Advanced Controls Test Bed
User Manual

Gregor P. Henze[1,2,4] Thibault Marzullo[1] Nicholas L. Long[2]
José A. Leiva Vilaplana[1,3] Sourav Dey[1]

November 19, 2021

[1] University of Colorado Boulder, U.S.A.
[2] National Renewable Energy Laboratory, Golden (CO), U.S.A.
[3] Universitat Politècnica de Catalunya, Barcelona, Spain.
[4] Renewable and Sustainable Energy Institute, Boulder (CO), U.S.A.

Abstract

The Advanced Controls Test Bed (ACTB) is a building performance simulation test bed designed for developing advanced controllers, such as model predictive control (MPC) or reinforcement learning (RL), for the efficient control of building systems. The ACTB relies on Spawn of EnergyPlus (or Spawn) for high-fidelity models, and on BOPTEST/Alfalfa for simulation management and controller benchmark. This document describes the prerequisites, installation, and usage of the ACTB from model design to controller evaluation. The user will be capable of generating a high-fidelity Spawn model, designing a simple controller, configuring the simulation and generating results. The manual also presents a practical application, the design and control of a Spawn reference Small Office Building.

1 Introduction

The ACTB is designed to provide a flexible test bed for model designers, control engineers, HVAC manufacturers and researchers to develop advanced control sequences. This guide therefore covers the following topics in distinct sections:

- Installing the ACTB and its prerequisites
- Designing a Spawn model
- Preparing the model for BOPTEST
- Loading and simulating a model in the ACTB
- Preparing an external controller script
- Generating results
- Using the OpenAI Gym interface
- Using the DO-MPC interface

2 Framework

The ACTB's framework is presented in Figure 1. The ACTB builds on BOPTEST, a framework for the standardized benchmark of control sequences for building systems. BOPTEST and its underlying Alfalfa components are used as the simulation engine for the ACTB, providing simulation management capabilities and acting as an interface between external controllers and building models. The simulation engine can theoretically[1] simulate any model as long as it is packaged in a FMU[2] format. The controller

[1]Currently, the ACTB has only been tested with Modelica-based FMUs.

[2]Alfalfa also accepts OpenStudio Workflows (.osw) and EnergyPlus models (.idf) but these features are not yet supported in the ACTB

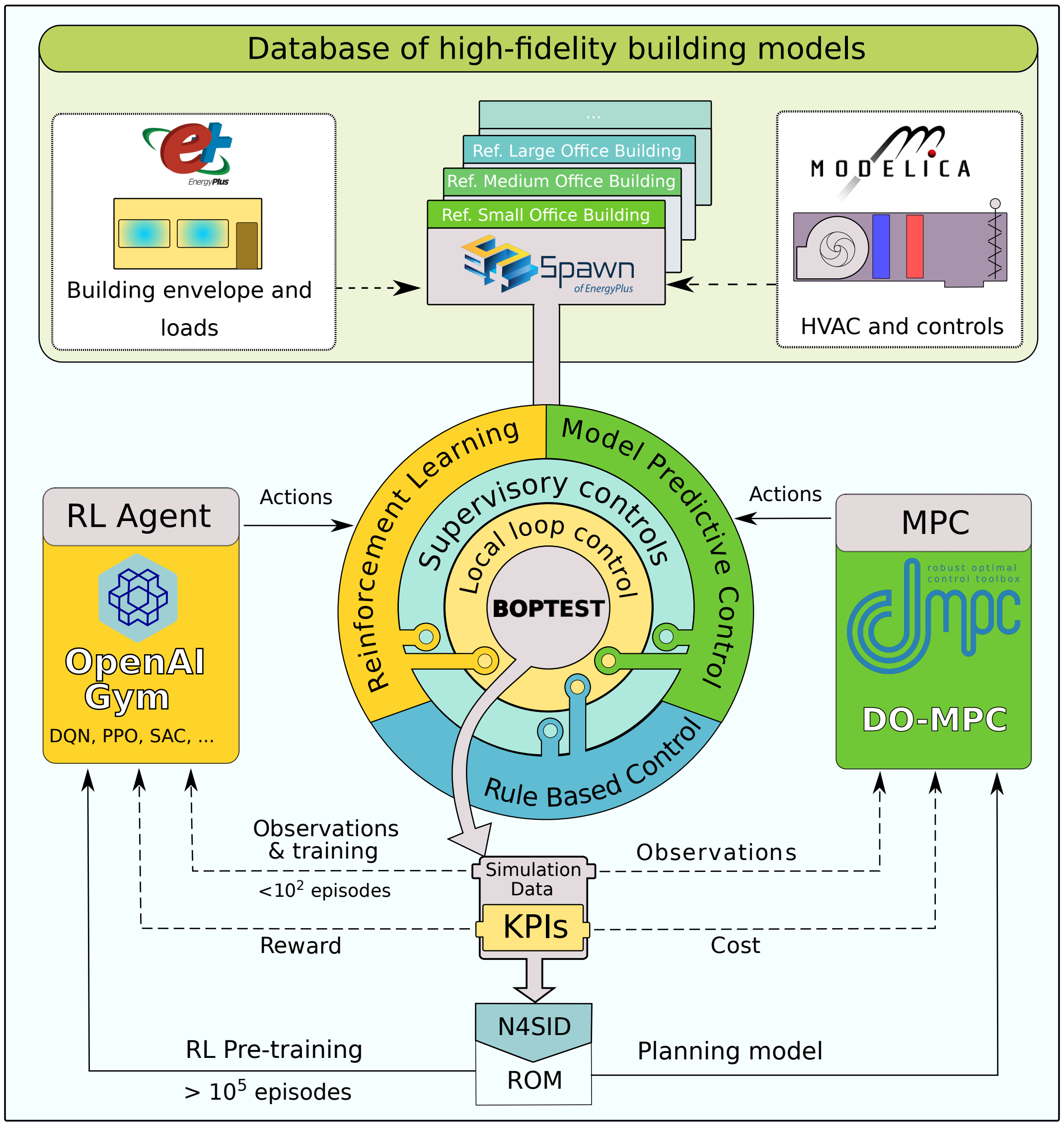


Figure 1: Graphical description of the ACTB framework.

design is not constrained, the ACTB can read sensors and overwrite controls using the real-world equivalent of software override points as specified by the user. The only constraint is that the data must be exchanged as floating point variables, no other data types are currently supported. The ACTB uses HTTP requests for communicating with the simulation engine, making the framework both operable remotely (e.g. cloud service) and agnostic to the user's programming language. The ACTB and BOPTEST include controller examples both in Python and Julia. Section 4.2 describes the design of a high-fidelity Spawn model, as shown on the top third of Figure 1, Section 5.2 describes the usage of a general-purpose controller and Section 6 introduces the OpenAI Gym and DO-MPC interfaces.

