BOPTEST Tutorial #1

An Introduction to BOPTEST Software for Test Case Development and Interaction

By David Blum and Javier Arroyo

IBPSA Project 1 Expert Meeting

Work Package 1.2

Rome, Italy

August 31, 2019

Last Updated: 8/29/2019

Introduction

The Building Optimization Test Framework (BOPTEST) is developed to enable the fair comparison and benchmarking of advanced control strategies in buildings. The framework is composed of three main parts:

1. Test Cases
2. Key Performance Indicators
3. Software Runtime Environment

Test Cases are complete building emulators. They not only include detailed physical models of buildings and their systems, but also all boundary condition data necessary to test a controller for the building. This includes weather, schedules, energy and fuel prices, occupant comfort definitions, and documentation. A test case is ultimately represented by a Functional Mockup Unit (FMU). This is known as the test case FMU.

Key Performance Indicators (KPI) are metrics to evaluate controller performance. A set of core KPIs have been identified and defined by the IBPSA Project WP1.2 team, which will get calculated for every tested controller for each test case.

The Software Runtime Environment is designed to create a controlled testing environment that is efficiently deployed. It makes use of Docker containerization to create a well-defined simulation environment which contains Python modules to manage simulation of the test case FMU, deliver boundary condition forecasts, store results, calculate KPIs, and deliver information about the test case. This functionality is accessed by a user through a RESTful HTTP API.

The purpose of this tutorial is to provide a basic introduction of the BOPTEST framework and associated software components to potential test case developers and controller testers. Therefore, the tutorial is split into three main parts, I) Software Requirements, II) Test Case Development and III) Test Case Interaction.

More information at https://github.com/ibpsa/project1-boptest and in [1].

---

[1] D. H. Blum, F. Jorissen, S. Huang, Y. Chen, J. Arroyo, K. Benne, Y. Li, V. Gavan, L. Rivalin, L. Helsen, D. Vrabie, M. Wetter, and M. Sofos. (2019). "Prototyping the BOPTEST framework for simulation-based testing of advanced control strategies in buildings." In Proc. of the 16th International Conference of IBPSA, Sep 2 – 4. Rome, Italy.

Table of Contents

Part I: Software Requirements

The following is a list of software needed for this tutorial. Note that items 1-3 are only needed for test case development, and not for test case interaction or controller testing.

1. Modelica IBPSA Library
2. Modelica Buildings Library
3. Dymola
4. Python 2.7
5. Docker
6. BOPTEST Repository Software
7. Cygwin (for Windows only).

Please follow the steps below to obtain links for downloading and installing the proper versions.

A. Download the Modelica IBPSA Library master branch from https://github.com/ibpsa/modelica-ibpsa. Extract the zip file to a directory location of your choice.

B. Download the Modelica Buildings Library master branch from https://github.com/lbl-srg/modelica-buildings. Extract the zip file to a directory location of your choice.

C. Install Dymola from https://www.3ds.com/products-services/catia/products/dymola/. A demo version will suffice for this tutorial.

D. Install Python 2.7 from https://www.python.org/downloads/. If other versions of Python are on the system, create an environment to run Python 2.7.

E. Install Docker by following the instructions for your system:

   Windows:
   https://docs.docker.com/docker-for-windows/install/

Ubuntu or Other Linux (see side panel at link):
https://docs.docker.com/install/linux/docker-ce/ubuntu/
MacOS:
https://docs.docker.com/docker-for-mac/install/

Be sure to test your Docker installation by opening a terminal and running a test example container stored in an image on the Docker Hub (For Windows, try PowerShell. For Linux and MacOS, you may need to add sudo permission):

$ docker run hello-world

If you are using Windows and getting a permission error, you may have to add yourself to the "docker-users" group in computer management. See here for more information and how to resolve:
https://icij.gitbook.io/datashare/faq/you-are-not-allowed-to-use-docker-you-must-be-in-the-docker-users-group-.-what-should-i-do

F. Download the BOPTEST Repository Software issue85 branch from https://github.com/ibpsa/project1-boptest/tree/issue85_testcase_bestest_air. Extract the zip file to a directory location of your choice and add the root directory for the repository to your PYTHONPATH environmental variable.

Note: All path references throughout the tutorial will be made with reference to the root directory for this repository!

G. For Windows users only, install Cygwin from http://www.cygwin.com/.

Note: Make sure to include the "make" command when installing Cygwin (this one is not installed by default), and also to add "C:\cygwin64\bin" to your path environmental variable. Notice that the previous absolute path could vary depending on your choice of Cygwin version and installation folder.

Part II: Test Case Development

This part will step through the process of creating the test case FMU. It begins with an already-developed Modelica model of a simple single zone building defined by the BESTEST Case 900 envelope with heating and cooling provided by a fan coil unit (FCU). There are four main steps in this part:

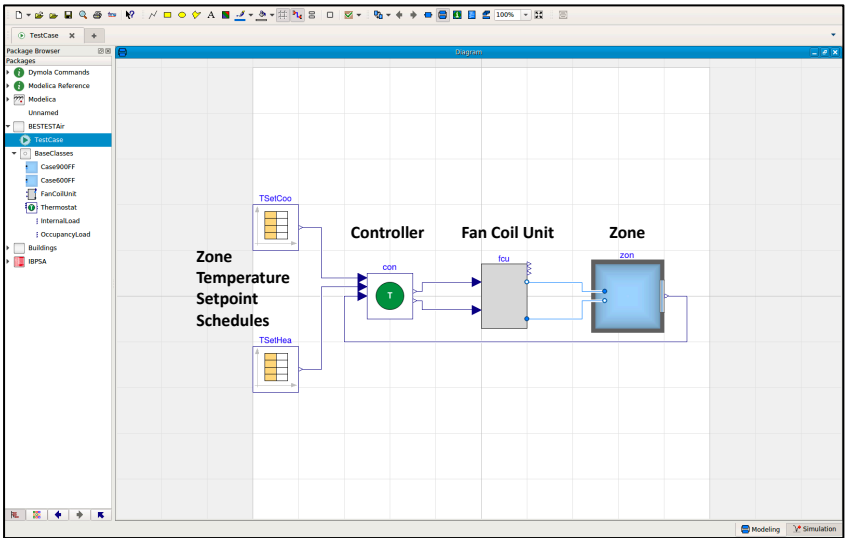
1. Develop and document test case building model.
2. Add and configure signal exchange blocks to define inputs, outputs, and variables needed for KPI calculations.
3. Create boundary condition data to include with building model.
4. Compile the test case FMU.

