eELib

elenia

Feb 09, 2024

CONTENTS

The **eELib** (**e**lenia **E**nergy **L**ibrary) is the software tool for simulations concerning **future power systems for prosumers**. The library with its functionalities and models can be used for various simulative investigations regarding research or current challenges in the field of a distributed electrical power system.

The goal of the eELib is creating a model library that is suitable for solving energy-related questions around prosumers (consumers that are now also producing energy). This includes, among other things, the …

- … creation and consideration of different **energy supply scenarios** (on building, district and grid level, among others with different penetration levels of distributed facilities like PV).
- … comparison of different **operating strategies for energy management systems**, including e.g. variable tariffs, multi-use concepts, operator models or schedule-based flexibility.
- … investigation of the **impacts and interactions** of prosumer households (e.g., sector coupling and electrification) **with the power grid** to identify violations of grid limits.
- … calculating the **economic values of different use cases and strategies** for components and systems.
- … investigation of innovative marketing strategies of market players in the spot and balancing **power markets**.

CHAPTER

ONE

ABOUT EELIB

The models in eELib - like other models too - represent the real processes of existing components in a quasi-stationary / quasi-dynamic approach under the assumption of simplifications. The implementations of the eELib hold some characteristics, that will shortly be explained here.

1.1 General Setup

To start a simulation, one has to set up a `scenario`-file that should look like the test scenarios in the `examples` folder. There, one has to set up the **models** to be used and how to **connect** them. **Data** for the models has to additionally be provided.

1.2 Folder Structure

The model library is in a public Gitlab-Repository.

This is to give an overview of where different parts of the eELib are stored:

- **docs** : Files for the documentation with AutodocSphinx into the GitLab Pages style. Documentation is stored in `.rst` files within the source subfolder.
- **eelib** : This stores the main part of the elenia Energy Library, as it contains the models and all other functionalities.
  - **core** : Here all of the models are stored. This is divided into the components (like PV system or electric vehicle), control models (like energy management system), grid models (like grid control) and market models (like intraday market).
  - **data** : This contains all models that are "just" retrieving input data (like a simple csv-reader). It also includes the functionalities for simulation data to be collected and assessed.
  - **utils** : Contains helper functions and classes.
    - **eval** : Here the functionalities for evaluating and plotting/presenting outputs of simulations are stored, mostly in accessible and modifiable python scripts or jupyter notebooks.
  - **testing** : Contains all of the testing for the models and the whole library.
- **examples** : This folder gives data and scripts for various test scenarios and should provide an overview of how the eELib can be used. Data is stored in the data which also contains the simulations results.

The highest folder also contains various files that are used for setup of the environment and gitlab communication.

Familiarize with the folder structure of the `eElib package` by exploring the *API Reference*.

1.3 Plug-and-Play Style

The programming of the models should be implemented in such a way that it can also be used in real-world applications without any adjustments. This ensures that, for example, in the case of changed simulation scenarios or also in laboratory investigations, the same source code can simply be applied within the framework of “**plug-and-play**”.

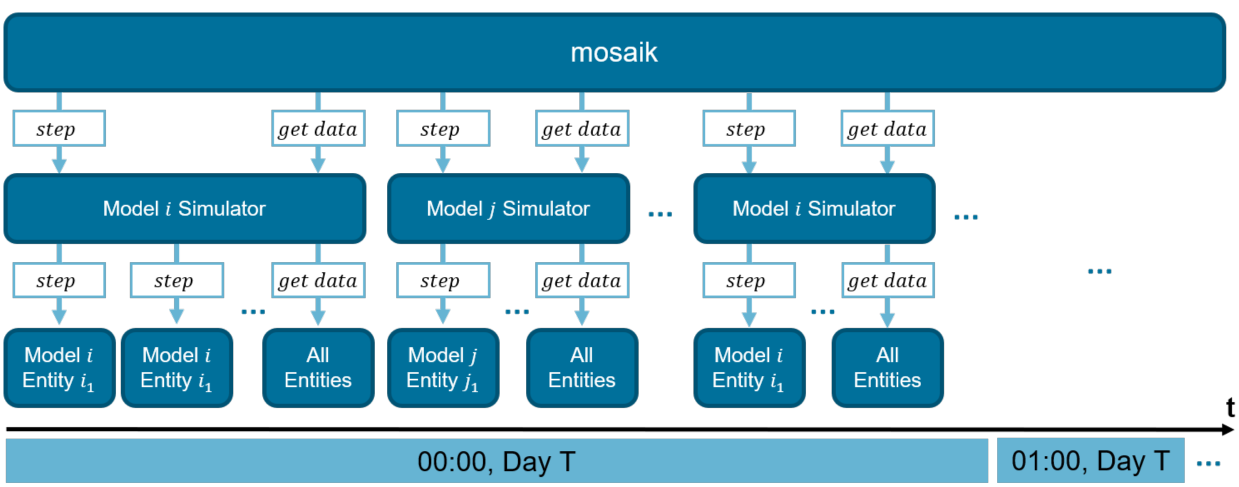
1.4 Coupling With mosaik

In order to make the above mentioned investigation cases possible, one needs to couple the different models of the library in scenarios to be created, e.g. a **PV** system with the energy management system (**EMS**). For this purpose, an *orchestrator* is to be used, which performs the exchange of data sets between models, calls the calculation of the individual models and controls the general model flow as well as the coupling with a database. mosaik is intended for these tasks. Certain input values are assumed, data sets are calculated internally and output data sets are issued. See the *mosaik Doc Part* for more information. The eELib should definetily be usable with other orchestrators of a simulation, but the explanations in this documentation are done for mosaik and its simulation orchestration. Additionally, the eELib provides simulators to its models that serve the prupose of APIs for a simulation with mosaik.

1.5 Event-Based Simulation

The computations within the eELib are executed in an **event-based** manner. This implies the process of a simulation to depend on the triggering of events. Independent of a *step_size* - the length of a simulation step in seconds - the process within one simulation step is executed by event triggering, as depicted in the following figure. To add to the explanation of mosaik, the execution of events depends on the triggering of such events. E.g. the simulation of a BSS is triggered by an EMS sending a power set value to the BSS. For this, mosaik knows about …

- … when each model has to be called for calculation.
- … which outputs for the models are send to which inputs for other models.

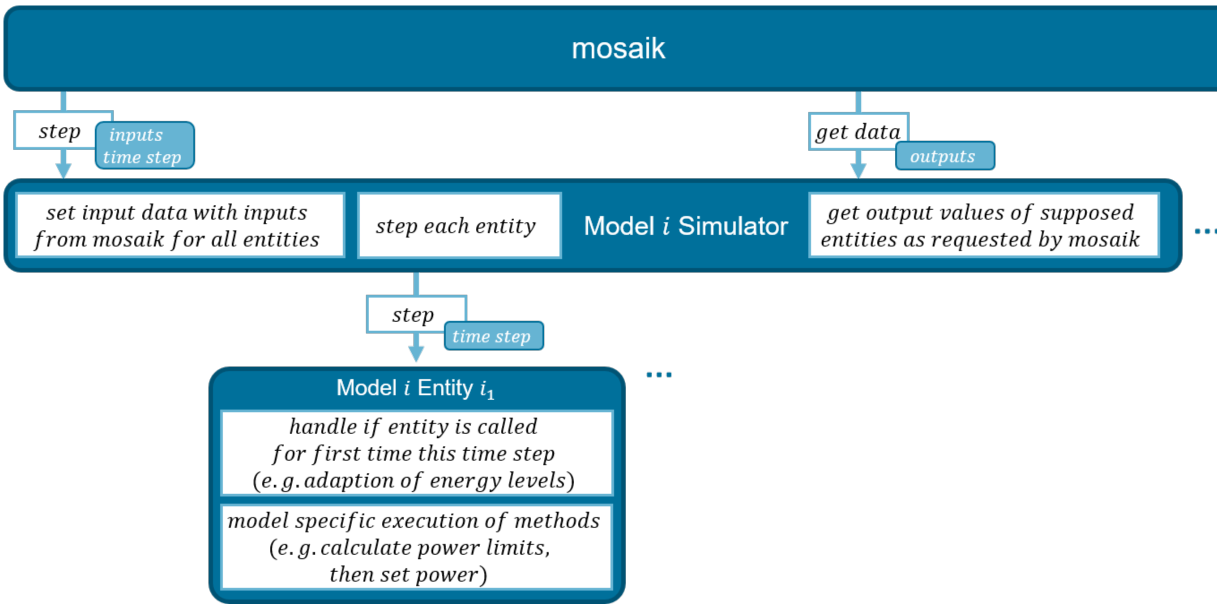


1. mosaik calls the calculation of the entities via the simulator
2. models calculate output with the already given input
3. simulator returns the time, when the models have to be calculated next
4. mosaik sends the output to a connected entity (e.g. the power generation of a PV system to the HEMS)

5. mosaik then calls the calculation of the next entity for which all inputs have been collected (this is done by means of the word "triggering", so the finalized calculation of one entity triggers the calculation of another entity…)

This is done within a single timestep as long as data is send between the entities such that the renewed calculation of a model entity is triggered. mosaik calls the execution of the models in the way they are connected to each other and send values each way. If everything is finished for this timestep, mosaik advances to the next timestep and the simulation process carries on.

