mosaik Documentation

Release 3.1.1

Stefan Scherfke

Sep 01, 2023

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 Modular Design **15**
- 4.1 Odysseus . . . 16

5 Tutorials **19**
- 5.1 Integrating a simulation model into the mosaik ecosystem . . . 19
- 5.2 Creating and running simple simulation scenarios . . . 27
- 5.3 Adding a control mechanism to a scenario . . . 32
- 5.4 Integrating a control mechanism . . . 37
- 5.5 Same-time loops . . . 39
- 5.6 Set external events . . . 44
- 5.7 Connecting mosaik and Odysseus . . . 49
- 5.8 Using Odysseus to process, visualize and store simulation data . . . 52
- 5.9 Integrating a Model in Java . . . 54

6 The mosaik API **61**
- 6.1 How mosaik communicates with a simulator . . . 62
- 6.2 The low-level API . . . 65
- 6.3 The high-level API . . . 76

7 Scenario definition **83**
- 7.1 The setup . . . 83
- 7.2 Starting simulators . . . 84
- 7.3 Instantiating simulation models . . . 85
- 7.4 Setting initial events . . . 85
- 7.5 Connecting entities . . . 86
- 7.6 Running the simulation . . . 86
- 7.7 How to achieve cyclic data-flows . . . 87
- 7.8 How to filter entity sets . . . 88
- 7.9 How to create user-defined connection rules . . . 89
- 7.10 How to retrieve static data from entities . . . 90

7.11 How to access topology and data-flow information . . . 90
7.12 How to destroy a world . . . 92
7.13 How to do real-time simulations . . . 92
7.14 How to call extra methods of a simulator . . . 92

8 The simulator manager **93**

9 Scheduling and simulation execution **97**
9.1 Time paradigms . . . 97
9.2 Advancing through time . . . 97
9.3 Synchronization and data-flows . . . 99
9.4 Cyclic data-flows . . . 101
9.5 Stepping and simulation duration . . . 103

10 Upgrading from mosaik 2 to 3 **107**
10.1 Component's type . . . 107
10.2 Time resolution . . . 107
10.3 Max_advance time . . . 107
10.4 Sticking with mosaik 2 . . . 107

11 FAQ **109**
11.1 General questions . . . 109
11.2 Coupling models with mosaik . . . 109
11.3 The mosaik demo . . . 110

12 Developer's Documentation **111**
12.1 Discussion of design decisions . . . 111
12.2 Logo and corporate identity . . . 113

13 API Reference **117**
13.1 `mosaik` — The ende-user API . . . 117
13.2 `mosaik.exceptions` — mosaik specific error types . . . 117
13.3 `mosaik.scenario` — Classes related to the scenario creation . . . 117
13.4 `mosaik.scheduler` — Coordinate and execute simulators . . . 121
13.5 `mosaik.simmanager` — Management of external processes . . . 122
13.6 `mosaik.util` — Utility classes and functions . . . 126

14 About mosaik **127**
14.1 Acknowledgments . . . 127
14.2 The history of mosaik . . . 127

15 Privacy Policies **135**

16 Legals **137**

17 Datenschutz **139**

18 Impressum **143**

19 Glossary **145**

20 Indices and tables **147**

Python Module Index **149**

Index **151**

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 3.6 or higher. To install everything, you need the package manager Pip which is bundled with Python 3.6 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.

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 >= 3.6, 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 >= 3.6. 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@macbook:~$`.

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 >= 3.6. 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 \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 C:\Python34\python.exe Envs\mosaik
   C:\Users\yourname> Envs\mosaik\Scripts\activate.bat
   ```

   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.bat
   ```

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.

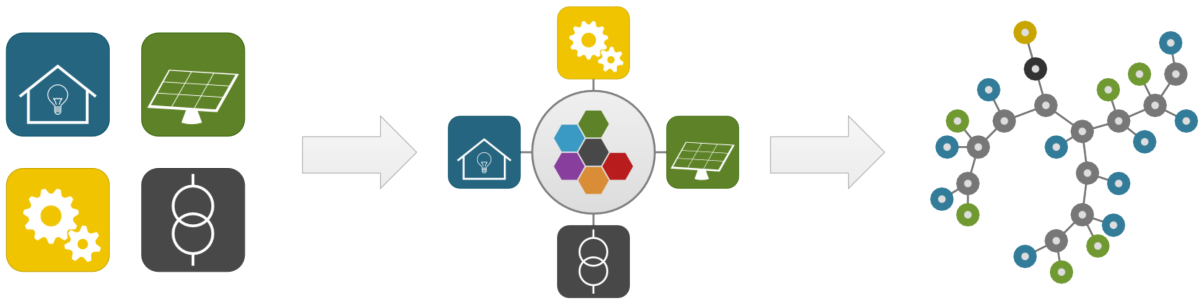
That means that all simulators (or other tools and hardware-in-the-loop) involved in a simulation usually run in their own process. Mosaik just tries to synchronize these processes and manages the exchange of data between them.

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 library SimPy 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

MODULAR DESIGN

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`.

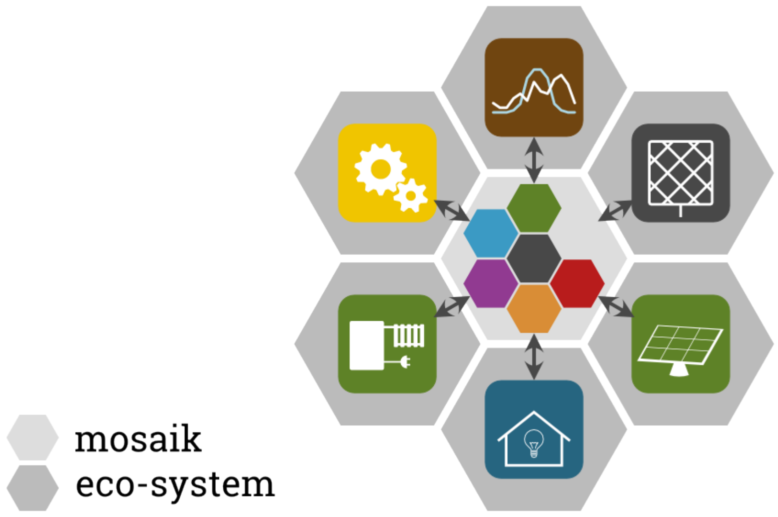


Fig. 1: Mosaik is a co-simulation library. The components and tools form the mosaik ecosystem.

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`.

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`-library.

Elements of the mosaik ecosystem:

`mosaik-tools`

4.1 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.

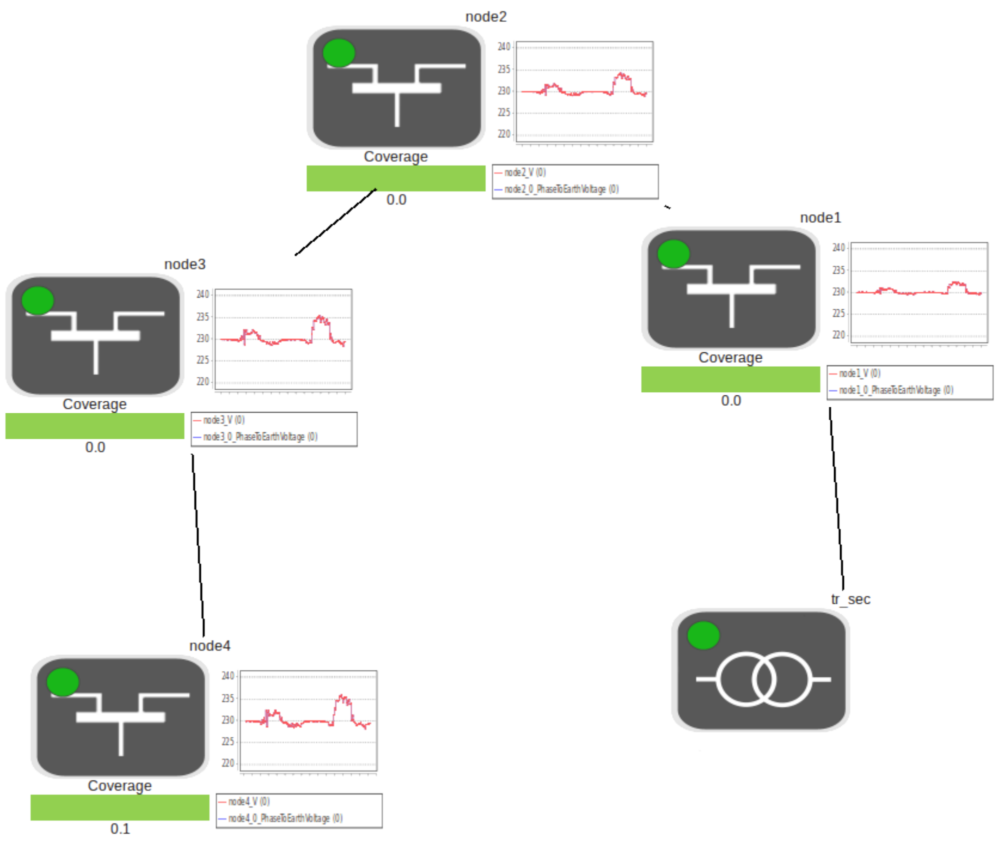
4.1.1 Process data

Odysseus allows to process data streams with different operators. For instance, the operators SELECT and PROJECT can be used to filter the data, aggregations for specific periods of time can be calculated and key-value data can be transformed in relational data and the other way round. Multiple operators are connected with each other, to form so-called operator-graphs. An example of such a graph is shown below:

The query for this graph can be found in the tutorial.