1. Develop and Document Test Case Building Model

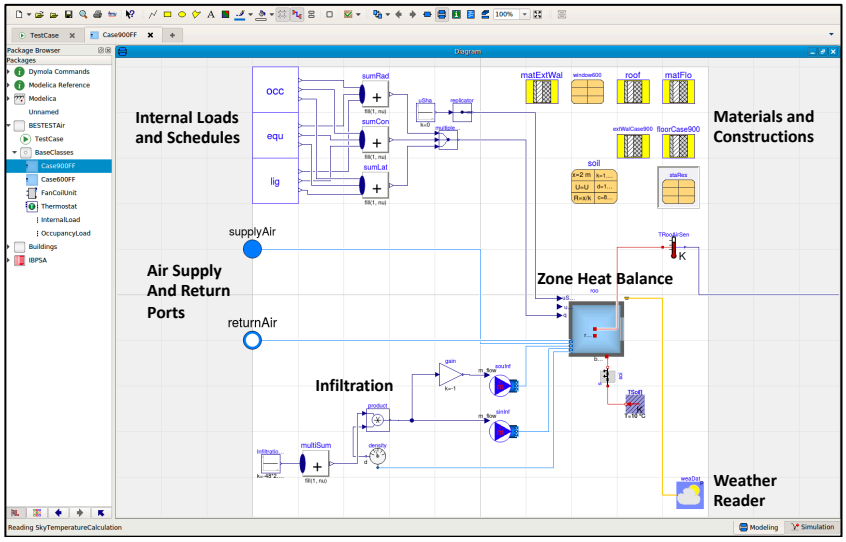
For this tutorial, this step is already completed. However, let's take some time to get familiar with the building design and model.

A. In Dymola, load the Modelica Buildings Library and the test case model at `testcase_bestest_air/models/BESTESTAir/package.mo`.

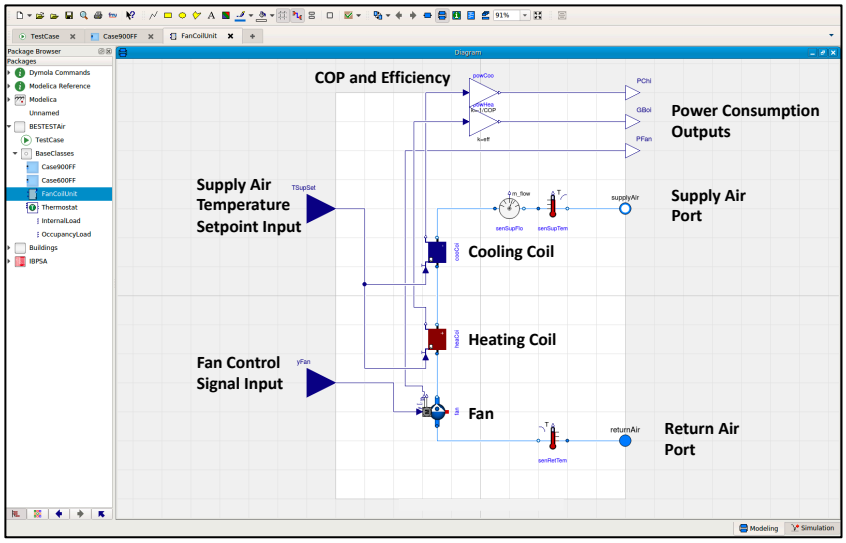
B. Open the model `BESTESTAir.Testcase`. Notice the zone envelope model, fan coil unit model, controller, and zone temperature setpoint schedules for heating and cooling.



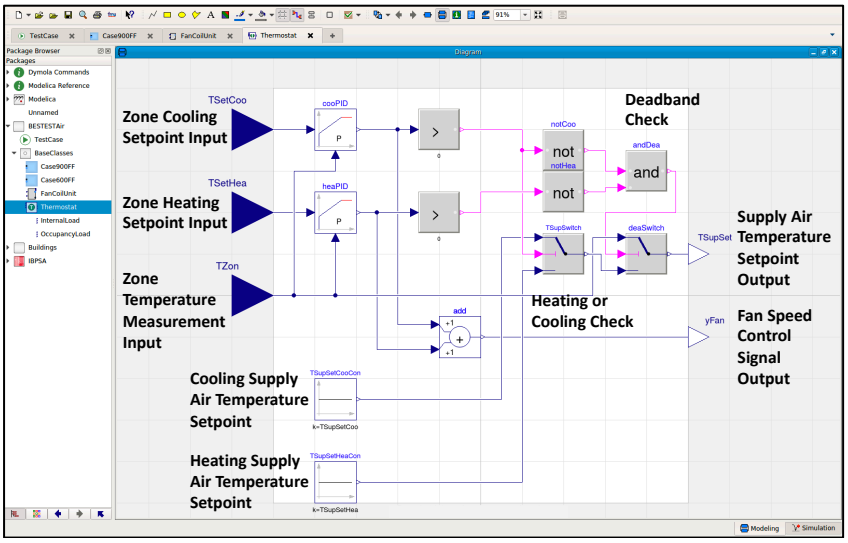
C. Open the model `BESTESTAir.Testcase.zon` and notice the zone heat balance implementation, materials and construction definitions, internal load models, infiltration model, weather reader, and supply and return air ports.