3 Installation

3.1 Pre-requisites

The pre-requisites for installing the ACTB are the following:

- A 64-bit UNIX machine;

- Python 2.7+, 3.6+ preferred;
- Git, available here:
- Docker;

Optionally, the user might install the following packages:

- OpenAI Gym, a machine learning framework available here;
- DO-MPC, a robust-MPC framework available here;
- SIPPy, a system identification package available here;
- A Modelica IDE and simulation engine for model development.[3]

3.2 Installing the ACTB

Simply clone the latest version of the ACTB from GitHub.

```
cd <install_dir>
git clone https://github.com/henze-research-group/MODRLC
```

Run tests to verify that the installation is working correctly:

```
cd <install_dir>/MODRLC/testing
python %\todo{cleanup and add the test script}
```

If tests do not complete successfully, please make sure that Docker is installed correctly, and that you are using the right version of Python.

3.3 Project structure

You will find the following folder tree in the install directory:

```
MODRLC
 ├─ boptest_client
 ├─ data
 ├─ forecasts
 ├─ examples
 ├─ kpis
 ├─ parsing
 ├─ template
 ├─ testcases
 ├─ testing
 ├─ web
 └─ worker
```

The `examples` and `testcases` folders contain the provided example controllers and available test cases, respectively. The `template` folder contains a Spawn model template. Other folders include BOPTEST resources and utilities (data, forecasts, kpis, parsing, testing) and Alfalfa source-code (web, worker). The user is not required to know or modify any of these files, their usage is not covered in this manual, but more information can be found on BOPTEST's and Alfalfa's web pages.

[3]This manual shows examples using Dassault Systèmes' Dymola 2021.

3.4 Quick start

This section will demonstrate how to start the ACTB and run a sample simulation of the Reference Small Office Building. The model files are located in the `testcases\smallofficechicago` folder and the example simulation script is in the `examples\python` folder. To start the worker, open a terminal session and `cd` to the root of your ACTB directory. Then, execute:

```
docker-compose up web
```

When all services have started, `cd` to the `examples\python` folder of your ACTB install and run the example simulation scriptOn first start, the worker will build all Docker containers, which could take some time.:

```
python sim_smalloffice.py
```

The script will upload the test case FMU to the worker, initialize the simulation, and run a simple supervisory control loop that simulates a demand-response event.

4 Developing a test case

This section provides guidance for the design, resource generation and compilation of a Spawn building model, prepared for simulation in the ACTB. It will also cover the usage of MBL'sModelica Buildings Library `signalExchange` blocks to provide sensor and control points that will be used to interface the model to an external controller.

4.1 Pre-requisites

This section will describe the development of a Spawn building model. The user should therefore:

- have good knowledge of the Modelica equation-based language;
- have an IDE capable of simulating models from the MBL;
- have selected an EnergyPlus building model to enhance;
- have installed and tested the ACTB.

4.2 Developing a model

The ACTB is provided with a sample test case, a *Spawn of EnergyPlus* model. *Spawn* builds on the *EnergyPlusToFMU* program, which is a utility for packaging an EnergyPlus model into an FMU, or Functional Mock-Up unit[4]. *Spawn* can simulate an EnergyPlus envelope and loads model inside the Modelica environment, using model exchange, allowing the designer to model building systems and their control loops in Modelica. The model can be exported in a portable FMU and then simulated in an environment like the ACTB, where it will use the EnergyPlus simulation engine to provide envelope and load calculations, and the Modelica simulation engine to accurately simulate the desired building systems.

4.2.1 The building model

Spawn is an ongoing project, and some features are missing or need revision. The current shortcomings that ACTB developers are aware of are listed below. Issues marked with * are planned to be solved in future Spawn updates.

- Spawn becomes unresponsive if the model is simulated beyond Dec. 31st of the simulated year. *

[4]The Functional Mock-Up unit is a standard format for model exchange and co-simulation built on the FMI standard, read more about it here.

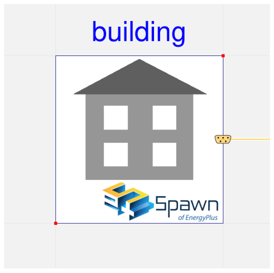


Figure 2: The *building* component.

- Portable Spawn FMUs do not have a consistent folder structure, the user must post-process the model to ensure directories are correct. *
- Spawn thermal zones do not warm-up, the routine is interrupted after the first iteration, before the solution has converged.
- Relative paths are changed to absolute paths when specifying model resources for a portable FMU[5].

Spawn is included as a component of the Modelica Buildings Library v9.0 (MBL). Spawn comes as a two-part component, found under *Buildings/Thermal Zones/EnergyPlus*: a *building* and a series of *thermal zones*. The *building* component shown in Figure 2 is used to specify the following model parameters:

- the path to the EnergyPlus building model (.idf),
- the path to the weather file (.mos),
- the portability of the Spawn model[6],
- whether the component also provides a weather bus, from the specified weather file,
- the verbosity of the EnergyPlus simulation.

