mosaik Documentation

Release 3.3.0b1

Mosaik Development Team

Mar 13, 2024

CONTENTS

1 Quickstart **3**

2 Installation **5**
- 2.1 Linux . . . 5
- 2.2 OS X . . . 6
- 2.3 Windows . . . 8

3 Overview **11**
- 3.1 What's mosaik supposed to do? . . . 11
- 3.2 Mosaik's main components . . . 12

4 mosaik ecosystem **15**
- 4.1 mosaik-heatpump . . . 17

5 Tutorials **51**
- 5.1 Integrating a simulation model into the mosaik ecosystem . . . 51
- 5.2 Creating and running simple simulation scenarios . . . 59
- 5.3 Adding a control mechanism to a scenario . . . 64
- 5.4 Integrating a control mechanism . . . 68
- 5.5 Same-time loops . . . 71
- 5.6 Set external events . . . 74
- 5.7 Connecting mosaik and Odysseus . . . 81
- 5.8 Using Odysseus to process, visualize and store simulation data . . . 84
- 5.9 Integrating a Model in Java . . . 86
- 5.10 Integrating a Model in Java with Generics / Annotations API . . . 91
- 5.11 Use mosaik with Jupyter Notebooks . . . 99
- 5.12 Plotting graphs . . . 100
- 5.13 Using Apache Superset to visualize simulation data . . . 108

6 The mosaik API **121**
- 6.1 How mosaik communicates with a simulator . . . 122
- 6.2 The low-level API . . . 125
- 6.3 The high-level API . . . 136
- 6.4 High-level APIs in other languages . . . 143

7 Scenario definition **145**
- 7.1 The setup . . . 145
- 7.2 Starting simulators . . . 146
- 7.3 Instantiating simulation models . . . 146
- 7.4 Setting initial events . . . 147
- 7.5 Connecting entities . . . 147

7.6 Running the simulation . . . 149
7.7 How to achieve cyclic data-flows . . . 150
7.8 How to filter entity sets . . . 152
7.9 How to create user-defined connection rules . . . 153
7.10 How to retrieve static data from entities . . . 154
7.11 How to access topology and data-flow information . . . 154
7.12 How to destroy a world . . . 155
7.13 How to do real-time simulations . . . 155
7.14 How to call extra methods of a simulator . . . 156

8 The simulator manager **157**

9 Scheduling and simulation execution **161**
9.1 Time paradigms . . . 161
9.2 Advancing through time . . . 161
9.3 Synchronization and data-flows . . . 163
9.4 Cyclic data-flows . . . 165
9.5 Stepping and simulation duration . . . 167

10 Upgrading from mosaik 2 to 3 **171**
10.1 Component's type . . . 171
10.2 Time resolution . . . 171
10.3 Max_advance time . . . 171
10.4 Sticking with mosaik 2 . . . 171

11 FAQ **173**
11.1 General questions . . . 173
11.2 Coupling models with mosaik . . . 173
11.3 The mosaik demo . . . 174

12 Developer's Documentation **175**
12.1 Discussion of design decisions . . . 175
12.2 Logo and corporate identity . . . 178

13 API Reference **183**
13.1 `mosaik` — The end-user API . . . 183
13.2 `mosaik.exceptions` — mosaik specific error types . . . 183
13.3 `mosaik.scenario` — Classes related to the scenario creation . . . 183
13.4 `mosaik.scheduler` — Coordinate and execute simulators . . . 189
13.5 `mosaik.simmanager` — Management of external processes . . . 192
13.6 `mosaik.util` — Utility classes and functions . . . 197
13.7 `mosaik.basic_simulators` — Basic simulators . . . 199

14 About mosaik **201**
14.1 Acknowledgments . . . 201
14.2 The history of mosaik . . . 201

15 Privacy Policies **209**

16 Legals **211**

17 Datenschutz **213**

18 Impressum **217**

19 Glossary **219**

20 Indices and tables 221

Python Module Index 223

Index 225

Mosaik is a flexible Smart Grid co-simulation framework.

Mosaik allows you to reuse and combine existing simulation models and simulators to create large-scale Smart Grid scenarios – and by large-scale we mean thousands of simulated entities distributed over multiple simulator processes. These scenarios can then serve as a test bed for various types of control strategies (e.g., multi-agent systems (MAS) or centralized control).

Here, we provide the documentation about mosaik.

Contents:

CHAPTER

ONE

QUICKSTART

This guide assumes that you are somewhat proficient with Python and know what *pip* and *virtualenv* is. Else, you should follow the *detailed instructions*.

Mosaik runs on Linux, OS X and Windows. It requires Python 3.8 or higher. To install everything, you need the package manager Pip which is bundled with Python 3.8 and above.

We also strongly recommend you to install everything into a virtualenv.

You can then install mosaik with pip:

```
$ pip install mosaik
```

This provides you with the mosaik framework. There is also a simple demo scenario which may help you to get started. Please refer to our *detailed instructions* for installation.

For more information about avaiable components and example scenarios visit the *mosaik ecosystem* page.

CHAPTER

TWO

INSTALLATION

This guide contains detailed installation instructions for *Linux*, *OS X* and *Windows*.

It covers the installation of the mosaik framework followed by the instructions to install the demo.

2.1 Linux

This guide is based on *(K)ubuntu 18.04 Bionic Beaver, 64bit*.

Mosaik and the demo scenario require Python >= 3.8, which should be fine for any recent linux distribution. Note that we test mosaik only for the most (typically three) recent python versions though.

1. We also need pip, a package manager for Python packages, and virtualenv, which can create isolated Python environments for different projects:

   ```
   $ wget https://bootstrap.pypa.io/get-pip.py
   $ sudo python get-pip.py
   $ sudo pip install -U virtualenv
   ```

2. Now we need to create a virtual environment for mosaik and its dependencies. The common location for venvs is under `~/.virtualenvs/`:

   ```
   $ virtualenv -p /usr/bin/python3 ~/.virtualenvs/mosaik
   $ source ~/.virtualenvs/mosaik/bin/activate
   ```

   Your command line prompt should now start with "(mosaik)" and roughly look like this: `(mosaik)user@kubuntu:~$`.

3. The final step is to install mosaik:

   ```
   (mosaik)$ pip install mosaik
   ```

   Mosaik should now be installed successfully.

2.1.1 Running the demo

Mosaik alone is not very useful (because it needs other simulators to perform a simulation), so we also provide a small demo scenario and some simple simulators as well as a mosaik binding for PYPOWER.

1. PYPOWER requires *NumPy* and *SciPy*. We also need to install the revision control tool *git*. You can use the packages shipped with Ubuntu. We use **`apt-get`** to install NumPy, SciPy, and h5py as well as git. By default, venvs are isolated from globally installed packages. To make them visible, we also have to recreate the venv and set the `--system-site-packages` flag:

```
$ sudo apt-get install git python3-numpy python3-scipy python3-h5py
$ rm -rf ~/.virtualenvs/mosaik
$ virtualenv -p /usr/bin/python3 --system-site-packages ~/.virtualenvs/mosaik
$ source ~/.virtualenvs/mosaik/bin/activate
```

2. You can now clone the mosaik-demo repository into a folder where you store all your code and repositories (we'll use `~/Code/`):

```
(mosaik)$ mkdir ~/Code
(mosaik)$ git clone https://gitlab.com/mosaik/mosaik-demo.git ~/Code/mosaik-demo
```

3. Now we only need to install all requirements (mosaik and the simulators) and can finally run the demo:

```
(mosaik)$ cd ~/Code/mosaik-demo/
(mosaik)$ pip install -r requirements.txt
(mosaik)$ python demo.py
```

   If no errors occur, the last command will start the demo. The web visualisation shows the demo in your browser: http://localhost:8000. You can click the nodes of the topology graph to show a time series of their values. You can also drag them around to rearrange them.

   You can cancel the simulation by pressing `Ctrl-C`.

2.2 OS X

This guide is based on *OS X 10.11 El Capitan*.

1. Mosaik and the demo scenario require Python >= 3.8. OS X only ships with some outdated versions of Python, so we need to install a recent Python 2 and 3 first. The recommended way of doing this is with the packet manager homebrew. To install homebrew, we need to open a *Terminal* and execute the following command:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install)"
```

   The homebrew installer asks you to install the *command line developer tools* for "xcode-select". Install them. When you are done, go back to the terminal and press `Enter` so that the installer continues.

   If this doesn't work for you, you'll find more detailed instructions in the homebrew wiki.

   Once the installation is successful, we can install `python` and `python3`:

```
$ brew install python python3
```

   This will also install the Python package manager pip.

2. Next, we need virtualenv which can create isolated Python environments for different projects:

```
$ pip install -U virtualenv
```

3. Now we need to create a virtual environment for mosaik and its dependencies. The common location for venvs is under `~/.virtualenvs/`:

```
$ virtualenv -p /usr/local/bin/python3 ~/.virtualenvs/mosaik
$ source ~/.virtualenvs/mosaik/bin/activate
```

   Your command line prompt should now start with "(mosaik)" and roughly look like this: `(mosaik)user@mac-book:~$`.

4. The final step is to install mosaik:

```
(mosaik)$ pip install mosaik
```

   Mosaik should now be installed successfully.

2.2.1 Running the demo

Mosaik alone is not very useful (because it needs other simulators to perform a simulation), so we also provide a small demo scenario and some simple simulators as well as a mosaik binding for PYPOWER.

1. To clone the demo repository, we need to install *git*. In order to compile *NumPy*, *SciPy* and *h5py* (which are required by PYPOWER and the database adapter) we also need to install *gfortran* which is included in *gcc*. You should deactivate the venv for this:

```
(mosaik)$ deactivate
$ brew install git gcc hdf5
$ source ~/.virtualenvs/mosaik/bin/activate
```

2. For NumPy and SciPy we build binary wheel packages that we can later reuse without re-compiling everything. We'll store these *wheels* in `~/wheelhouse/`:

```
(mosaik)$ pip install wheel
(mosaik)$ pip wheel numpy
(mosaik)$ pip install wheelhouse/numpy-1.10.1-cp35-cp35m-macosx_10_6_intel.macosx_
↪10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
(mosaik)$ pip wheel scipy
(mosaik)$ pip install wheelhouse/scipy-0.16.0-cp35-cp35m-macosx_10_6_intel.macosx_
↪10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
(mosaik)$ pip wheel h5py
(mosaik)$ pip install wheelhouse/h5py-2.5.0-cp35-cp35m-macosx_10_6_intel.macosx_
↪10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
```

Note: The file names of the *wheels* (*.whl-files) may change when version-numbers change. Please check the output of *pip install* or the directory `~/wheelhouse/` for the exact file names.

---

2. You can now clone the mosaik-demo repository into a folder where you store all your code and repositories (we'll use `~/Code/`):

```
(mosaik)$ mkdir ~/Code
(mosaik)$ git clone https://gitlab.com/mosaik/mosaik-demo.git ~/Code/mosaik-demo
```

3. Now we only need to install all requirements (mosaik and the simulators) and can finally run the demo:

```
(mosaik)$ cd ~/Code/mosaik-demo/
(mosaik)$ pip install -r requirements.txt
(mosaik)$ python demo.py
```

If no errors occur, the last command will start the demo. The web visualisation shows the demo in your browser: http://localhost:8000. You can click the nodes of the topology graph to show a time series of their values. You can also drag them around to rearrange them.

You can cancel the simulation by pressing `Ctrl-C`.

2.3 Windows

This guide is based on *Windows 10, 64bit*.

1. Mosaik and the demo scenario require Python >= 3.8. By default, it will offer you a 32bit installer. You can find the *Windows x86-64 MSI installer* here.

   1. When the download finished, double-click the installer.
   2. Select *Install for all users* and click *Next* >.
   3. The default installation path is okay. Click *Next* >.
   4. In the *Customize Python* page, click on the *Python* node and select *Entire feature will be installed on local hard drive*. Make sure that *Add python.exe to Path* is enabled. Click *Next* >.
   5. When Windows asks you to allow the installation, allow the installation. Wait. Click *Finish*.

   This also install the Python package manager pip.

2. We also need virtualenv which can create isolated Python environments for different projects.

   Open a terminal window: Press the `Windows` key (or click on the start menu) and enter `cmd`. Press `Enter`. Your terminal prompt should look like `C:\Users\yourname>`. Execute the following command to install virtualenv:

   ```
   C:\Users\yourname> pip install -U virtualenv
   ```

   Note: If your Windows account type is *Standard User*, you need to open the terminal with administarator privileges (right-click the Terminal icon, then *open as Administrator*). Make then sure that you are in your user directory:

   ```
   C:\Windows\system32> cd C:\Users\yourname
   C:\Users\yourname>
   ```

3. Now we need to create a virtual environment for mosaik and its dependencies. The common location for venvs is under `Envs/` in your users directory:

   ```
   C:\Users\yourname> virtualenv -p path\to\python.exe Envs\mosaik
   ```

   To activate the virtual environment use the following command:

   ```
   C:\Users\yourname> Envs\mosaik\Scripts\activate
   ```

   This command should also function when using the powershell, however the execution policy might need to be changed.

   Your command line prompt should now start with "(mosaik)" and roughly look like this: `(mosaik) C:\Users\yourname>`.

4. The final step is to install mosaik:

```
(mosaik) C:\Users\yourname> pip install mosaik
```

   Mosaik should now be installed successfully.

2.3.1 Running the demo

Mosaik alone is not very useful (because it needs other simulators to perform a simulation), so we also provide a small demo scenario and some simple simulators as well as a mosaik binding for PYPOWER.

1. Download and install git.

   Restart the command prompt (as Admin if necessary and make sure you are in the right directory again) and activate the virtualenv again:

```
C:\Users\yourname> Envs\mosaik\Scripts\activate
```

2. Clone the demo repository:

```
(mosaik)C:\Users\yourname> git clone https://gitlab.com/mosaik/mosaik-demo.git
```

3. Now we only need to install all requirements (mosaik and the simulators) and can finally run the demo:

```
(mosaik)C:\Users\yourname> cd mosaik-demo
(mosaik)C:\Users\yourname\mosaik-demo> pip install -r requirements.txt
(mosaik)C:\Users\yourname\mosaik-demo> python demo.py
```

   The web visualisation shows the demo in your browser: http://localhost:8000. You can click the nodes of the topology graph to show a timeline of their values. You can also drag them around to rearrange them.

   You can cancel the simulation by pressing `Ctrl-C`. More exceptions may be raised. No problem. :-)

CHAPTER

THREE

OVERVIEW

This section describes how mosaik works without going into too much detail. After reading this, you should have a general understanding of what mosaik does and how to proceed in order to implement the mosaik API or to create a *simulation scenario*.

3.1 What's mosaik supposed to do?

Mosaik's main goal is to use existing *simulators* in a common context in order to perform a coordinated simulation of a given (Smart Grid) scenario.

That means that all simulators (or other tools and hardware-in-the-loop) involved in a simulation usually run in their own process. Mosaik just tries to synchronize these processes and manages the exchange of data between them.

To allow this, mosaik

1. provides an API for simulators to communicate with mosaik,
2. implements handlers for different kinds of simulator processes,
3. allows the modelling of simulation scenarios involving the different simulators, and
4. schedules the step-wise execution of the different simulators and manages the exchange of data (*data-flows*) between them.

Although mosaik is written in Python 3, its simulator API completely language agnostic. It doesn't matter if your simulator is written in Python 2, Java, C, matlab or anything else.

3.1.1 A simple example

We have simulators for households (blue icon) and for photovoltaics (green). We're also gonna use a load flow analysis tool (grey), and a monitoring and analysis tool (yellow).

First, we have to implement the mosaik API for each of these "simulators". When we are done with this, we can create a scenario where we connect the households to nodes in the power grid. Some of the households will also get a PV module. The monitoring / analysis tool will be connected to the power grid's transformer node. When we connect all these *entities*, we also tell mosaik about the data-flows between them (e.g., active power feed-in from the PV modules to a grid node).

When we finally start the simulation, mosaik requests the simulators to perform simulation steps and exchanges data between them according to the data-flows described in the scenario. For our simple example, that would roughly look like this:

1. The household and PV simulator perform a simulation step for an interval *[0, t[*.
2. Mosaik gets the values for, e.g., *P* and *Q* (active and reactive power) for every household and every PV module.
3. Mosaik sets the values *P* and *Q* for every node of the power grid based on the data it collected in step 2. The load flow simulator performs a simulation step for *[0, t[* based on these inputs.
4. Mosaik collects data from the load flow simulator, sends it to the monitoring tool and lets it also perform a simulation step for *[0, t[*.
5. Now the whole process is repeated for *[t, t+i[* and so forth until the simulation ends.

In this example, all simulators had the same step size *t*, but this is not necessary. Every simulator can have its one step size (which may even vary during the simulation). It is also possible that a simulator (e.g., a control strategy) can set input values (e.g., a schedule) to another simulator (e.g., for "intelligent" consumers).

3.2 Mosaik's main components

Mosaik consists of four main components that implement the different aspects of a co-simulation framework:

1. The **mosaik Sim API** defines the communication protocol between *simulators* and mosaik.

   Mosaik uses plain network sockets and JSON encoded messages to communicate with the simulators. We call this the *low-level API*. For some programming languages there also exists a *high-level API* that implements everything networking related and offers an abstract base class. You then only have to write a subclass and implement a few methods.

   Read more …

2. The **Scenario API** provides a simple API that allows you to create your simulation scenarios in pure Python (yes, no graphical modelling!).

   The scenario API allows you to start simulators and instantiate models from them. This will give you *entity sets* (sets of *entities*). You can then connect the entities with each other in order to establish *data-flows* between the simulators.

   Mosaik allows you both, connecting one entity at a time as well as connecting whole entity sets with each other.

   Read more …

3. The **Simulator Manager** (or shorter, **SimManager**) is responsible for handling the simulator processes and communicating with them.

   It is able to *a)* start new simulator processes, *b)* connect to already running process instances, and *c)* import a simulator module and execute it *in-process* if it's written in Python 3.

   The in-process execution has some benefits: it reduces the amount of memory required (because less processes need to be started) and it avoids the overhead of (de)serializing and sending messages over the network.

   External processes, however, can be executed in parallel which is not possible with in-process simulators.

   Read more …

4. Mosaik's **Scheduler** uses the event-discrete simulation approach for the coordinated simulation of a scenario.

   Mosaik supports both time-discrete and event-discrete simulations as well as a combination of both paradigms.

   Mosaik is able to handle simulators with different step sizes. A simulator may even vary its step size during the simulation.

   Mosaik tracks the dependencies between the simulators and only lets them perform a simulation step if necessary (e.g., because its data is needed by another simulator). It is also able to let multiple simulators perform their simulation step in parallel if they don't depend on each other's data.

   Read more …

CHAPTER

FOUR

MOSAIK ECOSYSTEM

Mosaik as a co-simulation tool organizes the data exchange between simulators and coordinates the execution of the connected simulaters. This part is called *mosaik-core* and contains mosaik itself and APIs for multiple programming languages.

Fig. 1: Mosaik is a co-simulation library. The components and tools form the mosaik ecosystem.

Mosaik-core without any connected simulators doesn't do much. This is why we provide some simple and free simulators so that it is possible to start with a working Smart-Grid simulation. These simulators belong to a part of mosaik's ecosystem called *mosaik-components*. More detailed documentation for some components can be found in the *component documentation*.

To see how these components can be coupled to simulations, also some example scenarios are provided in *mosaik-examples*.

Mosaik is developed following the "lean and mean" principle. That means that we try to keep the software as simple as possible in order to keep it efficient and easy to maintain. In order to make it easier to set up and run experiments with mosaik we provide some tools that help building scenarios, connecting simulators or to visualize and analyze the simulation results. These tools are located in the *mosaik-tools*.

For testing simulators or scenarios, mosaik provides some *basic simulators*, which allow to specify specific data to be sent.

There are also some implementations done by external users of mosaik. We give an overview of the *external components* and *external scenarios* we know.

mosaik-core

The root folder contains mosaik itself and the high-level API implementations are provided in the API folder.

- mosaik
- API for Python
- API for Java
- generics for Java API
- *Java tutorial*
- API for Matlab
- API for Matlab via Java
- API for C#

mosaik-components

This lists the mosaik components that are available on pypi. There are always component in work that are not released yet but are in working condition so if you don't find what you are searching for here take a look in the repository.

- energy related components:
  - mosaik-pandapower is an adapter for the pandapower power system modeling, analysis and optimization tool. Not maintained anymore, please install the mosaik-pandapower2 adapter.
  - mosaik-pandapipes is an adapter for the pandapipes fluid system modeling, analysis and optimization tool.
  - mosaik-pypower is an adapter for the PYPOWER load flow analysis library. Not maintained anymore, please install the mosaik-pandapower2 adapter.
  - mosaik-heatpump contains different models for simulation of heatpump systems (detailled documentation can be found *here*).
  - mosaik-pv is a simple PV Simulator based on PyPVSim.
  - mosaik-pvlib is a simple PV Simulator based on pvlib.
  - mosaik-pvgis is a simple PV Simulator based on PVGIS.
  - mosaik-householdsim is a househol simulator that simulate households by serving residual load profiles.
- data related components:
  - mosaik-web is a web visualization for mosaik simulations.
  - mosaik-csv is a simple demo simulators that you can use to integrate CSV data sets into simulation. It can also write data into CSV data sets.
  - mosaik-hdf5 allows to write simulation results to a HDF5 file for further analysis.
  - InfluxDB adapter to store simulation results into InfluxDB 1 time series database.
  - InfluxDB 2 adapter to store simulation results into InfluxDB 2 time series database.
  - mosaik-sql adapter to store simulation results into SQL database.
  - mosaik-timescaledb adapter to store simulation results into a postgres or timescale database.
  - ZeroMQ adapter to connect components with the messaging library ZeroMQ.
  - *Odysseus-adapter* to write results to the data stream management system Odysseus to mosaik.
- communication related components:
  - communication simulator is a basic communication suite using delays.

  - mosaik-104 contains an adapter to receive IEC 60870-5-104 protocol messages and hands it over to mosaik.
- FMI adapter allows to couple Functional Mockup Units (FMU), which are based on the FMI standard.

component documentation

The components listed above and provided by the mosaik team, have usually a documentation directly in their repository. For components which need a more detailled documentation to describe how they work, the documentation is integrated here:

- *mosaik-heatpump*

4.1 mosaik-heatpump

4.1.1 Models

Heat pump

The heat pump model comprises four different calculation modes for simulating the performance of a heat pump - '*detailed*', '*fast*', '*hplib*', and '*fixed*' modes. The first two are based on the TESPy library, the third is based on the hplib library. In these three modes, a quasi-steady state modelling approach has been adopted, i.e., the conditions of the operation of the heat pump vary with each time-step and the steady state performance of the heat pump is calculated at these different conditions for each time-step. The '*fixed*' mode is the most simplified, operating the heat pump with a fixed performance irrespective of the different operating conditions.

The model based on hplib, hereafter referred as "*hplib*" model, is a parametric fit equation-based model, and thus takes a statistical approach to predict the performance of the heat pump at different operating conditions. The model based on TESPy, hereafter referred as "*tespy*" model, is more complex and considers the physical states of the fluids in the different components of the heat pump. Therefore, it offers greater flexibility than the *hplib* model in estimating the performance of the heat pump at different operating conditions. However, as a result of this increased complexity, the simulation time for the detailed calculations in *tespy* model is higher as compared to that of the simpler calculations in *hplib* model.

hplib model

The *hplib* model is based on hplib ("Heat Pump LIBrary"), an open-source Python library that simulates the performance heat pumps using parametric fit equations for the electric power and COP. The fit parameters are identified by applying a least square regression model on the publicly available heat pump keymark data of the European Heat Pump Association (EHPA). It is possible to simulate the performance of both air and water source heat pumps. The parameters are available for a generic heat pump of both the types, as well as specific models available in the market.

The limits on the operation of the heat pump, the supply water and source air temperature ranges available from the technical datasheets of the chosen heat pump model, have been added to the model directly available in the hplib library.

The equations 1 and 2 are the fit equations for the electric power and COP respectively. The reference values, $P_{el,ref}$ is the electrical power consumption at -7°C source temperature and 52°C supply water temperature. In both the equations, $p_{1\text{-}4}$ are the fit parameters, T_{in} is the source inlet temperature, T_{out} is the supply water temperature, and T_{amb} is the ambient temperature.

The evaporator and condenser inlet temperatures are the inputs to the model. The model checks if they are within the operating range and ensures that the source air temperature is lower than the incoming water temperature. The model then calculates the electric power, COP, the heating capacity, and the condenser mass flow as outputs. The electric power and COP are estimated as shown in equations 1 and 2 respectively. The heating capacity is calculated from the electric power and COP. The mass flow in the condenser is calculated assuming a temperature difference of 5°C.

$$P_{el} = P_{el,ref} * (p_1 * T_{in} + p_2 * T_{out} + p_3 + p_4 * T_{amb})$$

Equation 1: Fit equation for electric power

$$COP = p_1 * T_{in} + p_2 * T_{out} + p_3 + p_4 * T_{amb}$$

Equation 2: Fit equation for COP

How to use the *hplib* calculation mode

The user must specify the '*calc_mode*' parameter as '*hplib*', and the '*heat_source*', either '*air*' or '*water*', must be specified. For the '*hp_model*' parameter, the user can choose from the different heat pump models available in the public heatpump keymark database (the keywords can be obtained from the 'hplib_database.csv' file). If the '*hp_model*' is set to '*Generic*', the user must additionally specify '*cons_T*', '*heat_source_T*', and '*P_th*'.

The limits of operation for the heat pump are not available directly within the model in the *hplib* library. If a corresponding equivalent heat pump model based on TESPy is available, the keyword for that model can be specified in the '*equivalent_hp_model*'. If not, the operation limits can be specified via '*hp_limits*' parameter.

An example of the dictionary with the required parameters can be seen in the *module documentation*.

tespy model

The *tespy* model is based on TESPy ("Thermal Engineering Systems in Python"), an open-source Python library that provides a powerful simulation package for thermal processes like power plants, district heating systems, heat pumps etc. An initial version of this model has been used in a previous work, and significant changes have been made later for a master's thesis and for different research projects. The performance of the heat pump is simulated by considering the energy and mass balances in the individual "*components*" of the heat pump – condenser, evaporator, compressor, expansion valve, heat exchangers and pumps – and the state of fluids in the "*connections*" between these "*components*." The *connections* and *components* together form a topological *network* that is represented and solved as a system of equations. The schematic of the heat pump system used in this work is shown in the figure below.

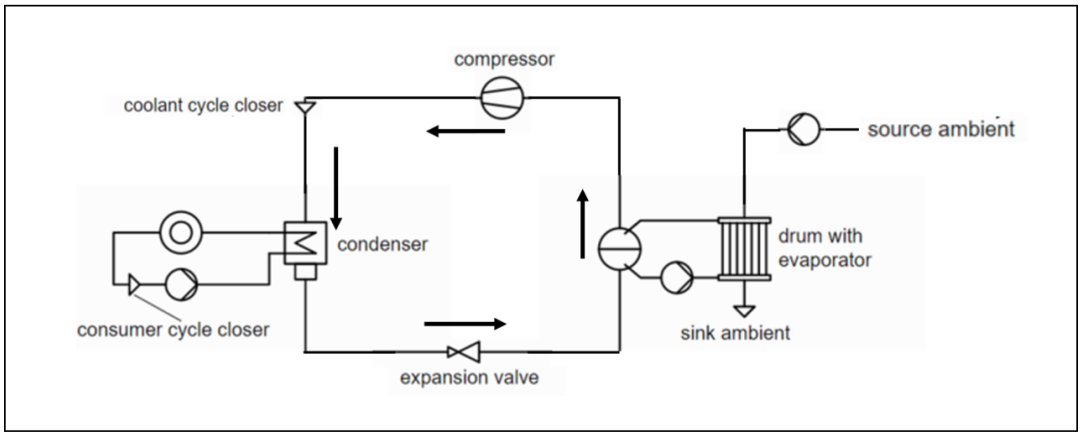


Fig. 2: Schematic of the heat pump system network

The flexibility offered by the TESPy library in choosing the components of the network has been implemented through the following features in the model:

Stages of compression

- The heat pump model is available in two system configurations, either with a one-stage compressor or a two-stage compressor.

Additional components

- Intercooler between the two stages of compression
- Superheater between the evaporator and the compressor

TESPy has two modes of calculation, *design* and *offdesign*, to solve the network. The *design* mode is used to design the system and forms the first calculation of the network. While designing the plant, TESPy offers much greater detail as compared to hplib, in terms of the parametrization of the individual components, for example, the isentropic efficiency of the compressor. The *offdesign* mode is used to calculate the performance of the system if parameters deviate from the design point, for example, operation at partial loads or operation at different temperature/pressure levels. The system calculations from the *design* mode form the basis for the *offdesign* mode. Both of these calculation modes have been implemented in this model.

How to use the *tespy* based calculation modes

The user must specify the '*calc_mode*' parameter as '*detailed*' or '*fast*'. A detailed description of these two modes of calculation can be found *here*.

The '*heat_source*', either '*air*' or '*water*', must be specified.

For the '*hp_model*' parameter, the user can choose from the different heat pump models available, shown in the table below.

Heat pump model from market	Keyword for 'calc_mode'	Configuration
Daikin Altherma ERLQ006CV3	Air_6kW	*Stages of compression* - 1 *Intercooler* - No *Superheater* - No *Heating capacity range* - 1.8 kW - 12.07 kW *Operating temperatures* - Source air: -20°C to 25°C; Water supply: 15°C to 55°C
Daikin Altherma ERLQ008CV3	Air_8kW	*Stages of compression* - 1 *Intercooler* - No *Superheater* - No *Heating capacity range* - 1.8 kW - 14.49 kW *Operating temperatures* - Source air: -20°C to 25°C; Water supply: 15°C to 55°C
Daikin Altherma ERLQ016CV3	Air_16kW	*Stages of compression* - 1 *Intercooler* - No *Superheater* - No *Heating capacity range* - 6.46 kW - 22.9 kW *Operating temperatures* - Source air: -20°C to 35°C; Water supply: 15°C to 55°C
Viessmann Vitocal 300-A	Air_25kW	*Stages of compression* - 2 *Intercooler* - No *Superheater* - No *Heating capacity range* - 10.76 kW - 39.4 kW *Operating temperatures* - Source air: -20°C to 35°C; Water supply: 15°C to 55°C
	Air_25kW_1s	*Stages of compression* - 1 *Intercooler* - No *Superheater* - No *Heating capacity range* - 5.21 kW - 22.45 kW *Operating temperatures* - Source air: -20°C to 35°C; Water supply: 15°C to 55°C
ait-deutschland LW-300(L)	Air_30kW	*Stages of compression* - 2 *Intercooler* - No *Superheater* - No *Heating capacity range* - 15.8 kW - 54 kW *Operating temperatures* - Source air: -20°C to 35°C; Water supply: 15°C to 60°C
	Air_30kW_1s	*Stages of compression* - 1 *Intercooler* - No *Superheater* - No *Heating capacity range* - 7.3 kW - 30.95 kW *Operating temperatures* - Source air: -20°C to 35°C; Water supply: 15°C to 60°C
Viessmann Vitocal 300-A	Air_40kW	*Stages of compression* - 2 *Intercooler* - No *Superheater* - No *Heating capacity range* - 16.73 kW - 53.41 kW *Operating temperatures* - Source air: -20°C to 35°C; Water supply: 15°C to 55°C
	Air_40kW_1s	*Stages of compression* - 1 *Intercooler* - No *Superheater* - No *Heating capacity range* - 9.75 kW - 30.35 kW *Operating temperatures* - Source air: -20°C to 35°C; Water supply: 15°C to 55°C
Viessmann Vitocal 300-A	Air_60kW	*Stages of compression* - 2 *Intercooler* - No *Superheater* - No *Heating capacity range* - 21.5 kW - 98.17 kW *Operating temperatures* - Source air: -20°C to 35°C; Water supply: 15°C to 65°C
	Air_60kW_1s	*Stages of compression* - 1 *Intercooler* - No *Superheater* - No *Heating capacity range* - 11.98 kW - 50.58 kW *Operating temperatures* - Source air: -20°C to 35°C; Water supply: 15°C to 65°C

Any other heat pump available in the market, with a different heating capacity and configuration, can be added to the model, following the procedure shown in the *example* of the "Air_30kW" heat pump.

Note: With TESPy, it is possible to simulate the performance of water-water heat pumps as well. However, this has not yet been integrated into this model and will be a part of a later release.

Example

An example scenario using the heat pump simulator in the mosaik environment is available in the 'run_heatpump.py' file.

The simulation is configured as shown below. The inputs to the heat pump model and the outputs from it are handled by *'mosaik-csv'* .

```
SIM_CONFIG = {
    'HeatPumpSim': {
        'python': 'mosaik_components.heatpump.Heat_Pump_mosaik:HeatPumpSimulator',
    },
    'CSV': {
        'python': 'mosaik_csv:CSV',
    },
    'CSV_writer': {
        'python': 'mosaik_csv_writer:CSVWriter'
    },
}

# Create World
world = mosaik.World(SIM_CONFIG)

START = '01.01.2016 00:00'
END = 10 * 15 * 60  # 2.5 Hours or 150 mins
```

The **tespy** model is used in the *'fast'* calculation mode. The *'Air_8kW'* heat pump is chosen. The required parameters are set as shown below.

```
# Heat pump
params = {'calc_mode': 'fast',
          'hp_model': 'Air_8kW',
          'heat_source': 'air',
          }
# configure the simulator
heatpumpsim = world.start('HeatPumpSim', step_size=15*60)
# Instantiate model
heatpump = heatpumpsim.HeatPump(params=params)
```

The timeseries of heat demand, heat source temperature, and the condenser water inlet temperature, that are needed as inputs for the model, are available in the 'heatpump_ data.csv' file.

```
# Input data csv
HEAT_LOAD_DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'data',
↪'heatpump_data.csv')
# configure the simulator
csv = world.start('CSV', sim_start=START, datafile=HEAT_LOAD_DATA)
# Instantiate model
heat_load = csv.HP()
```

The output data is saved into 'hp_trial.csv' file.

```
# Output data storage
# configure the simulator
csv_sim_writer = world.start('CSV_writer', start_date='01.01.2020 00:00', date_format=
↪'%d.%m.%Y %H:%M',
                             output_file='hp_trial.csv')
# Instantiate model
csv_writer = csv_sim_writer.CSVWriter(buff_size=60 * 60)
```

The different entities are then connected and the simulation is executed.

```
# Connect entities
world.connect(heat_load, heatpump, 'Q_Demand', 'heat_source_T', ('heat_source_T', 'T_
↪amb'), 'cond_in_T')
world.connect(heatpump, csv_writer, 'Q_Demand', 'Q_Supplied', 'heat_source_T', 'P_
↪Required', 'COP')

# Run simulation
world.run(until=END)
```

Module Documentation

This module contains a simulation model of a Heat Pump based on the library TESPy.

class mosaik_components.heatpump.Heat_Pump_Model.**Heat_Pump**(*params*, *COP_m_data*)

Simulation model of a heat pump based on the libraries **TESPy** and **hplib**.

Heat pump parameters are provided at instantiation by the dictionary **params**. The following dictionary contains the parameters that are mandatory:

```
{
    'calc_mode': 'hplib',
    'hp_model': 'LW 300(L)',
    'heat_source': 'air',
}
```

Explanation of the entries in the dictionary:

- **calc_mode**: The calculation mode that is used by the heat pump model. Currently, *'detailed'*, *'fast'*, *'hplib'*, and *'fixed'* calculation modes are available. The differences are explained in the documentation.
- **hp_model**: The specific model of the heat pump that must be simulated. The different models available currently can be found in the documentation. This need not be specified for the *'fixed'* calculation mode.
- **heat_source**: The fluid that acts as the source of heat for the heat pump, either *'water'* or *'air'*

If the **'hplib'** calculation mode is chosen, the following parameter is required in addition to the mandatory ones:

```
{
    'equivalent_hp_model': 'Air_30kW',
}
```

- **equivalent_hp_model**: The heat pump model from the saved data file whose limits of operation will be applied

Alternatively, the limits can be directly specified in the following parameter:

```
{
    'hp_limits': { 'heat_source_T_min': -10, 'heat_source_T_max': 35, 'cons_T_min
↪': 25, 'cons_T_max': 55,
                   'heatload_min': 15000 }
}
```

For the **'hplib'** calculation mode if the *'Generic'* heat pump model is chosen, the following parameters are required in addition to the mandatory ones:

```
{
    'cons_T': 35,
    'heat_source_T': 12,
    'P_th': 35000,
}
```

- **cons_T**: The temperature at which heat is supplied to the consumer (in °C).
- **heat_source_T**: The temperature at which the fluid (water or air) is available as the heat source (in °C).
- **P_th**: The heating capacity of the heat pump (in W).

If the *'fixed'* calculation mode is chosen, the following parameters are required in addition to the mandatory ones:

```
{
    'COP': 3.5,
    'heating capacity': 35000,
    'cond_m': 0.5,
}
```

- **COP**: The COP of the heat pump.
- **heating_capacity**: The heating capacity of the heat pump (in W).
- **cond_m**: The mass flow rate of water in the condenser (in kg/s).

design

stores the design of the heat pump in a `Heat_Pump_Model.Heat_Pump_Initiation` object

state

stores the state variables of the heat pump in a *Heat_Pump_Model.Heat_Pump_State* object

inputs

stores the input parameters of the heat pump model in a *Heat_Pump_Model.Heat_Pump_Inputs* object

step()

Perform simulation step with step size step_size

class mosaik_components.heatpump.Heat_Pump_Model.**Heat_Pump_Inputs**(*params*)

Inputs variables to the heat pump for each time step

Q_Demand

The heat demand of the consumer in W

heat_source_T

The temperature of the heat source (in °C)

T_amb
The ambient temperature (in °C)

cond_in_T
The temperature at which the water reenters the condenser (in °C)

step_size
step size in seconds

class mosaik_components.heatpump.Heat_Pump_Model.**Heat_Pump_State**
Attributes that define the state of the Heat_Pump

P_Required
Power consumption of the heat pump in W

COP
COP of the heat pump

Q_Demand
The heat demand of the consumer in W

Q_Supplied
The heat supplied to the consumer in W

Q_evap
The heat removed in the evaporator in W

cons_T
The temperature at which heat is supplied to the consumer (in °C)

cond_in_T
The temperature at which the water reenters the condenser (in °C)

heat_source_T
The temperature of the heat source (in °C)

T_amb
The ambient temperature (in °C)

cond_m
The mass flow rate of water in the condenser of the heat pump (in kg/s)

cond_m_neg
The negative of the mass flow rate of water in the condenser of the heat pump (in kg/s)

step_executed
The execution of the step function of the heat pump model

Hot water tank

The hot water tank model developed for another project, is used in this work to act as a buffer in between the heating device and the heat consumer. It is a multinode stratified thermal tank model, where the tank volume is divided into a specified number of layers (nodes) of equal volume, each characterized by a specific temperature. A traditional density distribution approach is adopted where the water flowing into the tank enters the layer that best matches its density (i.e., temperature). The model assumes that the fluid streams are fully mixed before leaving each of the layers and the flows between the layers follow the law of mass conservation. Heat transfer to the surrounding environment from the walls of the tank, and the heat transfer between the layers have been considered.

Parametrization of the model

The schematic of the hot water tank model is shown in the figure below. The dimensions of the tank are specified in terms of its height, and either the volume or diameter. The tank can be parametrized with sensors in the model to record its temperature. The initial temperature of all the layers must be set at the beginning of the simulation.

The flows into and out of the tank are specified as the connections of the hot water tank model. The flow going to the heat pump (*HP_out*), the space heating demand (*SH_out*), and the domestic hot water demand (*DHW_out*) are connected to the bottom layer, the fourth layer and the top layer respectively in the example schematic shown below. As explained earlier, the flows coming into the tank are not connected to a fixed layer in the tank. They are connected to the layer with a temperature closest to that of the flow.

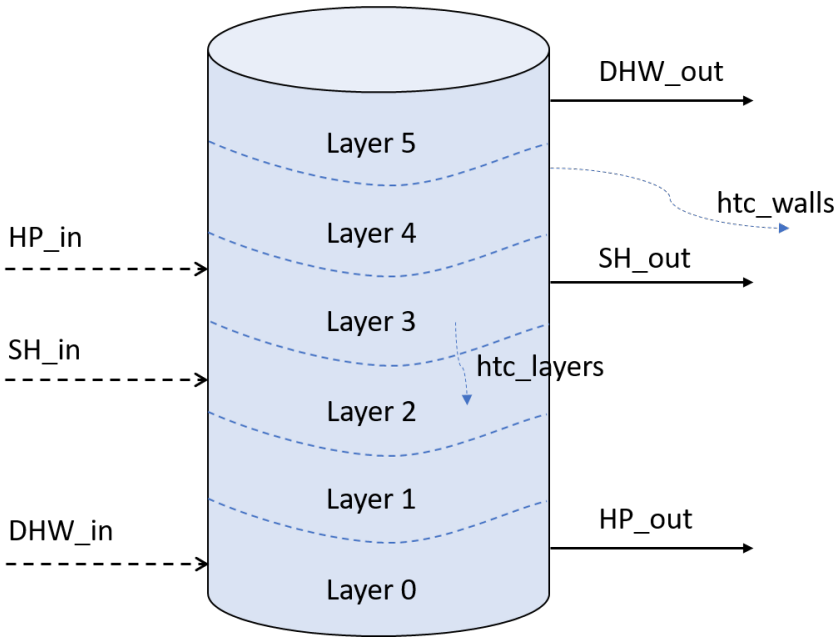


Fig. 3: Schematic representation of the hot water tank model (example with 6 layers)

The heat transfer coefficient of the walls of the tank (*htc_walls*) is assumed to be 0.28 W/m^2-K . The heat transfer coefficient for the heat transfer between the layers of the tank is assumed to be 1.5 times the thermal conductivity of water. The value is calculated as 0.897 W/m-K, considering the thermal conductivity of water to be 0.598 W/m-K. However, these values can be changed by modifying the parameters dictionary of the hot water tank model.

Calculation of the model

The initial temperature profile inside the tank can be specified at the time of initialization of the model. For flows coming into the tank, both the temperature and flow rate should be specified. For the flows going out of the tank, only the flow rate should be specified, as the temperature is obtained from the corresponding layer of the tank. The model ensures that the overall flow into and out of the tank is equal. The model then updates the temperatures of each layer based on the water flows through the specified connections, the heat transfer between the layers and the heat transfer to the surrounding environment. The model has the functionality to flip the layers to ensure a negative temperature gradient from the top to the bottom of the tank. Finally, the model updates the connections with respect to the updated layer temperatures. For the flows going out of the tank, the temperature is updated. For the flows coming into the tank, the corresponding layer is updated.

Example

An example scenario using the hot water tank simulator in the mosaik environment is available in the 'run_tank.py' file.

The simulation is configured as shown below. The inputs to the hot water tank model and the outputs from it are handled by *'mosaik-csv'*.

```
SIM_CONFIG = {
    'HotWaterTankSim': {
        'python': 'mosaik_components.heatpump.hotwatertank.hotwatertank_
↪mosaik:HotWaterTankSimulator',
    },
    'CSV': {
        'python': 'mosaik_csv:CSV',
    },
    'CSV_writer': {
        'python': 'mosaik_csv_writer:CSVWriter'
    },
}

world = mosaik.World(SIM_CONFIG)

START = '01.01.2016 00:00'
END = 7 * 15 * 60
```

The hot water tank model has one inlet connection (*'cc_in'*) and one outlet connection (*'cc_out'*). The required parameters and the initial values are set as shown below.

```
# Hot water tank
params = {
    'height': 2100,
    'diameter': 1200,
    'T_env': 20.0,
    'htc_walls': 0.28,
    'htc_layers': 0.897,
    'n_layers': 3,
    'n_sensors': 3,
    'connections': {
        'cc_in': {'pos': 1500},
        'cc_out': {'pos': 10},
        }
    }
init_vals = {
            'layers': {'T': [30, 50, 70]}
        }
# configure the simulator
hwtsim = world.start('HotWaterTankSim', step_size=15*60, config=params)
# Instantiate model
hwt = hwtsim.HotWaterTank(params=params, init_vals=init_vals)
```

The mass flow and temperature timeseries for these connections, that are needed as inputs for the model, are available in the 'tank_data.csv' file.

```
# Input data csv
HWT_FLOW_DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'data',
↪'tank_data.csv')
# configure the simulator
csv = world.start('CSV', sim_start=START, datafile=HWT_FLOW_DATA)
```

(continues on next page)

(continued from previous page)

```
# Instantiate model
csv_data = csv.HWT()
```

The output data is saved into 'hwt_trial.csv' file.

```
# Output data storage
# configure the simulator
csv_sim_writer = world.start('CSV_writer', start_date='01.01.2020 00:00', date_format=
↪'%d.%m.%Y %H:%M',
                             output_file='hwt_trial.csv')
# Instantiate model
csv_writer = csv_sim_writer.CSVWriter(buff_size=60 * 60)
```

The different entities are then connected and the simulation is executed.

```
# Connect entities
world.connect(csv_data, hwt, ('T_in', 'cc_in.T'), ('F_in', 'cc_in.F'), ('F_out', 'cc_
↪out.F'))
world.connect(hwt, csv_writer, 'cc_in.T', 'cc_in.F', 'cc_out.T', 'cc_out.F', 'sensor_
↪00.T', 'sensor_01.T', 'sensor_02.T')

# Run the simulation
world.run(until=END)  # As fast as possilbe
```

Module Documentation

The hotwatertank module contains classes for the components of the hotwatertank (*Layer*, *Connection*, *Sensor*, *HeatingRod*) and a class for the hotwater tank itself (*HotWaterTank*).

class mosaik_components.heatpump.hotwatertank.hotwatertank.**HotWaterTank**(*params*, *init_vals=None*)

Simulation model of a hotwater tank.

Stratification is modelled by a number of *Layer* objects. Heat producers and consumers can be connected to the hotwatertank via *Connection* objects. The temperature at different positions in the tank can be accessed via *Sensor* objects.

Hotwater tank parameters are provided at instantiation by the dictionary **params**. This is an example, how the dictionary might look like:

```
params = {
    'height': 2100,
    'diameter': 1200,
    'T_env': 20.0,
    'htc_walls': 1.0,
    'htc_layers': 20,
    'n_layers': 3,
    'n_sensors': 3,
    'connections': {
        'cc_in': {'pos': 0},
        'cc_out': {'pos': 2099},
        'gcb_in': {'pos': 1700},
        'gcb_out': {'pos': 500}
        },
    'heating_rods': {
```

(continues on next page)

(continued from previous page)

```
        'hr_1': {
            'pos': 1800,
            'P_th_stages': [0, 500, 1000, 2000, 3000]
            }
        }
    }
```

Explanation of the entries in the dictionary:

- **height**: height of the tank in mm
- **diameter**: diameter of the tank in mm
- **volume**: alternatively to the diameter the volume of the tank in liter can be specified
- **T_env**: ambient temperature in °C
- **htc**: heat transfer coefficient of tank walls in W/(m2K)
- **htc_layers**: imaginary heat transfer coefficient between layers in W/(m2K)
- **n_layers**: number of layers, n layers of the same dimension are created
- **n_sensors**: number of sensors, the sensors are equidistantly distributed in the hotwater tank, sensors are named 'sensor_00', 'sensor_01', …, 'sensor_n-1' with 'sensor_00' indexing the undermost sensor
- **connections**: each connection is specified by an dictionary with a structure analog to the example
- **heating_rods**: each heating rod is specified by an dictionary with a structure analog to the example

It is also possible to define layers and sensors explicitly:

```
params = {
    'height': 2100,
    'diameter': 1200,
    'T_env': 20.0,
    'htc_walls': 1.0,
    'htc_layers': 20,
    'layers': [
        {'bottom': 0, 'top': 500},
        {'bottom': 500, 'top': 1600},
        {'bottom': 1600, 'top': 2100}
        ],
    'sensors': {
        'sensor_1', {'pos': 200},
        'sensor_2', {'pos': 1900},
        },
    'connections': {
        'cc_in': {'pos': 0},
        'cc_out': {'pos': 2099},
        'gcb_in': {'pos': 1700},
        'gcb_out': {'pos': 500}
        },
    'heating_rods': {
        'hr_1': {
            'pos': 1800,
            'P_th_stages': [0, 500, 1000, 2000, 3000]
            }
        }
    }
```

Initial values for the temperature distribution in the tank or the initial el. power of the heating rod are provided by the dictionary init_vals, which might look like this:

```
{
    'layers': {'T': 50},
    'hr_1': { 'P_el': 2000}
}
```

this:

```
{
    'layers': {'T': [30, 50, 70]}
    'hr_1': { 'P_el': 2000}
}
```

or this:

```
{
    'layers': {'T': [30, 70]},
    'hr_1': { 'P_el': 2000}
}
```

- **T**: initial temperature of tank in °C, alternatively a temperature range can be specified, whereby the lower limit defines the temperature of ther undermost layer and the upper limit the temperature of the uppermost layer, inbetween a linear temperature gradient is set, it is also possible to specify the temperature of each layer individually by passing a list of length n_layers

step(*step_size*, *adapted_step_size_mode=False*)

Perform simulation step with step size step_size

property snapshot

serialize to json

property snapshot_connections

serialize connections to json

property T_mean

Returns mean temperature of hotwatertank in °C

class mosaik_components.heatpump.hotwatertank.hotwatertank.**Layer**(*params*)

Layer of hotwater tank

Parameters

layer_params – dictionary containing the following keys

- **T** - initial temperature of layer in °C
- **bottom** - bottom of layer relatively to hotwater tank bottom in mm
- **top** - top of layer relatively to hotwater tank bottom in mm
- **diameter** - diameter of layer in mm
- **bottom_top** - must be True for the bottom or top layer of the tank. This information is needed to calculate the outer surface of the layer which in turn is needed to calculate heat losses to the environment.

class mosaik_components.heatpump.hotwatertank.hotwatertank.**Sensor**(*params*, *layers*)

Temperature sensor in the tank.

Parameters

params – dictionary containing params which specify a sensor, so far it contains only one entry *pos*, which defines the position of the sensor above hotwater tank bottom

class mosaik_components.heatpump.hotwatertank.hotwatertank.**Connection**(*params*, *layers*)

Devices are connected to the hotwater tank via connections.

Each connection is associated with a *Layer*. For input connections (F>0) the corresponding layer is determined by temperature comparison. The layer whose temperature is closest to the connection temperature is the corresponding one. For output connections (F<0) the corresponding layer depends on the position of the connection. The corresponding layer of a connection is not fix, but may change during the simulation, if the flow or temperatures of the connection or the temperature of the layers changes.

class mosaik_components.heatpump.hotwatertank.hotwatertank.**MassFlow**(*F*, *T*)

Massflow

class mosaik_components.heatpump.hotwatertank.hotwatertank.**HeatingRod**(*params*, *layers*, *init_vals=None*)

Heating rod integrated into to the hotwater tank.

Heating rods are characterized by their position above tank level and their power stages. Efficiency is assumed to be constantly 100%.

Controller

The controller model used in this work utilizes simple Boolean logic to:

1. Match the heating demands with the supply from the hot water tank/back up heaters
2. Control the operation of the heat pump and adjust the mass flows in the heat pump circuit
3. Control the in-built heating rod in the hot water tank

In order to perform each of the above functions, the controller must be initialized with a set of parameters. The controller then analyzes the information received from the different models and sends the necessary information back to the models for the progression of the simulation. The specific parameters and the information exchange for each of the functions are detailed below.

Domestic hot water demand

The controller is initialized with a set point for the domestic hot water (DHW) supply temperature (*T_hr_sp_dhw*). For each time step of the simulation, the controller receives the domestic hot water demand (*dhw_demand*), in liters of water, the temperature of the water available for supply from the hot water tank (*dhw_out_T*), and the temperature of the cold inlet water (*dhw_in_T*). The controller then calculates the flow rate of water to be supplied from the tank by dividing the demand with the time step (*dhw_out_F*). If the water from the tank is available for supply at a temperature greater than the set point, flow is adjusted by mixing with the cold inlet water. If the temperature of water from tank is lower than the set point, the controller calculates the heat to be supplied by the backup heater to achieve the set point temperature, which is also the electric power required (*P_hr_dhw*) by the heater, assuming 100% efficiency. Finally, the controller sets the information of the flows to the hot water tank (*dhw_out_F, dhw_in_F*).

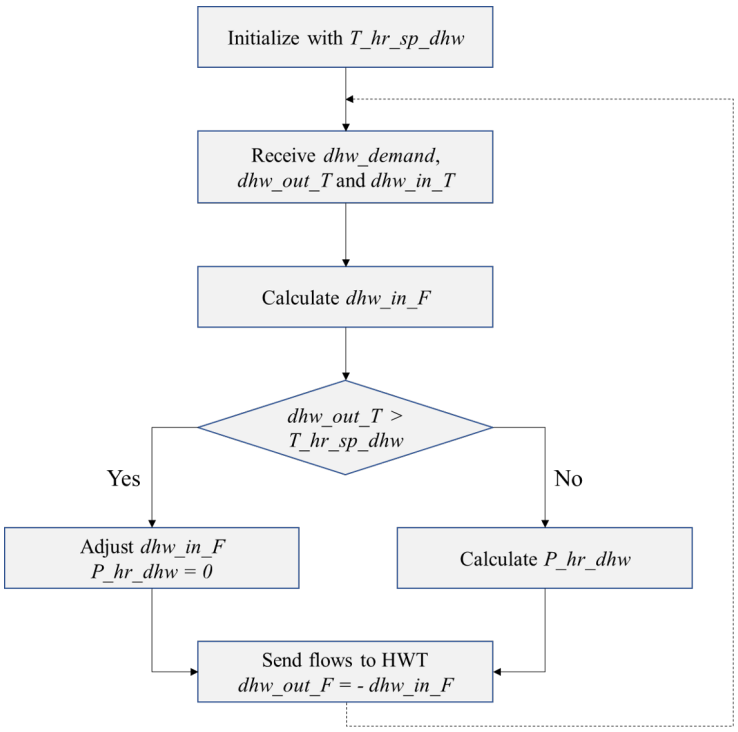


Fig. 4: Control flow for domestic hot water demand

Space heating demand

The controller is initialized with the set points for the space heating (SH) supply temperature (*T_hr_sp_sh*) and the temperature difference in the space heating circuit (*sh_dT*). For each time step of the simulation, the controller receives the space heating demand (*sh_demand*), in kW, the temperature of the water available for supply from the hot water tank (*sh_out_T*). The controller then calculates the flow rate of water to be supplied from the tank from the demand and the temperature difference (*sh_in_F*). If the water from the tank is available for supply at a temperature greater than the set point, the return temperature (*sh_in_T)* is calculated as the difference between the supply temperature and the temperature difference. If the temperature of water from tank is lower than the set point, the controller calculates the heat to be supplied by the backup heater to achieve the set point temperature, which is also the electric power required (*P_hr_sh*) by the heater, assuming 100% efficiency. The controller also calculates the return temperature as the difference between the set point temperature and the temperature difference. Finally, the controller sets the information of the flows to the hot water tank (*sh_out_F, sh_in_F, sh_in_T*).

Heat pump operation -control strategy 1

A simple hysteresis control based on the temperature of the bottom layer of the hot water tank has been implemented for the operation of the heat pump. The controller is initialized with a higher (*T_hp_sp_h)* and lower (*T_hp_sp_l)* temperature set point for the hot water tank. The controller then receives the temperature of the bottom layer (*bottom_layer_T*) of the hot water tank. The bottom layer of the tank is controlled to be maintained in between the two temperature limits, i.e., the heat pump is turned on when the temperature in the bottom layer of the tank falls below the lower set point. The heat pump is turned off only when the temperature in the bottom layer of the tank is greater than the higher set point. The heat pump continues to remain turned off and turns back on only when the temperature of the bottom layer falls below the lower set point again. The heat pump is controlled by setting its status (*hp_status*) to either '*on*' or '*off*' based on the control logic explained above. The heat demand from the heat pump (*hp_demand*) is calculated to be sent to the heat pump.

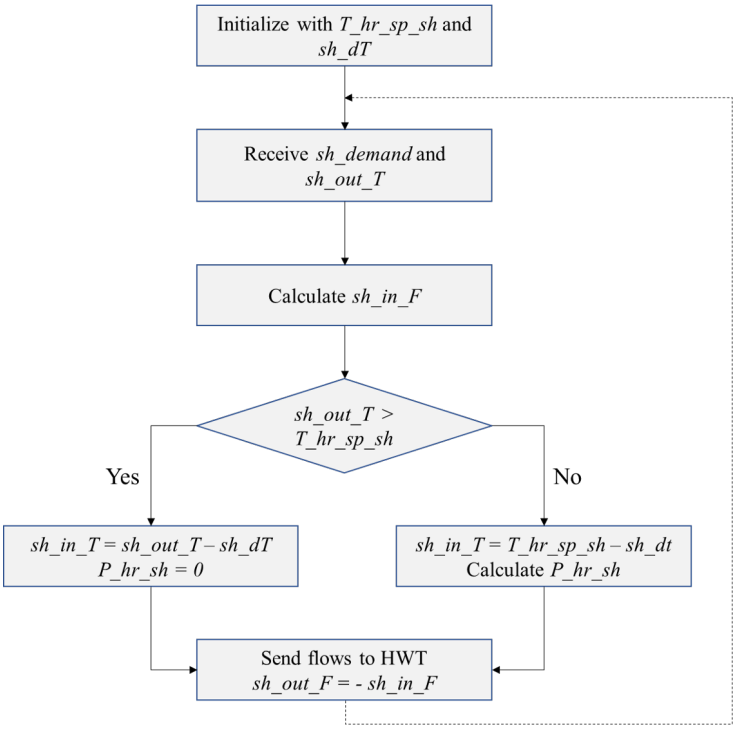


Fig. 5: Control flow for space heating demand

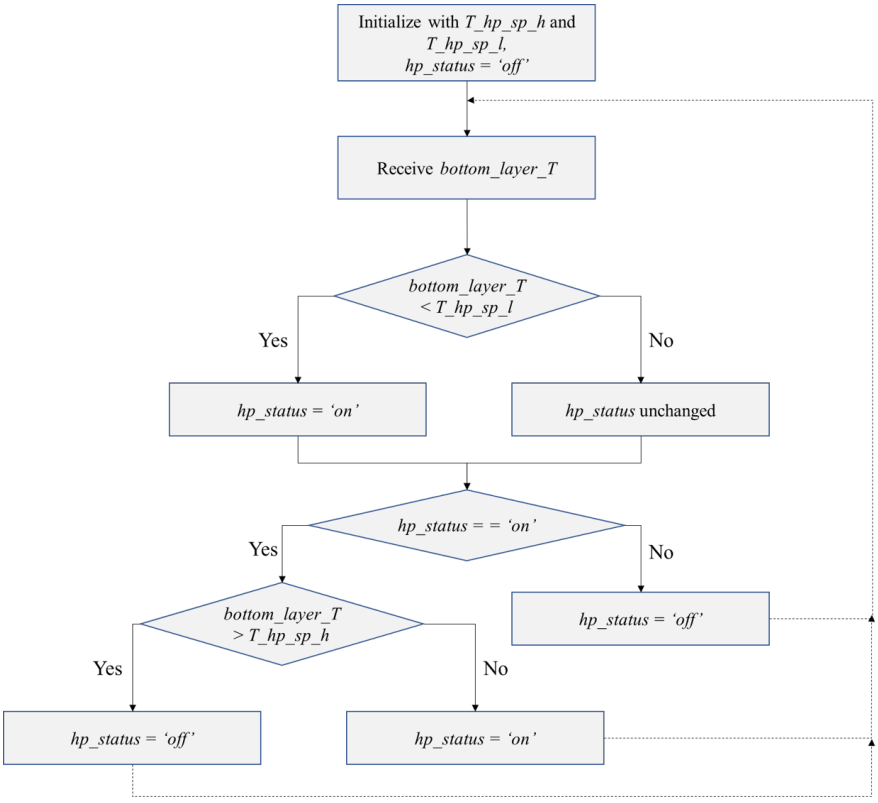


Fig. 6: Control strategy 1, for the operation of heat pump

Heat pump operation -control strategy 2

In addition to the control strategy for the heat pump operation explained earlier, which is based only on the temperature of the bottom layer of the hot water tank, a control strategy based on the temperatures of both the bottom and top layers of the tank has been implemented. The controller is initialized with a higher (*T_hp_sp_h)* and lower (*T_hp_sp_l)* temperature set point for the hot water tank, as done in the first control strategy. The controller then receives the temperatures of the top layer (*top_layer_T*) and the bottom layer (*bottom_layer_T*) of the hot water tank. The top layer of the tank is controlled against the higher set point, i.e., the heat pump is turned on when the temperature in the top layer of the tank falls below the higher set point. The bottom layer of the tank is controlled against the lower set point, i.e., the heat pump is turned off only when the temperature in the bottom layer of the tank is greater than the lower set point. In this case, the temperature of the top layer is expected to be greater than the higher set point due to stratification inside the tank. The heat pump continues to remain turned off and turns back on only when the temperature of the top layer falls below the higher set point again. The heat pump is controlled by setting its status (*hp_status*) to either '*on*' or '*off*' based on the control logic explained above. The heat demand from the heat pump (*hp_demand*) is calculated to be sent to the heat pump.

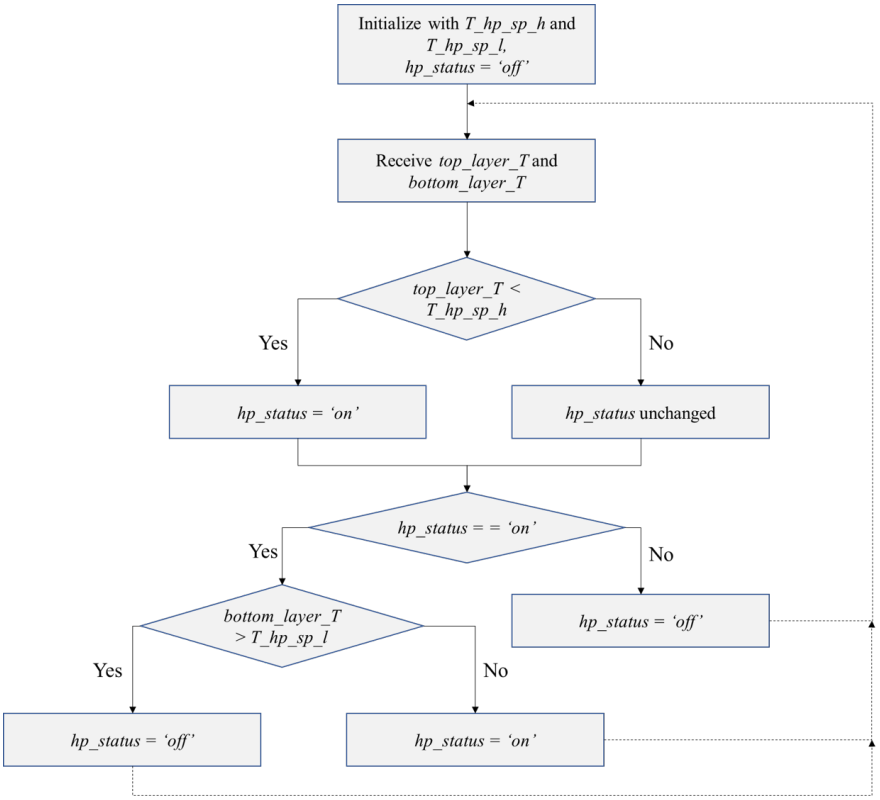


Fig. 7: Control strategy 2, for the operation of heat pump

Module Documentation

The controller module contains a class for the controller model (*Controller*).

class mosaik_components.heatpump.controller.controller.**Controller**(*params*)

Simulation model of a controller.

The controller model used in this work utilizes simple Boolean logic to:

1. Match the heating demands with the supply from the hot water tank/back up heaters
2. Control the operation of the heat pump using different control strategies

Controller parameters are provided at instantiation by the dictionary **params**. This is an example, how the dictionary might look like:

```
params = {
    'T_hp_sp_h': 50,
    'T_hp_sp_l': 40,
    'T_hr_sp_hwt': 40,
    'T_hr_sp_dhw': 40,
    'T_hr_sp_sh': 35,
    'dhw_in_T': 10,
    'sh_dT': 7,
    'operation_mode': 'heating',
    'control_strategy': '1'
    }
```

Explanation of the entries in the dictionary:

- **T_hp_sp_h**: The higher temperature set point for heat pump operation (in °C)
- **T_hp_sp_l**: The lower temperature set point for heat pump operation (in °C)
- **T_hr_sp_hwt**: The temperature set point for the back up heater within the hot water tank (in °C)
- **T_hr_sp_dhw**: The temperature set point for the back up heater for domestic hot water supply (in °C)
- **T_hr_sp_sh**: The temperature set point for the back up heater for space heating supply (in °C)
- **dhw_in_T**: The default temperature of cold water inlet for domestic hot water (in °C)
- **sh_dT**: The temperature difference of the water in the space heating supply circuit (in °C)
- **operation_mode**: The operation mode of the heating system, either 'heating' or 'cooling'
- **control_strategy**: The control strategy to be used for the heat pump operation. Currently, two strategies have been implemented ('1' & '2')

step()

Perform simulation step with step size step_size

calc_dhw_supply()

Calculate the mass flows and temperatures of water, and the heat from the back up heater in the domestic hot water (DHW) circuit

calc_sh_supply()

Calculate the mass flows and temperatures of water, and the heat from the back up heater in the space heating (SH) circuit

The package contains the models for the following components of the heating system:

1. *Heat pump*

2. *Hot water tank*
3. *Controller*

4.1.2 Example scenarios

Example scenarios of the co-simulation of the heat pump, hot water tank and controller models are available in the examples folder.

There are cyclic dependencies between the models for each time step, for ex., the hot water tank needing the information from the controller regarding the demands and the water flows, and the controller needing information from the hot water tank regarding the temperature of the water to calculate the flows. mosaik offers two different ways to resolve such cyclic dependencies. The first is the *time-shifted* resolution, where the information from one model is passed to the other model in the next time step. The second is the *same-time-loop* resolution, where the information exchange between the models is done in the same time step before progressing the simulation to the next time step. The mosaik documentation describes these two ways of dealing with cyclic dependencies in detail (*cyclic-data-flows*).

The user can choose between the two types of execution, by specifying the parameter *'same_time_loop'*, while initializing the simulators for each of the models. The default execution mode is the *time-shifted* resolution. For the *same-time-loop* resolution, the parameter *'same_time_loop'* has to be set to *'True'*. Depending on the type of execution, the way the connections between the different models are setup varies, and can be seen in the example scenarios below.

Note: All the simulators must be set to the same type of execution

Time-shifted resolution

The first example scenario uses the *time-based* resolution of the cyclic dependencies offered by mosaik. The different heat pumps, and calculation modes available in the heat pump model are simulated along with the hot water tank, with the controller model matching both the space heating and domestic hot water demand with the heat available in the hot water tank and controlling the operation of the heat pump.

The simulation is configured as shown below. The inputs/outputs to/from the models are handled by *'mosaik-csv'*.