D. Open the model `BESTESTAir.Testcase.fcu` and notice the supply fan, heating coil, cooling coil, and supply and return air ports. Cooling electrical power is calculated using the thermal load on the cooling coil and a constant COP, heating thermal power is calculated using the thermal load on the heating coil and a constant efficiency, and fan electrical power is calculated using the fan performance curves.

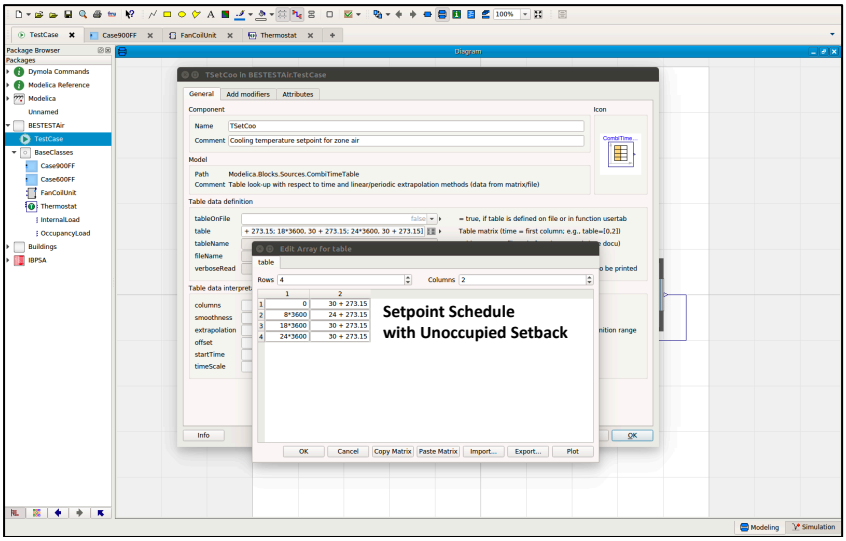


E. Open the model `BESTESTAir.Testcase.con` and notice the logic used to control the fan speed and supply air temperature setpoint. The fan speed is controlled by feedback controllers using zone temperature setpoints for heating and cooling and the measured zone air temperature. If the zone is in heating mode, the heating supply air temperature setpoint is used. If the zone is in cooling mode, the cooling supply air temperature setpoint is used. If the zone is in deadband (neither heating nor cooling), the supply air temperature setpoint is set to the zone air temperature to indicate no heating and cooling required.

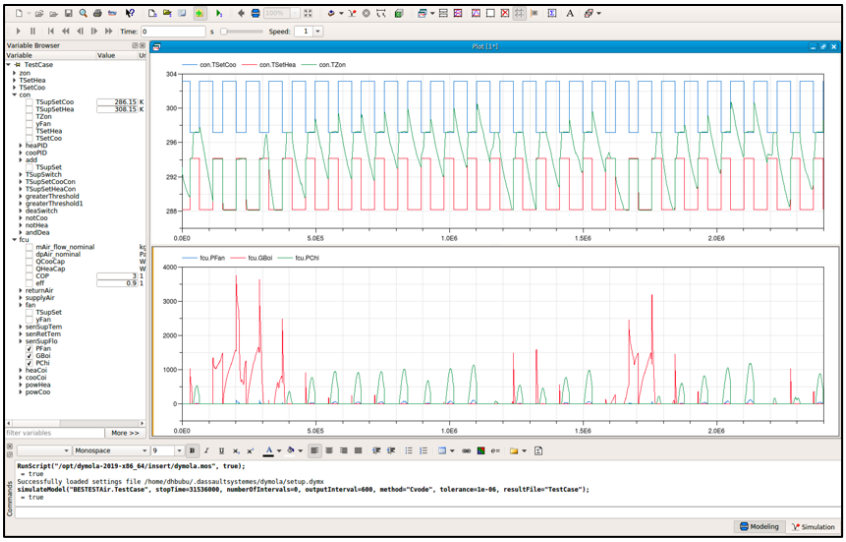


F. Open the models `BESTESTAir.Testcase.TSetCoo` and `BESTESTAir.Testcase.TSetHea` and notice that the zone temperature

setpoints have occupied and unoccupied values corresponding to the occupancy schedule in the zone model we explored before.



G. Simulate the model (not possible with Demo version) and notice the behavior of the FCU and controller to provide heating and cooling to the zone in order to maintain the zone temperature setpoints.



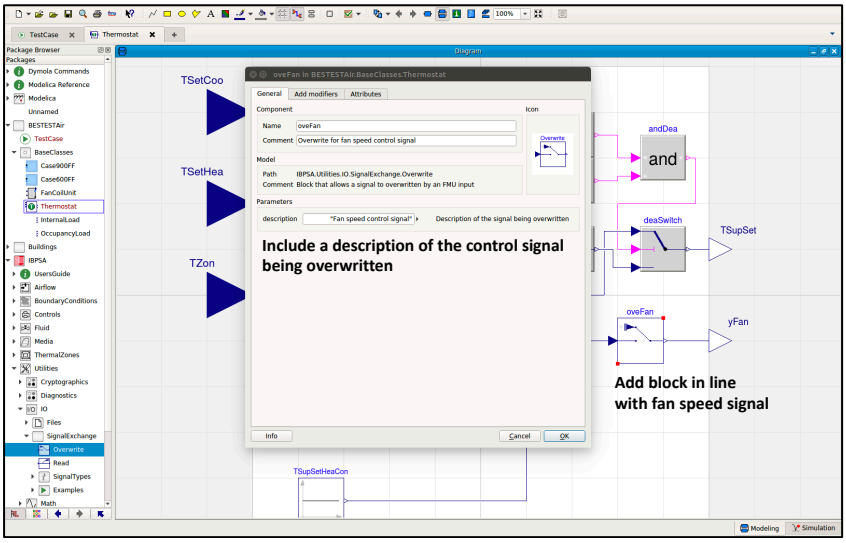
2. Add and Configure Signal Exchange Blocks

