Documentation of Code Utilized in Test Case To Prove Compatibility of MQTT Brokers and FMU Simulation Models in a Python Environment

David Anderson - July 10, 2023

The physical NTP system at ORNL has a large number of sensors producing valuable signals, which need to be utilized in the simulation of the nuclear reactor. These sensors will be sent to a MQTT broker, where they are pulled for the simulation. The output of the simulation will be sent back to the MQTT server. Node-Red is used to create a visualization dashboard.

Python is being used due to availability of packages for the FMU simulation (FMpy) and interaction with MQTT (paho-mqtt). Communication with MQTT is done with JSON strings.

Packages used:

```
from __future__ import division, print_function, unicode_literals, absolute_import
import os
from fmpy import simulate_fmu, read_model_description, extract, dump, instantiate_fmu,
read_csv, write_csv
import fmpy
import numpy as np
from paho.mqtt import client as mqtt
import random
import time
import json
```

Set MQTT broker:

- The broker and port connect you to the correct server
- JSON strings containing all of the sent and received data is contained in ‘dja/JSON_input’ and ‘dja/JSON_output’ respectively

---

```
# MQTT broker details
broker = "test.mosquitto.org"
port = 1883

# MQTT topics
```

```
topic_input = 'dja/JSON_input'
topic_output = 'dja/JSON_output'
```

Create the dictionaries which will store the data which is sent and received from MQTT:

- I'm sure this is a more slick method to pulling the topics, however this is my current iteration. I am working to use the 'fmuInputs' and 'fmuOutputs' variables (comes in later) to make this more generalized

---

```
# Input topics and corresponding keys for received inputs
input_topics = {
'drum1': 'drum1',
'drum2': 'drum2'
}

# Output keys for calculated outputs
output_keys = {
'output_dollars': 'output_dollars',
'output_dollars_2': 'output_dollars_2'
}

# Dictionary to store received inputs with timestamps
received_inputs = {key: {'values': [], 'timestamps': []} for key in
input_topics.values()}

# Dictionary to store calculated outputs with timestamps
calculated_outputs = {key: {'values': [], 'timestamps': []} for key in
output_keys.values()}

def initialize_dictionaries():
""" call to re-initialize """
# Dictionary to store received inputs with timestamps
received_inputs = {key: {'values': [], 'timestamps': []} for key in
input_topics.values()}

# Dictionary to store calculated outputs with timestamps
calculated_outputs = {key: {'values': [], 'timestamps': []} for key in
output_keys.values()}
```

Functions for paho-MQTT:

- The function ‘on_connect’ is used to help confirm the connection with MQTT
- Function ‘on_message’ sends a message every time an input variable is changed
- Function ‘process_data’ is called after the FMU has ran and data needs to be compiled
    - topic_output is the MQTT location where the data is to be sent
    - Automatically calls ‘submit_data’ function
- Function ‘submit_data’ formats data nicely into JSON format and ships it over to MQTT

```
# Callback function when the client is connected to the MQTT broker
def on_connect(client, userdata, flags, rc):
print("Connected to MQTT broker with result code: " + str(rc))
# Subscribe to the input topic
client.subscribe(topic_input)

# Callback function when a message is received on the subscribed topic
def on_message(client, userdata, msg):
received_data = json.loads(msg.payload.decode())
for input_key, input_topic in input_topics.items():
received_value = received_data.get(input_topic)
received_inputs[input_key]['values'].append(received_value)
received_inputs[input_key]['timestamps'].append(time.ctime())
print("Received variable on topic", input_topic + ":", received_value, '@',
time.ctime())

# Function to process received inputs and run submit_data
def process_data(topic_output, output_data):
# Store calculated outputs with timestamps
for output_key, output_value in output_data.items():
calculated_outputs[output_key]['values'].append(output_value)
calculated_outputs[output_key]['timestamps'].append(time.ctime())
submit_data(topic_output, output_data)

# Function to calculate output data and publish JSON
def submit_data(topic_output, output_data):
output_json = json.dumps(output_data)
client.publish(topic_output, output_json)
print("Published output JSON:", output_json)
```

Create MQTT Client, set callback functions, and connect to broker:

- Should print '0' to confirm it correctly connected to MQTT server
- Once this is ran, you will receive messages whenever a MQTT variable is changed
    - At this point (now that you are connected), go to the MQTT server and move around the control rod locations so the received_inputs dictionaries begin to populate. This needs to be done because we have to reference these dictionaries when we want to run the FMU

```
# Create a MQTT client instance
client = mqtt.Client()

# Set the callback functions
client.on_connect = on_connect
client.on_message = on_message

# Connect to the MQTT broker
client.connect(broker, port, 60)
```

Functions for running FMU:

- Adapted from code provided by Vineet Kumar

```
# use the step_finished callback to stop the simulation at pause_time
def step_finished(time, recorder):
""" Callback function that is called after every step """
return time < pause_time

def runFMU(inputs, fmu_state, start_time):
""" Run fmu every time step """
results = simulate_fmu(filename=settings['filename'],
start_time=start_time,
stop_time=start_time+step_time,
output_interval=step_time,
input = get_inputs(),
output=settings['outputs'],
fmu_instance=fmu_instance,
fmu_state=fmu_state,
terminate=False,
step_finished=step_finished,
```

```
debug_logging=False)
# retrieve the FMU state
fmu_state = fmu_instance.getFMUState()
return results, fmu_state

def get_inputs():
""" Returns input variables in JSON format """
################################################################################
# time varying input (set appropriate input variable names + data types here) #
# i.e. 'drum_angle_1' is an FMU input variable #
################################################################################
dtype = [('drum_angle_1', np.double), ('drum_angle_2', np.double)]

#######################################################################################
####
# pull most recent MQTT data values to be used as FMU input #
# i.e. 'received_inputs' holds 'drum1' data, where ['values'][-1] pulls most recent
data #
# this structure is here to show how inputs will be pulled and sent to FMU #
#######################################################################################
####
inputs = np.array((received_inputs['drum1']['values'][-1],
received_inputs['drum2']['values'][-1]), dtype=dtype)
return inputs
```