```
sim_config = {
    'CSV': {
        'python': 'mosaik_csv:CSV',
    },
    'CSV_writer': {
        'python': 'mosaik_csv_writer:CSVWriter'
    },
    'HeatPumpSim': {
        'python': 'mosaik_components.heatpump.Heat_Pump_mosaik:HeatPumpSimulator',
    },
    'HotWaterTankSim': {
        'python': 'mosaik_components.heatpump.hotwatertank.hotwatertank_
↪mosaik:HotWaterTankSimulator',
    },
    'ControllerSim': {
        'python': 'mosaik_components.heatpump.controller.controller_
↪mosaik:ControllerSimulator',
    },
}

# The start date, duration, and step size for the simulation
```

(continues on next page)

(continued from previous page)

```
END = 10 * 60
START = '01.01.2020 00:00'
STEP_SIZE = 60 * 1
```

The parameters and/or initial values for the different models are specified.

```
#Parameters for mosaik-heatpump
params_hp = {'hp_model': 'Air_30kW_1stage',
             'heat_source': 'Air',
             'cons_T': 35,
             'Q_Demand': 19780,
             'cond_in_T': 30,
             'heat_source_T': 7,
             }
#Parameters for hot water tank model
params_hwt = {
    'height': 3600,
    'volume': 4000,
    'T_env': 20.0,
    'htc_walls': 0.28,
    'htc_layers': 0.897,
    'n_layers': 6,
    'n_sensors': 6,
    'connections': {
        'sh_in': {'pos': 10},
        'sh_out': {'pos': 2150},
        'dhw_in': {'pos': 10},
        'dhw_out': {'pos': 3400},
        'hp_in': {'pos': 10},
        'hp_out': {'pos': 500},
        },
    }
init_vals_hwt = {
            'layers': {'T': [40.0, 40.0, 40.0, 40.0, 40.0, 40.0]}
        }
#Parameters for controller model
params_ctrl = {
    'T_hp_sp_h': 50,
    'T_hp_sp_l': 40,
    'T_hr_sp_dhw': 40,
    'T_hr_sp_sh': 35,
    'dhw_in_T': 10,
    'sh_dT': 7,
    'operation_mode': 'heating',
    'control_strategy': '1'
}
```

The different types of heat pumps and calculation modes that are simulated are specified.

```
# The different types of heat pumps and calculation modes that are simulated
model_list = ['Air_30kW_1stage', 'Air_30kW_1stage', 'LW 300(L)', None]
calc_mode_list = ['detailed', 'fast', 'hplib', 'fixed']
filename_list = ['detailed', 'fast', 'hplib', 'fixed']
```

The mosaik *'world'*, and the simulators of the different models are initialized. The inputs required for the different models – domestic hot water demand (*DHW Demand*); space heating demand (*SH Demand*); the heat source temperature, which is the ambient air in this case (*T_amb*); and the temperature of the cold water replacing the domestic hot water supplied

from the tank (*dhw_in_T*) – are available in the 'scenario_data.csv' file. The inputs and the outputs are handled by *'mosaik-csv'* and the output data is saved in csv files.

```
    # Initialize the world and the simulators.

    world = mosaik.World(sim_config)

    heatpumpsim = world.start('HeatPumpSim', step_size=STEP_SIZE)

    hwtsim = world.start('HotWaterTankSim', step_size=STEP_SIZE, config=params_hwt)

    ctrlsim = world.start('ControllerSim', step_size=STEP_SIZE)

    heat_load_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'data',
↪'scenario_data.csv')
    heat_load_sim = world.start('CSV', sim_start=START, datafile=heat_load_file)

    CSV_File = 'Scenario_' + filename_list[i] + '_time_shifted.csv'
    csv_sim_writer = world.start('CSV_writer', start_date='01.01.2020 00:00', date_
↪format='%d.%m.%Y %H:%M',
                                 output_file=CSV_File)
```

The specific parameters for the different heat pump models and calculation modes are added to the parameters.

```
    params_hp['calc_mode'] = calc_mode_list[i]
    params_hp['hp_model'] = model_list[i]

    if 'hplib' in params_hp['calc_mode']:
        params_hp['equivalent_hp_model'] = 'Air_30kW_1stage'
    elif 'fixed' in params_hp['calc_mode']:
        params_hp['COP'] = 3.5
        params_hp['heating capacity'] = 15000
        params_hp['cond_m'] = 0.5
```

The different models are instantiated.

```
    # Instantiate the models
    heatpumps = heatpumpsim.HeatPump.create(1, params=params_hp)

    hwts = hwtsim.HotWaterTank.create(1, params=params_hwt, init_vals=init_vals_hwt)

    ctrls = ctrlsim.Controller.create(1, params=params_ctrl)

    heat_load = heat_load_sim.HEATLOAD.create(1)

    csv_writer = csv_sim_writer.CSVWriter(buff_size=60 * 60)
```

The cyclic data flows between the different models are then set up in the time-shifted manner and the simulation is executed.

```
    # connections between the different models
    world.connect(heat_load[0], ctrls[0], 'T_amb', ('T_amb', 'heat_source_T'), ('SH␣
↪Demand [kW]', 'sh_demand'),
                  ('DHW Demand [L]', 'dhw_demand'), 'dhw_in_T')

    world.connect(hwts[0], ctrls[0], ('T_mean', 'T_mean_hwt'), ('mass', 'hwt_mass'),
                  ('sensor_00.T', 'bottom_layer_T'), ('sensor_04.T', 'top_layer_T'),
                  ('dhw_out.T', 'dhw_out_T'), ('sh_out.T', 'sh_out_T'), ('hp_out.T',
```

(continues on next page)

(continued from previous page)

```
↪'hp_out_T'))
123
124     world.connect(ctrls[0], hwts[0], ('sh_in_F', 'sh_in.F'), ('sh_in_T', 'sh_in.T'), (
↪'sh_out_F', 'sh_out.F'),
125                   ('dhw_in_F', 'dhw_in.F'), ('dhw_in_T', 'dhw_in.T'), ('dhw_out_F',
↪'dhw_out.F'), ('T_amb', 'T_env'),
126                   time_shifted=True,
127                   initial_data={'sh_in_F': 0, 'sh_in_T': 0, 'sh_out_F': 0,
128                                 'dhw_in_F': 0, 'dhw_in_T': 0, 'dhw_out_F': 0,
129                                 'T_amb': 0,
130                                 },
131                   )
132
133     world.connect(heatpumps[0], ctrls[0], ('Q_Supplied', 'hp_supply'), ('on_fraction',
↪ 'hp_on_fraction'),
134                   ('cond_m', 'hp_cond_m'))
135
136     world.connect(ctrls[0], heatpumps[0], ('hp_demand', 'Q_Demand'),
137                   'T_amb', 'heat_source_T', time_shifted=True,
138                   initial_data={'hp_demand': 0, 'T_amb': 5, 'heat_source_T': 5})
139
140     world.connect(hwts[0], heatpumps[0], ('hp_out.T', 'cond_in_T'))
141
142     world.connect(heatpumps[0], hwts[0], ('cons_T', 'hp_in.T'), ('cond_m', 'hp_in.F'),
↪ ('cond_m_neg', 'hp_out.F'),
143                   time_shifted=True, initial_data={'cons_T': 0, 'cond_m': 0, 'cond_m_
↪neg': 0})
144
145     world.connect(heat_load[0], csv_writer, 'T_amb', 'SH Demand [kW]', 'DHW Demand [L]
↪')
146     world.connect(heatpumps[0], csv_writer, 'Q_Demand', 'Q_Supplied', 'T_amb', 'heat_
↪source_T', 'cons_T', 'P_Required',
147                   'COP', 'cond_m', 'cond_in_T', 'on_fraction')
148
149     world.connect(ctrls[0], csv_writer, 'heat_demand', 'heat_supply', 'hp_demand',
↪'sh_supply', 'sh_demand', 'hp_supply',
150                   'sh_in_F', 'sh_in_T', 'sh_out_F', 'sh_out_T', 'dhw_in_F', 'dhw_in_T
↪', 'dhw_out_F', 'dhw_out_T',
151                   'hp_in_F', 'hp_in_T', 'hp_out_F', 'hp_out_T', 'P_hr_sh', 'P_hr_dhw',
↪ 'dhw_demand', 'dhw_supply')
152     world.connect(hwts[0], csv_writer, 'sensor_00.T', 'sensor_01.T', 'sensor_02.T',
↪'sensor_03.T', 'sensor_04.T',
153                   'sensor_05.T', 'sh_out.T', 'sh_out.F', 'dhw_out.T', 'dhw_out.F',
↪'hp_in.T', 'hp_in.F', 'hp_out.T', 'hp_out.F',
154                   'T_mean', 'sh_in.T', 'sh_in.F', 'dhw_in.T', 'dhw_in.F')
155
156     #Run
157     world.run(until=END)
```

Same-time-loop resolution

The second example scenario uses the *event-based* resolution of the same-time-loop cycles offered by mosaik. Only the things that need to be changed when compared to the *time-based* resolution are shown below.

While initializing the model simulators, the *'same_time_loop'* parameter has to be set to *'True'* for all the models.

```
    heatpumpsim = world.start('HeatPumpSim', step_size=STEP_SIZE, same_time_loop=True)

    hwtsim = world.start('HotWaterTankSim', step_size=STEP_SIZE, config=params_hwt,␣
↪same_time_loop=True)

    ctrlsim = world.start('ControllerSim', step_size=STEP_SIZE, same_time_loop=True)
```

The cyclic data flows between the different models are then set up in the same-time-loop manner.

```
    # connections between the different models
    world.connect(heat_load[0], ctrls[0], 'T_amb', ('T_amb', 'heat_source_T'), ('SH␣
↪Demand [kW]', 'sh_demand'),
                  ('DHW Demand [L]', 'dhw_demand'), 'dhw_in_T')

    world.connect(hwts[0], ctrls[0], ('T_mean', 'T_mean_hwt'), ('mass', 'hwt_mass'),
                  ('sensor_00.T', 'bottom_layer_T'), ('sensor_04.T', 'top_layer_T'),
                  ('dhw_out.T', 'dhw_out_T'), ('sh_out.T', 'sh_out_T'),
                  ('hp_out.T', 'hp_out_T'))

    world.connect(ctrls[0], hwts[0], ('sh_in_F', 'sh_in.F'), ('sh_in_T', 'sh_in.T'), (
↪'sh_out_F', 'sh_out.F'),
                  ('dhw_in_F', 'dhw_in.F'), ('dhw_in_T', 'dhw_in.T'), ('dhw_out_F',
↪'dhw_out.F'), ('T_amb_hwt', 'T_env'),
                  ('hp_in_T', 'hp_in.T'), ('hp_in_F', 'hp_in.F'), ('hp_out_F', 'hp_
↪out.F'), weak=True)

    world.connect(heatpumps[0], ctrls[0], ('Q_Supplied', 'hp_supply'), ('on_fraction',
↪ 'hp_on_fraction'),
                  ('cond_m', 'hp_in_F'), ('cond_m_neg', 'hp_out_F'), ('cons_T', 'hp_
↪in_T'), weak=True)

    world.connect(ctrls[0], heatpumps[0], ('hp_demand', 'Q_Demand'), ('hp_out_T',
↪'cond_in_T'),
                  'T_amb', 'heat_source_T')


    world.connect(heat_load[0], csv_writer, 'T_amb', 'SH Demand [kW]', 'DHW Demand [L]
↪')
    world.connect(heatpumps[0], csv_writer, 'Q_Demand', 'Q_Supplied', 'T_amb', 'heat_
↪source_T', 'cons_T', 'P_Required',
                  'COP', 'cond_m', 'cond_in_T', 'on_fraction')

    world.connect(ctrls[0], csv_writer, 'heat_demand', 'heat_supply', 'hp_demand',
↪'sh_supply', 'sh_demand', 'hp_supply',
                  'sh_in_F', 'sh_in_T', 'sh_out_F', 'sh_out_T','dhw_in_F', 'dhw_in_T',
↪ 'dhw_out_F', 'dhw_out_T',
                  'hp_in_F', 'hp_in_T', 'hp_out_F', 'hp_out_T',  'P_hr_sh', 'P_hr_dhw
↪', 'dhw_demand', 'dhw_supply')
    world.connect(hwts[0], csv_writer, 'sensor_00.T', 'sensor_01.T', 'sensor_02.T',
↪'sensor_03.T', 'sensor_04.T', 'sensor_05.T',
                  'sh_out.T', 'sh_out.F', 'dhw_out.T', 'dhw_out.F', 'hp_in.T', 'hp_in.
```

(continues on next page)

(continued from previous page)

```
↪F', 'hp_out.T', 'hp_out.F',
                  'T_mean', 'sh_in.T', 'sh_in.F', 'dhw_in.T', 'dhw_in.F')
```

For the same-time-loop execution, it is important to set the initial event that kick-starts the simulation, which is the simulation of the hot water tank for this scenario. The simulation is then executed.

```
    # To start hwts as first simulator
    world.set_initial_event(hwts[0].sid)

    #Run
    world.run(until=END)
```

4.1.3 Integrating new heat pumps

Initial Parametrization

The tutorial provided in TESPy's documentation for simulating heat pumps has been followed to develop the first design calculation, available in the 'Parametrization_NominalData.py' file. While the tutorial provides a detailed explanation of the complete parametrization of the model, only the most relevant parameters are discussed here.

The data obtained from the manufacturer's datasheet corresponding to the nominal operating point is shown in table below.

Table 1: Nominal operating point data

Parameter	Value	Units
Condenser inlet temperature	30	°C
Condenser outlet temperature	35	°C
Source air temperature	7	°C
Temperature difference for air in evaporator	5	°C
Heating capacity	32.5	kW
Electrical Power	8.56	kW
Refrigerant	R448A	–

This data has been used to set the parameters of the corresponding components and connections as described in the tutorial. Since the refrigerant R448A is not available in TESPy, R404A has been used due to the similarity in their properties (reference).

TESPy uses the isentropic efficiency of the compressor to calculate the power consumption as shown in equation below.

$$P = \frac{\dot{m}_{in} \cdot (h_{out,s} - h_{in})}{\eta_s}$$

P- power consumption of the compressor; $\dot{m}_{in}$- mass flow in the compressor,
$(h_{out,s} - h_{in})$- enthalpy change in an isentropic compression process, η_s- isentropic efficiency of the compressor.

Since the isentropic efficiency of the compressor (*'eta_s'*) is not available in the datasheet, the value has been changed on a trial-and-error basis to match the power consumption calculated by the model to that mentioned in the datasheet.

Note: For heat pumps having two stages of compression, like the one in this example, in addition to the isentropic efficiency, the pressure ratio (*'pr'*) for the second stage is also a required parameter.

This has to be adjusted on a trial and error basis as well, so that it works for the different range of operating conditions of the heat pump. This could be checked, by choosing data points from the edges of the operation range, and following the same procedure as done for nominal operating point data.

Compressor efficiency map

The design case from the previous step can be used as the basis for *offdesign* calculations to predict the system's performance (in terms of Electrical power/COP) at different operating conditions, i.e., a different source or consumer inlet temperature. While the tutorial in TESPy's documentation provides a detailed explanation of all the changes between the two modes of calculations, only the most relevant changes are discussed here.

The refrigerant and the temperature difference for air in the evaporator remain unchanged. The source air temperature and the condenser inlet temperature are available as inputs to the model. The maximum heating capacity at a particular source air temperature is available from the datasheet. The condenser outlet temperature is not known in the off-design case, but must be predicted. In the design mode, the heating capacity and temperature difference between condenser inlet and outlet are used to calculate the mass flow in the circuit. This mass flow is used to calculate the condenser outlet temperature in the off-design mode.

In order to determine the electrical power consumption, the isentropic efficiency of the compressor is required. Since this is not available in the datasheets for the entire range of operation, the default characteristic curve available in the TESPy library, shown in figure below, is used.

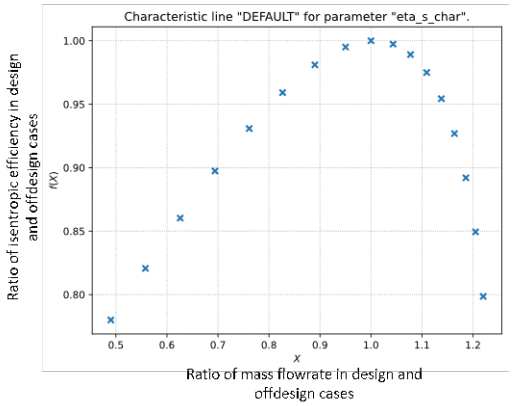


Fig. 8: Default characteristic curve for the isentropic efficiency of the compressor in TESPy library

The x-axis is the ratio of the mass flow into the compressor in the design and off-design cases, and the y-axis is the ratio of the isentropic efficiencies in the design and off-design conditions.

The default characteristic curve is generic and therefore does not accurately reflect the performance of the specific model of the heat pump chosen. Instead of relying on the isentropic efficiency from a single design point and the default characteristic curve across the entire range of operation, a series of design points have been developed based on the data available in the manufacturer's datasheet for the operating conditions shown in table 3.3.

Table 2: Design point conditions

Source air temperatures (°C)	-20, -15, -12, -10, -7, -2, 2, 7, 10, 12, 15, 20, 25, 30, 35
Condenser outlet temperatures (°C)	15, 20, 25, 30, 35, 40, 45, 50, 55

The operating range of the heat pump for the source air temperature is -20°C to 35°C. The actual operation range of the heat pump on the condenser outlet temperature is 25°C to 60°C. In the model, the range is further increased to 15°C to

60°C, in order to simulate low temperature lift conditions. The temperature difference in the condenser, constant at 5°C in the design case, has been used to calculate the condenser inlet temperature.

Extension of the heating capacity table of the heat pump

The manufacturer's datasheets contain the heating power curves and the electrical power curves as shown in figure below.

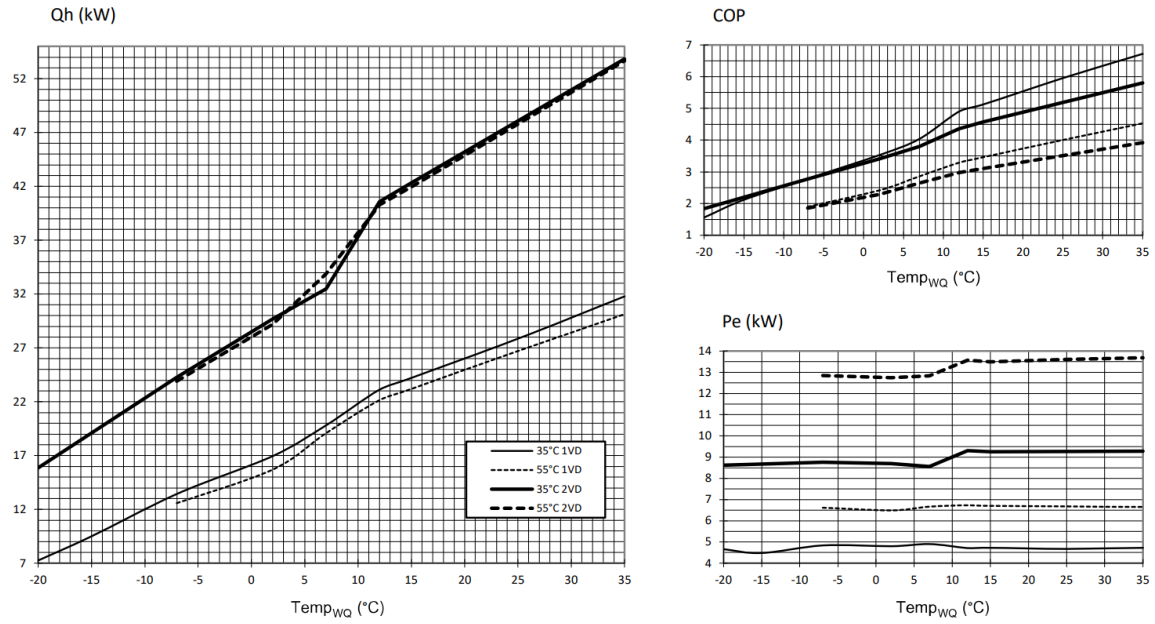


Fig. 9: Heating capacity, electrical power, and COP curves of the chosen heat pump

In all the plots, the x-axis corresponds to the source (air) temperature. The data is available for the entire range of source air temperature, but only for two condenser outlet temperatures, 35°C and 55°C.

The heating capacity increases with an increase in the source air temperature, but does not change significantly with a change in the condenser outlet temperature. At a given source air temperature, the heating capacity for all the other condenser outlet temperatures is assumed to be the average of the heating capacities at 35°C and 55°C.

The power consumption changes with both the source air temperature and the condenser outlet temperature. An approach based on Carnot efficiency has been used to predict the power consumption at the condenser outlet temperatures other than 35°C and 55°C. The ideal COP is calculated for all the operating points, using the equation below (note that the temperatures have to be in Kelvin scale).

$$COP_{ideal} = \frac{T_h}{T_h - T_c}$$

where T_h is the condenser water outlet temperature, T_c is the source air temperature, and COP_{ideal} is the ideal coefficient of performance of the heat pump

Fig. 10: Equation for ideal COP

For the operating points where the power consumption/COP is known, the Carnot efficiency has been calculated using the following equation.

$$\eta_{carnot} = \frac{COP_{real}}{COP_{ideal}}$$

where COP_{real} is the real coefficient of performance of the heat pump, and η_{carnot} is the Carnot efficiency of the heat pump

Fig. 11: Equation for Carnot efficiency

The temperature lift for all the operating points is calculated using equation below

$$T_{lift} = T_h - T_c$$

where T_{lift} is the temperature lift

Fig. 12: Equation for temperature lift

A second order polynomial equation has been fit to the pairs of the Carnot efficiencies and the corresponding temperature lifts, of operating points with condenser outlet temperatures 35°C and 55°C, as shown in figure below.

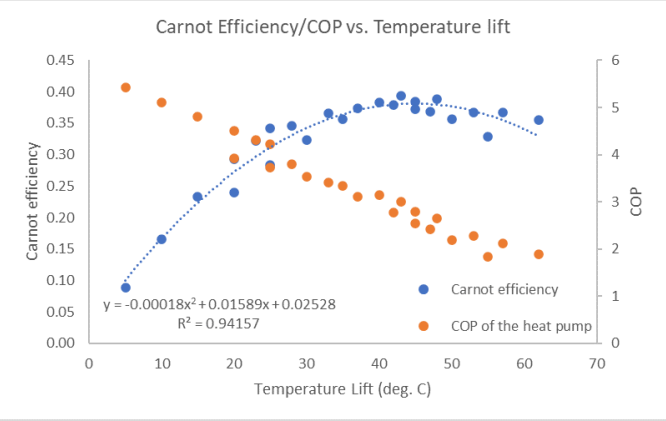


Fig. 13: Carnot efficiency/COP vs Temperature Lift plot for the chosen heat pump

In this figure, the COP of the heat pump is also plotted against the temperature lift. The Carnot efficiencies of the remaining operating points are estimated using the fit equation, which in turn are used to estimate the real COP/power consumption.

For the series of design points identified, the calculated heating capacity and power consumption data is summarized in the table below.

The heating capacity data has to be saved in the 'Heat_Load_Data.csv' file and the power consumption data has to be saved in the 'PI_Data.csv' file.

Heating Capacity (kW) / Power Consumption (kW)										
Source air temperature (deg. C)	Condenser outlet temperature (deg. C)									
	15	20	25	30	35	40	45	50	55	60
-20	15.8 / 5.27	15.8 / 5.72	15.8 / 6.26	15.8 / 6.92	15.8 / 8.6	15.8 / 8.9	15.8 / 10.45	-	-	-
-15	19 / 5.78	19 / 6.23	19 / 6.76	19 / 7.4	19 / 8.65	19 / 9.2	19 / 10.53	-	-	-
-12	21 / 6.06	21 / 6.51	21 / 7.04	21 / 7.66	21 / 8.7	21 / 9.37	21 / 10.6	-	-	-
-10	22.3 / 6.21	22.3 / 6.67	22.3 / 7.19	22.3 / 7.8	22.3 / 8.75	22.3 / 9.46	22.3 / 10.63	22.3 / 12.17	-	-
-7	24.28 / 6.42	24.28 / 6.88	24.28 / 7.4	24.28 / 8	24.28 / 8.75	24.28 / 9.58	24.28 / 10.67	24.28 / 12.07	24.28 / 12.9	-
-2	27.1 / 6.55	27.1 / 7.04	27.1 / 7.55	27.1 / 8.13	27.2 / 8.75	27.1 / 9.57	27.1 / 10.53	27.1 / 11.72	27 / 12.8	27.1 / 15.34
2	29.39 / 6.57	29.39 / 7.11	29.39 / 7.64	29.39 / 8.2	29.67 / 8.7	29.39 / 9.56	29.39 / 10.44	29.39 / 11.51	29.1 / 12.8	29.39 / 14.6
7	33.25 / 6.55	33.25 / 7.31	33.25 / 7.91	33.25 / 8.5	32.5 / 8.55	33.25 / 9.83	33.25 / 10.65	33.25 / 11.63	34 / 12.8	33.25 / 14.32
10	38.47 / 6.66	38.47 / 7.89	38.47 / 8.65	38.47 / 9.33	39.43 / 9.35	38.47 / 10.77	38.47 / 11.62	38.47 / 12.63	37.5 / 13.4	38.47 / 15.34
12	40.5 / 5.91	40.5 / 7.84	40.5 / 8.75	40.5 / 9.48	40.5 / 9.4	40.5 / 10.94	40.5 / 11.79	40.5 / 12.77	40.5 / 13.5	40.5 / 15.38
15	-	42.4 / 7.21	42.4 / 8.55	42.4 / 9.38	42.4 / 9.4	42.4 / 10.86	42.4 / 11.68	42.4 / 12.61	42.4 / 13.5	42.4 / 15.03
20	-	-	45.1 / 7.54	45.1 / 8.94	45.1 / 9.4	45.1 / 10.59	45.1 / 11.37	45.1 / 12.23	45.1 / 13.5	45.1 / 14.36
25	-	-	-	48 / 7.9	48 / 9.4	48 / 10.28	48 / 11.09	48 / 11.92	48 / 13.6	48 / 13.85
30	-	-	-	-	51 / 9.4	51 / 9.79	51 / 10.75	51 / 11.6	51 / 13.7	51 / 13.42
35	-	-	-	-	-	54 / 8.6	54 / 10.2	54 / 11.21	54 / 13.75	54 / 13

Fig. 14: Expanded heating capacity table of the heat pump

Generating the compressor efficiency map

The tutorial available in the 'script_etas_gen.ipynb' is followed to generate the compressor efficiency map. The model is parametrized for each of the design point in the expanded heating capacity table from the previous step, as done for the initial parametrization of the model for the nominal operating point. As the power consumption of the compressor is dependent on the isentropic efficiency, which is set as a parameter in the compressor, it is changed for each point in order to match the power consumption calculated by the model and that in the table. The isentropic efficiency values are restricted to the range of 0.3 - 0.95.

Note: In the instances when the power values cannot be matched even at the extreme values for the compressor isentropic efficiency, the extreme values are assumed despite the difference in power predicted by the model and that in the table.

The compressor isentropic efficiency map generated as described is summarized in table below.

'fast' mode data

'detailed' calculation mode of the model for the simulations

After establishing the series of design points, the model is then developed to estimate the performance of the heat pump at any operating point within its range. The heat pump is used as the primary heating source for the hot water tank. The temperature of the water flow to the heat pump from the hot water tank, and the source air temperature are the required inputs.

The model checks if the conditions are within the operation range of the heat pump and ensures that the source air temperature is lower than the incoming water temperature. The source air temperature closest to the input value is then identified from the *expanded heating capacity table*. The model checks that the inlet water temperature is lower than the maximum possible condenser outlet temperature at the identified design source air temperature. Assuming a temperature difference of 5°C in the condenser in the design case, an initial condenser outlet temperature is estimated. The closest design point to the estimated condenser outlet temperature is then identified from the *extended heating capacity table*. In case this estimated condenser temperature is greater than the maximum possible outlet temperature, the maximum value is used as the design point. The heating capacity and the isentropic efficiency of the compressor corresponding to the

Compressor Isentropic Efficiency										
Source air temperature (deg. C)	Condenser outlet temperature (deg. C)									
	15	20	25	30	35	40	45	50	55	60
-20	0.55	0.56	0.56	0.55	0.42	0.45	0.35	-	-	-
-15	0.54	0.56	0.58	0.58	0.52	0.54	0.48	-	-	-
-12	0.54	0.56	0.58	0.59	0.56	0.57	0.54	-	-	-
-10	0.53	0.56	0.58	0.60	0.58	0.59	0.57	0.52	-	-
-7	0.51	0.54	0.58	0.60	0.61	0.62	0.61	0.58	0.61	-
-2	0.47	0.52	0.56	0.59	0.63	0.64	0.65	0.65	0.67	0.59
2	0.44	0.49	0.53	0.58	0.63	0.65	0.67	0.69	0.69	0.69
7	0.41	0.45	0.50	0.55	0.63	0.64	0.68	0.71	0.77	0.78
10	0.40	0.43	0.48	0.53	0.65	0.63	0.67	0.72	0.77	0.81
12	0.43	0.41	0.46	0.51	0.62	0.62	0.67	0.72	0.80	0.83
15	-	0.41	0.44	0.49	0.59	0.60	0.65	0.71	0.79	0.85
20	-	-	0.42	0.45	0.53	0.56	0.62	0.69	0.74	0.86
25	-	-	-	0.42	0.45	0.52	0.58	0.66	0.68	0.84
30	-	-	-	-	0.37	0.47	0.54	0.61	0.62	0.81
35	-	-	-	-	-	0.45	0.49	0.56	0.56	0.77

Fig. 15: Compressor isentropic efficiency map

design point source air and condenser outlet water temperatures are identified from the respective tables, the *expanded heating capacity table* and *compressor efficiency table*.

The model first performs a network calculation in the *design* mode at the identified design point operating conditions. An *offdesign* mode calculation of the network is then performed for the actual input operating conditions, based on the design mode calculation and the default characteristics of *TESPy*, to obtain the condenser outlet water temperature (supply temperature), the mass flow of water in the condenser and the power consumption of the heat pump.

Inputs		Design Point Data			
Source air temperature	Condenser water inlet temperature	Source air temperature	Condenser water outlet temperature	Heat load	Compressor isentropic efficiency
3	32	2	35	29.67	0.65
18	48	20	55	45.1	0.91

Fig. 16: Example of detailed calculation mode

'fast' calculation mode of the model

In addition to the detailed mode of calculation explained above, a fast calculation mode has also been implemented in the model to improve its computational speed. The model is discreetly parametrized over the entire operation range of the heat pump, at a resolution of 1°C for both the inputs, the source air temperature, and the condenser water inlet temperature. The detailed calculation mode can be implemented over this range of inputs and the output data from the model- the coefficient of performance (COP) of the heat pump and the condenser mass flow rate- can be saved. During the actual simulation, the saved inputs that are lower/closest to the actual input data are identified, and the saved output data for these points are used to calculate the outputs of the model, rather than performing the actual *design* and *offdesign* calculations. Though the granularity of the model is reduced, there is a significant improvement in the simulation duration.

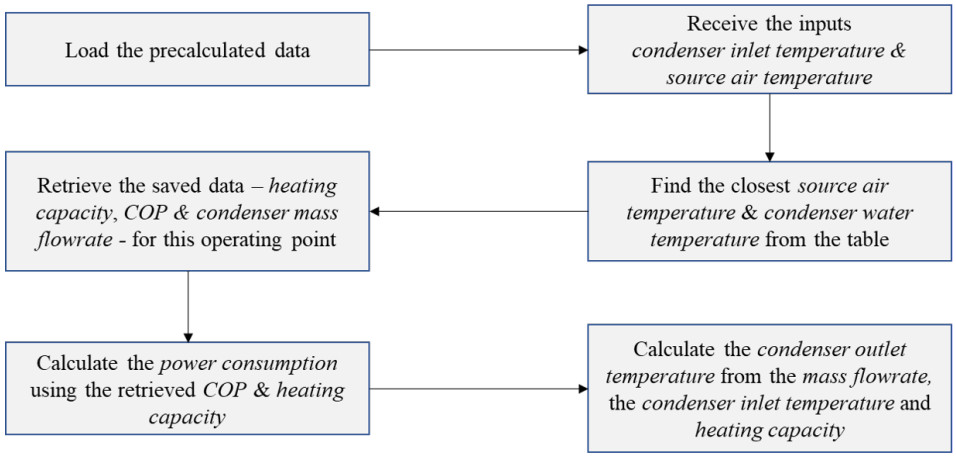


Fig. 17: Flowchart explaining the fast calculation mode of the TESPy heat pump model

Generation of *'fast'* mode data

The data for the *'fast'* calculation mode can be calculated and saved as follows:

1. **Adding the new heat pump model**

   The initial parametrization of the new heat pump model, based on the nominal operating point, from the 'Parametrization_NominalData.py' file, should be added to the *'_design_hp'* method of the heat_pump_design.py file (line 217).

2. **'fast' calculation mode data generation**

   The tutorial in the 'Fast_Calculation_Mode.ipynb', should be followed to generate the fast mode data.

3. **'fast' calculation mode data processing**

   The tutorial in the 'Fast_Mode_DataProcessing.ipynb' script should be used to process the fast mode data, to fill the missing values in the data that result from the errors in the model etc.

Module Documentation

class mosaik_components.heatpump.Heat_Pump_Design.**Heat_Pump_Design**(*params*, *COP_m_data=None*)

Design of the heat pump model for the different calculation modes

_take_closest(*myList*, *myNumber*)

Assumes myList is sorted. Returns closest value to myNumber. If two numbers are equally close, return the smallest number.

_etas_heatload_id()

- Used for all the calculation modes except 'fixed' mode.
- Uses the pre-saved data from the "eta_s_data.json" file
- Checks the inputs, **cond_in_T** and **heat_source_T**, against the limits of operation for the chosen heat pump model.
- Identifies the closest design point for the inputs, for the 'detailed' calculation mode

_design_hp()

- Used in the 'detailed' calculation mode to solve the TESPy network of the heat pump in the design mode
- Number of stages of compression can be 1 (default) or 2
- Intercooler between the two stages of compression is optional
- Superheater between the evaporator and the compressor is optional
- fixed and variable mass flow in the evaporator

p_cop_calc()

Calculates the power consumption and the COP of the heat pump in the 'detailed' calculation mode

step(*inputs*)

Performs simulation step with the step size 'step_size'

step_error()

Sets the outputs of the heat pump model to 0

While the *hplib* heat pump model available in the package can simulate the performance of the different heat pumps from the keymark data, the *tespy* model provides fewer options. In order to simulate different heat pumps, apart from the ones already available in this package, the heat pumps have to be intergrated to the *tespy* model. An example of the integration of the *"Air_30kW"* heat pump, based on "ait-deutschland LW-300(L)", shows the procedure in detail. The development of the model to simulate the performance of this specific heat pump is described in the steps below:

Step 1: *Initial Parametrization*

Step 2: *Compressor efficiency map*

Step 3: *'fast' mode data*

mosaik-heatpump (**|mosaik_hp_version|**) provides models for the simulation of heating systems- consisting of heat pumps, hot water tanks, and controllers - and adapters for the co-simulation of these models using *mosaik*.

Installation & Tests

You can install mosaik-heatpump with pip:

```
pip install mosaik-heatpump
```

You can run the tests with:

```
pytest
```

Getting started

A description of the different models available in the package and examples to use individual models can be found *here*.

Example scenarios for the co-simulation of all the models can be found *here*.

Getting help

Please report bugs and ideas for improvement to our issue tracker.

For questions and general discussion about mosaik-heatpump, you can use mosaik's GitHub Discussions.

Citation

The model was initially used and presented here , but it is not described in detail. We plan to publish a paper presenting the package, and will update this section after the publication.

License

The package is completely open source and is covered under the MIT License. **mosaik-examples**

- The mosaik-demo contains a simple demo scenario for mosaik.
- The DES demo is a simple example scenario showing the new mosaik 3.0 DES features
- COmmunication SIMulation for Agents (cosima) is an example scenario with integrated communication simulation based on OMNeT++.
- The aiomas demo is an example project, demonstrating how to couple a multi-agent system written in aiomas to mosaik.
- The mango demo is an example project, demonstrating how to couple a multi-agent system written in mango to mosaik.
- The binder tutorials contains python notebooks with example scenraios that can be executed on mybinder.

mosaik-tools

- icons for the energy domain
- maverig mosaik GUI is a visualization component, which is not maintained anymore.

basic simulators

In order to test custom-made simulators, two basic simulators are provided to use and connect to.

- The *InputSimulator* is a simulator that can be used to feed either a constant value or the value of a function into a designated simulator ready to handle the data.
- The *OutputSimulator* writes data from a custom simulator into a python dictionary. Users can access this dictionary by calling *get_dict* on a created output simulator entity.

Below is an example code snippet that connects the input simulator with the output simulator and executes ten time steps. After the simulation is done, the dictionary including the values received by the input simulator is printed.

```
# The output simulator is initialized.
output_dict = world.start("OutputSim")

# Two entities of the output simulator model are created.
output_model = output_dict.Dict.create(2)

# The input simulator is initialized.
input = world.start("InputSim", step_size=1)

# One function input simulator entity is created.
input_model_func = input.Function.create(1, function=sample_function)
```

(continues on next page)

(continued from previous page)

```
# One constant input simulator entity is created.
input_model_const = input.Constant.create(1, constant=2)

# The input entities are connected to separate output entities
world.connect(input_model_func[0], output_model[1], "value")
world.connect(input_model_const[0], output_model[0], "value")

# Run simulation.
world.run(until=END)

# Dictionary content is printed.
pprint(output_dict.get_dict(output_model[0].eid))
pprint(output_dict.get_dict(output_model[1].eid))
```

external components

These components are developed by external users of mosaik and we can not guarantee or support the flawless integration of these tools with mosaik. If you also have implemented additional tools for mosaik, simulation models or adapters, feel free to contact us at mosaik [A T] offis.de to be listed here.

- pysimmods contains some simulation models, which can be used in mosaik scenarios.
- MIDAS contains a semi-automatic scenario configuration tool.
- mosaik-docker is a package for the deployment of mosaik with Docker.
- ZDIN-ZLE components contains the research and development of digitalized energy systems in ZLE using mosaik (collection of simulation models).
- nestli (Neighborhood Energy System Testing towards Large-scale Integration) is a co-simulation environment for benchmarking the performance of BACS (building automation and control systems). Is uses EnergyPlus and FMUs with mosaik.
- toolbox_doe_sa is a toolbox with Design of Experiment (DoE) and Sensitivity Analysis (SA) methods developed in the ERIGrid 2.0 project.
- mosaik-demod is a domestic energy demand modeling simulator.
- palestrai-mosaik is an adapter to integrate palaestrAI (an universal framework for multi-agent artificial intelligence) into mosaik.
- QEMS - Quarter Energy Management System contains simulation components, which are used to simulate an energy management system for neighborhoods for analyzing and optimizing energy flows.

external scenarios

These scenarios are developed by external users of mosaik and we can not guarantee or support the flawless practicability.

- Benchmark Model Multi-Energy Networks contains the implementation of a multi-energy networks (heat and electricity grid) benchmark model developed in the ERIGrid 2.0 project.
- Benchmark Model Multi-Energy Networks STL is based on the multi-energy networks benchmark and contains a same time loop for improved initialization of the simulators.
- ZDIN-ZLE scenarios contains the research and development of digitalized energy systems in ZLE using mosaik (collection of simulation scenarios).
- QEMS - Quarter Energy Management System Scenarios contains scenarios of an energy management system for neighborhoods for analyzing and optimizing energy flows.

CHAPTER

FIVE

TUTORIALS

In the **basic tutorial** you'll learn how you can integrate simulators and control strategy into the mosaik ecosystem as well as how you create simulation scenarios and execute them.

In the first part, we'll implement the Sim API for a simple example simulator. We'll also create a simulation scenario in which that simulator will send its data to mosaik-hdf5 which will store it in an HDF5 database.

In the second part, we'll also integrate a simple control mechanism into mosaik. We'll then create a scenario in which that control mechanism controls the example simulator from part one.

In the third part, we'll implement an additional master controller, which communicates with the other controllers. This communication takes place as **same-time loop** without progress in simulation time and illustrated this new mosaik 3.0 feature. It can be used for negotiation between multiple agents or controllers, like shown in the tutorial at hand, but also for initialization of simulations consisting of multiple phsycial systems.

In the next part, we'll implement a scenario with a new controller, which sets **external events**. These external events come from a simple button click-event of a graphical user interface. Therefore, with this new mosaik 3.0 feature it is possible to do Human-in-the-Loop simulations to support human interactions.

The **Odysseus tutorial** you'll learn how to connect the data-stream-management-tool Odysseus to mosaik. The second part shows some examples on how to use Odysseus. This tutorial may also be of some use when you want to connect any other component via ZeroMQ.

The **Java API tutorial** shows you how to use the Java API. This API is intended to connect simulators written in Java to mosaik. You can use the Java-API also as a RCP-Server if you want to run your Java-simulator on a separate machine.

Basic tutorial

5.1 Integrating a simulation model into the mosaik ecosystem

In this section we'll first implement a simple example simulator. We'll then implement mosaik's Sim-API step-by-step.

5.1.1 The model

We want to implement a very simple model with the following behavior:

- $val_0 = init_val$
- $val_i = val_{i-1} + delta$ for $i \in \mathbf{N}$, $i > 0$, $delta \in \mathbf{Z}$

That simply means our model has a value *val* to which we add some *delta* (which is a positive or negative integer) at every simulation step. Our model has the attribute *delta* (with value 1 by default) which can be changed by control mechanisms to alter the behavior of the model. And it has the (output) attribute *val* which is its current value.

Here is a possible implementation of that simulation model in Python:

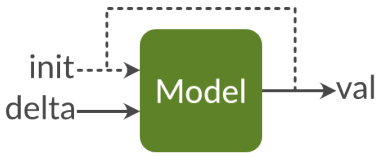


Fig. 1: Schematic diagram of our example model. You can change the *delta* and collect the *val* as output.

```
# example_model.py
"""
This module contains a simple example model.

"""


class Model:
    """Simple model that increases its value *val* with some *delta* every
    step.

    You can optionally set the initial value *init_val*. It defaults to ``0``.

    """
    def __init__(self, init_val=0):
        self.val = init_val
        self.delta = 1

    def step(self):
        """Perform a simulation step by adding *delta* to *val*."""
        self.val += self.delta
```

5.1.2 Setup for the API implementation

So lets start implementing mosaik's Sim-API for this model. We can use the Python *high-level API* for this. This package eases our workload, because it already implements everything necessary for communicating with mosaik. It provides an abstract base class which we can sub-class. So we only need to implement four methods and we are done.

If you already *installed* mosaik and the demo, you already have this package installed in your mosaik virtualenv.

We start by creating a new `simulator_mosaik.py` and import the module containing the mosaik API as well as our model:

```
# simulator_mosaik.py
"""
Mosaik interface for the example simulator.

"""
import mosaik_api_v3

import example_model
```

5.1.3 Simulator meta data

Next, we prepare the meta data dictionary that tells mosaik which *time paradigm* it follows (*time-based*, *event-based*, or *hybrid*), which models our simulator implements and which parameters and attributes it has. Since this data is usually constant, we define this at module level (which improves readability):

```
    'type': 'hybrid',
    'models': {
        'ExampleModel': {
            'public': True,
            'params': ['init_val'],
            'attrs': ['delta', 'val'],
            'trigger': ['delta'],
        },
    },
}
```

In this case we create a *hybrid* simulator, because we want to be able to control it using `delta` events later. For now, we won't use `delta`, though. We added our "ExampleModel" model with the parameter *init_val* and the attributes *delta* and *val*. At this point we don't care if they are inputs or outputs. We just list everything we can read or write. The *public* flag should usually be `True`. You can read more about it in the *Sim API docs*. From this information, mosaik deduces that our model could be used in the following way:

```
# Model name and "params" are used for constructing instances:
model = example_model.Model(init_val=42)
# "attrs" are normal attributes:
print(model.val)
print(model.delta)
```

5.1.4 The `Simulator` class

The package `mosaik_api_v3` defines a base class `Simulator` for which we now need to write a sub-class:

```
class ExampleSim(mosaik_api_v3.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'Model_'
        self.entities = {}  # Maps EIDs to model instances/entities
        self.time = 0
```

In our simulator's `__init__()` method (the constructor) we need to call `Simulator.__init__()` and pass the meta data dictionary to it. `Simulator.__init__()` will add some more information to the meta data and set it as `self.meta` to our instance.

We also set a prefix for our entity IDs and prepare a dictionary which will hold some information about the entities that we gonna create.

We can now start to implement the four API calls `init`, `create`, `step` and `get_data`:

5.1.5 init()

This method will be called exactly once while the simulator is being started via *World.start*. It is used for additional initialization tasks (e.g., it can handle parameters that you pass to a simulator in your scenario definition). It must return the meta data dictionary `self.meta`:

```
    def init(self, sid, time_resolution, eid_prefix=None):
        if float(time_resolution) != 1.:
            raise ValueError('ExampleSim only supports time_resolution=1., but'
                             ' %s was set.' % time_resolution)
        if eid_prefix is not None:
            self.eid_prefix = eid_prefix
        return self.meta
```

The first argument is the ID that mosaik gave to that simulator instance. The second argument is the *time resolution* of the scenario. In this example only the default value of *1.* (second per integer time step) is supported. If you set another value in the scenario, the simulator would throw an error and stop.

In addition to that, you can define further (optional) parameters which you can later set in your scenario. In this case, we can optionally overwrite the `eid_prefix` that we defined in `__init__()`.

5.1.6 create()

`create()` is called in order to initialize a number of simulation model instances *(entities)* within that simulator. It must return a list with some information about each entity created:

```
    def create(self, num, model, init_val):
        next_eid = len(self.entities)
        entities = []

        for i in range(next_eid, next_eid + num):
            model_instance = example_model.Model(init_val)
            eid = '%s%d' % (self.eid_prefix, i)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})

        return entities
```

The first two parameters tell mosaik how many instances of which model you want to create. As in `init()`, you can specify additional parameters for your model. They must also appear in the *params* list in the simulator meta data or mosaik will reject them. In this case, we allow setting the initial value *init_val* for the model instances.

For each entity, we create a new entity ID[1] and a model instance. We also create a mapping (`self.entities`) from the entity ID to our model. For each entity we create we also add a dictionary containing its ID and type to the `entities` list which is returned to mosaik. In this example, it has *num* entries for the model *model*, but it may get more complicated if you have, e.g., hierarchical models.

[1] Although entity IDs can be plain integers, it is advisable to use something more meaningful to ease debugging and analysis.

5.1.7 step()

The `step()` method tells your simulator to perform a simulation step. It receives its current simulation time, a dictionary with input values from other simulators (if there are any), and the time until the simulator can safely advance its internal time without creating a causality error. For time-based simulators (as in our example) it can be safely ignored (it is equal to the end of the simulation then). The method returns to mosaik the time at which it wants to do its next step. For event-based and hybrid simulators a next (self-)step is optional. If there is no next self-step, the return value is None/null.

Note: The *max_advance* value is not necessarily used and is only for special use cases where simulators can advance in time without expecting new inputs from other simulators, e.g. for the integration of a communication simulation.

```
    def step(self, time, inputs, max_advance):
        self.time = time
        # Check for new delta and do step for each model instance:
        for eid, model_instance in self.entities.items():
            if eid in inputs:
                attrs = inputs[eid]
                for attr, values in attrs.items():
                    new_delta = sum(values.values())
                model_instance.delta = new_delta

            model_instance.step()

        return time + 1  # Step size is 1 second
```

In this example, the *inputs* could be something like this:

```
{
    'Model_0': {
        'delta': {'src_id_0': 23},
    },
    'Model_1':
        'delta': {'src_id_1': 42},
        'val': {'src_id_1': 20},
    },
}
```

The inner dictionaries containing the actual values may contain multiple entries if multiple source entities provide input for another entity. In the case above, we have two source entities, 'Model_0' providing the delta value to the destination entity (object of ExampleSim) and 'Model_1' providing the delta and val value to the destination entity (object of ExampleSim). The source entitiy, 'Model_0' has the attribute 'delta' as the key to another nested dictionary which contains the simulator id and its corresponding 'delta' value. Similarly the source entity 'Model_1' has the attributes 'delta' and 'val' as the keys to two other nested dictionaries which contain the simulator id and its corresponding 'delta' and 'val' values.

The structure of the *inputs* dictionary created by mosaik is always the same as depicted above, only the number of source entities (dependent on the connections in the scenario ('Model_0' and 'Model_1' in our case)) and the number of attributes passed by the source entity varies. The first key of the nested dictionary will be the source entity ('Model_1'), the following keys will be the attributes passed by this source entity to the destination entity ('delta': {'src_id_1': 42}, 'val': {'src_id_1': 20}).

The simulator receiving these inputs is responsible for aggregating them (e.g., by taking their sum, minimum or maximum. Since we are not interested in the source's IDs, we convert that dict to a list with `values.values()` before we calculate the sum of all input values.

After we converted the inputs to something that our simulator can work with, we let it finally perform its next simulation step.

The return value `time + 1` tells mosaik that we wish to perform the next step in one second (in simulation time), as the *time_resolution* is 1. (second per integer step). Instead of using a fixed (hardcoded) step size you can easily implement any other stepping behavior.

5.1.8 get_data()

The `get_data()` call allows other simulators to get the values of the `delta` and `val` attributes of our models (the attributes we listed in the simulator meta data):

```
    def get_data(self, outputs):
        data = {}
        for eid, attrs in outputs.items():
            model = self.entities[eid]
            data['time'] = self.time
            data[eid] = {}
            for attr in attrs:
                if attr not in self.meta['models']['ExampleModel']['attrs']:
                    raise ValueError('Unknown output attribute: %s' % attr)

                # Get model.val or model.delta:
                data[eid][attr] = getattr(model, attr)

        return data
```

The *outputs* parameter contains the query and may in our case look like this:

```
{
    'Model_0': ['delta', 'value'],
    'Model_1': ['value'],
}
```

The Outputs dictionary may contain multiple keys if multiple destination entities ask for the output from the source entity. In this case we have two destination entities 'Model_0' and 'Model_1' which are requesting for the source attributes. 'Model_0' is requesting for the two source attributes 'delta' and 'value', whereas 'Model_1' is requesting for 1 source attribute 'value'. The structure of the Outputs dictionary created by mosaik is always the same as depicted above, only the number of destination entities (dependent on the connections in the scenario ('Model_0' and 'Model_1' in our case)) and the number of attributes requested by the destination entity varies.

In our implementation we loop over each entity ID for which data is requested. We then loop over all requested attributes and check if they are valid. If so, we dynamically get the requested value from our model instance via `getattr(obj, 'attr')`. We store all values in the `data` dictionary and return it when we are done.

The expected return value would then be:

```
{
    'Model_0': {'delta': 1, 'value': 24},
    'Model_1': {'value': 3},
}
```

5.1.9 Making it executable

The last step is adding a `main()` method to make our simulator executable (e.g., via `python -m simulator_mosaik HOST:PORT`). The package `mosaik_api_v3` contains the method `start_simulation()` which creates a socket, connects to mosaik and listens for requests from it. You just call it in your `main()` and pass an instance of your simulator class to it:

```
def main():
    return mosaik_api_v3.start_simulation(ExampleSim())


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

Simulators running on different nodes than the mosaik instance are supported explicitly with the mosaik Python-API v2.4 upward via the **remote** flag. A simulator with the `start_simulation()` method in its `main()` can then be called e.g. via

```
python simulator_mosaik -r HOST:PORT
```

in the command line. The mosaik scenario, started independently, can then connect to the simulator via the statement connect: `HOST:PORT` in its "*sim_config*" (Configuration). Note that it may make sense to introduce a short waiting time into your scenario to give you enough time to start both processes. Alternatively, the remote connection of simulators supports also a timeout (via the **timeout** flag, e.g. **–t 60** in the command line call will cause your simulator to wait for 60 seconds for an initial message from mosaik).