The implementation of the process within one simulation step can be better seen in the next figure. It is shown, that when a model is called, the inputs for the models are provided by mosaik and set by the simulator. Afterwards the models are stepped and ultimately the simulators are called to extract the (by mosaik requested) result outputs from the models.



CHAPTER

TWO

WIKI

The Wiki provides a walkthrough for different tasks, as well as self-aid in case of common questions.

For a detailed start, we recommend fully reading this Wiki. It covers. . .

- **Basics** - Setup of a *programming environment*,
- **Usage** - Guides to *configure a scenario* and *run a simulation*,
- **Contribution** - Introduction to *git workflow* and adding *models*/*simulators*/*strategies*
- . . . and much more

After fully going through the Wiki section and completing the installation, we recommend to . . .

- . . . familiarize with the folder structure of the `eElib package` by exploring the *API Reference*.
- . . . have a look at the test scenarios in the folder examples - esp. `test_scenario_building.py` - and to comprehend the processes/procedures.
- . . . run the `test_scenario_building.py` as declared at the end of the installation guide and try if it works!

2.1 Installation and Setup

2.1.1 Installation and Setup of Python for working wit eELib

1. Download Python version 3.10.X from https://www.python.org/downloads/
2. When the download is finished, double-click the installer.
3. Select *Install for all users* and click *Next* >.
4. The default installation path is okay. Click *Next* >.
5. 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* >.
   - If not, the Python Path has to be added to system variables by hand.
6. When Windows asks you to allow the installation, do so. Wait for it to happen. Click *Finish*.

Note: This will also install the Python package manager pip. For checking and if not, see https://pip.pypa.io/en/stable/getting-started/ (can also be used generally for working with pip)

2.1.2 Installation and Setup of Python IDE (VSC)

Easier than using command window or PowerShell is the use of an IDE (integrated development environment) for Python, especially when working with the code.

1. Decide for an IDE. There are several good options: PyCharm, Visual Studio Code, Jupyter Notebook, IDLE, Spyder, Pydev
   - We recommend Visual Studio Code (VSC) for eELib, so this tutorial will be based on VSC
2. If needed, download VSC from the homepage and install it: https://code.visualstudio.com/
3. Configuration of User Settings
   1. If not installed, install the Python Extension under Extensions (`Ctrl` + `Shift` + `X`)
   2. Settings -> Extensions -> Python -> Formatting: Provider -> set to "black"
   3. Settings -> Text Editor -> Formatting -> Format on Save should be true
   4. Settings -> Features -> Notebook -> Format on Save should be true
   5. Install the autoDocstring extension the same way as the Python Extension
   6. Install the H5Web Extension for a quick look at the HDF5-Simulation-Output
   7. For max line lenght of 100 set: Settings -> Editor -> Rulers -> click "Edit in settings.json" and then type "100" instead of the default "80" (you can directly see the vertical line shift to the right when saving the settings.json file

2.1.3 Cloning eELib Repository

1. For Git communication, have a look at the page *Git Workflow*
2. Clone the Git Repository with VSC: When all folders are closed, select *Clone (Git) Repository*
   - https-Address: "https://gitlab.com/elenia1/elenia-energy-library"
3. You can also use GitBash for cloning (for some it seems easier)

```
$ git clone https://gitlab.com/elenia1/elenia-energy-library
```

Note: The path to the project folder will now be noted as <*Project Folder*>.

2.1.4 Working in VSC with eELib

1. Open VSC and navigate to <*Project Folder*>
2. Open new Terminal: PowerShell is recommended (GitBash or Command Window are possible too, but not as mighty)
3. Create a virtual environment in the directory of your repository:
   1. Run `python -m venv <_VENV-PATH_>`
      - For virtual environment path `<_VENV-PATH_>`, we typically use `.venv`
      - Accept VSC for acknowledging new environment, if it is detected
   2. Run the activation script for Powershell: `.\<VENV-PATH>\Scripts\Activate.ps1`

- In case Scripts can't be executed, you have to adjust the Execution Policy by running `Set-ExecutionPolicy Bypass -Scope CurrentUser -Force` and try again
- In case you don't use relative paths, the `.\` at the beginning isn't needed
- (Command window has different activation file 'activate.bat')
- Check: If successful, the prompt should now start with `(.venv)`
- Check whether the correct python interpreter is selected: `python --version` (Output: Python 3.10.X)

4. Install requirements into the virtual environment
   - If VSC explorer isn't already in the repository folder, you have to navigate there
   - Run `pip install -r requirements.txt`
5. If a new release of pip is available, you can update it via `python.exe -m pip install --upgrade pip`
6. Check: If you open a Python file, the selected virtual environment is listed in the lower right corner in the blue row *('.venv')*
7. Install Configurations for Processes, that are executed before each commit
   - Run `pre-commit install`

2.1.5 Test successfull installation

1. **You can test the functionality and correct installation by running a testcase**
   - Open the file `test_scenario_building.py` in the examples folder and click on the *Run* sign in the upper right corner
   - (Or run `python test_scenario_building.py` in the terminal)
2. If you are not able to run the `test_scenario` and get the error `no module named 'eelib'` …

   If your are in your *local project path* …
   execute `pip install -e.`

   Otherwise you have to insert your local project path …
   execute `pip install -e <local project path>`

2.2 Git Workflow

This wiki page will give detailed instruction on how to work with the commands in Git, especially via Visual Studio Code (VSC)! It will not provide a full overview of how the git process works but informs about the necessary commands.

2.2.1 1. Create Personal Access Token

1. In Gitlab, via 'Edit profile', you go to 'Access Tokens' and create a Token.
    - The name is irrelevant and the expiration date can be set to a year.
    - You can select all scopes.
2. The Token has to be saved somewhere, because you have to give it in order to establish a connection from your local repository to the online repository.

2.2.2 2. Cloning

The next thing that has to be done is cloning the online Git repository onto your local computer.

1. You can do this via Visual Studio Code by typing in `gitcl` in the command line and following the processes.
    - Use either the URL or SSH-key from the repository.
    - When asked for username and password you have to give your **GitLab username** and the **Personal Access Token**.
    - This process can also be done via GitBash, when going to the target folder and typing in `git clone` <REPO-URL>.
2. It is recommended to end this process by saving the Personal Access Token in this Git project by running `git remote set-url origin https://oauth2:[PersonalAccessToken]@<GIT-REPO>` via GitBash, then your Git program will not ask for the Token every time.
3. You should also set your username and mail adress for when using Git by running `git config --global user.name "Your Name"` resp. `git config --global user.email "youremail@yourdomain.de"`.

2.2.3 3. Visual Studio Code

- Visual Studio Code allows easy version management via Git.
- You should use the window "Source Control" (left side) to always see the current changes you did on the code.
- When using the Extension Git Graph , you can also have an overview of all the changes that have been done in the repository (possibly by others too!).

2.2.4 4. Change the Branch

- Before any adaption to the code, you should always check your currently selected branch.
- The default is always the `main` branch, in which you should **not change anything!**
- Changing the selected branch can be done by the Command "Source Control -> Branches -> Switch to Another Branch" and then selecting that specific branch. Or you can use the command `check-out to` in order to switch to another branch.

Hint: You can only change the selected branch, when you currently have no changes in the files.

2.2.5 5. Saving Changes (Commit)

The next thing is saving changes, that were coded locally - this is done by 'committing' the code, which is still a local process but kind of saves the stage at the moment.

1. You have to "stage" the relevant code files by pressing the "plus" (+) sign.
2. Then type in a fitting commit message (What have you done? Short!).
3. Last thing is to hit the commit button ("check" sign).

2.2.6 6. Getting Changes from Online Repository (Pull)

When others made relevant code changes, which you might need, you can get their changes from the online repository by "pulling" them.

1. For that you must't have everything committed - so do that first.
2. To pull the current state of the branch from the online repository, simply click on the 3 points and select "Pull".
3. In case any merge conflicts occur, see step *8. Merge Changes*

2.2.7 7. Sending Changes to Online Repository (Push)

- When having implemented relevant code changes, which other programmers/users might need, AFTER VALIDATING THEM you can (and should regularly) push your saved commits.
- This should only be done after pulling the online repository first.
- Also, all current changes have to be staged.
- To push, simply click on the 3 points and select "Push".