Figures 3 through 5 show the building component's *General*, *Debug* and *Advanced* tabs respectively. In the General tab, when specifying the paths to the model and weather files, the user should pay close attention to not filling the field with a string, but rather with:

```
Modelica.Utilities.Files.loadResource("path/to/file")
```

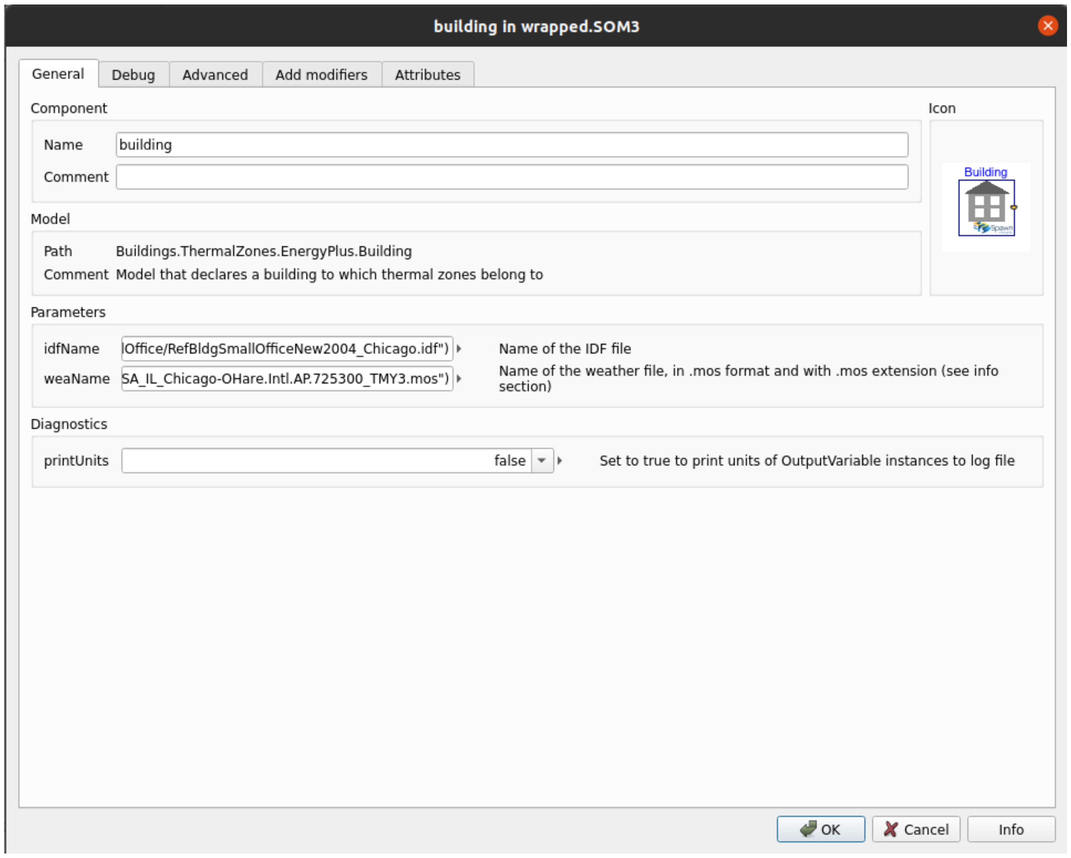
Where "path/to/file" can be an absolute path or a URI in the form:

```
/absolute/path/to/file
relative/path/to/file
modelica://path/within/modelica/libraries
```

After the user has parametrized the *building* component, they can proceed to defining thermal zones. Spawn includes the *ThermalZone* component, which model the zone as a mixing volume with fluid and heat ports that can be connected to the rest of the Modelica model. The user must pay attention to specifying the correct zone name in the `zoneName` field. This name is case-sensitive and must match the zone name in the .idf file. The initialization currently has no effect as the building envelope is not warmed-up, as explained above.

The user can generate as many `ThermalZone`s as necessary, and it is not required to model all the thermal zones originally present in the .idf. The `ThermalZone`s provide a variety of fluid and heat ports, inputs and outputs, and the user is encouraged to familiarize themselves with all the exposed

[5]this is not strictly a Spawn issue
[6]the ACTB requires a portable Spawn FMU.


building in wrapped.SOM3

General | Debug | Advanced | Add modifiers | Attributes

Component

Name: building

Comment:

Model

Path: Buildings.ThermalZones.EnergyPlus.Building

Comment: Model that declares a building to which thermal zones belong to

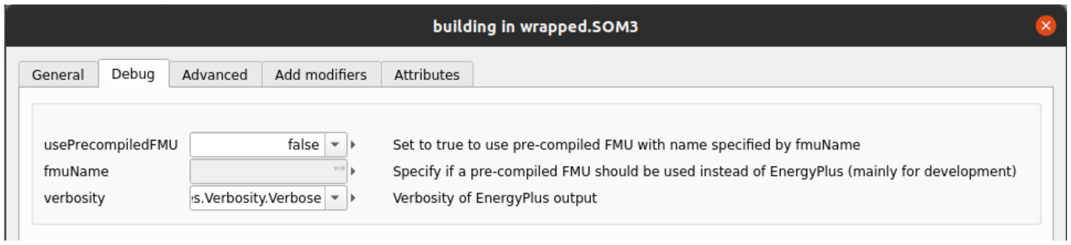
Parameters

idfName	lOffice/RefBldgSmallOfficeNew2004_Chicago.idf")	Name of the IDF file
weaName	SA_IL_Chicago-OHare.Intl.AP.725300_TMY3.mos")	Name of the weather file, in .mos format and with .mos extension (see info section)

Diagnostics

printUnits	false	Set to true to print units of OutputVariable instances to log file

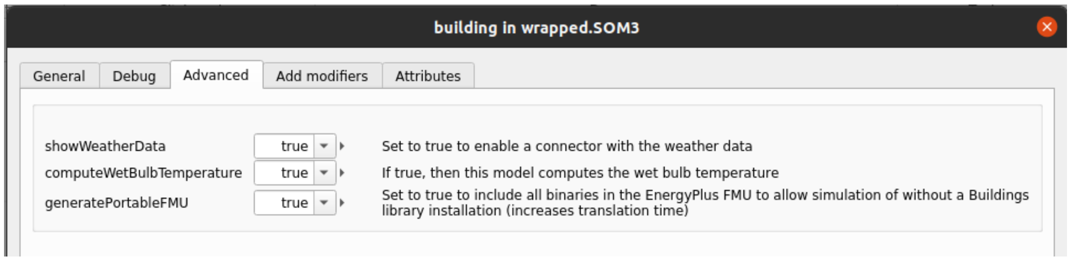
Figure 3: Dymola Graphical User Interface showing the *building*'s "General" tab, where the user must specify the path to the .idf model and the .mos weather file.