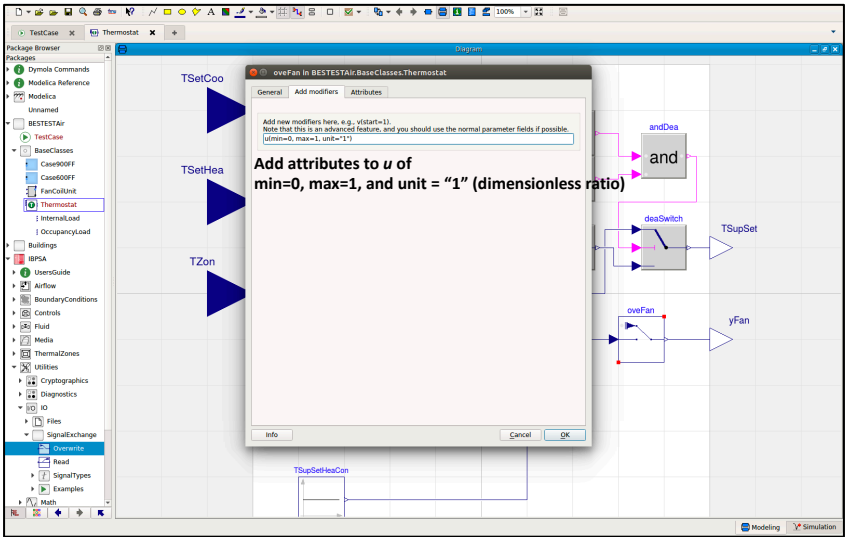
This step will modify the model we've just explored to enable it for controller testing in BOPTEST. It will do so by adding Signal Exchange blocks that help identify model control signals that can be overwritten by an external controller,

model measurements that can be read by an external controller, and model measurements that are needed for BOPTEST to calculate KPIs.

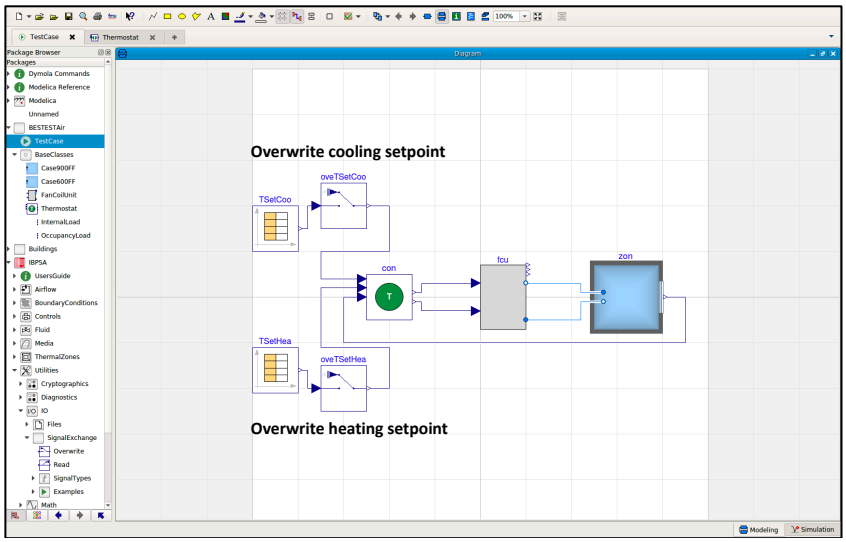
A. In Dymola, load the Modelica IBPSA Library.

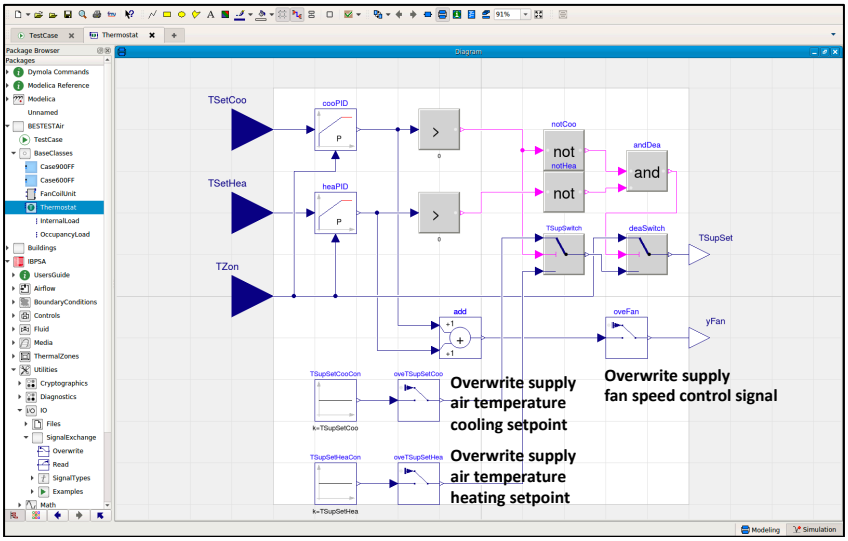
B. Use `IBPSA.Utilities.IO.SignalWrite` to implement and configure signal exchange blocks for each control signal we'd like to be able to **overwrite with an external controller**. Configuration is done through editing the parameters of the block and defining min, max, and unit attributes to the block output. An example is shown below for `con.yFan`. Then, repeat this step for the variables in the table below.