2.2.8 8. Merge Changes

- Merging Changes is generally needed, when two programmers changed parts of the same code and Git does not know, which "solutions" it should select.
- This process should be done **carefully**, as nobody wants to discard changes, that have been done by others.
- Merging can be needed in two cases: when you merge another branch into your own local branch, or when you pull the online repository status into your local state.
- In the process, click on the files with "*Merge Conflicts*", go through the problems and try to find a solution that approriately takes both solutions into account.
- If you have concerns, ask for help!
- Merges with merge conflicts always have to be committed after the conflicts have been solved, and the file is saved and staged.

2.2.9 9. Create a New Branch for each Topic

The *work flow* is supposed to include the creation of a new branch for each problem. So in case you want to create a new model, you create a branch named `model/[model_name]` and work on this problem only in this branch. Or if you want to fix something existing, create a branch named `bugfix/[problem_name]` etc.

This is a way to structurize the current work and additions to the joint use of the library in the `main` branch.

2.2.10 10. Merge Your Changes into the main Branch

When you

1. completed a task in a specific branch (don't use the `main` branch for that!)
2. and tested your stuff,

you can make this accessible for others via the `main` branch.

For that you should …

1. … merge the current state of the main branch into your (feature) branch.
2. … push your changes into the remote branch.
3. … go to the GitLab Repository and create a merge request for your branch into `main`. There you should also assign a *developer* to shortly check your updates and may even assign a *reviewer*.

The merge will then be completed and afterwards your changes are also part of the `main` branch.

2.3 Mosaik

mosaik, according to the mosaik documentation is a "flexible Smart Grid co-simulation framework". It can combine various existing models to run large-scale scenarios - and this is what we intend it for in our use. mosaik can combine our prosumer models like electric vehicles, energy management systemens, grid calculations, and all the others. While we in our eELib focus on the implementation of power system models and energy management strategies, mosaik is used for simulatory behaviour.

For introduction to mosaik you can or should have a look at its tutorial. Recommended articles of this tutorial for the use within the eELib are:

1. Integrating a simulation model into the mosaik ecosystem
2. Creating and running simple simulation scenarios

Some more articles that could prove helpful:

1. Adding a control mechanism to a scenario
2. Integrating a control mechanism

2.4 Configure a Scenario With an Excel File

Caution: Still **WIP** and might not work.

2.4.1 Excel-file setup

1. Open Excel-file (`eelib\utils\simulation_setup\sim_config_data.xlsx`). This one is part of the `utils package`.
2. The sheets **bus, load, ext_grid, trafo, line, sgen, storage** are used to configure the grid.

   bus | load | ext_grid | trafo | line | sgen | storage | ems | household | pv | bss | hp | cs | ev | heat

3. To **add a new entity** to the grid you create a new row and include the index name and the init_vals of the element. More information about the characteristics of the grid elements: https://pandapower.readthedocs.io/en/v2.13.1/elements.html

   (Make sure you have at least **one transformer**, **one external grid** and **buses are connected through lines**)

	name	vn_kv	type	in_service	x	y
0	bus_0	20	b	True	1	0
1	bus_1	20	b	True	2	0
2	bus_2	0.40	b	True	3	0
3	bus_3	0.40	b	True	4	0
4	bus_4	0.40	b	True	5	-1
5	bus_5	0.40	b	True	6	-1
6	bus_6	0.40	b	True	7	-1
7	bus_7	0.40	b	True	6	0
8	bus_8	0.40	b	True	7	0
9	bus_9	0.40	b	True	8	0
10	bus_10	0.40	b	True	9	0
11	bus_11	0.40	b	True	6	1
12	bus_12	0.40	b	True	7	1

4. After configurating the grid you can add the model entities.
   1. Open the sheet ems and define the number of ems and connect them to a bus
   2. Households, charging stations or pv can be connected to a bus too (but only one per bus)
   3. Adding a new entitiy of a model type is similar to add a new grid element (important is to set the connection to an ems or a bus and differentiate between csv_reader model or exact mode type
   4. Now set the init_vals for every entity of the model type

	bus	ems	name	type	output_type	p_rated	strategy	p_rated_csv	p_rated_profile	cos_phi	filename	header_rows	date_format	start_time
0		2	charging_station_0	exact	AC	11000	max_P							
1		4	charging_station_csv_0	csv				11000	11000	0.95	examples	2	YYYY-MM-DD HH:mm:	01.01.2020 00:00
2	12		charging_station_csv_1	csv				11000	11000	0.95	examples	2	YYYY-MM-DD HH:mm:	01.01.2020 00:00

2.4.2 Create .json files for scenario

1. Open jupyter notebook `eelib\\utils\\simulation_setup\\script_sim_setup.ipynb`
2. Run the corresponding cells to import the packages and set the input and output paths:

   To create a grid

   1. Read the sheets from the excel file
   2. Run the "create grid data file" cell
   3. An image of the grid is shown underneath and the `.json`-file is safed at `C:/Users/Puplic/Documents`

   To create model data file

   1. Read the model type sheets frome the excel file
   2. Run "create model data file" cell
   3. `.json`-file is safed at `C:/Users/Public/Documents`

   To create connection file

   1. Run the cell "Create model_grid_config file"
   2. File is safed at `C:/Users/Public/Documents`
3. Now run a scenario file from examples folder, e.g. `test_scenario_grid` or `test_scenario_building`

2.4.3 Add a completely new model type

1. Create a new sheet for the model type in the excel file
2. Create entities and set the init_vals and the connection to a bus or ems for the new model type
3. Open the jupyter notebook `eelib\\utils\\simulation_setup\\script_sim_setup.ipynb`

   1. Read the new excel sheet

   ```
   df_new_model_type = pd.read_excel(FILE_SCENARIO_INPUT, sheet_name="new_model_
   ↪type", index_col=0)
   ```

   2. Add new model entities to model_data file (if necessery differentiate between exact- and csv-model)

   ```
   #create an empty list for the new model type
   new_model_type = []
   for i in df_new_model_type.index:
   # Add all init_vals for model type
      new_model_type_data = {
         "datafile": df_new_model_type.at[i, "datafile"],
         "start_time": df_new_model_type.at[i, "start_time"],
         "date_format": df_new_model_type.at[i, "date_format"],
         "header_rows": int(df_new_model_type.at[i, "header_rows"]),
      }
      new_model_type.append(new_model_type_data)
   # add new model type list to dictionary
   model_data= { ...
               new_model_type : new_model_type
   }
   ```

3. Add new model type to `model_grid_config` file

```
# create an empty list for the new model type
new_model_types = []
# loop over all entities
for j in df_new_model_type.index:
      # if the entity is connected with bus or ems the name of the entity is added to the list new_model_types
      if df_loads.at[i, "bus"] == df_new_model_type.at[j, "bus"] or ems_idx == df_new_model_type.at[j, "ems"]:
            new_model_type = df_new_model_type.at[j, "name"]
            new_model_types.append(new_model_type)
# add new model type list to dictionary
elements = {...
      new_model_type: new_model_types
}
```

4. Now run the cells to create new `.json` files

2.5 Set Up and Run a Simulation

2.5.1 What files are needed for a simulation?

Scenario script

Start the simulators, build the models, create the connections and start mosaik. Exemplary scripts for building, grid etc. can be found in the `examples` folder.

Model data file

Information on number of models and their parameterization.

Listing 1: examples/data/model_data_scenario/model_data_building.json (01/24)

```
{
   "ems": [
      {
            "strategy": "HEMS_default",
            "cs_strategy": "balanced"
      }
   ],
   "HouseholdCSV": [
      {
            "p_rated": 4500,
            "p_rated_profile": 4000,
            "cos_phi": 1.0,
            "datafile": "examples/data/load/4_persons_profile/load_34.csv",
            "date_format": "YYYY-MM-DD HH:mm:ss",
            "header_rows": 2,
            "start_time": "2014-01-01 00:00:00"
```

(continues on next page)

test_scenario_building.py

examples > test_scenario_building.py > ...

```python
# define paths and filenames
DIR = sim_help.get_default_dirs(
    os.path.realpath(os.path.dirname(__file__)), scenario="building", grid=None, format_db="hdf5"
)

# print paths
_logger.info(f"Selected model data: {DIR['MODEL_DATA']}")
_logger.info(f"Results will be stored in {DIR['DATABASE_FILENAME']}")


# Sim config.: Simulators and their used model types with the properties to store into DB
SIM_CONFIG = {
    # used database, will be left out for model creation and connections
    "DBSim": {"python": "eelib.data.database.hdf5:Hdf5Database"},
    # all the used simulators and their models for this simulation
    "EMSSim": {
        "python": "eelib.core.control.EMS.EMS_simulator:Sim",
        "models": {"ems": ["p_balance", "q_balance", "p_th_balance", "p_th_dem"]},
    },
    "CSVSim": {
        "python": "eelib.data.csv_reader.csv_reader_simulator:Sim",
        "models": {
            "HouseholdCSV": ["p", "q"],
            "PvCSV": ["p", "q"],
            "ChargingStationCSV": ["p", "q"],
            "HeatpumpCSV": ["p_el", "q_el"],
            "HouseholdThermalCSV": ["p_th_room", "p_th_water"],
        },
    },
    "CSSim": {
        "python": "eelib.core.devices.charging_station.charging_station_simulator:Sim",
        "models": {"ChargingStation": ["p"]},
    }
```