building in wrapped.SOM3

General | Debug | Advanced | Add modifiers | Attributes

usePrecompiledFMU	false	Set to true to use pre-compiled FMU with name specified by fmuName
fmuName	""	Specify if a pre-compiled FMU should be used instead of EnergyPlus (mainly for development)
verbosity	:s.Verbosity.Verbose	Verbosity of EnergyPlus output

Figure 4: The *building*'s "Debug" tab.


building in wrapped.SOM3

General | Debug | Advanced | Add modifiers | Attributes

showWeatherData	true	Set to true to enable a connector with the weather data
computeWetBulbTemperature	true	If true, then this model computes the wet bulb temperature
generatePortableFMU	true	Set to true to include all binaries in the EnergyPlus FMU to allow simulation of without a Buildings library installation (increases translation time)

Figure 5: The *building*'s "Advanced" tab, where the user can specify if they want to be able to generate a portable FMU or expose a weather bus from the file specified in the "General" tab

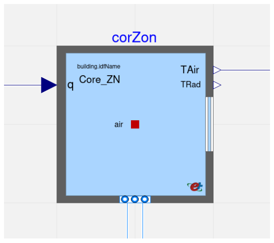
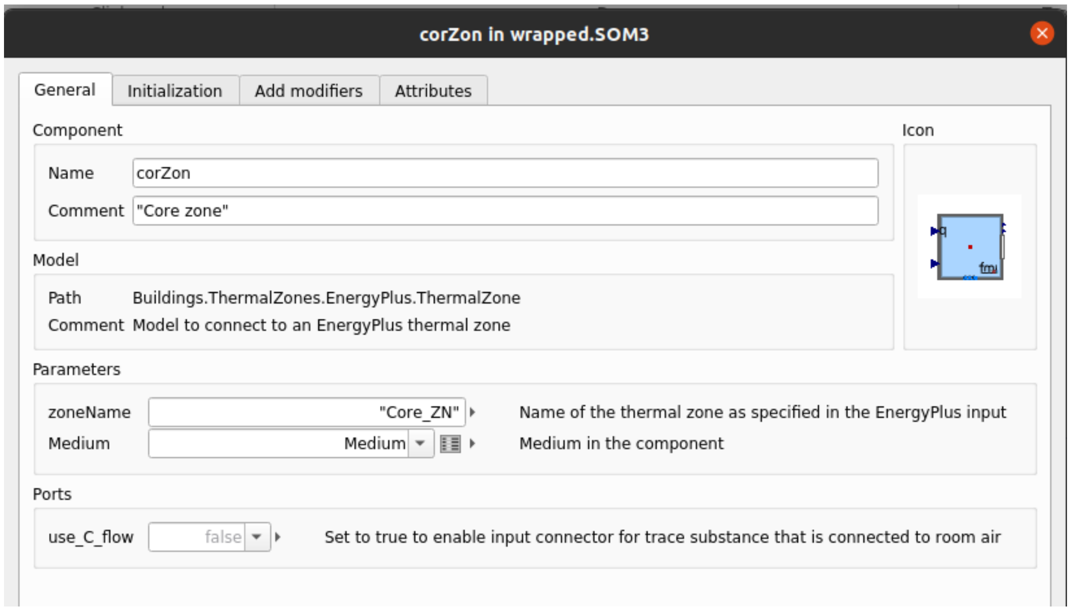


Figure 6: The *ThermalZone* component


corZon in wrapped.SOM3

General | Initialization | Add modifiers | Attributes

Component

Name	corZon
Comment	"Core zone"

Icon

Model

Path	Buildings.ThermalZones.EnergyPlus.ThermalZone
Comment	Model to connect to an EnergyPlus thermal zone

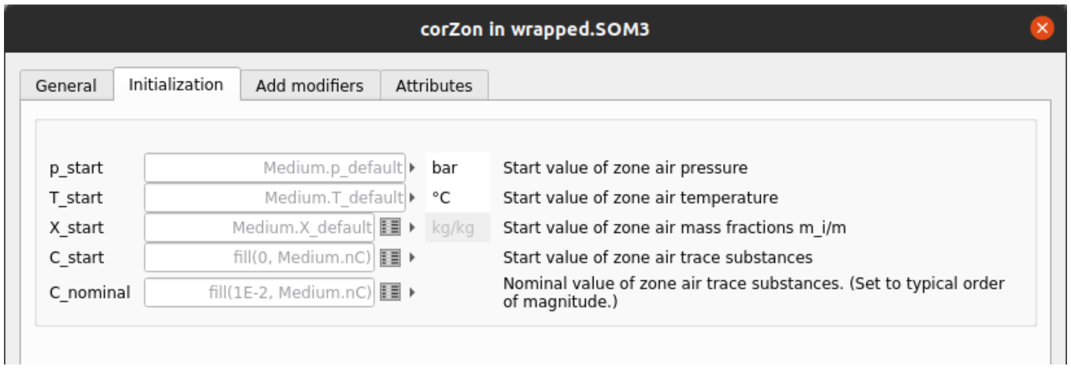
Parameters

zoneName	"Core_ZN"	Name of the thermal zone as specified in the EnergyPlus input
Medium	Medium	Medium in the component

Ports

use_C_flow	false	Set to true to enable input connector for trace substance that is connected to room air

Figure 7: The "General" tab of the *ThermalZone* component, showing the zoneName field. The user must pay attention to using the same zone names as in the .idf file.


corZon in wrapped.SOM3

General | Initialization | Add modifiers | Attributes

p_start	Medium.p_default	bar	Start value of zone air pressure
T_start	Medium.T_default	°C	Start value of zone air temperature
X_start	Medium.X_default	kg/kg	Start value of zone air mass fractions m_i/m
C_start	fill(0, Medium.nC)		Start value of zone air trace substances
C_nominal	fill(1E-2, Medium.nC)		Nominal value of zone air trace substances. (Set to typical order of magnitude.)

Figure 8: The "Advanced" tab of the *ThermalZone* component, showing the zoneName field. While the fluid medium will be initialized to these values, the building envelope is not warmed-up as explained previously, hence the building's thermal mass will drive the building's temperature away from these initial values.