5.1.10 Summary

We have now implemented the mosaik Sim-API for our simulator. The following listing combines all the bits explained above:

```
# simulator_mosaik.py
"""
Mosaik interface for the example simulator.

"""
import mosaik_api_v3

import example_model


META = {
    'type': 'hybrid',
    'models': {
        'ExampleModel': {
            'public': True,
            'params': ['init_val'],
            'attrs': ['delta', 'val'],
            'trigger': ['delta'],
        },
    },
}


class ExampleSim(mosaik_api_v3.Simulator):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'Model_'
        self.entities = {}  # Maps EIDs to model instances/entities
        self.time = 0

    def init(self, sid, time_resolution, eid_prefix=None):
        if float(time_resolution) != 1.:
            raise ValueError('ExampleSim only supports time_resolution=1., but'
                             ' %s was set.' % time_resolution)
        if eid_prefix is not None:
            self.eid_prefix = eid_prefix
        return self.meta

    def create(self, num, model, init_val):
        next_eid = len(self.entities)
        entities = []

        for i in range(next_eid, next_eid + num):
            model_instance = example_model.Model(init_val)
            eid = '%s%d' % (self.eid_prefix, i)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})

        return entities


    def step(self, time, inputs, max_advance):
        self.time = time
        # Check for new delta and do step for each model instance:
        for eid, model_instance in self.entities.items():
            if eid in inputs:
                attrs = inputs[eid]
                for attr, values in attrs.items():
                    new_delta = sum(values.values())
                model_instance.delta = new_delta

            model_instance.step()

        return time + 1  # Step size is 1 second

    def get_data(self, outputs):
        data = {}
        for eid, attrs in outputs.items():
            model = self.entities[eid]
            data['time'] = self.time
            data[eid] = {}
            for attr in attrs:
                if attr not in self.meta['models']['ExampleModel']['attrs']:
                    raise ValueError('Unknown output attribute: %s' % attr)

                # Get model.val or model.delta:
                data[eid][attr] = getattr(model, attr)

        return data
```

(continues on next page)

(continued from previous page)

```
def main():
    return mosaik_api_v3.start_simulation(ExampleSim())


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

We can now start to write our first scenario, which we will do in the next section.

5.2 Creating and running simple simulation scenarios

We will now create a simple scenario with mosaik in which we use a simple data collector to print some output from our simulation. That means, we will instantiate a few ExampleModels and a data monitor. We will then connect the model instances to that monitor and simulate that for some time.

5.2.1 Configuration

You should define the most important configuration values for your simulation as "constants" on top of your scenario file. This makes it easier to see what's going on and change the parameter values.

Two of the most important parameters that you need in almost every simulation are the *simulator configuration* and the *duration* of your simulation:

```
# Sim config
SIM_CONFIG = {
    'ExampleSim': {
        'python': 'simulator_mosaik:ExampleSim',
    },
    'Collector': {
        'cmd': '%(python)s collector.py %(addr)s',
    },
}
END = 10  # 10 seconds
```

The *sim config* specifies which simulators are available and how to start them. In the example above, we list our *ExampleSim* as well as *Collector* (the names are arbitrarily chosen). For each simulator listed, we also specify how to start it. (If you are using type checking, you can import `SimConfig` from `mosaik.scenario` and change the first line to `SIM_CONFIG: SimConfig = {`, instead.)

Since our example simulator is, like mosaik, written in Python 3, mosaik can just import it and execute it in-process. The line `'python': 'simulator_mosaik:ExampleSim'` tells mosaik to import the package `simulator_mosaik` and instantiate the class `ExampleSim` from it.

The data collector will be started as external process which will communicate with mosaik via sockets. The line `'cmd': '%(python)s collector.py %(addr)s'` tells mosaik to start the simulator by executing the command `python collector.py`. Beforehand, mosaik replaces the placeholder `%(python)s` with the current python interpreter (the same as used to execute the scenario script) and `%(addr)s` with its actual socket address HOSTNAME:PORT so that the simulator knows where to connect to.

The section about the *Sim Manager* explains all this in detail.

Here is the complete file of the data collector. Take care of adding it to your example:

```
"""
A simple data collector that prints all data when the simulation finishes.

"""
import collections

import mosaik_api_v3


META = {
    'type': 'event-based',
    'models': {
        'Monitor': {
            'public': True,
            'any_inputs': True,
            'params': [],
            'attrs': [],
        },
    },
}


class Collector(mosaik_api_v3.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid = None
        self.data = collections.defaultdict(lambda:
                                            collections.defaultdict(dict))

    def init(self, sid, time_resolution):
        return self.meta

    def create(self, num, model):
        if num > 1 or self.eid is not None:
            raise RuntimeError('Can only create one instance of Monitor.')

        self.eid = 'Monitor'
        return [{'eid': self.eid, 'type': model}]

    def step(self, time, inputs, max_advance):
        data = inputs.get(self.eid, {})
        for attr, values in data.items():
            for src, value in values.items():
                self.data[src][attr][time] = value

        return None

    def finalize(self):
        print('Collected data:')
        for sim, sim_data in sorted(self.data.items()):
            print('- %s:' % sim)
            for attr, values in sorted(sim_data.items()):
                print('  - %s: %s' % (attr, values))


if __name__ == '__main__':
    mosaik_api_v3.start_simulation(Collector())
```

As its name suggests it collects all data it receives each step in a dictionary (including the current simulation time) and simply prints everything at the end of the simulation.

5.2.2 The World

The next thing we do is instantiating a *World* object. This object will hold all simulation state. It knows which simulators are available and started, which entities exist and how they are connected. It also provides most of the functionality that you need for modelling your scenario:

```
# Create World
world = mosaik.World(SIM_CONFIG)
```

To get access to mosaik's world, we need to import *mosaik* at the beginning of our scenario script. We also import *mosaik.util* to get access to some helper methods later on.

```
# demo_1.py
import mosaik
import mosaik.util
```

5.2.3 The scenario

Before we can instantiate any simulation models, we first need to start the respective simulators. This can be done by calling *World.start*. It takes the name of the simulator to start and, optionally, some simulator parameters which will be passed to the simulators `init()` method. So lets start the example simulator and the data collector:

```
# Start simulators
examplesim = world.start('ExampleSim', eid_prefix='Model_')
collector = world.start('Collector')
```

We also set the *eid_prefix* for our example simulator. What gets returned by *World.start* is called a *model factory*.

We can use this factory object to create model instances within the respective simulator. In your scenario, such an instance is represented as an *Entity*. The model factory presents the available models as if they were classes within the factory's namespace. So this is how we can create one instance of our example model and one 'Monitor' instance:

```
# Instantiate models
model = examplesim.ExampleModel(init_val=2)
monitor = collector.Monitor()
```

The *init_val* parameter that we passed to `ExampleModel` is the same as in the `create()` method of our Sim API implementation.

Now, we need to connect the example model to the monitor. That's how we tell mosaik to send the outputs of the example model to the monitor.

```
# Connect entities
world.connect(model, monitor, 'val', 'delta')
```

The method *World.connect* takes one entity pair – the source and the destination entity, as well as a list of attributes or attribute tuples. If you only provide single attribute names, mosaik assumes that the source and destination use the same attribute name. If they differ, you can instead pass a tuple like `('val_out', 'val_in')`.

Quite often, you will neither create single entities nor connect single entity pairs, but work with large(r) sets of entities. Mosaik allows you to easily create multiple entities with the same parameters at once. It also provides some utility functions for connecting sets of entities with each other. So lets create two more entities and connect them to our monitor:

```
# Create more entities
more_models = examplesim.ExampleModel.create(2, init_val=3)
mosaik.util.connect_many_to_one(world, more_models, monitor, 'val', 'delta')
```

Instead of instantiating the example model directly, we called its static method `create()` and passed the number of instances to it. It returns a list of entities (two in this case). We used the utility function *mosaik.util.connect_many_to_one* to connect all of them to the database. This function has a similar signature as *World.connect*, but the first two parameters are a *world* instance and a set (or list) of entities that are all connected to the *dest_entity*.

Mosaik also provides the function *mosaik.util.connect_randomly*. This method randomly connects one set of entities to another set. These two methods should cover most use cases. For more special ones, you can implement custom functions based on the primitive *World.connect*.

5.2.4 The simulation

In order to start the simulation, we call *World.run* and specify for how long we want our simulation to run:

```
# Run simulation
world.run(until=END)
```

Executing the scenario script will then give us the following output:

```
Collected data:
- ExampleSim-0.Model_0:
  - delta: {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1}
  - val: {0: 3, 1: 4, 2: 5, 3: 6, 4: 7, 5: 8, 6: 9, 7: 10, 8: 11, 9: 12}
- ExampleSim-0.Model_1:
  - delta: {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1}
  - val: {0: 4, 1: 5, 2: 6, 3: 7, 4: 8, 5: 9, 6: 10, 7: 11, 8: 12, 9: 13}
- ExampleSim-0.Model_2:
  - delta: {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1}
  - val: {0: 4, 1: 5, 2: 6, 3: 7, 4: 8, 5: 9, 6: 10, 7: 11, 8: 12, 9: 13}
```

Mosaik will also produce some diagnostic output along the lines of

```
2022-10-12 15:31:01.351 | INFO     | mosaik.scenario:start:131 - Starting "ExampleSim
↪" as "ExampleSim-0" ...
2022-10-12 15:31:01.352 | INFO     | mosaik.scenario:start:131 - Starting "Collector"␣
↪as "Collector-0" ...
INFO:mosaik_api_v3:Starting Collector ...
2022-10-12 15:31:01.430 | INFO     | mosaik.scenario:run:381 - Starting simulation.
100%|████████████████████████████████████████| 10/10 [00:00<00:00, 1996.05steps/s]
2022-10-12 15:31:01.446 | INFO     | mosaik.scenario:run:425 - Simulation finished␣
↪successfully.
```

If you don't want the progress bar, you can run the simulation with

```
world.run(until=END, print_progress=False)
```

instead. For even more progress bars, set `print_progress='individual'`, instead.

5.2.5 Summary

This section introduced you to the basic of scenario creation in mosaik. For more details you can check the *guide to scenarios*.

For your convenience, here is the complete scenario that we created in this tutorial. You can use this for some more experiments before continuing with this tutorial:

```
# demo_1.py
import mosaik
import mosaik.util
# End: Imports


# Sim config
SIM_CONFIG = {
    'ExampleSim': {
        'python': 'simulator_mosaik:ExampleSim',
    },
    'Collector': {
        'cmd': '%(python)s collector.py %(addr)s',
    },
}
END = 10  # 10 seconds
# End: Sim config

# Create World
world = mosaik.World(SIM_CONFIG)
# End: Create World

# Start simulators
examplesim = world.start('ExampleSim', eid_prefix='Model_')
collector = world.start('Collector')
# End: Start simulators

# Instantiate models
model = examplesim.ExampleModel(init_val=2)
monitor = collector.Monitor()
# End: Instantiate models

# Connect entities
world.connect(model, monitor, 'val', 'delta')
# End: Connect entities

# Create more entities
more_models = examplesim.ExampleModel.create(2, init_val=3)
mosaik.util.connect_many_to_one(world, more_models, monitor, 'val', 'delta')
# End: Create more entities

# Run simulation
world.run(until=END)
```

The next part of the tutorial will be about integrating control mechanisms into a simulation.

5.3 Adding a control mechanism to a scenario

Now that we integrated our first simulator into mosaik and tested it in a simple scenario, we should implement a control mechanism and mess around with our example simulator a little bit.

As you remember, our example models had a *value* to which they added something in each step. Eventually, their value will end up being very high. We'll use a multi-agent system to keep the values of our models in [-3, 3]. The agents will monitor the current *value* of their respective models and when it reaches -3/3, they will set *delta* to 1/-1 for their model.

Implementing the Sim API for control strategies is very similar to implementing it for normal simulators. We start again by importing the *mosaik_api_v3* package and defining the simulator meta data:

```
# controller.py
"""
A simple demo controller.

"""
import mosaik_api_v3


META = {
    'type': 'event-based',
    'models': {
        'Agent': {
            'public': True,
            'params': [],
            'attrs': ['val_in', 'delta'],
        },
    },
}
```

We set the `type` of the simulator to 'event-based'. As we have learned, this has two main implications:

1. Whenever another simulator provides new input for the simulator, a step is triggered (at the output time). So we don't need to take care of the synchronisation of the models and agents. As our example simulator is of type time-based, it is only stepped at its self-defined times and will thus not be triggered by (potential) outputs of the agents. It will receive any output of the agents in its subsequent step.

2. The provision of output of event-based simulators is optional. So if there's nothing to report at a specific step, the attributes can (and should be) omitted in the get_data's return dictionary.

Our control mechanism will use agents to control other entities. The agent has no parameters and two attributes, the input 'val_in' and the output 'delta'.

Let's continue and implement *mosaik_api_v3.Simulator*:

```
class Controller(mosaik_api_v3.Simulator):
    def __init__(self):
        super().__init__(META)
        self.agents = []
        self.data = {}
        self.time = 0
```

Again, nothing special is going on here. We pass our meta data dictionary to our super class and set an empty list for our agents.

Because our agents don't have an internal concept of time, we don't need to take care of the time_resolution of the scenario. And as there aren't any simulator parameters either, we don't need to implement *init*. The default implementation will return the meta data, so there's nothing we need to do in this case.

Implementing `create` is also straight forward:

```
    def create(self, num, model):
        n_agents = len(self.agents)
        entities = []
        for i in range(n_agents, n_agents + num):
            eid = 'Agent_%d' % i
            self.agents.append(eid)
            entities.append({'eid': eid, 'type': model})

        return entities
```

Every agent gets an ID like "Agent_*<num>*". Because there might be multiple `create` calls, we need to keep track of how many agents we already created in order to generate correct entity IDs. We also create a list of *{'eid': 'Agent_<num>', 'type': 'Agent'}* dictionaries for mosaik.

You may have noticed that we, in contrast to our example simulator, did not actually instantiate any *real* simulation models this time. We just pretend to do it. This okay, since we'll implement the agent's "intelligence" directly in `step`:

```
    def step(self, time, inputs, max_advance):
        self.time = time
        data = {}
        for agent_eid, attrs in inputs.items():
            delta_dict = attrs.get('delta', {})
            if len(delta_dict) > 0:
                data[agent_eid] = {'delta': list(delta_dict.values())[0]}
                continue

            values_dict = attrs.get('val_in', {})
            if len(values_dict) != 1:
                raise RuntimeError('Only one ingoing connection allowed per '
                                   'agent, but "%s" has %i.'
                                   % (agent_eid, len(values_dict)))
            value = list(values_dict.values())[0]
```

The `inputs` arguments is a nested dictionary and will look like this:

```
{
    'Agent_0': {'val_in': {'ExampleSim-0.Model_0': -1}},
    'Agent_1': {'val_in': {'ExampleSim-0.Model_1': 1}},
    'Agent_2': {'val_in': {'ExampleSim-0.Model_2': 3}}
}
```

For each agent, there's a dictionary with all input attributes (in this case only 'val_in'), containing the source entities (their full_id) with the corresponding values as key-value pairs.

First we initialize an empty `data` dict that will contain the set-points that our control mechanism is creating for the models of the example simulator. We'll fill this dict in the following loop. We iterate over all agents and extract its input 'val_in'; so `values_dict` is a dict containing the current values of all models connected to that agent. In our example we only allow to connect one model per agent, and fetch its value.

We now do the actual check:

```
            if value >= 3:
                delta = -1
            elif value <= -3:
                delta = 1
            else:
                continue
```

If the value is ≤ -3 or ≥ 3, we have to set a new *delta* value. Else, we don't need to do anything and can continue with a new iteration of the loop.

If we have a new *delta*, we add it to the `data` dict:

```
            data[agent_eid] = {'delta': delta}
```

After finishing the loop, the `data` dict may look like this:

```
{
    'Agent_0': {'delta': 1},
    'Agent_2': {'delta': -1},
}
```

Agent_0 sets the new *delta* = 1, and *Agent_2* sets the new *delta* = -1. *Agent_1* did not set a new *delta*.

At the end of the step, we put the data dict to the class attribute self.data, to make it accessible in the get_data method

```
        self.data = data
```

We return *None* to mosaik, as we don't want to step ourself, but only when the controlled models provide new values.

```
        return None
```

After having called step, mosaik requests the new set-points via the get_data function. In principle we could just return the self.data dictionary, as we already constructed that in the adequate format. For illustrative purposes we do it manually anyhow. Additionally, if we do it like that, we can only send back the attributes that are actually needed by (connected to) other simulators:

```
    def get_data(self, outputs):
        data = {}
        for agent_eid, attrs in outputs.items():
            for attr in attrs:
                if attr != 'delta':
                    raise ValueError('Unknown output attribute "%s"' % attr)
                if agent_eid in self.data:
                    data['time'] = self.time
                    data.setdefault(agent_eid, {})[attr] = self.data[agent_eid][attr]

        return data
```

Here is the complete code for our (very simple) controller / mutli-agent system:

```
# controller.py
"""
A simple demo controller.

"""
import mosaik_api_v3


META = {
    'type': 'event-based',
    'models': {
        'Agent': {
            'public': True,
            'params': [],
            'attrs': ['val_in', 'delta'],
```

(continues on next page)

(continued from previous page)

```
        },
    },
}


class Controller(mosaik_api_v3.Simulator):
    def __init__(self):
        super().__init__(META)
        self.agents = []
        self.data = {}
        self.time = 0

    def create(self, num, model):
        n_agents = len(self.agents)
        entities = []
        for i in range(n_agents, n_agents + num):
            eid = 'Agent_%d' % i
            self.agents.append(eid)
            entities.append({'eid': eid, 'type': model})

        return entities

    def step(self, time, inputs, max_advance):
        self.time = time
        data = {}
        for agent_eid, attrs in inputs.items():
            delta_dict = attrs.get('delta', {})
            if len(delta_dict) > 0:
                data[agent_eid] = {'delta': list(delta_dict.values())[0]}
                continue

            values_dict = attrs.get('val_in', {})
            if len(values_dict) != 1:
                raise RuntimeError('Only one ingoing connection allowed per '
                                   'agent, but "%s" has %i.'
                                   % (agent_eid, len(values_dict)))
            value = list(values_dict.values())[0]

            if value >= 3:
                delta = -1
            elif value <= -3:
                delta = 1
            else:
                continue

            data[agent_eid] = {'delta': delta}

        self.data = data

        return None

    def get_data(self, outputs):
        data = {}
        for agent_eid, attrs in outputs.items():
            for attr in attrs:
                if attr != 'delta':
                    raise ValueError('Unknown output attribute "%s"' % attr)
```

(continues on next page)

(continued from previous page)

```
                if agent_eid in self.data:
                    data['time'] = self.time
                    data.setdefault(agent_eid, {})[attr] = self.data[agent_eid][attr]

        return data


def main():
    return mosaik_api_v3.start_simulation(Controller())


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

Next, we'll create a new scenario to test our controller.

5.4 Integrating a control mechanism

The scenario that we're going to create in this part of the tutorial will be similar to the one we created before but incorporate the control mechanism that we just created.

Again, we start by setting some configuration values and creating a simulation world:

```
# demo_2.py
import mosaik
import mosaik.util


# Sim config
SIM_CONFIG = {
    'ExampleSim': {
        'python': 'simulator_mosaik:ExampleSim',
    },
    'ExampleCtrl': {
        'python': 'controller:Controller',
    },
    'Collector': {
        'cmd': '%(python)s collector.py %(addr)s',
    },
}
END = 10  # 10 seconds

# Create World
world = mosaik.World(SIM_CONFIG)
```

We added *ExampleCtrl* to the sim config and let it be executed in-process with mosaik.

We can now start one instance of each simulator:

```
# Start simulators
with world.group():
    examplesim = world.start('ExampleSim', eid_prefix='Model_')
    examplectrl = world.start('ExampleCtrl')
collector = world.start('Collector')
```

We'll create three model instances, the same number of agents, and one database:

```
# Instantiate models
models = [examplesim.ExampleModel(init_val=i) for i in range(-2, 3, 2)]
agents = examplectrl.Agent.create(len(models))
monitor = collector.Monitor()
```

We use a list comprehension to create three model instances with individual initial values (-2, 0 and 2). For instantiating the same number of agent instances we use `create()` which does the same as a list comprehension but is a bit shorter.

Finally we establish pairwise bi-directional connections between the models and the agents:

```
# Connect entities
for model, agent in zip(models, agents):
    world.connect(model, agent, ('val', 'val_in'))
    world.connect(agent, model, 'delta', weak=True)
```

The important thing here is the `weak=True` argument that we pass to the second connection. This tells mosaik how to resolve the cyclic dependency, i.e. which simulator should be stepped first in case that both simulators have a scheduled step at the same time. (In our example this will not happen, as the agents are only stepped by the models' outputs.)

Finally, we can connect the models and the agents to the monitor and run the simulation:

```
# Connect to monitor
mosaik.util.connect_many_to_one(world, models, monitor, 'val', 'delta')
mosaik.util.connect_many_to_one(world, agents, monitor, 'delta')

# Run simulation
world.run(until=END)
```

In the printed output of the collector, you can see two important things: The first is that the agents only provide output when the delta of the controlled model is to be changed. And second, that the new delta is set at the models' subsequent step after it has been derived by the agents.

```
Collected data:
- ExampleCtrl-0.Agent_0:
  - delta: {2: -1, 5: 1, 8: -1}
- ExampleCtrl-0.Agent_1:
  - delta: {1: -1, 4: 1, 7: -1}
- ExampleCtrl-0.Agent_2:
  - delta: {0: -1, 3: 1, 6: -1, 9: 1}
- ExampleSim-0.Model_0:
  - delta: {0: 1, 1: 1, 2: -1, 3: -1, 4: -1, 5: 1, 6: 1, 7: 1, 8: -1, 9: -1}
  - val: {0: 0, 1: 2, 2: 2, 3: 0, 4: -2, 5: -2, 6: 0, 7: 2, 8: 2, 9: 0}
- ExampleSim-0.Model_1:
  - delta: {0: 1, 1: -1, 2: -1, 3: -1, 4: 1, 5: 1, 6: 1, 7: -1, 8: -1, 9: -1}
  - val: {0: 2, 1: 2, 2: 0, 3: -2, 4: -2, 5: 0, 6: 2, 7: 2, 8: 0, 9: -2}
- ExampleSim-0.Model_2:
  - delta: {0: -1, 1: -1, 2: -1, 3: 1, 4: 1, 5: 1, 6: -1, 7: -1, 8: -1, 9: 1}
  - val: {0: 2, 1: 0, 2: -2, 3: -2, 4: 0, 5: 2, 6: 2, 7: 0, 8: -2, 9: -2}
```

This is the complete scenario:

```
# demo_2.py
import mosaik
import mosaik.util


# Sim config
```

(continues on next page)

(continued from previous page)

```
SIM_CONFIG = {
    'ExampleSim': {
        'python': 'simulator_mosaik:ExampleSim',
    },
    'ExampleCtrl': {
        'python': 'controller:Controller',
    },
    'Collector': {
        'cmd': '%(python)s collector.py %(addr)s',
    },
}
END = 10  # 10 seconds

# Create World
world = mosaik.World(SIM_CONFIG)
# End: Create World

# Start simulators
with world.group():
    examplesim = world.start('ExampleSim', eid_prefix='Model_')
    examplectrl = world.start('ExampleCtrl')
collector = world.start('Collector')
# End: Start simulators

# Instantiate models
models = [examplesim.ExampleModel(init_val=i) for i in range(-2, 3, 2)]
agents = examplectrl.Agent.create(len(models))
monitor = collector.Monitor()
# End: Instantiate models

# Connect entities
for model, agent in zip(models, agents):
    world.connect(model, agent, ('val', 'val_in'))
    world.connect(agent, model, 'delta', weak=True)
# End: Connect entities

# Connect to monitor
mosaik.util.connect_many_to_one(world, models, monitor, 'val', 'delta')
mosaik.util.connect_many_to_one(world, agents, monitor, 'delta')

# Run simulation
world.run(until=END)
```

Congratulations, you have mastered the mosaik tutorial. The following sections provide a more detailed description of everything you learned so far.

5.5 Same-time loops

Important use cases for *same-time loops* can be the initialization of simulation and communication between controllers or agents. As the scenario definition has to provide initialization values for cyclic data-flows and every cyclic data-flow will lead to an incrementing simulation time, it may take some simulation steps until all simulation components are in a stable state, especially, for simulations consisting of multiple physical systems. The communication between controllers or agents usually takes place at a different time scale than the simulation of the technical systems. Thus, same-time loops can be helpful to model this behavior in a realistic way.

To give an example of same-time loops in mosaik, the previously shown *scenario* is extended with a master controller, which takes control over the other controllers. The communication between these two layers of controllers will take place in the same step without incrementing the simulation time. The code of the previous scenario is used as a base and extended as shown in the following.

5.5.1 Master controller

The master controller bases on the code of the *controller* of the previous scenario. The first small change for the master controller is in the meta data dictionary, where new attribute names are defined. The 'delta_in' represent the delta values of the controllers, which will be limited by the master controller. The results of this control function will be returned to the controllers as 'delta_out'.

```
META = {
    'type': 'event-based',
    'models': {
        'Agent': {
            'public': True,
            'params': [],
            'attrs': ['delta_in', 'delta_out'],
        },
    },
}
```

`__init__` is extended with `self.cache` for storing the inputs and `self.time` for storing the current simulation time, which is initialized with 0.

```
class Controller(mosaik_api_v3.Simulator):
    def __init__(self):
        super().__init__(META)
        self.agents = []
        self.data = {}
        self.cache = {}
        self.time = 0
```

The `step` is changed, so that first the current time is updated in the `self.time` variable. Also the control function is changed. The master controller gets the delta output of the other controllers as 'delta_in' and stores the last value of each controller in the `self.cache`. This is needed, because the controllers are event-based and the current values are only sent if the values changes. The control function of the master controller limits the sum of all deltas to be `< 1` and `> -1`. If these limits are exceeded the delta of all controllers will be overwritten by the master controller with `0` and sent to the other controller as 'delta_out'.

```
    def step(self, time, inputs, max_advance):
        self.time = time
        data = {}
        for agent_eid, attrs in inputs.items():
```

(continues on next page)

(continued from previous page)

```
            values_dict = attrs.get('delta_in', {})
            for key, value in values_dict.items():
                self.cache[key] = value

        if sum(self.cache.values()) < -1 or sum(self.cache.values()) > 1:
            data[agent_eid] = {'delta_out': 0}

        self.data = data

        return None
```

Additionally, two small changes in the `get_data` method were done. First, the name was updated to 'delta_out' in the check for the correct attribute name. Second, the current time, which was stored previously in the `step`, is added to the output cache dictionary. This informs mosaik that the simulation should start or stay in a same-time loop if also output data for 'delta_out' is provided.

```
    def get_data(self, outputs):
        data = {}
        for agent_eid, attrs in outputs.items():
            for attr in attrs:
                if attr != 'delta_out':
                    raise ValueError('Unknown output attribute "%s"' % attr)
                if agent_eid in self.data:
                    data['time'] = self.time
                    data.setdefault(agent_eid, {})[attr] = self.data[agent_eid][attr]

        return data
```

5.5.2 Controller

The *controller* has to be extended to handle the 'delta_out' from the master controller as input. If it receives an input value for the attribute 'delta', it will not calculate a new delta value, but use the one from the master controller.

```
    def step(self, time, inputs, max_advance):
        self.time = time
        data = {}
        for agent_eid, attrs in inputs.items():
            delta_dict = attrs.get('delta', {})
            if len(delta_dict) > 0:
                data[agent_eid] = {'delta': list(delta_dict.values())[0]}
                continue
```

The same-time loop in this scenario will always be finished after the second iteration, because the master controller will overwrite the deltas of the controller and will get back zeros as 'delta_in'. Thus, it will produce no output in the second iteration and the same-time loop will be finished.

5.5.3 Scenario

This scenario is based on the *previous scenario*. In the following description only the changes are explained, but the full code is shown. The updated controller and the new master controller are added to the sim config of the scenario.

```
# demo_3.py
import mosaik
import mosaik.util

# Sim config. and other parameters
SIM_CONFIG = {
    'ExampleSim': {
        'python': 'simulator_mosaik:ExampleSim',
    },
    'ExampleCtrl': {
        'python': 'controller_demo_3:Controller',
    },
    'ExampleMasterCtrl': {
        'python': 'controller_master:Controller',
    },
    'Collector': {
        'cmd': '%(python)s collector.py %(addr)s',
    },
}
END = 6  # 10 seconds

# Create World
world = mosaik.World(SIM_CONFIG)
```

The master controller is also started and initialized. The controllers get different 'init_val' values compared to the previous scenario. Here, it is changed to `(-2, 0, -2)` to have the right timing to get into the same-time loop.

```
# Start simulators
with world.group():
    examplesim = world.start('ExampleSim', eid_prefix='Model_')
    examplectrl = world.start('ExampleCtrl')
    examplemasterctrl = world.start('ExampleMasterCtrl')
collector = world.start('Collector')

# Instantiate models
models = [examplesim.ExampleModel(init_val=i) for i in (-2, 0, -2)]
agents = examplectrl.Agent.create(len(models))
master_agent = examplemasterctrl.Agent.create(1)
monitor = collector.Monitor()
```

The 'delta' outputs of the controllers are connected to the new master controller and the 'delta_out' of the master controller is connected to the respective controller. The `weak=True` argument defines, that the connection from the controllers to the master controller will be the first to be executed by mosaik.

```
# Connect entities
for model, agent in zip(models, agents):
    world.connect(model, agent, ('val', 'val_in'))
    world.connect(agent, model, 'delta', weak=True)

for agent in agents:
    world.connect(agent, master_agent[0], ('delta', 'delta_in'))
    world.connect(master_agent[0], agent, ('delta_out', 'delta'), weak=True)
```

(continues on next page)

(continued from previous page)

```
mosaik.util.connect_many_to_one(world, models, monitor, 'val', 'delta')
mosaik.util.connect_many_to_one(world, agents, monitor, 'delta')
world.connect(master_agent[0], monitor, 'delta_out')

# Run simulation
world.run(until=END)
```

The printed output of the collector shows the states of the different simulators. The collector just shows the final result of the same-time loop and not the steps during the loop. It can be seen that the 'delta' of 'Agent_1' changes to -1 at time step 2 and at time step 4 all 'delta' attributes are set to 0 by the master controller.

```
Collected data:
- ExampleCtrl-0.Agent_0:
  - delta: {3: 0}
- ExampleCtrl-0.Agent_1:
  - delta: {2: -1, 3: 0}
- ExampleCtrl-0.Agent_2:
  - delta: {3: 0}
- ExampleMasterCtrl-0.Master_Agent_0:
  - delta_out: {3: 0}
- ExampleSim-0.Model_0:
  - delta: {0: 1, 1: 1, 2: 1, 3: 0, 4: 0, 5: 0}
  - val: {0: -1, 1: 0, 2: 2, 3: 2, 4: 2, 5: 2}
- ExampleSim-0.Model_1:
  - delta: {0: 1, 1: 1, 2: -1, 3: 0, 4: 0, 5: 0}
  - val: {0: 1, 1: 2, 2: 2, 3: 0, 4: 0, 5: 0}
- ExampleSim-0.Model_2:
  - delta: {0: 1, 1: 1, 2: 1, 3: 0, 4: 0, 5: 0}
  - val: {0: -1, 1: 0, 2: 2, 3: 2, 4: 2, 5: 2}
```

A visualization of the execution graph shows the data flows in the simulation. For the first two time steps, only the controllers are executed, as they do not provide any output for 'delta'. Thus, the master controller was not stepped and the simulation was proceeded directly with the next simulation time step. At simulation time 2, the master controller is stepped, but as the sum of delta values is not exceeding the limits no control action takes place. At simulation time 4, the master controller is stepped again and this time sends back a value to the controllers to limit their 'delta' value. It can be seen, that the controllers are stepped a second time within the same simulation time and send data again to the master controller. After this second step of the master controller, it does not send an output again and the simulation proceeds to simulation time 5, where the same-time loop occures again.

5.6 Set external events

This tutorial gives an example on how to set external events for integrating unforeseen interactions of an external system in soft real-time simulation with `rt_factor=1.0`. A typical use case for this feature would be Human-in-the-Loop simulations to support human interactions, e.g., control actions. In mosaik, such external events can be implemented via the the asynchronous `set_event` method. These events will then be scheduled for the next simulation time step.

To give an example of external events in mosaik, a new scenario is created that includes a controller to set external events. In addition to the controller, a graphical user interface (GUI) is implemented and started in a subprocess for external control actions by the user.

The example code and additional requirements are shown in the following.

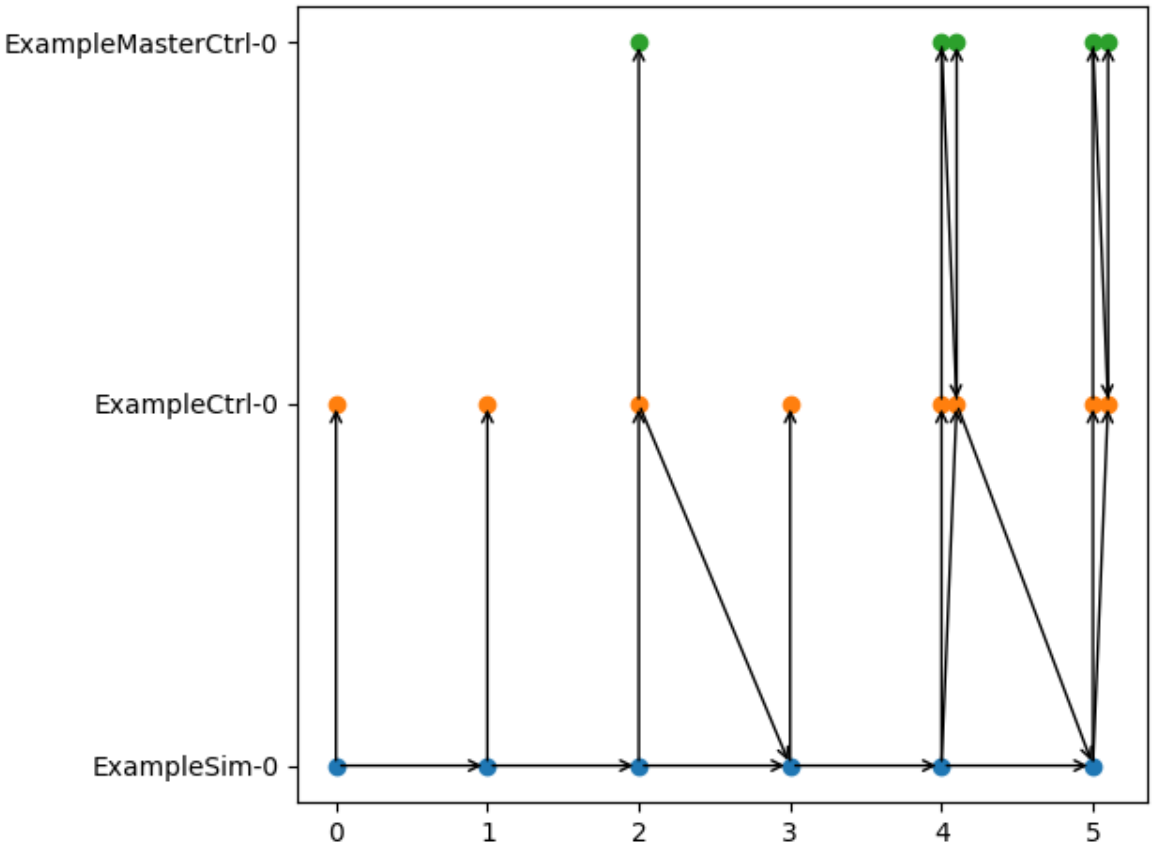


Fig. 2: Scheduling of demo 3.

5.6.1 Requirements

First of all, we need to install some additional requirements within the virtual environment (see *installation guide* for setting up a virtual environment)

- pyzmq (https://zeromq.org/languages/python/)
- PyQt5 (https://pypi.org/project/PyQt5/)

```
$ pip install pyzmq PyQt5
```

5.6.2 Set-event controller

Next, we need to create a new python module for the set-event controller, e.g., `controller_set_event.py`.

In the meta data dictionary of the set-event controller, we specify that this is an event-based simulator.

```
# controller_set_event.py

import sys
import zmq
import threading
import math

import mosaik_api_v3


META = {
    'type': 'event-based',
    'set_events': True,
    'models': {
        'Controller': {
            'public': True,
            'params': [],
            'attrs': [],
        },
    },
}
```

The set-event controller subscribes to external events from the GUI via a zeromq subscriber socket using the publish-subscribe pattern. Herefore, a listener thread is created which receives external event messages from the GUI. More information about the listener thread can be found in the next section.

```
class Controller(mosaik_api_v3.Simulator):
    def __init__(self):
        super().__init__(META)
        self.data = {}
        self.time = 0
        self.eid = None
        self.thread = None
        self.initial_timestamp = 0
        self.once = True
        self.context = zmq.Context()

        # Subscribe to external events from the GUI
        self.subscriber = self.context.socket(zmq.SUB)
        self.subscriber.connect("tcp://localhost:5563")
```

(continues on next page)

(continued from previous page)

```
        self.subscriber.setsockopt(zmq.SUBSCRIBE, b"B")

        # Listener THREAD
        self.thread = listen_to_external_events(self)

    def create(self, num, model):
        if num > 1 or self.eid is not None:
            raise RuntimeError('Can only create one instance of Controller.')

        self.eid = 'Controller_set_event'
        return [{'eid': self.eid, 'type': model}]

    def finalize(self):
        self.thread.join(0)
        sys.exit()
```

In order to set the event for the next time step, it is necessary to determine the current simulation time in wall clock time. For this, we need to store the initial timestamp in `step` once for the first simulation step.

```
    def step(self, time, inputs, max_advance):
        # Needed in listener thread to determine the current simulation time in wall␣
↪clock time.
        if self.once:
            self.initial_timestamp = self.mosaik.world.env.now
            self.once = False

        self.time = time
        print(f"In step at time {self.time}")
        print(f"max_advance {max_advance}")

        return None
```

5.6.3 Listener thread

The listener thread can be included in the same file as the set-event controller: `controller_set_event.py`.

The object of the controller class needs to be passed as a parameter to the `listen_to_external_events` function, which is called as a thread via the defined decorator `@threaded`. The listener thread listens to external event messages from the GUI. Once a message arrives, the listener thread calls the `set_event` method to set an external event for the next simulation step in mosaik.

```
def threaded(fn):
    def wrapper(*args, **kwargs):
        thread = threading.Thread(target=fn, args=args, kwargs=kwargs, daemon=True)
        thread.start()
        return thread
    return wrapper
```

```
@threaded
def listen_to_external_events(controller):
    while True:
        try:
            # Receive external event message from GUI
            [address, contents] = controller.subscriber.recv_multipart(zmq.NOBLOCK)
```

(continues on next page)

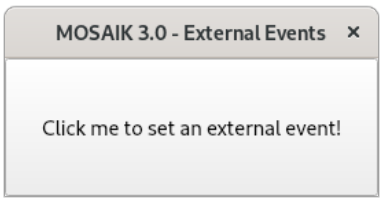
(continued from previous page)

```
            print(f"[{address}] {contents}")

            current_timestamp = controller.mosaik.world.env.now
            real_time = math.ceil(current_timestamp - controller.initial_timestamp)
            event_time = real_time + 1
            print(f"Current simulation time: {real_time}")

            if controller.time < event_time < controller.mosaik.world.until:
                print(f"Set external Event at time {event_time}")
                # Set external event in mosaik via asynchronous call
                controller.mosaik.set_event(event_time)

        except zmq.ZMQError as e:
            if e.errno == zmq.EAGAIN:
                # state changed since poll event
                pass
            else:
                raise
```

5.6.4 Graphical user interface

For the GUI, we create a new python module, e.g., `gui_button.py`.

The GUI is created with PyQt5 and provides a button to set external events in mosaik every time we click on it. To enable the set-event controller to perform this control action, a zeromq publisher socket is used to send a message to the controller's subscriber that the button has been clicked.




```
# gui_button.py

import sys
import zmq
from PyQt5 import QtWidgets
from PyQt5.QtWidgets import QApplication, QMainWindow


class PushButtonWindow(QMainWindow):
    def __init__(self):
        super(PushButtonWindow,self).__init__()
        self.button = None
        self.context = zmq.Context()

        # For external events
        self.publisher = self.context.socket(zmq.PUB)
        self.publisher.bind("tcp://*:5563")

    def button_clicked(self):
```

(continues on next page)

(continued from previous page)

```
        self.publisher.send_multipart([b"B", b"Push button was clicked!"])

    def create(self):
        self.setWindowTitle("MOSAIK 3.0 - External Events")

        self.button = QtWidgets.QPushButton(self)
        self.button.setText("Click me to set an external event!")
        self.button.clicked.connect(self.button_clicked)

        # Set the central widget of the Window.
        self.setCentralWidget(self.button)


def main():
    app = QApplication(sys.argv)
    window = PushButtonWindow()
    window.create()
    window.show()
    sys.exit(app.exec_())


if __name__ == "__main__":
    main()
```

5.6.5 Scenario

Next, we need to create a new python script for the external events scenario, e.g., `demo_4.py`.

For this scenario, the set-event controller is added to the `SIM_CONFIG` of the scenario.

```
# demo_4.py
import subprocess

import mosaik
import mosaik.util


SIM_CONFIG = {
    'Controller': {
        'python': 'controller_set_event:Controller',
    },
}

END = 60  # 60 seconds

# Create World
world = mosaik.World(SIM_CONFIG)
```

The set-event controller is started and initialized. Here, an initial event is added to the set-event controller so that the controller is executed at `time=0` to set the initial timestamp. This is needed for the determination of the current simulation time.

```
# Start simulators
controller = world.start('Controller')
```

(continues on next page)

(continued from previous page)

```
# Instantiate models
external_event_controller = controller.Controller()
world.set_initial_event(external_event_controller.sid)
```

The GUI is started in a subprocess and must be manually closed after the simulation is completed.

```
# Start GUI in a subprocess
proc = subprocess.Popen(['python', 'gui_button.py'])
```

In order to run the simulation scenario in soft real-time, the `rt_factor` is set to `1.0`.

```
# Run simulation in real-time
world.run(until=END, rt_factor=1.0)
```

Finally, we can run the scenario script as follows:

```
$ python demo_4.py
```

The printed output shows when the external events are triggered (button was clicked) and executed during simulation.

```
Starting "Controller" as "Controller-0" ...
WARNING: Controller-0 has no connections.
Starting simulation.
In step at time 0
max_advance 60
Simulation too slow for real-time factor 1.0 - 9.655498433858156e-05s behind time.
[b'B'] b'Push button was clicked!'
Current simulation time: 11
Set external Event at time 12
In step at time 12
max_advance 60
Simulation too slow for real-time factor 1.0 - 0.000688756990712136s behind time.
[b'B'] b'Push button was clicked!'
Current simulation time: 16
Set external Event at time 17
In step at time 17
max_advance 60
Simulation too slow for real-time factor 1.0 - 0.0013458110042847693s behind time.
[b'B'] b'Push button was clicked!'
Current simulation time: 26
Set external Event at time 27
In step at time 27
max_advance 60
Simulation too slow for real-time factor 1.0 - 0.0013047059765085578s behind time.
[b'B'] b'Push button was clicked!'
Current simulation time: 29
Set external Event at time 30
In step at time 30
max_advance 60
Simulation too slow for real-time factor 1.0 - 0.0019755829707719386s behind time.
[b'B'] b'Push button was clicked!'
Current simulation time: 33
Set external Event at time 34
In step at time 34
max_advance 60
Simulation too slow for real-time factor 1.0 - 0.0011994789820164442s behind time.
Simulation finished successfully.
```

Odysseus tutorial

5.7 Connecting mosaik and Odysseus

Odysseus is a framework for in-memory data stream management that is designed for online processing of big data. Large volumes of data such as continuously occurring events or sensor data can be processed in real time. In combination with mosaik Odysseus can be used to process, visualise and store the results of mosaik during a simulation.

In this first part of the tutorial we cover the two ways to connect mosaik and Odysseus, the *second part* is about how to use Odysseus to process, visualize and store simulation data.

Note: Connecting mosaik and Odysseus works mosaik >= 3.0

Note: Connecting mosaik and Odysseus works only with mosaik >= 3.0

You can choose between two different solutions to connect mosaik and Odysseus. Both have their advantages and disadvantages and therefore, the right choice depends on your use case. We recommend to use the SimAPI version for beginners.

No matter which connection we use, we first have to download Odysseus Server and Studio Client. For the first start of Odysseus Studio the default user "System" and password "manager" have to be used, the tenant can be left empty.

5.7.1 Connecting via mosaik protocol handler

The easiest way to connect to mosaik is to use the mosaik protocol handler in Odysseus, which is available as installable feature in Odysseus Studio. It uses the mosaik API through remote procedure calls (RPC) and offers a close coupling of mosaik and Odysseus. With this, a blocked simulation in mosaik or a blocked processing in Odysseus will block the other system as well. If this is a problem in your use case, you should look in the section *Connecting via ZeroMQ*.

First we have to install the mosaik feature from the incubcation site in odysseus, which can be found in the Odysseus Wrapper Plugins.

After installing the feature we create a new Odysseus project and in the project a new Odysseus script file (more information on Odysseus projects and script files can be found in this tutorial). To use mosaik as source we can use the mosaik operator which contains a standard configuration of mandatory parameters. The script-code in the Odysseus query language PQL looks like this:

```
#PARSER PQL
#METADATA TimeInterval
#QUERY
mosaikCon := MOSAIK({SOURCE = 'mosaik', type='simapi'})
```

This is for the standard configuration. If you want to change something, for example to use another port, you need a more detailed configuration:

```
#PARSER PQL
#METADATA TimeInterval
#QUERY
mosaikCon1 := ACCESS({TRANSPORT = 'TCPServer',
                      PROTOCOL = 'mosaik',
                      SOURCE = 'mosaik',
```

(continues on next page)

(continued from previous page)

```
                    DATAHANDLER = 'KeyValueObject',
                    WRAPPER = 'GenericPush',
                    OPTIONS = [
                      ['port', '5555'],
                      ['mosaikPort', '5554'],
                      ['byteorder', 'LITTLE_ENDIAN']
                    ]})
```

As we can see the protocol 'mosaik' is chosen. When the query is started, the mosaik protocol handler in Odysseus opens a TCP server for receiving data from mosaik.

Before we can receive data, we have to adapt our mosaik *scenario*. Here we take the mosaik-demo as an example. The Odysseus simulator is treated just like any other component in mosaik. It has to be added to the `SIM_CONFIG` parameter. For the connection to the simulator the `connect` command is used and the IP address and port of Odysseus have to be specified:

```
sim_config = {
    'Odysseus': {
        'connect': '127.0.0.1:5554',
    }
```

After that, we have to initialize the simulator and connect it to all components whose data we want to revceive in Odysseus. For the mosaik-demo, we have to add the following lines of code to the scenario definition:

```
    # Start simulators
    odysseusModel = world.start('Odysseus', step_size=60*15)

    # Instantiate models
    odysseus = odysseusModel.Odysseus.create(1)
    ody = odysseus[0]

    # Connect entities
    connect_many_to_one(world, nodes, ody, 'P', 'Vm')
    connect_many_to_one(world, houses, ody, 'P_out')
    connect_many_to_one(world, pvs, ody, 'P')
```

Now we have set up everything to receive mosaiks data in Odysseus. To begin transfering data we have to start first the query in Odysseus and then the simulation in mosaik.

For more information on how to use Odysseus visit *part two*.

5.7.2 Connecting via ZeroMQ

In contrast to the close coupling via mosaik protocol handler the coupling via ZeroMQ is more loose. Mosaik sends all data as data stream with ZeroMQ and Odysseus can even be closed and restarted during the simulation without affecting mosaik. This behaviour holds the risk of loosing data so it should only be used if this doesn't cause problems.

First we have to install the following features for Odysseus from incubation site:

- Odysseus Wrapper Plugins / Zero MQ
- Odysseus Wrapper Plugins / mosaik (only if you want to use the mosaik operator)

And from the update site:

- Odysseus Odysseus_core Plugins / Json Wrapper

After installing the features we create a new Odysseus project and in the project a new Odysseus script file. The messages sent by mosaik are formatted in JSON format and sent via ZeroMQ. So we have to choose the corresponding ZeroMQ transport handler and JSON protocol handler:

```
#PARSER PQL
#METADATA TimeInterval
#QUERY
mosaikCon3 := ACCESS({TRANSPORT = 'ZeroMQ',
                      PROTOCOL = 'JSON',
                      SOURCE = 'mosaik',
                      DATAHANDLER = 'KeyValueObject',
                      WRAPPER = 'GenericPush',
                      OPTIONS = [
                        ['host', '127.0.0.1'],
                        ['readport', '5558'],
                        ['writeport', '5559'],
                        ['byteorder', 'LITTLE_ENDIAN']
                      ]})
```

If you use the standard configurtion you can use the short version (feature "wrapper / mosaik" has to be installed):

```
#PARSER PQL
#METADATA TimeInterval
#QUERY
mosaikCon2 := MOSAIK({SOURCE = 'mosaik', type='zeromq'})
```

After setting up Odysseus we have to install the mosaik-zmq adapter in our mosaik virtualenv. It is available on GitLab and PyPI. To install it we have to activate our mosaik virtualenv and execute (if there are errors during installation have a look in the readme):

```
pip install mosaik-zmq
```

The mosaik-zmq adapter is treated in mosaik like any other component of the simulation. If we use the mosaik demo, we have to add the new simulator to the `SIM_CONFIG` parameter:

```
sim_config = {
    'ZMQ': {
        'cmd': 'mosaik-zmq %(addr)s',
    },
```

Also we have to initialize the ZeroMQ simulator and connect it to other components:

```
    # Start simulators
    zmqModel = world.start('ZMQ', step_size=15*60, duration=END)

    # Instantiate models
    zmq = zmqModel.Socket(host='tcp://*:', port=5558, socket_type='PUB')

    # Connect entities
    connect_many_to_one(world, nodes, zmq, 'P', 'Vm')
    connect_many_to_one(world, houses, zmq, 'P_out')
    connect_many_to_one(world, pvs, zmq, 'P')
```

For more information on how to use Odysseus visit *part two*.

5.8 Using Odysseus to process, visualize and store simulation data

This tutorial will give some examples on how you can use Odysseus to process, visualize and store the data from mosaik. More information about connecting mosaik and Odysseus can be found in the *first part* of the tutorial and more about Odysseus in general can be found in its documentation. If you have no experience with Odysseus you should first visit the tutorials in its documentation. Simple query processing and selection, projection and map should explain the basics.

5.8.1 Processing

Mosaik sends data in JSON format and so the key-value-object has to be used as datatype for receiving in Odysseus. But most operators in Odysseus are based on relational tuples with a fixed schema, so it can be useful to transform arriving key-value objects to relational tuples. For this the totuple operator can be used. It creates relational tuples with the given attributes and omitts all data, which is not included in the schema:

```
tuples = TOTUPLE({
        SCHEMA = [
                ['odysseus_0.Vm.PyPower-0.0-tr_sec', 'Double'],
                ['odysseus_0.Vm.PyPower-0.0-node_b1', 'Double'],
                ['odysseus_0.Vm.PyPower-0.0-node_b2', 'Double'],
                ['odysseus_0.Vm.PyPower-0.0-node_b3', 'Double'],
                ['odysseus_0.Vm.PyPower-0.0-node_b4', 'Double'],
                ['timestamp', 'STARTTIMESTAMP']
        ],
        TYPE = 'mosaik'},
        mosaikCon)
```

For better handling we can rename the attributes with the rename operator:

```
renamedTuples = RENAME({aliases =
                ['tr_sec_Vm', 'node1_Vm', 'node2_Vm', 'node3_Vm', 'node4_Vm',
↪'timestamp']
        }, tuples)
```

We can also add computations to the data with a map operator. The expressions parameter contains first the computation and second the new name for every attribute. In this example the deviation of voltage to the nominal voltage of 230 V is calculated (more information about the offered functions can be found here):

```
voltageDeviation = MAP({EXPRESSIONS = [
                ['abs(230 - tr_sec_Vm)', 'dev_tr_sec_Vm'],
                ['abs(230 - Node1_Vm)', 'dev_Node1_Vm'],
                ['abs(230 - Node2_Vm)', 'dev_Node2_Vm'],
                ['abs(230 - Node3_Vm)', 'dev_Node3_Vm'],
                ['abs(230 - Node4_Vm)', 'dev_Node4_Vm']
        ]}, renamedTuples)
```

By using the aggregate operator we are able to calculate e.g. the average values. We have to add an timewindow operator first to have the right timestamps for aggregating.

```
windowedTuples = TIMEWINDOW({SIZE = [5, 'MINUTES']}, voltageDeviation)
aggregatedTuples = AGGREGATE({
        AGGREGATIONS = [
                ['AVG', 'dev_tr_sec_Vm', 'AVG_dev_tr_sec_P'],
                ['AVG', 'dev_Node1_Vm', 'AVG_dev_Node1_P'],
                ['AVG', 'dev_Node2_Vm', 'AVG_dev_Node2_P'],
                ['AVG', 'dev_Node3_Vm', 'AVG_dev_Node3_P'],
```

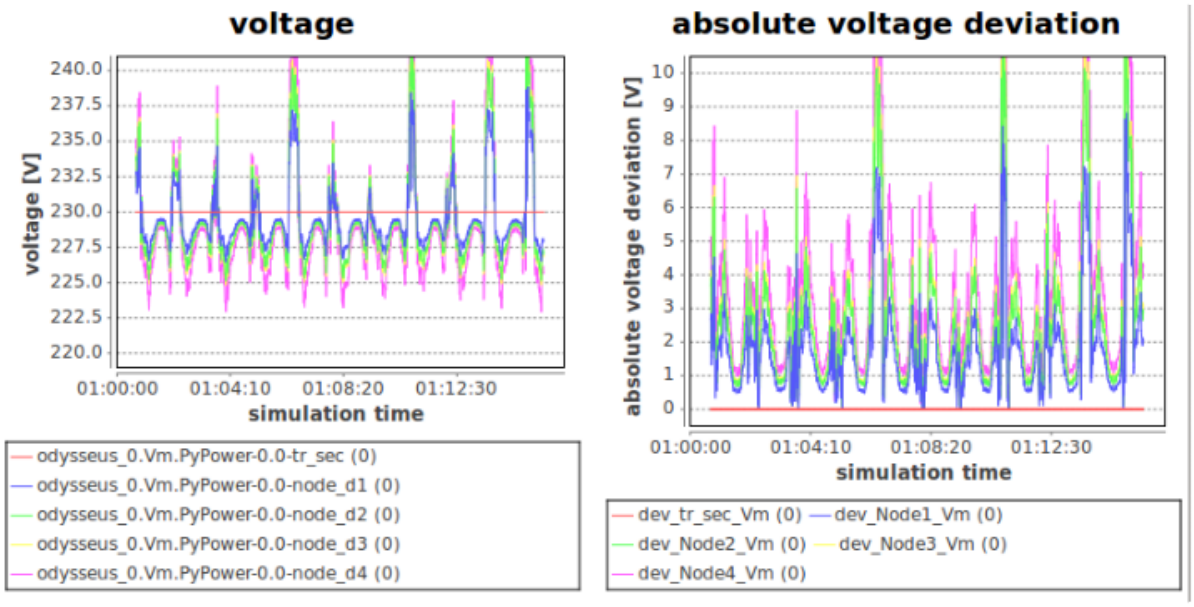
(continues on next page)

(continued from previous page)

```
                ['AVG', 'dev_Node4_Vm', 'AVG_dev_Node4_P']
                ]},
        windowedTuples)
```

5.8.2 Visualisation

To visualize data in Odysseus dashboards can be used, which can contain different graphs. For the data stream shown in the section above an exemplary dashboard could look like the following picture:



More information about dashboards in Odysseus can be found in the documentation.

5.8.3 Storing

If we want to save the results of our Odysseus query, we can use the sender operator to export it, e.g. to a csv file:

```
send = SENDER({
        SINK='writeCSV',
        transport='File',
        wrapper='GenericPush',
        protocol='CSV',
        dataHandler='Tuple',
        options=[
                ['filename','${WORKSPACEPROJECT}\output2.csv']
        ]}, aggregatedTuples)
```

Odysseus also offers adapters to store the processed data to different databases (e.g. mysql, postgres and oracle). More details can be found here.

Java API tutorial

5.9 Integrating a Model in Java

What do we do if we want to connect a simulator to mosaik which ist written in Java? In this tutorial we will describe how to create a simple model in Java and integrate it into mosaik using the mosaik-Java *high level API*. We will do this with the help of our simple model from the Python tutorial, i. e. we will try to replicate the first part of the *Python tutorial* as close as possible.

5.9.1 Getting the Java API

You can add the java package via maven or gradle, following the instructions at the mosaik-java-API package repository.

If you want to compile the jar yourself, you have to get the sources of the mosaik-java-API for Java which is provided on Gitlab. Clone it and put it in the development environment of your choice.