Fig. 1: examples/test_scenario_building.py (01/24)

(continued from previous page)

```
    }
  ],
...
```

Model connections

- Connections between grid buses and the ems models (or directly the devices)
- Connections between ems and the devices

Listing 2: examples/data/grid/grid_model_config.json (01/24)

```
{
  "0-load_1_1": {
     "ems": "HEMS_default_0",
     "load": [
           "HouseholdCSV_0"
     ],
     "household_thermal": [],
     "pv": [],
     "bss": [],
     "hp": [],
     "cs": [
           "ChargingStation_0"
     ],
     "ev": [
           "EV_0"
     ]
  },
  "0-load_1_2": {
     "ems": "HEMS_default_1",
     "load": [
           "HouseholdCSV_1"
     ],
...
```

Grid file

In `.json` format (possibly created via pandapower)

Listing 3: examples/data/grid/example_grid_kerber.json (01/24)

```
{
 "_module": "pandapower.auxiliary",
 "_class": "pandapowerNet",
 "_object": {
   "bus": {
     "_module": "pandas.core.frame",
     "_class": "DataFrame",
     "_object": "{\"columns\":[\"name\",\"vn_kv\",\"type\",\"zone\",\"in_service\"],\
↪"index\":[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17],\"data\":[[\"Trafostation_OS\",
```

(continues on next page)

(continued from previous page)


```
↪10.0,\"b\",null,true],[\"main_busbar\",0.4,\"b\",null,true],[\"MUF_1_1\",0.4,\"n\",
↪null,true],[\"loadbus_1_1\",0.4,\"b\",null,true],[\"KV_1_2\",0.4,\"b\",null,true],[\
↪"loadbus_1_2\",0.4,\"b\",null,true],[\"MUF_1_3\",0.4,\"n\",null,true],[\"loadbus_1_3\",
↪0.4,\"b\",null,true],[\"KV_1_4\",0.4,\"b\",null,true],[\"loadbus_1_4\",0.4,\"b\",null,
↪true],[\"MUF_1_5\",0.4,\"n\",null,true],[\"loadbus_1_5\",0.4,\"b\",null,true],[\"KV_1_
↪6\",0.4,\"b\",null,true],[\"loadbus_1_6\",0.4,\"b\",null,true],[\"MUF_2_1\",0.4,\"n\",
↪null,true],[\"loadbus_2_1\",0.4,\"b\",null,true],[\"KV_2_2\",0.4,\"b\",null,true],[\
↪"loadbus_2_2\",0.4,\"b\",null,true]]}",
      "orient": "split",
      "dtype": {
        "name": "object",
        "vn_kv": "float64",
        "type": "object",
        "zone": "object",
        "in_service": "bool"
      }
    },
...
```


Tip: All files can be created (more easily) with a *Scenario Configurator* (`.ipynb`) via an excel file. Or use existing files and adapt the parameterization.

2.5.2 Configuration of a Scenario Script

Note: All of these code-blocks derive from `examples/test_scenario_building.py` as of (01/24) if not stated otherwise.

Setup

Listing 4: `import` of used packages

```python
import os
import json
import mosaik
import mosaik.util
import eelib.utils.simulation_helper as sim_help
from eelib.model_connections.connections import get_default_connections
from eelib.utils.logging_helpers import set_console_logger
import arrow
import logging
```

Listing 5: Setting of paths for simulation data and used model simulators

```python
# define paths and filenames
DIR = sim_help.get_default_dirs(
```

(continues on next page)

(continued from previous page)

```
    os.path.realpath(os.path.dirname(__file__)), scenario="building", grid=None, format_
↪db="hdf5"
)
```

Listing 6: Define simulators and models for the simulation

```
# Sim config.: Simulators and their used model types with the properties to store into DB
SIM_CONFIG = {
   # used database, will be left out for model creation and connections
   "DBSim": {"python": "eelib.data.database.hdf5:Hdf5Database"},
   # all the used simulators and their models for this simulation
   "EMSSim": {
      "python": "eelib.core.control.EMS.EMS_simulator:Sim",
      "models": {"ems": ["p_balance", "q_balance", "p_th_balance", "p_th_dem"]},
   },
   "CSVSim": {
      "python": "eelib.data.csv_reader.csv_reader_simulator:Sim",
      "models": {
            "HouseholdCSV": ["p", "q"],
            "PvCSV": ["p", "q"],
            "ChargingStationCSV": ["p", "q"],
            "HeatpumpCSV": ["p_el", "q_el"],
            "HouseholdThermalCSV": ["p_th_room", "p_th_water"],
      },
   },
   ...
```

Listing 7: Configure time/steps, model data and conncetions and hand SIM_CONFIG to mosaik

```
# Configuration of scenario: time and granularity
START = "2020-01-01 00:00:00"
END = "2020-01-04 00:00:00"
STEP_SIZE_IN_SECONDS = 900  # 1=sec-steps, 3600=hour-steps, 900=15min-steps, 600=10min-
↪steps
N_SECONDS = int(
    (
        arrow.get(END, "YYYY-MM-DD HH:mm:ss") - arrow.get(START, "YYYY-MM-DD HH:mm:ss")
    ).total_seconds()
)
N_STEPS = int(N_SECONDS / STEP_SIZE_IN_SECONDS)
scenario_config = {
    "start": START,  # time of beginning for simulation
    "end": END,  # time of ending
    "step_size": STEP_SIZE_IN_SECONDS,
    "n_steps": N_STEPS,
    "bool_plot": False,
}

# Read Scenario file with data for model entities
with open(DIR["MODEL_DATA"]) as f:
```

(continues on next page)

(continued from previous page)

```
    model_data = json.load(f)

# Read configuration file with data for connections between prosumer devices
model_connect_config = get_default_connections()

# Create world
world = mosaik.World(SIM_CONFIG, debug=True)
```

Start Simulators

Listing 8: Simulators for the used models

```
# start all simulators/model factories with mosaik for data given in SIM_CONFIG
dict_simulators = sim_help.start_simulators(
    sim_config=SIM_CONFIG, world=world, scenario_config=scenario_config
)
```

Initiate Models

Listing 9: Create and collect the model entities for each device type

```
# create all models based on given SIM_CONFIG
dict_entities = sim_help.create_entities(
    sim_config=SIM_CONFIG, model_data=model_data, dict_simulators=dict_simulators
)
```

Connect Entities

Listing 10: The connections for each entity are listed in `model_connect_config`. Now tell mosaik.

```
# connect all models to each other
sim_help.connect_entities(
    world=world,
    dict_entities=dict_entities,
    model_connect_config=model_connect_config,
    dict_simulators=dict_simulators,
)
```

Run Simulation

```
world.run(until=scenario_config["n_steps"], print_progress=True)
```

2.5.3 Running a simulation

- **You can run one of the `test_scenario` s in the `examples` folder**
  `building`: Just one single building to see the operation of devices inside the household.

  `grid`: Simple low voltage grid (2 feeders with six resp. two household connection points) to get an estimation of the impact of different operating strategies on the local grid.

  `multi_fam_house`: TBD

  `residential_district`: TBD
- Adapting the parameterization in the *simulation files* can yield quite different results.
- Running one of the simulations will create a `.hdf5` results data file in the folder `/examples/results`.
- You can view the information of this file via the H5Web Extension in Microsoft VSC and plot the profiles (stored under Series and the name of the corresponding simulator) of the used devices.

2.5.4 Create your own simulation

1. Copy one of the *test scenarios* and delete all of the redundant simulators/devices/connections.
2. Set up corresponding *model* (and *grid*) data as well as a *model configuration file*.

2.6 Configuration of a Model

This page explains the structure of a `model.py` to enable you to create your own models. The `model.py` defines the properties and methods of instances of the respective model. Within a scenario, you may have multiple e.g. electric vehicles with differing property values, configured within the `model_data.json`. A `model.py` is always paired with a *simulator.py*.

Note: All code-blocks derive from `charging_station_model.py` as of (01/24) if not stated otherwise.