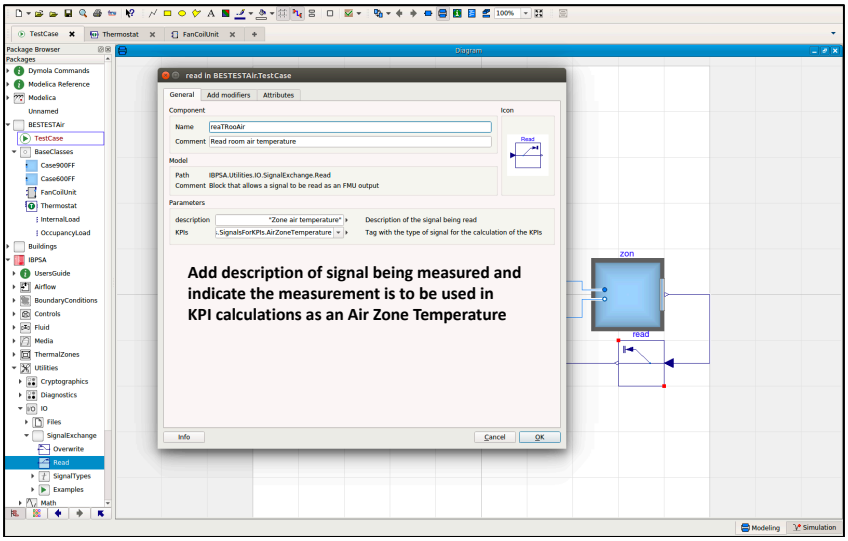
Variable	Description	Min	Max	Unit
`con.yFan`	Fan speed control signal	0	1	“1”
`con.TSupSwitch.u1`	Supply air temperature setpoint for cooling	273.15+12	273.15+18	“K”
`con.TsupSwitch.u3`	Supply air temperature setpoint for heating	273.15+30	273.15+40	“K”
`con.TsetCoo`	Zone temperature setpoint for cooling	273.15+23	273.15+30	“K”
`con.TsetHea`	Zone temperature setpoint for heating	273.15+15	273.15+23	“K”

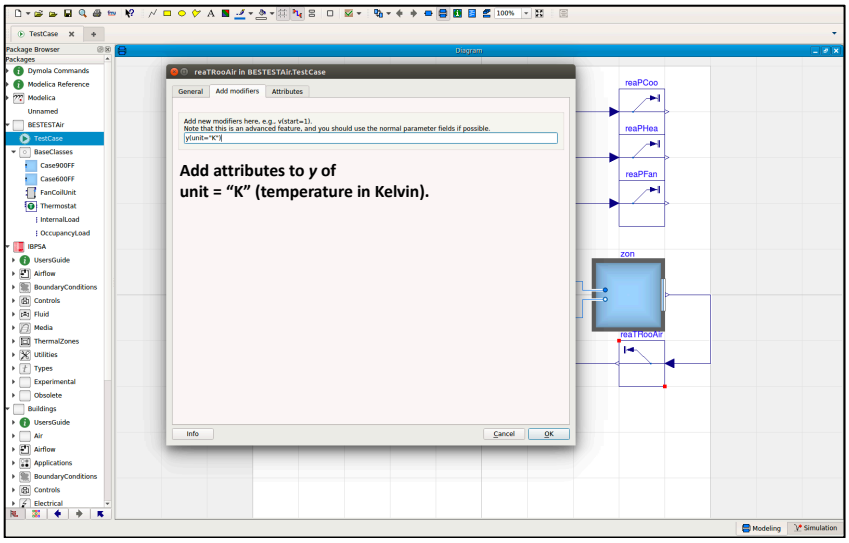




C. Use `IBPSA.Utilities.IO.SignalRead` to implement and configure signal exchange blocks for each measurement signal we’d like to be able to **read with an external controller**. Configuration is done through editing the parameters of the block and defining unit attributes to the block output. One of these parameters can be used to identify the measurement as being

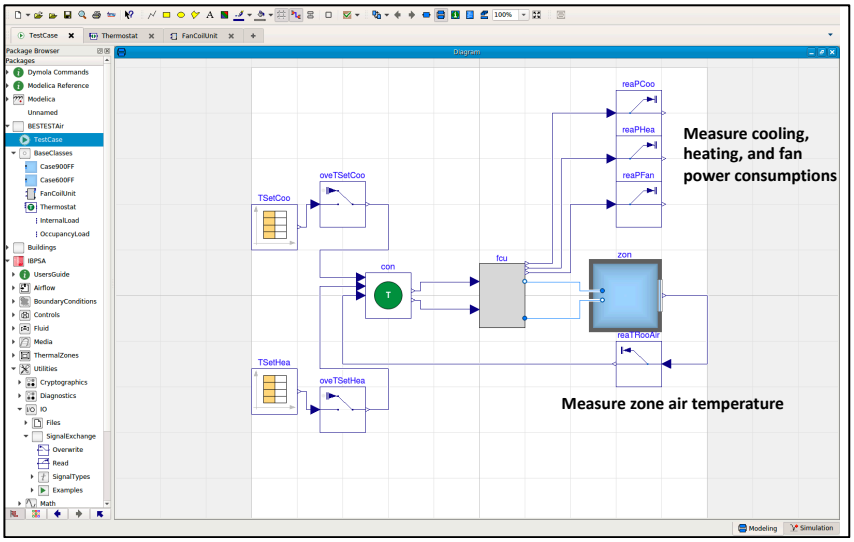
needed for KPI calculation. An example is shown below for `zon.TrooAir`. Then, repeat this step for the variables in the table below.

