API wrapper of a simulator exported by SimulatorToFMU.

At simulaton runtime, an import tool which implements the FMI interface will communicate with the simulator through the Python API wrapper. The next section describes how this communication has been enabled.

Modelica Wrapper Library

To enable communication with the Python API wrapper through the FMI interface, a Modelica library was developed. This library contains the Modelica function *simulator* (see Modelica snippet below) which is responsible for interfacing with the simulator. The Modelica function passes its input arguments to an external C function which invokes the Python API wrapper of the simlator to compute its output values. The inputs of the Modelica function are the Python API wrapper module name, the Python API wrapper function name, the simulation time, the number of inputs, the input names, the input values, the number of outputs, the output names, a flag to specify whether the simulator should write results to a file, and an external object which holds the memory of a Python object. This object is used to save and restore the states of a simulator which has memory. The return values of the function are the output values of the simulator.

```
function simulator
  "Function that communicates with the
  SimulatorToFMU Python API"
  input String moduleName
    "Name of the python module that contains the function";
  input String functionName=moduleName
    "Name of the python function";
  input String  conFilNam "Name of the python function";
  …
  output Real    dblOutVal[max(1, …)]
    "Double output values read from SimulatorToFMU";
  external "C" modelicaToSimulator(moduleName, …);
…
end simulator;
```

The external C function (*modelicaToSimulator*) which is called by the Modelica function is implemented in a C-library which is provided by SimulatorToFMU. The C-function uses the C-API of Python27 to communicate with the Python API wrapper of the simulator.

To export the simulator API as an FMU, SimulatorToFMU extracts from the XML configuration input file the input and output names of the model. It then writes a Modelia model which defines the inputs and outputs of the model that are passed to the function that communicates with the simulator.

The code snippet below shows the Modelica model of a simulator which has one input and one output extracted from the configuration input file shown in Figure 2. The Modelica model is autogenerated by SimulatorToFMU.

```
model Simulator
 "Block that exchanges a vector of real values with Simulator"
 extends Modelica.Blocks.Interfaces.BlockIcon;
 Modelica.Blocks.Interfaces.RealInput v(unit="V") "Voltage";
 Modelica.Blocks.Interfaces.RealOutput i(unit="A") "Current";
protected
 parameter Integer nDblInp(min=1) = 1
   "Number of double input values to be sent to Simulator";
 parameter Integer nDblOut(min=1) = 1
   "Number of double output values to receive from Simulator";
 Real dblInpVal[nDblInp] "Value to be sent to Simulator";
 Real uR[nDblInp]={v}
   "Variables used to collect values to be sent to Simulator";
 Real yR[nDblOut]={i}
   "Variables used to collect values received from Simulator";
 parameter String dblInpNam[nDblInp]={"v"}
   "Input variable name to be sent to Simulator";
 parameter String dblOutNam[nDblOut]={"i"}
   "Output variable names to be received from Simulator";
 parameter String moduleName="simulator_wrapper"
   "Name of the Python module that contains the function";
 parameter String functionName="exchange"
   "Name of the Python function";
equation
 // Compute values that will be sent to Simulator
 for _cnt in 1:nDblInp loop
   dblInpVal[_cnt] = uR[_cnt];
 end for;
 // Exchange data
 yR = SimulatorToFMU.Python27.Functions.simulator(...);
end Simulator;
```

Export of a Simulator as an FMU

To export the autogenerated Modelica model as an FMU, SimulatorToFMU invokes a Modelica parser to translate and compile the model as an FMU. SimulatorToFMU supports three Modelica parsers – the commercial Dymola[10] tool and the two open source tools JModelica[11] and OpenModelica[12].

Leveraging Modelica parsers to generate FMUs ensure a backward and forward compatibility of SimulatorToFMU with past and future versions of the FMI standard as these tools typically implement all versions of the FMI standard.
The snippet below shows the standard invocation of SimulatorToFMU

```
> python  SimulatorToFMU.py -s
simulator_wrapper.py -i simulator.xml
```

With –s, and –i, the user can specify the path to the Python API wrapper and the configuration input file.

The main functions of SimulatorToFMU are

- reading, validating, and parsing the simulator XML configuration input file. This includes removing and replacing invalid characters in variable names such as “*+-“ with “_”,
- writing Modelica code with valid input and output names,
- invoking a Modelica parser to translate and compile the Modelica code as an FMU for model exchange or co-simulation “1.0`” or ”2.0”.

SimulatorToFMU has been tested with Dymola 2018. JModelica 2.0, and 2.1, and with OpenModelica 1.11. SimulatorToFMU supports the export of FMUs which implement the FMI for model exchange, and the FMI for co-simulation APIs. FMI 1.0 and 2.0 are both supported.