4.1.2 Visualise data

All data can be visualised in lists, tables or graphs. For more complex visualisation, dashboards can be created and individually customised:



4.1.3 Store data

Odysseus also offers connectors to transform data to different formats and databases, which makes it suitable to store simulation data.

Further information about how to install Odysseus and how to use it with mosaik can be found in the tutorial.

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 $i \; \mathbf{N}$, $i > 0$, $delta \; \mathbf{Z}$

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:

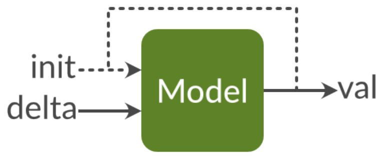


Fig. 1: Schematic diagram of our example model. You can change the *delta* and collect the *val* as output.

```python
# 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:

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

"""
import mosaik_api

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):

```
META = {
    'type': 'time-based',
    'models': {
        'ExampleModel': {
            'public': True,
            'params': ['init_val'],
            'attrs': ['delta', 'val'],
        },
    },
}
```

In this case we create a *time-based* simulator. 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` defines a base class `Simulator` for which we now need to write a sub-class:

```
class ExampleSim(mosaik_api.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.

```
    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` 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.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

import example_model


META = {
    'type': 'time-based',
    'models': {
        'ExampleModel': {
            'public': True,
            'params': ['init_val'],
            'attrs': ['delta', 'val'],
        },
    },
}
```

(continues on next page)

(continued from previous page)

```python
class ExampleSim(mosaik_api.Simulator):
    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)
```

(continues on next page)

(continued from previous page)

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

        return data


def main():
    return mosaik_api.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. and other parameters
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:

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

"""
import collections

import mosaik_api


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


class Collector(mosaik_api.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
```

(continues on next page)

(continued from previous page)

```
    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.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:

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

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:

```
import mosaik
import mosaik.util
# 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" ...
```

(continues on next page)

(continued from previous page)

```
INFO:mosaik_api: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

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:

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


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

# Create World
world = mosaik.World(SIM_CONFIG)

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

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

# Connect entities
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')

```

(continues on next page)

(continued from previous page)

```
# 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` package and defining the simulator meta data:

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

"""
import mosaik_api


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.Simulator`:

```
class Controller(mosaik_api.Simulator):
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        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}},
```

(continues on next page)

(continued from previous page)

```
    '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)
```

(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
```

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

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

"""
import mosaik_api


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


class Controller(mosaik_api.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]}
```

(continues on next page)

(continued from previous page)

```
                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)
                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.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. and other parameters
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
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:

```
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: {4: -1}
- ExampleCtrl-0.Agent_1:
  - delta: {2: -1, 8: 1}
- ExampleCtrl-0.Agent_2:
  - delta: {0: -1, 6: 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: -1, 1: 0, 2: 1, 3: 2, 4: 3, 5: 2, 6: 1, 7: 0, 8: -1, 9: -2}
- 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: 1, 1: 2, 2: 3, 3: 2, 4: 1, 5: 0, 6: -1, 7: -2, 8: -3, 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: 3, 1: 2, 2: 1, 3: 0, 4: -1, 5: -2, 6: -3, 7: -2, 8: -1, 9: 0}
```

This is the complete scenario:

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


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

```

(continues on next page)

(continued from previous page)

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

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

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

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

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'],
        },
    },
}
```

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

        if 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': {
```

(continues on next page)

(continued from previous page)

```
        '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
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)

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: {4: 0, 5: 0}
- ExampleCtrl-0.Agent_1:
  - delta: {2: -1, 4: 0, 5: 0}
- ExampleCtrl-0.Agent_2:
```

(continues on next page)

(continued from previous page)

```
  - delta: {4: 0, 5: 0}
- ExampleMasterCtrl-0.Master_Agent_0:
  - delta_out: {4: 0, 5: 0}
- ExampleSim-0.Model_0:
  - delta: {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 0}
  - val: {0: -1, 1: 0, 2: 1, 3: 2, 4: 3, 5: 3}
- ExampleSim-0.Model_1:
  - delta: {0: 1, 1: 1, 2: 1, 3: -1, 4: -1, 5: 0}
  - val: {0: 1, 1: 2, 2: 3, 3: 2, 4: 1, 5: 1}
- ExampleSim-0.Model_2:
  - delta: {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 0}
  - val: {0: -1, 1: 0, 2: 1, 3: 2, 4: 3, 5: 3}
```

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.

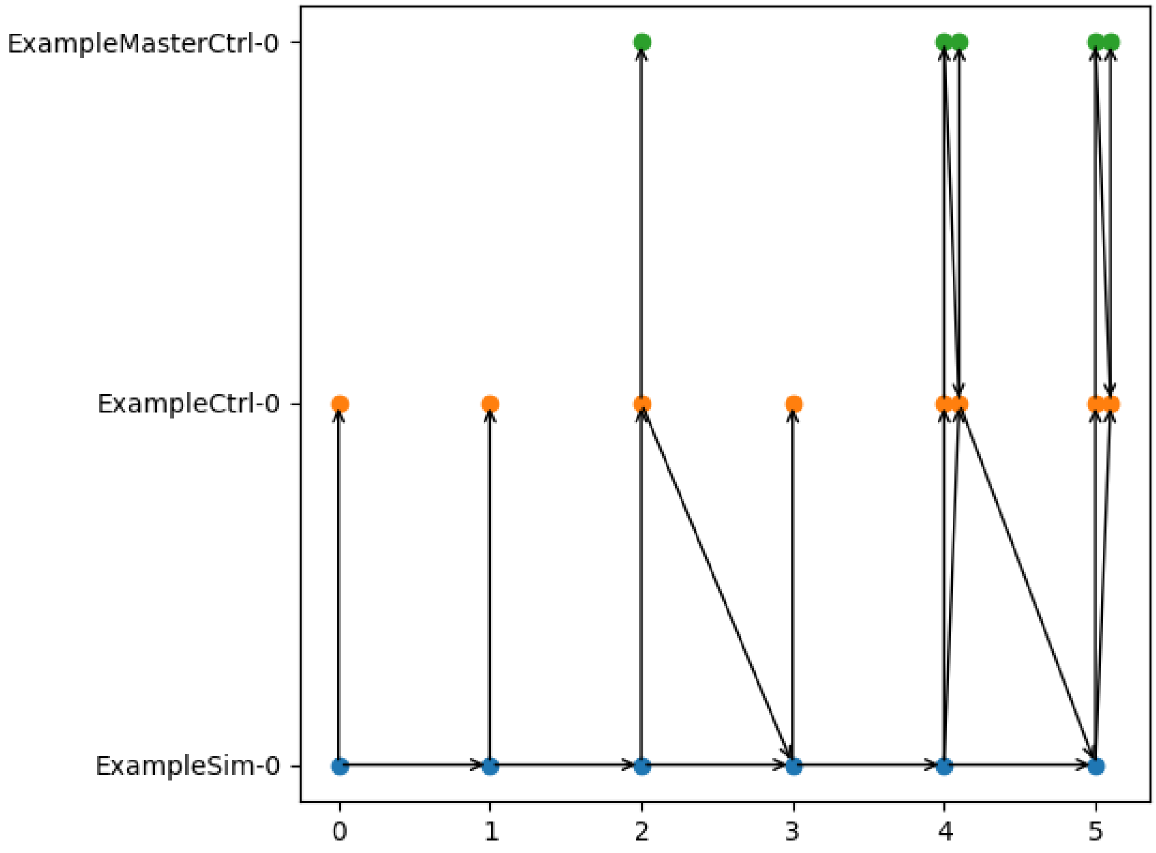


Fig. 2: Schedulung of demo 3.

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.

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


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.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")
        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)
            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

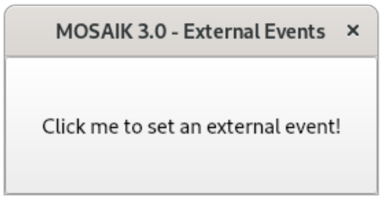
For the GUI, we create a new python module, e.g., `gui_button.py`.

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
```

(continues on next page)


MOSAIK 3.0 - External Events
Click me to set an external event!

(continued from previous page)

```
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):
        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.

```python
# 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.

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

# 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.

```python
# 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`.