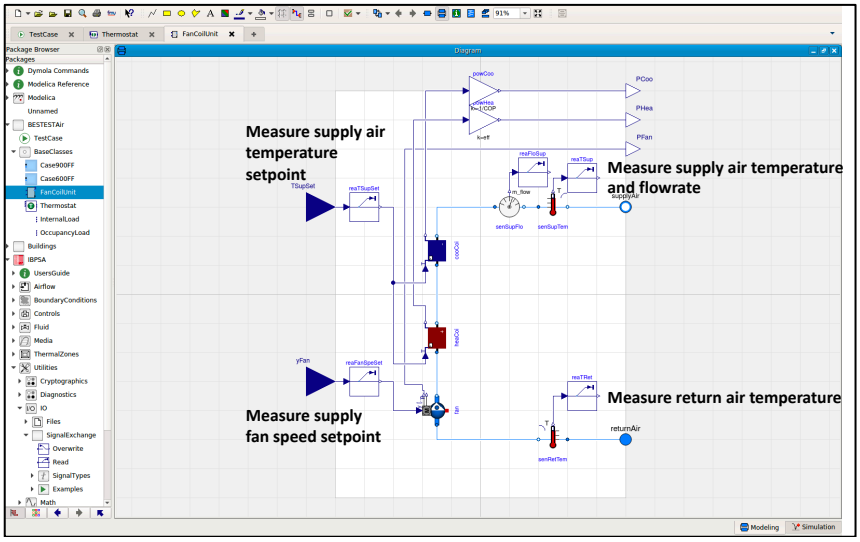


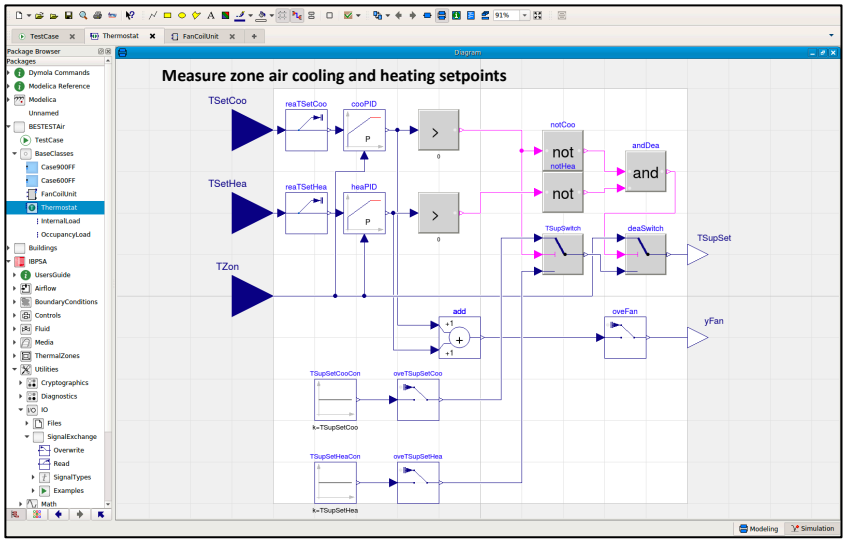


Variable	Description	KPI	Unit
`zon.TrooAir`	Zone air temperature	Air Zone Temperature	"K"
`fcu.Pcoo`	Cooling electrical power consumption	Electric Power From Grid	"W"
`fcu.Phea`	Heating thermal power consumption	Thermal Power From Natural Gas	"W"
`fcu.Pfan`	Supply fan electrical power consumption	Electric Power From Grid	"W"
`fcu.senSupFlo.m_flow`	Supply air mass flowrate	None	"kg/s"
`fcu.senSupTem.T`	Supply air temperature	None	"K"
`fcu.senRetTem.T`	Return air temperature	None	"K"
`fcu.yFan`	Supply fan speed setpoint	None	"1"

`fcu.TsupSet`	Supply air temperature setpoint	None	“K”
`con.TsetCoo`	Zone air temperature setpoint for cooling	None	“K”
`con.TsetHea`	Zone air temperature setpoint for heating	None	“K”







3. Create Boundary Condition Data

This step will create CSV files that represent the necessary boundary condition data to run the test case and calculate KPIs, such as weather, internal load schedules, energy and fuel prices, and occupant comfort ranges. A Python module has been created to help with this process, `data/data_generator.py`. However, the CSV files can be created in any way if more customization is needed, as long as their column names adhere to the naming conventions defined by BOPTEST and they are saved in the appropriate Resources directory.

A. Create the directory `testcase_bestest_air/models/Resources`, which is used to store the boundary condition data.

B. Copy the weather file used for the test case into the new Resources directory. The weather file is located in the Modelica Buildings Library at `Buildings/Resources/weatherdata/DRYCOLD.mos`.

C. Save the Python script below to `testcase_bestest_air/models/generate_data.py`. The script generates the data using the `data/data_generator.py` module at an interval of 15 minutes. Note that the function arguments are based on the building model we explored earlier. We will use this module to create the following test case data:

a. Weather
b. Energy and fuel prices
c. Energy and fuel emission factors
d. Occupancy
e. Internal heat gains
f. Thermal comfort setpoint ranges

```
from data.data_generator import Data_Generator
import os

# Set the location of the Resource directory relative to this file location
file_dir = os.path.dirname(os.path.realpath(__file__))
resources_dir = os.path.join(file_dir, 'Resources')
# Create data generator object with time interval to 15 minutes
gen = Data_Generator(resources_dir, period=900)
# Generate weather data from .mos in Resources folder with default values
gen.generate_weather()
# Generate prices data with default values
gen.generate_prices()
# Generate emission factors data with default values
gen.generate_emissions()
# Generate occupancy data for our case
gen.generate_occupancy(2,
                       start_day_time = '08:00:00',
                       end_day_time = '18:00:00')
# Generate internal gains data for our case
gen.generate_internalGains(start_day_time = '08:00:00',
                           end_day_time   = '18:00:00',
                           RadOcc = 10.325*48,
                           RadUnocc = 0.85*48,
                           ConOcc = 7.71667*48,
                           ConUnocc = 0.65*48,
                           LatOcc = 1.875*48,
                           LatUnocc = 0*48)
# Generates comfort range data for our case
gen.generate_setpoints(start_day_time = '08:00:00',
                       end_day_time = '18:00:00',
                       THeaOn  = 21+273.15,
                       THeaOff = 15+273.15,
                       TCooOn  = 24+273.15,
                       TCooOff = 30+273.15)
```

The function `gen.generate_weather()` requires the simulation of an FMU (specifically, the model `IBPSA.BoundaryConditions.WeatherData.ReaderTMY3`) using the `pyFMI` package. So that a user does not need to install this, we will use a Docker image that has been developed to run the `generate_data.py` script.

D. First, we must build the Docker image we will use to generate the test case data. To do this, change the working directory to `testing/`. Then, use the following command:

```
$ make build_jm_image
```

This may take a minute or two. Check that the image built successfully by the command:

```
$ docker images
```

There should be a listed image called `jm`.