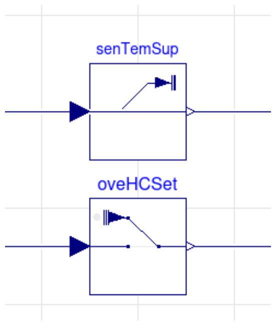


Figure 9: The *read* (top) and *write* (bottom) blocks

variables and ports in the component's Modelica description. The example provided with the ACTB, for instance, uses the `ThermalZone`'s heat port, fluid ports, and mean air temperature output. With the `building` and the `ThermalZone` components, the Modelica model can now exchange data with the EnergyPlus simulation engine.

Apart from `signalExchange` blocks, presented in the next section, the rest of the model is left to the user's imagination, as it will not differ from another standard Modelica model. Users are strongly encouraged to visit the Modelica libraries webpage to learn more about the possibilities offered by Modelica modeling.

4.2.2 signalExchange read and write blocks

In order to read and write to the Spawn model, the ACTB leverages signalExchange blocks. These components come in two flavors: the read and the write block. While both blocks are placed on a control line or on a sensor output, they differ in usage and functionality. *read* blocks, presented in Figure 9 , are used to expose sensor readings. Figure 10 shows how a *read* block is parametrized. The user should first select the "KPIs" drop-down menu, and specify whether the value that this block reads is used for KPI calculation or not. For some KPIs, the "zone" field should comport a string indicating which zone is being reported on. The "y" field specifies the output's unit and limits.

Similarly, the *write* block shown in Figure 9 is parametrized as shown in Figure 11. The user must only specify the unit and limits of the "u" input, which will be used to override the signal the block is placed onto. The *read* blocks include a hidden property, the **_activate** flag, which is used to choose whether the block overrides the signal or lets it pass through.

The user must pay attention to including the minimum and maximum values and the unit for each *read* and *write* block. In both blocks, the user should add a custom parameter that defines the upper and lower bounds of the input or output, and the unit. For *write* blocks, the user should include:

```
u(min=<min_value>, max=<max_value>, unit="<unit>")
```

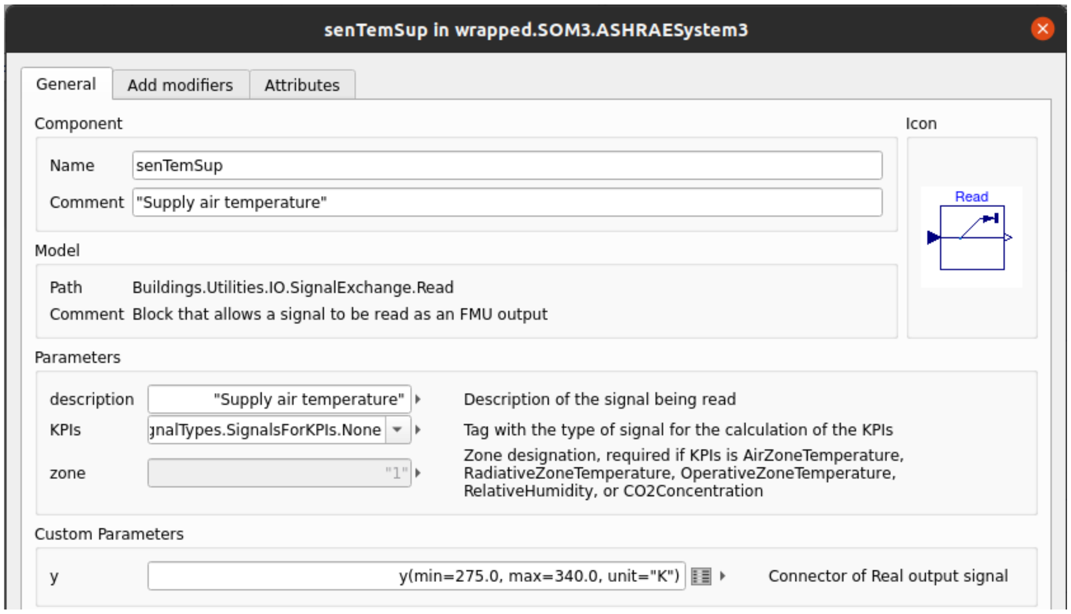
The full declaration for a sample *write* block is

```
Buildings.Utilities.IO.SignalExchange.Overwrite oveCooSetpoint(u(
        unit="K",
        min=0,
        max=310), description="Cooling setpoint override")
```

For *read* blocks, the user should include:

```
y(min=<min_value>, max=<max_value>, unit="<unit>")
```

The full declaration of a sample *read* block becomes:


senTemSup in wrapped.SOM3.ASHRAESystem3

General | Add modifiers | Attributes

Component

Name: senTemSup

Comment: "Supply air temperature"