2.6.1 Introduction and imports

Listing 11: Basic explanation about the models and `import` relevant packages.

```
"""
eElib charging station model is built to manage the charging processes of EVs.

Author: elenia@TUBS
"""

import warnings
```

(continues on next page)

(continued from previous page)

```python
import math

from eelib.utils.ancillary_services.voltage_control_concepts import cos_phi_fix
```

2.6.2 Class definition

Listing 12: Short explanation, listing of parameters with their allowed values for initialization (+ method to `return` them)

```python
class ChargingStation:
    """Models a charging station for electric vehicles of different types."""

    # Valid values and types for each parameter
    _VALID_PARAMETERS = {
        "p_rated": {"types": [int], "values": (0, math.inf)},
        "output_type": {"types": [str], "values": ["AC", "DC"]},
        "charge_efficiency": {"types": [float, int], "values": (0, 1)},
        "discharge_efficiency": {"types": [float, int], "values": (0, 1)},
        "cos_phi": {"types": [float, int], "values": (0, 1)},
    }

    @classmethod
    def get_valid_parameters(cls):
        """Returns dictionary containing valid parameter types and values.

        Returns:
            dict: valid parameters for this model
        """
        return cls._VALID_PARAMETERS
```

2.6.3 Initialization of model properties

Listing 13: To initialize static properties and input-output-values of the model (to not set them in `init` function)

```python
# dynamic INPUT properties
appearance = {}  # Identifier if car at station [-]
e_bat = {}  # Actual charging level [kWh]
e_bat_max = {}  # Maximal capacity [kWh]
p_charge_max = {}  # Maximal charging active power [W]
p_discharge_max = {}  # Maximal discharging active power [W]
appearance_end_step = {}  # index for ending point of standing time [-]
bev_consumption_period = {}  # Electricity consumption of EV in next period not being␣
↪home,

# control signals & charging strategy values
```

(continues on next page)

(continued from previous page)

```
p_set = {}  # active power set-point [W]

# dynamic OUTPUT properties
p = 0  # Active Power (after control) [W]
q = 0  # Reactive Power (after control) [W]
p_device = {}  # active power for every vehicle [W]
p_min = 0  # Minimal active Power [W]
p_max = 0  # Maximal active Power [W]

efficiency = 1.0  # efficiency of the current time step for the charging station
```

2.6.4 Initialization method `__init__()`

Listing 14: takes parameter values as inputs

```
def __init__(
    self,
    ename: str,
    p_rated: int,
    output_type: str = "AC",
    charge_efficiency: float = 0.99,
    discharge_efficiency: float = 0.99,
    cos_phi: float = 1.0,
    step_size=60 * 15,  # step size in seconds
):
```

Listing 15: creates entity of this model

```
    # Set attributes of init_vals to static properties
    self.ename = ename
    self.p_rated = p_rated  # rated active power (AC/DC) [W]
    self.output_type = output_type  # source AC/DC [-]
    self.discharge_efficiency = discharge_efficiency  # discharging efficiency [-]
    self.charge_efficiency = charge_efficiency  # charging efficiency [-]
    self.cos_phi = cos_phi

    # edit efficiency rates
    if self.output_type == "AC" and (charge_efficiency != 1.0 or discharge_efficiency != 1.0):
        self.charge_efficiency = 1
        self.discharge_efficiency = 1
        warnings.warn(
            "WARNING: Efficiency of AC charging is instead set to 1!",
            UserWarning,
        )

    # save time step length and current time step
    self.step_size = step_size
    self.time = 0
```

2.6.5 Model methods

Listing 16: Type-specific function like calculation of power limits, aging, efficiency, adaption of stored energy etc.

```
def _calc_power_limits(self):
    """Calculate the power limits for the charging station with the input thats coming from the
    electric vehicles.

    Raises:
        ValueError: If the power limits of at least one connected ev do not work together.
    """

    # current efficiency depending on the direction of power flow
    self._calc_current_efficiency()

    # in case no ev is connected to cs - no active power flexibility
    self.p_min = 0
    self.p_max = 0
    for ev_id, ev_appearance in self.appearance.items():
        # check for each ev if connected - consider their limits, efficiency and nominal power
        if ev_appearance:
            # check if min. and max. power are correct
            if self.p_discharge_max[ev_id] > self.p_charge_max[ev_id]:
                raise ValueError(f"Min. and max. power of ev {ev_id} do not comply.")
            # handle the power limits
            self.p_min = max(
                self.p_min + self.p_discharge_max[ev_id] / self.efficiency,
                -self.p_rated,
            )
            self.p_max = min(
                self.p_max + self.p_charge_max[ev_id] / self.efficiency,
                self.p_rated,
            )
...
```

2.6.6 `step()` method

The `step()` method of `storage_model.py` (01/24).

Listing 17: For handling of the processes of the model within a time step

```python
def step(self, time):
    """Performs simulation step of eELib battery model.
    Calculates all of the Dynamic Properties based on the Input Properties.

    Args:
        time (int): Current simulation time
    """
```

Listing 18: At first: Handling of a new time step (if entity was called for first time, do some processes once, like adapting energy)

```python
    # handle current time step
    if not self.time == time:
        self.time = time

        # adapt energy content from last time step ( + self-discharge)
        e_bat_self_discharge = -(self.e_bat * self.loss_rate) * (
            self.step_size / (60 * 60 * 24 * 30)
        )
        if self.p >= 0:  # charging
            self.e_bat_step_volume = (
                self.p * self.charge_efficiency * (self.step_size / 3600) + e_bat_self_discharge
            )
        else:  # discharging
            self.e_bat_step_volume = (
                self.p / self.discharge_efficiency * (self.step_size / 3600)
                + e_bat_self_discharge
            )
        self.e_bat += self.e_bat_step_volume

        # Calculate battery cycles
        self.bat_cycles += abs(self.e_bat_step_volume / self.e_cycle)

        # Calculate battery state of health and aging properties
        if self.status_aging:
            self.__calculate_aging_status()
```

Listing 19: Call model-specific methods in supposed order

```python
    # Set active power and energy within limits
    self.__set_power_within_limit()
    self.__set_energy_within_limit()
    self.soc = self.e_bat / self.e_bat_usable

    self.__calc_charging_efficiency()

```

(continues on next page)

(continued from previous page)

```
    self.__calc_power_limits()
```

2.6.7 Checklist for adding / enhancing a model

What changes?

adapting current implementation?
Try to make use of the existing properties and methods of the model

adding new implementation (e.g. new method) or need for new properties?

1. Add the part of code to the model
2. Write proper comments and documentation (docstrings for every method!)
3. Write a corresponding test function!

New packages have been added?
add them to the `requirements.txt` file

Where to add?

New model attributes need to be …

1. … added to the `META` of the simulator.
2. If they are also input data, add them to the `model_data` of the test scenarios (`examples/data/model_data_scenario`) as well as the `VALID_PARAMETERS` of the model.

New connections between properties of different models …
… need to be integrated to the `model_connections/model_connect_config.json` file, simply paste them in the direction of the connection.

New models need to be integrated …

1. … into the test scenario scripts (in the `SIM_CONFIG`).
2. … into the `model_data` of the test scenarios (`examples/data/model_data_scenario`).
3. … into the `model_connections/model_connect_config.json` file with their connections to/from other models. If the direction of the connection is of importance, there may be a need to adapt the `simulation_helper` functions `connect_entities()` and `connect_entities_of_two_model_types()`.
4. … into the simulator `META` data.
5. … with a unit test file that checks all the relevant model functionalities.

2.7 Configuration of a Simulator

This page explains the structure of a `simulator.py` to enable you to create your own simulators. The `simulator.py` stores information about the connected models and mediates between them and mosaik (e.g. passing the order to `step()` as well as associated values). As there may be multiple instances of a model connected to a single simulator, you only need one simulator of a type per scenario, set within the `SIM_CONFIG` of scenario file. A `simulator.py` is always paired with a *model.py*.

Note: All code-blocks derive from `charging_station_simulator.py` as of (01/24) if not stated otherwise.

2.7.1 Introduction and imports

Listing 20: Basic explanation `import` of relevant packages

```
"""
Mosaik interface for the eELib charging station model.
Simulator for communication between orchestrator (mosaik) and charging station entities.

Author: elenia@TUBS
"""

import mosaik_api_v3
from eelib.core.devices.charging_station import charging_station_model
import eelib.utils.validation as vld
from copy import deepcopy
```

2.7.2 Listing of model `META`

Listing 21: State of simulator (whether it is time-discrete or event-based/hybrid)

```
META = {
    "type": "hybrid",
```

Listing 22: Listing of **ALL** attributes for each modeltype: At first, all attributes . . .

```
    "models": {
        "ChargingStation": {
            "public": True,
            "params": ["init_vals"],
            "attrs": [
                "type",
                "output_type",
                "step_size",
                "time",
                ...
```

Listing 23: … **input attributes** also listed in "trigger" list …

```
        "trigger": [
           "p_set",
           "e_bat",
           "e_bat_max",
           "p_charge_max",
           "p_discharge_max",
           "appearance",
           "appearance_end_step",
           "bev_consumption_period",
        ],  # input attributes
```

Listing 24: … **output attributes** also listed in ""non-persistent" list.

```
        "non-persistent": [
           "p_min",
           "p_max",
           "appearance_end_step",
           "p_device",
           "p",
           "discharge_efficiency",
           "charge_efficiency",
           "e_bat",
           "e_bat_max",
        ],  # output attributes
```

2.7.3 Initialization of simulator class

Listing 25: Short explanation as well as constructor __init__ for this simulator and the initialization function init()

```
class Sim(mosaik_api_v3.Simulator):
    """Simulator class for eELib charging station model.

    Args:
        mosaik_api_v3 (module): defines communication between mosaik and simulator

    Raises:
        ValueError: Unknown output attribute, when not described in META of simulator
    """

    def __init__(self):
        """Constructs an object of the Charging-Station:Sim class."""

        super(Sim, self).__init__(META)

        # storing of event-based output info (for same-time loop or next time step)
```

(continues on next page)

(continued from previous page)

```
        self.output_cache = {}

        # initiate empty dict for model entities
        self.entities = {}

    def init(self, sid, scenario_config, time_resolution=1.0):
        """Initializes parameters for an object of the Charging-Station:Sim class.

        Args:
            sid (str): Id of the created instance of the simulator (e.g. CSSim-0)
            scenario_config (dict): scenario configuration data, like resolution or step size
            time_resolution (float): Time resolution of current scenario.

        Returns:
            meta: description of the simulator
        """

        # assign properties
        self.sid = sid
        self.scenario_config = scenario_config

        return self.meta
```

Following are the so called **core functions** `create()`, `step()` and `get_data()` that are found in every `simulator.py`.