E. Now, to generate the data, change the working directory to `testing/` and use the following command:

```
$ make generate_testcase_data TESTCASE=testcase_bestest_air
```

Note: that if an error occurred during generation of the test case data, before making any fixes and trying the above command again, run the following command:

```
$ docker stop jm
```

F. After running this command successfully, there should be six new files within `testcase_bestest_air/models/Resources`: `emissions.csv`, `prices.csv`, `occupancy.csv`, `weather.csv`, `ingernalGains.csv`, and `setpoints.csv`. Each csv file should have a time column to define the emulation time in seconds at 15 minute (900 second) intervals and the appropriate column header names according to the naming conventions defined by BOPTEST.

4. Compile Test Case FMU

This final step will compile the building model with signal exchange blocks and the boundary condition data into our test case FMU, which will be ready for us to use for controller testing! Since the run-time environment Docker container has a Linux-based OS (Ubuntu), the test case FMU needs to be compiled with binaries suitable for simulation in Linux environments. Therefore, a Docker image and process has been developed for Windows users to be able to compile the test case FMU. Linux/MacOS users can also use this same process with Docker, or compile the test case FMU in a native environment. Such compilation requires the `parsing/parser.py` module and the JModelica.org open source Modelica compiler to be installed, though these will not be detailed in this tutorial.

A. Create a Python module as below and save it as `testcase_bestest_air/models/compile_fmu.py`.

```
from parsing import parser

def compile_fmu():
    '''Compile the fmu.

    Returns
    -------
    fmupath : str
        Path to compiled fmu.

    '''

    # DEFINE MODEL
    mopath = 'BESTESTAir/package.mo'
    modelpath = 'BESTESTAir.TestCase'

    # COMPILE FMU
    fmupath = parser.export_fmu(modelpath, [mopath])

    return fmupath

if __name__ == "__main__":
    fmupath = compile_fmu()
```

B. Build the Docker image we will use to compile the test case FMU (note this is the same image we used in Part II.3 to generate test data, and does not need to be re-done if it was already built there). Change the working directory to `testing/`. Then, use the following command:

```
$ make build_jm_image
```

Check that the image built successfully by the command:

```
$ docker images
```

There should be a listed image called `jm`.

C. To compile the test case FMU, change the working directory to `testing/` and use the following command:

```
$ make compile_testcase_model TESTCASE=testcase_bestest_air
```

Once complete, there should be a new file `testcase_bestest_air/models/wrapped.fmu`, which is the test case FMU.

Note: that if an error occurred during compilation of the test case, before making any fixes and trying the above command again, run the following command:

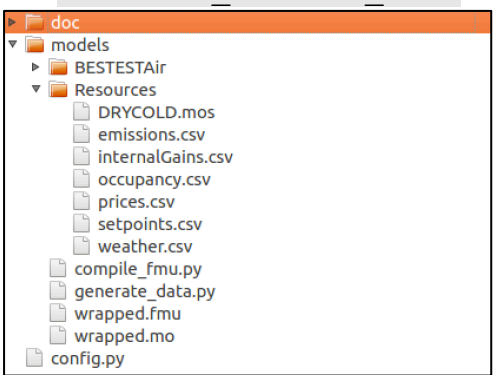
```
$ docker stop jm
```

D. Create the file `testcase_bestest_air/config.py` as shown below. This file holds configuration parameters for when we deploy and use the test case in the next section.

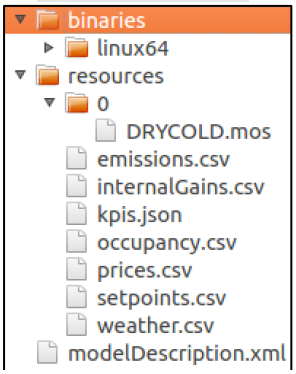
```
def get_config():
    '''Returns the configuration structure for the test case.

    Returns
    -------
    config : dict()
    Dictionary containing configuration information.
    {
    'fmupath'  : string, location of model fmu
    'step'     : int, default control step size in seconds
    'horizon'  : int, default forecast horizon in seconds
    'interval' : int, default forecast interval in seconds
    }

    '''

    config = {
    # Enter configuration information
    'fmupath'  : 'models/wrapped.fmu',
    'step'     : 3600,
    'horizon'  : 86400,
    'interval' : 3600
    }

    return config
```

E. The final step in creating the test case is to add documentation. Create a directory `testcase_bestest_air/doc` to hold any documentation. We will hold off on creating content to fill to this directory for this tutorial. The final test case directory `testcase_bestest_air/` and test case FMU (`wrapped.fmu` in this case) that will be needed for deployment in the next section looks like (it is ok if there are extra files from FMU compilation):

`testcase_bestest_air/`



`wrapped.fmu`



Part III: Test Case Interaction

This part will step through how to deploy the software run-time environment with the test case, explore the ways in which we can interact with the test case using the HTTP RESTful API, and perform a simple example controller comparison test.

1. Deploy Test Case

This step will use Docker to deploy the test case we have just created.

A. First, we need to build a test case Docker image for our run-time environment using our test case directory. Change the working directory to the root of the BOPTEST repository and use the following command:

```
$ make build TESTCASE=testcase_bestest_air
```

Confirm the test case image was built successfully by using the following command and seeing that there is an image called `boptest_testcase_bestest_air` listed:

```
$ docker images
```

B. Then, deploy the test case Docker container using this image with the following command from the root of the BOPTEST repository:

```
$ make run TESTCASE=testcase_bestest_air
```