```python
# 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
```

(continues on next page)

(continued from previous page)

```
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

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
```

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',
                    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)
```

(continues on next page)

(continued from previous page)

```
    # 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
```

(continues on next page)

(continued from previous page)

```
#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'],
```

(continues on next page)

(continued from previous page)

```
                ['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'],
                ['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

First you have to get the sources of the mosaik-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() {
```

(continues on next page)

(continued from previous page)

```
        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();
    }
```

(continues on next page)

(continued from previous page)

```
    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) {
```

(continues on next page)

(continued from previous page)

```
        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);
        }
        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',
```

(continues on next page)

(continued from previous page)

```
    },
}
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`.

CHAPTER

SIX

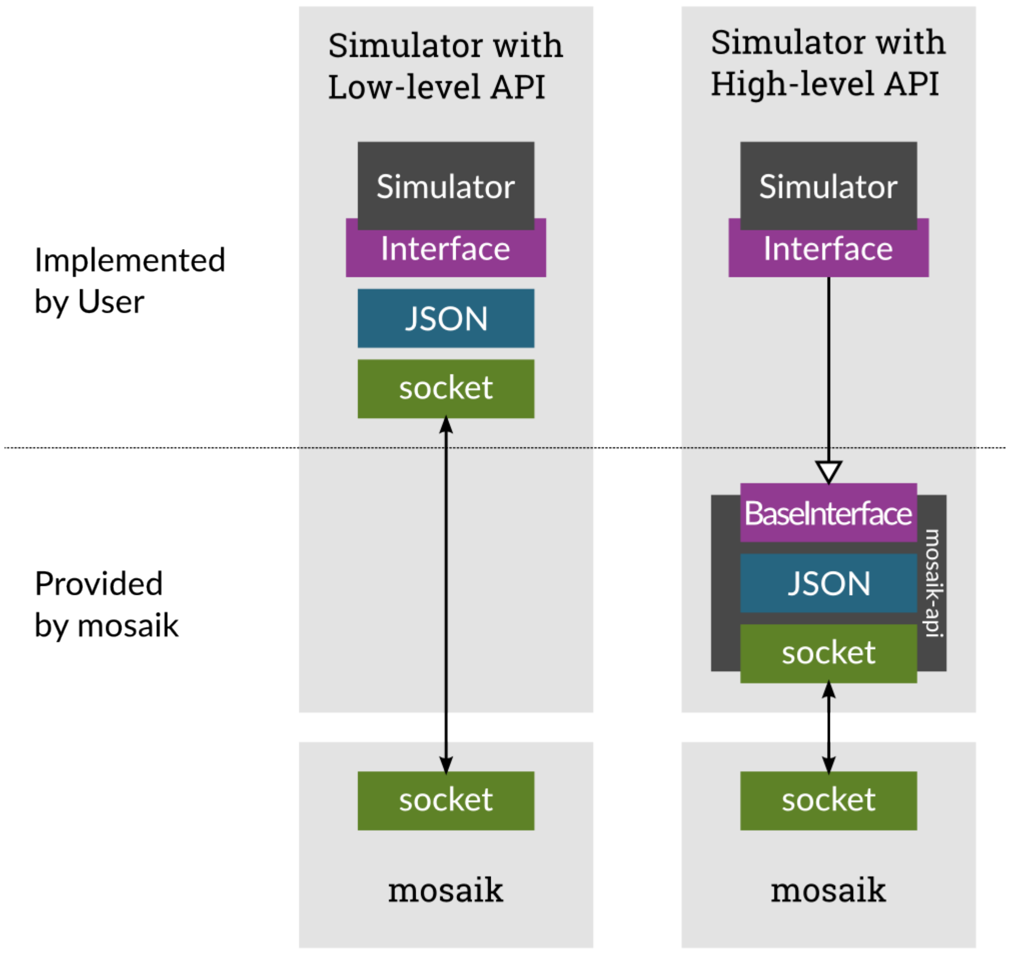
THE MOSAIK API

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 and Java. 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.

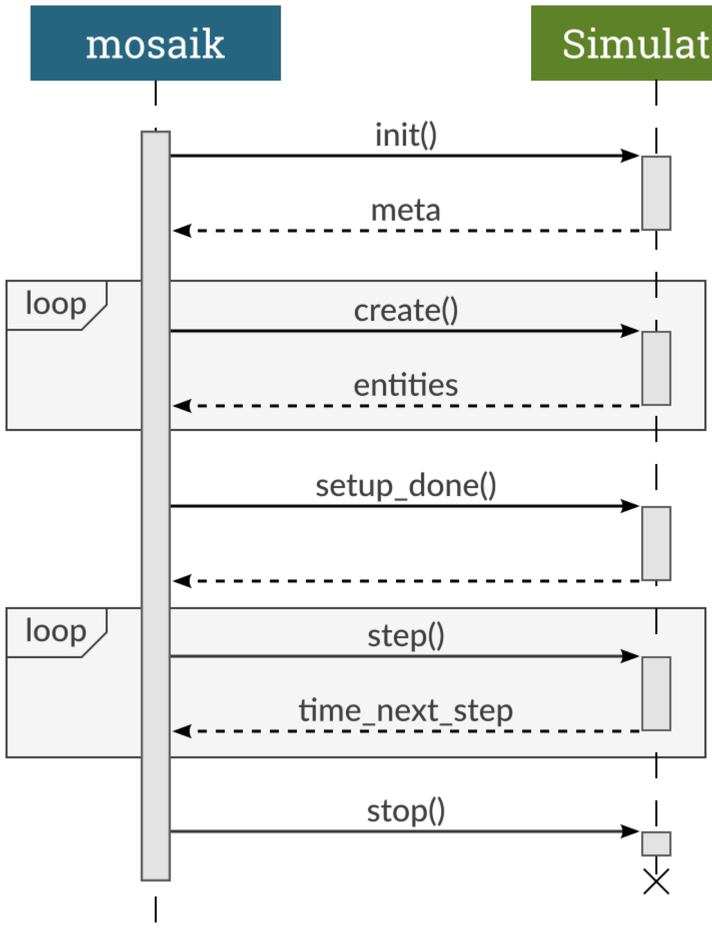
Following this, mosaik may call `create()` multiple times in order to instantiate one of the models that the simulator implements. The return value contains information describing the entities created.

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.

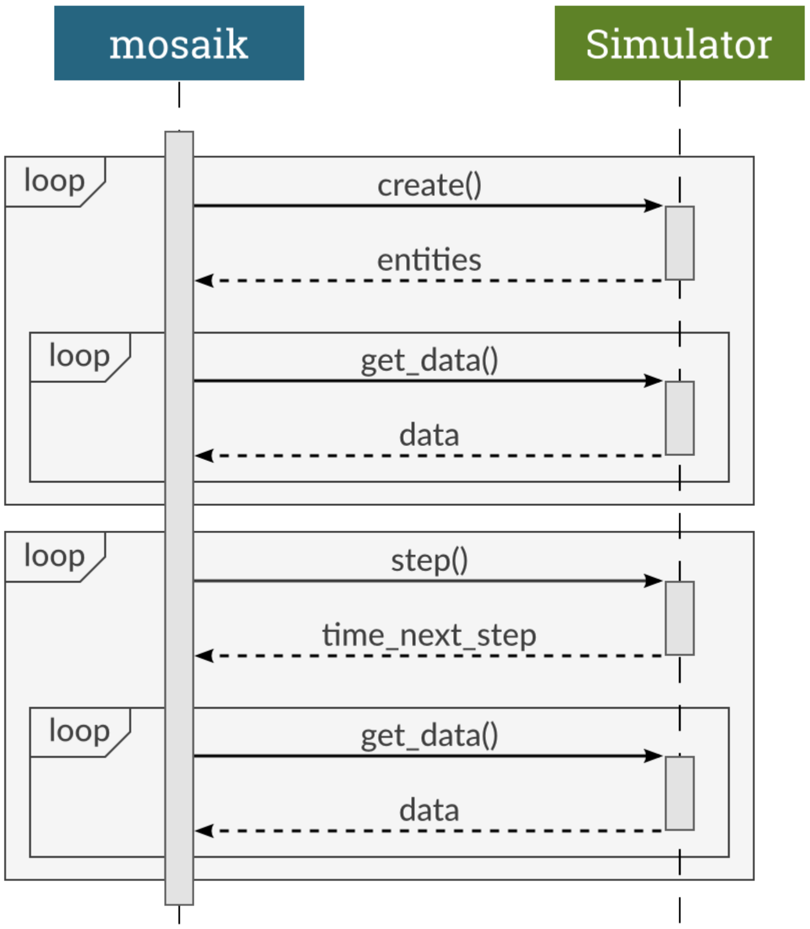
When the simulation has been started, mosaik repeatedly calls `step()`. This allows the simulator to step forward in time. It returns the time at which it wants to perform its next step.

Finally, mosaik sends a `stop()` message to every simulator to request its shut-down.

The following figure depicts the sequence of these messages:

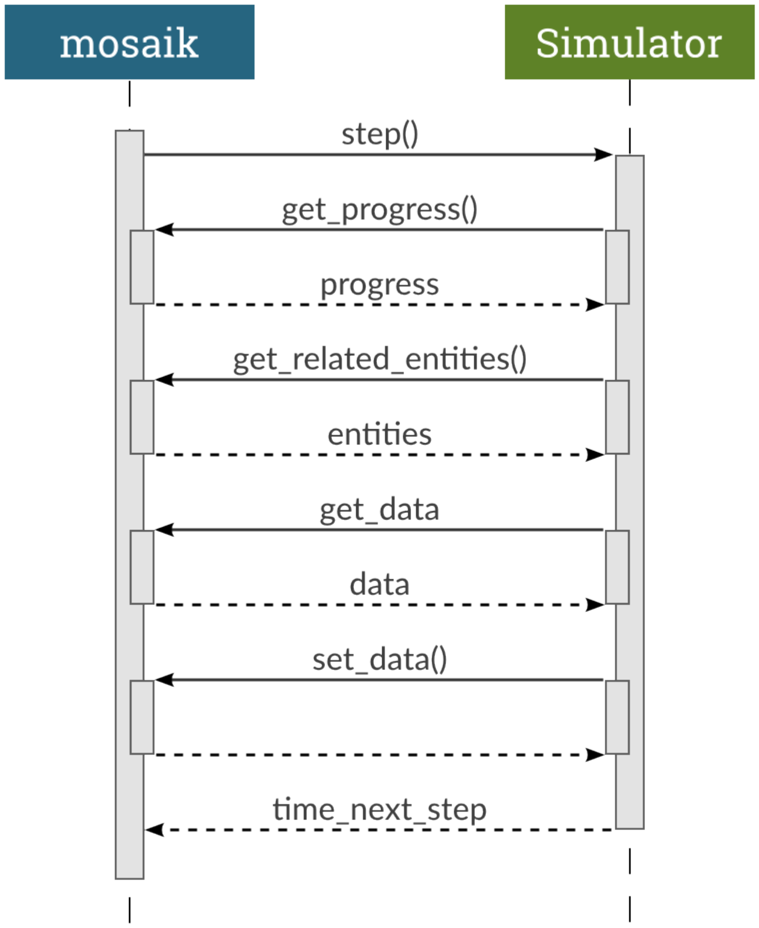


After `create()` or `step()` have been called, there may be an arbitrary amount of `get_data()` calls where mosaik requests the current values of some entities' attributes:



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()`).

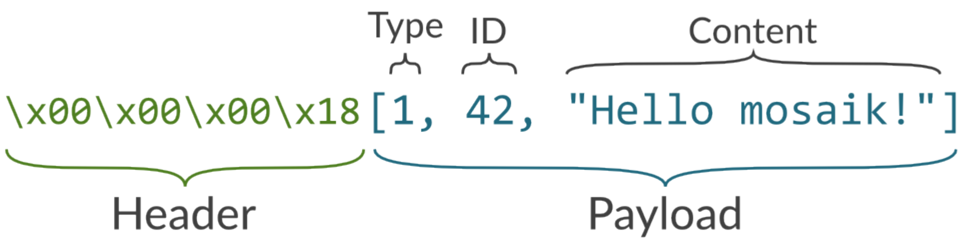


The next two sections explain the *low-level API* and the *Python high-level API* in more detail.

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*

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",
                "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"},
    },
```