> **Caution:** All **core functions** of the simulators are called by mosaik and should not be deleted!

2.7.4 Creation of model entities in `create()`

Listing 26: Core functions should also be labeled in their docstring

```
def create(self, num, model_type, init_vals):
    """Creates entities of the eELib charging station model.
    Core function of mosaik.

    Args:
        num (int): Number of cs models to be created
        model_type (str): type of created instance (e.g. "charging_station")
        init_vals (list): list with initial values for each charging_station entity

    Returns:
        dict: created entities
    """
```

Listing 27: Creation of entities by assigning individual entity names and calling the initialization method of that model type

```
# generated next unused ID for entity
next_eid = len(self.entities)

# create empty list for created entities
entities_orchestrator = []

for i in range(next_eid, next_eid + num):
    # create entity by specified name and ID
    ename = "%s%s%d" % (model_type, "_", i)
    full_id = self.sid + "." + ename

    # get class of specific model and create entity with init values after validation
    entity_cls = getattr(charging_station_model, model_type)
    vld.validate_init_parameters(entity_cls, init_vals[i])
    entity = entity_cls(
        ename,
        **init_vals[i],
        step_size=self.scenario_config["step_size"],
    )
```

Listing 28: Simulator stores information about the entities

```
    # add info to the simulators entity-list and current entities
    self.entities[ename] = {
        "ename": ename,
        "etype": model_type,
        "model": entity,
        "full_id": full_id,
    }
    entities_orchestrator.append({"eid": ename, "type": model_type})

return entities_orchestrator
```

2.7.5 Stepping of models in `step()`

The `step()` method of `storage_simulator.py` (01/24).

Listing 29: First take input data (from mosaik) and set values of entities

```
# assign property values for each entity and attribute with entity ID
# process input signals: for the entities (eid), attr is a dict for attributes to be set
for eid, attrs in inputs.items():
    # for the attributes (attr), setter is a dict for entities with corresponding set values
    for attr, setter in attrs.items():
        # for transmitter (eid_setter), value_dict contains set values (with ids, when dict)
```

(continues on next page)

(continued from previous page)

```
        setting_value_dict = deepcopy(getattr(self.entities[eid]["model"], attr))
        for eid_setter, value_dict in setter.items():
            if isinstance(value_dict, dict):
                # go by each id and search for corresponding entity id
                for getter_id, value in value_dict.items():
                    if eid in getter_id:
                        setting_value_dict[eid_setter] = value
            # value_dict is not a dict, only a single value -> write directly
            elif isinstance(value_dict, (float, int)):
                setting_value_dict[eid_setter] = value_dict
            else:
                raise TypeError("Unknown format for value_dict")
        setattr(self.entities[eid]["model"], attr, setting_value_dict)

        # check if there is more than one power set value - otherwise directly set it
        if attr == "p_set":
            if len(setting_value_dict) > 1:
                raise ValueError("There is more than one power set value for " + eid)
```

Listing 30: Then simply step each model for this time step

```
# call step function for each entity in the list
for ename, entity_dict in self.entities.items():
    entity_dict["model"].step(time)
```

Note: You might `return` the next timestep for when this model should be called again.

2.7.6 Handling of output data in `get_data()`

Listing 31: From a defined set of output properties, the values of the transmitting entities are read and stored into data dict

```
for transmitter_ename, attrs in outputs.items():
    # get name for current entity and create dict field
    entry = self.entities[transmitter_ename]
    if transmitter_ename not in self.output_cache:
        self.output_cache[transmitter_ename] = {}

    # loop over all targeted attributes and check if info is available
    for attr in attrs:
        if attr not in self.meta["models"][type(entry["model"]).__name__]["attrs"]:
            raise ValueError("Unknown output attribute: %s" % attr)

        # create empty field for cache and output data
        if attr not in self.output_cache[transmitter_ename]:
            self.output_cache[transmitter_ename][attr] = {}

        output_data_to_save = getattr(entry["model"], attr)
```

Listing 32: For each time step, the output data is continuously stored and compared to the lastly sent (output_cache) such that if nothing new is to be send out, only the time step will be send

```
# check if nothing is to be send out - send output 1 step later to avoid waiting for data
if not flag_output_changed:
    if self.time == self.scenario_config["n_steps"] - 1:  # is last time step?
        data["time"] = self.time + 1
    else:
        data = {"time": self.time}
```

2.8 Implementing an EMS strategy

Energy Management Strategies are handled by Energy Management Systems (EMS). To implement a new strategy, additions in multiple places are necessary.

2.8.1 1. Adapt the `EMS_model.py` file and implement the operating strategy

Create a new class that is inheriting from the general `HEMS` class.

Listing 33: eelib/core/control/EMS/EMS_model.py (01/24)

```
class HEMS_default(HEMS):
    """Default strategy for Energy Management System.
    Should be copied and adapted for the use of a specific EMS concept.
    """

    @classmethod
    def get_valid_parameters(cls):
        """Returns dictionary containing valid parameter types and values.

        Returns:
            dict: valid parameters for this model
        """

        # use parent's parameter list and modify them for this class
        result = HEMS.get_valid_parameters().copy()
        result.update({})
        return result

    def __init__(self, ename: str, step_size: int = 900, **kwargs):
        """Initializes the eELib HEMS default model.

        Args:
            ename (str): name of the entity to create
            step_size (int): length of a simulation step in seconds
            **kwargs: initial values for the HEMS entity

        Raises:
            ValueError: Error if selected strategy does not comply with model type.
```

(continues on next page)

(continued from previous page)

```python
        """

        # check given strategy
        if "strategy" in kwargs.keys() and kwargs["strategy"] != "HEMS_default":
            raise ValueError("Created a HEMS_default entity with strategy not 'HEMS_default'!")
        else:
            kwargs["strategy"] = "HEMS_default"  # set strategy if not already given

        # call init function of super HEMS class
        super().__init__(ename=ename, step_size=step_size, **kwargs)
```

Add a `step()` function that first calls the HEMS `step()` and afterwards implement the functionalities of your operating strategy

Listing 34: eelib/core/control/EMS/EMS_model.py (01/24)

```python
def step(self, time):
    """Calculates power set values for each connected component according to the strategy.

    Args:
        time (int): Current simulation time
    """

    # execute general processes (aggregation of power values etc.)
    super().step(time)

    # from here on: execute strategy-specific processes
    ...
```

2.8.2 2. Add the strategy and its input to the `model_data` of the scenarios

Listing 35: examples/data/model_data_scenario/model_data_building.json (01/24)

```json
{
    "ems": [
        {
            "strategy": "HEMS_default",
            "cs_strategy": "balanced"
        }
    ],
    ...
```

2.8.3 3. Add your EMS class to the `model_connections/model_connect_config.json` file

Add the data that is **sent out**. . .

Listing 36: eelib/model_connections/model_connect_config.json (01/24)

```
"HEMS_default": {
    "HouseholdCSV": [],
    "PvCSV": [],
    "PVLib": [["p_set_pv", "p_set"]],
    "PVLibExact": [["p_set_pv", "p_set"]],
    "BSS": [
        [
            "p_set_storage",
            "p_set"
        ]
    ],
    "ChargingStation": [
        [
            "p_set_charging_station",
            "p_set"
        ]
    ],
    "ChargingStationCSV": [],
    "EV": [],
    "HouseholdThermalCSV": [],
    "HeatpumpCSV": [],
    "Heatpump": [
        [
            "p_th_set_heatpump",
            "p_th_set"
        ]
    ],
    "grid_load": [["p_balance", "p_w"], ["q_balance", "q_var"]]
},
```

. . . but also to every model, which **data is sent to your HEMS!**

Listing 37: e.g. but not only the BSS

```
"BSS": {
    "HEMS_default": [
        [
            "p_discharge_max",
            "p_min"
        ],
        [
            "p_charge_max",
            "p_max"
        ],
        [
            "p",
```

(continues on next page)

(continued from previous page)

```
            "p"
        ]
    ],
    "HouseholdCSV": [],
    "PvCSV": [],
    "PVLib": [],
    "PVLibExact": [],
    "ChargingStation": [],
    "ChargingStationCSV": [],
    "EV": [],
    "HouseholdThermalCSV": [],
    "HeatpumpCSV": [],
    "Heatpump": []
},
```

As you can see, also connections that **do not share data are added**.