Model

Path: Buildings.Utilities.IO.SignalExchange.Read

Comment: Block that allows a signal to be read as an FMU output

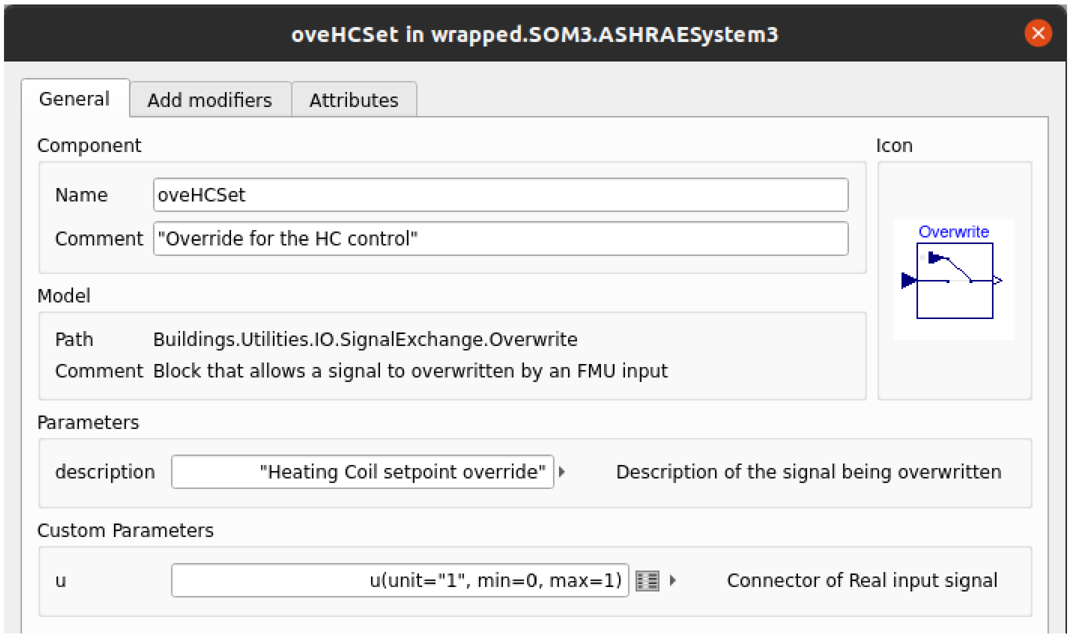
Parameters

description	"Supply air temperature"	Description of the signal being read
KPIs	ɡnalTypes.SignalsForKPIs.None	Tag with the type of signal for the calculation of the KPIs
zone	"1"	Zone designation, required if KPIs is AirZoneTemperature, RadiativeZoneTemperature, OperativeZoneTemperature, RelativeHumidity, or CO2Concentration

Custom Parameters

y	y(min=275.0, max=340.0, unit="K")	Connector of Real output signal

Figure 10: The *read* block's parameters


oveHCSet in wrapped.SOM3.ASHRAESystem3

General | Add modifiers | Attributes

Component

Name: oveHCSet

Comment: "Override for the HC control"

Model

Path: Buildings.Utilities.IO.SignalExchange.Overwrite

Comment: Block that allows a signal to overwritten by an FMU input

Parameters

description	"Heating Coil setpoint override"	Description of the signal being overwritten

Custom Parameters

u	u(unit="1", min=0, max=1)	Connector of Real input signal

Figure 11: The *write* block's parameters

```
Buildings.Utilities.IO.SignalExchange.Read senOcc(y(
min=0.0,
max=10.0,
unit="1"), description="occupancy sensor",
KPIs=Buildings.Utilities.IO.SignalExchange.SignalTypes.SignalsForKPIs.None)
```

4.3 Exporting the model and resources

Once the Modelica model has been designed and saved as a .mo file, the user should create a new folder in the "testcases" folder. The folder name should be in lower case characters, with no spaces. In that folder, the user should include model documentation in the "doc" folder and model files and resources in the "models" folder. Resources should go in a folder named "Resources" (case-sensitive). The user can use the template provided in the "template" folder, it contains model and resources templates, as well as the directory structure. After the directories have been generated, the user should see the following folder structure:

```
ACTB_root
└─testcases
  └─mymodelname
    ├─config.py
    ├─benchmark.py
    ├─library_versions.json
    ├─docs
    │ └─model_documentation.pdf
    └─models
      ├─mymodel.mo
      └─Resources
        ├─electricity_prices_dynamic.csv
        ├─emission_factors.csv
        └─...
```

In order to export the model, the user must update the following files:

- config.py: update the model step, forecast horizon, and energy pricing scenario
- models\compile_fmu.py: update `mopath` and `modelpath`.
- models\library_versions.json: update the hash of the MBL commit, if needed.
- models\resources\*: verify that all data has been generated correctly by the data_manager.py script.

Once the test case directory is ready, and all files have been updated to correspond to the current model, the model should be compiled into an FMU. To do that, `cd` into the `<ACTB_root_dir>\testing\` directory and run the following command:

```
sudo make compile_testcase_model TESTCASE=<testcase_folder_name>
```

If all resources are available, the script should complete and generate two files: `wrapped.mo` and `wrapped.fmu` , which are respectively the model wrapper that maps FMU inputs and outputs, and the self-contained FMU that will be used in the simulation manager. You can now move or copy the `wrapped.fmu` archive to a more convenient location, and later upload it to the ACTB.

4.4 Additional design considerations

TK include Spawn considerations, file paths explanation for FMU export, notes on Windows, etc.

5 Using the ACTB

This section of the user manual provides guidance for interfacing an external controller to a model that has already been prepared and compiled into an FMU.

5.1 Startup

In order to interact with the ACTB, its Docker container must be active, and running. First, start a terminal window[7] and `cd` to the root folder of the ACTB. To verify which containers are currently active, use:

```
docker container ls
```

If you already have active Docker containers, they will be listed there. The command to start the ACTB through Alfalfa depends on the number of simulation workers that the user needs. By using the `docker-compose` commands described below, the user will be launching BOPTEST-service, or the Alfalfa-based simulation manager version of BOPTEST. This is the intended usage of the ACTB, however users can decide to bypass Alfalfa and use only the BOPTEST portion. Users are invited to refer to BOPTEST's user guide available here in that case.

Single simulation worker:
To start a single simulation worker, `cd` to the ACTB root folder and use:

```
docker-compose up web
```

This is the terminal output that you should see if the service has started correctly:

```
user@machine:~$  sudo docker-compose up web
Starting modrlc_redis_1 ... done
Starting modrlc_minio_1 ... done
Starting modrlc_goaws_1 ... done
Starting modrlc_mongo_1 ... done
Starting modrlc_mc_1    ... done
Starting modrlc_worker_1 ... done
Starting modrlc_web_1    ... done
Attaching to modrlc_web_1
```

Please note that the container name prefix, here `modrlc` , is automatically selected to match that of the parent folder. The rest of the names are formatted in a standard way: *folder_service_ID*
To stop the Docker containers, first stop the process with Ctrl+C and use:

```
docker-compose down
```

You will see the Docker containers stopping:

```
user@machine:~$  sudo docker-compose down
Stopping modrlc_worker_1 ... done
Stopping modrlc_redis_1  ... done
Stopping modrlc_mongo_1  ... done
Stopping modrlc_goaws_1  ... done
Stopping modrlc_minio_1  ... done
Removing modrlc_web_1    ... done
Removing modrlc_worker_1 ... done
Removing modrlc_mc_1     ... done
Removing modrlc_redis_1  ... done
Removing modrlc_mongo_1  ... done
Removing modrlc_goaws_1  ... done
Removing modrlc_minio_1  ... done
Removing network modrlc_default
```

Multiple simulation workers:

[7]you may need elevated privileges for some commands, a common sign of insufficient privileges is a refused access or connection. Simply add "sudo" before the commands

To start multiple simulation workers, `cd` to the ACTB root folder and use:

```
docker-compose up --scale worker=<number_of_workers>
```

where `<number_of_workers>` is an integer number representing the number of simulation workers that are desired.