(continues on next page)

(continued from previous page)

```
    "sid_0.eid_1": {
        "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
```

It supports Python 2.7, >= 3.3 and PyPy.

For the use of the Java-API please refer to the *Java-tutorial*.

6.3.2 Usage

You create a subclass of `mosaik_api.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.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.**Simulator**(*meta*)

This is the base class that you need to inherit from and implement the API calls.

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.

time_resolution

The time resolution of the scenario.

`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.

`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.

`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

`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*).

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).

configure(*args*, *backend*, *env*)

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* 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()
    # ...
```

`MosaikRemote.`**`get_progress`**`()`

Return the current simulation progress from `sim_progress`.

`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'},
    },
}
```

`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}}`).

`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'}}}`).

`MosaikRemote.`**`set_event`**`(`*`event_time`*`)`

Schedules an event/step at simulation time *event_time*.

The *mosaik-api* 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.`**`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:

```python
import mosaik_api


example_meta = {
    'type': 'time-based',
    'models' {
        'A': {
              'public': True,
              'params': ['init_val'],
              'attrs': ['val_out', 'dummy_out'],
        },
    }
}


class ExampleSim(mosaik_api.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

```

(continues on next page)

(continued from previous page)

```python
    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.start_simulation(ExampleSim(), description, extra_options)


if __name__ == '__main__':
    sys.exit(main())
```

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
>>> from mosaik.scenario import SimConfig
>>>
>>> sim_config: SimConfig = {
...     'ExampleSim': {'python': 'example_sim.mosaik:ExampleSim'},
... }
>>>
>>> world = mosaik.World(sim_config)
```

(You can leave off the type annotation on `sim_config` and the line importing `SimConfig` 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 instances 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, *simulator_0* and *simulator_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')`).

You can only connect entities that belong to different simulators with each other (that's why we created two instances of the *ExampleSim*).

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 also 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'},
```

(continues on next page)

(continued from previous page)

```python
}

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

Bi-directional (or cyclic) data-flows can occur easily in many scenarios, e.g. when you want to integrate control strategies. In this case you have to explicitly define in which order the simulators have to be stepped in case they are scheduled at the same time step simultaneously. Otherwise the simulation would get stuck in a dead-lock. Therefore this trivial approach is not allowed in mosaik:

```python
# 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 with this is that mosaik cannot know whether to compute *battery.P* or *controller.schedule* first.

There are different ways to solve this problem, depending of the stepping type of your simulators:

7.7.1 Time-based

For time-based simulators the easiest way is to indicate explicitly that the output of at least one simulator (e.g. the schedule of the controller)is to be used for time steps afterwards (here by the battery) via the *time_shifted* flag:

```python
world.connect(battery, controller, 'P')
world.connect(controller, battery, 'schedule', time_shifted=True,
              initial_data={'schedule': initial_schedule})
```

As for the first step this data cannot be provided yet, you have to set it via the *initial_data* argument. This example would result in a sequential execution of the two simulators. If you set the time_shifted flag for both connections, you get a parallel execution.