2.8.4 4. Add the model with its name and (input/output) attributes to the `META` of the `EMS_simulator`

Listing 38: eelib/core/control/EMS/EMS_simulator.py (01/24)

```
META = {
    "type": "hybrid",
    "models": {
```

```
        "HEMS_default": {
            "public": True,
            "params": ["init_vals"],
            "attrs": [
                "q",
                "p",
                "p_max",
                "p_min",
                "p_set_storage",
                "p_set_charging_station",
                "p_set_pv",
                "p_balance",
                "q_balance",
                "appearance_end_step",
                "discharge_cs_efficiency",
                "charge_cs_efficiency",
                "e_bat_car",
                "e_bat_max_car",
                "p_th_room",
                "p_th_water",
                "p_th_dem",
                "p_th_dev",
                "p_th_balance",
```

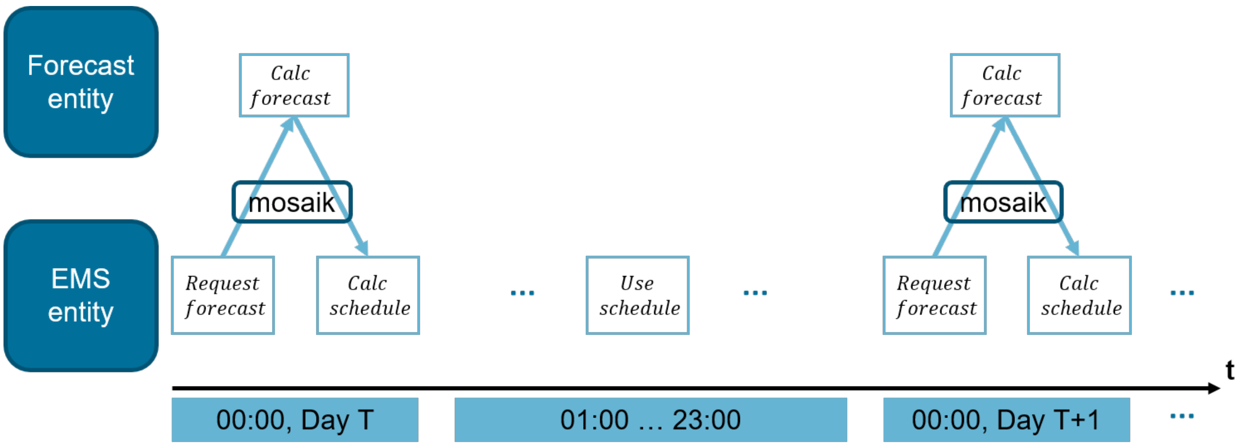
(continues on next page)

(continued from previous page)

```
                "p_th_min",
                "p_th_max",
                "p_th_min_on",
                "p_th_set_heatpump",
            ],
```

2.9 Forecasts and Schedules

Implemented into the eELib is a possibility to calculate a *forecast* and *schedule* for devices, EMSs, and the grid. For this, a forecast model is given to calculate requested forecasts. Additionally, the calculation of schedules can be implemented in each (control) model.



2.9.1 Integration into scenario

Forecasts and schedules can be integrated into simulations like the `examples/test_scenario_***.py` files exemplarily show. In there, one has to declare the forecast simulator to mosaik via the `SIM_CONFIG`. Additionally, one has to set the parameter `USE_FORECAST` in the scenario configuration to true.

Listing 39: simulator configuration [test_scenario_building.py 01/24]

```
# Sim config.: Simulators and their used model types with the properties to store into DB
SIM_CONFIG = {
   # used database, will be left out for model creation and connections
   "DBSim": {"python": "eelib.data.database.hdf5:Hdf5Database"},
   # forecast, will be left out for model creation and connections
   "ForecastSim": {
      "python": "eelib.core.control.forecast.forecast_simulator:Sim",
      "models": {"Forecast": []},
   },
```

Listing 40: scenario configuration [test_scenario_building.py 01/24]

```
# Configuration of scenario: time and granularity
START = "2020-01-01 00:00:00"
END = "2020-01-04 00:00:00"
STEP_SIZE_IN_SECONDS = 900  # 1=sec-steps, 3600=hour-steps, 900=15min-steps, 600=10min-
↪steps
USE_FORECAST = True
```

After this, the forecast simulator and model will be created and the connections to the models will be instantiated.

2.9.2 Forecast Model

The forecast model and its simulator are implemented in the folder `eelib/core/control/forecast`. The implemented behaviour is quite simple and straight-forward, as all model entities of the simulation are stored within the forecast entity (as deep-copies) via the `add_forecasted_entity()` method. This allows to create forecasts by simply iterating over all requested forecasts for all entities, stepping the model for the requested timesteps and collecting the values (of the model entity) for the requested time steps. After that, the forecast model simply returns those calculated forecasts.

Listing 41: forecast calculation [forecast_model.py 01/24]

```
# check if request for a forecast was sent
if self.forecast_request == {}:
   self.forecast = {}  # no forecast requested, simply return
else:
   # clear earlier forecasts
   self.forecast = {}

   # go by all entities to create a forecast for
   for forecast_getter_id, forecast_req_eid_dict in self.forecast_request.items():
      # create empty dict for forecasts connected to this request entity
      self.forecast[forecast_getter_id] = {}
      # check if no forecast requested
      if forecast_req_eid_dict == {}:
         continue

      # go by all forecasted model entities
      for forecast_eid, forecast_info in forecast_req_eid_dict.items():
         # check if forecast can be done for this model type
         if forecast_eid in self.forecast_eid.keys():
            # create structure for forecast of each attribute for this entity
            forecast_save = {}
            for attr in forecast_info["attr"]:
               forecast_save[attr] = []

            # store the copy of this model entity to execute the stepping
            entity = self.forecast_eid[forecast_eid]

            # run the model for each time step and collect the calculated attr values
            for t in forecast_info["t"]:
               entity.step(t)
```

(continues on next page)

(continued from previous page)

```
            for attr in forecast_info["attr"]:
                forecast_save[attr].append(getattr(entity, attr))
```

2.9.3 Integration into (Control) Models

For the forecast model to take effect, the forecasts have to be requested by other models, e.g. the energy management system. Additionally, the calculated forecasts should afterwards be used for strategic behaviour (in operating strategies).

First of all, the mosaik needs to create a connection between the model and the forecast model. This is done in the `connect_to_forecast()` function of the simulation helpers. Here, there is a connection added from EMS to the forecast model for the `forecast_request` attribute, while a `forecast` attribute is send back the other way around. All other models are simply added to the forecast entity such that forecasts can be calculated.

Listing 42: forecast connection function [simulation_helper.py 01/24]

```
def connect_to_forecast(
    world: object,
    dict_entities: dict,
    dict_simulators: dict,
    forecast: object,
    forecast_sim: object,
):
    """Create connections for the forecasts to work.
    Includes mosaik connections to ems model and adding of the model entities to the␣
↪forecasts list.

    Args:
        world (object): mosaik world object to orchestrate the simulation process
        dict_entities (dict): dict of all used model entity objects
        dict_simulators (dict): dict of all used simulators with their ModelFactory-objects
        forecast (object): forecast model entity
        forecast_sim (object): simulator for the forecast model
    """

    # create connections for each entity of each model type
    for model_name, ent_list in dict_entities.items():
        for entity in ent_list:
            # for ems create connections to forecast entity
            if "ems" in model_name or "EMS" in model_name:
                world.connect(entity, forecast, "forecast_request")
                world.connect(
                    forecast,
                    entity,
                    "forecast",
                    weak=True,
                    initial_data={"forecast": {forecast.full_id: {}}},
                )
            # for other models (devices) add those entities to the forecast entity list
            else:
                forecast_sim.add_forecasted_entity(
                    forecast.eid,
```

(continues on next page)

(continued from previous page)

```
                {entity.full_id: dict_simulators[model_name].get_entity_by_id(entity.eid)}
↪,
            )
```

Additionally, forecasts have to be requested by the ems, which should be done only in the corresponding time steps.