Running the FMU while pulling and sending data in JSON format:

- Adapted from code provided by Vineet Kumar
- Runs the FMU for an initial time (until pause time) then runs it with a defined timestep before termination
  - FMU must be compiled with a fixed-step solver with the same time steps you wish to simulate with
- Runs the FMU at a constant time step (very small right now, should eventually be running in real-time)

```
initialize_dictionaries

# define which FMU file you will be running
fmu_path = "/Users/davidanderson/skoo/URSI/FMUs/simple_rxt_lookup.fmu"
fmuInputs = []
fmuOutputs = []

if __name__ == "__main__":
# get the FMU file name
fmu_filename = fmu_path

# read and dump the FMU file
dump(fmu_filename)
model_description = read_model_description(fmu_filename)

unzipdir = extract(fmu_filename)

# instantiate the FMU before simulating it, so we can keep it alive
fmu_instance = instantiate_fmu(
unzipdir=unzipdir,
model_description=model_description,
)

# gather and print info for inputs + outputs
for variable in model_description.modelVariables:
if variable.causality == 'input':
fmuInputs.append(variable.name)
if variable.causality == 'output':
fmuOutputs.append(variable.name)

print("Correct variable input names")
```

```
print("FMU Input variable list is ", fmuInputs)
print("FMU Output variable list is ", fmuOutputs)


#############################
# define runtime parameters #
#############################
step_time = 0.02 # time between FMU outputted data points (indiv. simulation lengths)
start_time = 0 # Start time of the FMU in s
stop_time = 5000 # Stop time of the FMU in s
pause_time = 1 # Initial pause time in s

dtype = [('drum_angle_1', np.double), ('drum_angle_2', np.double)]

inputs = np.array((received_inputs['drum1']['values'][-1],
received_inputs['drum2']['values'][-1]), dtype=dtype)

inputs = get_inputs()

settings = {
'filename':unzipdir,
'start_time':start_time,
'stop_time':stop_time,
'output_interval':step_time,
'outputs':fmuOutputs
}

# Run the FMU for an initial time until pause time
results = simulate_fmu(filename=settings['filename'],
start_time=settings['start_time'],
output_interval=settings['output_interval'],
input=get_inputs(),
output=settings['outputs'],
fmu_instance=fmu_instance,
terminate=False,
step_finished=step_finished,
debug_logging=False)
# Log the results of the FMU for the initial run
# After this point, the FMU output gets sent to the MQTT broker (allows for weird
transient to pass)
resultSummary = results
```

```
# process_data saves the data and sents it to MQTT
process_data(topic_output, {'output_dollars':resultSummary['output_dollars'][-1],
'output_dollars_2':resultSummary['output_dollars_2'][-1]})
# retrieve the FMU state
fmu_state = fmu_instance.getFMUState()
start_time = pause_time
print("FMU ran for initial time, now starting primary loop.")

# While loop to run the FMU with a predefined timestep
while start_time <= stop_time - 2 * step_time:
inputs = np.array((received_inputs['drum1']['values'][-1],
received_inputs['drum2']['values'][-1]), dtype=dtype)
# truncate start time to 2 decimals
start_time = round(start_time, 3)
# Advance the pause time
pause_time = start_time + step_time
# run FMU
resultsApp, fmu_state = runFMU(inputs, fmu_state, start_time)
resultsApp = resultsApp[0:]
# publish
process_data(topic_output, {'output_dollars':resultsApp['output_dollars'][-1],
'output_dollars_2':resultsApp['output_dollars_2'][-1]})
print('Start time = ', start_time, "input = ", get_inputs(), "results = ", resultsApp)
# Append the results every time step
resultSummary = np.concatenate((resultSummary, np.expand_dims(resultsApp[-1],
axis=0)))
# Advance the start time
start_time += step_time
time.sleep(.1)


print("The FMU will terminate in 1 time step.")
results = simulate_fmu(filename=settings['filename'],
start_time=round(start_time, 2),
stop_time=settings['stop_time'],
output_interval=step_time,
input=np.array((received_inputs['drum1']['values'][-1],
received_inputs['drum2']['values'][-1]), dtype=dtype),
output=settings['outputs'],
fmu_instance=fmu_instance,
fmu_state=fmu_state,
terminate=True,
```

```
debug_logging=False)

# Log the results of the FMU for the final run
process_data(topic_output, {'output_dollars':results['output_dollars'][-1],
'output_dollars_2':results['output_dollars_2'][-1]})
resultSummary = np.concatenate((resultSummary, np.expand_dims(resultsApp[-1],
axis=0)))
# combine and output the results to a csv file
# write_csv(os.path.join(mypath, '..', 'output','fmu_results.csv'), resultSummary,
columns=None)
print("Simulation finished")
```