The other 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. 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
```

7.7.2 Event-based

For cyclic dependencies of event-based simulators you have to set one (and only one) connection to *weak*, analogous to the *time_shifted* connections. This allows mosaik to create a topological ranking of the simulators which is used to resolve eventual deadlocks (when two or more simulators have scheduled steps at the same time). In contrast to the time-shifted connections, weakly connected output can also be valid/used at the same point in time. This enables algebraic loops within one time step for example. You just have to make sure that you don't construct infinite loops. See section *Same-time Loops* for details.

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_grididx = lambda e: e.eid.split('-', 1)[1]  # Little helper function
cps_by_name = {remove_grididx(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.

`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:

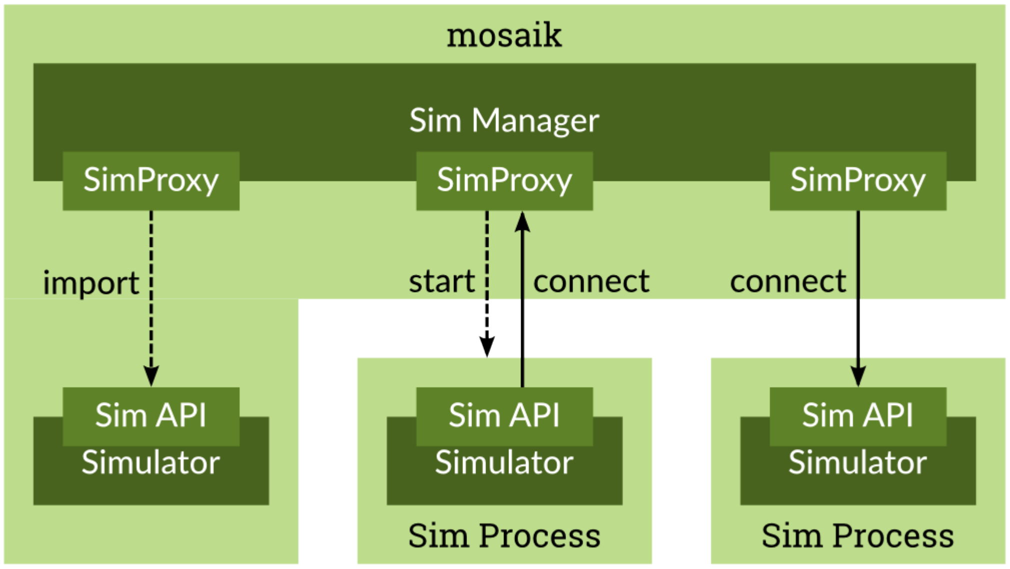
1. Simulators can be written in any language.
2. Simulation steps can be performed in parallel, if two processes don't depend on each others data.

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.

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.). It's 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, 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 than 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 continuous and event-type 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 $[t_{\text{now}}, t_{\text{next}})$ as shown in the following figure.

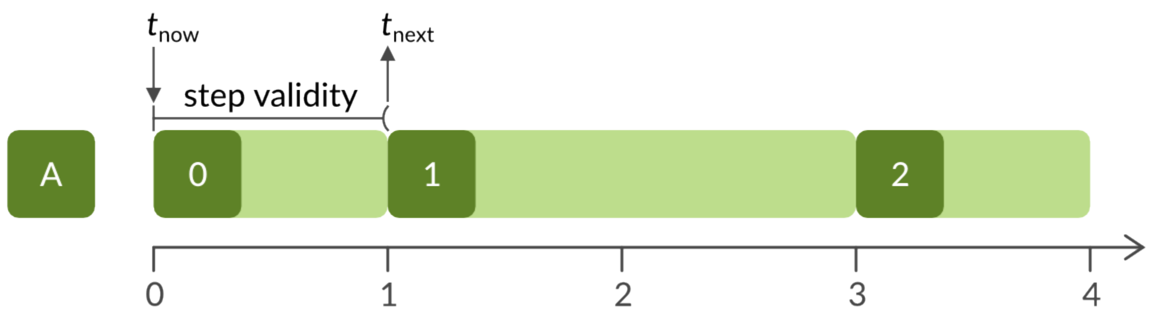


Fig. 1: Schematic execution of a time-based simulator A. t_{now}, t_{next} and the validity interval for its first step 0 are shown. The figure also shows that the step size of a simulator may vary during the simulation.

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 w 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.

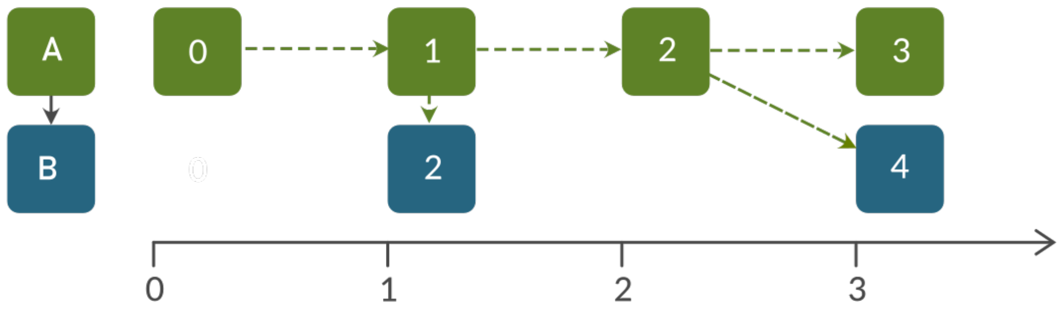


Fig. 2: Schematic execution of an event-based simulation. Depending on A's actual output a step of B is triggered (or not), at A's step time or later. Simulator A also schedules itself.

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 $t_{\mathrm{now(B)}}$. 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 $t_{\mathrm{now(B)}}$, that means $t_{\mathrm{next(A)}} > t_{\mathrm{now(B)}}$ as the following figure illustrates.

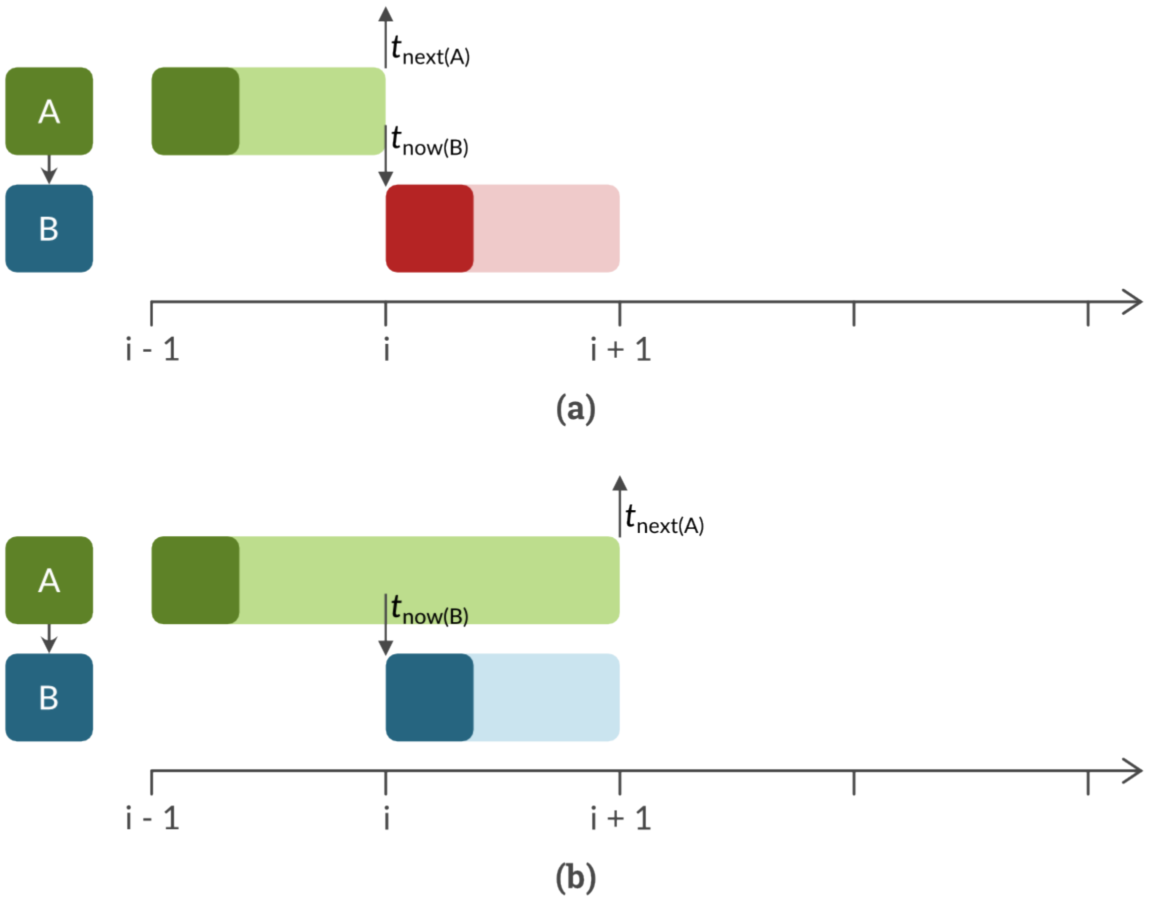


Fig. 3: **(a)** *B* cannot yet step because *A* has not progressed far enough yet ($t_{\mathrm{next(A)}} <= t_{\mathrm{now(B)}}$).
(b) *B* can perform its next step, because *A* now has progressed far enough ($t_{\mathrm{next(A)}} > t_{\mathrm{now(B)}}$).

If this condition is met for all simulators providing input for *B*, mosaik collects all input data for *B* that is valid at $t_{\mathrm{now(B)}}$ (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 $[t_{\mathrm{now(B)}}, t_{\mathrm{next(B)}})$.

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 then *A*, *A* would produce new data while *B* steps. *B* would still only use the data that was valid at $t_{\mathrm{now(B)}}$, because it only "measures" its inputs once at the beginning of its step:

On the other hand, if *A* had a larger step size then *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.

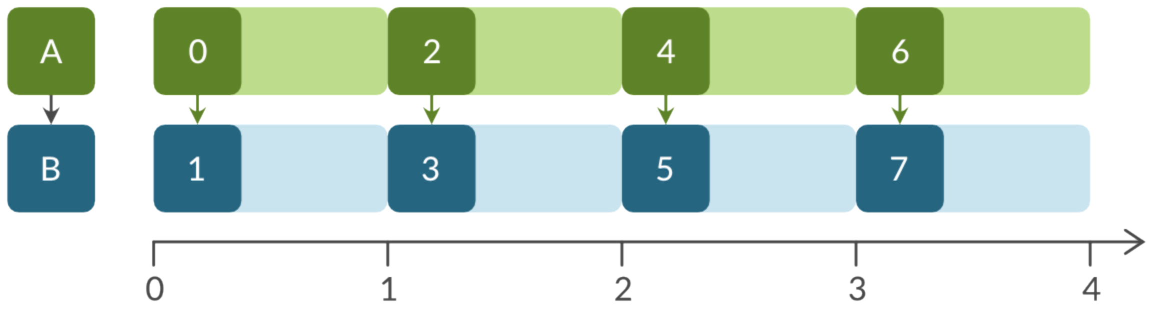


Fig. 4: In this example, *A* and *B* have the same step size. Mosaik steps them in an alternating order starting with *A*, because it provides the input data for *B*.

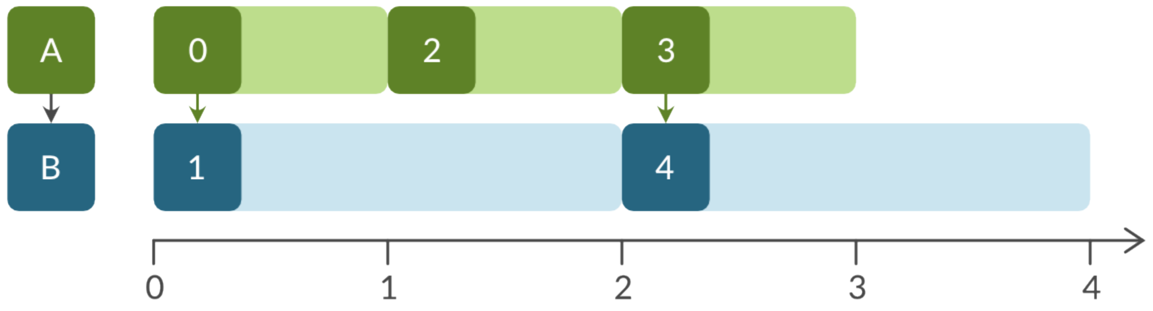


Fig. 5: In this example, *B* has a larger step size. It doesn't consume all data that *A* produces, because it only gets data once at the beginning of its step.