5.9.2 Creating the model

Next we create a Java class for our model. Our example model has exact the same behaviour as our simple *model* in the Python tutorial. To distinguish it from the Python-model we call it *JModel*. The only difference to the Python-model is that in Java we need two constructors (with and without init value) and getter and setter methods to access the variables *val* and *delta*.

```
class JModel {
    private float val;
    private float delta = 1;

    public JModel() {
        this.val = 0;
    }

    public JModel(float initVal) {
        this.val = initVal;
    }

    public float get_val() {
        return this.val;
    }

    public float get_delta() {
        return this.delta;
    }

    public void set_delta(float delta) {
        this.delta = delta;
    }

    public void step() {
        this.val += this.delta;
    }
}
```

5.9.3 Creating the simulator

A simulator provides the functionality that is necessary to manages instances of our model and to execute the models. We need a method *addModel* to create instances of our model and an ArrayList *models* to store them. The *step* method executes a simulation for each model instance. To access the *values* and *deltas* we need getter and setter methods. In our example the class implementing these functionalities is called JSimulator.

```
class JSimulator {
    private final ArrayList<JModel> models;

    public JSimulator() {
            this.models = new ArrayList<JModel>();
    }

    public void add_model(Number init_val) {
        JModel model;
        if (init_val == null) {
                model = new JModel();
        } else {
                model = new JModel(init_val.floatValue());
        }
        this.models.add(model);
    }

    public void step() {
            for (int i = 0; i < this.models.size(); i++) {
                    JModel model = this.models.get(i);
                    model.step();
            }
    }

    public float get_val(int idx) {
            JModel model = this.models.get(idx);
            return model.get_val();
    }

    public float get_delta(int idx) {
            JModel model = this.models.get(idx);
            return model.get_delta();
    }

    public void set_delta(int idx, float delta) {
            JModel model = this.models.get(idx);
            model.set_delta(delta);
    }
}
```

5.9.4 Implementing the mosaik API

Finally we need to implement the mosaik-API methods. In our example this is done in a class called JExampleSim. This class has to extent the abstract class *Simulator* from mosaik-java-api which is the Java-equivalent to the *Simulator class* in Python. The class *Simulator* provides the four mosaik-API-calls `init()`, `create()`, `step()`, and `getData()` which we have to implement. For a more detailed explanation of the API-calls see the *API-documentation*.

But first we have to put together the *meta-data* containing information about models, attributes, and parameters of our simulator. 'models' are all models our simulator provides. In our case this is only *JModel*. 'public': true tells mosaik that it is allowed to create models of this class. 'params' are parameter that are passed during initialisation, in our case this is *init_val*. 'attrs' is a list of values that can be exchanged.

```
    private static final JSONObject meta = (JSONObject) JSONValue.parse(("{"
            + "    'api_version': " + Simulator.API_VERSION + ","
            + "    'models': {"
            + "        'JModel': {"
            + "            'public': true,"
            + "            'params': ['init_val'],"
            + "            'attrs': ['val', 'delta']"
            + "        }"
            + "    }"
            + "}").replace("'", "\""));
```

First method is **init()** that returns the meta data. In addition it is possible to pass arguments for initialization. In our case there is *eid_prefix* which will be used to name instances of the models:

```
    public Map<String, Object> init(String sid, Map<String, Object> simParams) {
            if (simParams.containsKey("eid_prefix")) {
                    this.eid_prefix = simParams.get("eid_prefix").toString();
            }
        return JExampleSim.meta;
    }
```

create() creates new instances of the model *JModel* by calling the *add_model*-method of *JSimulator*. It also assigns ID (eid) to the models, so that it is able to keep track of them. It has to return a list with the name (eid) and type of the models. You can find more details about the return object in the *API-documentation*.

```
    @Override
    public List<Map<String, Object>> create(int num, String model,
                    Map<String, Object> modelParams) {
        JSONArray entities = new JSONArray();
        for (int i = 0; i < num; i++) {
            String eid = this.eid_prefix + (this.idCounter + i);
            if (modelParams.containsKey("init_val")) {
                Number init_val = (Number) modelParams.get("init_val");
                this.simulator.add_model(init_val);
            }
            JSONObject entity = new JSONObject();
            entity.put("eid", eid);
            entity.put("type", model);
            entity.put("rel", new JSONArray());
            entities.add(entity);
            this.entities.put(eid, this.idCounter + i);
        }
        this.idCounter += num;
        return entities;
    }
```

step() tells the simulator to perform a simulation step. It passes the *time*, the current simulation time, and *inputs*, a JSON data object with input data from preceding simulators. The structure of *inputs* is explained in the *API-documentation*. If there are new *delta*-values in *inputs* they are set in the appropriate model instance. Finally it calls the simulator's *step()*-method which, on its part, calls the *step()*-methods of the individual model-instances.

```
    public long step(long time, Map<String, Object> inputs) {
            //go through entities in inputs
        for (Map.Entry<String, Object> entity : inputs.entrySet()) {
            //get attrs from entity
            Map<String, Object> attrs = (Map<String, Object>) entity.getValue();
            //go through attrs of the entity
            for (Map.Entry<String, Object> attr : attrs.entrySet()) {
                    //check if there is a new delta
                String attrName = attr.getKey();
                if (attrName.equals("delta")) {
                    //sum up deltas from different sources
                    Object[] values = ((Map<String, Object>) attr.getValue()).
↪values().toArray();
                    float value = 0;
                    for (int i = 0; i < values.length; i++) {
                        value += ((Number) values[i]).floatValue();
                    }
                    //set delta
                    String eid = entity.getKey();
                    int idx = this.entities.get(eid);
                    this.simulator.set_delta(idx, value);
                }
            }
        }
        //call step-method
        this.simulator.step();

        return time + this.stepSize;
    }
```

getData() gets the simulator's output data from the last simulation step. It passes *outputs*, a JSON data object that describes which parameters are requested. *getData()* goes through *outputs*, retrieves the requested values from the appropriate instances of JModel and puts it in *data*. The structure of *outputs* and *data* is explained in the *API-documentation*.

```
    public Map<String, Object> getData(Map<String, List<String>> outputs) {
        Map<String, Object> data = new HashMap<String, Object>();
        //*outputs* lists the models and the output values that are requested
        //go through entities in outputs
        for (Map.Entry<String, List<String>> entity : outputs.entrySet()) {
            String eid = entity.getKey();
            List<String> attrs = entity.getValue();
            HashMap<String, Object> values = new HashMap<String, Object>();
            int idx = this.entities.get(eid);
            //go through attrs of the entity
            for (String attr : attrs) {
                if (attr.equals("val")) {
                        values.put(attr, this.simulator.get_val(idx));
                }
                else if (attr.equals("delta")) {
                        values.put(attr, this.simulator.get_delta(idx));
                }
            }
            data.put(eid, values);
```

(continues on next page)

(continued from previous page)

```
        }
        return data;
    }
```

5.9.5 Connecting the simulator to mosaik

We use the same scenario as in our Python example *demo1*. The only thing we have to change is the way we connect our simulator to mosaik. There are two ways to do this:

- **cmd:** mosaik calls the Java API by executing the command given in *cmd*. Mosaik starts the Java-API in a new process and connects it to mosaik. This works only if your simulator runs on the same machine as mosaik.
- **connect:** mosaik connects to the Java-API which runs as a TCP server. This works also if mosaik and the simulator are running on different machines.

For more details about how to connect simulators to mosaik see the section about the *Sim Manager* in the mosaik-documentation.

Connecting the simulator using *cmd*

We have to give mosaik the command how to start our Java simulator. This is done in *SIM_CONFIG*. The marked lines show the differences to our Python simulator.

```
# Sim config. and other parameters
SIM_CONFIG = {
    'JExampleSim': {
        'cmd': 'java -cp JExampleSim.jar de.offis.mosaik.api.JExampleSim %(addr)s',
    },
    'Collector': {
        'cmd': 'python collector.py %(addr)s',
    },
}
END = 10 * 60  # 10 minutes
```

The placeholder *%(addr)s* is later replaced with IP address and port by mosaik. If we now execute demo_1.py we get the same *output* as in our Python-example.

Note: The command how to start the Java simulator may differ depending on your operating system. If the command is complex, e. g. if it contains several libraries, it is usually better to put it in a script and than call the script in *cmd*.

Connecting the simulator using *connect*

In this case the Java API acts as TCP server and listens at the given address and port. Let's say the simulator runs on a computer with the IP-address 1.2.3.4. We can now choose a port that is not assigned by default. In our example we choose port 5678. Make sure that IP-address and port is accessible from the computer that hosts mosaik (firewalls etc.).

Note: Of course you can run mosaik and your simulator on the same machine by using `127.0.0.1:5678` (localhost). You may want to do this for testing and experimenting. Apart from that the connection with cmd (*see above*) is usually the better alternative because you don't have to start the Java part separately.

We have to tell mosaik how to connect to the simulator. This is done in *SIM_CONFIG* in our scenario (demo1):

```
# Sim config. and other parameters
SIM_CONFIG = {
    'JExampleSim': {
        'connect': '1.2.3.4:5678',
    },
    'Collector': {
        'cmd': 'python collector.py %(addr)s',
    },
}
END = 10 * 60  # 10 minutes
```

The marked lines show the differences to our Python simulator. Our simulator is now called *JExampleSim* and we need to give the simulator's address and port after the *connect* key word.

Now we start JExampleSim. To tell the mosaik-Java-API to run as TCP-server is done by starting it with "server" as second argument. The first command line argument is IP-address and port. The command line in our example looks like this:

```
java -cp JExampleSim.jar de.offis.mosaik.api.JExampleSim 1.2.3.4:5678 server
```

If we now execute demo_1.py we get the same *output* as in our Python-example.

Note: You can find the source code used in this tutorial in the mosaik-source-files in the folder `docs/tutorial/code`.

Java Generics API tutorial

5.10 Integrating a Model in Java with Generics / Annotations API

Additionally to the basic mosaik Java API, another API based on the basic one can be used, called mosaik-java-api-generics, that contains some quality of life changes, for example automatic meta-model generation based on your model, automatic model instantiation, input parsing via generics and automatic gathering of data for mosaik.

Note: When to use basic Java API and when to use this API? If you want to use the flexibility from Python, for example using all inputs. If you want to have a more type strict Java-like experience, use this API instead.

This tutorial is also based on the Python tutorial *Python tutorial* for the simple model and shows how to use the most features of this API.

5.10.1 Getting the Java API

You can add the java package via maven or gradle, following the instructions at the mosaik-java-generics-API package repository.

If you want to compile the jar yourself, you have to get the sources of the mosaik-java-generics-API for Java which is provided on Gitlab. Clone it and put it in the development environment of your choice.

5.10.2 Creating the model

The example model is again a replication from *model*. This model is annotated with @Model, which will tell the parser that this is a model for mosaik. Normally, the simple class name will be used as model name. This can be customized by setting the value of the annotation. The model annotation has several different sub-annotations that are useful to customize the behaviour of the of the model parser and automatic class instantiation. Some are also needed for some java versions to work, namely where constructor parameter names are not set.

Note: All fields MUST be a non-primitive type. This is needed for internal purposes for the input data parsing. Else it could be possible that you read input values (for example 0 as demand) that were never set.

Annotated models, that are added to a simulator, will be searched for public fields and getter, which will then be transformed into attributes. For the mosaik params, a denoted constructor parameters will be used. If more than one constructor are available, the constructor used must be annotated with @Model.Constructor. You can rename constructor parameter by using @Model.Param("other_name"). This is especially useful if parameter naming is turned off (default for basic java).

The following example shows how to use the model annotation for the example model:

```
@Model("ExampleModel")
public class AnnotatedExampleModel {
    private Double val;
    private Double delta = 1.d;

    public AnnotatedExampleModel() {
        this.val = 0.d;
    }

    @Model.Constructor
    public AnnotatedExampleModel(@Model.Param("init_val") Double initVal) {
        this.val = initVal;
    }

    public Double getVal() {
        return this.val;
    }

    public Double getDelta() {
        return this.delta;
    }

    public void setDelta(Double delta) {
        this.delta = delta;
    }

    public void step() {
        this.val += this.delta;
    }
}
```

Other usages annotations:

- @Model.Id: If you want to pass the model id into the model, you can annotate a constructor parameter with this annotation. The parameter will not be included into the meta-model.
- @Model.Suppress: If you want to use public getter or fields, which should only be used internally, you can annotate it with Suppress. These getter or fields will not be part of the meta-model created.

5.10.3 Creating the simulator

The simulator implements an interface to the mosaik API. This util package has two new simulators you can inherit from. One for simulators with only one model and one for multi model simulation. The reason to have two simulators is simpler use of generics for single models compared to multi-model simulations. The simulators also separate entity model and input model. Separating the input model from the entity model enables you to use only a subset of the entity model as input.

Note: Input models need a default constructor! This is due to the input parsing where an initially empty model's fields will be filled iteratively.

Single model simulator

The single model simulators fits perfectly for our small example. We need to inherit from the ModelSimulator class and set the generic types. The model generics are first the entity model (the created model instances) and second the input model.

```
public class ExampleModelSim extends ModelSimulator<AnnotatedExampleModel,␣
↪AnnotatedExampleModel> {
```

The constructor must call the parent constructor with the simulator name, simulation type and entity and input model. This will be then used to create the meta-model.

```
    public ExampleModelSim() throws Exception {
        super("ExampleSim", SimulationType.TimeBased, AnnotatedExampleModel.class,␣
↪AnnotatedExampleModel.class);
    }
```

The second method that must be implemented is initialize. It contains the same parameter as the original init method but without does not expect you to return the meta-model, as this will be done automatically.

```
    public void initialize(String sid, Float timeResolution, Map<String, Object>␣
↪simParams) {
        if (simParams.containsKey("step_size")) {
            this.stepSize = ((Number) simParams.get("step_size")).intValue();
        }
    }
```

The last mandatory method is the modelStep, an abbreviation of the step method. The parameter contain time, maxAdvance and parsed input from mosaik. The input will be parsed into the generic input class passed to the simulator and put into a map which contains the model id and a collection of InputMessages with sender and input class instance method. The modelStep method for example will first sum the delta of all controller and then call the step method of the different models.

```
    @Override
    public long modelStep(long time, Map<String, Collection<InputMessage
↪<AnnotatedExampleModel>>> inputs, long maxAdvance) {
        for (String id :
                inputs.keySet()) {
            Collection<InputMessage<AnnotatedExampleModel>> modelCollection = inputs.
↪get(id);
            double sum = modelCollection.stream().mapToDouble(modelInputMessage ->␣
↪modelInputMessage.getInputMessage().getDelta()).sum();
```

(continues on next page)

(continued from previous page)

```
            getEntities().get(id).setDelta(sum);
        }
        for (AnnotatedExampleModel instance : getEntities().values()) {
            instance.step();
        }

        return time + this.stepSize;
    }
```

There are other optional methods that can be overridden. - finishEntityCreation: Can be used to call auxiliary functions for models or modify the entity models and map itself. - prepareGetData: Method will be called before the data for get_data will be gathered. Can be used if the mosaik model is only a DTO - setupDone: Can be used to add functionality when all models are created. - cleanup: Will be called after the simulation is finished, call super or else the entities will not be cleaned up!

The whole code for the simulator can be found here:

```
public class ExampleModelSim extends ModelSimulator<AnnotatedExampleModel,␣
↪AnnotatedExampleModel> {

    private int stepSize = 60;

    public ExampleModelSim() throws Exception {
        super("ExampleSim", SimulationType.TimeBased, AnnotatedExampleModel.class,␣
↪AnnotatedExampleModel.class);
    }

    @Override
    public void initialize(String sid, Float timeResolution, Map<String, Object>␣
↪simParams) {
        if (simParams.containsKey("step_size")) {
            this.stepSize = ((Number) simParams.get("step_size")).intValue();
        }
    }

    @Override
    public void finishEntityCreation(Map<String, AnnotatedExampleModel> entities) {
        // Change the entity map or call auxiliary methods here
    }

    @Override
    public void setupDone() throws Exception {
        // Call auxiliary methods after entities are created here
    }

    @Override
    public long modelStep(long time, Map<String, Collection<InputMessage
↪<AnnotatedExampleModel>>> inputs, long maxAdvance) {
        for (String id :
                inputs.keySet()) {
            Collection<InputMessage<AnnotatedExampleModel>> modelCollection = inputs.
↪get(id);
            double sum = modelCollection.stream().mapToDouble(modelInputMessage ->␣
↪modelInputMessage.getInputMessage().getDelta()).sum();
            getEntities().get(id).setDelta(sum);
        }
        for (AnnotatedExampleModel instance : getEntities().values()) {
```

(continues on next page)

(continued from previous page)

```
            instance.step();
        }

        return time + this.stepSize;
    }
}
```

Multi model simulator

The second simulator is designed to be able to simulate multiple models. While this guide only covers an example with one model, the basic concept of this simulator should come clear. Instead of using generics directly in the class, the simulator contains auxiliary methods to register methods for: - Model registration - Model creation finalization - Input data processing

To show how to use a different input model, the following model is used in this example:

```
@Model
public class MessageModel {
    public Double delta;
}
```

We first need to inherit from MultiModelInputSimulator and implement the constructor. The constructor is the best place to register the models and processing methods. For our example, we need to register our model and register an input step. We can additionally register a model finishing method.

```
    public MultiModelExampleSim() {
        super("ExampleSim", SimulationType.TimeBased, AnnotatedExampleModel.class);

        registerStepMethod(MessageModel.class, this::parseInput);
        registerFinishCreationMethod(AnnotatedExampleModel.class,␣
↪this::finishModelCreation);
    }
```

Note: You can only register models and methods when no simulation is running. Since there is currently no way to remove registered methods, try to only register them in the constructor.

First call the parent constructor with the simulator name and simulation type. Additionally, already known entity models can be passed. They could also be registered via registerModel method.

registerStepMethod will take an input model class and a method to process the input data. Parameters of the method are: time, Map of Map of InputMessage<T>, where T is the passed input model class. In our method, we will sum all deltas for every model set it for our models. We then return maxAdvance. This will make the simulationStep the step method which will dictate when we will be called again. If you have a reason that a registered step method returns a time sooner than the one returned in simulationStep, the soonest time will be returned to mosaik.

```
    private long parseInput(long time, Map<String, Map<String, InputMessage
↪<MessageModel>>> inputResult, long maxAdvance) {
        inputResult.forEach((model, inputMessageMap) -> {
            getEntities(AnnotatedExampleModel.class)
                    .get(model)
                    .setDelta(inputMessageMap
                            .values()
                            .stream()
```

(continues on next page)

(continued from previous page)

```
                                .mapToDouble(value -> value.getInputMessage().delta)
                                .sum());
        });
        return maxAdvance;
    }
```

registerFinishCreationMethod will take an entity model class and a function which passes a map of entity model instances, just like the similar single model simulator method, but you need to return the entity map. Since we have no additionally modifications to make, this method is empty and only there fore display purposes.

```
    private Map<String, AnnotatedExampleModel> finishModelCreation(Map<String,␣
↪AnnotatedExampleModel> modelMap) {
        // Put you entity modifications or auxiliary calls here
        return modelMap;
    }
```

Two methods have to be implemented. First, the initialize method, just like before in the single model simulator example, where the step size of the simulation is set:

```
    @Override
    public void initialize(String sid, Float timeResolution, Map<String, Object>␣
↪simParams) {
        if (simParams.containsKey("step_size")) {
            this.stepSize = ((Number) simParams.get("step_size")).intValue();
        }
    }
```

Second, the simulationStep method needs to be implemented. Here, we gather all entities of the AnnotatedExampleModel class and call step for every instance.

```
    @Override
    public long simulationStep(long time, Map<String, Object> inputs, long␣
↪maxAdvance) {
        getEntities(AnnotatedExampleModel.class).values().
↪forEach(AnnotatedExampleModel::step);
        return time + stepSize;
    }
```

You can also register methods for entity models before get_data will be called. The procedure is the same as for the other two register methods.

The whole code for the simulator:

```
public class MultiModelExampleSim extends MultiModelInputSimulator {
    private int stepSize = 60;

    public MultiModelExampleSim() {
        super("ExampleSim", SimulationType.TimeBased, AnnotatedExampleModel.class);

        registerStepMethod(MessageModel.class, this::parseInput);
        registerFinishCreationMethod(AnnotatedExampleModel.class,␣
↪this::finishModelCreation);
    }

    private Map<String, AnnotatedExampleModel> finishModelCreation(Map<String,␣
↪AnnotatedExampleModel> modelMap) {
```

(continues on next page)

(continued from previous page)

```
        // Put you entity modifications or auxiliary calls here
        return modelMap;
    }

    @Override
    public void initialize(String sid, Float timeResolution, Map<String, Object>␣
↪simParams) {
        if (simParams.containsKey("step_size")) {
            this.stepSize = ((Number) simParams.get("step_size")).intValue();
        }
    }

    private long parseInput(long time, Map<String, Map<String, InputMessage
↪<MessageModel>>> inputResult, long maxAdvance) {
        inputResult.forEach((model, inputMessageMap) -> {
            getEntities(AnnotatedExampleModel.class)
                    .get(model)
                    .setDelta(inputMessageMap
                            .values()
                            .stream()
                            .mapToDouble(value -> value.getInputMessage().delta)
                            .sum());
        });
        return maxAdvance;
    }

    @Override
    public long simulationStep(long time, Map<String, Object> inputs, long␣
↪maxAdvance) {
        getEntities(AnnotatedExampleModel.class).values().
↪forEach(AnnotatedExampleModel::step);
        return time + stepSize;
    }
}
```

Starting the simulator

For the simulators to work, we need to add a main method that will start the simulation for the java side:

```
public class SimulationStarter {
    public static void main(String[] args) throws Throwable {
        Simulator sim = new ExampleModelSim();
        //TODO: Implement command line arguments parser (http://commons.apache.org/
↪proper/commons-cli/)
        if (args.length < 1) {
            String ipaddr[] = {"127.0.0.1:5678"};
            SimProcess.startSimulation(ipaddr, sim);
        } else {
            SimProcess.startSimulation(args, sim);
        }
    }
}
```

Swap line 11 with the MultiModelExampleSimulator to use the second simulator.

5.10.4 Connecting the simulator to mosaik

We use the same scenario as in our Python example *demo1*. The only thing we have to change is the way we connect our simulator to mosaik. There are two ways to do this:

- **cmd:** mosaik calls the Java API by executing the command given in *cmd*. Mosaik starts the Java-API in a new process and connects it to mosaik. This works only if your simulator runs on the same machine as mosaik.
- **connect:** mosaik connects to the Java-API which runs as a TCP server. This works also if mosaik and the simulator are running on different machines.

For more details about how to connect simulators to mosaik see the section about the *Sim Manager* in the mosaik-documentation.

Connecting the simulator using *cmd*

We have to give mosaik the command how to start our Java simulator. This is done in *SIM_CONFIG*. The marked lines show the differences to our Python simulator.

```
# Sim config. and other parameters
SIM_CONFIG = {
    'JExampleSim': {
        'cmd': 'java -cp JExampleSim.jar de.offis.mosaik.api.utils.generics.
↪SimulationStarter %(addr)s',
    },
    'Collector': {
        'cmd': 'python collector.py %(addr)s',
    },
}
END = 10 * 60  # 10 minutes
```

The placeholder *%(addr)s* is later replaced with IP address and port by mosaik. If we now execute demo_1.py we get the same *output* as in our Python-example.

Note: The command how to start the Java simulator may differ depending on your operating system. If the command is complex, e. g. if it contains several libraries, it is usually better to put it in a script and than call the script in *cmd*.

Connecting the simulator using *connect*

In this case the Java API acts as TCP server and listens at the given address and port. Let's say the simulator runs on a computer with the IP-address 1.2.3.4. We can now choose a port that is not assigned by default. In our example we choose port 5678. Make sure that IP-address and port is accessible from the computer that hosts mosaik (firewalls etc.).

Note: Of course you can run mosaik and your simulator on the same machine by using `127.0.0.1:5678` (localhost). You may want to do this for testing and experimenting. Apart from that the connection with cmd (*see above*) is usually the better alternative because you don't have to start the Java part separately.

We have to tell mosaik how to connect to the simulator. This is done in *SIM_CONFIG* in our scenario (demo1):

```
# Sim config. and other parameters
SIM_CONFIG = {
    'JExampleSim': {
```

(continues on next page)

(continued from previous page)

```
        'connect': '1.2.3.4:5678',
    },
    'Collector': {
        'cmd': 'python collector.py %(addr)s',
    },
}
END = 10 * 60  # 10 minutes
```

The marked lines show the differences to our Python simulator. Our simulator is now called *JExampleSim* and we need to give the simulator's address and port after the *connect* key word.

Now we start JExampleSim. To tell the mosaik-Java-API to run as TCP-server is done by starting it with "server" as second argument. The first command line argument is IP-address and port. The command line in our example looks like this:

```
java -cp JExampleSim.jar de.offis.mosaik.api.JExampleSim 1.2.3.4:5678 server
```

If we now execute demo_1.py we get the same *output* as in our Python-example.

Note: You can find the source code used in this tutorial in the mosaik-source-files in the folder `docs/tutorial/code`.

5.10.5 ParserHelper class

As last note, the functions used to parse the mosaik data into models and creating the metamodel are available via the ParserHelper class. Even if you don't want to use one of the simulators, this class has some QoL methods you may want to use. Check the inbuilt java-docs of the library to learn more about the methods available and how to use them.

Jupyter tutorial

5.11 Use mosaik with Jupyter Notebooks

You can use mosaik with Jupyter notebooks to have an interactive experience and to write or use tutorials or to document your steps. We have pre-created tutorials written in Jupyter notebooks which you can use. This page gives a short explanation on how to use those.

Note: Starting from version 3.2, mosaik uses asyncio, which does not support nested event loops natively. This leads to an error when using mosaik in a Jupyter notebook. Luckily, the issue can be resolved by installing the library nest_asyncio and calling

---

```
import nest_asyncio
nest_asyncio.apply()
```

at the beginning of your notebook (before creating the mosaik `World`).

Step 1 - Use Jupyter in VS Code: Jupyter notebooks can be used with different UIs. A simple approach is to use Visual Studio Code with its Jupyter extension. You can find detailed information on how to use Jupyter in VS Code in the VS Code documentation.

Step 2 - mosaik Jupyter repository: Checkout our Jupyter notebook examples repository, e.g. via `git clone https://gitlab.com/mosaik/examples/mosaik-tutorials-on-binder.git` and open the folder in VS Code.

Step 3 - Virtual Environment: Create a virtual environment with the requirements installed. See the following screenshots for an example on how to create a virtual environment.

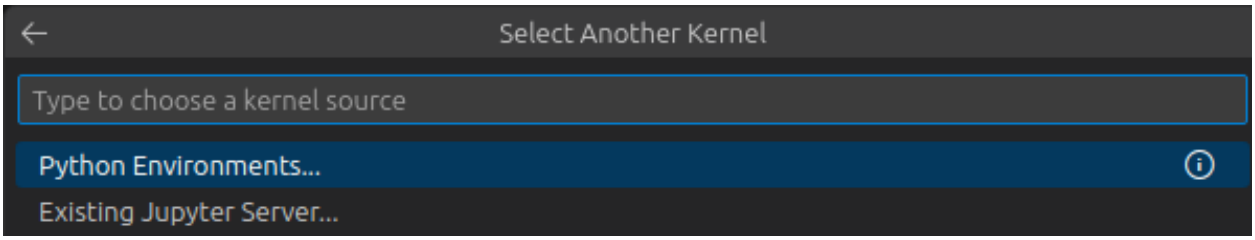


Fig. 3: Choose the kernel.

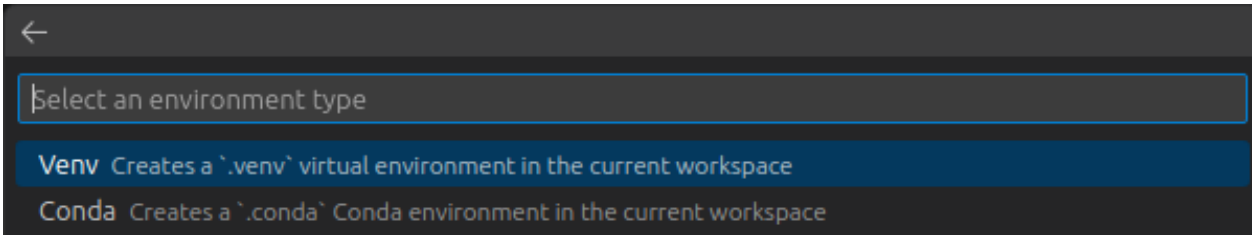


Fig. 4: Select the environment type.

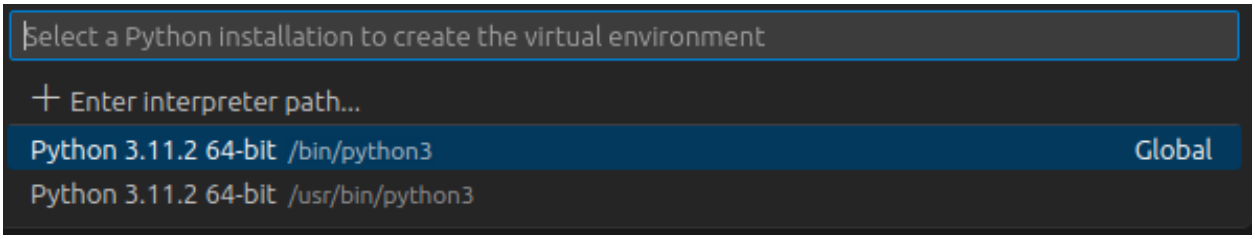


Fig. 5: Select a python version to create a virtual environment.

Step 4 - Run a Jupyter notebook: Choose one of the available notebooks, e.g., `_02_simulator_mosaik.ipynb` and open it. You can now run the code blocks step by step or all at once with the "Run All" button on the top. Feel free to play with the example code, extend or change it to your needs or to create your own notebooks based on these examples.

Visualize scenarios

5.12 Plotting graphs

Sometimes it is useful to visualize your scenario to understand the behavior of mosaik. You can use the plotting functions in *utils* for different graphs. The parameters are always the same: the world object and the name of the folder where the figures shall be stored in.

Optional parameters are *slice* (see below) and *show_plot* (default: True). With *show_plot* you can control if a window is opened to show the plot in an interactive window. If set to false, the plot is stored directly. If set to true, you can interact with the plot and the chosen view in stored after you close the window.

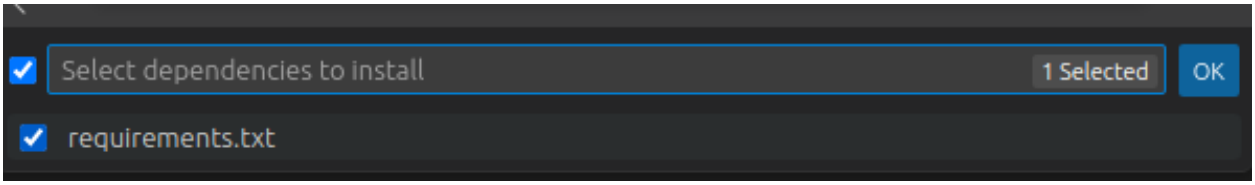


Fig. 6: Choose to install the requirements.

There are four different plots available:

```
world = mosaik.World(SIM_CONFIG, debug=True)
...
mosaik.util.plot_df_graph(world, folder='util_figures')
mosaik.util.plot_execution_graph(world, folder='util_figures')
mosaik.util.plot_execution_time(world, folder='util_figures')
mosaik.util.plot_execution_time_per_simulator(world, folder='util_figures')
```

You need to install *matplotlib* in your environment before using these functions.

5.12.1 Examples

The following examples will be done with the following scenario. This code is just to show how the connections are set up, so that the graphs can be interpreted accordingly. The important part is the part where the entities are connected.

```
import mosaik.util

# Sim config. and other parameters
SIM_CONFIG = {
    'ExampleSim': {
        'python': 'simulator_mosaik:ExampleSim',
    },
    'ExampleSim2': {
        'python': 'simulator_mosaik:ExampleSim',
    },
    'Collector': {
        'cmd': '%(python)s collector.py %(addr)s',
    },
}
END = 10  # 10 seconds

# Create World
world = mosaik.World(SIM_CONFIG, debug=True)

# Start simulators
examplesim = world.start('ExampleSim', eid_prefix='Model_')
examplesim2 = world.start('ExampleSim2', eid_prefix='Model2_')
collector = world.start('Collector')

# Instantiate models
model = examplesim.ExampleModel(init_val=2)
model2 = examplesim2.ExampleModel(init_val=2)
monitor = collector.Monitor()

# Connect entities
world.connect(model2, model, 'val', 'delta')
```

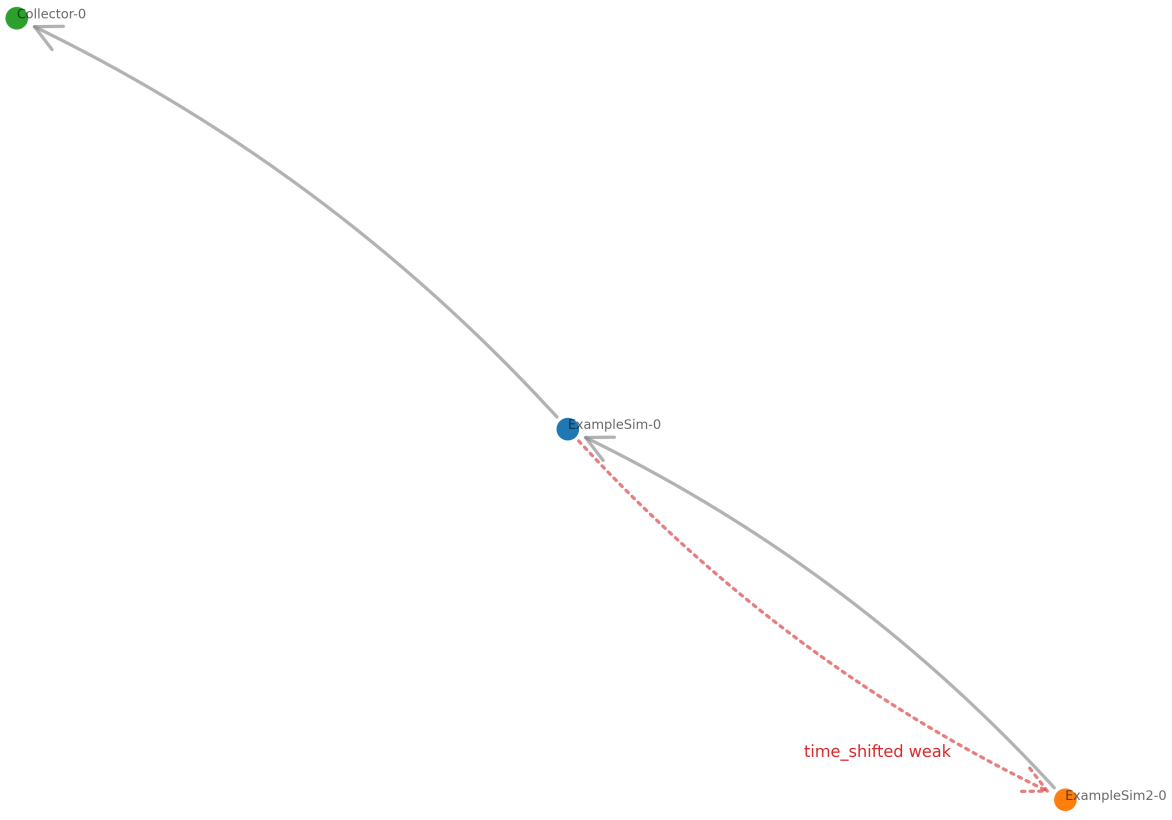
(continues on next page)

(continued from previous page)

```
world.connect(model, model2, 'val', 'delta', initial_data={"val": 1, "delta": 1},␣
↪time_shifted=True, weak=True)
world.connect(model, monitor, 'val', 'delta')

# Create more entities
more_models = examplesim.ExampleModel.create(2, init_val=3)
mosaik.util.connect_many_to_one(world, more_models, monitor, 'val', 'delta')

# Run simulation
world.run(until=END)

mosaik.util.plot_dataflow_graph(world, folder='util_figures')
#mosaik.util.plot_execution_graph(world, folder='util_figures')
#mosaik.util.plot_execution_time(world, folder='util_figures')
#mosaik.util.plot_execution_time_per_simulator(world, folder='util_figures')
```

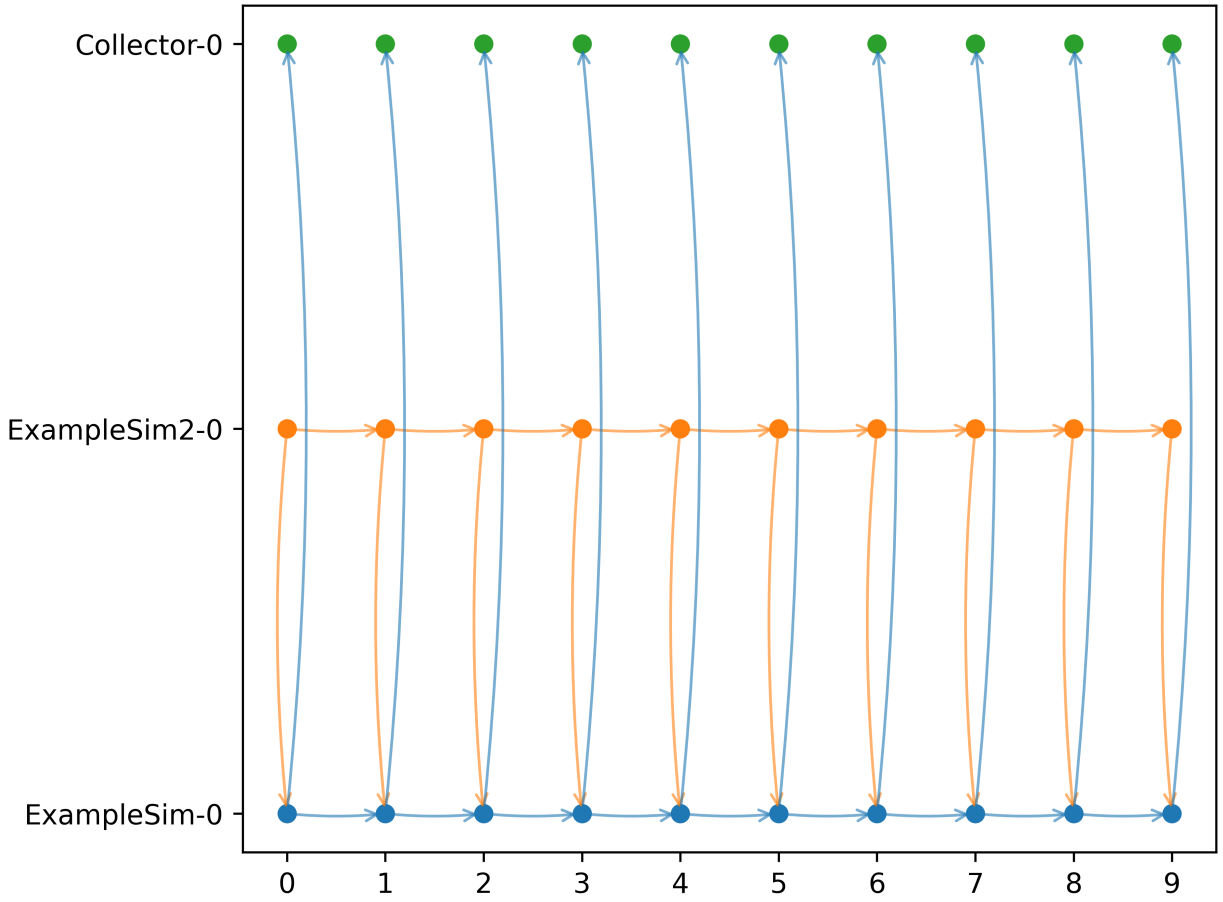
5.12.2 Dataflow graph

The dataflow graph shows the direction of the dataflow between the simulators. In the example below, the ExampleSim simulator sends data to the Collector. The ExampleSim2 sends data to ExampleSim. The dataflow connection from ExampleSim to ExampleSim2 is both weak (dotted line) and timeshifted (red line), which can be seen in the red label.



5.12.3 Execution graph

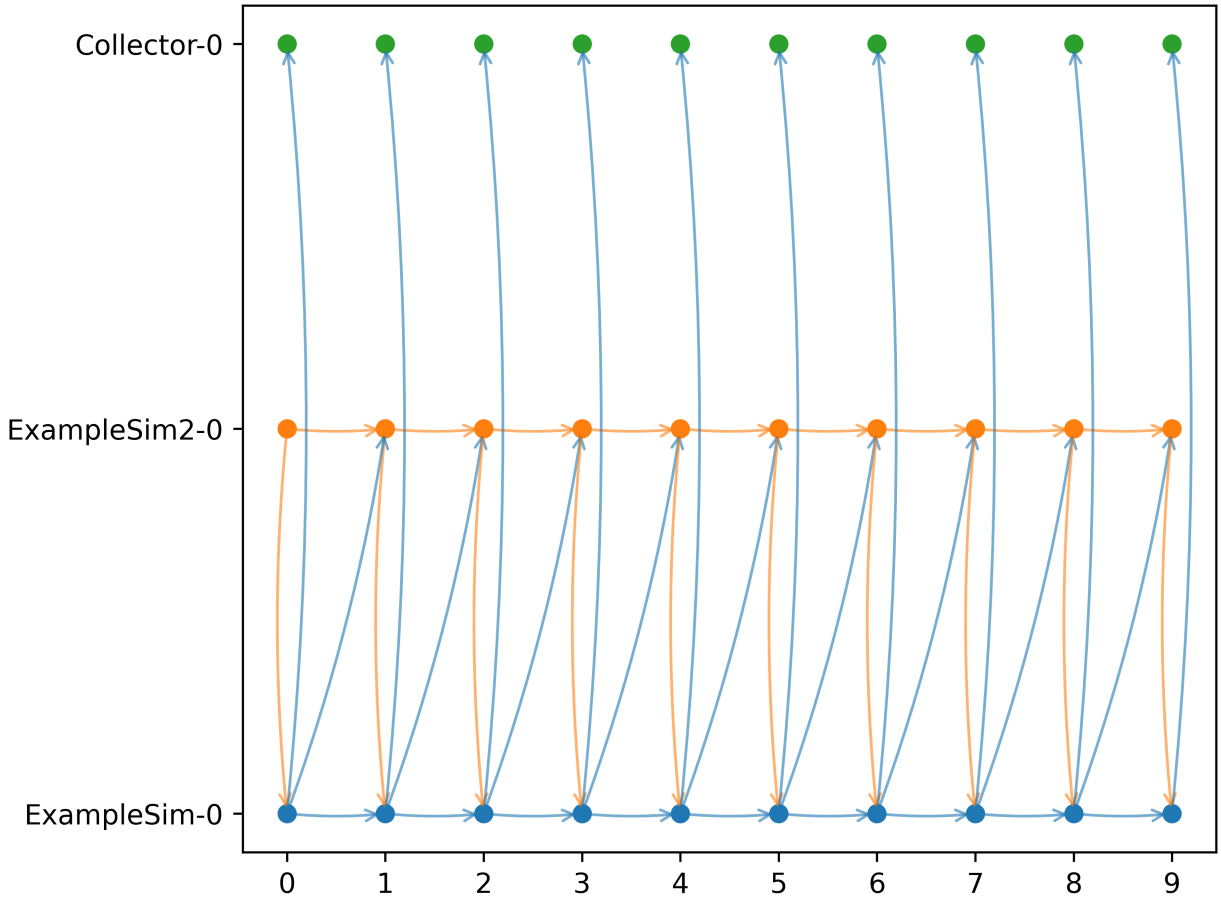
The execution graph shows the order in which the simulators are executed. Differing from the example above, the connection between ExampleSim and ExampleSim2 is only marked as weak, not as timeshifted.

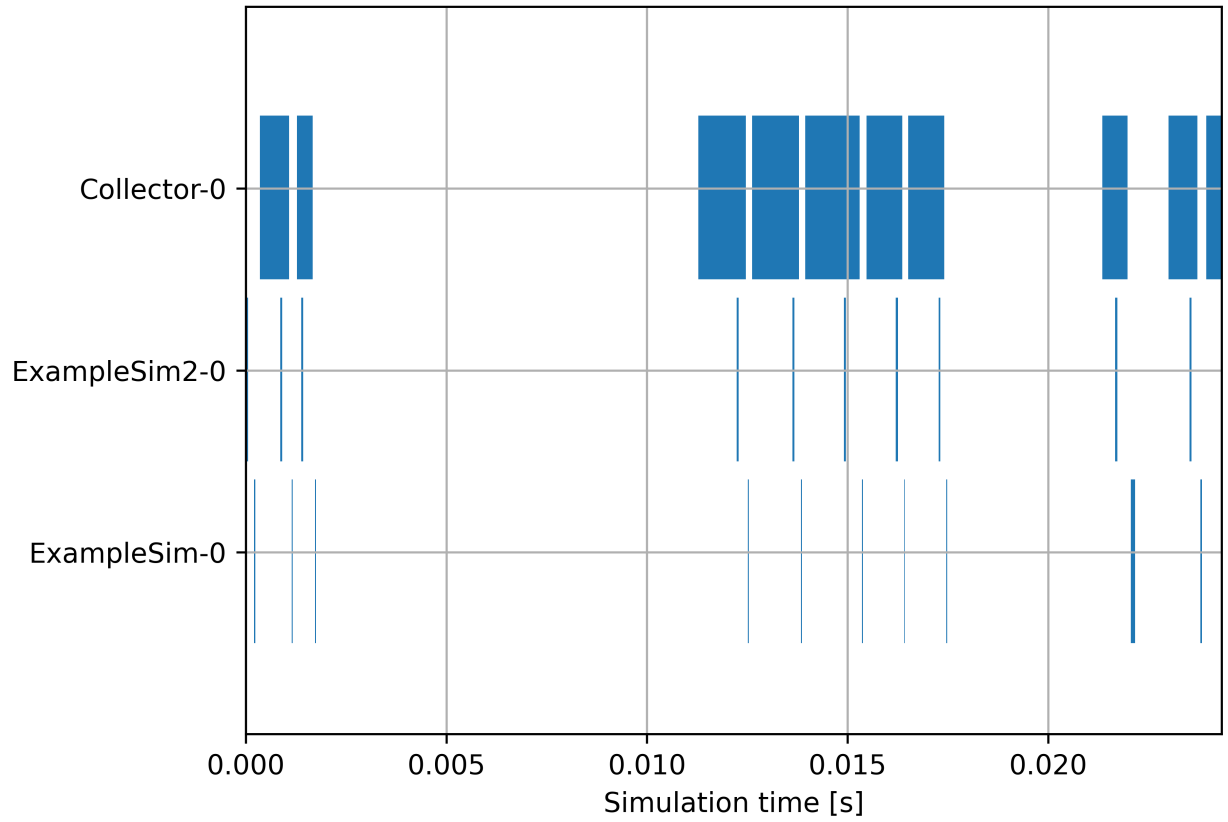


If we add back the timeshift parameter, we get an additional arrow from ExampleSim to ExampleSim2. That is because the data from ExampleSim is used in ExampleSim2 in a timeshifted manner, i.e., from the previous step. This is the *Gauss-Seidel scheme*.

5.12.4 Execution time

The execution time graph shows the execution time of the different simulators so that it can be seen where the simulation takes more or less time. In the example below it can be seen that the Collector uses comparatively more time than the ExampleSim simulators.


Collector-0
ExampleSim2-0
ExampleSim-0
0
1
2
3
4
5
6
7
8
9



5.12.5 Execution time per simulator

The execution time can also be plotted over the simulation steps per simulator, as can be seen in the figure below. You can also set the parameter *plot_per_simulator=True*. In that case, the plots for the different are separated. This is especially useful if the simulators have different step sizes.

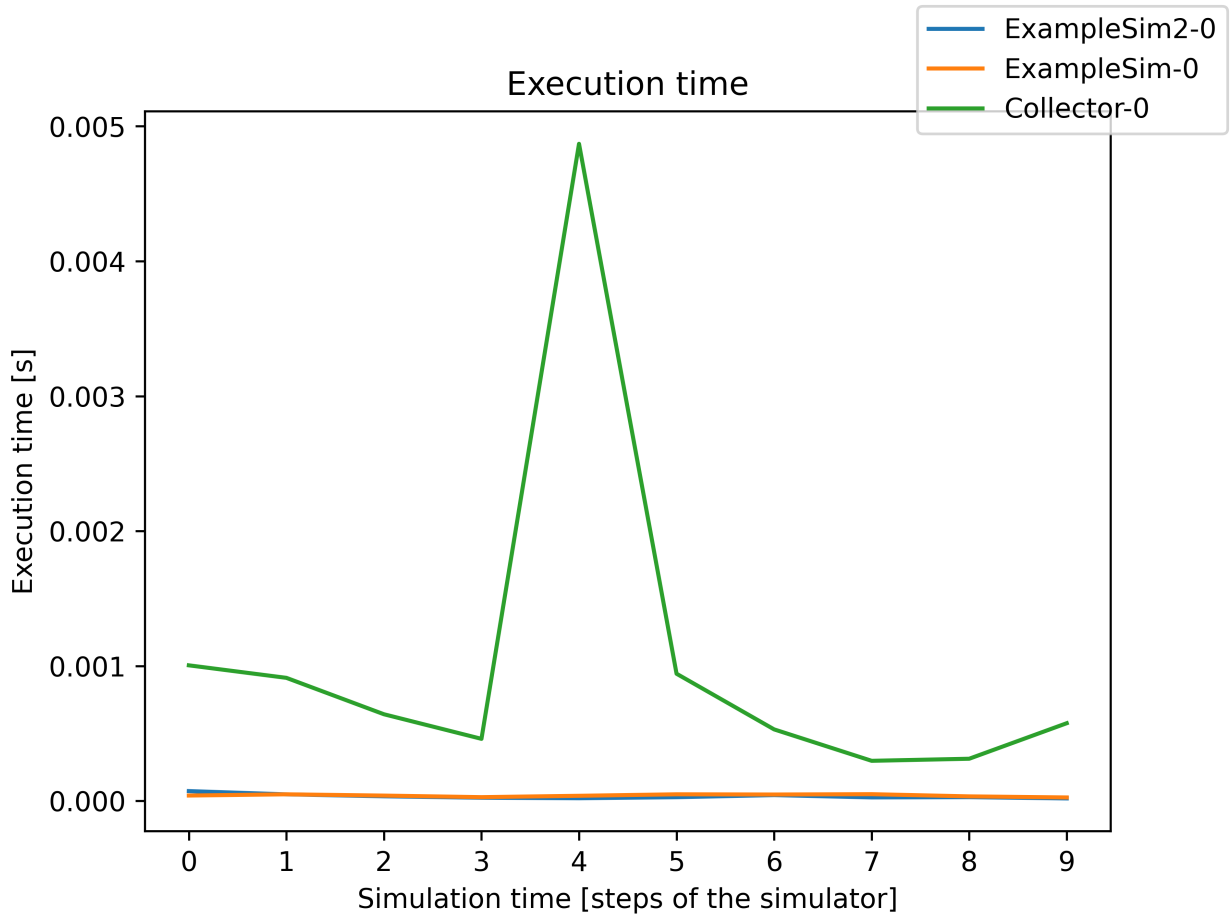
5.12.6 Slicing the graphs

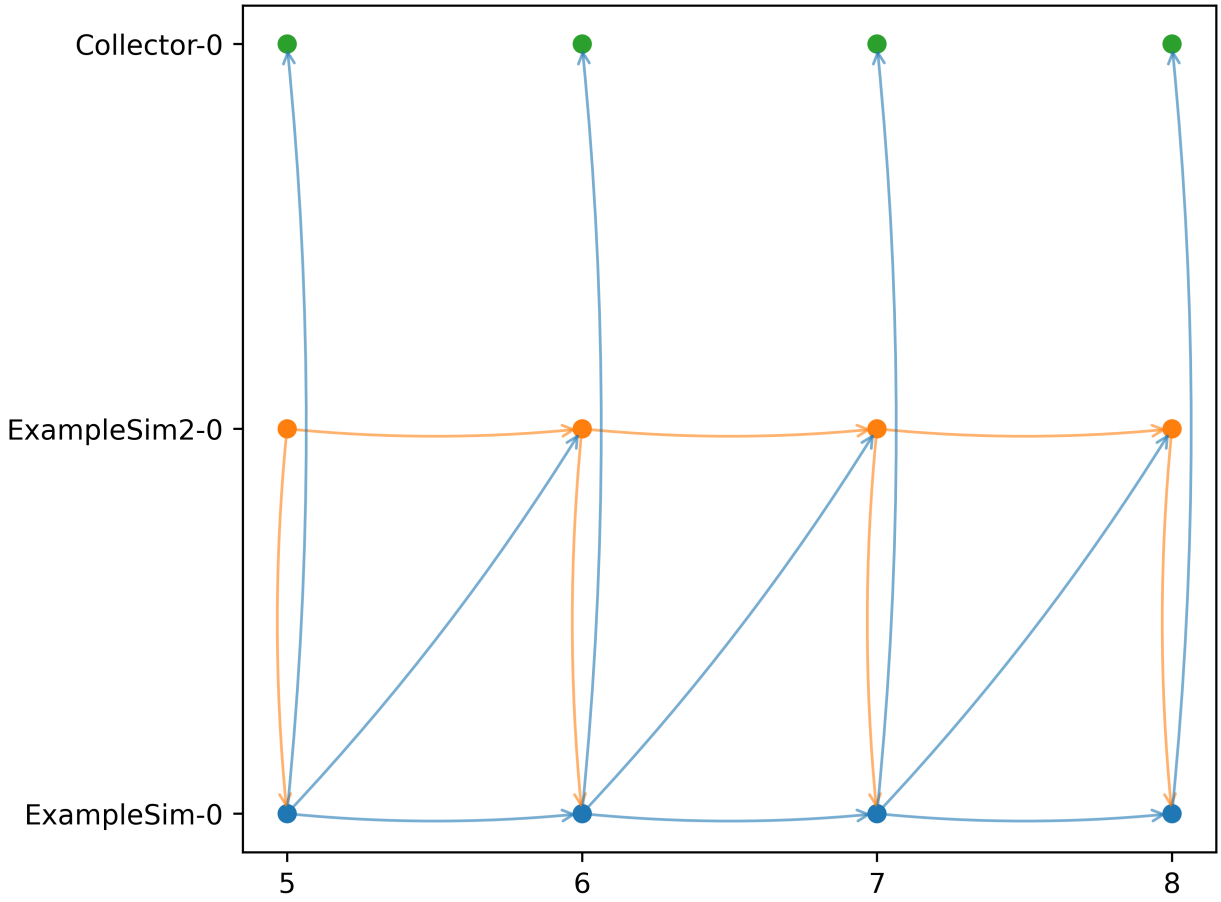
If you are especially interested in a certain part of the simulation to be shown you can slice the time steps for the execution graph, the execution time graph, and the execution time per simulator. You can use the slicing as with Python list slicing. Jumps are not possible. Below you can see a few examples:

```
mosaik.util.plot_execution_graph(world, folder='util_figures', slice=[-5,-1])
mosaik.util.plot_execution_time(world, folder='util_figures', slice=[0,5])
mosaik.util.plot_execution_time_per_simulator(world, folder='util_figures', slice=[-4,
↪-1])
```

Below is the execution graph sliced as shown in the example code above.

Connecting mosaik with Apache Superset


ExampleSim2-0
ExampleSim-0
Collector-0
Execution time
Execution time [s]
0.005
0.004
0.003
0.002
0.001
0.000
0
1
2
3
4
5
6
7
8
9
Simulation time [steps of the simulator]


Collector-0
ExampleSim2-0
ExampleSim-0
5
6
7
8

5.13 Using Apache Superset to visualize simulation data

How can data that is generated through a mosaik simulation be explored without too much hassle and programming knowledge? In this tutorial we will look at Apache Superset, an open source tool for data exploration and visualization. This tutorial is divided into two parts, the installation of Apache Superset and how to couple it with mosaik and the basic usage of Superset. For a more in-depth look at Apache Superset and all of its features please consult the tutorials on the Superset website.

This tutorial is based on a superset and timescaledb instance. Both instances were run through docker on ubuntu.

5.13.1 Getting Started

In this part of the tutorial we will show you how to install the necessary tools to use Apache Superset with mosaik.

5.13.2 Installing a database and collecting data

To use Apache Superset a SQL database that contains simulation data is needed. For this there are currently three options in mosaik.

- A MySQL database
- A PostgreSQL database
- A Timescale database

To install one of the databases locally follow corresponding the link. (This tutorial is written with the Timescale database in mind. However, the other databases, should follow similar steps.)

Make sure to change the port of the database to something different from 5432 if you want to run the database and superset locally, as superset uses the same default port for its databse. When using docker the command should look something like this:

```
$ docker run -d --name [YOURCONTAINERNAME] -p [YOURPORT]:5432 ...
```

After installing the database the corresponding mosaik adapter can be used to save simulation data into the database:

- MySQL adapter
- PostgreSQL/Timescale adapter

For further explanation regarding this.

5.13.3 Installing Superset

This tutorial is written using the production version of Superset based on the commit 0c083bdc1af4e6a3e17155246a3134cb5cb5887d .

To install the production version of Superset locally clone the Superset repository using the following command:

```
$ git clone https://github.com/apache/superset.git
$ cd superset
```

Afterwards a secret key needs to be set for the production version. For this the file superset_config.py is needed. It can be copied into the right place using the command:

```
$ cp ./docker/pythonpath_dev/superset_config_local.example ./docker/pythonpath_dev/
↪superset_config_docker.py
```

When this is done a Secret Key can be generated. The following command can be used on linux:

```
$ openssl rand -base64 42
```

and then be added into the superset_config `./docker/pythonpath_dev/superset_config_docker.py` like so:

```
$ SECRET_KEY = 'YOUR_OWN_RANDOM_GENERATED_SECRET_KEY'
```

Please be sure to remove any other option from the configuration or make sure you need the other configuration options and know what they do.

The Superset instance can be started with the following commands:

```
$ docker-compose -f docker-compose-non-dev.yml pull
$ docker-compose -f docker-compose-non-dev.yml up
```

Afterwards the instance can be found at the webaddress `http://localhost:8088/`. The default login username and password are admin.

5.13.4 Connecting Superset with the Mosaik database

To connect superset with the database both superset and the database need to be online. This connection is done in the superset web application. The connection between superset and the database is done in the `settings -> Database Connections` menu.

