CyDER Master Algorithm

Release 1.0.0

LBNL - Building Technology and Urban Systems Division

Jan 25, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

This user manual explains how to install and use PyFMI to couple the CYMDISTToFMU with other FMUs such as a GridDyn FMU.

CHAPTER

TWO

INSTALLATION AND CONFIGURATION

This chapter describes how to install PyFMI on Windows.

Running PyFMI with Python 3.4 on Windows 32 bit

PyFMI is a python package which can be used to import and run a CYMDIST FMU. In *PyFMI* version 2.3.1, a master algorithm was added to import and link multiple FMUs for co-simulation. At time of writing, there was no *PyFMI* 2.3.1 executable available for Python 3.4 for Windows 32bit (See PyPyi.). The next steps describe requirements and steps to perform to compile *PyFMI* version 2.3.1 from source.

Note: To avoid having to recompile *PyFMI* dependent libraries from source, we recommend to use pre-compiled Windows binaries whenever available.

Requirements

The next table shows the list of Python modules and softwares used to compile version 2.3.1 of PyFMI from source so it can run with Python 3.4 on Windows 32 bit.

Install PyFMI dependencies with

```
pip install -r fmu\master\bin\pyfmi-dependencies.txt
```

Below is a table with dependencies which fail to install using pip. For those, we recommend to use the MS Windows installer directly.

Modules	Version	Link
FMI Library	2.0.2 (binaries)	http://www.jmodelica.org/downloads/FMIL/FMILibrary-2.0.2-win32.zip
Scipy	0.16.1	https://sourceforge.net/projects/scipy/files/scipy/0.16.1
lxml	3.4.4	https://pypi.python.org/pypi/lxml/3.4.4
Assimulo	2.7b1	https://pypi.python.org/pypi/Assimulo/2.7b1
PyFMI	2.3.1 (source)	https://pypi.python.org/pypi/PyFMI

Note: *PyFMI* needs a C-compiler to compile the source codes. We used the Microsoft Visual Studio 10 Professional.

Compilation

To compile *PyFMI* from source, run

```
python setup.py install --fmil-home=path_to_FMI_Library\
```

where `path_to_FMI_Library\` is the path to the FMI library.

CHAPTER

THREE

USAGE OF CYMDIST AS AN FMU

The following requirements must be met to import and run a CYMDIST FMU:

1. CYME version 7.2 must be installed. CYME can be downloaded from www.cyme.com.
2. The CYMDIST Python API directory must be added to the `PYTHONPATH`. This directory contains scripts needed at runtime by the CYMDIST FMU.

   The CYMDIST Python API directory is in the installation folder of CYME. It can typically be found in `path_to_CYME/CYME/cympy`, where `path_to_CYME` is the path to the installation folder of CYME 7.2.

   To add the CYMDIST Python API scripts folder to the `PYTHONPATH`, add `path_to_CYME/CYME` to the `PYTHONPATH` with following steps:

   - 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 a variable named `PYTHONPATH` environment variable and select it. If the variable does not exist, create it. Click Edit.
   - In the Edit System Variable (or New System Variable) window, specify the value of the PYTHONPATH environment variable which should be in our case `path_to_CYME/CYME`.
3. The CYMDIST installation directory must be added to the system `PATH`. This directory contains runtime DLLS (`mkl_core.dll`, `mkl_def.dll`) that are needed at runtime by the CYMDIST FMU.
4. Upon request, the simulation results are saved in a result file which is created in the current working directory. The name of the result file is `xxx_result_.pickle`, where xxx is the FMU model name as defined in the XML input file.

CHAPTER

FOUR

CO-SIMULATION

This section explains how to link a CYMDIST FMU with another FMU for co-simulation. We used the GridDyn FMU for the simulation coupling and explain the problematic caused by coupling the two tools with the solution implemented in CyDER.

Solving Algebraic Loops between CYMDIST and GridDyn

Coupling GridDyn FMU to CYMDIST creates an algebraic loop that requires an iterative solution. Solving this nonlinear system of equations is not likely to be robust, because GridDyn and CYMDIST both contain iterative solvers. Therefore, the residual function of this algebraic loop will have numerical noise which is caused by the embedded iterative solvers. To avoid this problem, LLNL proposes to approximate CYMDIST by a polynomial and then solve the coupled system of equations that is formed by this polynomial and GridDyn. Details about approximating CYMDIST with a polynomial are discussed in the next sections. On occasion, this will require that CYMDIST to be called multiple times with varying input, but the same state variables, in order to compute its polynomial approximation.