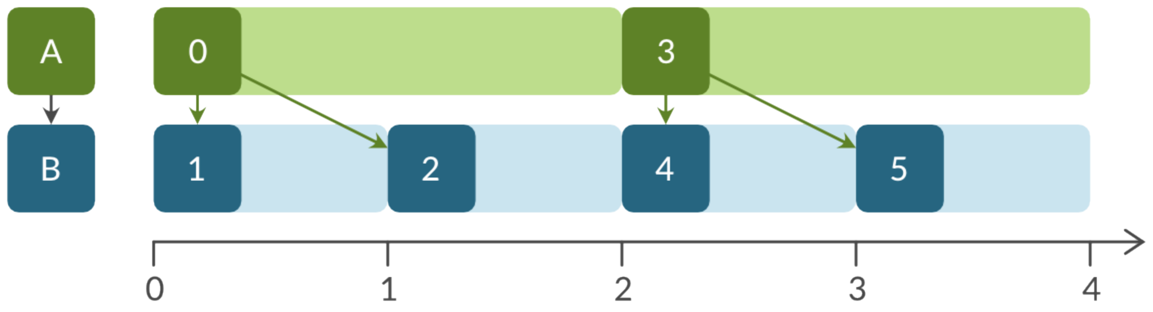


Fig. 6: In this example, *A* has a larger step size. *B* reuses the same data multiple times because it is still valid.

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.

TODO: Add info for rt-simulations

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 t which would be send to the controlled unit at $t + t$.

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 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.

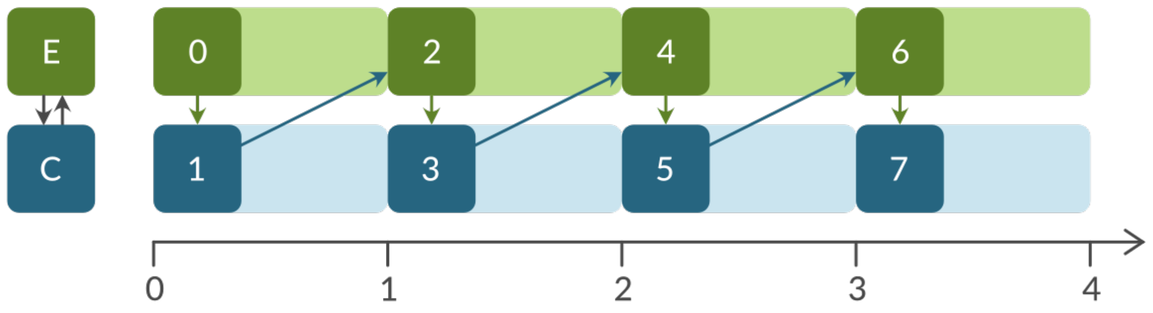


Fig. 7: In this example, a controlled entity *E* provides state data to the controller *C*. The commands or schedule from *C* is used by *E* in its next step.

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.

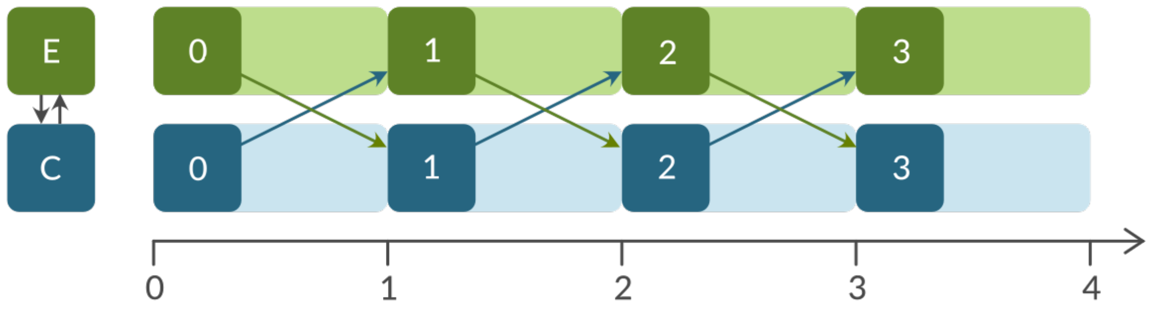


Fig. 8: In this example, two entities are running in parallel. The outputs of each simulator are used by the other one in its next step afterwards.

You can take a look at our *discussion of design decisions* for details.

9.4.1 Same-time (algebraic) loops

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`).

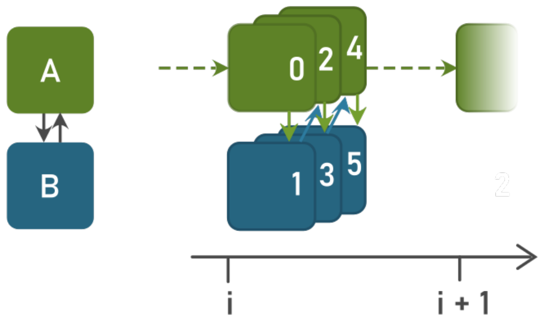


Fig. 9: A same-time loop with three repetitions between simulator *A* and *B*.

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 then 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.

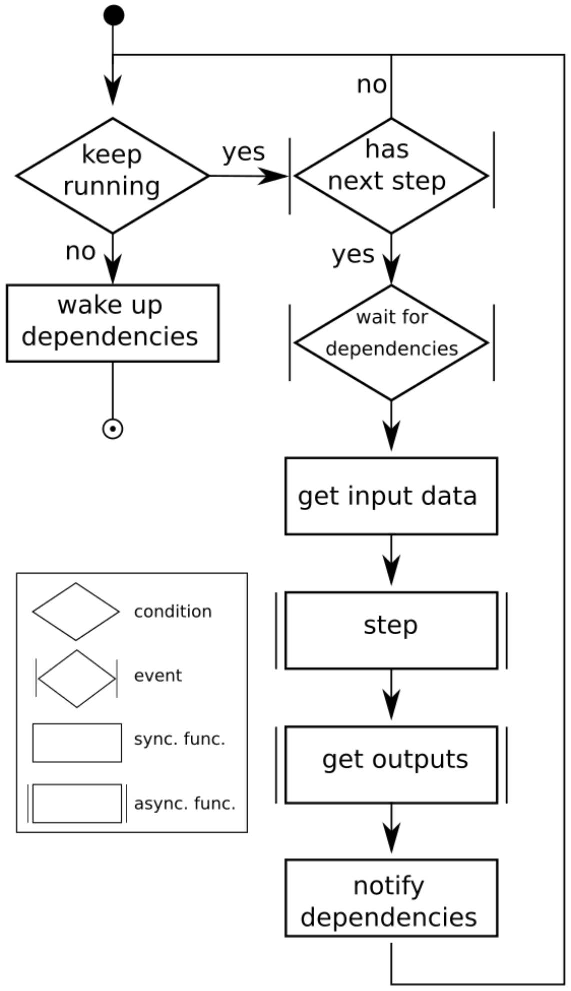


Fig. 10: Sim-process running for each simulator in parallel

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'e 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*.

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 throw 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* roll. 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*.

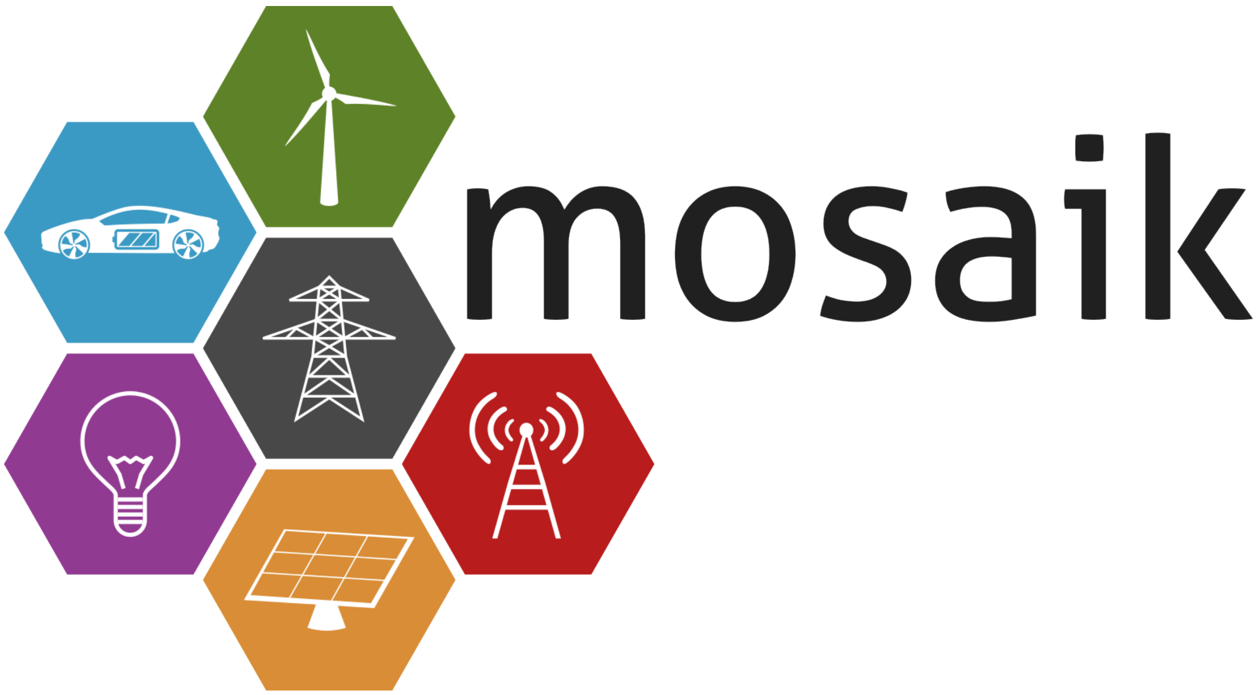
12.2 Logo and corporate identity

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 ende-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.

class mosaik.scenario.**World**(*sim_config: SimConfig*, *mosaik_config=None*, *time_resolution: float = 1.0*, *debug: bool = False*, *cache: bool = True*, *max_loop_iterations: int = 100*)

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.

`until: int`
The time until which this simulation will run.

`rt_factor: Optional[float]`
The number of real-time seconds corresponding to one mosaik step.

`sim_config`
The config dictionary that tells mosaik how to start a simulator.

`config`
The config dictionary for general mosaik settings.

`time_resolution`
An optional global *time_resolution* (in seconds) for the scenario, 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.

`sims: Dict[SimId, simmanager.SimProxy]`
A dictionary of already started simulators instances.

`env: Environment`
The SimPy.io networking `Environment`.

`srv_sock`
Mosaik's server socket.

`df_graph`
The directed data-flow `graph` for this scenario.

`trigger_graph`
The directed `graph` from all triggering connections for this scenario.

`entity_graph`
The `graph` of related entities. Nodes are `(sid, eid)` tuples. Each note has an attribute *entity* with an `Entity`.

`sim_progress: float`
Progress of the current simulation (in percent).

`start`(*sim_name: str*, ***sim_params*) → *mosaik.scenario.ModelFactory*
Start the simulator named *sim_name* and return a `ModelFactory` for it.

`connect`(*src: Entity*, *dest: Entity*, **attr_pairs: Union[str, Tuple[str, str]]*, *async_requests: bool = False*, *time_shifted: bool = False*, *initial_data: Dict[Attr, Any] = {}*, *weak: bool = 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}*).

set_initial_event(*sid: SimId*, *time: int = 0*)
Set an initial step for simulator *sid* at time *time* (default=0).

get_data(*entity_set: Iterable[Entity]*, **attributes: Attr*) → Dict[*Entity*, Dict[Attr, Any]]
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',
        ...
    },
    ...
}
```