TK: Ask Nick about the fact that messages are queued, and delays are long, so the single HTTP queue seems to be slowing down the process quite a bit. Do several web containers solve the issue?

If the command is successful, you should see the Docker containers starting. In the following example, 7 workers are launched:

```
user@machine:~$  sudo docker-compose up --scale worker=7
Creating network "modrlc_default" with the default driver
Creating modrlc_minio_1 ... done
Creating modrlc_mongo_1 ... done
Creating modrlc_goaws_1 ... done
Creating modrlc_redis_1 ... done
Creating modrlc_mc_1    ... done
Creating modrlc_worker_1 ... done
Creating modrlc_worker_2 ... done
Creating modrlc_worker_3 ... done
Creating modrlc_worker_4 ... done
Creating modrlc_worker_5 ... done
Creating modrlc_worker_6 ... done
Creating modrlc_worker_7 ... done
Creating modrlc_web_1    ... done
Attaching to modrlc_minio_1, modrlc_mongo_1, modrlc_goaws_1, modrlc_redis_1,
modrlc_mc_1, modrlc_worker_2, modrlc_worker_4, modrlc_worker_3,
modrlc_worker_7, modrlc_worker_1, modrlc_worker_6, modrlc_worker_5, modrlc_web_1
```

Please note that unlike the command for launching a single worker, stopping the Alfalfa stack will automatically stop the Docker containers that have been started. By doing Ctrl+C, the following output confirms that all containers were stopped:

```
Gracefully stopping... (press Ctrl+C again to force)
Stopping modrlc_web_1    ... done
Stopping modrlc_worker_6 ... done
Stopping modrlc_worker_5 ... done
Stopping modrlc_worker_4 ... done
Stopping modrlc_worker_7 ... done
Stopping modrlc_worker_2 ... done
Stopping modrlc_worker_3 ... done
Stopping modrlc_worker_1 ... done
Stopping modrlc_redis_1  ... done
Stopping modrlc_mongo_1  ... done
Stopping modrlc_goaws_1  ... done
Stopping modrlc_minio_1  ... done
```

While the ACTB relies on its Alfalfa component to upload new models on demand, it is technically possible to build a dedicated BOPTEST Docker container that includes one or several building models. This usage is not the main purpose of the BOPTEST framework and will not be covered in this user guide, but some information is presented to the user who wishes to use the ACTB differently.

5.2 Client API

While the ACTB is running, the user can interact with the simulation worker by sending RESTful requests or by using the Alfalfa client API. The client API is a more user-friendly method for interacting with the simulation worker. The available RESTful requests are reported[8] in Table 1. For example, in Python, a user who needs to fetch the model name can use[9]:

```
import requests
url = 'http://127.0.0.1:5000'
name = requests.get('{0}/name'.format(url)).json()
```

The ACTBleverages Alfalfa's Python APISee the Alfalfa-Client's Pypi entry which can be installed either manually or with pip:

```
pip install alfalfa-client
```

After installing the client, the user can use it in their code by importing the client and historian if needed:

```
import alfalfa_client.alfalfa_client as ac
import alfalfa_client.historian as ah

client = ac.AlfalfaClient
historian = ah.Historian
```

The user can then use the methods reported in Table 2 to control the simulation, read sensor data and write commands to/from the simulated building systems. More methods are available, but are not needed by the user directly. Please refer to Alfalfa's GitHub webpage for more information on the Alfalfa client.

5.3 Interfacing an external clock and controller

The user can define both an "internal" controller, built into the Spawn model, and an external controller that will read sensor measurements and override one or several control variables. The control variables and sensors are defined in the Spawn model and can be retrieved using the Alfalfa client API described in Section 5.2.

The typical usage of the external controller is as follows:

1. **Initialize the simulation worker:** set up the communication parameters, and set simulation parameters such as the step in seconds, and the simulation start and end time in seconds. Initialize a historian if needed.

2. **Retrieve test case information:** retrieve useful information from the model, such as the available control points or sensor points.

3. **Initialize the control vector:** the $\mathbf{u}$ control vector can be any combination of available inputs, or can be left empty. Please refer to Section TK **??**.

4. **Step the simulation:** at each step the simulation returns the sensor readings, which can be used to update the control vector. It is possible to retrieve the last set of KPIs at any time in the process.

5. **Update the control vector:** this can be any function that yields a formatted $\mathbf{u}$ control vector.

6. **Retrieve results:** at the end of a simulation, the results are stored until the Docker container is reset, along with the KPIs calculated over the entire simulation run.

A simple controller that uses the Alfalfa client API will look as follows:

[8]reproduced from BOPTEST's user guide here

[9]assuming that the *requests* Python package has been installed. See the project's PyPi webpage

Request	Description
PUT	
PUT initialize with arguments start_time=, warmup_time=	Initialize simulation to a start time using a warmup period in seconds
PUT step with argument step=<value>	Set communication step in seconds.
PUT results with arguments point_name=, start_time=, final_time=	Receive test result data for the given point name between the start and final time in seconds.
PUT forecast_parameters with arguments horizon=, interval=	Set boundary condition forecast parameters in seconds.
PUT scenario with optional arguments electricity_price=, time_period=. See README in /testcases for options and test case documentation for details.	Set test scenario. Setting the argument time_period performs an initialization with predefined start time and warmup period and will only simulate for predefined duration.
GET	
GET step	Receive communication step in seconds.
GET measurements	Receive sensor signal point names (y) and metadata.
GET inputs	Receive control signal point names (u) and metadata.
GET kpi	Receive test KPIs.
GET name	Receive test case name.
GET forecast	Receive boundary condition forecast from current communication step.
GET forecast_parameters	Receive boundary condition forecast parameters in seconds.
GET scenario	Receive current test scenario.
POST	
POST advance with json data "{:}"	Advance simulation with control input and receive measurements.