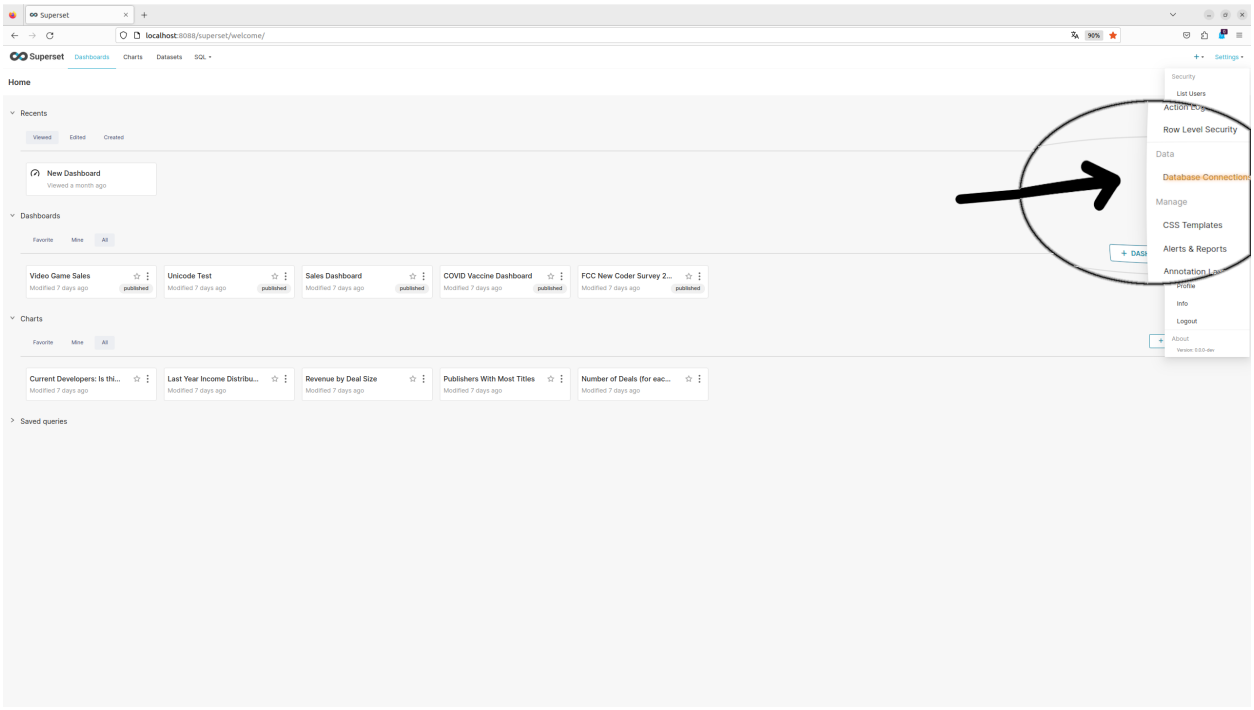


Fig. 7: Database Connections Setting

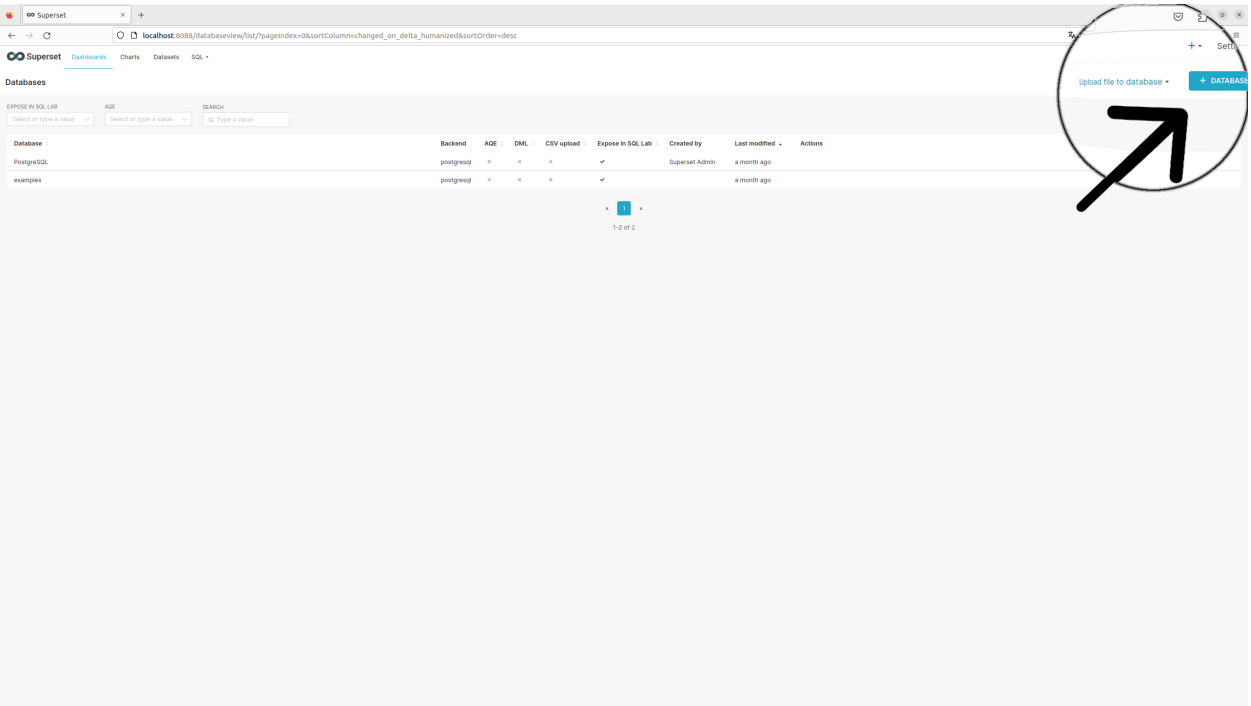


Fig. 8: Button to click for adding a database

Afterwards a new Database is added by clickin on the `Database +` Button.

This initiates the add database dialog consisting of three steps:

5.13.5 Visualizing Data in Apache Superset

After connecting the database to superset the data can now be visualized. This tutorial shows data that is saved in a Timescale database. *This data is saved using the MultiWriter2 of the mosaik Timescale adapter.* To do this first the data needs to be extracted from the databae using SQL. This is done in the SQL Lab:

I the SQL Lab the database the database, schema and table schema of a table in the database can be selected on the left side. On the right side a sql query can be built. In this example we use a simple query to get all of the data from the table. If you are using the single writer from the mosaik timescale component the SQL query will look a bit different with it either being a double cast in case of the json table_type:

```
SELECT time, CAST(CAST(values->'Grid-0.0-LV1.1 Bus 1' AS VARCHAR) AS DOUBLE␣
↪PRECISION) AS "BUS 1" FROM testing_json
 WHERE value_type = 'va_degree'
```

And it being a single cast when it being the table_type string:

```
SELECT time, CAST(value AS DOUBLE PRECISION) FROM testing_string
 WHERE value_type = 'va_degree'
```

After extracting the wanted data using a SQL query it needs to be saved as a dataset by running the query and afterwards using the save button:

Clicking the `Save & Explore` Button will open up the Chart creation view of superset. This can also be done afterwards by selecting the wanted dataset in the datasets tab.

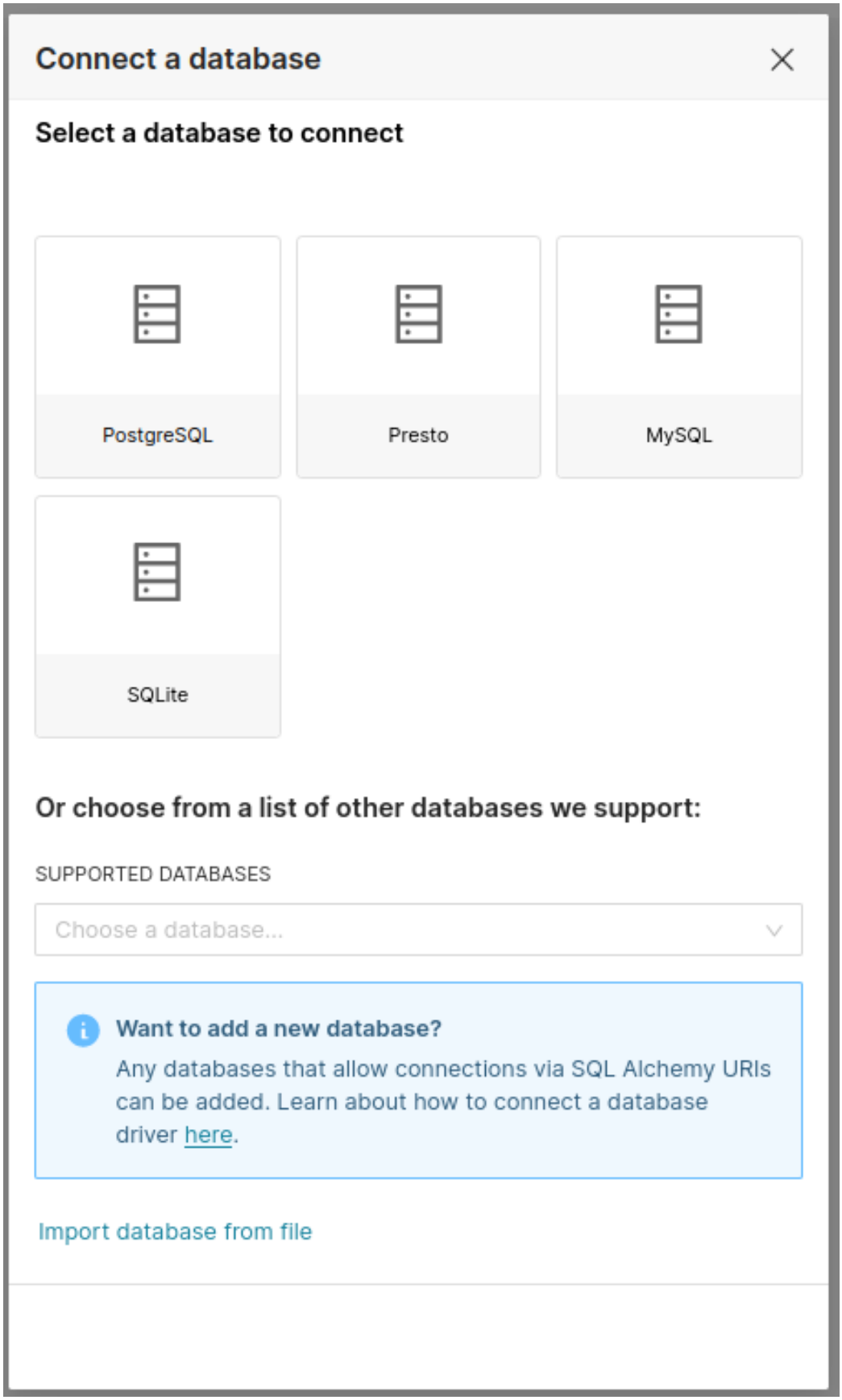


Fig. 9: Step 1: Choosing the correct database(PostgreSQL in this example)

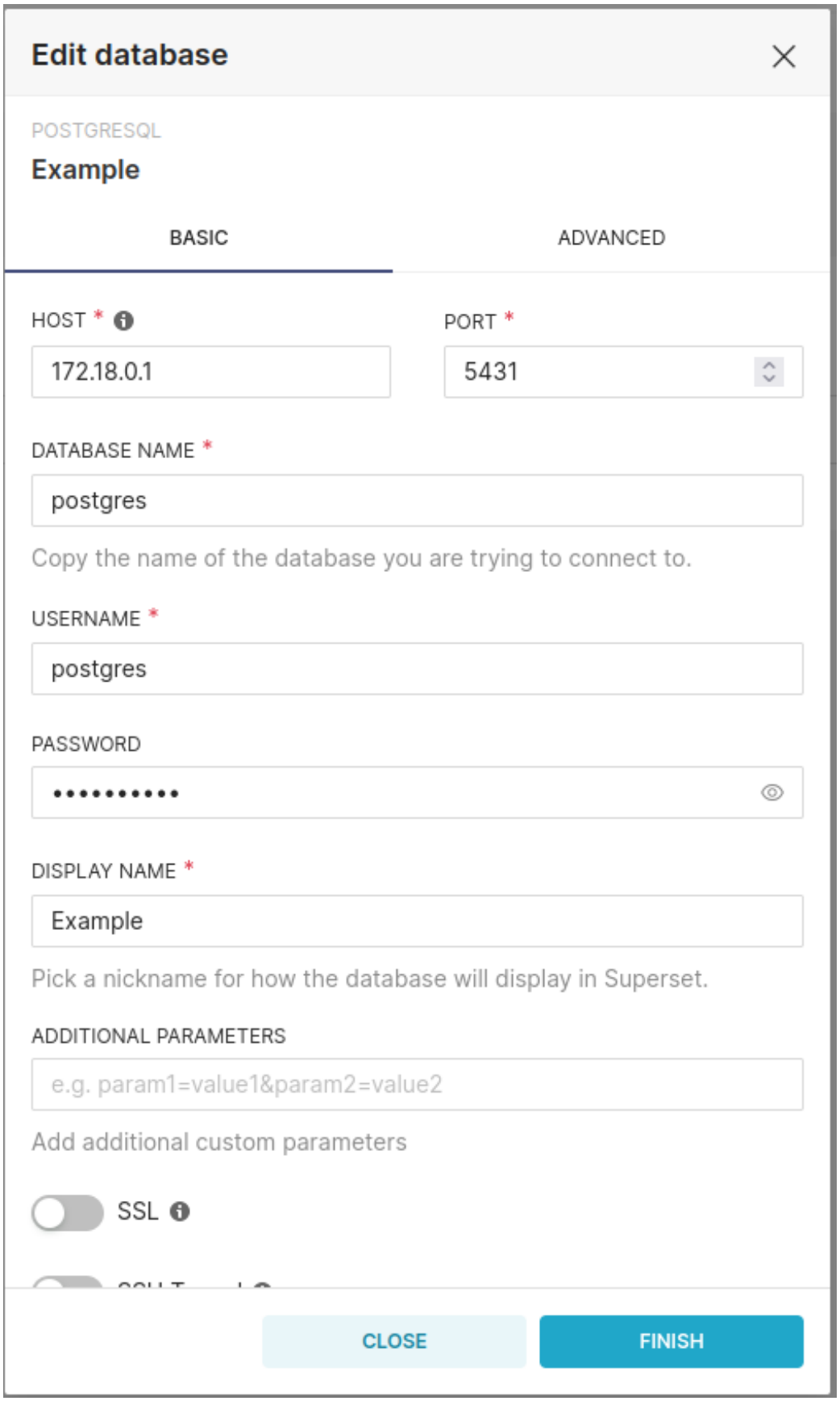


Fig. 10: Step 2: Adding the database Credentials. If the database i run locally the IP-Address is 172.18.0.1 by default. If using Windows the IP might be host.docker.internal.

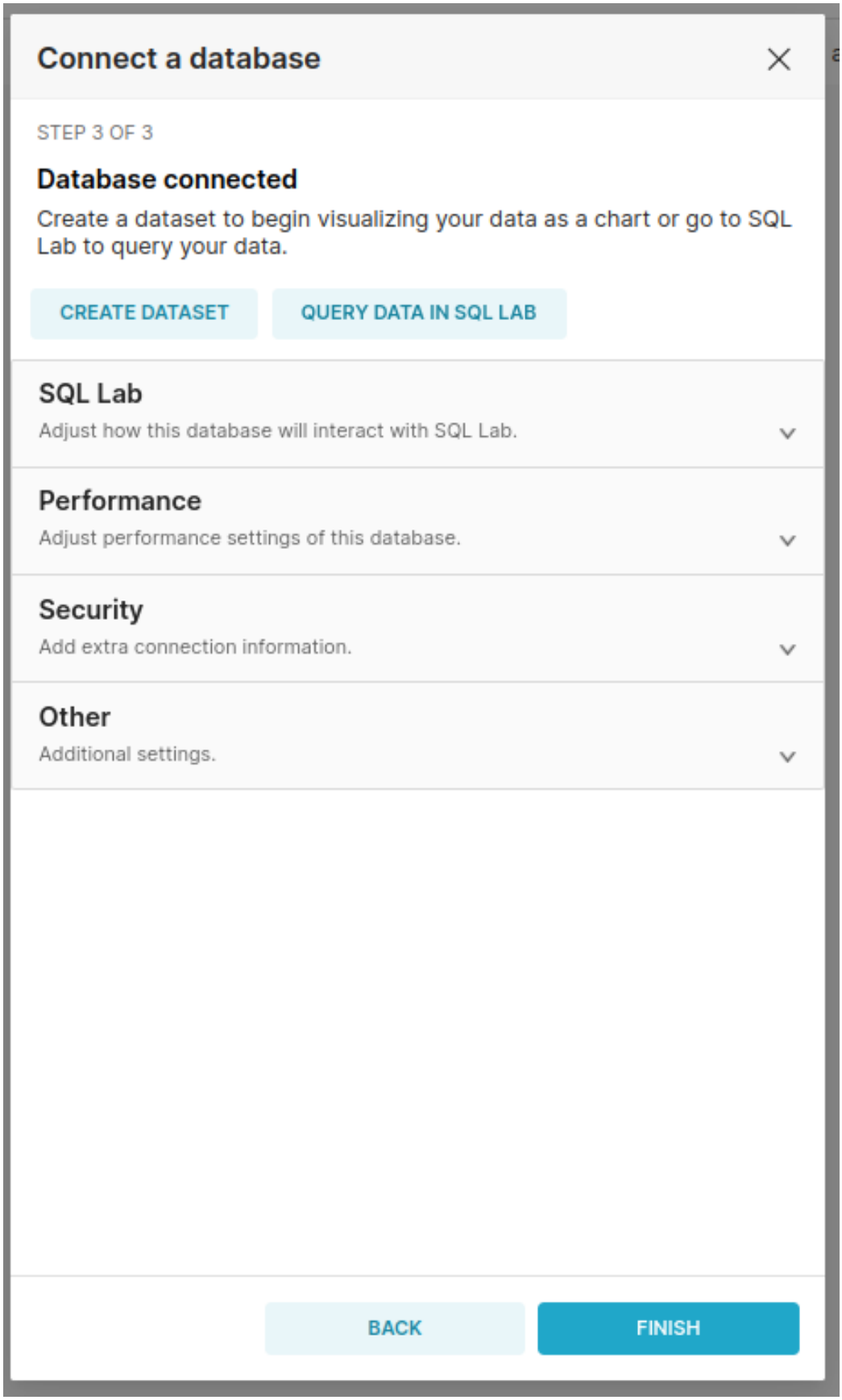


Fig. 11: Step 3: Finishing the setup

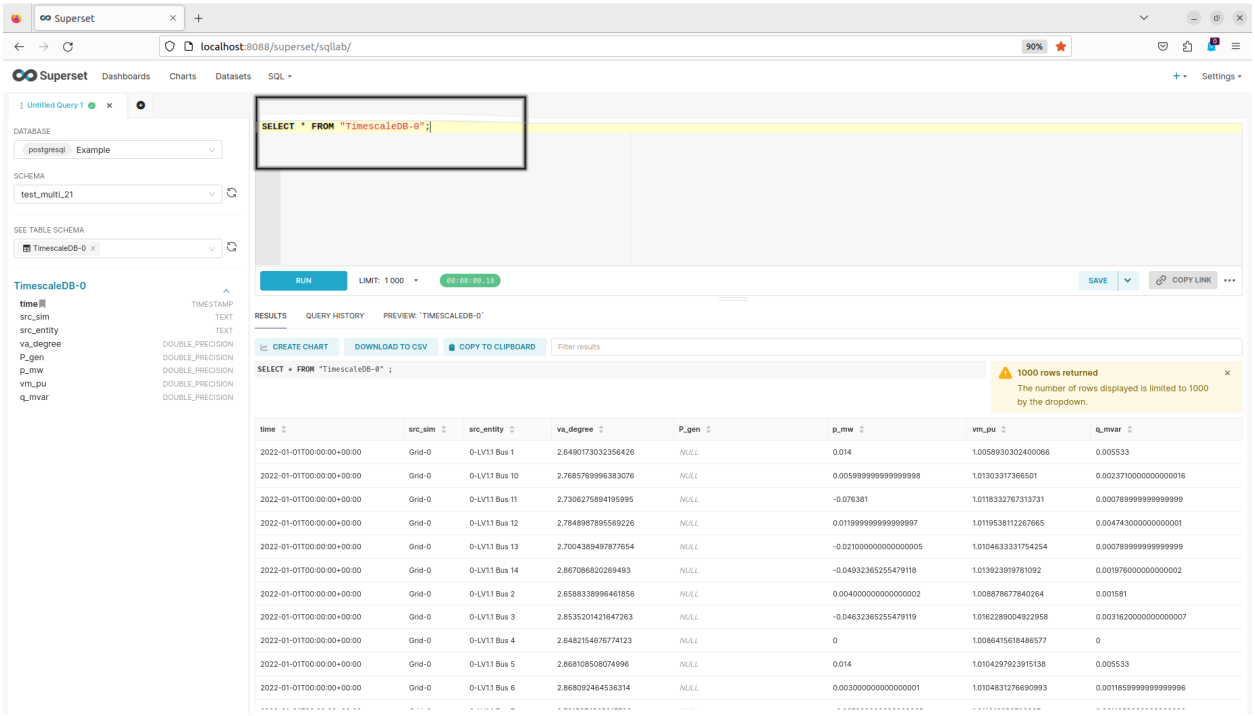


Fig. 12: View of the SQL Lab

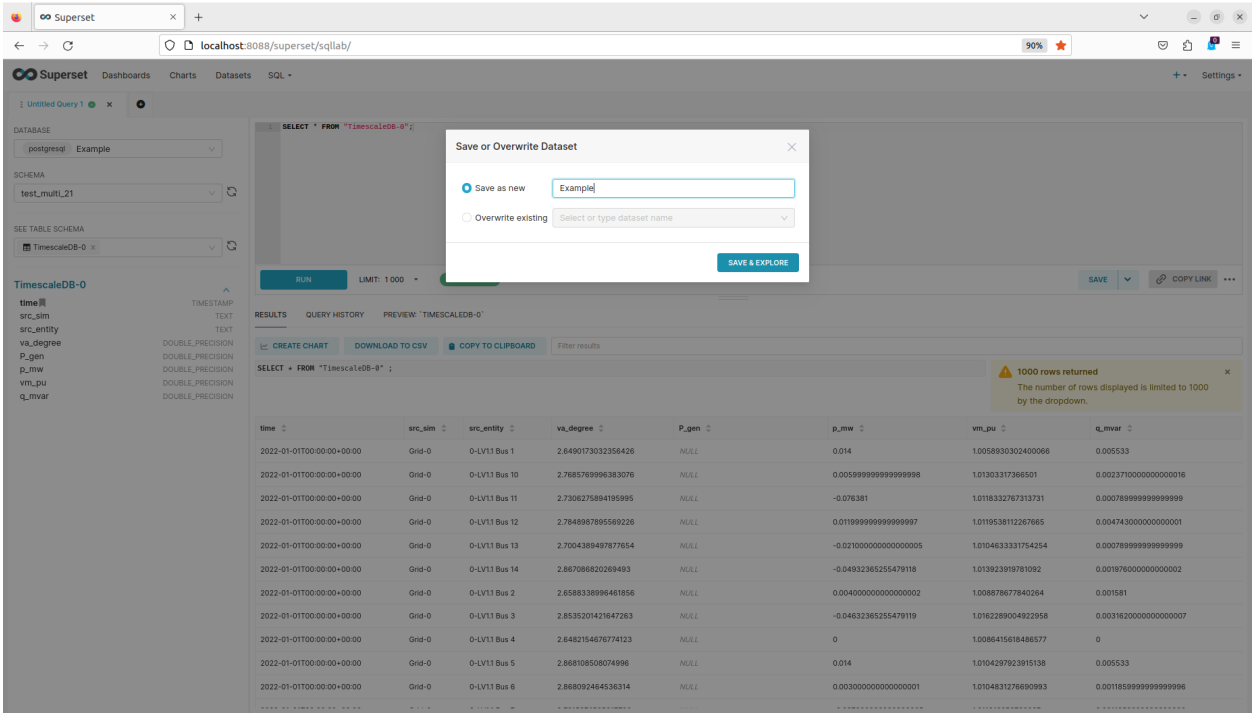


Fig. 13: View of saving the dataset in the SQL Lab

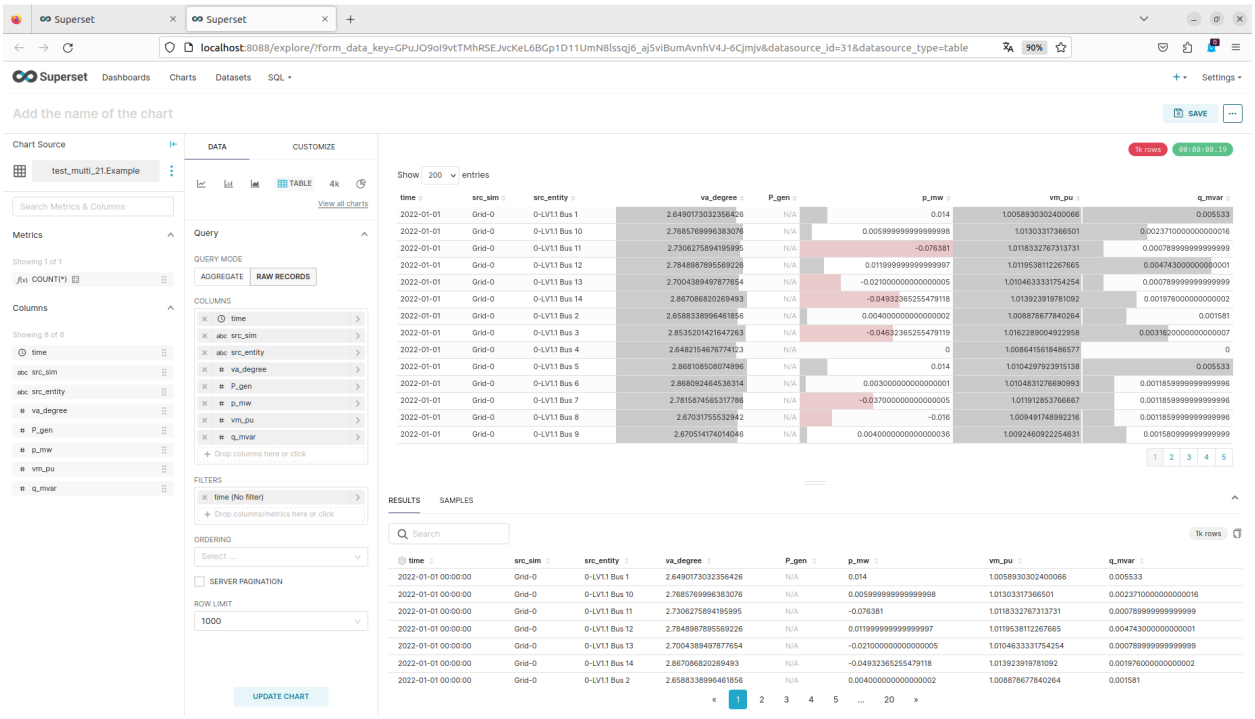


Fig. 14: Chart View of superset

The default chart view of superset can be divided into two important parts. The left side where you can chose the kind of chart to create as well as input the data from the dataset into the chart and the right chart where the chart will be displayed.

For this example lets start by selecting a line chart from the left side and then adding data to the relevant fields.

After changing the chart to line chart the relevant fields to fill out are **the x-Axis, which in most cases will be the time column, and the metrics, which represent te y values.** Superset can not display simple y value, it is always a sql function. If a simple x/y comparison is needed the avg/min/max of the y values can be used since for only one value this is the value itself.

For selecting the x Axis you can chose from your dataset columns. Most of the time you want the simple time value but a custom sql query can also be used.

When selecting a metric there are many basic sql aggregation functions to choose from.

After selecting the metrics you can render the chart by clickin the `Create Chart` or `Update Chart button`

Multiple metrics can be selected but only one x-Axis.

For this example I selected the average, minmum and maximum va_degree of Electric Buses over the timespan of one day in seconds. If for your chart you cannot see the graph try making the time grain smaller.

There is a number of different charts available to visualize the data. After finishing your chart it needs to be saved inside a dashboard. This is done by clicking the save button and giving the chart a name and either picking an existing dashboard or selecting the name of a new dashboard to be created.

After saving the chart in a dashboard the created/picked dashboard can be found in the dashboard view.

If a dashboard is selected it displays all charts that are saved in it.

Inside a dashboard charts can be updated, removed, looked at in fullscreen, exported and more.

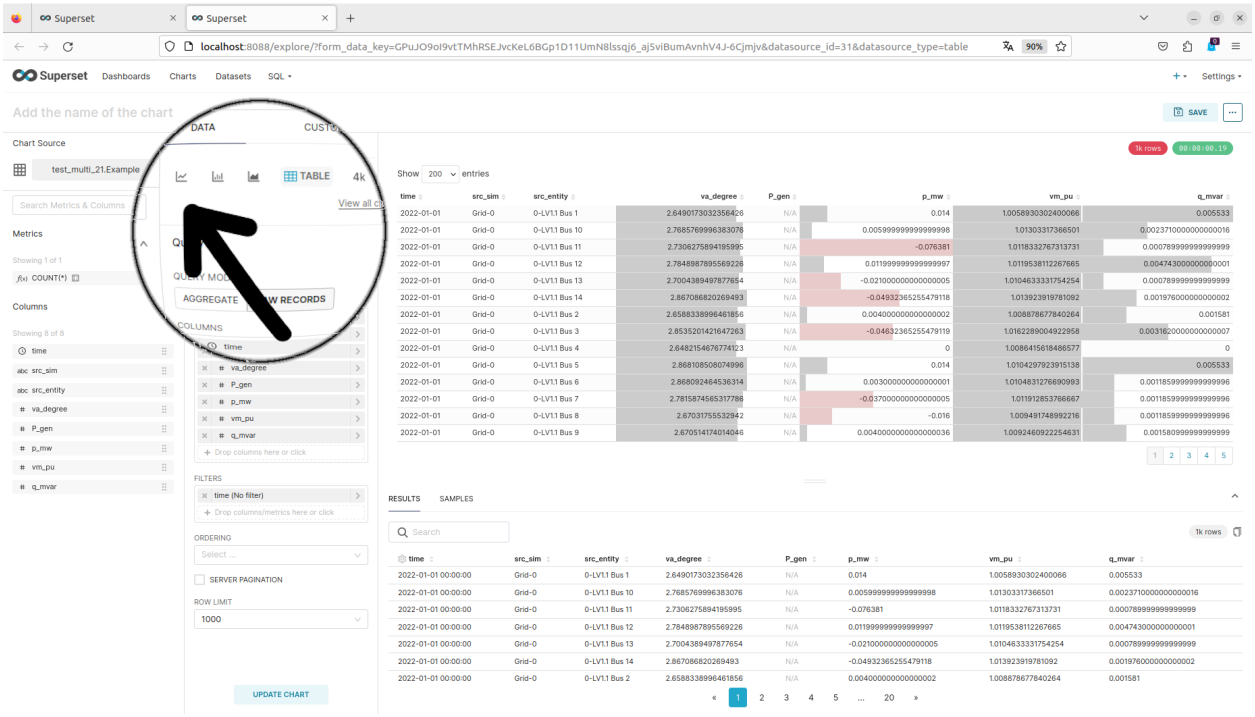
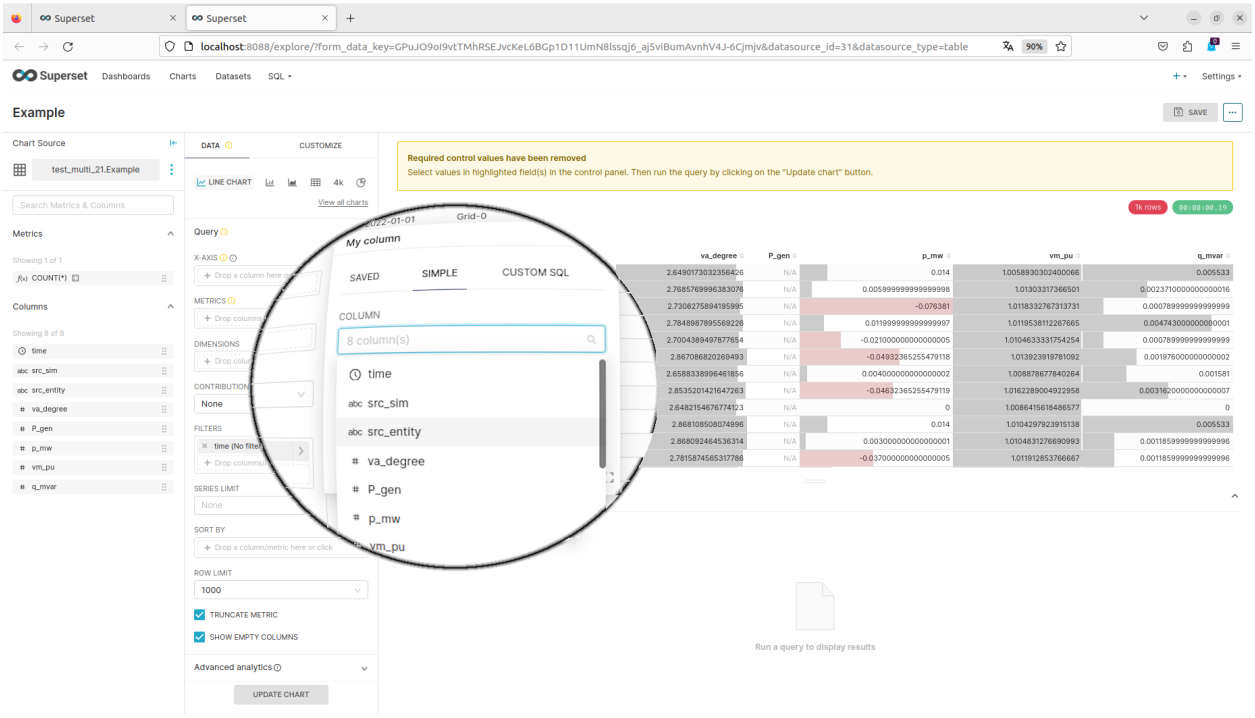
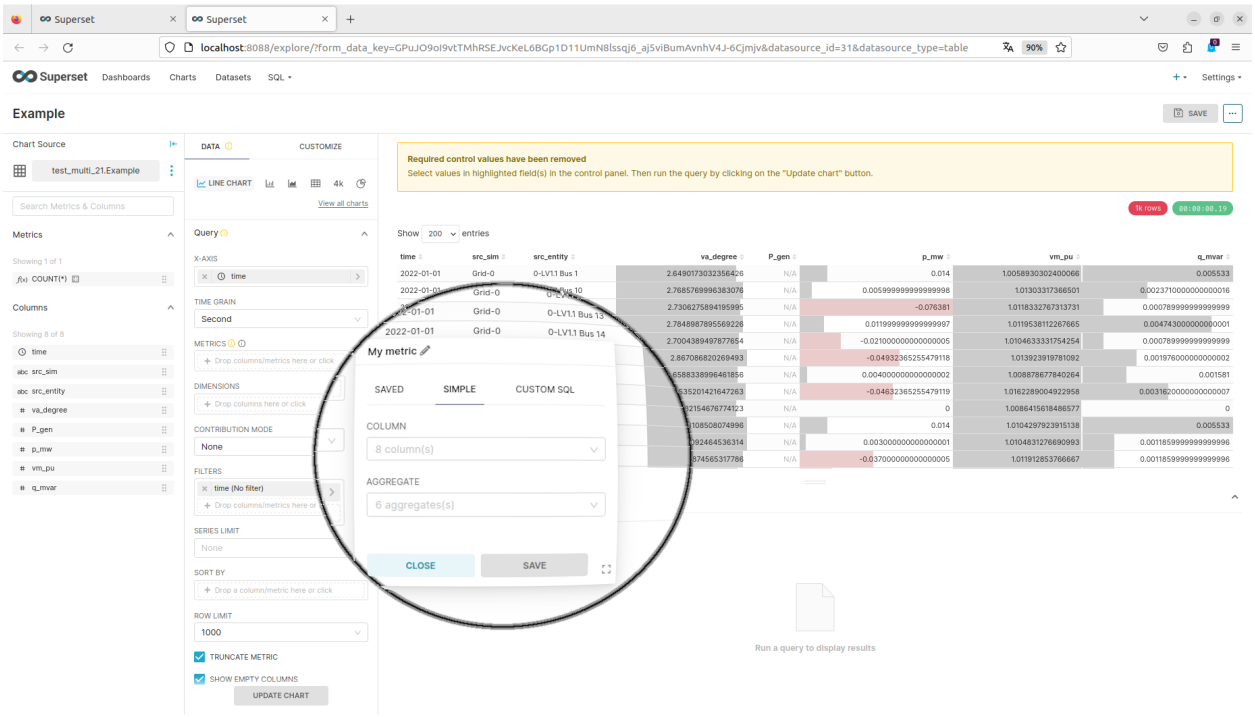
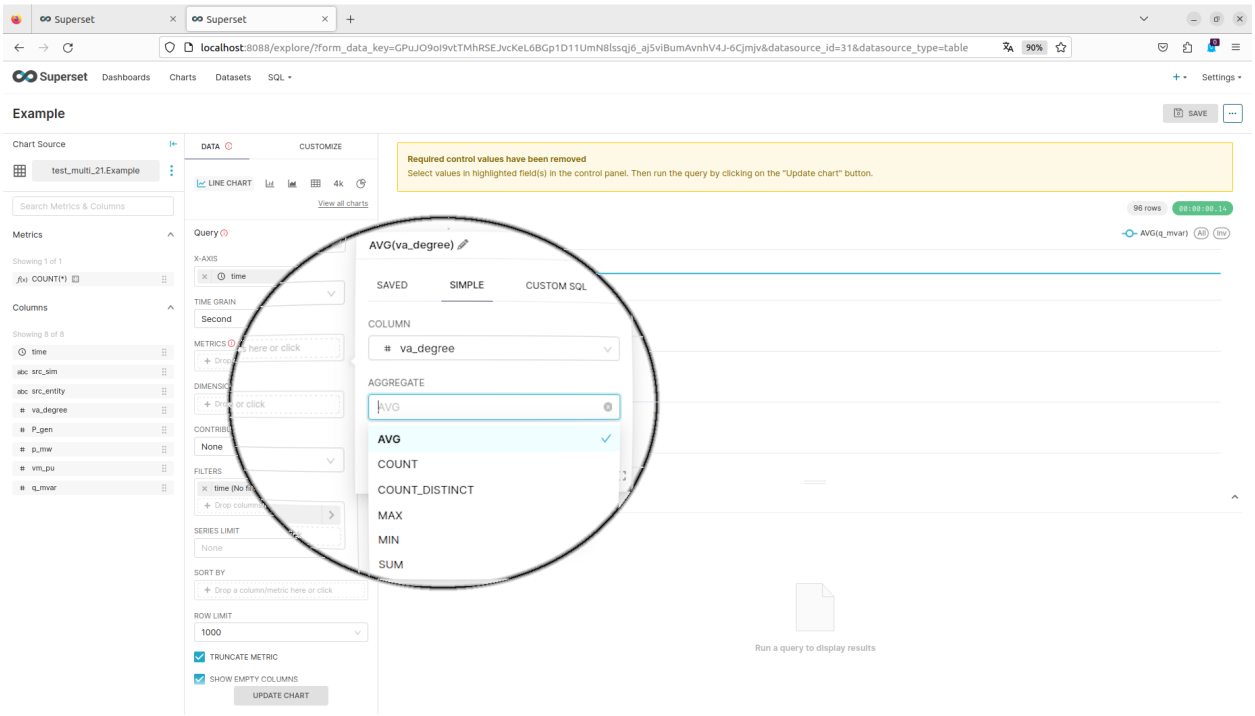


Fig. 15: Changing chart to line chart.




Superset
localhost:8088/explore/?form_data_key=GPuJO9oI9vtTMhRSEJvcKeL6BGp1D11UmN8lssqj6_aj5viBumAvnhV4J-6Cjmjv&datasource_id=31&datasource_type=table
Dashboards
Charts
Datasets
SQL
Settings
Example
SAVE
Chart Source
test_multi_21.Example
DATA
CUSTOMIZE
LINE CHART
Query
X-AXIS
TIME GRAIN
Second
METRICS
DIMENSIONS
CONTRIBUTION MODE
None
FILTERS
SERIES LIMIT
SORT BY
ROW LIMIT
1000
TRUNCATE METRIC
SHOW EMPTY COLUMNS
UPDATE CHART
Required control values have been removed
Select values in highlighted field(s) in the control panel. Then run the query by clicking on the "Update chart" button.
Show 200 entries
My metric
SAVED
SIMPLE
CUSTOM SQL
COLUMN
8 column(s)
AGGREGATE
6 aggregates(s)
CLOSE
SAVE
Run a query to display results


Superset
localhost:8088/explore/?form_data_key=GPuJO9oI9vtTMhRSEJvcKeL6BGp1D11UmN8lssqj6_aj5viBumAvnhV4J-6Cjmjv&datasource_id=31&datasource_type=table
Dashboards
Charts
Datasets
SQL
Settings
Example
SAVE
Chart Source
test_multi_21.Example
DATA
CUSTOMIZE
LINE CHART
Query
X-AXIS
TIME GRAIN
Second
METRICS
DIMENSIONS
CONTRIBUTION MODE
None
FILTERS
SERIES LIMIT
SORT BY
ROW LIMIT
1000
TRUNCATE METRIC
SHOW EMPTY COLUMNS
UPDATE CHART
Required control values have been removed
Select values in highlighted field(s) in the control panel. Then run the query by clicking on the "Update chart" button.
AVG(va_degree)
SAVED
SIMPLE
CUSTOM SQL
COLUMN
va_degree
AGGREGATE
AVG
COUNT
COUNT_DISTINCT
MAX
MIN
SUM
Run a query to display results

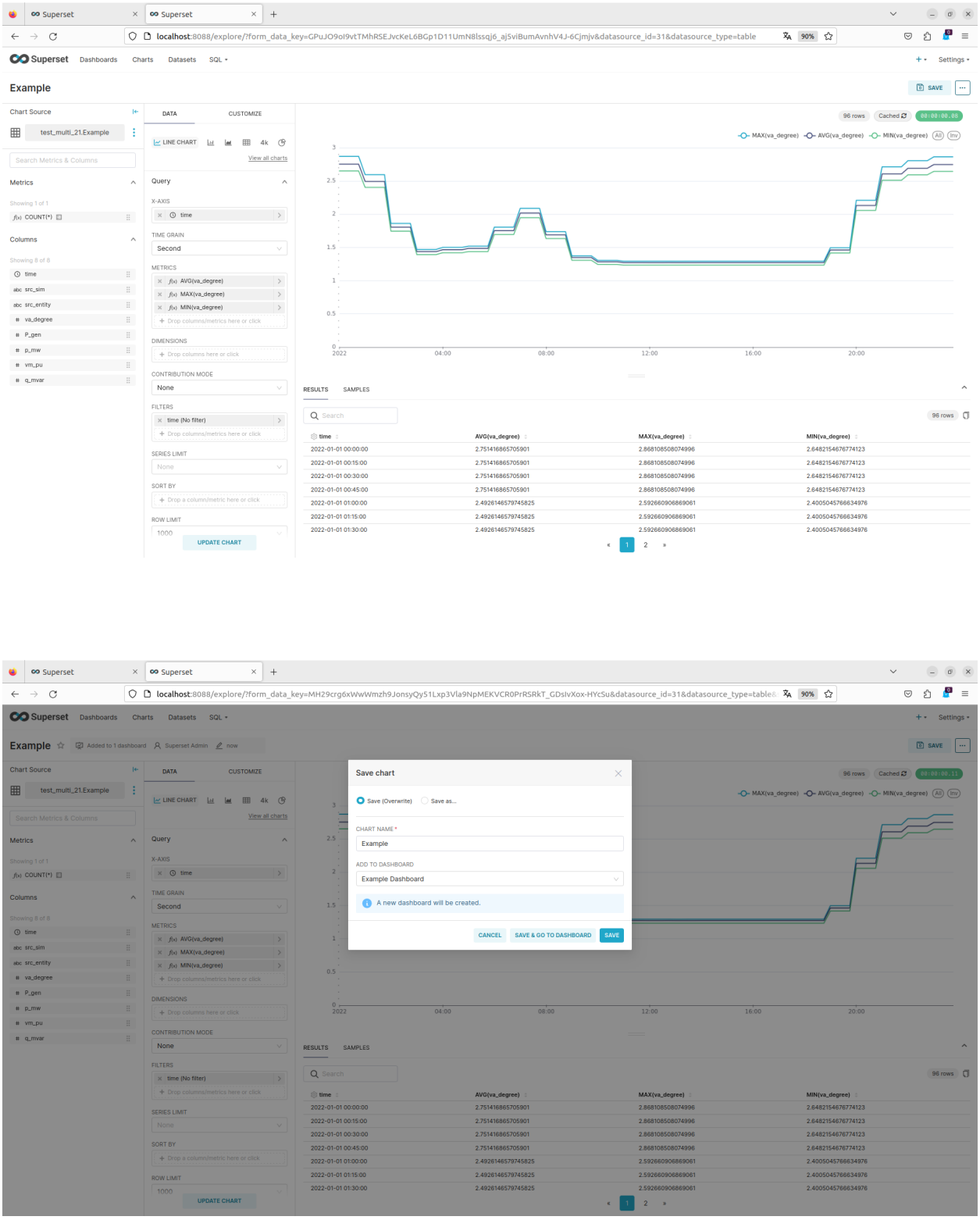


Fig. 16: This is the saving menu of the chart view.

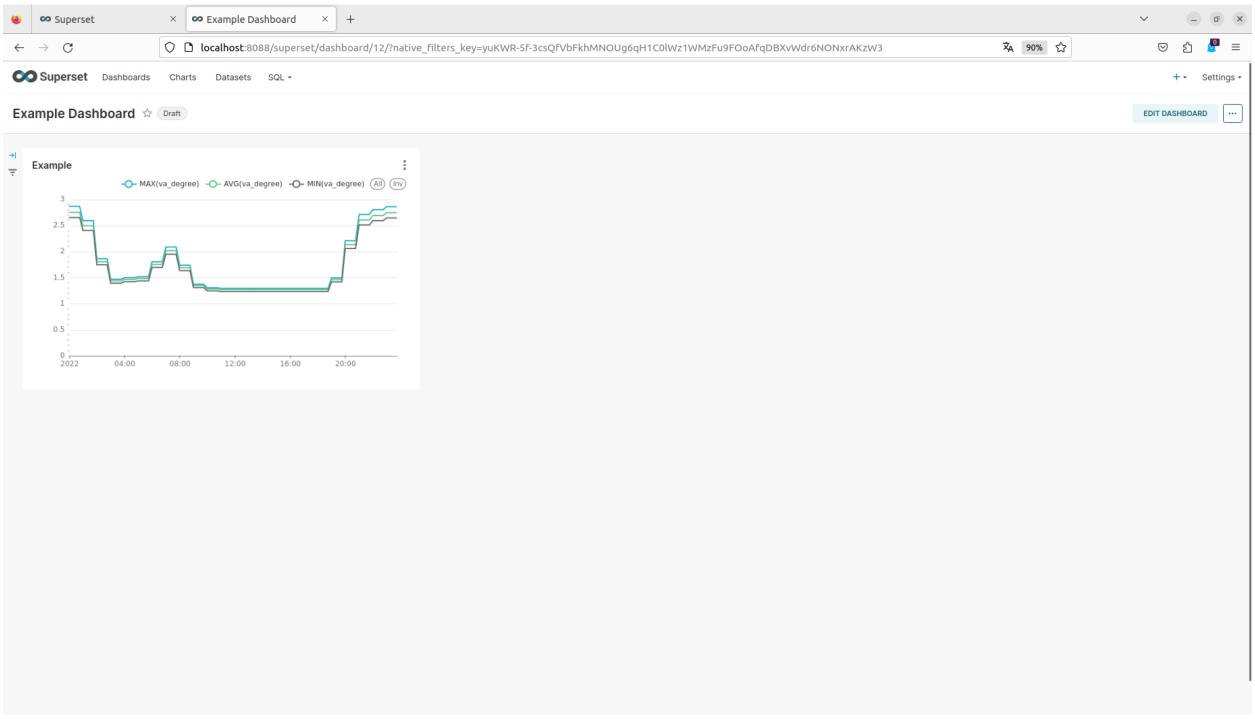


Fig. 17: This is the dashboard view.

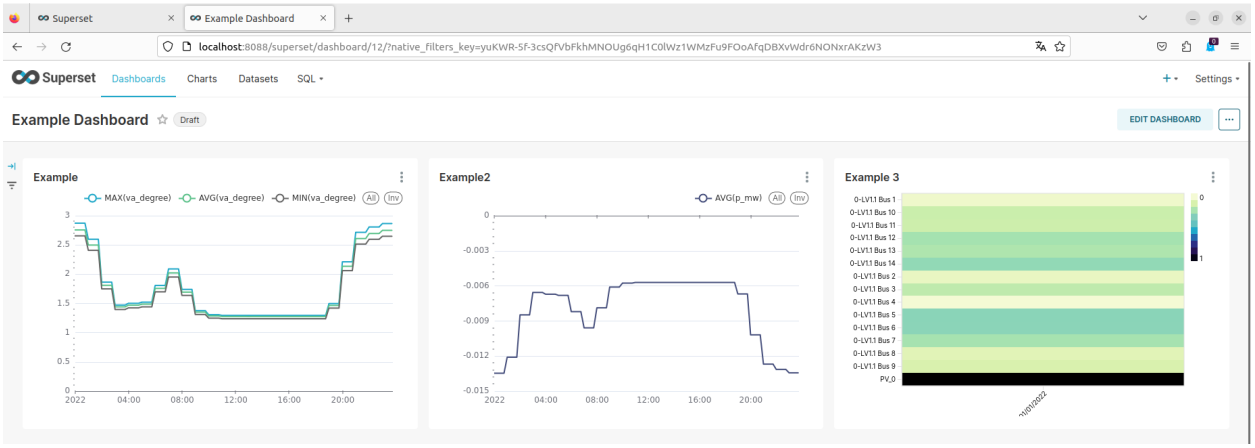


Fig. 18: This is the created example dashboard.

Fig. 19: This is the created example chart in fullscreen.

CHAPTER

SIX

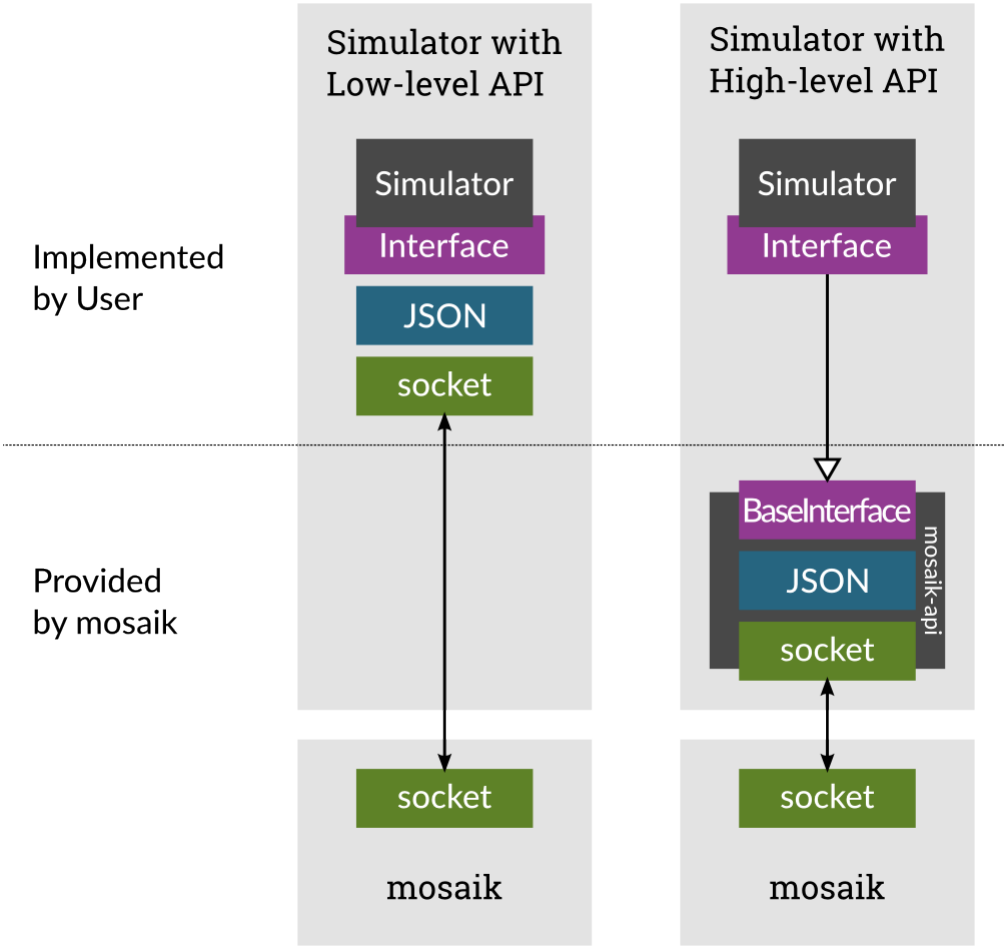
THE MOSAIK API

The mosaik API defines the communication protocol between mosaik and the simulators it couples. We differentiate between a *low-level* and a *high-level* version of the API.

The low-level API uses plain network sockets to exchange JSON encoded messages.

The high-level API is an implementation of the low-level API in a specific programming language. It encapsulates all parts related to networking (socket handling, an event loop, message (de)serialization) and provides an abstract base class with a few methods that have to be implemented in a subclass. A high-level API implementation is currently available for Python, Java and Julia. Implementations for other languages will be added when needed.

The figure below depicts the differences between the two API levels.



Contents:

6.1 How mosaik communicates with a simulator

This section provides a general overview which API calls exists and when mosaik calls them. The following sections will go into more detail.

When the connection between a simulator and mosaik is established, mosaik will first call `init()`, optionally passing some global parameters to the simulator. The simulators returns some meta data describing itself.

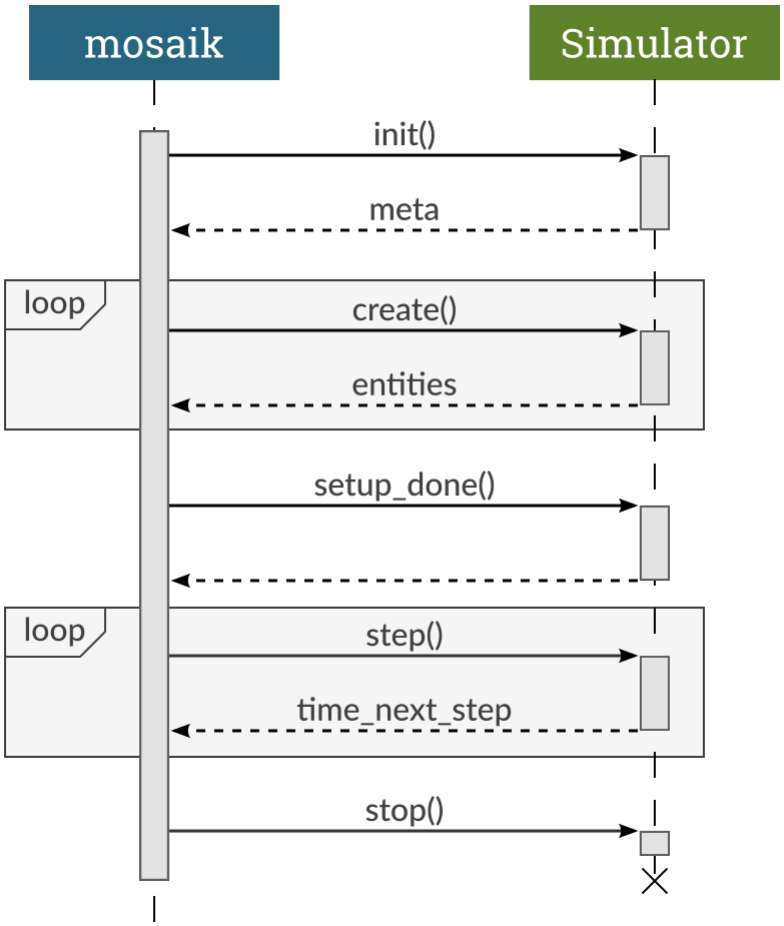
Following this, mosaik may call `create()` multiple times in order to instantiate one of the models that the simulator implements. The return value contains information describing the entities created.

The end of *create* phase and the beginning of the *step* (or simulation) phase is marked by a call to `setup_done()`. At this point, all entities are created and all relations between them are established.

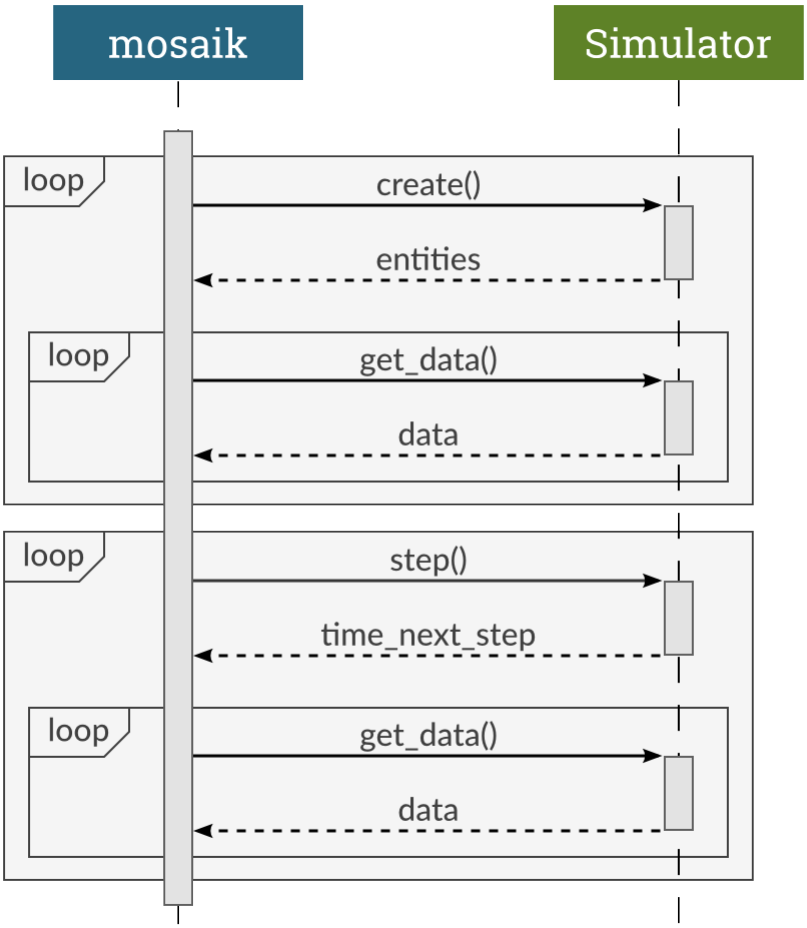
When the simulation has been started, mosaik repeatedly calls `step()`. This allows the simulator to step forward in time. It returns the time at which it wants to perform its next step.

Finally, mosaik sends a `stop()` message to every simulator to request its shut-down.

The following figure depicts the sequence of these messages:

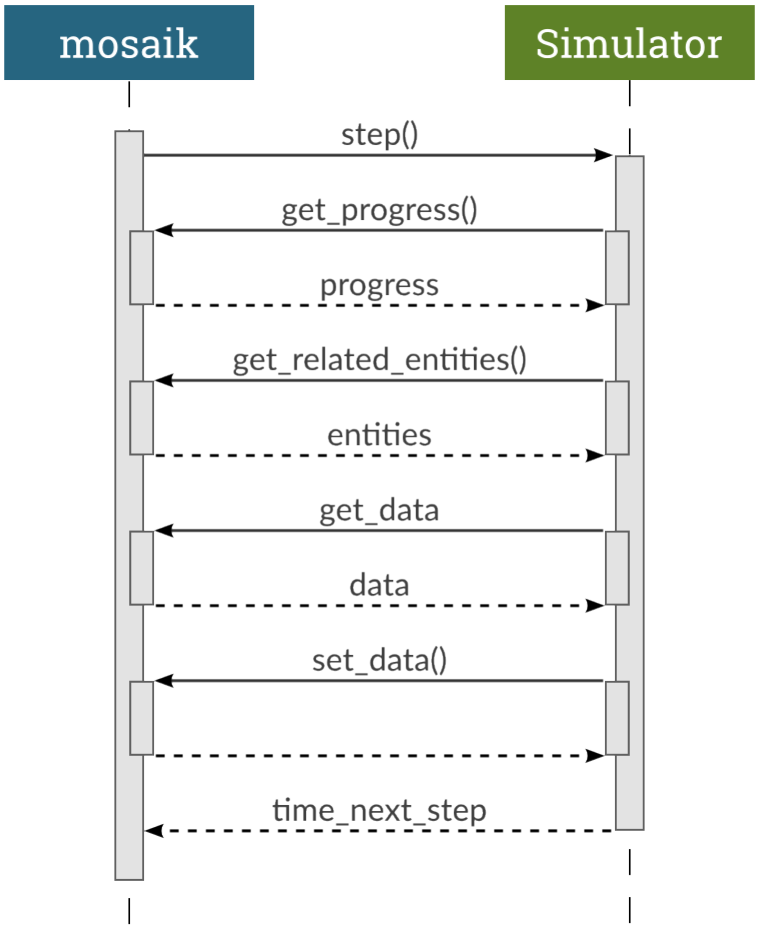


After `create()` or `step()` have been called, there may be an arbitrary amount of `get_data()` calls where mosaik requests the current values of some entities' attributes:



These methods are usually sufficient to connect simple simulators to mosaik. However, control strategies, visualizations or database adapters may need to actively query mosaik for additional data.