run(*until: int*, *rt_factor: Optional[float] = None*, *rt_strict: bool = False*, *print_progress: Union[bool, Literal['individual']] = True*, *lazy_stepping: bool = True*)
Start the simulation until the simulation time *until* is reached.

In order to perform real-time simulations, you can set *rt_factor* to a number > 0. A rt-factor of 1. means that 1 second in simulated time takes 1 second in real-time. An rt-factor 0f 0.5 will let the simulation run twice as fast as real-time. For correct behavior of the rt_factor the time_resolution of the scenario has to be set adequately, which is 1. [second] by default.

If the simulators are too slow for the rt-factor you chose, mosaik prints by default only a warning. In order to raise a `RuntimeError`, you can set *rt_strict* to `True`.

`print_progress` controls 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 `print_progress='individual'` to get one bar per simulator in your simulation (in addition to the global one). ``print_progress=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.

You can also set the *lazy_stepping* flag (default: `True`). If `True` a simulator can only run ahead one step of it's successors. If `False` a simulator always steps as long all input is provided. This might decrease the simulation time but increase the memory consumption.

Before this method returns, it stops all simulators and closes mosaik's server socket. So this method should only be called once.

create_simulator_ranking()

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: mosaik.scenario.World*, *sim: mosaik.simmanager.SimProxy*)

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.

class mosaik.scenario.**ModelMock**(*world: mosaik.scenario.World*, *name: str*, *sim: mosaik.simmanager.SimProxy*)

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)`.

create(*num: int*, ***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.Simulator.create()`, but the simulator is prepended to every entity ID to make them globally unique.

class mosaik.scenario.**Entity**(*sid*, *eid*, *sim_name*, *type*, *children*, *sim*)

An entity represents an instance of a simulation model within mosaik.

sid

The ID of the simulator this entity belongs to.

eid

The entity's ID.

sim_name

The entity's simulator name.

type

The entity's type (or class).

children

An entity set containing subordinate entities.

sim

The `SimProxy` containing the entity.

property full_id: FullId

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.

`mosaik.scheduler.`**`run`**(*world: World*, *until: int*, *rt_factor: Optional[float] = None*, *rt_strict: bool = False*, *lazy_stepping: bool = True*) → Iterator[Event]

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.

`mosaik.scheduler.`**`sim_process`**(*world: World*, *sim: SimProxy*, *until: int*, *rt_factor: Optional[float]*, *rt_strict: bool*, *lazy_stepping: bool*) → Iterator[Event]

SimPy simulation process for a certain simulator *sim*.

`mosaik.scheduler.`**`has_next_step`**(*world: World*, *sim: SimProxy*) → Iterable[Event]

Return an `Event` that is triggered when *sim* has a next step.

world is a mosaik `World`.

`mosaik.scheduler.`**`wait_for_dependencies`**(*world: World*, *sim: SimProxy*, *lazy_stepping: bool*) → Event

Return an event (`simpy.events.AllOf`) that is triggered when all dependencies can provide input data for *sim*.

Also notify any simulator that is already waiting to perform its next step.

world is a mosaik `World`.

`mosaik.scheduler.`**`get_input_data`**(*world: World*, *sim: SimProxy*) → InputData

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`.

`mosaik.scheduler.`**`step`**(*world: World*, *sim: SimProxy*, *inputs: InputData*, *max_advance: int*) → Generator[Event, int, int]

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.

`mosaik.scheduler.`**`get_outputs`**(*world: World*, *sim: SimProxy*, *progress: int*) → Generator[Any, OutputData, int]

Get all required output data from a simulator *sim*. Yield an event that is triggered when all output data is received.

world is a mosaik `World`.

`mosaik.scheduler.get_progress`(*sims: Dict[SimId, SimProxy]*, *until: int*) → float

Return the current progress of the simulation in percent.

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.

`mosaik.simmanager.start`(*world: World*, *sim_name: str*, *sim_id: SimId*, *time_resolution: float*, *sim_params: Dict[str, Any]*)

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.Simulator` instance.

It raises a `SimulationError` if the simulator could not be started.

Return a `SimProxy` instance.

`mosaik.simmanager.start_inproc`(*world: World*, *sim_name: str*, *sim_config: Dict[Literal['python', 'env'], str]*, *sim_id: SimId*, *time_resolution: float*, *sim_params: Dict[str, Any]*) → *SimProxy*

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.

`mosaik.simmanager.`**`start_proc`**(*world: World*, *sim_name: str*, *sim_config: Dict[Literal['cmd', 'cwd', 'env'], str]*, *sim_id: SimId*, *time_resolution: float*, *sim_params: Dict[str, Any]*) → *SimProxy*

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.

`mosaik.simmanager.`**`start_connect`**(*world: World*, *sim_name: str*, *sim_config*, *sim_id: SimId*, *time_resolution: float*, *sim_params: Dict[str, Any]*)

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.

`class mosaik.simmanager.`**`SimProxy`**(*name: str*, *sid: SimId*, *meta: Meta*, *world: World*)

Handler for an external simulator.

It stores its simulation state and own the proxy object to the external simulator.

`rt_start: float`

The real time when this simulator started (as returned by *perf_counter()*.

`next_step: Optional[int]`

This simulator's immediate next step once it has been determined. The step is removed from the *next_steps* heap at that point and the *has_next_step* event is triggered.

`predecessors: Dict[Any, Tuple[SimProxy, DataflowEdge]]`

This simulator's predecessors in the dataflow graph and the corresponding edges.

`successors: Dict[Any, Tuple[SimProxy, DataflowEdge]]`

This simulator's successors in the dataflow graph and the corresponding edge..

`triggering_ancestors: Iterable[Tuple[SimId, bool]]`

An iterable of this sim's ancestors that can trigger a step of this simulator. The second component specifies whether the connecting is weak or time-shifted (False) or immediate (True).

`output_time: int`

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.

`related_sims: Iterable[SimProxy]`

Simulators related to this simulator. (Currently all other simulators.)

`name: str`

The name of this simulator (in the SIM_CONFIG).

`sid: SimId`

This simulator's ID.

`meta: Meta`

This simulator's meta.

`proxy: Any`

The actual proxy for this simulator.

`last_step: int`

The most recent step this simulator performed.

next_steps: List[int]
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: Optional[int]
The next self-scheduled step for this simulator.

progress: int
This simulator's progress in mosaik time.

This simulator has done all its work before time *progress*; its next stept will be at time *progress* or later. For time-based simulators, the next step will happen at time *progress*.

input_buffer: Dict
Inputs received via *set_data*.

input_memory: Dict
Memory of previous inputs for persistent attributes.

timed_input_buffer: TimedInputBuffer
'Usual' inputs. (But also see *world._df_cache*.)

sim_proc: Process
The SimPy process for this simulator.

has_next_step: Event
An event that is triggered once this simulator's next step has been determined.

wait_events: Event
The event (usually an AllOf event) this simulator is waiting for.

interruptable: bool
Set when this simulator's next step has been scheduled but it is still waiting for dependencies which might trigger earlier steps for this simulator.

stop()
Stop the simulator behind the proxy.