Next, we describe how the system model needs to be configured to allow GridDyn to approximate CYMDIST as a polynomial. We consider the case where CYMDIST is coupled to GridDyn. For the discussion, we use p for parameters (which do not change in time), u for inputs, which may change in time, y for outputs and t for time. We will denote with subscript g quantities of GridDyn, and with subscript c quantities of CYMDIST. We will use $f()$ to denote a function with unassigned variables.

Hence, the GridDyn model is of the form

$y_g = f_g(p_g, u_g, t)$,

while the CYMDIST model is of the form

$y_c = f_c(p_c, u_c, t)$.

Note that we make the assumption that both models only take voltage or current as inputs, that is, we assume $u_g = y_c$ and $u_c = y_g$, and the CYMDIST model has no state. These models are connected as shown in **:num:‘Figure #fig-couplingwloop‘**.

To implement the computation, these models are encapsulated as FMUs. To approximate CYMDIST as a polynomial, we will need to also propagate parameters p_g and a connection list that declares the connection between y_g and u_c and between y_c and u_g. We will call this connection list p_l. Hence, the CYMDIST FMU needs to be a function of the form

$yc = f_c(p_c, u_c, t)$,

while the GridDyn FMU is of the form

$yg = f_g(p_g, p_c, p_l, u_g, t)$.

Hence, the GridDyn model needs to take additional parameters the parameters p_c to parameterize the CYMDIST model, and p_l to connect the outputs to the inputs. The CYMDIST FMU is an FMU-ME 2.0, because it needs to

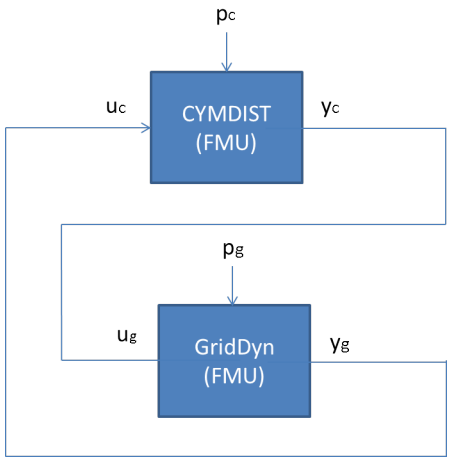

Fig. 4.1: Coupling of CYMDIST with GridDyn with algebraic loop.

be evaluated by GridDyn without advancing time when approximating it as a polynomial, which is not allowed for FMI-CS 2.0.

As CYMDIST obtains from GridDyn the input u_c that corresponds to the solution of the closed loop, the FMUs are connected as in **:num:'Figure #fig-couplingwoloop'**.

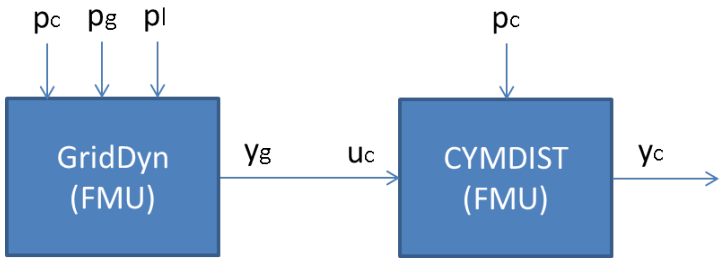

Fig. 4.2: Coupling of CYMDIST with GridDyn without algebraic loop.

The sequence of evaluations will be as follows: During instantiation, both FMUs get their parameters assigned. The GridDyn parameters include the parameters for CYMDIST p_c and the output-input connection list p_l. Then, GridDyn will instantiate a CYMDIST FMU, and connect u_c to y_g. This CYMDIST FMU, which we call $f_c^*(,,)$, will not be visible to the outside. When GridDyn is invoked, it will approximate $f_c^*(,,)$, compute a converged solution using this approximation, and compute the output y_g. The master algorithm will then assign $u_g := y_c$ and evaluate the FMU $f_g(p_g, p_c, p_l, u_g, t)$, which completes the time step.

Coupling CYMDIST and GridDyn FMUs using PyFMI

Next, we showed a snippet of the master algorithm which is used to couple a CYMDIST FMU (`CYMDIST.FMU`) with a GridDyn FMU (`GridDyn.fmu`).

Line imports the *PyFMI* modules which is needed for the coupling.

Line 25 loads the CYMDIST FMU

Line 26 loads the GridDyn FMU. We used in this example a GridDyn FMU which models the IEEE 14-Bus System.

Line 28 and 29 set-up the parameters for the simulation.

Line 32 - 37 create the vector of input and output names for both FMUs.

Line 45 - 53 get the value references of the CYMDIST and GridDyn variables

Line 69 and 70 initialize the FMUs.

Line 73 calls event update for the CYMDIST FMU. This is required by CYMDIST which is a model exchange FMU and hence needs to call this function prior to entering the continous time mode.