Thus, while a simulator is executing a simulation step, it may make asynchronous requests to mosaik. It can collect information about the simulated topology (`get_related_entities()`), request a new step for itself (`set_event()`), get the current simulation progress (`get_progress()`), query other entities for data (`get_data()`) and set data for other entities (`set_data()`).

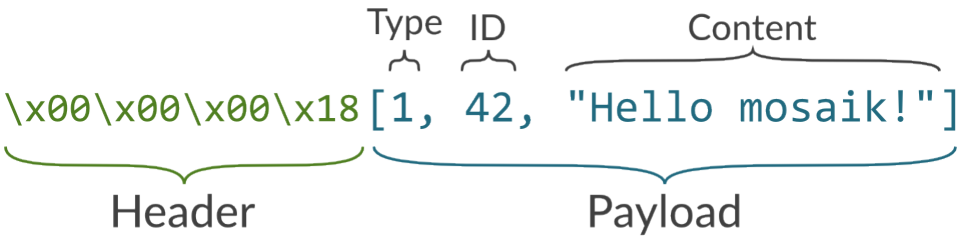


The next two sections explain the *low-level API* and the *Python high-level API* in more detail. The last section, *High-level APIs in other languages*, presents APIs available for other popular programming languages.

6.2 The low-level API

The low-level API uses standard TCP sockets. If mosaik starts a simulator, that simulator needs to connect to mosaik. If mosaik connects to a running instance of a simulator, that simulator obviously needs to provide a server socket that mosaik can connect to.

Network messages consists of a four bytes long *header* and a *payload* of arbitrary length. The header is an unsigned integer (uint32) in network byte order (big-endian) and stores the number of bytes in the payload. The payload itself is a UTF-8 encoded JSON list containing the message type, a message ID and the actual content:



Messages send between mosaik and a simulator must follow the request-reply pattern. That means, that every request that one party makes must be responded by the other party. Request use the message type `0`, replies uses `1` for success or `2` to indicate a failure. The message ID is an integer that is unique for every request that a network socket makes. Replies (no matter if successful or failed) need to use the message ID of the corresponding request.

The content of a request roughly map to a Python function call:

```
[function, [arg0, arg1, ...], {kwarg0: val0, kwar1: val1}]
```

Thereby, *function* is always a string. The type of the arguments and keyword arguments may vary depending on the function.

The content of replies is either the return value of the request, or an error message or stack trace. Error messages and stack traces should always be strings. The return value for successful requests depends on the function.

6.2.1 Example

We want to perform the following function call on the remote site: `my_func('hello', 'world', times=23) --> 'the return value'`. This would map to the following message payload:

```
[0, 1, ["my_func", ["hello", "world"], {"times": 23}]]
```

Our message is a request (message type `0`), the message ID is `1` and the content is a JSON list containing the function name as well as its arguments and keyword arguments.

The complete message sent via the network will be:

```
\x00\x00\x00\x36[0, 1, ["my_func", ["hello", "world"], {"times": 23}]]
```

In case of success, the reply's payload to this request could look like this:

```
[1, 1, "the return value"]
```

In case of error, this could be the reply's payload:

```
[2, 1, "Error in your code line 23: ..."]
```

The actual network messages would be:

```
\x00\x00\x00\x1a[1, 1, "the return value"]
\x00\x00\x00\x29[2, 1, "Error in your code line 23: ..."]
```

All commands that mosaik may send to a simulator are described in-depth in the *next section*. All asynchronous requests that a simulator may make are described in *Asynchronous requests*.

API calls:

- *init*

- *create*
- *setup_done*
- *step*
- *get_data*
- *stop*

Async. requests:

- *get_progress*
- *get_related_entities*
- *get_data*
- *set_data*
- *set_event*

6.2.2 API calls

This section describes the API calls `init()`, `create()`, `setup_done()`, `step()`, `get_data()` and `stop()`. In addition to these, a simulator may *optionally* expose additional functions (referred to as *extra methods*). These methods can be called at composition time (when you create your scenario).

init

```
["init", [sim_id], {time_resolution=time_resolution, **sim_params}] -> meta
```

The `init` call is made once to initialize the simulator. It has one positional argument, the simulator ID, and *time_resolution* and an arbitrary amount of further parameters (*sim_params*) as keyword arguments.

The return value *meta* is an object with meta data about the simulator:

```
{
    "api_version": "x.y",
    "type": "time-based"|"event-based"|"hybrid",
    "models": {
        "ModelName": {
            "public": true|false,
            "params": ["param_1", ...],
            "attrs": ["attr_1", ...],
            "any_inputs": true|false,
        },
        ...
    },
    "extra_methods": [
        "do_cool_stuff",
        "set_static_data"
    ]
}
```

The *api_version* is a string that defines which version of the mosaik API the simulator implements. Since mosaik API version 2.2, the simulator's major version ("x", in the snippet above) has to be equal to mosaik's. Mosaik will cancel the simulation if a version mismatch occurs.

The *type* defines how the simulator is stepped and for which time the output is valid. *Time-based* simulators only decide themselves on which points in time they want to be stepped (i.e. communicate with the other simulators). Their output is valid until the next step. *Event-based* simulators are always stepped when a predecessor provides new input, but they can also schedule steps for themselves. A more fine-grained behavior can be set for *hybrid* simulators. See the *scheduler description* for details.

models is an object describing the models provided by this simulator. The entry *public* determines whether a model can be instantiated by a user (`true`) or if it is a sub-model that cannot be created directly (`false`). *params* is a list of parameter names that can be passed to the model when creating it. *attrs* is a list of attribute names that can be accessed (reading or writing). If the optional *any_inputs* flag is set to `true`, any attributes can be connected to the model, even if they are not *attrs*. This may, for example, be useful for databases that don't know in advance which attributes of an entity they'll receive.

extra_methods is an optional list of methods that a simulator provides in addition to the standard API calls (`init()`, `create()` and so on). These methods can be called while the scenario is being created and can be used for operations that don't really belong into `init()` or `create()`.

Example

Request:

```
["init", ["PowerGridSim-0"], {"time_resolution": 1., "step_size": 60}]
```

Reply:

```
{
   "api_version": "3.0",
   "type": "time-based",
   "models": {
        "Grid": {
            "public": true,
            "params": ["topology_file"],
            "attrs": []
        },
        "Node": {
            "public": false,
            "params": [],
            "attrs": ["P", "Q"]
        },
        "Branch": {
            "public": false,
            "params": [],
            "attrs": ["I", "I_max"]
        }
    }
}
```

create

```
["create", [num, model], {**model_params}] -> entity_list
```

Create *num* instances of *model* using the provided *model_params*

num is an integer for the number of model instances to create.

model needs to be a public entry in the simulator's `meta['models']` (see *init*).

model_params is an object mapping parameters (from `meta['models'][model]['params']`, see *init*) to their values.

Return a (nested) list of objects describing the created model instances (entities). The root list must contain exactly *num* elements. The number of objects in sub-lists is not constrained:

```
[
      {
         "eid": "eid_1",
         "type": "model_name",
         "rel": ["eid_2", ...],
         "children": <entity_list>,
      },
      ...
]
```

The entity ID (*eid*) of an object must be unique within a simulator instance. For entities in the root list, *type* must be the same as the *model* parameter. The type for objects in sub-lists may be anything that can be found in `meta['models']` (see *init*). *rel* is an optional list of related entities; "related" means that two entities are somehow connect within the simulator, either logically or via a real *data-flow* (e.g., grid nodes are related to their adjacent branches). The *children* entry is optional and may contain a sub-list of entities.

Example

Request:

```
["create", [1, "Grid"], {"topology_file": "data/grid.json"}]
```

Reply:

```
[
    {
        "eid": "Grid_1",
        "type": "Grid",
        "rel": [],
        "children": [
            {
                "eid": "node_0",
                "type": "Node",
            },
            {
                "eid": "node_1",
                "type": "Node",
            },
            {
                "eid": "branch_0",
                "type": "Branch",
```

(continues on next page)

(continued from previous page)

```
                "rel": ["node_0", "node_1"]
            }
        ]
    }
]
```

setup_done

```
["setup_done", [], {}] -> null
```

Callback that indicates that the scenario setup is done and the actual simulation is about to start.

At this point, all entities and all connections between them are know but no simulator has been stepped yet.

Implementing this method is optional.

Added in mosaik API version 2.2.

Example

Request:

```
["setup_done", [], {}]
```

Reply:

```
null
```

step

```
["step", [time, inputs, max_advance], {}] -> Optional[time_next_step]
```

Perform the next simulation step at time *time* using input values from *inputs* and return the new simulation time (the time at which *step* should be called again) or null if the simulator doesn't need to step itself.

time, *max_advance*, and the time_next_step are integers (or null). Their unit is arbitrary, e.g. *seconds* (counted from simulation start), but has to be consistent among all simulators used in a scenario.

inputs is a dict of dicts mapping entity IDs to attributes and dicts of values (each simulator has to decide on its own how to reduce the values (e.g., as its sum, average or maximum):

```
{
    "eid_1": {
        "attr_1": {'src_full_id_1': val_1_1, 'src_full_id_2': val_1_2, ...},
        "attr_2": {'src_full_id_1': val_2_1, 'src_full_id_2': val_2_2, ...},
        ...
    },
    ...
}
```

max_advance tells the simulator how far it can advance its time without risking any causality error, i.e. it is guaranteed that no external step will be triggered before max_advance + 1, unless the simulator activates an output loop earlier than that. For time-based simulators (or hybrid ones without any triggering input) *max_advance* is always equal to the end of the simulation (*until*). See the description of the *scheduler* for more details.

Example

Request:

```
[
    "step",
    [
        60,
        {
              "node_1": {"P": [20, 3.14], "Q": [3, -2.5]},
              "node_2": {"P": [42], "Q": [-23.2]},
        },
        3600
    ],
    {}
]
```

Reply:

```
120
```

get_data

```
["get_data", [outputs], {}] -> data
```

Return the data for the requested attributes in *outputs*

outputs is an object mapping entity IDs to lists of attribute names whose values are requested (connected to any other simulator):

```
{
    "eid_1": ["attr_1", "attr_2", ...],
    ...
}
```

The return value needs to be an object of objects mapping entity IDs and attribute names to their values:

```
{
    "eid_1: {
       "attr_1": "val_1",
       "attr_2": "val_2",
       ...
    },
    ...
    "time": *output_time* (optional)
}
```

Time-based simulators have set an entry for all requested attributes, whereas for event-based and hybrid simulators this is optional (e.g. if there's no new event).

Event-based and hybrid simulators can optionally set a timing of their non-persistent output attributes via a *time* entry, which is valid for all given (non-persistent) attributes. If not given, it defaults to the current time of the step. Thus only one output time is possible per step. For further output times the simulator has to schedule another self-step (via the step's return value).

Examples

Request:

```
["get_data", [{"branch_0": ["I"]}], {}]
```

Reply:

```
{
    "branch_0": {
        "I": 42.5
    }
}
```

Request:

```
["get_data", [{"node_0": ["msg"], "node_1": ["msg"]}], {}]
```

Reply:

```
{
    "node_1": {
        "msg": "Hi"
    },
    "time": 140
}
```

stop

```
["stop", [], {}] -> null
```

Immediately stop the simulation and terminate.

This call has no parameters and no reply is required.

Example

Request:

```
["stop", [], {}]
```

Reply:

no reply required

6.2.3 Asynchronous requests

get_progress

```
["get_progress", [], {}] -> progress
```

Return the current overall simulation progress in percent.

Example

Request:

```
["get_progress", [], {}]
```

Reply:

```
23.42
```

get_related_entities

```
["get_related_entities", [entities], {}] -> related_entities
```

Return information about the related entities of *entities*.

If *entitites* omitted (or `null`), return the complete entity graph, e.g.:

```
{
    "nodes": {
        "sid_0.eid_0": {"type": "A"},
        "sid_0.eid_1": {"type": "B"},
        "sid_1.eid_0": {"type": "C"},
    },
    "edges": [
        ["sid_0.eid_0", "sid_1.eid0", {}],
        ["sid_0.eid_1", "sid_1.eid0", {}],
    ],
}
```

If *entities* is a single string (e.g., `sid_1.eid_0`), return an object containing all entities related to that entity:

```
{
    "sid_0.eid_0": {"type": "A"},
    "sid_0.eid_1": {"type": "B"},
}
```

If *entities* is a list of entity IDs (e.g., `["sid_0.eid_0", "sid_0.eid_1"]`), return an object mapping each entity to an object of related entities:

```
{
    "sid_0.eid_0": {
        "sid_1.eid_0": {"type": "B"},
    },
    "sid_0.eid_1": {
```

(continues on next page)

(continued from previous page)

```
        "sid_1.eid_1": {"type": "B"},
    },
}
```

Example

Request:

```
["get_related_entities", [["grid_sim_0.node_0", "grid_sim_0.node_1"]] {}]
```

Reply:

```
{
    "grid_sim_0.node_0": {
        "grid_sim_0.branch_0": {"type": "Branch"},
        "pv_sim_0.pv_0": {"type": "PV"}
    },
    "grid_sim_0.node_1": {
        "grid_sim_0.branch_0": {"type": "Branch"}
    }
}
```

get_data

```
["get_data", [attrs], {}] -> data
```

Get the data for the requested attributes in *attrs*.

attrs is an object of (fully qualified) entity IDs mapping to lists of attribute names:

```
{
    "sim_id.eid_1": ["attr_1", "attr_2", ...],
    ...
}
```

The return value is an object mapping the input entity IDs to data objects mapping attribute names to there respective values:

```
{
    "sim_id.eid_1: {
        "attr_1": "val_1",
        "attr_2": "val_2",
        ...
    },
     ...
}
```

Example

Request:

```
["get_data", [{"grid_sim_0.branch_0": ["I"]}], {}]
```

Reply:

```
{
    "grid_sim_0.branch_0": {
        "I": 42.5
    }
}
```

set_data

```
["set_data", [data], {}] -> null
```

Set *data* as input data for all affected simulators.

data is an object mapping source entity IDs to objects which in turn map destination entity IDs to objects of attributes and values (`{"src_full_id": {"dest_full_id": {"attr1": "val1", "attr2": "val2"}}}`)

Example

Request:

```
[
    "set_data",
    [{
        "mas_0.agent_0": {"pvsim_0.pv_0": {"P_target": 20,
                                           "Q_target": 3.14}},
        "mas_0.agent_1": {"pvsim_0.pv_1": {"P_target": 21,
                                           "Q_target": 2.718}}
    }],
    {}
]
```

Reply:

```
null
```

set_event

```
["set_event", [time], {}] -> null
```

Request an event/step for itself at time *time*.

Example

Request:

```
["set_event", [5], {}]
```

Reply:

```
null
```

6.3 The high-level API

Currently, there are high-level APIs for Python, Java, C#, and MatLab.

6.3.1 Installation

The Python implementation of the mosaik API is available as a separate package an can easily be installed via pip:

```
pip install mosaik-api-v3
```

It supports Python 2.7, >= 3.3 and PyPy.

6.3.2 Usage

You create a subclass of *mosaik_api_v3.Simulator* which implements the four API calls *init*, *create*, *step* and *get_data*. You can optionally override *configure* and *finalize*. The former allows you to handle additional command line arguments that your simulator may need. The latter is called just before the simulator terminates and allows you to perform some clean-up.

You then call *mosaik_api_v3.start_simulation* from your *main()* function to get everything set-up and running. That function handles the networking as well as serialization and de-serialization of messages. Commands from the low-level API are translated to simple function calls. The return value of these functions is used for the reply.

For example, the message

```
["create", [2, "Model", {"param1": 15, "param2": "spam"}]
```

will result in a call

```
create(2, 'Model', param1=15, param2='spam')
```

API calls

class mosaik_api_v3.**Simulator**(*meta*)

This is the base class that you need to inherit from and implement the API calls.

Parameters

meta (*Meta*) –

meta

Meta data describing the simulator (the same that is returned by *init*).

```
{
    'api_version': 'x.y',
    'type': 'time-based'|'event-based'|'hybrid',
    'models': {
        'ModelName': {
            'public': True|False,
            'params': ['param_1', ...],
            'attrs': ['attr_1', ...],
            'any_inputs': True|False,
            'trigger': ['attr_1', ...],
            'non-persistent': ['attr_2', ...],
        },
        ...
    },
    'extra_methods': [
        'do_cool_stuff',
        'set_static_data'
    ]
}
```

The *api_version* is a string that defines which version of the mosaik API the simulator implements. Since mosaik API version 2.3, the simulator's major version ("x", in the snippet above) has to be equal to mosaik's. Mosaik will cancel the simulation if a version mismatch occurs.

The *type* defines how the simulator is advanced through time and whether its attributes are persistent in time or transient.

models is a dictionary describing the models provided by this simulator. The entry *public* determines whether a model can be instantiated by a user (`True`) or if it is a sub-model that cannot be created directly (`False`). *params* is a list of parameter names that can be passed to the model when creating it. *attrs* is a list of attribute names that can be accessed (reading or writing). If the optional *any_inputs* flag is set to `true`, any attributes can be connected to the model, even if they are not *attrs*. This may, for example, be useful for databases that don't know in advance which attributes of an entity they'll receive. *trigger* is a list of attribute names that cause the simulator to be stepped when another simulator provides output which is connected to one of those.

extra_methods is an optional list of methods that a simulator provides in addition to the standard API calls (`init()`, `create()` and so on). These methods can be called while the scenario is being created and can be used for operations that don't really belong into `init()` or `create()`.

mosaik

An RPC proxy to mosaik.

init(*sid*, *time_resolution=1.0*, ***sim_params*)

Initialize the simulator with the ID *sid* and pass the
time_resolution and additional parameters *(sim_params)* sent by mosaik. Return the meta data *meta*.

If your simulator has no *sim_params*, you don't need to override this method.

Parameters

- **sid** (*str*) –
- **time_resolution** (*float*) –

Return type
Meta

create(*num*, *model*, ***model_params*)

Create *num* instances of *model* using the provided *model_params*.

num is an integer for the number of model instances to create.

model needs to be a public entry in the simulator's `meta['models']`.

model_params is a dictionary mapping parameters (from `meta['models'][model]['params']`) to their values.

Return a (nested) list of dictionaries describing the created model instances (entities). The root list must contain exactly *num* elements. The number of objects in sub-lists is not constrained:

```
[
    {
        'eid': 'eid_1',
        'type': 'model_name',
        'rel': ['eid_2', ...],
        'children': [
            {'eid': 'child_1', 'type': 'child'},
            ...
        ],
    },
    ...
]
```

The entity ID (*eid*) of an object must be unique within a simulator instance. For entities in the root list, *type* must be the same as the *model* parameter. The type for objects in sub-lists may be anything that can be found in `meta['models']`. *rel* is an optional list of related entities; "related" means that two entities are somehow connect within the simulator, either logically or via a real data-flow (e.g., grid nodes are related to their adjacent branches). The *children* entry is optional and may contain a sub-list of entities.

Parameters

- **num** (*int*) –
- **model** (*str*) –

Return type
List[*CreateResult*]

setup_done()

Callback that indicates that the scenario setup is done and the actual simulation is about to start.

At this point, all entities and all connections between them are know but no simulator has been stepped yet.

Implementing this method is optional.

Added in mosaik API version 2.3

Return type
None

step(*time*, *inputs*, *max_advance*)

Perform the next simulation step from time *time* using input values from *inputs* and return the new simulation time (the time at which `step()` should be called again).

time and the time returned are integers. Their unit is arbitrary, e.g. *seconds* (from simulation start), but has to be consistent among all simulators used in a simulation.

inputs is a dict of dicts mapping entity IDs to attributes and dicts of values (each simulator has to decide on its own how to reduce the values (e.g., as its sum, average or maximum):

```
{
    'dest_eid': {
        'attr': {'src_fullid': val, ...},
        ...
    },
    ...
}
```

max_advance tells the simulator how far it can advance its time without risking any causality error, i.e. it is guaranteed that no external step will be triggered before max_advance + 1, unless the simulator activates an output loop earlier than that. For time-based simulators (or hybrid ones without any triggering input) *max_advance* is always equal to the end of the simulation (*until*).

Parameters

- **time** (*int*) –
- **inputs** (*mosaik_api_v3.types.InputData*) –
- **max_advance** (*int*) –

Return type
int | None

get_data(*outputs*)

Return the data for the requested attributes in *outputs*

outputs is a dict mapping entity IDs to lists of attribute names whose values are requested:

```
{
    'eid_1': ['attr_1', 'attr_2', ...],
    ...
}
```

The return value needs to be a dict of dicts mapping entity IDs and attribute names to their values:

```
{
    'eid_1: {
        'attr_1': 'val_1',
        'attr_2': 'val_2',
        ...
    },
    ...
    'time': output_time (for event-based sims, optional)
}
```

Time-based simulators have set an entry for all requested attributes, whereas for event-based and hybrid simulators this is optional (e.g. if there's no new event). Event-based and hybrid simulators can optionally set a timing of their non-persistent output attributes via a *time* entry, which is valid for all given (non-persistent) attributes. If not given, it defaults to the current time of the step. Thus only one output time is possible per step. For further output times the simulator has to schedule another self-step (via the step's return value).

Parameters
outputs (*Dict[str, List[str]]*) –

Return type
Dict[str, *Dict*[str, *Any*]]

configure(*args*)

This method can be overridden to configure the simulation with the command line *args* as created by docopt.

backend and *env* are the *simpy.io* backend and environment used for networking. You can use them to start extra processes (e.g., a web server).

The default implementation simply ignores them.

finalize()

This method can be overridden to do some clean-up operations after the simulation finished (e.g., shutting down external processes).

The *mosaik-api-v3* package provides an example simulator that demonstrates how the API can be implemented.

Asynchronous requests

The *asynchronous requests* can be called via the `MosaikRemote` proxy `self.mosaik` from within *step*, except for `set_data()` which has to be called from another thread/process (see below). They don't return the actual results but an *event* (similar to a *future* of *deferred*). The event will eventually hold the actual result. To wait for that result to arrive, you simply yield the event, e.g.:

```
def step(self, time, inputs, max_advance):
    progress = yield self.mosaik.get_progress()
    # ...
```

async MosaikRemote.**get_progress**()

Return the current simulation progress from *sim_progress*.

Return type
float

async MosaikRemote.**get_related_entities**(*entities=None*)

Return information about the related entities of *entities*.

If *entities* omitted (or `None`), return the complete entity graph, e.g.:

```
{
    'nodes': {
        'sid_0.eid_0': {'type': 'A'},
        'sid_0.eid_1': {'type': 'B'},
        'sid_1.eid_0': {'type': 'C'},
    },
    'edges': [
        ['sid_0.eid_0', 'sid_1.eid0', {}],
        ['sid_0.eid_1', 'sid_1.eid0', {}],
    ],
}
```

If *entities* is a single string (e.g., `sid_1.eid_0`), return a dict containing all entities related to that entity:

```
{
    'sid_0.eid_0': {'type': 'A'},
    'sid_0.eid_1': {'type': 'B'},
}
```

If *entities* is a list of entity IDs (e.g., `['sid_0.eid_0', 'sid_0.eid_1']`), return a dict mapping each entity to a dict of related entities:

```
{
    'sid_0.eid_0': {
        'sid_1.eid_0': {'type': 'B'},
    },
    'sid_0.eid_1': {
        'sid_1.eid_1': {'type': 'B'},
    },
}
```

Parameters
entities (*str | List[str] | None*) –

Return type
Dict[str, *Any*] | *Dict*[str, *Dict*[str, *Any*]]

async MosaikRemote.**get_data**(*attrs*)

Return the data for the requested attributes *attrs*.

attrs is a dict of (fully qualified) entity IDs mapping to lists of attribute names (`{'sid/eid': ['attr1', 'attr2']}`).

The return value is a dictionary, which maps the input entity IDs to data dictionaries, which in turn map attribute names to their respective values: (`{'sid/eid': {'attr1': val1, 'attr2': val2}}`).

Parameters
attrs (*Dict[str, List[str]]*) –

Return type
Dict[str, *Any*]

async MosaikRemote.**set_data**(*data*)

Set *data* as input data for all affected simulators.

data is a dictionary mapping source entity IDs to destination entity IDs with dictionaries of attributes and values (`{'src_full_id': {'dest_full_id': {'attr1': 'val1', 'attr2': 'val2'}}}`).

async MosaikRemote.**set_event**(*event_time*)

Schedules an event/step at simulation time *event_time*.

Parameters
event_time (*int*) –

The *mosaik-api-v3* package provides an example "multi-agent system" that demonstrates how asynchronous requests can be implemented.

Starting the simulator

To start your simulator, you just need to create an instance of your *Simulator* sub-class and pass it to *start_simulation*:

mosaik_api_v3.**start_simulation**(*simulator*, *description=''*, *extra_options=None*)

Start the simulation process for `simulation`.

simulation is the instance of your API implementation (see *Simulator*).

description may override the default description printed with the help on the command line.

extra_option may be a list of options for docopt (example: `['-e, --example Enable example mode']`). Commandline arguments are passed to *Simulator.configure* so that your API implementation can handle them.

Here is an example with a bit more context:

```
import mosaik_api_v3


example_meta = {
    'type': 'time-based',
    'models' {
        'A': {
              'public': True,
              'params': ['init_val'],
              'attrs': ['val_out', 'dummy_out'],
        },
    }
}


class ExampleSim(mosaik_api_v3.Simulator):
    def __init__(self):
        super().__init__(example_meta)

    sim_name = 'ExampleSimulation'

    def configure(self, args, backend, env):
        # Here you could handle additional command line arguments

    def init(self, sid):
        # Initialize the simulator
        return self.meta

    # Implement the remaining methods (create, step, get_data, ...)


def main():
    import sys

    description = 'A simple example simulation for mosaik.'
    extra_options = [
       '--foo       Enable foo',
       '--bar BAR   The bar parameter',
    ]

    return mosaik_api_v3.start_simulation(ExampleSim(), description, extra_options)


if __name__ == '__main__':
    sys.exit(main())
```

6.4 High-level APIs in other languages

Mosaik provides high-level APIs not only for simulators written in Python, but also in various other languages. Currently, Java and Julia are supported.

6.4.1 Java API

For the use of the Java API please refer to the *Java tutorial*.

Note: There is also an extended version for Java at *Java tutorial* that incorporates Quality of Life changes for a more Java-like experience.

6.4.2 Julia API

For the use of the Julia API, please refer to the Julia mosaik API docs.

CHAPTER

SEVEN

SCENARIO DEFINITION

Modeling or composing a scenario in mosaik comprises three steps:

1. starting simulators,
2. instantiating models within the simulators, and
3. connecting the model instances of different simulators to establish the data flow between them.

This page will show you how to create simple scenarios in these three steps. It will also provide some recipes that allow you to create more complex scenarios.

7.1 The setup

The central class for creating scenarios is *mosaik.scenario.World* (for your convenience, you can also import `World` directly from `mosaik`). This class stores all data and state that belongs to your scenario and its simulation. It also provides various methods that allow you to start simulators and establish the data flows between them.

In this tutorial, we'll create a very simple scenario using the example simulation that is provided with the Python implementation of the simulator API.

We start by importing the *mosaik* package and creating a *World* instance:

```
>>> import mosaik
>>>
>>> sim_config: mosaik.SimConfig = {
...     'ExampleSim': {'python': 'example_sim.mosaik:ExampleSim'},
... }
>>>
>>> world = mosaik.World(sim_config)
```

(You can leave off the type annotation on `sim_config` if you're not using type checking.)

As we start simulator instances by using `world`, it needs to know what simulators are available and how to start them. This is called the *sim config* and is a dict that contains every simulator we want to use together with some information on how to start it.

In our case, the only simulator is the *ExampleSim*. It will be started by importing the module `example_sim.mosaik` and instantiating the class `ExampleSim`. This is only possible with simulators written in Python 3. You can also let mosaik start simulator as external processes or let it connect to already running processes. The *simulator manager docs* explain how this all works and give you some hints when to use which method of starting a simulator.

In addition to the sim config you can optionally pass the `mosaik_config` dictionary to *World* in order to overwrite some general parameters for mosaik (e.g., the host and port number for its network socket or timeouts). Usually, the defaults work just well.

```
>>> world = mosaik.World(sim_config, mosaik_config={'addr': ('127.0.0.1', 5555),
↪'start_timeout': 10, 'stop_timeout': 10,})
```

Via the `time_resolution` parameter you can set a global time resolution for the scenario, which will be passed to each simulator as keyword argument via the init function (see *API init*). It tells each simulator how to translate mosaik's integer time to simulated time (in seconds from simulation start). It has to be a float and it defaults to `1.`.

```
>>> world = mosaik.World(sim_config, time_resolution=1.)
```

If you set the `debug` flag to `True` an execution graph will be created during the simulation. This may be useful for debugging and testing. Note that this increases the memory consumption and simulation time.

```
>>> world = mosaik.World(sim_config, debug=False)
```

There are two more technical parameters: You can set the `cache` flag to `False` if the average step size of the simulators is orders of magnitudes larger than the time resolution, i.e. a time resolution of microseconds where the typical step size is in the seconds range. This will considerably reduce the simulation time.

```
>>> world = mosaik.World(sim_config, cache=True)
```

Via `max_loop_iterations` you can limit the maximum iteration count within one time step for *same-time loops*. It's default value is 100.

```
>>> world = mosaik.World(sim_config, max_loop_iterations=100)
```

7.2 Starting simulators

Now that the basic set-up is done, we can start our simulators:

```
>>> simulator_0 = world.start('ExampleSim', step_size=2)
Starting "ExampleSim" as "ExampleSim-0" ...
>>> simulator_1 = world.start('ExampleSim')
Starting "ExampleSim" as "ExampleSim-1" ...
```

To start a simulator, we call *`World.start`* and pass the name of the simulator. Mosaik looks up that name in its sim config, starts the simulator for us and returns a *`ModelFactory`*. This factory allows us to instantiate simulation models within that simulator.

In addition to the simulator name, you can pass further parameters for the simulators. These parameters are passed to the simulator via the *init API call*.

7.3 Instantiating simulation models

Simulators specify a set of public models in their meta data (see *init API call*). These models can be accessed with the *`ModelFactory`* that *`World.start`* returns as if they were normal Python classes. So to create one instance of *ExampleSim*'s model *A* we just write:

```
>>> a = simulator_0.A(init_val=0)
```

This will create one instance of the *A* simulation model and pass the model parameter `init_val=0` to it (see *create API call*). Lets see what it is that gets returned to us:

```
>>> print(a)
Entity('ExampleSim-0', '0.0', 'ExampleSim', A)
>>> a.sid, a.eid, a.full_id
('ExampleSim-0', '0.0', 'ExampleSim-0.0.0')
>>> a.sim_name, a.type
('ExampleSim', 'A')
>>> a.children
[]
```

A model instance is represented in your scenario as an *Entity*. The entity belongs to the simulator *ExampleSim-0*, has the ID *0.0* and its type is *A*. The entity ID is unique within a simulator. To make it globally unique, we prepend it with the simulator ID. This is called the entity's *full ID* (see *Entity.full_id*). You can also get a list of its child entities (which is empty in this case).

In order to instantiate multiple instances of a model, you can either use a simple list comprehension (or `for` loop) or call the static method *create* of the model:

```
>>> a_set = [simulator_0.A(init_val=i) for i in range(2)]
>>> b_set = simulator_1.B.create(3, init_val=1)
```

The list comprehension is more verbose but allows you to pass individual parameter values to each instance. Using `create` is more concise but all three instance will have the same value for `init_val`. In both cases you'll get a list of entities (aka *entity sets*).

7.4 Setting initial events

Time-based and hybrid simulators are automatically scheduled for time step 0, and will organize their scheduling until the simulation's end themselves afterward. For event-based simulators this is not the case, as they might only want to be stepped if an event is created by another simulator for example. Therefore you might need to set initial events for some event-based ones via *World.set_initial_event*, which sets an event for time 0 by default, or at later times if explicitly stated:

```
>>> world.set_initial_event(a.sid)
>>> world.set_initial_event(b.sid, time=3)
```

7.5 Connecting entities

If we would now run our simulation, both, *ExampleSim-0* and *Example-Sim-1* would run in parallel and never exchange any data. To change that, we need to connect the models providing input data to entities requiring this data. In our case, we will connect the *val_out* attribute of the *A* instances with the *val_in* attribute of the *B* instances:

```
>>> a_set.insert(0, a)  # Put our first A instance to the others
>>> for a, b in zip(a_set, b_set):
...     world.connect(a, b, ('val_out', 'val_in'))
```

The method *World.connect* takes the source entity, the destination entity and an arbitrary amount of *(source attribute, dest. attribute)* tuples. If the name of the source attributes equals that of the destination attribute, you can alternatively just pass a single string (e.g., `connect(a, b, 'attr')` is the same as `connect(a, b, ('attr', 'attr'))`).

mosaik deals with two separate types of data exchange between simulators:

- First, there are measurements that have a value at each point of time. Examples include all kinds of physical measurements like the voltage at a grid node or the power output of a PV system.

- Second, there are events (those were introduced in mosaik 3) which happen at a particular point in time. A typical example is a message between ICT devices or a set-point message from some controller to the PV system that it controls.

Each attribute of a mosaik simulator can deal either with measurements or with events. In case of time-based simulators, all attributes work as measurements, in case of event-based simulators, all attributes work as events. Hybrid-simulators can work with either on an attribute-by-attribute basis (i.e. each attribute is either for measurements or for events). For historic reasons, input attributes are called trigger attributes when they deal with events, and output attributes are called non-persistent when they deal with events. The opposite terms 'non-trigger' and 'persistent' are used for measurement attributes (input and output, respectively).

mosaik will complain if you connect a non-persistent output to a non-trigger input. This is because the target simulator should be able to rely on always receiving input on its non-trigger attributes, but the source simulator could potentially omit output on its non-persistent attribute. (mosaik could provide output that the source simulator produced in earlier steps, delaying or even repeating it. However, this would be semantically unsound as the event is associated with the time at which it was generated by the source simulator, not with the time at which the target simulator happens to run.)

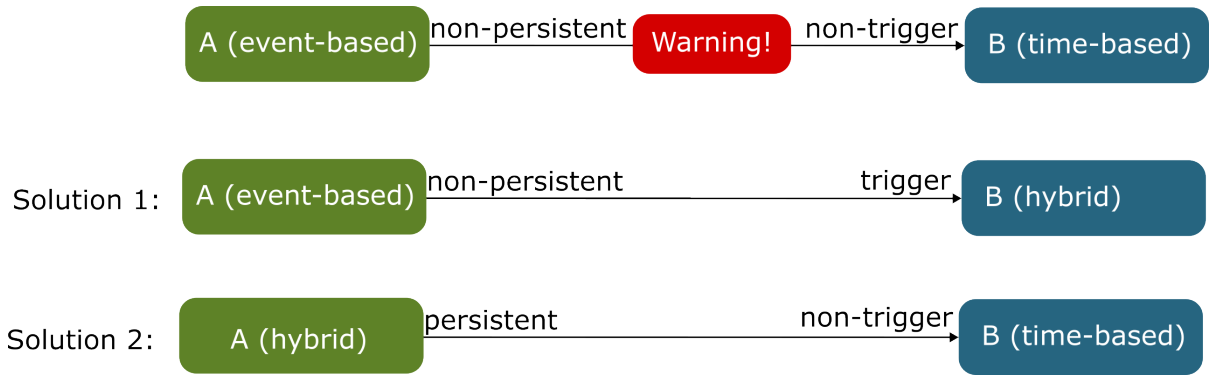


Fig. 1: A non-persistent output connected to a non-trigger input leads to a warning.

Usually, the solution to resolve this warning is to change the type of one of the affected attributes: the source attribute from non-persistent to persistent, or the target attribute from non-trigger to trigger. You can do this without affecting the simulator's other attributes by changing the simulator's type to hybrid, where you can then specify which attributes should be trigger and/or non-persistent. *(See here for the format of META.)* Note that the attributes of hybrid simulator behave like measurements by default, so if you are changing an event-based simulator to hybrid, you will have to specify all attributes except for the affected one to be trigger and/or non-persistent if you want to preserve their previous behavior.

We also encourage you to carefully think about the opposite case where you attempt to connect a persistent output to a trigger input, as wanting to do this often indicates ambiguity about what your data represents. However, because there is the common case of saving data generated in the simulation using some database or writer simulator (regardless of how whether it is an event or a measurement), mosaik will not complain when you set up connections like this.

You are also not allowed to create circular dependencies via standard connections only (e.g., connect *a* to *b* and then connect *b* to *a*). There are several ways to allow a bidirectional or cyclic exchange of data, which is required for things like control strategies, e.g. via *time-shifted* or *weak* connections. See section *How to achieve cyclic data-flows* for details.

7.6 Running the simulation

When all simulators are started, models are instantiated and connected, we can finally run our simulation:

```
>>> world.run(until=10)
Starting simulation.
Simulation finished successfully.
```

This will execute the simulation from time 0 until we reach the time *until* (in simulated time units). The *scheduler section* explains in detail what happens when you call `run`.

While the simulation is running, the current progress is visualized using a tqdm progress bar. You can turn this off using the `print_progress` parameter of `world.run`:

```
world.run(until=10, print_progress=False)
```

If you want a more detailed progress report, you can set `print_progress='individual'` which will produce a separate progress bar for each simulator in your simulation.

We can also set the *lazy_stepping* flag (default: `True`). If `True`, a simulator can only run ahead one step of its successors. If `False`, a simulator always steps as long as all inputs are provided. This might decrease the simulation time but increase the memory consumption.

```
>>> world.run(until=END, lazy_stepping=False)
```

To wrap it all up, this is how our small example scenario finally looks like:

```
# Setup
import mosaik

sim_config = {
    'ExampleSim': {'python': 'example_sim.mosaik:ExampleSim'},
}

world = mosaik.World(sim_config)

# Start simulators
simulator_0 = world.start('ExampleSim', step_size=2)
simulator_1 = world.start('ExampleSim')

# Instantiate models
a_set = [simulator_0.A(init_val=i) for i in range(3)]
b_set = simulator_1.B.create(3, init_val=1)

# Connect entities
for a, b in zip(a_set, b_set):
    world.connect(a, b, ('val_out', 'val_in'))

# Run simulation
world.run(until=10)
```

7.7 How to achieve cyclic data-flows

The guiding principle behind mosaik's scheduler is that simulator steps that happen at the same (mosaik) time are handled in data-dependency order. In other words, if simulator *A* is connected to simulator *B* (with data flowing from *A* to *B*) and both simulators are scheduled to step at time *t*, simulator *A* will run first so that simulator *B* can use the most up-to-date outputs from *A* in its calculation.

This leads to problems when naively setting up bi-directional or cyclic data flow, like this

```
# Send battery's active power value to the controller
world.connect(battery, controller, 'P')
# Controller sends back a schedule to the battery
world.connect(controller, battery, 'schedule')
```

The problem is that mosaik cannot step either of the involved simulators as both are waiting for input from the other one. mosaik will recognize cycles like this and raise an error when you attempt to `run` a simulation containing them. To avoid this error, the standard connections in your scenario must form an acyclic (directed) graph (on the level of simulators).

Of course, cyclic data flow is common in co-simulations and mosaik offers two options for this: time-shifted connections and weak connections. (There are also asynchronous requests, which are deprecated.)

7.7.1 Time-shifted connections

You can resolve a cycle by marking (at least) one of its constituent connections as *time shifted*, like so:

```
world.connect(battery, controller, 'P')
world.connect(controller, battery, 'schedule', time_shifted=True,
              initial_data={'schedule': initial_schedule})
```

This indicates to mosaik that the output of the source simulator should only be passed to the target simulator at the next time step. Now the loop is resolved, as the non-time-shifted connections form an acyclic graph again. (So in the example, `battery` will run before `controller` in each step.)

For the first step, the source simulator in a time-shifted connection will not have run, yet. However, the target simulator might still require input on the connected attribute. Therefore, you must provide `initial_data`, which is a dictionary of the attributes of the source simulator and the values they should have for the first step.

In the example above, the result would be a sequential execution of the two simulators. You can also set the `time_shifted` flag for both connections, in which case you would get parallel execution of both simulators, with each simulators using the output of the other simulator from the previous step.

Finally, you can set `time-shifted` to an integer value instead of `True`. This will delay data along that connection by that many steps.

7.7.2 Weak connections

Occasionally, you want parts of your simulation to run on a much finer time scale than the rest, without committing to a precise factor between the two. This is often the case for control strategies that are distributed over two or simulators.

In this case, you can use *weak* connection. A weak connection resolves cycles within one time step in much the same way as a time-shifted connection. However, if the source simulator produces output, this will trigger another step of the target simulator within the same time step. This might in turn trigger additional of the source simulator again, leading to a so-called *same-time loop* which runs until one of the simulators interrupts it by not producing output. (Weak connections therefore only make sense if the target attribute is a trigger attribute. If it is not, the effect will be identical to a time-shifted connection.)

In most scenarios with same-time loops only some of the simulators will be involved in the loop. This poses the question when the other simulators are run in relation to them. In the past, this depended in subtle ways on the order in which connections were made in the scenario script. Starting with mosaik 3.3, we force you to be more explicit about it. Weak connections are only allowed between simulators that are part of a common *simulator group*. A simulator group can be created using a Python `with` statement like so:

```
with world.group():
    sim_a = world.start("A")
    sim_b = world.start("B")
with world.group():
    sim_c = world.start("C")
sim_d = world.start("D")
```

In this example, simulators *A* and *B* are part of a common simulator group, so weak connections between entities of *A* and *B* would be admissible. Simulator *C* is also part of a group, but its a different group from the one shared by *A* and *B*, so weak connections between *A* (or *B*) and *C* would not be allowed. (You could connect *C* weakly to itself, though.) Simulator *D* is not part of any group and therefore does not support weak connections at all.

Weak connections are implemented by keeping track of a second time component for each simulator within a group. (This time component is opaque to the simulators.) When sending data along the weak connection, this sub-time is increased instead of the main time.

Normal (and time-shifted) connections can still be established between simulators within and without the group. When a connection leaves a group, the corresponding sub-time is forgotten. For example, suppose that simulator *A* is connected to simulator *D* and that it's the start of the simulation. As long as *A* is performing steps within the same-time loop, its main time will stay at 0 and only its sub-time will increase. Simulator *D*, being outside of *A*'s group, does not see *A*'s sub-time. It will not consider simulator *A*'s step 0 to be done, and simulator *D* therefore will not step. Once the same-time loop is over, *A*'s main time will progress at which time *D* will see the progress and perform its step.

On the other hand, if *D* were part of *A*'s group (but still connected non-weakly), it would see *A*'s sub-time progress right away and therefore perform its step after *A*'s very first step. The decision of whether or not to include a simulator in a group therefore depends on how you want the timing to work.

It is in fact possible to nest groups, by nesting the correponding `with world.group()` blocks. In this case, each additional group will introduce yet another finer level of time. Weak connections between two simulators will increase the sub-time component associated to their closest shared group.

7.7.3 Asynchronous requests

This type of connection is deprecated because it couples the involved simulators too closely.

The final option to resolve the cycle is to use asynchronous requests. For this you only connect the battery's *P* to the controller and let the control strategy set the new schedule via the asynchronous request *set_data*. To indicate this in your scenario, you set the `async_request` flag of *`World.connect`* to `True`:

```
world.connect(battery, controller, 'P', async_requests=True)
```

This way, mosaik will push the value for *P* from the battery to the controller. It will then wait until the controller's *step* is done before the next step for the battery will be computed.

The advantage of this approach is that the call of `set_data` is optional, so you don't need to send a schedule on every step if there's no new schedule. (However, much the same effect can be achieved by using some trigger attributes.) The disadvantage is that you have to implement the `set_data` call within the simulator with the specific destination, making it less modular.

The *step* implementation of the controller could roughly look like this:

```
class Controller(Simulator):

    def step(self, t, inputs):
        schedule = self._get_schedule(inputs)
        yield self.mosaik.set_data(schedule)
        return t + self.step_size
```

7.8 How to filter entity sets

When you create large-scale scenarios, you often work with large sets of entities rather than single ones. This section provides some examples how you can extract a sub-set of entities from a larger entity set based on arbitrary criteria.

Let's assume that we have created a power grid with mosaik-pypower:

```
grid = pypower.Grid(gridfile='data/grid.json').children
```

Since mosaik-pypower's *Grid* entity only serves as a container for the buses and branches of our power grid, we directly bound its *children* to the name `grid`. So *grid* is now a list containing a *RefBus* entity and multiple *Transformer*, *PQBus* and *Branch* entities.

So how do we get a list of all transformers? This way:

```
transformers = [e for e in grid if e.type == 'Transformer']
```

How do we get the single *RefBus*? This way:

```
refbus = [e for e in grid if e.type == 'RefBus'][0]
```

Our *PQBus* entities are named like *Busbar_<i>* and *ConnectionPoint_<i>* to indicate to which buses we can connect consumers and producers and to which we shouldn't connect anything. How do we get a list of all *ConnectionPoint* buses? We might be tempted to do it this way:

```
conpoints = [e for e in grid if e.eid.startswith('ConnectionPoint_')]
```

The problem in this particular case is, that *mosaik-pypower* prepends a "grid ID" to each entity ID, because it can handle multiple grid instances at once. So our entity IDs are actually looking like this: *<grid_idx>-ConnectionPoint_<i>*. Using regular expressions, we can get our list:

```
import re

regex_conpoint = re.compile(r'\d+-ConnectionPoint_\d+')

conpoints = [e for e in grid if regex_conpoint.match(e.eid)]
```

If we want to connect certain consumers or producers to defined nodes in our grid (e.g., your boss says: "This PV module needs to be connected to *ConnectionPoint_23*!"), creating a dict instead of a list is a good idea:

```
remove_gridiidx = lambda e: e.eid.split('-', 1)[1]  # Little helper function
cps_by_name = {remove_gridiidx(e): e for e in grid if regex_conpoint.match(e)}
```

This will create a mapping where the string `'ConnectionPoint_23'` maps to the corresponding `Entity` instance.

This was just a small selection of how you can filter entity sets using list/dict comprehensions. Alternatively, you can also use the `filter` function or a normal `for` loop. You should also take at look at the `itertools` and `functools` modules. You'll find even more functionality in specialized packages like PyToolz.

7.9 How to create user-defined connection rules

The method *World.connect* allows you to only connect one pair of entities with each other. When you work with larger entity sets, you might not want to connect every entity manually, but use functions that take to sets of entities and connect them with each other based on some criteria.

The most common case is that you want to randomly connect the entities of one set to another, for example, when you distribute a number of PV modules over a power grid.

For this use case, mosaik provides *mosaik.util.connect_randomly*. It takes two sets and connects them either evenly or purely randomly:

```
world = mosaik.World(sim_config)

grid = pypower.Grid(gridfile=GRID_FILE).children
pq_buses = [e for e in grid if e.type == 'PQBus']
pvs = pvsim.PV.create(20)

# Assuming that len(pvs) < len(pq_buses), this will
# connect 0 or 1 PV module to each bus:
mosaik.util.connect_randomly(world, pvs, pq_buses, 'P')

# This will distribute the PV modules purely randomly, but every
# bus will have at most 3 modules connected to it.
mosaik.util.connect_randomly(world, pvs, pq_buses, 'P',
                             evenly=False, max_connects=3)
```

Another relatively common use case is connecting a set of entities to one other entity, e.g., when you want to connect a number of controllable energy producers to a central scheduler. For this use case, mosaik provides *mosaik.util.connect_many_to_one*

```
...
pvs = pvsim.PV.create(30)
chps = chpsim.CHP.create(20)
controller = cs.Scheduler()

# Connect all producers to the controller, remember to set the
# "async_requests" flag.
connect_many_to_one(world, chain(pvs, chps), controller, 'P',
                    async_requests=True)
```

Connection rules are oftentimes highly specific for a project. *connect_randomly* and *connect_many_to_one* are currently the only functions that are useful and complicated enough to ship it with mosaik. But writing your own connection method is not that hard, as you can see in the `connect_many_to_one` example:

```
from itertools import chain

def connect_many_to_one(world, src_set, dest_entity, *attrs,
                        async_requests=False):
    for src_entity in src_set:
        world.connect(src_entity, dest_entity, *attrs,
                      async_requests=async_requests)
```

7.10 How to retrieve static data from entities

Sometimes, the entities don't contain all the information that you need in order to decide which entity connect to which, but your simulation model could provide that data. An example for this might be the maximum amount of active power that a producer is able to produce.

Mosaik allows you to query a simulator for that data during composition time via *World.get_data*:

```
>>> example_simulator = world.start('ExampleSim')
Starting "ExampleSim" as "ExampleSim-2" ...
>>> entities = example_simulator.A.create(3, init_val=42)
>>> data = world.get_data(entities, 'val_out')
>>> data[entities[0]]
{'val_out': 42}
```

The entities that you pass to this function don't need to belong to the same simulator (instance) as long as they all can provide the required attributes.

7.11 How to access topology and data-flow information

The *World* contains two networkx Graphs which hold information about the data-flows between simulators and the simulation topology that you created in your scenario. You can use these graphs, for example, to export the simulation topology that mosaik created into a custom data or file format.

This is outdated; mosaik now stored information about the data flow differently. `World.df_graph` is the directed *dataflow graph* for your scenarios. It contains a node for every simulator that you started. The simulator ID is used to label the nodes. If you established a data-flow between two simulators (by connecting at least two of their entities), a directed edge between two nodes is inserted. The edges contain a list of the data-flows as well as the *async_requests*, *time_shifted*, and *weak* flags (see *How to achieve cyclic data-flows*) and the *trigger* and *pred_waiting* flags.

The data-flow graph may, for example, look like this:

```
world.df_graph.node == {
    'PvSim-0': {},
    'PyPower-0': {},
}
world.df_graph.edge == {
    'PvSim-0': {'PyPower-0': {
        'async_requests': False,
        'dataflows': [
            ('PV_0', 'bus_0', ('P_out', 'P'), ('Q_out', 'Q')),
            ('PV_1', 'bus_1', ('P_out', 'P'), ('Q_out', 'Q')),
         ],
    }},
}
```

World.entity_graph is the undirected *entity graph*. It contains a node for every entity. The full entity ID (`'sim_id.entity_id'`) is used as node label. Every node also stores the simulator name and entity type. An edge between two entities is inserted

- if they are somehow related within a simulator (e.g., a PyPower branch is related to the two PyPower buses to which it is adjacent) (see *create*); or
- if they are connected via *World.connect*.

The entity graph may, for example, look like this:

```
world.entity_graph.node == {
    'PvSim_0.PV_0': {'sim': 'PvSim', 'type': 'PV'},
    'PvSim_0.PV_1': {'sim': 'PvSim', 'type': 'PV'},
    'PyPower_0.branch_0': {'sim': 'PyPower', 'type': 'Branch'},
    'PyPower_0.bus_0': {'sim': 'PyPower', 'type': 'PQBus'},
    'PyPower_0.bus_1': {'sim': 'PyPower', 'type': 'PQBus'},
}
world.entity_graph.edge == {
    'PvSim_0.PV_0': {'PyPower_0.bus_0': {}},
    'PvSim_0.PV_1': {'PyPower_0.bus_1': {}},
    'PyPower_0.branch_0': {'PyPower_0.bus_0': {}, 'PyPower_0.bus_1': {}},
    'PyPower_0.bus_0': {'PvSim_0.PV_0': {}, 'PyPower_0.branch_0': {}},
    'PyPower_0.bus_1': {'PvSim_0.PV_1': {}, 'PyPower_0.branch_0': {}},
}
```

The *get_related_entities* API call also uses and returns (parts of) the entity graph. So you can access it in your scenario definition as well as from with a simulator, control strategy or monitoring tool.

Please consult the networkx documentation for more details about working with graphs and directed graphs.

7.12 How to destroy a world

When you are done working with a world, you should shut it down properly:

```
>>> world.shutdown()
```

This will, for instance, close mosaik's socket and allows new `World` instances to reuse the same port again.

`World.run` automatically calls *`World.shutdown`* for you.

7.13 How to do real-time simulations

It is very easy to do real-time (or "wall-clock time") simulations in mosaik. You just pass an *rt_factor* to *`World.run`* to enable it:

```
world.run(until=10, rt_factor=1)
```

A real-time factor of 1 means, that 1 simulation time unit (usually a simulation second) takes 1 second of real time. Thus, if you set the real-time factor to 0.5, the simulation will run twice as fast as the real time. If you set it to 1/60, one simulated minute will take one real-time second.

It may happen that the simulators are too slow for the real-time factor chosen. That means, they take longer than, e.g., one second to compute a step when a real-time factor of one second is set. If this happens, mosaik will by default just print a warning message to stdout. However, you can also let your simulation crash in this case by setting the parameter *rt_strict* to `True`. Mosaik will then raise a `RuntimeError` if your simulation is too slow:

```
world.run(until=10, rt_factor=1/60, rt_strict=True)
```

7.14 How to call extra methods of a simulator

A simulator may optionally define additional API methods (see *init*) that you can call from your scenario. These methods can implement operations, like setting some static data to a simulator, which don't really fit into `init()` or `create()`.

These methods are exposed via the model factory that you get when you start a simulator. In the following example, we'll call the `example_method()` that the example simulator shipped with the mosaik Python API:

```
>>> world = mosaik.World({'ExampleSim': {
...     'python': 'example_sim.mosaik:ExampleSim'}})
>>> es = world.start('ExampleSim')
Starting "ExampleSim" as "ExampleSim-0" ...
>>>
>>> # Now brace yourself ...
>>> es.example_method(23)
23
>>>
>>> world.shutdown()
```

CHAPTER

EIGHT

THE SIMULATOR MANAGER

The simulator manager (or just *sim manager*) is responsible for starting and handling the external simulator processes involved in a simulation as well as for the communication with them.

Usually, these simulators will be started as separate sub-processes which will then connect to mosaik via network sockets. This has two benefits:

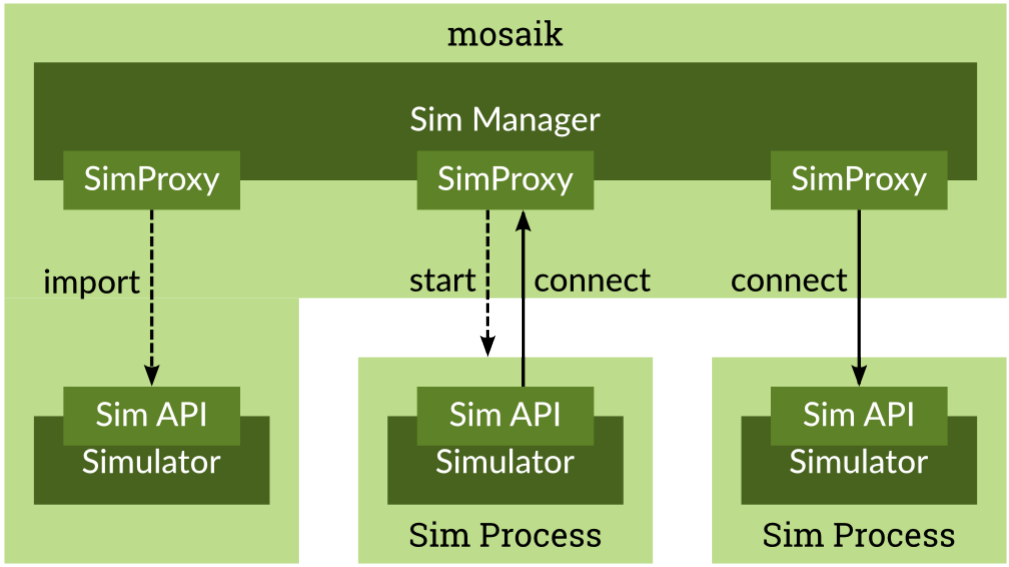
1. Simulators can be written in any language.
2. Simulation steps can be performed in parallel, if two processes don't depend on each others data.

Simulators written in Python 3 can, for performance reasons, be imported and executed like normal Python modules. This way, all Sim API calls will be plain functions calls without the overhead of network communication and message (de)serialization. However, since Python only runs in one thread at a time, this will also prevent parallel execution of simulators.

When a (Python 3) simulator is computationally inexpensive, running it in-process may give you good results. If it performs a lot of expensive computations, it may be better to start separate processes which can then do these computations in parallel. In practice, you should try and profile both ways in order to get the maximum performance out of it.

Sometimes, both ways won't work because you simply cannot start the simulator process by yourself. This might be the reason for hardware-in-the-loop or if a simulator needs to run on a separate machine. In these cases, you can simply let mosaik connect to a running instance of such a simulator.

Internally, all three kinds of simulator processes (in-process with mosaik, started by mosaik, connected to by mosaik) are represented by *SimProxy* objects so that they all look the same to the other components of mosaik:



The sim manager gets its configuration via the *World*'s *sim_config* argument. The *sim_config* is a dictionary containing simulator names and description of how to start them:

```
>>> import mosaik
>>>
>>> sim_config = {
...     'SimA': {
...         'python': 'package.module:SimClass',
...     },
...     'SimB': {
...         'cmd': 'java -jar simB.jar %(addr)s',
...         'cwd': 'simB/dist/',
...     },
...     'SimC': {
...         'connect': 'localhost:5678',
...     },
... }
>>>
>>> world = mosaik.World(sim_config)
>>> world.shutdown()
```

In the example above, we declare three different simulators. You can freely choose a name for a simulator. Its configuration should either contain a *python*, *cmd* or *connect* entry:

Since mosaik 2.3.0 it is possible to pass environment variables to sub-processes. Using the key *env* in sim_config allows you to set new environment variables. That could look like this:

```
>>> import mosaik
>>>
>>> sim_config = {
...     'SimA': {
...         'python': 'package.module:SimClass',
...         'env': {
...            'PYTHONPATH': 'src/',
...         },
...     },
```

(continues on next page)

(continued from previous page)

```
... }
>>>
>>> world = mosaik.World(sim_config)
>>> world.shutdown()
```

python

This tells mosaik to run the simulator in process. As a value, you need to specify the module and class name of the simulator separated by a colon. In the example, mosaik will `import package.module` and instantiate `sim = package.module.SimClass()`. This only works for simulators written in Python 3.

cmd

This tells mosaik to execute the specified command *cmd* in order to start a new sub-process for the simulator.