The default implementation does nothing.

class mosaik.simmanager.**LocalProcess**(*name*, *sid*, *meta*, *inst*, *world*)
Proxy for internal simulators.

stop()
Yield a triggered event but do nothing else.

class mosaik.simmanager.**RemoteProcess**(*name*, *sid*, *meta*, *proc*, *rpc_con*, *world*)
Proxy for external simulator processes.

stop()
Send a *stop* message to the process represented by this proxy and wait for it to terminate.

class mosaik.simmanager.**MosaikRemote**(*world*, *sim_id*)
This class provides an RPC interface for remote processes to query mosaik and other processes (simulators) for data while they are executing a `step()` command.

get_progress()
Return the current simulation progress from *sim_progress*.

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'},
    },
}
```

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}}`).

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'}}}`).

set_event(*event_time*)

Schedules an event/step at simulation time *event_time*.

13.6 `mosaik.util` — Utility classes and functions

This module contains some utility functions and classes.

`mosaik.util.`**`sync_process`**`(`*generator*, *world*, ***, *errback=None*, *ignore_errors=False*`)`

Synchronously execute a SimPy process defined by the generator object *generator*.

A *world* instance is required to run the event loop.

You can optionally provide a *errback* (error callback) which will be called with no arguments if an error occurs.

If *ignore_errors* is set to `True`, no errors will be printed.

`mosaik.util.`**`sync_call`**`(`*sim*, *funcname*, *args*, *kwargs*`)`

Start a SimPy process to make the *func()* call to a simulator behave like it was synchronous.

Return the result of the *func()* call.

Raise an `SimulationError` if an exception occurs.

`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`.

CHAPTER FOURTEEN

ABOUT MOSAIK

The latest version of mosaik is 3.1.1 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 Nieße* 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 3.1.1 - 2022-01-11

- [FIX] Fix compatibilty with mosaik 2 simulators (https://gitlab.com/mosaik/mosaik/-/issues/152)

14.2.2 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)

14.2.3 3.0.2 - 2022-06-01

- [CHANGE] Updated mosaik-api version to 3.0.2

14.2.4 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

14.2.5 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'|'event-based'|'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

14.2.6 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

Register Court

Amtsgericht Oldenburg
Registernumber VR 1956

VAT Identification Number

DE 811582102

Responsible in the sense of press law

Dr. Christoph Mayer (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. Christoph Mayer (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, 117
mosaik.exceptions, 117
mosaik.scenario, 117
mosaik.scheduler, 121
mosaik.simmanager, 122
mosaik.util, 126
mosaik_api, 77

INDEX

C

children (*mosaik.scenario.Entity attribute*), 120
Co-simulation, **145**
config (*mosaik.scenario.World attribute*), 118
configure() (*mosaik_api.Simulator method*), 79
connect() (*mosaik.scenario.World method*), 118
connect_many_to_one() (*in module mosaik.util*), 126
connect_randomly() (*in module mosaik.util*), 126
Control strategy, **145**
create() (*mosaik.scenario.ModelMock method*), 120
create() (*mosaik_api.Simulator method*), 78
create_simulator_ranking() (*mosaik.scenario.World method*), 120

D

data (*mosaik.simmanager.SimProxy attribute*), 123
Data-flow, **145**
df_graph (*mosaik.scenario.World attribute*), 118

E

eid (*mosaik.scenario.Entity attribute*), 120
Entity, **145**
Entity (*class in mosaik.scenario*), 120
Entity Set, **145**
entity_graph (*mosaik.scenario.World attribute*), 118
env (*mosaik.scenario.World attribute*), 118

F

finalize() (*mosaik_api.Simulator method*), 79
Framework, **145**
full_id (*mosaik.scenario.Entity property*), 120

G

get_data() (*mosaik.scenario.World method*), 119
get_data() (*mosaik.simmanager.MosaikRemote method*), 125
get_data() (*mosaik_api.Simulator method*), 79
get_input_data() (*in module mosaik.scheduler*), 121
get_outputs() (*in module mosaik.scheduler*), 121
get_progress() (*in module mosaik.scheduler*), 122
get_progress() (*mosaik.simmanager.MosaikRemote method*), 124
get_related_entities() (*mosaik.simmanager.MosaikRemote method*), 124

H

has_next_step (*mosaik.simmanager.SimProxy attribute*), 124
has_next_step() (*in module mosaik.scheduler*), 121

I

init() (*mosaik_api.Simulator method*), 77
input_buffer (*mosaik.simmanager.SimProxy attribute*), 124
input_memory (*mosaik.simmanager.SimProxy attribute*), 124
interruptable (*mosaik.simmanager.SimProxy attribute*), 124

L

last_step (*mosaik.simmanager.SimProxy attribute*), 123
LocalProcess (*class in mosaik.simmanager*), 124

M

meta (*mosaik.simmanager.SimProxy attribute*), 123
meta (*mosaik_api.Simulator attribute*), 77
Model, **145**
ModelFactory (*class in mosaik.scenario*), 120
ModelMock (*class in mosaik.scenario*), 120
module
    mosaik, 117
    mosaik.exceptions, 117
    mosaik.scenario, 117
    mosaik.scheduler, 121
    mosaik.simmanager, 122
    mosaik.util, 126
    mosaik_api, 77
mosaik
    module, 117
mosaik (*mosaik_api.Simulator attribute*), 77
mosaik.exceptions
    module, 117

mosaik.scenario
    module, 117
mosaik.scheduler
    module, 121
mosaik.simmanager
    module, 122
mosaik.util
    module, 126
mosaik_api
    module, 77
MosaikRemote (*class in mosaik.simmanager*), 124

N

name (*mosaik.simmanager.SimProxy attribute*), 123
next_self_step (*mosaik.simmanager.SimProxy attribute*), 124
next_step (*mosaik.simmanager.SimProxy attribute*), 123
next_steps (*mosaik.simmanager.SimProxy attribute*), 123

O

output_time (*mosaik.simmanager.SimProxy attribute*), 123

P

predecessors (*mosaik.simmanager.SimProxy attribute*), 123
progress (*mosaik.simmanager.SimProxy attribute*), 124
proxy (*mosaik.simmanager.SimProxy attribute*), 123

R

related_sims (*mosaik.simmanager.SimProxy attribute*), 123
RemoteProcess (*class in mosaik.simmanager*), 124
rt_factor (*mosaik.scenario.World attribute*), 118
rt_start (*mosaik.simmanager.SimProxy attribute*), 123
run() (*in module mosaik.scheduler*), 121
run() (*mosaik.scenario.World method*), 119

S

Scenario, **145**
ScenarioError (*class in mosaik.exceptions*), 117
set_data() (*mosaik.simmanager.MosaikRemote method*), 125
set_event() (*mosaik.simmanager.MosaikRemote method*), 125
set_initial_event() (*mosaik.scenario.World method*), 119
setup_done() (*mosaik_api.Simulator method*), 78
shutdown() (*mosaik.scenario.World method*), 120
sid (*mosaik.scenario.Entity attribute*), 120
sid (*mosaik.simmanager.SimProxy attribute*), 123
sim (*mosaik.scenario.Entity attribute*), 120
sim_config (*mosaik.scenario.World attribute*), 118
sim_name (*mosaik.scenario.Entity attribute*), 120
sim_proc (*mosaik.simmanager.SimProxy attribute*), 124
sim_process() (*in module mosaik.scheduler*), 121
sim_progress (*mosaik.scenario.World attribute*), 118
SimProxy (*class in mosaik.simmanager*), 123
sims (*mosaik.scenario.World attribute*), 118
Simulation, **145**
Simulation Model, **145**
SimulationError (*class in mosaik.exceptions*), 117
Simulator, **145**
Simulator (*class in mosaik_api*), 77
Smart Grid, **145**
srv_sock (*mosaik.scenario.World attribute*), 118
start() (*in module mosaik.simmanager*), 122
start() (*mosaik.scenario.World method*), 118
start_connect() (*in module mosaik.simmanager*), 123
start_inproc() (*in module mosaik.simmanager*), 122
start_proc() (*in module mosaik.simmanager*), 122
start_simulation() (*in module mosaik_api*), 81
Step, **145**
step() (*in module mosaik.scheduler*), 121
step() (*mosaik_api.Simulator method*), 78
stop() (*mosaik.simmanager.LocalProcess method*), 124
stop() (*mosaik.simmanager.RemoteProcess method*), 124
stop() (*mosaik.simmanager.SimProxy method*), 124
successors (*mosaik.simmanager.SimProxy attribute*), 123
sync_call() (*in module mosaik.util*), 126
sync_process() (*in module mosaik.util*), 126

T

time_resolution (*mosaik.scenario.World attribute*), 118
time_resolution (*mosaik_api.Simulator attribute*), 77
timed_input_buffer (*mosaik.simmanager.SimProxy attribute*), 124
trigger_graph (*mosaik.scenario.World attribute*), 118
triggering_ancestors (*mosaik.simmanager.SimProxy attribute*), 123
type (*mosaik.scenario.Entity attribute*), 120

U

until (*mosaik.scenario.World attribute*), 118

W

wait_events (*mosaik.simmanager.SimProxy attribute*), 124
wait_for_dependencies() (*in module mosaik.scheduler*), 121
World (*class in mosaik.scenario*), 117