On a Windows 7 machine, SimulatorToFMU needs about 14 seconds to export an FMU with JModelica 2.0 and Dymola 2018, and 75 seconds to export with OpenModelica 1.11. On a Linux Ubuntu 16.04 machine, it needs about 4 seconds to export FMUs with JModelica 2.0 and Dymola 2018.

APPLICATION

This section describes an application where SimulatorToFMU has been used to export a real-time simulation tool as an FMU to support hardware-in-the-loop simulation (HIL). The HIL scenario emulates a medium sized distribution grid with PV panels and an inverter which is controlled to limit PV generation. The micro-grid is physically installed at LBNL’s FLEXLAB[13]. The objective of the HIL is to develop and test new inverter control algorithms (See Figure 4).

To test the controller prior to deployment, a model of the grid with PV and inverter was developed and imported in the real-time simulator OPAL-RT, which emulates the physical micro-grid (See Figure 5). OPAL-RT has a Python API which allows interfacing with its models from Python. For the HIL, we developed a controller in Modelica. The controller limits the PV generation depending on grid voltage. We exported the controller as an FMU for model exchange 2.0 using JModelica 2.0. We modified the *exchange* function of SimulatorToFMU for this simulator. Specifically, we implemented functions which allows interfacing with the Python API of OPAL-RT to import, compile and execute OPAL-RT models. We used SimulatorToFMU to export the OPAL-RT models as FMUs for model exchange 2.0. For simulating the coupling of both FMUs, we used the master algorithm PyFMI[14].

[10] https://www.3ds.com/products-services/catia/products/dymola/
[11] http://www.jmodelica.org/
[12] https://openmodelica.org/
[13] http:/www.flexlab.com
[14] https://pypi.python.org/pypi/PyFMI

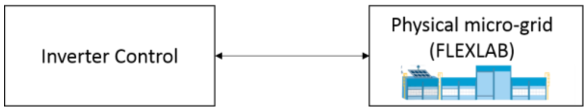

Figure 4 Development of PV inverter control installed at FLEXLAB.

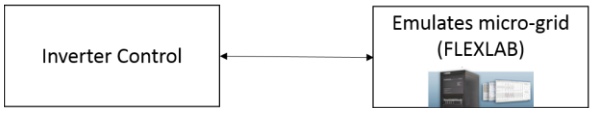

Figure 5 HIL testing of PV inverter control installed at FLEXLAB.

POTENTIAL APPLICATIONS

An application for which we anticipate the use of SimulatorToFMU is the evaluation of Building Controls or Machine Learning algorithms for Fault Detection and Diagnostics (FDD). If such algorithms are written in Python then a use case could be to

- export such algorithms as FMUs using SimulatorToFMU,
- use EnergyPlusToFMU to export EnergyPlus building models as FMUs,
- and use a master algorithm such as PyFMI to couple and co-simulate the FMUs (See Figure 6).

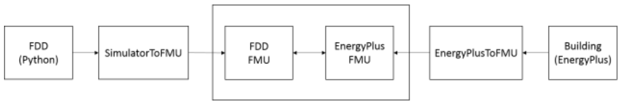

Figure 6 SimulatorToFMU exports FDD algorithm which is coupled to a building model FMU exported with EnergyPlusToFMU.

Another application for which we anticipate SimulatorToFMU to be used is the export of the daylighting simulation tool Radiance[15] as an FMU so it can for instance be linked with buildings FMUs. The exchange *function* of SimulatorToFMU could for this use case be implemented to invoke Radiance using the *subprocess*[16] library of Python.

CONCLUSION

This paper demonstrates how SimulatorToFMU can be used to facilitate interoperability between simulation tools. Although the application is focussing on a real-time simulation tool, it could be extended to other simulation tools which have a Python API. Future work should extend SimulatorToFMU to support other API languages such as JAVA or C. This will broaden the number of tools which can be exported as FMUs. SimulatorToFMU is open source and freely available at https://pypi.python.org/pypi/SimulatorToFMU/. Details on how to use SimulatorToFMU are in the user guide.

ACKNOWLEDGMENT

This work is supported by the U.S. Department of Energy SunShot National Laboratory Multiyear Partnership (SuNLaMP) program award number 31266.

REFERENCES

Blochwitz, T, Otter, M., Akesson, J., Arnold, M., Clauß, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A., 2012. Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. MODELICA Conference, September 3-5, 2012, Munich, Germany.

Nouidui, T. S., Wetter, M., Zuo, W., 2014. Functional Mock-up Unit for co-simulation import in EnergyPlus. Journal of Building Performance Simulation, 7(3):192-202, 2014.

Pazold M., Burhenne S., Radon J., Herkel S., Antretter F., 2012. Integration of Modelica models into an existing simulation software using FMI for Co-Simulation, Proceedings of the 9th International MODELICA Conference, September 3-5, 2012, Munich, Germany.

[15] https://www.radiance-online.org/

[16] https://docs.python.org/2/library/subprocess.html