Listing 43: forecast request by EMS [EMS_model.py 01/24]

```python
# request forecasts if needed
if self.use_forecast and self.calc_forecast:
    self.forecast_request = {}
    for model_type, entity_list in self.controlled_eid_by_type.items():
        if model_type in self.forecasted_attrs.keys():
            # add forecast request for every entity of this model type
            for e_full_id in entity_list.keys():
                self.forecast_request[e_full_id] = {
                    "attr": self.forecasted_attrs[model_type],
                    "t": range(self.forecast_start, self.forecast_end),
                }
```

Due to the connection by mosaik, the forecasts are calculated and afterwards send back, such that they should be processed. It is now also possible to calculate schedules with set values for the devices that no forecast can be directly extracted from (charging station, heatpump, battery).

Listing 44: forecast request by EMS [EMS_model.py 01/24]

```python
# CALC SCHEDULE WITH UNCONTROLLED (CSV) DEVICES
if self.forecast != {} and self.calc_forecast is True:
    # calculate the residual load schedule including all not controllable devices
    schedule_residual_uncontrollable = schedule_help.residual_calc_schedule_
↪uncontrollable(...)

    # calc schedules for charging station
    schedule_help.cs_calc_schedule_uncontrolled(...)

    # calc schedules for heatpump
    th_residual_forecast = schedule_help.thermal_calc_forecast_residual(...)
    schedule_help.hp_calc_schedule(...)

    # get schedule from battery storage based on residual load schedule
    schedule_help.bss_calc_schedule(...)
```

2.10 FAQ & Glossary

2.10.1 FAQ

Where are the instanciated models stored? How is the process with the building of Model-Factories?

```
examplesim = world.start("ExampleSim", eidprefix="Model")
```

is an entity of the class `mosaik.scenario.ModelFactory` and stores the entities of the example scenario within `_sim._inst`

What can one do in case of an ImportError when running the example scenarios?

`pip install -e .`

What attributes does a model have?

`META` of the simulator:

Listing 45: e.g. EMS_simulator.py (01/24)

```
# SIMULATION META DATA
META = {
    "type": "hybrid",
    "models": {
        "HEMS": {
            "public": True,
            "params": ["init_vals"],
            "attrs": [
                "q",
                "p",
                "p_max",
                "p_min",
                "p_th_room",
                "p_th_water",
                "p_th",
...
```

Optionally, within the model class itself

What inputs does a model have?

`VALID_PARAMETERS` of the model class:

Listing 46: e.g. EMS_model.py (01/24)

```
# Valid values and types for each parameter that apply for all subclasses
_VALID_PARAMETERS = {
    "strategy": {"types": [str], "values": ["HEMS_default"]},
    "cs_strategy": {
        "types": [str],
        "values": ["max_p", "balanced", "night_charging", "solar_charging"],
    },
    "bss_strategy": {
        "types": [None, str],
        "values": [None, "reduce_curtailment"],
    },
}
```

`model_data` of the test scenarios exemplary set these parameters and init values:

Listing 47: e.g. model_data_building.json (01/24)

```
{
    "ems": [
        {
            "strategy": "HEMS_default",
            "cs_strategy": "balanced"
        }
```

(continues on next page)

(continued from previous page)

```
    ],
```

What connections does a model have?

See the `model_connections/model_connect_config.json` file, where the `FROM-TO-CONNECTIONS` are given for each model type of the eELib:

Listing 48: e.g. model_connect_config.json (01/24)

```
"HEMS_default": {
    "HouseholdCSV": [],
    "PvCSV": [],
    "PVLib": [["p_set_pv", "p_set"]],
    "PVLibExact": [["p_set_pv", "p_set"]],
    "BSS": [
        [
            "p_set_storage",
            "p_set"
        ]
    ],
    "ChargingStation": [
        [
            "p_set_charging_station",
            "p_set"
        ]
    ],
    "ChargingStationCSV": [],
    "EV": [],
    "HouseholdThermalCSV": [],
    "HeatpumpCSV": [],
    "Heatpump": [
        [
            "p_th_set_heatpump",
            "p_th_set"
        ]
    ],
    "grid_load": [["p_balance", "p_w"], ["q_balance", "q_var"]]
},
```

Listing 49: e.g. model_connect_config.json (01/24)

```
"BSS": {
    "HEMS_default": [
        [
            "p_discharge_max",
            "p_min"
        ],
        [
            "p_charge_max",
            "p_max"
        ],
        [
            "p",
```

(continues on next page)

(continued from previous page)

```
                "p"
            ]
        ],
        "HouseholdCSV": [],
        "PvCSV": [],
        "PVLib": [],
        "PVLibExact": [],
        "ChargingStation": [],
        "ChargingStationCSV": [],
        "EV": [],
        "HouseholdThermalCSV": [],
        "HeatpumpCSV": [],
        "Heatpump": []
},
```

`BSS` sends `p`, `p_min` and `p_max` to `HEMS` and recieves `p_set`.

2.10.2 Glossary

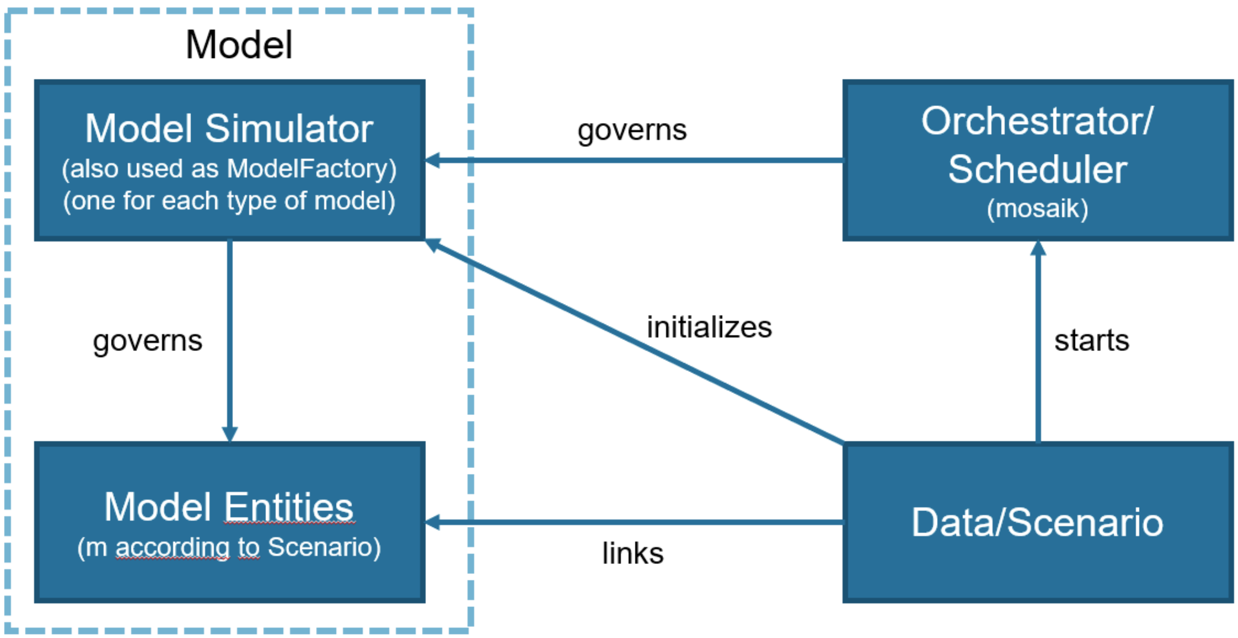


Fig. 2: This illustrates the different units participating in a simulation.

Orchestrator
: **Mosaik**: Coordination of the whole simulation and model coupling.

Simulator
: API for communication between orchestrator and the entities of the specific model.

Entity
: Created instance of a model type. In "ename", "etype", "eid" etc. the "e" stands for entity.

PSC
: We use the **passive sign convention** (german: **Verbraucherzählpfeilsystem**), therefore loads are positive while

generation is negative.

Forecast
Prediction of a behaviour for a defined time horizon in the future, e.g. power values for the upcoming 24h steps for a household base load from a csv reader.

Schedule
Calculated set values for a defined time horizon in the future, e.g. target power values for the upcoming 24h steps for a battery storage system.

2.10.3 Parameters used in test scenarios

`start_time`
simulation starting time (e.g. 2023-01-01 00:00:00)

`n_steps`
number of steps that should be simulated (for time-based calculations only) (e.g. 96 steps for a day with 15 min time steps)

`step_size`
length of one (pre-defined) time step (e.g. 15 min = 15*60 sec = 900 sec)

`end_time`
simulation ending time (e.g. 2023-12-31 23:59:59)

2.10.4 Units

Make use of the SI-units!!

- **Power**: W
- **Time (e.g. simulation time)**: s
- **Energy**: Wh
- …

CHAPTER

THREE

API REFERENCE

The API reference provides detailed descriptions of eElib's classes and methods. This is taken from the implementations of the models, which can be taken from the *public Gitlab-Repository <https://gitlab.com/elenia1/elenia-energy-library>*.

CHAPTER

FOUR

DISCLAIMER / AUTHORS

Author: elenia@TUBS

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex

INDEX