Line 74 CYMDIST enters in continuous time mode.

Line 77 In the loop, CYMDIST and GridDyn are evaluated.

First, The outputs of GridDyn are retrieved. these outputs must be the coverged solution between CYMDIST and GridDyn at the time when GridDyn is invoked.

Second, The outputs of GridDyn are set as inputs of CYMDIST at the same time instant. CYMDIST computes the outputs at that time instant and send the updated outputs.

Line 88 and 89 complete the simulation and terminate both FMUs.

```
1  from pyfmi import load_fmu
2
3  def simulate_cymdist_griddyn14bus_fmus():
4      """Simulate coupled GridDyn and CYMDIST FMUs.
5
6      """
7      # Simulation parameters
8      start_time = 0.0
9      stop_time  = 300
10     step_size  = 300
11
12     # Path to the CYMDIST configuration file
13     path_config=os.path.abspath("config.json")
14     # Conversion to byte for PyFMI
15     cymdist_con_val_str = bytes(path_config, 'utf-8')
16
17     griddyn_input_valref=[]
18     griddyn_output_valref=[]
19     griddyn_output_values=[]
20
21     cymdist_input_valref=[]
22     cymdist_output_valref=[]
23     cymdist_output_values=[]
24
25     cymdist = load_fmu("../fmus/CYMDIST/CYMDIST.fmu", log_level=7)
26     griddyn=load_fmu("../fmus/griddyn/griddyn14bus.fmu", log_level=7)
27
28     cymdist.setup_experiment(start_time=start_time, stop_time=stop_time)
29     griddyn.setup_experiment(start_time=start_time, stop_time=stop_time)
30
31     # Define the inputs
32     cymdist_input_names = ['VMAG_A', 'VMAG_B', 'VMAG_C', 'VANG_A', 'VANG_B', 'VANG_C']
33     cymdist_output_names = ['IA', 'IB', 'IC', 'IAngleA', 'IAngleB', 'IAngleC']
34
35     griddyn_input_names = ['Bus11_IA', 'Bus11_IB', 'Bus11_IC',
36                            'Bus11_IAngleA', 'Bus11_IAngleB', 'Bus11_IAngleC']
37     griddyn_output_names = ['Bus11_VA', 'Bus11_VB', 'Bus11_VC',
38                     'Bus11_VAngleA', 'Bus11_VAngleB', 'Bus11_VAngleC']
39
40     # Get the value references of griddyn inputs
```

```
    for elem in griddyn_input_names:
        griddyn_input_valref.append(griddyn.get_variable_valueref(elem))

    # Get the value references of griddyn outputs
    for elem in griddyn_output_names:
        griddyn_output_valref.append(griddyn.get_variable_valueref(elem))

    # Get the value references of cymdist inputs
    for elem in cymdist_input_names:
        cymdist_input_valref.append(cymdist.get_variable_valueref(elem))

    # Get the value references of cymdist outputs
    for elem in cymdist_output_names:
        cymdist_output_valref.append(cymdist.get_variable_valueref(elem))

    # Set the flag to save the results
    cymdist.set("_saveToFile", 0)
    # Get the initial outputs from griddyn
    # Get value reference of the configuration file
    cymdist_con_val_ref = cymdist.get_variable_valueref("_configurationFileName")

    # Set the configuration file
    cymdist.set_string([cymdist_con_val_ref], [cymdist_con_val_str])

    # Set the value of the multiplier
    griddyn.set('multiplier', 3.0)

    # Initialize the FMUs
    cymdist.initialize()
    griddyn.initialize()

    # Call event update prior to entering continuous mode.
    cymdist.event_update()
    cymdist.enter_continuous_time_mode()

    # Co-simulation loop
    for tim in np.arange(start_time, stop_time, step_size):
        cnt+=1
        # Get the outputs from griddyn
        griddyn_output_values = (griddyn.get_real(griddyn_output_valref))
        # set the time in cymdist
        cymdist.time = tim
        # Set the inputs of cymdist
        cymdist.set_real(cymdist_input_valref, griddyn_output_values)
        # Get the outputs of cymdist
        cymdist_output_values = (cymdist.get_real(cymdist_output_valref))
    # Terminate FMUs
    cymdist.terminate()
    griddyn.terminate()

if __name__ == '__main__':
    simulate_cymdist_griddyn14bus_fmus()
```

CHAPTER

FIVE

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

SIX

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

SEVEN

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
- Jonathan Coignard, Lawrence Berkeley National Laboratory
- Michael Wetter, Lawrence Berkeley National Laboratory

CHAPTER

EIGHT

DISCLAIMERS

CHAPTER

NINE

COPYRIGHT AND LICENSE

Copyright

License Agreement

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code ("Enhancements") to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free, perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.

INDEX