You can use the placeholder `%(python)s` to use the same Python interpreter and virtualenv that mosaik currently uses (see `sys.executable`).

In order to create a socket connection to mosaik the simulator needs to know the address of mosaik's server socket. Mosaik will pass this address (in the form `host:port`) as a command line argument, so you need to include the placeholder `%(addr)s` in your command. Mosaik will replace this with the actual address.

If the simulator should open a seperate console window, the *new_console* keyword can be used. This option is only available on Windows machines and mosaik version >= 3.2.0.

```
>>> sim_config = {
...       'SimB': {
...           'cmd': 'java -jar simB.jar %(addr)s',
...           'new_console': True,
...       },
... }
```

You can optionally specify a current working directory *(cwd)*. If it is present, mosaik will change to that directory before executing *cmd*. Its default value is `'.'`.

In our example, mosaik would execute:

```
$ cd simB/java
$ java -jar simB.jar localhost:5555
```

in order to start *SimB*.

Note: Please use for the *cwd* command and for paths in the *cmd* call only the UNIX/Linux path notation with slashes even if you are using windows. Do not use backslashes or double backslashes.

---

connect

This tells mosaik to establish a network connection to a running simulator instance. It will simply connect to `host:port` – `localhost:5678` for *SimC*

CHAPTER

NINE

SCHEDULING AND SIMULATION EXECUTION

When you *defined your scenario* and *start the simulation*, mosaik's scheduler becomes active. It manages the execution of all involved simulators, keeps them in sync and handles the *data-flows* between them.

Mosaik runs the simulation by *stepping* simulators through time. Mosaik uses integers for the representation of time (to avoid rounding errors etc.). Its unit (to how many seconds one integer step corresponds) can be defined in the scenario, and is passed to every simulation component via the *init function* as key-word parameter *time_resolution*. It's a floating point number and defaults to *1.*.

9.1 Time paradigms

There are various paradigms for time in simulations: discrete time, continuous time, and discrete event, to name the probably most common ones. Mosaik supports discrete-time and discrete-event simulations, including the combination of both. As these concepts are not always strictly distinguishable, we use a slightly different notation for the simulator's types, namely time-based, event-based, and hybrid. It's not always obvious which type of simulator is the most appropriate one for a simulator, and in many cases both would be possible. As a rough guide we could say:

Time-based simulators are more related to the physical world, where the state, inputs, and outputs of a system are continuous (e.g. active power of a PV module). The mapping of those continuous signals to discrete points in time is then somewhat arbitrary (and depends on the desired precision and the available computing resources). The lower limit for the temporal resolution in a mosaik scenario is the unit assigned to the integer time steps.

Event-based simulators are related to the cyber world, where the state(s) of a system can instantaneously change, and inputs and outputs also occur at a specific point in time (example: sending/receiving of messages in a communication simulation). The native way of stepping through time would then be to just jump between all occurring events. In a mosaik simulation the time of the events have to be rounded to the mosaik's integer time steps.

Hybrid simulators can represent any kind of combined systems with both time-based and event-based components.

9.2 Advancing through time

Mosaik tracks the current simulation time for every simulator individually. How simulators step through the time from simulation start to end depends on their stepping type described above:

9.2.1 Time-based simulators

When the simulation starts, all time-based (and hybrid) simulators are at time 0. When it asks a simulator to perform its next step, it passes its current simulation time t_{now} to it. After its step, the simulator returns the time at which it wants to perform its next step (t_{next}). Thus, a simulator's step size doesn't need to be constant but can vary during the simulation.

All data that a time-based simulator computes during a step is valid for the right-open interval $[t_{now}, t_{next})$ as shown in the following figure.

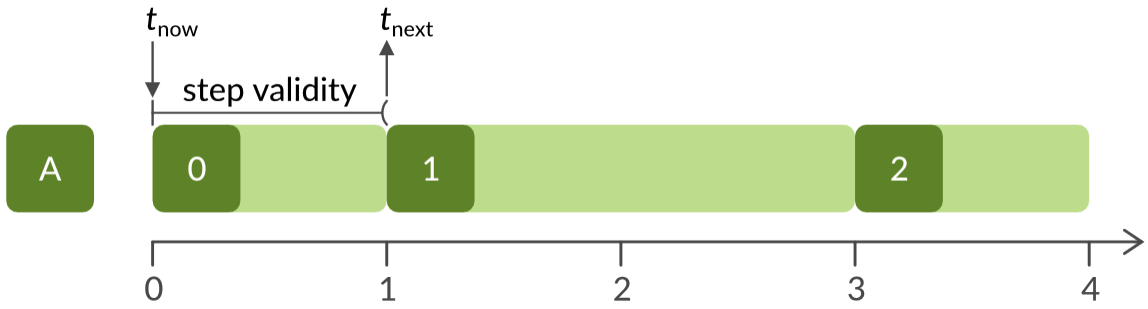


Fig. 1: Schematic execution of a time-based simulator *A*. t_{now}, t_{next} and the validity interval for its first step 0 are shown. The figure also shows that the step size of a simulator may vary during the simulation.

9.2.2 Event-based simulators

The stepping through time of event-based simulators is rather different. Event-based simulators are stepped at all times an event is created at. These events can either be created by other simulators that are connected to this simulator via providing the connected attribute, or the simulator can also schedule events for itself via the step function's return value. The output provided by event-based simulators is only valid for a specific point in time, by default for the current time of the step, or for any later time if explicitly set via the (optional) output time. Providing the output attributes is optional for event-based simulators. As consequence a simulator connected to a specific attribute is only triggered/stepped if the output is actually provided. See the *API description* for implementation details. Event-based simulators do not necessarily start at time 0, but whenever their first event is scheduled, either by other simulators or via `World.set_initial_event` from the scenario definition.

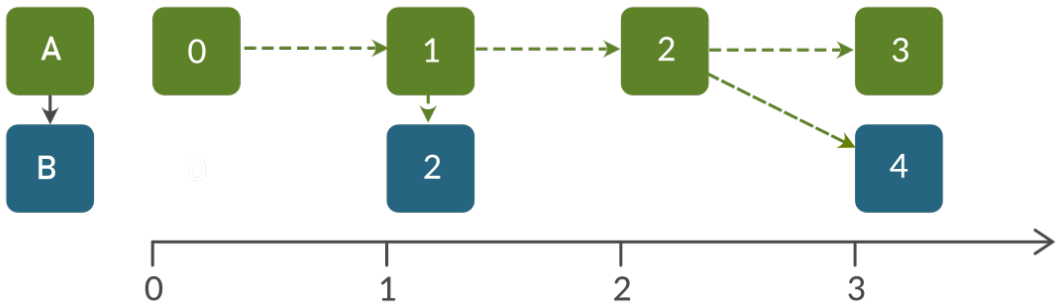


Fig. 2: Schematic execution of an event-based simulation. Depending on *A*'s actual output a step of *B* is triggered (or not), at *A*'s step time or later. Simulator *A* also schedules itself.

Note that it is possible that a simulator is stepped several times at a specific point in time. See *Same-time loops* for details.

9.3 Synchronization and data-flows

If there are data-flows between two simulators (because you connected some of their entities), a simulator can only perform a step if all input data has been computed.

Let's assume we created a data-flow from a simulator *A* to a simulator *B* and *B* wants to perform a step from $t_{\text{now(B)}}$. Mosaik determines which simulators provide input data for *B*. This is only *A* in this example. In order to provide data for *B*, *A* needs to step far enough to produce data for $t_{\text{now(B)}}$, that means $t_{\text{next(A)}} > t_{\text{now(B)}}$ as the following figure illustrates.

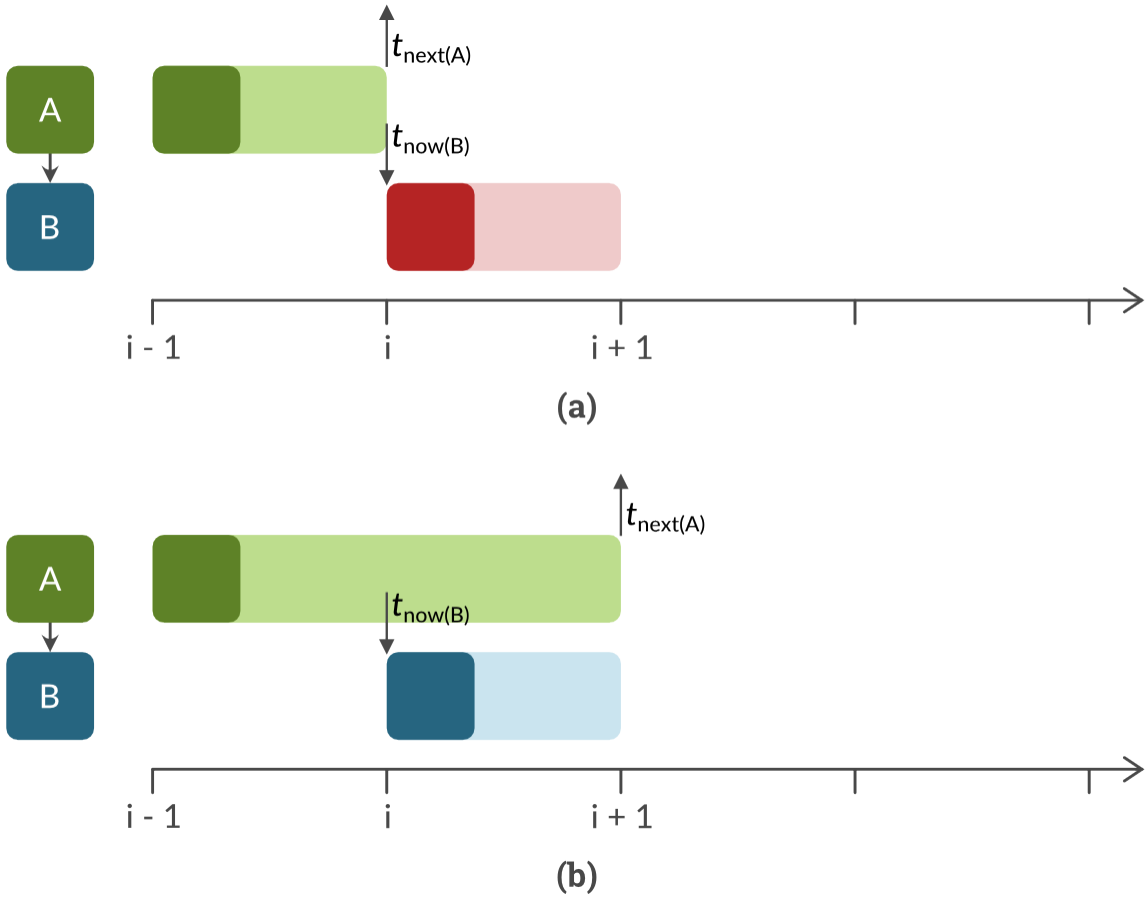


Fig. 3: **(a)** *B* cannot yet step because *A* has not progressed far enough yet ($t_{\text{next(A)}} <= t_{\text{now(B)}}$).
(b) *B* can perform its next step, because *A* now has progressed far enough ($t_{\text{next(A)}} > t_{\text{now(B)}}$).

When this condition is met for all simulators providing input for *B*, mosaik collects all input data for *B* that is valid at $t_{\text{now(B)}}$ (you could say it takes *one* snapshot of the global simulation state). It passes this data to *B*. Based upon this (and *only* this) data, *B* performs its step $[t_{\text{now(B)}}, t_{\text{next(B)}})$.

This is relatively easy to understand if *A* and *B* have the same step size, as the following figures shows:

If *B* had a larger step size than *A*, *A* would produce new data while *B* steps. *B* would still only use the data that was valid at $t_{\text{now(B)}}$, because it only "measures" its inputs once at the beginning of its step:

On the other hand, if *A* had a larger step size than *B*, we would reuse the same data from *A* multiple times as long as it is valid:

The last two examples may look like special cases, but they actually arise from the approach explained above.

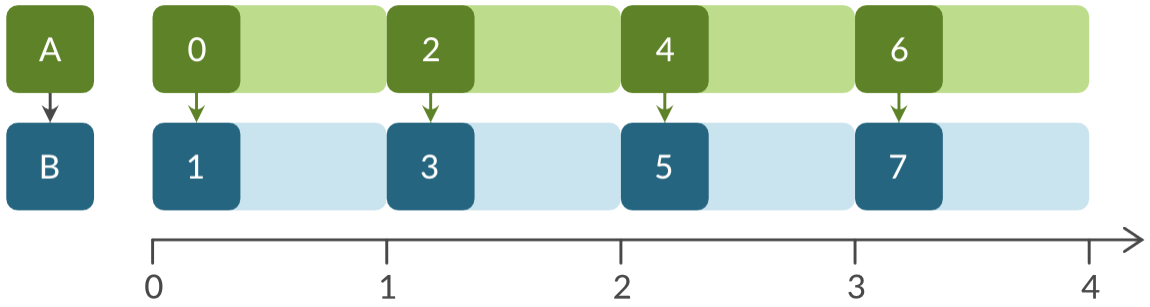


Fig. 4: In this example, *A* and *B* have the same step size. Mosaik steps them in an alternating order starting with *A*, because it provides the input data for *B*.

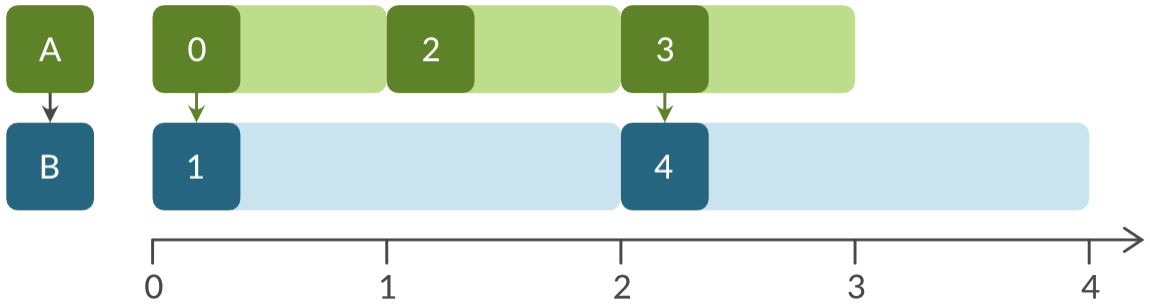


Fig. 5: In this example, *B* has a larger step size. It doesn't consume all data that *A* produces, because it only gets data once at the beginning of its step.

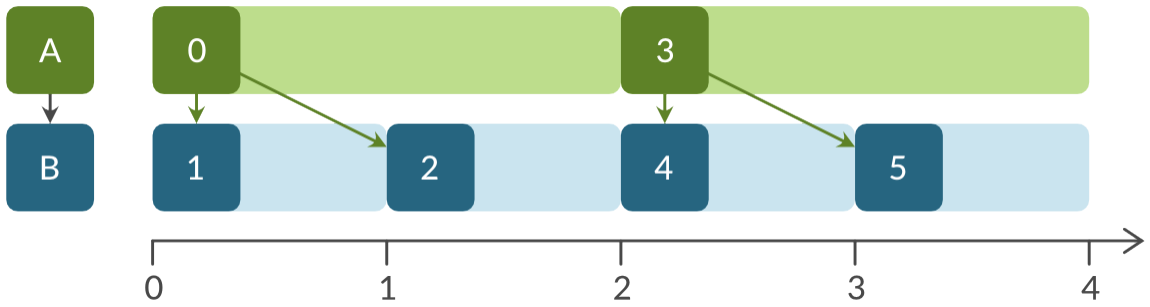


Fig. 6: In this example, *A* has a larger step size. *B* reuses the same data multiple times because it is still valid.

9.3.1 How far is a simulator allowed to advance its time?

As described in the *API documentation*, mosaik tells the simulator each step how far it is allowed to advance its internal simulation time via the *max_advance* argument. It is guaranteed that no step will be scheduled until then (inclusively), unless the simulator activates a triggering dependency loop earlier than that. Mosaik deduces this from the simulation topology and the progress of the simulators. Note that the simulator will not necessarily be stepped at *max_advance + 1* as this will only happen if the predecessor actually provides the connected output attribute(s).

As time-based simulators (or hybrid ones without any triggering input) only decide themselves when they are stepped, *max_advance* is always equal to the end of the simulation for those. But of course they will most likely miss some updates of the input data if their step size is too large and not synchronized with their input providers. In order not to miss any input update, you can change the type of the simulator to *hybrid*. Then the simulator will be stepped on each update.

Note: The *max_advance* value is not necessarily appropriate for real-time simulations as it does not consider eventual steps which are scheduled via the asynchronous *set_event()* method.

9.3.2 How data flows through mosaik

After a simulator is done with its step, mosaik determines, based on the data-flows that you created in your scenario, which data other simulators need from it. It makes a *get_data()* API call to the simulator and stores the data that this call returns in an internal buffer. It also memorizes for which time this data is valid.

Before a simulator steps, mosaik determines in a similar fashion what input data the simulator needs. Mosaik checks if all input-providing simulators have stepped far enough to (potentially) provide that data and waits otherwise. After that all input data is collected and then passed to the *inputs* parameter of the *step()* API call.

It is important to understand that simulators don't talk to each other directly but that all data flows through mosaik were it can be cached and managed.

9.4 Cyclic data-flows

Sometimes the simulated system requires cyclic data-flows between components, e.g. a control mechanism *(C)* that controls another entity *(E)* based on its state, e.g. by sending commands or a schedule.

It is not possible to perform both data-flows (the state from *E* to *C* and the commands/schedule from *C* to *E*) at the same time because they depend on each other (yes, this is similar to the chicken or egg dilemma).

The cycle can be resolved by first stepping *E* (e.g., from $t = 0$ to $t = 1$). *E*'s state for that interval can then be used as input for *C* 's step for the same interval. The commands/schedule that *C* generates for *E* will then be used in *E*'s next step. This results in a serial execution, also called Gauss-Seidel scheme.

This resolution of the cycle makes sense if you think how this would work in real life. The controller would measure the data from the controlled unit at a certain point *t*. It would then do some calculation which take a certain amount of time Δt which would be send to the controlled unit at $t + \Delta t$.

However, mosaik is not able to automatically resolve that cycle. That's why you are not allowed to `connect(E, C)` and `connect(C, E)` in a scenario. This can be done via the time-shifted connection `connect(C, E, ('c_out', 'a_in'), time_shifted=True, initial_data={'c_out': 0})`, which tells mosaik that the output of *C* is to be used for E's next time step(s) afterwards. As for the first step (at time 0) this data cannot be provided yet, you have to set it via the initial_data argument. In this case, the initial data for *'a_in'* is 0.

Another way to resolve this cycle is to allow async. requests via the async_requests flag `connect(E, C, async_requests=True)` and use the *asynchronous callback* `set_data()` in *C*'s *step()* implementation in order to send the

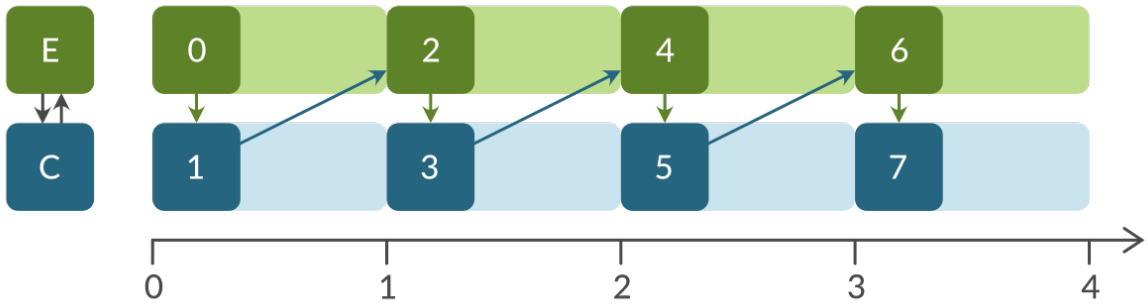


Fig. 7: In this example, a controlled entity *E* provides state data to the controller *C*. The commands or schedule from *C* is used by *E* in its next step.

commands or schedule from *C* to *E*. The advantage of this approach is that the call of set_data is optional, i.e. the commands or schedules don't need to be sent on every step.

If you set the time_shifted flag for both connections, the simulators can be executed in parallel (Jacobi scheme). Note that a computationally parallel execution is only possible for simulators that are not run in-process.

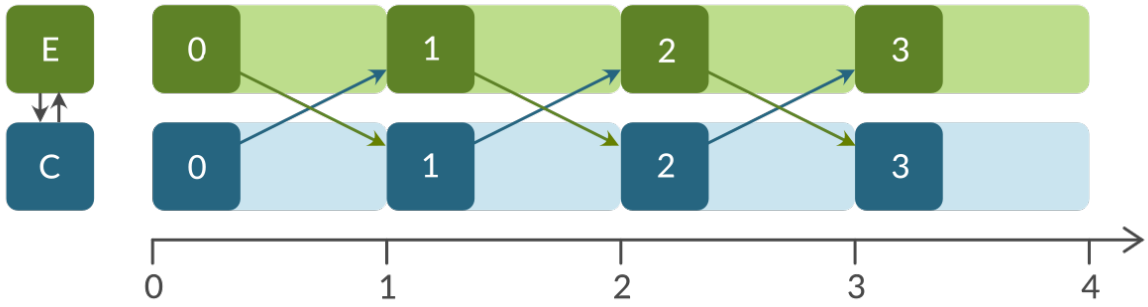


Fig. 8: In this example, two entities are running in parallel. The outputs of each simulator are used by the other one in its next step afterwards.

You can take a look at our *discussion of design decisions* for details.

9.4.1 Same-time (algebraic) loops

Sometimes, simulators need to exchange data back and forth at the same time before they can step to their next time step. Such same-time loop can be defined via a weak connection. In the connect statement, the connection is marked as weak, e.g. via `world.connect(agent, model, 'delta', weak=True)`.

Loops which are closed by a weak connection can be run multiple times within the same mosaik time step, as weak connections do not necessarily imply a temporal progress. This can be used for example to only advance the simulation time when the state has converged to a stable solution. To activate (and also stay in) a same-time cycle, a simulator has to provide its 'cyclic' attribute(s) via the *get_data* function and indicating as output time the current step time. To escape the cycle, the attribute(s) in the get_data's return dictionary have to be omitted or a time later than the step's time indicated. An example scenario for this is shown in a *tutorial*.

To prevent the loop to be run infinite times, mosaik raises a runtime error when a certain number of iterations within one time step has been reached. The default maximum iteration count is 100 and can be adjusted via the *max_loop_iterations* parameter within the scenario definition if needed (see `mosaik.scenario.World`).

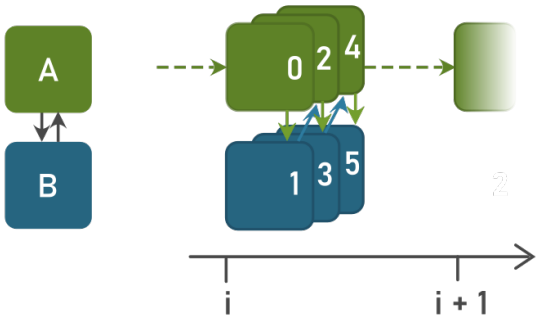


Fig. 9: A same-time loop with three repetitions between simulator *A* and *B*.

9.5 Stepping and simulation duration

By now you should have a general idea of how mosaik handles data-flows between simulators. You should also have the idea that simulators only perform a step when all input-providing simulators have stepped far enough. But what if they don't have any (connected) inputs? In this section you'll learn about the algorithm that mosaik uses to determine whether a simulator can be stepped or not.

This is how it works:

1. Should there be a next step at all? *

   Yes: Go to step 2.

   No: Stop the simulator.

   * *We'll explain how to answer this question below.*

2. Is a next step already scheduled, either self-scheduled via step or by triggering input?

   Yes: Go to step 3.

   No: Wait until a next step is set. Then go to step 3.

3. Have all dependent simulators stepped far enough?

   Yes: Go to step 4.

   No: Wait for all dependencies. Then go step 4.

4. Collect all required input data.
5. Send collected input data to simulator, perform the simulation step and eventually get the time of a next step.
6. Get all data from this simulator that are connected to other simulators and store it internally.
7. Notify other simulators that already wait for this simulator. If there's any output which is connected to a triggering input of another simulator, schedule new steps for it (at output time).

So how do we determine whether a simulator must perform another step or it is done?

When we start the simulation, we pass a time unto which our simulation should run (`world.run(until=END)`). Usually a simulator is done if the time of its next step is equal or larger than the value of *until*. This is, however, not true for *all* simulators in a simulation. If no one needs the data of a simulator step, why perform this step?

So the actual algorithm is as follows:

If a simulator has no outgoing data-flows (no other simulator needs its data) it simulates until the condition $t_{\text{next}} > t_{\text{until}}$ is met or none of the simulators which could trigger a step are running anymore.

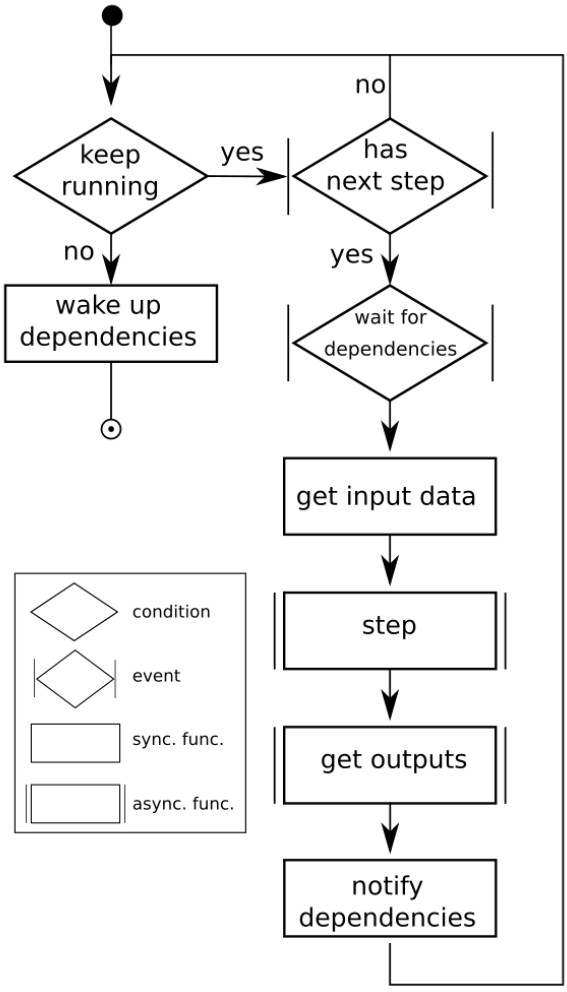


Fig. 10: Sim-process running for each simulator in parallel

Else, if a simulator needs to provide data for other simulators, it keeps running until all of these simulators have stopped.

CHAPTER

TEN

UPGRADING FROM MOSAIK 2 TO 3

Mosaik 3 has some new features which required some API changes. For the time being simulators implementing mosaik-api 3 are still supported, but you will get a deprecation warning at the beginning. In case that you don't want to use any of the new features you could *stay with mosaik 2*. But otherwise the upgrade to mosaik 3 is quite easy, as you will see in the following sections:

10.1 Component's type

Simulation components now have a *type* (*time-based*, *event-based*, or *hybrid*) to indicate which simulation time paradigm they implement. See details *here*. The type is set in the component's meta data, which the component returns to mosaik via the API's *init* function.

10.2 Time resolution

A global *time resolution* is now defined for each scenario indicating how to translate mosaik's integer time to simulated time. It can be set at the instantiation of the simulation's `World` in the *setup of the scenario* and is set to *1.* by default. It's value is passed to the components via the *init* function.

10.3 Max_advance time

max_advance tells the simulator how far it can advance its time without risking any causality error, i.e. it is guaranteed that no external step will be triggered before max_advance + 1. It is determined for each step and passed to the component as third positional argument of the API's *step* function.

10.4 Sticking with mosaik 2

In case that you don't want to upgrade to mosaik 3, the mosaik dependency must be pinned to version 2.x.

When installing mosaik with `pip`, you get the latest mosaik 2.x version via:

```
pip install 'mosaik<3'
```

You can also specify this constraint in a requirements files:

```
mosaik<3
```

Or in the `install_requires` list in a `setup.py` file:

```
setup(
    ...
    install_requires=['mosaik<3'],
    ...
)
```

CHAPTER

ELEVEN

FAQ

This is a list of some questions we recently got. If you cannot find an answer for your questions here, you are welcome to post it on our mailing list.

11.1 General questions

11.1.1 Are there graphical tools for scenario design?

Maverig, an environment for graphical analysis as well as scenario design has been developed and is currently in a prototypical state. It may be used for demonstration purposes, but its maintenance and further development is not our top priority right now. Please understand, if the installation guide is not up-to-date with the newest package versions. However, we argue that graphical tools are not feasible for the design of large and complex scenarios. For most applications the more flexible scenario design by code is advantageous.

11.1.2 Is there a mosaik-Wiki?

We like Wikis but consider them the wrong tool for documenting software (and we are not alone with that).

There are a lot of other resources, though:

- All documentation is concentrated here, at Read the Docs.
- Source code and issues are managed by GitLab.
- Discussion takes place in our mailing list.
- News are spread through our blog.

There's not much that a wiki could add here.

11.2 Coupling models with mosaik

11.2.1 Can I use continuous models with mosaik?

Yes. Since mosaik 2 you can even use a variable step size for your simulator that can dynamically change during the simulation.

11.2.2 Can I use mosaik only with Python programs?

No, mosaik can be used with any language that provides network sockets and ways to (de)serialize JSON.

Since implementing network event loops and message (de)serialization is repetitive work and unnecessary overhead, we provide so called *high-level* APIs for certain languages that provide a base class that you can inherit and just need to implement a few methods representing the API calls.

Currently, high-level API are available for Python, JAVA, and C#, but an implementation for C++ will follow soon.

11.2.3 Can I use my MATLAB/Simulink models with mosaik? How do I do it?

Yes, you can. We will provide an example soon. In the end, if you manage to let your model communicate via sockets and are capable of serializing/deserializing JSON data objects you can use it with mosaik (see *Can I use mosaik only with Python programs?* for details).

11.3 The mosaik demo

11.3.1 I can only see active power values in the web visualization. Does mosaik also support reactive power values with PyPower?

Of course, it only depends on your models. The basic models distributed with mosaik 2 only produce active power outputs (*cos phi = 1*), so we don't display reactive power.

11.3.2 What are the power grid's parameters? How are the cables'/lines' parameters formatted?

Check https://gitlab.com/mosaik/mosaik-pypower under "input file format". Typically, line values are given in *R per km* and *X per km*.

11.3.3 Port 8000 on my local machine is already in use. How can I see the visualisation with WebVis?

Port 8000 is used as default when using WebVis. You can overwrite the default value in demo.py.

```
...
sim_config = {
    ...
    'WebVis': {
        'cmd': 'mosaik-web -s 0.0.0.0:8000 %(addr)s',
    },
}
...
```

CHAPTER

TWELVE

DEVELOPER'S DOCUMENTATION

Contents:

12.1 Discussion of design decisions

On this page, we discuss some of the design decisions that we made. This should explain why some features are (not) present and why they work the way that they work.

Note: For the sake of readability, some concepts are simplified in the following sections. For example, the snippet `connect(A, B)` means we're connecting some entities of a simulator *A* to some entities of simulator *B*; *simulator* and *entity* are used as if they were the same concept; *A.step()* means, that mosaik calls the *step()* function of simulator/entity *A*.

Here are the topics:

- *Circular data-flows*
- *Same-time loops for control cycles*
  - *The category of tiered intervals*

12.1.1 Circular data-flows

Circular data-flows were one of the harder problems to solve.

When you connect the entities of two simulators with each other, mosaik tracks the new dependency between these simulators:

```
connect(A, B)
```

A would now provide input data for *B*. When mosaik runs the simulation, the step for a certain time *t* would first be computed for *A* and then for *B* with the inputs of *A*.

In order to connect control strategies (like multi-agent systems) with to-be-controlled entities, you usually need a circular data-flow. The entity provides state information for the controller which in turn sends new commands or schedules to the controlled entity. The naïve way of doing this would be:

```
connect(A, B, 'state')
connect(B, A, 'schedule')
```

A would receive schedules via the inputs of *A.step()*. In *A.step()*, it would compute new state information which mosaik would get via *A.get_data()*. Mosaik would forward this to the inputs of *B.step()*. *B.step()* would calculate some schedules, which mosaik would again get via *B.get_data()* and pass to *A.step()* …

The question that arise here is: Which simulator do we step first – *A* or *B*? Mosaik has no clue. You *could* say that *A* needs to step first, because the data-flow from *A* to *B* was established first. However, if you re-arrange your code and (accidentally) flip both lines, you would get a different behavior and a very hard to find bug.

What do we learn from that? We need to explicitly tell mosaik how to resolve these cycles and prohibit normal circular data-flows as in the snippet above.

Mosaik provides two ways for this. The first is via time-shifted connections:

```
connect(A, B, 'state')
connect(B, A, 'schedule', time_shifted=True)
```

This tells mosaik how to resolve the cycle and throws an error if you accidentally flip both lines.

Theoretically, we could be done here. But we aren't. The data-flows in the example above are passive, meaning that *A* and *B* compute data hoping that someone will use them. This abstraction works reasonably well for normal simulation models, but control mechanism usually have an *active* role. They actively decide whether or not to send commands to the entities they control.

Accordingly, mosaik provides ways for control mechanisms and monitoring tools to actively collect more data from the simulation and set data to other entities. These means are implemented as *asyncronous requests* that a simulator can perform during its step. Similar to the cyclic data-flows, this requires you to tell mosaik about it to prevent some scheduling problems:

```
connect(A, B, async_requests=True)
```

This prevents *A* from stepping too far into the future so that *B* can get additional data from or set new data to *A* in *B.step()*.

Since you can set data via an asynchronous request, you can implement cyclic data-flows with it:

```
connect(A, B, 'state', async_requests=True)
```

The implementation of *A.step()* and *A.get_data()* would be the same. In *B.step()* you would still receive the state information from *A* and compute the schedules. However, you wouldn't store them somewhere so that *B.get_data()* can return them. Instead, you would just pass them actively to *set_data()*. Mosaik stores that data in a special input_buffer of *A* which will be added to the input of *A*'s next step.

So to wrap this up, there are two possibilities to achieve cyclic data-flows:

1. Passive controller:

   ```
   connect(A, B, 'state')
   connect(B, A, 'schedules', time_shifted=True)
   ```

   B.step() computes schedules and caches them somewhere. Mosaik gets these schedules via *B.get_data()* and sends them to *A*.

   If you forget to set the `time_shifted=True` flag, mosaik will raise an error at *composition time*.

   If you forget the second *connect()*, nothing will happen with the schedules. You may not notice this for a while.

2. Active controller:

   ```
   connect(A, B, 'state', async_requests=True)
   ```

   B.step() computes schedules and immediately passes them to *set_data()*. Mosaik sends them to *A*.

   If you forget to set the `async_requests=True` flag, mosaik will raise an error at *simulation time*.

12.1.2 Same-time loops for control cycles

The first implementation of same-time loops was pretty ad-hoc and therefore showed some difficult-to-explain behavior. In particular, the order in which simulators in a same-time loop would trigger each other depended in subtle ways on the order in which connections were established. (The execution order was governed by the *rank* of a simulator, which was calculated using a topological sort method from networkx.)

As a first solution, we attempted a concept of dense time, where each internal time stamp had two components, called *time* and *micro step*. Only the time component is communicated to the simulators, whereas the micro step component is only used internally. Progress along weak connections would only increase the micro step component, allowing simulators to perform multiple steps in one time step, while still preserving an order for all steps.

Unfortunately, a very common control structure that usually happened to work under the old scheme was not possible anymore: The simplest version of the case in question is when there are two simulators that communicate via a same-time loop and which then send the result of their negotiation to a third simulator. Because users would normally create the connections for the negotiation part earlier in their scenario, the corresponding same-time loop would run first, leading to the expected result. With the dense-time setup, the third simulator would instead run at micro step 0, and thus only receive the results of the first round of negotiation.

To resolve this problem, we introduce a concept of *simulator groups*. A user can create a group in their script using a `with` statement like

```
with world.group():
    world.start("SimulatorA")
    world.start("SimulatorB")
```

All simulators started in the `with` block are automatically added to the group. Users can nest groups by nesting the `with world.group()` blocks. (Using the `with` statement has the advantage that users cannot nest the groups incorrectly.)

Each group adds an additional tier to the internal times used for simulators contained within, leading to a concept of *tiered time*. (The simulators still only get to see the highest level.) When a connection leaves a group (i.e. leads from a simulator in a group to a simulator not in that group), the corresponding part of the time is cut off. This results in the simulator outside the group not seeing the steps inside the group. It therefore only considers the steps of its inputs done when the simulators in the group progress their actual time, i.e. when they conclude their negotiation.

Similarly, a time entering a group will be padded with zeros at the end.

To force users to update their simulations (so they don't silently start producing completely different results), we decided to outlaw weak connections outside of groups. A user using weak connections will receive an error message leading them to the documentation that explains how to adapt their scenario.

The category of tiered intervals

This is additional mathematical information on tiered time.

Times are used in two different ways: They can represent specific points in the simulation or they can represent intervals of time. Previously, both of these concepts were represented by the same data structure internally. With tiered time, we also introduce *tiered interval*, so tiered times now only represent points in time. It is legal to add an interval to a time, and to add two intervals, but it is not legal to add two times.

Speaking of intervals, there are three basic types:

- Connections within a group keep the number of tiers fixed an just add something to some tiers (usually nothing at all, or 1 to the first tier for time-shifted connections, or 1 to the last tier for weak connections).
- Connections leaving a group cut off some tiers at the end.
- Connections entering a group add some tiers at the end.

When calculating the progression within the simulation, it becomes necessary to add these types of intervals together, leading to mixed types. Therefore, a tiered interval consists of:

- A list of addition tiers. These are added to the corresponding tiers of the tiered time. All further tiers of the tiered time are cut off.
- A list of extension tiers. There are appended to the result of the addition and cutoff.
- A pre-length, which is used as a sanity check. The pre-length must equal the number of tiers of the time to which the interval is added. Also, the pre-length serves as an upper bound for the number of addition tiers, so that the time always has enough components to add to.

The addition rules for intervals are set up such that adding two intervals to a time one after the other is the same as adding the sum of the two intervals to the time, and such that the addition of intervals is associative. (Mathematically, this turns tiered times and intervals into a category with sets of tiered times of the same length as objects and tiered intervals as arrows.)

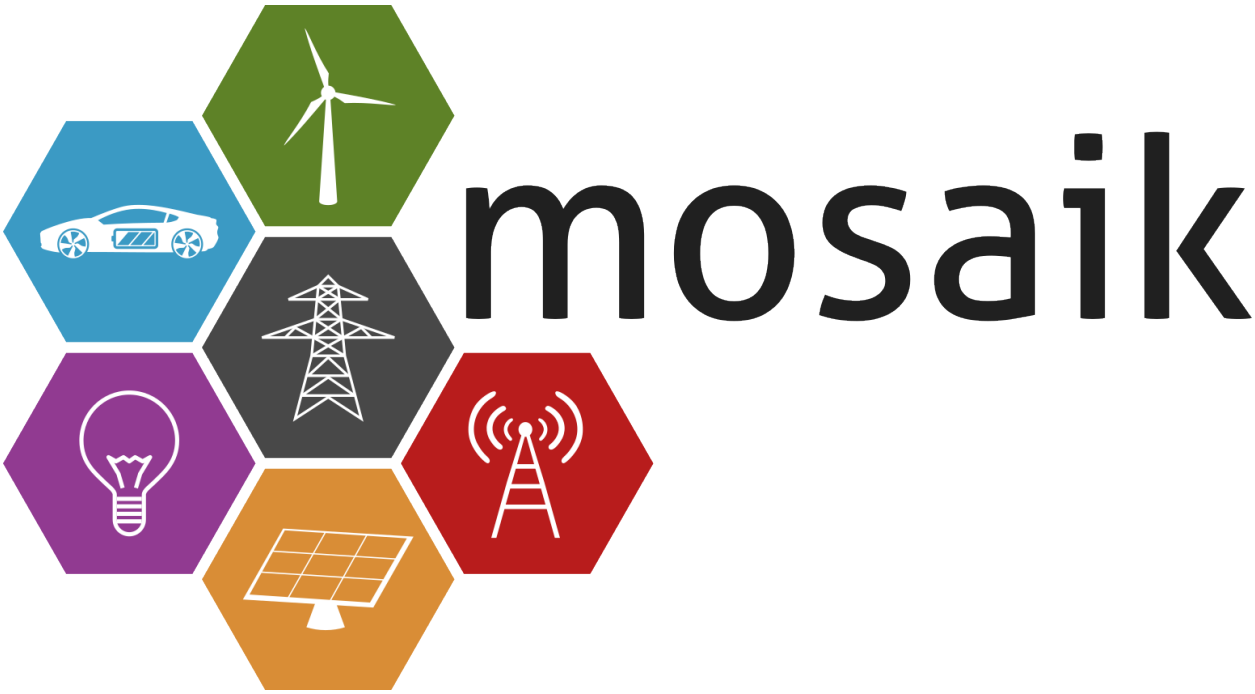
12.2 Logo and corporate identity

This page contains download links to the mosaik logos and lists the official mosaik colors.

12.2.1 Normal logo

This is the large version of the logo.

PNG, 1920 × 1052 px, with transparency



SVG for the web

Letters converted to objects so that it nicely renders on all devices.

12.2.2 Simple logo

The simple variant of the logo should be used when the display size for the logo is so small, that the icons would not be recognizable very well, e.g. in the headers of letters or papers.

PNG, 512 × 280 px, with transparency

SVG for the web

Letters converted to objects so that it nicely renders on all devices.

SVG for Inkscape

This is the Inkscape master file for the logo. In order to edit it in Inkscape, you need to install the Aller font.

12.2.3 Icon

This version should be used for (program) Icons.

PNG, 512 × 512 px, with transparency

SVG

12.2.4 Colors

Value ranges RGB:
[0-255] [0-255] [0-255]

Value ranges HSB/HSL:
[0-359]° [0-100]% [0-100]%

mosaik logo

These colors are usually only used in the logo.

Mod.	Green	Orange	Red	Purple	Blue	Gray
CMYK	28 0 70 49	0 35 75 15	0 85 85 28	0 60 0 43	70 21 0 12	0 0 0 72
RGB	94 130 39	217 141 54	184 28 28	145 58 145	59 154 196	72 72 72
LAB	50 -27 43	65 22 56	40 59 42	40 48 -31	60 -14 -30	31 0 0
HSB	84 70 51	32 75 85	0 85 72	300 60 57	198 70 77	0 0 28
HSL	84 54 33	32 68 53	0 74 42	300 43 30	198 54 50	0 0 28
HEX	#5E8227	#D98D36	#B81C1C	#913A91	#3B9AC4	#484848

mosaik dark

These colors are usually used for figures, diagrams and in presentations.

Mod.	Green	Orange	Red	Purple	Blue	Gray
CMYK	28 0 70 49	0 37 81 34	0 80 80 29	0 60 0 43	70 21 0 50	0 0 0 72
RGB	94 130 39	168 105 32	181 36 36	145 58 145	38 101 128	72 72 72
LAB	50 -27 43	50 19 49	40 56 38	40 48 -31	40 -11 -21	31 0 0
HSB	84 70 51	32 81 66	0 80 71	300 60 57	198 70 50	0 0 28
HSL	84 54 33	32 68 39	0 67 43	300 43 30	198 54 33	0 0 28
HEX	#5E8227	#A86920	#B52424	#913A91	#266580	#484848

CHAPTER

THIRTEEN

API REFERENCE

The API reference provides detailed descriptions of mosaik's classes and functions. It should be helpful if you plan to extend mosaik with custom components.

13.1 `mosaik` — The end-user API

This module provides convenient access to all classes and functions required to create scenarios and run simulations.

Currently, this is only *`mosaik.scenario.World`*.

13.2 `mosaik.exceptions` — mosaik specific error types

This module provides mosaik specific exception types.

class mosaik.exceptions.**ScenarioError**

This exception is raised if something fails during the creation of a scenario.

class mosaik.exceptions.**SimulationError**(*msg*, *exc=None*)

This exception is raised if a simulator cannot be started or if a problem arises during the execution of a simulation.

13.3 `mosaik.scenario` — Classes related to the scenario creation

This module provides the interface for users to create simulation scenarios for mosaik.

The *`World`* holds all necessary data for the simulation and allows the user to start simulators. It provides a *`ModelFactory`* (and a *`ModelMock`*) via which the user can instantiate model instances (*entities*). The method *`World.run`* finally starts the simulation.

mosaik.scenario.**SimConfig**

Description of all the simulators you intend to use in your simulation.

alias of `Dict`[`str`, `Union`[*`PythonModel`*, *`ConnectModel`*, *`CmdModel`*]]

typeddict mosaik.scenario.**PythonModel**

`typing.TypedDict`.

Required Keys

- **python** (`str`) – The Simulator subclass for this simulator, encoded as a string *module_name:ClassName*.

Optional Keys

- **env** (Dict[str, str]) – The environment variables to set for this simulator.
- **cwd** (str) – The current working directory for this simulator.
- **api_version** (str) – The API version of the connected simulator. Set this to suppress warnings about this simulator being outdated.

typeddict mosaik.scenario.**ConnectModel**

typing.TypedDict.

Required Keys

- **connect** (str) – The *host:port* address for this simulator.

Optional Keys

- **env** (Dict[str, str]) – The environment variables to set for this simulator.
- **cwd** (str) – The current working directory for this simulator.
- **api_version** (str) – The API version of the connected simulator. Set this to suppress warnings about this simulator being outdated.

typeddict mosaik.scenario.**CmdModel**

typing.TypedDict.

Required Keys

- **cmd** (str) – The command to start this simulator. String %(python)s will be replaced by the python command used to start this scenario, %(addr)s will be replaced by the *host:port* combination to which the simulator should connect.

Optional Keys

- **env** (Dict[str, str]) – The environment variables to set for this simulator.
- **cwd** (str) – The current working directory for this simulator.
- **api_version** (str) – The API version of the connected simulator. Set this to suppress warnings about this simulator being outdated.

class mosaik.scenario.**World**(*sim_config*, *mosaik_config=None*, *time_resolution=1.0*, *debug=False*, *cache=True*, *max_loop_iterations=100*, *asyncio_loop=None*)

The world holds all data required to specify and run the scenario.

It provides a method to start a simulator process (*start*) and manages the simulator instances.

You have to provide a *sim_config* which tells the world which simulators are available and how to start them. See *mosaik.simmanager.start* for more details.

mosaik_config can be a dict or list of key-value pairs to set addional parameters overriding the defaults:

```
{
    'addr': ('127.0.0.1', 5555),
    'start_timeout': 2,  # seconds
    'stop_timeout': 2,   # seconds
}
```

Here, *addr* is the network address that mosaik will bind its socket to. *start_timeout* and *stop_timeout* specifiy a timeout (in seconds) for starting/stopping external simulator processes.

If *execution_graph* is set to True, an execution graph will be created during the simulation. This may be useful for debugging and testing. Note, that this increases the memory consumption and simulation time.

Parameters

- **sim_config** (*SimConfig*) –
- **mosaik_config** (*Optional[MosaikConfig]*) –
- **time_resolution** (*float*) –
- **debug** (*bool*) –
- **cache** (*bool*) –
- **max_loop_iterations** (*int*) –
- **asyncio_loop** (*Optional[asyncio.AbstractEventLoop]*) –

until: int

The time until which this simulation will run.

rt_factor: float | None

The number of real-time seconds corresponding to one mosaik step.

sim_config: SimConfig

The config dictionary that tells mosaik how to start a simulator.

config: *MosaikConfigTotal*

The config dictionary for general mosaik settings.

sims: Dict[SimId, *simmanager.SimRunner*]

A dictionary of already started simulators instances.

time_resolution: float

The number of seconds that correspond to one mosaik time step in this situation. The default value is 1.0, meaning that one integer step corresponds to one second simulated time.

max_loop_iterations: int

The number of iterations allowed for same-time loops within one time step. This is checked to prevent accidental infinite loops. Increase this value if your same-time loops require many iterations to converge.

entity_graph: networkx.Graph[FullId]

The `graph` of related entities. Nodes are `(sid, eid)` tuples. Each note has an attribute *entity* with an *Entity*.

sim_progress: float

The progress of the entire simulation (in percent).

start (*sim_name*, *sim_id=None*, ***sim_params*)

Start the simulator named *sim_name* and return a *ModelFactory* for it.

Parameters

- **sim_name** (*str*) –
- **sim_id** (*str | None*) –
- **sim_params** (*Any*) –

Return type

ModelFactory

connect(*src*, *dest*, **attr_pairs*, *async_requests=False*, *time_shifted=False*, *initial_data={}*, *weak=False*)

Connect the *src* entity to *dest* entity.

Establish a data-flow for each `(src_attr, dest_attr)` tuple in *attr_pairs*. If *src_attr* and *dest_attr* have the same name, you you can optionally only pass one of them as a single string.

Raise a *ScenarioError* if both entities share the same simulator instance, if at least one (src. or dest.) attribute in *attr_pairs* does not exist, or if the connection would introduce a cycle in the data-flow (e.g., A → B → C → A).

If the *dest* simulator may make asynchronous requests to mosaik to query data from *src* (or set data to it), *async_requests* should be set to `True` so that the *src* simulator stays in sync with *dest*.

An alternative to asynchronous requests are time-shifted connections. Their data flow is always resolved after normal connections so that cycles in the data-flow can be realized without introducing deadlocks. For such a connection *time_shifted* should be set to `True` and *initial_data* should contain a dict with input data for the first simulation step of the receiving simulator.

An alternative to using async_requests to realize cyclic data-flow is given by the time_shifted kwarg. If set to `True` it marks the connection as cycle-closing (e.g. C → A). It must always be used with initial_data specifying a dict with the data sent to the destination simulator at the first step (e.g. *{'src_attr': value}*).

Parameters

- **src** (*Entity*) –
- **dest** (*Entity*) –
- **attr_pairs** (*str | Tuple[str, str]*) –
- **async_requests** (*bool*) –
- **time_shifted** (*bool | int*) –
- **initial_data** (*Dict[str, Any]*) –
- **weak** (*bool*) –

set_initial_event(*sid*, *time=0*)

Set an initial step for simulator *sid* at time *time* (default=0).

Parameters

- **sid** (*str*) –
- **time** (*int*) –

get_data(*entity_set*, **attributes*)

Get and return the values of all *attributes* for each entity of an *entity_set*.

The return value is a dict mapping the entities of *entity_set* to dicts containing the values of each attribute in *attributes*:

```
{
    Entity(...): {
        'attr_1': 'val_1',
        'attr_2': 'val_2',
        ...
    },
    ...
}
```

Parameters

- **entity_set** (*Iterable[Entity]*) –
- **attributes** (*str*) –

Return type
Dict[Entity, *Dict*[str, *Any*]]

run(*until*, *rt_factor=None*, *rt_strict=False*, *print_progress=True*, *lazy_stepping=True*)

Start the simulation until the simulation time *until* is reached.

Before this method returns, it stops all simulators and closes mosaik's server socket. So this method should only be called once.

Parameters

- **until** (*int*) – The end of the simulation in mosaik time steps (exclusive).
- **rt_factor** (*float | None*) – The real-time factor. If set to a number > 0, the simulation will run in real-time mode. A real-time factor of 1. means that 1 second in simulated time takes 1 second in real time. An real-time factor of 0.5 will let the simulation run twice as fast as real time. For correct behavior of the real-time factor, the time resolution of the scenario has to be set adequately (the default is 1 second).
- **rt_strict** (*bool*) – If the simulators are too slow for the real-time factor you chose, mosaik will only print a warning by default. In order to raise a `RuntimeError` instead, you can set *rt_strict* to `True`.
- **print_progress** (*bool | Literal['individual']*) – Whether progress bars are printed while the simulation is running. The default is to print one bar representing the global progress of the simulation. You can also set the value to `'individual'` to get one bar per simulator in your simulation (in addition to the global one). A value of `False` turns off the progress bars completely.

  The progress bars use tqdm; see their documentation on how to write to the console without interfering with the bars.
- **lazy_stepping** (*bool*) – Whether to prevent simulators from running ahead of their successors by more than one step. If `False` a simulator always steps as long all input is provided. This might decrease the simulation time but increase the memory consumption.

cache_triggering_ancestors()

Collects the ancestors of each simulator and stores them in the respective simulator object.

ensure_no_dataflow_cycles()

Deduce a simulator ranking from a topological sort of the df_graph.

shutdown()

Shut-down all simulators and close the server socket.

class mosaik.scenario.**ModelFactory**(*world*, *group*, *sid*, *proxy*)

This is a facade for a simulator *sim* that allows the user to create new model instances (entities) within that simulator.

For every model that a simulator publicly exposes, the `ModelFactory` provides a *ModelMock* attribute that actually creates the entities.

If you access an attribute that is not a model or if the model is not marked as *public*, an *ScenarioError* is raised.

Parameters

- **world** (`World`) –

- **group** (*SimGroup*) –
- **sid** (*SimId*) –
- **proxy** (*Proxy*) –

class mosaik.scenario.**ModelMock**(*world*, *factory*, *model*, *proxy*)

Instances of this class are exposed as attributes of *ModelFactory* and allow the instantiation of simulator models.

You can *call* an instance of this class to create exactly one entity: `sim.ModelName(x=23)`. Alternatively, you can use the *create* method to create multiple entities with the same set of parameters at once: `sim.ModelName.create(3, x=23)`.

Parameters

- **world** (*World*) –
- **factory** (*ModelFactory*) –
- **model** (*ModelName*) –
- **proxy** (*Proxy*) –

create(*num*, ***model_params*)

Create *num* entities with the specified *model_params* and return a list with the entity dicts.

The returned list of entities is the same as returned by *mosaik_api_v3.Simulator.create*, but the simulator is prepended to every entity ID to make them globally unique.

Parameters

- **num** (*int*) –
- **model_params** (*Any*) –

class mosaik.scenario.**Entity**(*sid*, *eid*, *sim_name*, *model_mock*, *children*, *extra_info=None*)

An entity represents an instance of a simulation model within mosaik.

Parameters

- **sid** (*str*) –
- **eid** (*str*) –
- **sim_name** (*str*) –
- **model_mock** (*ModelMock*) –
- **children** (*List[Entity]*) –
- **extra_info** (*Any*) –

sid: str

The ID of the simulator this entity belongs to.

eid: str

The entity's ID.

sim_name: str

The entity's simulator name.

model_mock: *ModelMock*

The entity's type (or class).

children: List[*Entity*]
An entity set containing subordinate entities.

property full_id: str
Full, globally unique entity id `sid.eid`.

13.4 mosaik.scheduler — Coordinate and execute simulators

This module is responsible for performing the simulation of a scenario.

async mosaik.scheduler.**run**(*world*, *until*, *rt_factor=None*, *rt_strict=False*, *lazy_stepping=True*)
Run the simulation for a *World* until the simulation time *until* has been reached.

Return the final simulation time.

See *mosaik.scenario.World.run* for a detailed description of the *rt_factor* and *rt_strict* arguments.

Parameters

- **world** (*World*) –
- **until** (*int*) –
- **rt_factor** (*Optional[float]*) –
- **rt_strict** (*bool*) –
- **lazy_stepping** (*bool*) –

async mosaik.scheduler.**sim_process**(*world*, *sim*, *until*, *rt_factor*, *rt_strict*, *lazy_stepping*)
Coroutine running the simulator *sim*.

Parameters

- **world** (*World*) –
- **sim** (*SimRunner*) –
- **until** (*int*) –
- **rt_factor** (*Optional[float]*) –
- **rt_strict** (*bool*) –
- **lazy_stepping** (*bool*) –

async mosaik.scheduler.**rt_sleep**(*sim*, *world*)
If in real-time mode, check if to sleep and do so if necessary.

Parameters

- **sim** (*SimRunner*) –
- **world** (*World*) –

Return type
None

async mosaik.scheduler.**wait_for_dependencies**(*sim*, *lazy_stepping*)
Wait until all simulators that can provide input for this simulator have run for this step.

Also notify any simulator that is already waiting to perform its next step.

world is a mosaik *World*.

Parameters

- **sim** (SimRunner) –
- **lazy_stepping** (*bool*) –

Return type
None

mosaik.scheduler.**get_input_data**(*world*, *sim*)

Return a dictionary with the input data for *sim*.

The dict will look like:

```
{
    'eid': {
        'attrname': {'src_eid_0': val_0, ... 'src_eid_n': val_n},
        ...
    },
    ...
}
```

For every entity, there is an entry in the dict and each entry is itself a dict with attributes and a list of values. This is, because we may have inputs from multiple simulators (e.g., different consumers that provide loads for a node in a power grid) and cannot know how to aggregate that data (sum, max, …?).

world is a mosaik *World*.

Parameters

- **world** (World) –
- **sim** (SimRunner) –

Return type
InputData

mosaik.scheduler.**get_max_advance**(*world*, *sim*, *until*)

Checks how far *sim* can safely advance its internal time during next step without causing a causality error.

Parameters

- **world** (World) –
- **sim** (SimRunner) –
- **until** (*int*) –

Return type
int

async mosaik.scheduler.**step**(*world*, *sim*, *inputs*, *max_advance*)

Advance (step) a simulator *sim* with the given *inputs*. Return an event that is triggered when the step was performed.

inputs is a dictionary, that maps entity IDs to data dictionaries which map attribute names to lists of values (see *get_input_data*).

max_advance is the simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Parameters

- **world** (World) –
- **sim** (SimRunner) –

- **inputs** (*InputData*) –
- **max_advance** (*int*) –

mosaik.scheduler.**rt_check**(*rt_factor*, *rt_start*, *rt_strict*, *sim*)

Check if simulation is fast enough for a given real-time factor.

Parameters

- **rt_factor** (*float | None*) –
- **rt_start** (*float*) –
- **rt_strict** (*bool*) –
- **sim** (SimRunner) –

async mosaik.scheduler.**get_outputs**(*world*, *sim*)

Wait for all required output data from a simulator *sim*.

world is a mosaik *World*.

Parameters

- **world** (World) –
- **sim** (SimRunner) –

mosaik.scheduler.**notify_dependencies**(*sim*)

Notify all simulators waiting for us.

Parameters
sim (SimRunner) –

Return type
None

mosaik.scheduler.**prune_dataflow_cache**(*world*)

Prunes the dataflow cache.

Parameters
world (World) –

mosaik.scheduler.**get_progress**(*sims*, *until*)

Return the current progress of the simulation in percent.

Parameters

- **sims** (*Dict[str, SimRunner]*) –
- **until** (*int*) –

Return type
float

mosaik.scheduler.**get_avg_progress**(*sims*, *until*)

Get the average progress of all simulations (in time steps).