Table 1: List of RESTful requests and their usage

Method	**Description**
submit(path)	Upload a model found in path. Returns a site object.
start(site, kwargs)	Start a site and initialize the simulation
setInputs(site, inputs)	Set the control vector of a site
advance([sites])	Step the simulation of a list of sites
outputs(site)	Retrieve results from the last step
stop(site)	Stop a site

Table 2: List of useful methods for interacting with the Alfalfa API

```python
import os
from alfalfa_client import AlfalfaClient, Historian

end_time = 365 * 24 * 3600
alfalfa = AlfalfaClient(url='http://localhost')
file = os.path.join(os.path.dirname(__file__), 'files', 'wrapped.fmu')
site = alfalfa.submit(file)
alfalfa.start(
    site,
    external_clock="true",
    end_datetime=end_time
)
for i in range(steps):
    y =
    u = process_controls(y)
    alfalfa.setInputs(site, u)
```

Below, an example using HTTP requests as shown in Table 1:

```python
import requests, json, collections, os
url = 'http://0.0.0.0:5000'
length = 1 * 24 * 3600
step = 300
simStartTime = 186*24*3600
warmupPeriod = 30 * 24 * 3600

# GET TEST INFORMATION
# --------------------

name = requests.get('{0}/name'.format(url)).json()
inputs = requests.get('{0}/inputs'.format(url)).json()
measurements = requests.get('{0}/measurements'.format(url)).json()
step_def = requests.get('{0}/step'.format(url)).json()

# RUN TEST CASE
# -------------

res = requests.put('{0}/initialize'.format(url),
                    data={
                        'start_time': simStartTime,
                        'warmup_period': warmupPeriod})
res = requests.put('{0}/step'.format(url), data={'step': step})
u = initializeControls()
for i in range(int(length / step)):
    y = requests.post('{0}/advance'.format(url), data=u).json()

    u = rulebased.compute(y)

# POST PROCESS RESULTS
# --------------------

res = requests.get('{0}/results'.format(url)).json()
kpi = requests.get('{0}/kpi'.format(url)).json()
```

For details on interfacing an OpenAI Gym agent or DO-MPC, please refer to Section 6.

5.4 Additional simulation considerations

Users should be aware of the following limitations, and should regularly check if a new ACTB release is available:

-

6 Advanced controller interfaces

The ACTB includes interfaces to two popular machine learning and MPC toolboxes: the OpenAI Gym framework and DO-MPC. Both interfaces are based in Python, and this manual assumes that the user has already installed and tested the packages.

6.1 OpenAI Gym

A custom Gym environment wrapper is included with the ACTB, which acts as an interface between Gym and the RESTful API. First, the user should instantiate the custom Gym environment and provide the following information:

- **Time variables**: length, start time and time step of the simulation, in seconds.
- **Actions**: the control points, as displayed when requesting the available model inputs.
- **Reward function**: The custom Gym environment uses BOPTEST built-in KPIs to calculate the reward function. The user can specify the KPIs to return for computing the reward at each control step, as well as the weighing (linear and exponential) of the objective. The weighing consists of one linear and exponential hyper-parameter. The type of KPI objectives could be any subset of the KPIs provided by BOPTEST. These could be energy, thermal discomfort, indoor air quality discomfort, CO_2 emissions and operational costs. The user can specify to return the KPIs from all zones or only a subset with the `kpi_zones` argument, and the rewards are summed over all the zones.
- **Observation states**: There are two types of observation states that needs to be specified - one is the set of observation states from the building sensors while the other one is external weather variables and the associated forecastcurrently, the weather forecast is assumed perfect.. These two type of variables are concatenated and are returned as full observation states to the RL controller agent.

Below, an example of building environment that interfaces a RL agent to the included Reference Small Office Building test case:

```
episode_length = 24 * 3600     # Simulate for a day
step = 300                     # Setting Step size in seconds
start_time = 3*24*3600         # Start from the third day of the year
kpi_zones = ["1"]              # Select the zones for the reward function

actions = ['oveHeaSet1_u']

building_obs = ['senTRoom1_y','senHou_y']
forecast_obs = {'TDryBul': [4], 'TWetBul': [0], 'HGloHor': [0]}

''' Final observations are the combined 'building_obs' and 'forecast_obs' '''

# Customize the reward objective -- uses BOPTEST KPIs
KPI_rewards  = { "ener_tot": {"hyper": -20, "power": 2},
                 "tdis_tot": {"hyper": -50, "power": 1},
                 "idis_tot": {"hyper": 0,   "power": 1},
                 "cost_tot": {"hyper": 0,   "power": 1},
                 "emis_tot": {"hyper": 0,   "power": 1} }
# --------------------

env = BoptestGymEnv( max_episode_length=episode_length,
                     Ts                = step,
                     actions           = actions,
                     building_obs      = building_obs,
                     forecast_obs      = forecast_obs,
                     lower_obs_bounds  = [273,  0, 273, 273, 0,   0],
                     upper_obs_bounds  = [330, 24, 330, 330, 4, 700],
                     kpi_zones         = kpi_zones,
                     KPI_rewards       = KPI_rewards,
                     start_time        = start_time)
```

Figure 12: Setting up a RL problem

The user can control the simulation as described in Section 5.2, with the addition of the following methods for the custom environment:

- *step([actions])*: returns a tuple composed of the current state, rewards, and additional custom information that can be defined in TK ADD SCRIPT NAME. The *step()* function takes a list of actions as an input.
- *reset()*: resets all the building states to the initial start time.
- *get_building_states()*: returns the current building states at as a dictionary.
- *get_weather_forecast()*: returns the current weather and its forecast.
- *get_KPIs()*: returns the sum of all KPIs

6.2 DO-MPC

7 Additional features

7.1 N4SID model reduction

Appendices

A DOE Reference Small Office Building

A.1 Design of a Spawn model for the ACTB

A.2 Design of a Python controller interface

A.3 Simulation in the ACTB and results

B DOE Reference Medium Office Building

B.1 Design of a Spawn model for the ACTB

B.2 Design of a Python controller interface

B.3 Simulation in the ACTB and results

References