The test case is successfully running if you see the following message in your terminal window:

```
 * Serving Flask app "restapi" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
INFO:werkzeug: * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

C. At this point, the test case is waiting for more instructions by listening to the `localhost:5000` port for HTTP RESTful API commands.

2. Using the HTTP RESTful API

The HTTP RESTful API commands for the test case are displayed in the Figure below. Let's create a Python script to demonstrate some interactions with our test case. We will use the `requests` package for this. In general, the test case can be utilized by any software that can send HTTP requests, we are only using a script written in Python here as an example.

Interaction	Request
Advance simulation with control input and receive measurements	POST `advance` with json data "{<input_name>:}"
Reset simulation to beginning	PUT `reset` with no argument
Receive communication step in seconds	GET `step`
Set communication step in seconds	PUT `step` with argument `step=<value>`
Receive sensor signal names (y) and metadata	GET `measurements`
Receive control signals names (u) and metadata	GET `inputs`
Receive test result data	GET `results`
Receive test KPIs	GET `kpi`
Receive test case name	GET `name`
Receive boundary condition forecast from current communication step	GET `forecast`
Receive boundary condition forecast parameters in seconds	GET `forecast_parameters`
Set boundary condition forecast parameters in seconds	PUT `forecast_parameters` with arguments `horizon=<value>`, `interval=<value>`

A. Create a new file from the code below in the root of the BOPTEST repository called `api_example.py`. Then, run the code and analyze the output. Try running the simulation for longer and/or plotting the results.

```
import requests
import pprint
pp = pprint.PrettyPrinter(indent=4)

# Set URL for testcase
url = 'http://127.0.0.1:5000'


# DEMONSTRATE API FOR GETTING TEST CASE INFORMATION
# -------------------------------------------
pp.pprint('\nTEST CASE INFORMATION\n----------------------')
# Get the test case name
name = requests.get('{0}/name'.format(url)).json()
print('\nName is:')
pp.pprint(name)
# Inputs available
inputs = requests.get('{0}/inputs'.format(url)).json()
print('\nControl Inputs are:')
pp.pprint(inputs)
# Measurements available
measurements = requests.get('{0}/measurements'.format(url)).json()
print('\nMeasurements are:')
pp.pprint(measurements)
# Default simulation step
step_def = requests.get('{0}/step'.format(url)).json()
print('\nDefault Simulation Step is:\t{0}'.format(step_def))

# DEMONSTRATE API FOR SIMULATING TEST CASE
# ----------------------------------------
# Set simulation step
print('\nSetting simulation step to 3600.')
res = requests.put('{0}/step'.format(url), data={'step':3600})
print(res)
# Advance simulation
print('\nAdvancing emulator one step...')
y = requests.post('{0}/advance'.format(url), data={}).json()
print('\nMeasurements at next step are:')
pp.pprint(y)
# Get a forecast of boundary conditions
print('\nAdvancing emulator one step...')
w = requests.get('{0}/forecast'.format(url)).json()
print('Forecast is:')
pp.pprint(w)

# DEMONSTRATE API FOR VIEWING RESULTS
# -----------------------------------
# Get all simulation results
data = requests.get('{0}/results'.format(url)).json()
print('\nSimulation results are:')
pp.pprint(data)
# Report KPIs
kpi = requests.get('{0}/kpi'.format(url)).json()
print('\nKPIs are:')
pp.pprint(kpi)

# Reset test case to beginning
print('\nResetting test case to beginning.')
res = requests.put('{0}/reset'.format(url))
print(res)
```

3. Example Controller Test

Let's now edit the api_example.py script developed in the previous section to simulate the model for 1 day (24 steps of 3600 seconds) and implement control signals to overwrite those within the model. We will implement new zone heating and cooling setpoint temperatures that are constant and make a narrower deadband range. Save the Python code below to a file named control_example.py and run it.

Feel free to edit control signals or parameters to explore how the KPIs and results change. Try implementing a feedback controller using the measurements that are taken at each timestep. Remember that stability may require the use of a smaller control time step.

```
import requests
import pprint
pp = pprint.PrettyPrinter(indent=4)

# Set URL for testcase
url = 'http://127.0.0.1:5000'


# DEMONSTRATE API FOR GETTING TEST CASE INFORMATION
# -------------------------------------------
pp.pprint('\nTEST CASE INFORMATION\n---------------------')
# Get the test case name
name = requests.get('{0}/name'.format(url)).json()
print('\nName is:')
pp.pprint(name)
# Inputs available
inputs = requests.get('{0}/inputs'.format(url)).json()
print('\nControl Inputs are:')
pp.pprint(inputs)
# Measurements available
measurements = requests.get('{0}/measurements'.format(url)).json()
print('\nMeasurements are:')
pp.pprint(measurements)
# Default simulation step
step_def = requests.get('{0}/step'.format(url)).json()
print('\nDefault Simulation Step is:\t{0}'.format(step_def))

# DEMONSTRATE API FOR SIMULATING TEST CASE
# ----------------------------------------
# Set simulation step
print('\nSetting simulation step to 3600.')
res = requests.put('{0}/step'.format(url), data={'step':3600})
print(res)
# Advance simulation in loop for test
for i in range(24):
    u = {'oveTSetHea_u':273.15+22, # Set zone heating setpoint to 22 C
         'oveTSetHea_activate':1, # Activate overwrite of zone heating setpoint
         'oveTSetCoo_u':273.15+23, # Set zone cooling setpoint to 23 C
         'oveTSetCoo_activate':1 # Activate overwrite of zone cooling setpoint
         }
    print('\nAdvancing emulator one step with control data...')
    y = requests.post('{0}/advance'.format(url), data=u).json()
    print('\nMeasurements at next step are:')
    pp.pprint(y)
    # Get a forecast of boundary conditions
    print('\nAdvancing emulator one step...')
    w = requests.get('{0}/forecast'.format(url)).json()
    print('Forecast is:')
    pp.pprint(w)

# DEMONSTRATE API FOR VIEWING RESULTS
# -----------------------------------
# Get all simulation results
data = requests.get('{0}/results'.format(url)).json()
print('\nSimulation results are:')
pp.pprint(data)
# Report KPIs
kpi = requests.get('{0}/kpi'.format(url)).json()
print('\nKPIs are:')
pp.pprint(kpi)

# Reset test case
print('\nResetting test case to beginning.')
res = requests.put('{0}/reset'.format(url))
print(res)
```