Parameters

- **sims** (*Dict[str, SimRunner]*) –
- **until** (*int*) –

Return type
int

13.5 mosaik.simmanager — Management of external processes

The simulation manager is responsible for starting simulation processes and shutting them down. It also manages the communication between mosaik and the processes.

It is able to start pure Python simulators in-process (by importing and instantiating them), to start external simulation processes and to connect to already running simulators and manage access to them.

typeddict mosaik.simmanager.**MosaikConfigTotal**

A total version for **:cls:`MosaikConfig`** for internal use.

Required Keys

- **addr** (Tuple[str, Optional[int]])
- **start_timeout** (float)
- **stop_timeout** (float)

async mosaik.simmanager.**start**(*world*, *sim_name*, *sim_id*, *time_resolution*, *sim_params*)

Start the simulator *sim_name* based on the configuration im *world.sim_config*, give it the ID *sim_id* and pass the time_resolution and the parameters of the dict *sim_params* to it.

The sim config is a dictionary with one entry for every simulator. The entry itself tells mosaik how to start the simulator:

```
{
    'ExampleSimA': {
        'python': 'example_sim.mosaik:ExampleSim',
    },
    'ExampleSimB': {
        'cmd': 'example_sim %(addr)s',
        'cwd': '.',
    },
    'ExampleSimC': {
        'connect': 'host:port',
    },
}
```

ExampleSimA is a pure Python simulator. Mosaik will import the module `example_sim.mosaik` and instantiate the class `ExampleSim` to start the simulator.

ExampleSimB would be started by executing the command *example_sim* and passing the network address of mosaik das command line argument. You can optionally specify a *current working directory*. It defaults to `.`.

ExampleSimC can not be started by mosaik, so mosaik tries to connect to it.

time_resolution (in seconds) is a global scenario parameter, which tells the simulators what the integer time step means in seconds. Its default value is 1., meaning one integer step corresponds to one second simulated time.

The function returns a *mosaik_api_v3.Simulator* instance.

It raises a *SimulationError* if the simulator could not be started.

Return a SimProxy instance.

Parameters

- **world** (World) –
- **sim_name** (*str*) –
- **sim_id** (*SimId*) –

- **time_resolution** (*float*) –
- **sim_params** (*Dict[str, Any]*) –

Return type
Proxy

async mosaik.simmanager.**start_inproc**(*mosaik_config*, *sim_name*, *sim_config*, *mosaik_remote*)

Import and instantiate the Python simulator *sim_name* based on its config entry *sim_config*.

Return a `LocalProcess` instance.

Raise a *ScenarioError* if the simulator cannot be instantiated.

Parameters

- **mosaik_config** (MosaikConfigTotal) –
- **sim_name** (*str*) –
- **sim_config** (*Dict[Literal['python', 'env'], str]*) –
- **mosaik_remote** (MosaikRemote) –

Return type
BaseProxy

async mosaik.simmanager.**start_proc**(*mosaik_config*, *sim_name*, *sim_config*, *mosaik_remote*)

Start a new process for simulator *sim_name* based on its config entry *sim_config*.

Return a `RemoteProcess` instance.

Raise a *ScenarioError* if the simulator cannot be instantiated.

Parameters

- **mosaik_config** (MosaikConfigTotal) –
- **sim_name** (*str*) –
- **sim_config** (*Dict[Literal['cmd', 'cwd', 'env', 'posix', 'new_console'], ~typing.Any]*) –
- **mosaik_remote** (MosaikRemote) –

Return type
BaseProxy

async mosaik.simmanager.**start_connect**(*mosaik_config*, *sim_name*, *sim_config*, *mosaik_remote*)

Connect to the already running simulator *sim_name* based on its config entry *sim_config*.

Return a `RemoteProcess` instance.

Raise a *ScenarioError* if the simulator cannot be instantiated.

Parameters

- **mosaik_config** (MosaikConfigTotal) –
- **sim_name** (*str*) –
- **sim_config** (ConnectModel) –
- **mosaik_remote** (MosaikRemote) –

Return type
BaseProxy

mosaik.simmanager.**Port**

Pair of an entity ID and an attribute of that entity

alias of `Tuple[str, str]`

class mosaik.simmanager.**SimRunner**(*sid*, *connection*, *depth=1*)

Handler for an external simulator.

It stores its simulation state and own the proxy object to the external simulator.

Parameters

- **sid** (*SimId*) –
- **connection** (*Proxy*) –
- **depth** (*int*) –

rt_start: float

The real time when this simulator started (as returned by *perf_counter()*.

output_time: TieredTime

The output time associated with *data*. Usually, this will be equal to *last_step* but simulators may specify a different time for their output.

data: OutputData

The newest data returned by this simulator.

sid: SimId

This simulator's ID.

last_step: TieredTime

The most recent step this simulator performed.

next_steps: List[TieredTime]

The scheduled next steps this simulator will take, organized as a heap. Once the immediate next step has been chosen (and the *has_next_step* event has been triggered), the step is moved to *next_step* instead.

next_self_step: TieredTime | None

The next self-scheduled step for this simulator.

progress: Progress

This simulator's progress in mosaik time.

This simulator has done all its work before time *progress*, so other simulator can rely on this simulator's output until this time.

inputs_from_set_data: InputData

Inputs received via *set_data*.

persistent_inputs: InputData

Memory of previous inputs for persistent attributes.

timed_input_buffer: *TimedInputBuffer*

Inputs for this simulator.

triggering_ancestors: Dict[*SimRunner*, TieredInterval]

An iterable of this sim's ancestors that can trigger a step of this simulator. The second component specifies the least amount of time that output from the ancestor needs to reach us.

triggers: Dict[Port, List[Tuple[*SimRunner*, TieredInterval]]]

For each port of this simulator, the simulators that are triggered by output on that port and the delay accrued along that edge.

output_to_push: Dict[Port, List[Tuple[*SimRunner*, TieredInterval, Port]]]

This lists those connections that use the timed_input_buffer. The keys are the entity-attribute pairs of this simulator with the corresponding list of simulator-time-entity-attribute triples describing the destinations for that data and the time-shift occuring along the connection.

pulled_inputs: Dict[Tuple[*SimRunner*, TieredInterval], Set[Tuple[Port, Port]]]

Output to pull in whenever this simulator performs a step. The keys are the source SimRunner and the time shift, the values are the source and destination entity-attribute pairs.

task: asyncio.Task[None]

The asyncio.Task for this simulator.

input_delays: Dict[*SimRunner*, TieredInterval]

For each simulator that provides data to this simulator, the minimum over all input delays. This is used while waiting for dependencies.

schedule_step(*tiered_time*)

Schedule a step for this simulator at the given time. This will trigger a re-evaluation whether this simulator's next step is settled, provided that the new step is earlier than the old one and the simulator is currently awaiting it's next settled step.

Parameters
tiered_time (*TieredTime*) –

async stop()

Stop the simulator behind the proxy.

class mosaik.simmanager.**MosaikRemote**(*world*, *sid*)

Parameters

- **world** (World) –
- **sid** (*SimId*) –

async get_progress()

Return the current simulation progress from *sim_progress*.

Return type
float

async get_related_entities(*entities=None*)

Return information about the related entities of *entities*.

If *entities* omitted (or None), return the complete entity graph, e.g.:

```
{
    'nodes': {
        'sid_0.eid_0': {'type': 'A'},
        'sid_0.eid_1': {'type': 'B'},
        'sid_1.eid_0': {'type': 'C'},
    },
    'edges': [
        ['sid_0.eid_0', 'sid_1.eid0', {}],
```

(continues on next page)

(continued from previous page)

```
        ['sid_0.eid_1', 'sid_1.eid0', {}],
    ],
}
```

If *entities* is a single string (e.g., `sid_1.eid_0`), return a dict containing all entities related to that entity:

```
{
    'sid_0.eid_0': {'type': 'A'},
    'sid_0.eid_1': {'type': 'B'},
}
```

If *entities* is a list of entity IDs (e.g., `['sid_0.eid_0', 'sid_0.eid_1']`), return a dict mapping each entity to a dict of related entities:

```
{
    'sid_0.eid_0': {
        'sid_1.eid_0': {'type': 'B'},
    },
    'sid_0.eid_1': {
        'sid_1.eid_1': {'type': 'B'},
    },
}
```

Parameters
entities (*str | List[str] | None*) –

Return type
Dict[str, *Any*] | *Dict*[str, *Dict*[str, *Any*]]

async get_data(*attrs*)

Return the data for the requested attributes *attrs*.

attrs is a dict of (fully qualified) entity IDs mapping to lists of attribute names (`{'sid/eid': ['attr1', 'attr2']}`).

The return value is a dictionary, which maps the input entity IDs to data dictionaries, which in turn map attribute names to their respective values: (`{'sid/eid': {'attr1': val1, 'attr2': val2}}`).

Parameters
attrs (*Dict[str, List[str]]*) –

Return type
Dict[str, *Any*]

async set_data(*data*)

Set *data* as input data for all affected simulators.

data is a dictionary mapping source entity IDs to destination entity IDs with dictionaries of attributes and values (`{'src_full_id': {'dest_full_id': {'attr1': 'val1', 'attr2': 'val2'}}}`).

async set_event(*event_time*)

Schedules an event/step at simulation time *event_time*.

Parameters
event_time (*int*) –

class mosaik.simmanager.**StarterCollection**

This class provides a singleton instance of a collection of simulation starters. Default starters are: - python: start_inproc - cmd: start_proc - connect: start_connect

External packages may add additional methods of starting simulations by adding new elements:

from mosaik.simmanager import StarterCollection s = StarterCollection() s['my_starter'] = my_starter_func

Return type
OrderedDict[str, Callable[…, Coroutine[Any, Any, BaseProxy]]]

class mosaik.simmanager.**TimedInputBuffer**

A buffer to store inputs with its corresponding *time*.

When the data is queried for a specific *step* time, all entries with *time* <= *step* are added to the input_dictionary.

If there are several entries for the same connection at the same time, only the most recent value is added.

13.6 mosaik.util — Utility classes and functions

This module contains some utility functions and classes.

mosaik.util.**connect_many_to_one**(*world*, *src_set*, *dest*, **attrs*, *async_requests=False*)

connect each entity in *src_set* to *dest*.

See the *connect* for more details.

mosaik.util.**connect_randomly**(*world*, *src_set*, *dest_set*, **attrs*, *evenly=True*, *max_connects=inf*)

Randomly *connect* the entities from *src_set* to the entities from *dest_set* and return a subset of *dest_set* containing all entities with a connection.

world is an instance of the *World* to which the entities belong.

src_set and *dest_set* are iterables containing *Entity* instances. *src_set* may be empty, *dest_set* must not be empty. Each entity of *src_set* will be connected to an entity of *dest_set*, but not every entity of *dest_set* will necessarily have a connection (e.g., if you connect a set of three entities to a set of four entities). A set of all entities from *dest_set*, to which at least one entity from *src_set* was connected, will be returned.

attrs is a list of attribute names of pairs as in *connect*.

If the flag *evenly* is set to `True`, entities connections will be distributed as evenly as possible. That means if you connect a set of three entities to a set of three entities, there will be three 1:1 connections; if you connect four entities to three entities, there will be one 2:1 and two 1:1 connections. If *evenly* is set to `False`, connections will be truly random. That means if you connect three entities to three entities, you may either have three 1:1 connections, one 2:1 and two 1:1 connections or just one 3:1 connection.

max_connects lets you set the maximum number of connections that an entity of *dest_set* may receive. This argument is only taken into account if *evenly* is set to `False`.

mosaik.util.**plot_execution_time**(*world*, *folder='figures'*, *hdf5path=None*, *dpi=600*, *format='png'*, *show_plot=True*, *slice=None*)

Creates an image visualizing the execution time of the different simulators of a mosaik scenario.

Parameters

- **world** – mosaik world object
- **folder** – folder to store the image (only if no hdf5path is provided)

- **hdf5path** – Path to HDF5 file, which will be used as path for the created image
- **dpi** – DPI for created images
- **format** – format for created image
- **show_plot** – whether to open a window to show the plot
- **slice** – reduce the timeframe that you show in the plot. Usage as in Python list slicing, i.e., negative values are possible to start from the end of the list. Jumps are not possible. `slice` needs to be a two-element integer list, e.g. `[0, 5]`.

Returns
`None` but image file will be written to file system

`mosaik.util.`**plot_dataflow_graph**(*world*, *folder='figures'*, *hdf5path=None*, *dpi=600*, *format='png'*, *show_plot=True*)

Creates an image visualizing the data flow graph of a mosaik scenario. Using the spring layout from Matplotlib (Fruchterman- Reingold force-directed algorithm) to position the nodes.

Parameters

- **world** (*World*) – mosaik world object
- **folder** (*str*) – folder to store the image (only if no hdf5path is provided)
- **hdf5path** (*str | None*) – Path to HDF5 file, which will be used as path for the created image
- **dpi** (*int*) – DPI for created images
- **format** (*Literal['png', 'pdf', 'svg']*) – format for created image
- **show_plot** (*bool*) – whether open a window to show the plot

Returns
`None` but image file will be written to file system

`mosaik.util.`**plot_execution_graph**(*world*, *title=''*, *folder='figures'*, *hdf5path=None*, *dpi=600*, *format='png'*, *show_plot=True*, *save_plot=True*, *slice=None*)

Creates an image visualizing the execution graph of a mosaik scenario.

Parameters

- **world** (*World*) – mosaik world object
- **title** (*str*) – the title of the graph
- **folder** – folder to store the image (only if no hdf5path is provided)
- **hdf5path** – Path to HDF5 file, which will be used as path for the created image
- **dpi** – DPI for created images
- **format** – format for created image
- **show_plot** – whether to open a window to show the plot
- **slice** – reduce the timeframe that you show in the plot. Usage as in Python list slicing, i.e., negative values are possible to start from the end of the list. Jumps are not possible. `slice` needs to be a two-element integer list, e.g. `[0, 5]`.
- **save_plot** (*bool*) –

Returns
`None` but image file will be written to file system

mosaik.util.**plot_execution_time_per_simulator**(*world*, *folder='figures'*, *hdf5path=None*, *dpi=600*, *format='png'*, *show_plot=True*, *plot_per_simulator=False*, *slice=None*)

Creates images visualizing the execution time of each of the different simulators of a mosaik scenario.

Parameters

- **world** – mosaik world object
- **folder** – folder to store the image (only if no hdf5path is provided)
- **hdf5path** – Path to HDF5 file, which will be used as path for the created image
- **dpi** – DPI for created images
- **format** – format for created image
- **show_plot** – whether to open a window to show the plot
- **plot_per_simulator** (*bool*) – whether to create a separated plot per simulator. This is especially useful if the step sizes of the simulators are very different.
- **slice** – reduce the timeframe that you show in the plot. Usage as in Python list slicing, i.e., negative values are possible to start from the end of the list. Jumps are not possible. `slice` needs to be a two-element integer list, e.g. `[0, 5]`.

Returns

`None` but image file will be written to file system

13.7 mosaik.basic_simulators — Basic simulators

class mosaik.basic_simulators.**InputSimulator**

This simulator gives a steady input to a connected simulator. This input can either be an constant value or given by a custom function based on the current time.

When starting the simulator, a custom step size may be provided using the *step_size* parameter. The default is 1.

When creating a *Constant* entity, the constant must be passed as the parameter *constant*.

When creating a *Function* entity, a function should be passed as the *function* parameter. This function should take the current mosaik time and return the desired value. This type of entity only works when the simulator is started using the *"python"* method.

In either case, the entity will produce its output on the *value* attribute.

class mosaik.basic_simulators.**OutputSimulator**

This simulator takes the input it is given and writes it into a python dictionary where the keys are the timestamps of the input and the values are the inputs values.

The dictionary can be retrieved using the *get_dict* method.

get_dict(*eid*)

Returns the dict of the simulator entity specified by the `eid`.

Parameters

eid (*str*) – The entity id of the selected entity.

Returns

The dict of the entity.

Return type
Dict[int, *Any*]

CHAPTER

FOURTEEN

ABOUT MOSAIK

The latest version of mosaik is 3.3.0b1 and it is licensed under LGPL.

Contents:

14.1 Acknowledgments

A lot of people were involved in the creation of mosaik and mosaik wouldn't be what it is today without any of them:

- *Martin Tröschel* and *Astrid Nießе* had the original idea for a tool that lets you integrate existing simulators to perform large-scale Smart Grid simulations. They also accompanied mosaik's development many years as group and project leaders and provided the necessary time and resources for mosaik's development.
- *Steffen Schütte* wrote his PhD around mosaik. He and *Stefan Scherfke* are the primary authors of mosaik 1.
- *Ontje Lünsdorf* not only contributed code to mosaik 1, but also a lot of ideas. He held countless discussions with Steffen and Stefan whose results often greatly improved mosaik.
- *Stefan Scherfke* is the primary author of mosaik 2.0 and 2.1.
- *Sebastian Rohjans* and *Sebastian Lehnhoff* accompanied mosaik's development as group and scientific leaders. They also put lots of effort into making mosaik open-source software.
- *Okko Nannen* and *Florian Schlögl* joined the team in May / July 2014.

We'd also like to thank everyone who worked with mosaik and gave us feedback to make it better.

14.2 The history of mosaik

Our work on mosaik started on July 15th, 2010 – at least, the initial commit happened on that day. Since then, we've come a long way …

14.2.1 3.3.0 - 2024-01-17

- [NEW] A simulator can now be connected to itself if the connection is weak or time_shifted (https://gitlab.com/mosaik/mosaik/-/issues/156)
- [NEW] mosaik now provides a couple of basic simulators (https://gitlab.com/mosaik/mosaik/-/issues/192 , https://mosaik.readthedocs.io/en/latest/ecosystem/index.html#mosaik-basic-simulators)
- [NEW] Simulators can return extra_info for their entities to be used in scenario script (https://gitlab.com/mosaik/mosaik/-/issues/197)
- [IMPROVEMENT] Internal rework and refactoring of the time handling in mosaik (https://gitlab.com/mosaik/mosaik/-/issues/162)

14.2.2 3.2.0 - 2023-08-31

- [NEW] Visualizations for the simulation debug information (https://gitlab.com/mosaik/mosaik/-/issues/173)
- [NEW] Allow to open a new console for simulator (https://gitlab.com/mosaik/mosaik/-/issues/84)
- [FIX] Lift restriction of only one weak connection per cycle (https://gitlab.com/mosaik/mosaik/-/issues/151)
- [FIX] Incorrect triggering when adding several edges (https://gitlab.com/mosaik/mosaik/-/issues/92)
- [IMPROVEMENT] Switch from simpy to asycnio (https://gitlab.com/mosaik/mosaik/-/issues/103)
- [CHANGE] Removed support for Python 3.7, added support for Python 3.11 (https://gitlab.com/mosaik/mosaik/-/issues/171)

14.2.3 3.1.1 - 2023-01-11

- [FIX] Fix compatibilty with mosaik 2 simulators (https://gitlab.com/mosaik/mosaik/-/issues/152)

14.2.4 3.1.0 - 2022-11-23

- [NEW] Add progress bar to visualize simulation progress (https://gitlab.com/mosaik/mosaik/-/merge_requests/58)
- [NEW] Add type annotations (https://gitlab.com/mosaik/mosaik/-/issues/107)
- [NEW] Add proper logging (https://gitlab.com/mosaik/mosaik/-/issues/98)
- [DEPRECATED] Deprecated tags for set_data und async_requests (https://gitlab.com/mosaik/mosaik/-/issues/102)
- [CHANGE] Improved benchmarks with new result table (https://gitlab.com/mosaik/mosaik/-/issues/94)
- [FIX] Unexpected behavior of (time-based) simulators whose output is not used anymore (https://gitlab.com/mosaik/mosaik/-/issues/90)
- [FIX] Lazy stepping does not work (https://gitlab.com/mosaik/mosaik/-/issues/89)
- [FIX] Negative max_advance values in same time loop (https://gitlab.com/mosaik/mosaik/-/issues/82)
- [FIX] Initial data for time-shifted connection for hybrid simulator (https://gitlab.com/mosaik/mosaik/-/issues/81)
- [FIX] Bug related to None value for “dest_sim.next_step” in particular connection structure (https://gitlab.com/mosaik/mosaik/-/issues/80)

14.2.5 3.0.2 - 2022-06-01

- [CHANGE] Updated mosaik-api version to 3.0.2

14.2.6 3.0.1 - 2022-05-02

- [CHANGE] Set external events via highlevel function call
- [FIX] Allow PATCH version to be included in the mosaik-api version format

14.2.7 3.0.0 - 2021-06-07

- This is a major upgrade to improve the discrete-event capabilities. Simulators' steps can now also be triggered by the output of other simulators.
- [NEW] Native support of discrete-event simulations
- [NEW] A global time resolution can be set for the scenario.
- [NEW] Simulators can request steps asynchronously via *set_event()* to react to external events.
- [NEW] Ability to specify output data as non-persistent (i.e. transient)
- [CHANGE] New api 3: - Simulators have now a *type* ('time-based'l'event-based'l'hybrid'). - *time_resolution* is passed as argument of the *init* function. - *max_advance* is passed as argument of the *step* function.
- [CHANGE] Update of the documentation

14.2.8 2.6.1 - 2021-06-04

- [CHANGE] Updated ReadTheDocs to support versioning
- [CHANGE] Updated setup: mosaik-api>=2.3,<3
- [CHANGE] Updated networkx version to 2.5

2.6.0 - 2020-05-08

- [NEW] The print of the simulation progress is now optional and can be disabled via a flag world.run(END, print_progress=False).
- [NEW] Additional starters can now be added via external packages (the standard ones are 'python', 'cmd', and 'connect').

2.5.3 - 2020-04-30

- [FIX] Constrain simpy version to <4.0.0 due to simpy.io incompatibility
- [CHANGE] Updated Odysseus tutorial
- [CHANGE] Eliminated shifted_cache which reduces memory consumption

2.5.2 - 2019-11-01

- [NEW] Special characters are now allowed in path names
- [NEW] Compatible to the new versions of networkx
- [CHANGE] python 3.6, 3.7 and 3.8 are currently supported, python 3.4 and 3.5 not anymore.
- [FIX] Various minor internal changes
- [FIX] Various documentation updates and fixes

2.5.1 - 2018-11-29

- [NEW] When calling the world.start() command for a simulator, users can now set a predefined value for the posix flag (e.g. True) to prevent automatic detection of the operating system. This facilitates the creation of some co-simulation cases across OS (e.g. Windows and Linux).

2.5.0 - 2018-09-05

- [NEW] Connection option "time_shifted" added as alternative to async_requests. This will make creating cyclic data dependencies between simulators more usable since usage of set_data with an API implementation will no longer be needed.

2.4.0 - 2017-12-06

- [NEW] Compatible to the new versions of networkx, simpy and simpy.io
- [CHANGE] python 3.4, 3.5 and 3.6 are currently supported python 3.3 is no longer supported
- [FIX] Various bug fixes

2.3.0 - 2016-04-26

- [NEW] Allow passing environment vars to sup processes
- [FIX] Fixed a bug in the version validation which raised an error when using a floating point for the version

2.2.0 - 2016-02-15

- [NEW] API version 2.2: Added an optional "setup_done()" method.
- [CHANGE] API version validation: The API version is no longer an integer but a "major.minor" string. The *major* part has to math with mosaiks major version. The *minor* part may be lower or equal to mosaik's minor version.
- [FIX] Various minor fixes and stability improvements.
- [FIX] Various docuentation updates and fixes.

2.1 – 2014-10-24

- [NEW] Mosaik can now perform real-time simulations. Before, this functionality needed to be implemented by simulators. Now it's just `World.run(until=x, rt_factor=y)`, where `rt_factor` defines the simulation speed relative to the wall-clock time (issue #24).
- [NEW] Simulators can now expose extra methods via their API that can be called from a mosaik scenario. This allows you to, e.g., store static data in a data base. These extra API methods need to be specified in the simulator's meta data (issue #26).
- [NEW] `util.connect_many_to_one()` helper function.
- [NEW] More and better documentation:
  - Tutorial for integrating simulators, control strategies and for creating scenarios.
  - Sim API description
  - Scenario API description
  - Sim Manager documentation
  - Scheduler documentation
  - Discussion of design decisions
  - Logo, colors, CI
- [NEW] Added `util.sync_call()` which eases calling proxied methods of a simulator synchronously.
- [CHANGE] The *rel* attribute in the entity description returned by *create()* is now optional.
- [CHANGE] Moved proxied methods from `SimProxy` to `SimProxy.proxy` in order to avoid potential name clashes with other attributes.
- [CHANGE] Check a simulator's models and extra API methods for potential name clashes with the built-in API methods.
- [CHANGE] The argument *execution_graph* of `World` was renamed to *debug*. The execution graph now also stores the time after a simulation step (in addition to the time before the step).
- [FIX] issue #22: The asynchronous requests *get_data()* and *set_data()* now check if the `async_requests` flag was set in `World.connect()`.
- [FIX] issue #23: *finalize()* is now called for in-process Python simulators.
- [FIX] issue #27: Dramatically improved simulation performance (30 times as fast in some cases) if simulators use different step sizes (e.g. 1 minute and 1 hour) by improving some internal data structures.

2.0 – 2014-09-22

- [NEW] Updated documentation
- [CHANGE] Separated mosaik's package and API version. The former stays a string with a semantic version number; the later is now a simple integer (issue #17).
- [CHANGE] Start/stop timeout for simulators was raised from 2 to 10 seconds.
- [CHANGE] Updated the mosaik logo. It now uses the flat colors and has some improved icon graphics.
- [CHANGE] Renamed `mosaik.simulator` to `mosaik.scheduler`.
- [CHANGE] `Entity` and the World's entity graph now store their simulator name.
- [FIX] issue #16: Mosaik now always prints the name of the simulator if it closes its socket.

2.0a4 – 2014-07-31

- [NEW] The model meta data may now contain the `any_inputs` which, if set to `True`, allows any attribute to be connected to that model (useful for databases and alike).
- [CHANGE] The dictionary of input values in the API's `step()` call now also contains the source of a particular value. This is also usefull to for databases. This may break existing simulators.
- [CHANGE] "." is now used as separator in full entiy IDs instead of "/" (issue #19).

2.0a3 – 2014-06-26

- [NEW] Hierarchical entities: Entities can now have a list of child entities (issue #14).
- [NEW] The `World` class now has a `get_data()` method that allows you to get data from entities while creating a scenario.
- [NEW] `World.connect(a, b, ('X', 'X'))` can now be simplified to `World.connect(a, b, 'X')`.
- [NEW] Attribute `Entity.full_id` which uniquely identifies an entity: `'<sid>/<eid>'`
- [NEW] Attribute `ModelFactory.meta` which is the meta data dictionary of a simulator.
- [NEW] `World()` now accepts a configuration dictionary which can, e.g., specify the network address for mosaik.
- [NEW] Overview section for the docs
- [NEW] Description of the mosaik API in the docs
- [CHANGE] When you create entities, mosaik checks if the model parameters actually exists and raises an error if not (issue #9).
- [CHANGE] The mosaik API's `init()` function now receives the simulator ID as first argument (issue #15).
- [CHANGE] The behavior of the `get_related_entities()` RPC that simulators can perform has been changed.
- [CHANGE] Various internal improvements
- [FIX] issue #18. Improved the error message if a Python simulator could not be imported.
- [REMOVED] Attribute `Entity.rel`.

2.0a2 – 2014-05-05

- [NEW] Preliminary documentation and installation instructions (https://mosaik.readthedocs.org)
- [NEW] Simulators can now set data to other simulators using the asynchronous request *set_data* (issue #1).
- [NEW] There is now a start timeout for external processes (issue #11).
- [CHANGE] Mosaik now raises an error if a simulator uses the wrong API version (issue #4).
- [CHANGE] Mosaik prints everything to *stdout* instead of using the Python logging module (issue #7).
- [FIX] issue #2. Scheduling now works properly for processes using async. requests. New keyword argument *async_requests* for `World.connect()`.
- [FIX] issue #3. Local (in-process) Simulators can now also perform async. requests to mosaik (*get_progress*, *get_related_entities*, *get_data*, *set_data*).
- [FIX] issue #8. Cleaned up the code a bit.

- [FIX] issue #10. Tests for the sim manager improved.

2.0a1 – 2014-03-26

- Mosaik 2 is a complete rewrite of mosaik 1 in order to improve its maintainability and flexibility. It is still an early alpha version and neither feature complete nor bug free.
- Removed features:
  - The *mosl* DSL (including Eclipse xtext and Java) are now gone. Mosaik now only uses Python.
  - Mosaik now longer has executables but is now used as a library.
  - The platform manager is gone.
  - Mosaik no longer includes a database.
  - Mosaik no longer includes a web UI.
- Mosaik now consists of four core components with the following feature sets:
  - mosaik API
    - The API has bean cleaned up and simplified.
    - Simulators and control strategies share the same API.
    - There are only four calls from mosaik to a simulator: *init*, *create*, *step* and *get_data*.
    - Simulators / processes can make asynchronous requests to mosaik during a step: *get_progress*, *get_related_entities*, *get_data*.
    - ZeroMQ with JSON is replaced by plain network sockets with JSON.
  - Scenarios:
    - Pure Python is now used to describe scenarios. This offers you more flexibility to create complex scenarios.
    - Scenario creation simplified: Start a simulator to get a model factory. Use the factory to create model instances (*entities*). Connect entities. Run simulation.
    - Connection rules are are no based on a primitive *connect* function that only connects two entities with each other. On top of that, any connection strategy can be implemented.
  - Simulation Manager:
    - Simulators written in Python 3 can be executed *in process*.
    - Simulators can be started as external processes.
    - Mosaik can connect to an already running instance of a simulator. This can be used as a replacement for the now gone platform manager.
  - Simulation execution:
    - The simulation is now event-based. No schedule and no synchronization points need to be computed.
    - Simulators can have different and varying step sizes.
- Mosaik ecosystem:
  - A high-level implementation of the mosaik 2 API currently only exists for Python. See https://gitlab.com/mosaik/mosaik-api-python.
  - *mosaik-web* is a simple visualization for mosaik simulations. See https://gitlab.com/mosaik/mosaik-web.

- *mosaik-pypower* is an adapter for the *PYPOWER* load flow analysis library. See https://gitlab.com/mosaik/mosaik-pypower and https://github.com/rwl/PYPOWER.
- *mosaik-csv* and *mosaik-householdsim* are simple demo simulators that you can use to "simulate" CSV data sets and load-profile based households. See https://gitlab.com/mosaik/mosaik-csv and https://gitlab.com/mosaik/mosaik-householdsim.
- There is a repository containing a simple demo scenario for mosaik. See https://gitlab.com/mosaik/mosaik-demo.

1.1 – 2013-10-25

- [NEW] New API for control strategies.
- [NEW] Mosaik can be configured via environment variables.
- [NEW] Various changes and improvements implemented during Steffen's dissertation.

1.0 – 2013-01-25

Mosaik 1 was nearly a complete rewrite of the previous version and already incorporated many of the concepts and features described in Steffen Schütte's Phd thesis.

It used *mosl*, a DSL implemented with Eclipse and xtext, to describe simulators and scenarios. Interprocess communication was done with ZeroMQ and JSON encoded messages.

0.5 – 2011-08-22

This was the first actual version of mosaik that actually worked. However, the simulators we were using at that time were hard coded into the simulation loop and we used XML-RPC to communicate with the simulators.

CHAPTER

FIFTEEN

PRIVACY POLICIES

We welcome you to our website. We would like to inform you about the management of your personal data in accordance with Art. 13 General Data Protection Regulation (GDPR).

Controller

The controller responsible for the described data collection and processing is OFFIS e.V., Escherweg 2, 26121 Oldenburg/Germany.

Usage Data

When you visit our website, the data collected from the use of the website is temporarily stored on our web server for statistical purposes in order to improve the quality of our website. This data set contains:

- the page, from which the data is requested
- the name of the data file,
- the date and time of the query,
- the amount of data transferred,
- the access status (file transmitted, file not found),
- a description of the type of browser used,
- the IP address of the requesting computer shortened to such an extent that no reidentification of any persona data is possible.

The listed usage data is stored anonymously. The legal basis for the processing of this personal data is provided for in Art. 6 para. 1 lit. f GDPR.

Data Transfer to Third Parties

We do not transfer your personal data to third parties.

Cookies

We use cookies on our website. Cookies are small pieces of data that are stored and read in your end-device. A distinction is made between session cookies, which are deleted when you close your browser, and permanent cookies, which are stored even after your visit has expired. Cookies may contain data that enables the recognition of the device being used. However, in some cases cookies only contain information on certain settings which are not personal data.

We use session cookies and permanent cookies on our website. The data is processed in accordance to Art. 6 para. 1 lit. f GDPR and in the interest of optimizing or enabling user guidance and improving our website presence.

Please be aware that you can set your browser to inform you when cookies are being stored or used on the website you are visiting. Thus, any use of cookies is transparent to you. You have the possibility to delete your browser configuration at any time and prevent any use of new cookies. In the event you refuse the use of cookies, please note that our web sites may not be displayed optimally and some functions are then no longer technically available.

Data Security

To avoid unauthorized access to your data, we have implemented technical and organizational measures. We use encryption technologies on our website. Your data will be transferred to our servers and back again via a connection that is protected by a TLS encryption technology. You can recognize that you are browsing on an encryption secured website by the lock-symbol shown in the address bar of your browser and by the address bar starting with https://.

Your Rights as a User

As a website user, the GDPR grants you certain rights when processing your personal data.

1. Right of access (Art. 15 GDPR): You have the right to obtain confirmation as to whether or not personal data concerning you is being processed, and, where that is the case access to the personal data and the information specified in Art. 15 GDPR.
2. Right to rectification and erasure (Art. 16 and 17 GDPR): You have the right to obtain without undue delay the rectification of inaccurate personal data concerning you and, if necessary, the right to have incomplete personal data completed. You also have the right to obtain an erasure of the personal data concerning you without undue delay, if one of the reasons listed in Art. 17 GDPR applies, e.g. if the data is no longer necessary for the intended purpose.
3. Right to restriction of processing (Art. 18 GDPR): If one of the conditions set forth in Art. 18 GDPR applies, you shall have the right to restrict the processing of your data to mere storage, e.g. if you revoke consent, to the processing, for the duration of a possible examination.
4. Right to data portability (Art. 20 GDPR): In certain situations, listed in Art. 20 GDPR, you have the right to receive the personal data concerning you in a structured, common and machine-readable format or demand a transmission of the data to another third party.
5. Right to object (Art. 21 GDPR): If the data is processed pursuant to Art. 6 para. 1 lit. f GDPR (data processing for the purposes of the legitimate interests), you have the right to object to the processing at any time for reasons arising out of your particular situation. We will then no longer process personal data, unless there are demonstrably compelling legitimate grounds for processing, which override the interests, rights and freedoms of the person concerned, or the processing serves the purpose of asserting, exercising or defending legal claims.
6. Right to lodge a complaint with a supervisory authority: Pursuant to Art. 77 GDPR, you have the right to lodge a complaint with a supervisory authority if you consider the processing of the data concerning you infringing data protection regulations. The right to lodge a complaint may be invoked in particular in the Member State of your habitual residence, place of work or the place of the alleged infringement.

Contact Details of the Data Protection Officer

Please contact our data protection officer if you have any further questions, suggestions or wishes regarding data protection:

Dr. Uwe Schläger
datenschutz nord GmbH
Web: www.datenschutz-nord-guppe.de
E-Mail: office@datenschutz-nord.de
Telefon: +49 421 69 66 32 0

CHAPTER

SIXTEEN

LEGALS

Address

OFFIS e. V.
Escherweg 2
26121 Oldenburg
Germany
Phone +49 441 9722-0
Fax +49 441 9722-102
Email: institut [A T] offis.de
Internet: www.offis.de

Board Members

Prof. Dr. Sebastian Lehnhoff (Chairman)
Prof. Dr. techn. Susanne Boll-Westermann
Prof. Dr.-Ing. Andreas Hein

Register Court

Amtsgericht Oldenburg
Registernumber VR 1956

VAT Identification Number

DE 811582102

Responsible in the sense of press law

Dr. Ing. Jürgen Meister (Director)
OFFIS e.V.
Escherweg 2
26121 Oldenburg

Disclaimer

Despite careful control OFFIS assumes no liability for the content of external links. The operators of such a website are solely responsible for its content. At the time of linking the concerned sites were checked for possible violations of law. Illegal contents were not identifiable at that time. A permanent control of the linked pages is not reasonable without specific indications of a violation. Upon notification of violations, OFFIS will remove such links immediately.

CHAPTER

SEVENTEEN

DATENSCHUTZ

Datenschutzerklärung

Wir nehmen den Schutz Ihrer persönlichen Daten sehr ernst. Ihre Daten werden im Rahmen der gesetzlichen Vorschriften geschützt. Personenbezogene Daten werden auf unseren Internetseiten nur im notwendigen Umfang erhoben. In keinem Fall werden die erhobenen Daten verkauft oder aus anderen Gründen an Dritte weitergegeben.

Verantwortlicher

Verantwortlich für die hier erläuterte Datenverarbeitung ist der OFFIS e.V., Escherweg 2, 26121 Oldenburg.

Erhebung und Verarbeitung von Daten

Jeder Zugriff auf eine unserer Internetseiten und jeder Abruf einer auf den Internetseiten hinterlegten Datei werden von gängigen Webserver-Log-Dateien protokolliert. Die Speicherung dient internen systembezogenen und statistischen Zwecken. Protokolliert wird u.a.:

- welche Datei angefordert wurde,
- der Name der Datei,
- das Datum und die Uhrzeit der Anforderung,
- die übertragene Datenmenge,
- der Zugriffsstatus (Datei nicht gefunden, Datei übertragen etc.),
- der Typ des verwendeten Webbrowsers und
- die IP-Adresse des Internetseitenbesuchers

Sämtliche dieser Daten werden ausschließlich anonymisiert gespeichert und ausgewertet. Zu diesem Zweck wird die IP-Adresse des Systems, von dem aus die Internetseite oder Datei angefordert wurde, geeignet anonymisiert. Rechtsgrundlage ist Art. 6 Abs. 1 lit. f DSGVO (Datenschutz-Grundverordnung). Es ist somit weder ein Rückschluss auf eine bestimmte Person möglich, noch erfolgt eine Zusammenführung mit anderen Daten.

Cookies

Darüber hinaus verwenden wir weitere Cookies, um unsere Internetseiten besser auf Ihre Wünsche ausrichten und um statistische Daten über die Nutzung unserer Internetseiten erheben zu können. Durch diese Cookies werden keine Daten erhoben, die einen Rückschluss auf eine bestimmte Person ermöglich. Auch die Installation dieser Cookies wird durch eine entsprechende Browser-Einstellung verhindert. Einmal gesetzte Cookies können Sie jederzeit selbst löschen, indem Sie den entsprechenden Menüpunkt in Ihrem Internet-Browser aufrufen oder die Cookies auf Ihrer Festplatte löschen. Einzelheiten hierzu finden Sie im Hilfemenü Ihres Internet-Browsers. Weitergehende personenbezogene Daten werden nur erfasst, wenn Sie diese Angaben freiwillig, z.B. im Rahmen einer Anfrage oder Registrierung, machen.

Datensicherheit

Um Ihre Daten vor unerwünschten Zugriffen möglichst umfassend zu schützen, treffen wir technische und organisatorische Maßnahmen. Wir setzen auf unseren Seiten ein Verschlüsselungsverfahren ein. Ihre Angaben werden von Ihrem Rechner

zu unserem Server und umgekehrt über das Internet mittels einer TLS-Verschlüsselung übertragen. Sie erkennen dies daran, dass in der Statusleiste Ihres Browsers das Schloss-Symbol geschlossen ist und die Adresszeile mit https:// beginnt. Bitte beachten Sie, dass außerhalb der vorgenannten Rahmenbedingungen, insbesondere bei der Kommunikation per E-Mail die vollständige Datensicherheit von uns naturgemäß nicht gewährleistet werden kann.

Verwendung der Daten Wir beachten den Grundsatz der zweckgebundenen Datenverwendung und erheben, verarbeiten und speichern Ihre personenbezogenen Daten nur für die Zwecke, für welche Sie uns diese mitgeteilt haben und für die technische Administration. Eine Weitergabe Ihrer persönlichen Daten an Dritte erfolgt ohne Ihre ausdrückliche Einwilligung nicht, sofern dies nicht zur Erbringung der Dienstleistung oder zur Vertragsdurchführung notwendig ist. Auch die Übermittlung an auskunftsberechtigte staatliche Institution und Behörden erfolgt nur im Rahmen der gesetzlichen Auskunftspflichten oder wenn wir durch eine gerichtliche Entscheidung zur Auskunft verpflichtet werden.

Löschung der Daten

Eine Löschung der gespeicherten personenbezogenen Daten erfolgt, wenn Sie Ihre Einwilligung zur Speicherung widerrufen, wenn deren Kenntnis zur Erfüllung des mit der Speicherung verfolgten Zwecks nicht mehr erforderlich ist oder wenn deren Speicherung aus sonstigen gesetzlichen Gründen unzulässig ist.

Ihre Rechte als Nutzer

Bei Verarbeitung Ihrer personenbezogenen Daten gewährt die DSGVO Ihnen als Webseitennutzer bestimmte Rechte:

1. Auskunftsrecht (Art. 15 DSGVO): Sie haben das Recht eine Bestätigung darüber zu verlangen, ob sie betreffende personenbezogene Daten verarbeitet werden; ist dies der Fall, so haben Sie ein Recht auf Auskunft über diese personenbezogenen Daten und auf die in Art. 15 DSGVO im einzelnen aufgeführten Informationen.
2. Recht auf Berichtigung und Löschung (Art. 16 und 17 DSGVO): Sie haben das Recht, unverzüglich die Berichtigung sie betreffender unrichtiger personenbezogener Daten und ggf. die Vervollständigung unvollständiger perso-nenbezogener Daten zu verlangen. Sie haben zudem das Recht, zu verlangen, dass sie betreffende personenbezogene Daten unverzüglich gelöscht werden, sofern einer der in Art. 17 DSGVO im einzelnen aufgeführten Gründe zutrifft, z. B. wenn die Daten für die verfolgten Zwecke nicht mehr benötigt werden.
3. Recht auf Einschränkung der Verarbeitung (Art. 18 DSGVO): Sie haben das Recht, die Einschränkung der Verarbeitung zu verlangen, wenn eine der in Art. 18 DSGVO aufgeführten Voraussetzungen gegeben ist, z. B. wenn Sie Widerspruch gegen die Verarbeitung eingelegt haben, für die Dauer einer etwai-gen Prüfung.
4. Recht auf Datenübertragbarkeit (Art. 20 DSGVO): In bestimmten Fällen, die in Art. 20 DSGVO im Einzelnen aufgeführt werden, haben Sie das Recht, die sie betreffenden personenbezogenen Daten in einem struktu-rierten, gängigen und maschinenlesbaren Format zu erhalten bzw. die Übermittlung dieser Daten an einen Dritten zu verlangen.
5. Widerspruchsrecht (Art. 21 DSGVO): Werden Daten auf Grundlage von Art. 6 Abs. 1 lit. f erhoben (Datenverarbeitung zur Wahrung berechtigter Interessen), steht Ihnen das Recht zu, aus Gründen, die sich aus Ihrer besonderen Situation ergeben, jederzeit gegen die Verarbeitung Wider-spruch einzulegen. Wir verarbeiten die personenbezogenen Daten dann nicht mehr, es sei denn, es liegen nachweisbar zwingende schutzwürdige Gründe für die Verarbeitung vor, die die Interessen, Rechte und Freiheiten der betroffenen Person überwiegen, oder die Verarbeitung dient der Geltendmachung, Ausübung oder Verteidigung von Rechtsansprüchen.
6. Beschwerderecht bei einer Aufsichtsbehörde Sie haben gem. Art. 77 DSGVO das Recht auf Beschwerde bei einer Aufsichtsbehör-de, wenn Sie der Ansicht sind, dass die Verarbeitung der Sie betreffenden Daten gegen datenschutzrechtliche Bestimmungen verstößt. Das Beschwerderecht kann insbesondere bei einer Aufsichtsbehörde in dem Mitgliedstaat Ihres Aufenthaltsorts, Ihres Arbeitsplatzes oder des Orts des mutmaßlichen Verstoßes geltend gemacht werden.

Datenschutzbeauftragter

Dr. Uwe Schläger
datenschutz nord GmbH
www.datenschutz-nord.de

office(at)datenschutz-nord.de

CHAPTER

EIGHTEEN

IMPRESSUM

Anschrift

OFFIS e. V.
Escherweg 2
26121 Oldenburg
Telefon +49 441 9722-0
Fax +49 441 9722-102
E-Mail: institut [A T] offis.de
Internet: www.offis.de

Vertretungsberechtigter Vorstand

Prof. Dr. Sebastian Lehnhoff (Vorsitzender)
Prof. Dr. techn. Susanne Boll-Westermann
Prof. Dr.-Ing. Andreas Hein

Registergericht

Amtsgericht Oldenburg
Registernummer VR 1956

Umsatzsteuer-Identifikationsnummer (USt-IdNr.)

DE 811582102

Verantwortlich im Sinne der Presse

Dr. Ing. Jürgen Meister (Bereichsleiter)
OFFIS e.V.
Escherweg 2
26121 Oldenburg

Datenschutz

Mehr zum Thema Datenschutz finden Sie *hier*.

CHAPTER

NINETEEN

GLOSSARY

Control strategy
A program that is intended to observe and manipulate the state of objects (simulated or real) of a power system or those that are somehow connected to the power system; for example a multi-agent system that controls the feed-in of decentralized producers.

Co-simulation
In co-simulation the different subsystems which form a coupled problem are modeled and simulated in a distributed manner. The modeling is done on the subsystem level without having the coupled problem in mind. The coupled simulation is carried out by running the subsystems in a black-box manner. During the simulation the subsystems will exchange data. (source: Wikipedia)

Data-flow
The exchange of data between two *simulators* or between the *entities* of two simulators.

Example: the (re)active power feed-in of a PV model that is sent to a node of a power system simulator.

Entity
Represents an instance of a *Model* within a mosaik *simulation*. Entities can be connected to establish a data-flow between them. Examples are the nodes and lines of a power grid or single electric vehicles.

Entity Set
A set or list of *entities*.

Framework
A software framework provides generic functionality that can be selectively changed and expanded by additional user-written code.

Model
A Model is a simplified representation of a real world object or system. It reproduces the relevant aspects of that object or system for its systematic analysis.

Scenario
Description of the system to be simulated. It includes the used *models* and their relations. It includes the state of the models and their data base. In the mosaik-context it includes also the *simulators*.

Simulation
The process of executing a scenario (and the simulation models).

Simulation Model
The representation of a *model* in programming code..

Simulator
A program that contains the implementation of one or more *simulation models* and is able to execute these models (that is, to perform a *simulation*).

Sometimes, the term *simulator* also refers all kinds of processes that can talk to mosaik, including actual simulators, control strategies, visualization servers, database adapters and so on.

Smart Grid

An electric power system that utilizes information exchange and control technologies, distributed computing and associated sensors and actuators, for purposes such as:

- to integrate the behaviour and actions of the network users and other stakeholders,
- to efficiently deliver sustainable, economic and secure electricity supplies.

(source: IEC)

Step

Mosaik executes simulators in discrete time steps. The step size of a time-based simulator can be an arbitrary integer. It can also vary during the simulation. Event-based simulators are stepped whenever its inputs are updated (by other simulators). They can also schedule steps for themselves.

Mosaik uses integers for the representation of time (to avoid rounding errors etc.). It's unit (i.e. to how many seconds one integer step corresponds) can be defined in the scenario, and is passed to every simulation component via the *init function* as key-word parameter *time_resolution. It's a floating point number and defaults to *1.*.

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

mosaik, 183
mosaik.exceptions, 183
mosaik.scenario, 183
mosaik.scheduler, 189
mosaik.simmanager, 192
mosaik.util, 197
mosaik_api_v3, 136
mosaik_components.heatpump.controller.controller, 34
mosaik_components.heatpump.Heat_Pump_Design, 46
mosaik_components.heatpump.Heat_Pump_Model, 22
mosaik_components.heatpump.hotwatertank.hotwatertank, 27

INDEX

Symbols

_design_hp() (*mosaik_components.heat-pump.Heat_Pump_Design.Heat_Pump_Design method*), 46
_etas_heatload_id() (*mosaik_components.heat-pump.Heat_Pump_Design.Heat_Pump_Design method*), 46
_take_closest() (*mosaik_components.heat-pump.Heat_Pump_Design.Heat_Pump_Design method*), 46

C

cache_triggering_ancestors() (*mosaik.scenario.World method*), 187
calc_dhw_supply() (*mosaik_components.heat-pump.controller.controller.Controller method*), 34
calc_sh_supply() (*mosaik_components.heat-pump.controller.controller.Controller method*), 34
children (*mosaik.scenario.Entity attribute*), 188
CmdModel (*typeddict in mosaik.scenario*), 184
Co-simulation, **219**
cond_in_T (*mosaik_components.heat-pump.Heat_Pump_Model.Heat_Pump_Inputs attribute*), 24
cond_in_T (*mosaik_components.heat-pump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
cond_m (*mosaik_components.heat-pump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
cond_m_neg (*mosaik_components.heat-pump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
config (*mosaik.scenario.World attribute*), 185
configure() (*mosaik_api_v3.Simulator method*), 139
connect() (*mosaik.scenario.World method*), 185
connect_many_to_one() (*in module mosaik.util*), 197
connect_randomly() (*in module mosaik.util*), 197
Connection (*class in mosaik_components.heat-pump.hotwatertank.hotwatertank*), 30
ConnectModel (*typeddict in mosaik.scenario*), 184
cons_T (*mosaik_components.heat-pump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
Control strategy, **219**
Controller (*class in mosaik_components.heat-pump.controller.controller*), 34
COP (*mosaik_components.heat-pump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
create() (*mosaik.scenario.ModelMock method*), 188
create() (*mosaik_api_v3.Simulator method*), 137

D

data (*mosaik.simmanager.SimRunner attribute*), 194
Data-flow, **219**
design (*mosaik_components.heat-pump.Heat_Pump_Model.Heat_Pump attribute*), 23

E

eid (*mosaik.scenario.Entity attribute*), 188
ensure_no_dataflow_cycles() (*mosaik.scenario.World method*), 187
Entity, **219**
Entity (*class in mosaik.scenario*), 188
Entity Set, **219**
entity_graph (*mosaik.scenario.World attribute*), 185

F

finalize() (*mosaik_api_v3.Simulator method*), 140
Framework, **219**
full_id (*mosaik.scenario.Entity property*), 189

G

get_avg_progress() (*in module mosaik.scheduler*), 191
get_data() (*mosaik.scenario.World method*), 186
get_data() (*mosaik.simmanager.MosaikRemote method*), 196

get_data() (*mosaik_api_v3.Simulator method*), 139
get_dict() (*mosaik.basic_simulators.OutputSimulator method*), 199
get_input_data() (*in module mosaik.scheduler*), 190
get_max_advance() (*in module mosaik.scheduler*), 190
get_outputs() (*in module mosaik.scheduler*), 191
get_progress() (*in module mosaik.scheduler*), 191
get_progress() (*mosaik.simmanager.MosaikRemote method*), 195
get_related_entities() (*mosaik.simmanager.MosaikRemote method*), 195

H

Heat_Pump (*class in mosaik_components.heatpump.Heat_Pump_Model*), 22
Heat_Pump_Design (*class in mosaik_components.heatpump.Heat_Pump_Design*), 46
Heat_Pump_Inputs (*class in mosaik_components.heatpump.Heat_Pump_Model*), 23
Heat_Pump_State (*class in mosaik_components.heatpump.Heat_Pump_Model*), 24
heat_source_T (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_Inputs attribute*), 23
heat_source_T (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
HeatingRod (*class in mosaik_components.heatpump.hotwatertank.hotwatertank*), 30
HotWaterTank (*class in mosaik_components.heatpump.hotwatertank.hotwatertank*), 27

I

init() (*mosaik_api_v3.Simulator method*), 137
input_delays (*mosaik.simmanager.SimRunner attribute*), 195
inputs (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump attribute*), 23
inputs_from_set_data (*mosaik.simmanager.SimRunner attribute*), 194
InputSimulator (*class in mosaik.basic_simulators*), 199

L

last_step (*mosaik.simmanager.SimRunner attribute*), 194
Layer (*class in mosaik_components.heatpump.hotwatertank.hotwatertank*), 29

M

MassFlow (*class in mosaik_components.heatpump.hotwatertank.hotwatertank*), 30
max_loop_iterations (*mosaik.scenario.World attribute*), 185
meta (*mosaik_api_v3.Simulator attribute*), 136
Model, **219**
model_mock (*mosaik.scenario.Entity attribute*), 188
ModelFactory (*class in mosaik.scenario*), 187
ModelMock (*class in mosaik.scenario*), 188
module
    mosaik, 183
    mosaik.exceptions, 183
    mosaik.scenario, 183
    mosaik.scheduler, 189
    mosaik.simmanager, 192
    mosaik.util, 197
    mosaik_api_v3, 136
    mosaik_components.heatpump.controller.controller, 34
    mosaik_components.heatpump.Heat_Pump_Design, 46
    mosaik_components.heatpump.Heat_Pump_Model, 22
    mosaik_components.heatpump.hotwatertank.hotwatertank, 27
mosaik
    module, 183
mosaik (*mosaik_api_v3.Simulator attribute*), 137
mosaik.exceptions
    module, 183
mosaik.scenario
    module, 183
mosaik.scheduler
    module, 189
mosaik.simmanager
    module, 192
mosaik.util
    module, 197
mosaik_api_v3
    module, 136
mosaik_components.heatpump.controller.controller
    module, 34
mosaik_components.heatpump.Heat_Pump_Design
    module, 46
mosaik_components.heatpump.Heat_Pump_Model
    module, 22
mosaik_components.heatpump.hotwatertank.hotwatertank
    module, 27
MosaikConfigTotal (*typeddict in mosaik.simmanager*), 192
MosaikRemote (*class in mosaik.simmanager*), 195

N

next_self_step (*mosaik.simmanager.SimRunner attribute*), 194
next_steps (*mosaik.simmanager.SimRunner attribute*), 194
notify_dependencies() (*in module mosaik.scheduler*), 191

O

output_time (*mosaik.simmanager.SimRunner attribute*), 194
output_to_push (*mosaik.simmanager.SimRunner attribute*), 195
OutputSimulator (*class in mosaik.basic_simulators*), 199

P

p_cop_calc() (*mosaik_components.heatpump.Heat_Pump_Design.Heat_Pump_Design method*), 47
P_Required (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
persistent_inputs (*mosaik.simmanager.SimRunner attribute*), 194
plot_dataflow_graph() (*in module mosaik.util*), 198
plot_execution_graph() (*in module mosaik.util*), 198
plot_execution_time() (*in module mosaik.util*), 197
plot_execution_time_per_simulator() (*in module mosaik.util*), 198
Port (*in module mosaik.simmanager*), 193
progress (*mosaik.simmanager.SimRunner attribute*), 194
prune_dataflow_cache() (*in module mosaik.scheduler*), 191
pulled_inputs (*mosaik.simmanager.SimRunner attribute*), 195
PythonModel (*typeddict in mosaik.scenario*), 183

Q

Q_Demand (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_Inputs attribute*), 23
Q_Demand (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
Q_evap (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
Q_Supplied (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_State attribute*), 24

R

rt_check() (*in module mosaik.scheduler*), 191
rt_factor (*mosaik.scenario.World attribute*), 185
rt_sleep() (*in module mosaik.scheduler*), 189
rt_start (*mosaik.simmanager.SimRunner attribute*), 194
run() (*in module mosaik.scheduler*), 189
run() (*mosaik.scenario.World method*), 187

S

Scenario, **219**
ScenarioError (*class in mosaik.exceptions*), 183
schedule_step() (*mosaik.simmanager.SimRunner method*), 195
Sensor (*class in mosaik_components.heatpump.hotwatertank.hotwatertank*), 29
set_data() (*mosaik.simmanager.MosaikRemote method*), 196
set_event() (*mosaik.simmanager.MosaikRemote method*), 196
set_initial_event() (*mosaik.scenario.World method*), 186
setup_done() (*mosaik_api_v3.Simulator method*), 138
shutdown() (*mosaik.scenario.World method*), 187
sid (*mosaik.scenario.Entity attribute*), 188
sid (*mosaik.simmanager.SimRunner attribute*), 194
sim_config (*mosaik.scenario.World attribute*), 185
sim_name (*mosaik.scenario.Entity attribute*), 188
sim_process() (*in module mosaik.scheduler*), 189
sim_progress (*mosaik.scenario.World attribute*), 185
SimConfig (*in module mosaik.scenario*), 183
SimRunner (*class in mosaik.simmanager*), 194
sims (*mosaik.scenario.World attribute*), 185
Simulation, **219**
Simulation Model, **219**
SimulationError (*class in mosaik.exceptions*), 183
Simulator, **219**
Simulator (*class in mosaik_api_v3*), 136
Smart Grid, **220**
snapshot (*mosaik_components.heatpump.hotwatertank.hotwatertank.HotWaterTank property*), 29
snapshot_connections (*mosaik_components.heatpump.hotwatertank.hotwatertank.HotWaterTank property*), 29
start() (*in module mosaik.simmanager*), 192
start() (*mosaik.scenario.World method*), 185
start_connect() (*in module mosaik.simmanager*), 193
start_inproc() (*in module mosaik.simmanager*), 193

start_proc() (*in module mosaik.simmanager*), 193
start_simulation() (*in module mosaik_api_v3*), 141
StarterCollection (*class in mosaik.simmanager*), 196
state (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump attribute*), 23
Step, **220**
step() (*in module mosaik.scheduler*), 190
step() (*mosaik_api_v3.Simulator method*), 138
step() (*mosaik_components.heatpump.controller.controller.Controller method*), 34
step() (*mosaik_components.heatpump.Heat_Pump_Design.Heat_Pump_Design method*), 47
step() (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump method*), 23
step() (*mosaik_components.heatpump.hotwatertank.hotwatertank.HotWaterTank method*), 29
step_error() (*mosaik_components.heatpump.Heat_Pump_Design.Heat_Pump_Design method*), 47
step_executed (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
step_size (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_Inputs attribute*), 24
stop() (*mosaik.simmanager.SimRunner method*), 195

T

T_amb (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_Inputs attribute*), 23
T_amb (*mosaik_components.heatpump.Heat_Pump_Model.Heat_Pump_State attribute*), 24
T_mean (*mosaik_components.heatpump.hotwatertank.hotwatertank.HotWaterTank property*), 29
task (*mosaik.simmanager.SimRunner attribute*), 195
time_resolution (*mosaik.scenario.World attribute*), 185
timed_input_buffer (*mosaik.simmanager.SimRunner attribute*), 194
TimedInputBuffer (*class in mosaik.simmanager*), 197
triggering_ancestors (*mosaik.simmanager.SimRunner attribute*), 194
triggers (*mosaik.simmanager.SimRunner attribute*), 194

U

until (*mosaik.scenario.World attribute*), 185

W

wait_for_dependencies() (*in module mosaik.scheduler*), 189
World (*class in mosaik.scenario*), 184