SOFTWARE DESCRIPTOR

SOFIRpy

Co-Simulation Of Functional Mock-up Units (FMUs) with Integrated Research Data Management

Daniele Inturri [1]
Kevin T. Logan [1]
Michaela Leštáková [1]
Tobias C. Meck [1]
Peter F. Pelz [1]

1. Chair of Fluid Systems, Technische Universität Darmstadt, Darmstadt.

Date Submitted:
2024-11-29

Keywords:
Python, Functional Mockup Units, control engineering, fluid systems, research software, research data management

Data availability:
There are no data associated with this submission.

Software availability:
Software can be found here:
https://github.com/fluid-systems/SOFIRpy

Corresponding Author:
Kevin T. Logan
kevin.logan@tu-darmstadt.de

Abstract. Optimising the operation of physical systems can lead to significant energy savings. This underscores the importance of researchers, planners and operators to focus on innovative control strategies. **SOFIRpy**, a framework for co-simulation of functional mock-up units (FMUs) with integrated research data management proposed in this paper, aims to assist them in studying and implementing these novel approaches. **SOFIRpy** provides tools for building FMUs from models of physical systems written in Modelica, implementing custom controllers, running co-simulations and performing research data management (RDM). It is a Python package hosted on PyPI, adhering to best practices in research software engineering.

1 Introduction

Fluid systems are responsible for up to 8 % of electrical energy consumption in the European Union [1], [2]. Some of them are used for fulfilling the most basic human needs, e.g., water distribution systems. Others play an essential role in waste water disposal and treatment or in industry, e.g. for cooling chemical processes. Thus, there is great potential for achieving the energy savings targets declared by the European Commission in the Energy Efficiency Directive [3] by focusing on fluid systems. Up to 20 % of the energy consumption can be saved by optimising control for an entire fluid system rather than focusing on component level control [4], which underscores the importance for researchers, planners and operators to focus on this approach. Simultaneously, it is crucial to keep other challenges faced by fluid systems in mind, such as their safe operation.

To support researchers, planners and operators innovating control methods for a wide range of physical systems – not just fluid systems – the software package **SOFIRpy** (Co-Simulation of Functional Mock-up Units with Integrated Research Data Management) presented in this work has been developed. It is a framework for co-simulation of custom controllers and physical systems, represented by Functional Mock-up Units (FMUs). **SOFIRpy** allows for implementing, comparing and analysing various control methods and configurations of physical systems. The software package includes features for easy-to-use research data management (RDM) as the importance of RDM for transparency and reproducibility of scientific results was considered from the beginning of the development. The development was motivated by a use-case in which distributed control of fluid systems using multi-agent control was studied. The design of the package was generalised from fluid systems to any physical system in order to ensure its usefulness beyond the motivating use-case.

1.1 Statement of need

Researchers, planners and operators need to optimise control of physical systems to realise the potential savings mentioned above. For this purpose, they need a tool that allows them to easily implement a variety of highly customised controllers and easily apply them to models of physical systems. This tool must allow them to run co-simulations of the controllers and the physical systems with a multitude of different configurations and parameters. The tool should provide the results of these simulations in an easily processable format that allows the users to analyse and visualise the data in order to evaluate the different control approaches. In order for them to keep track of the different simulation runs and configurations, the tool should provide integrated methods for RDM that allow users to trace and to reproduce individual runs. Additionally, this tool should be open source and not rely on proprietary dependencies. It should be possible to find and install the tool easily and incorporate it in common software setups.

Such a tool did not exist prior to the development of **SOFIRpy** to the best of the authors' knowledge. Some related tools are discussed in Section 1.2

Keeping the original use-case in mind but generalising the application, these requirements can be summarised in five workflow steps:

1. modelling physical systems
2. creating custom controllers
3. simulating controllers and physical systems with reciprocal dependencies
4. results analysis
5. data storage and RDM

Detailed requirements for the different steps are given in the following.

Users need to be able to easily create, configure and change models of the physical systems. They need a tool that can integrate representations of these systems and that provides simple interfaces for input of starting values and system configuration parameters. Users need to be able to run simulations of the physical systems without having to write their own solvers.

Users need a tool that allows them as much freedom as possible to implement innovative control methods with potentially highly customised algorithms. They need the tool to allow them to implement the control methods using a widely disseminated and broadly used programming language. Users further need the framework of the tool to place minimal requirements on the implementations.

Users need to run co-simulations of the controllers and the physical systems. This means that there needs to be a defined interface for controllers and physical systems, through which inputs and outputs of each of the entities are exchanged at defined intervals during a simulation run.

Users need to run large parameter studies for studying and evaluating the performance of the control methods. Accordingly, they need to easily set up and configure, as well as parametrise simulation runs.

After running the simulations, users need the tool to allow them to process the results using other established tools compatible with the programming language of the tool itself. Users need the tool to store the results in a file using a non-proprietary, interoperable format. They further need to be able to supplement the stored results with metadata containing information on the setup and start values of the simulation that gave the results. This serves the joint purposes of keeping track of the results for the users of the tool and making the results more transparent and comprehensible for subsequent users of the data.

Finally, users need to be able to reproduce results easily or rerun simulations with parameter variations. For this purpose, they need to be able to load all the data required to recreate the simulation setup from the file containing the results in such a way that the simulation can be rerun.

1.2 State of the art

While there are no software tools that can address all the workflow steps and requirements outlined above according to the best of the authors' knowledge, several tools exist for individual ones. Some of these tools are among the dependencies of **SOFIRpy**.

With regard to the first workflow step (modelling physical systems), **SOFIRpy** expands on OMPython, the OpenModelica Python API for building, compiling and solving Modelica models [5]. **SOFIRpy** uses OMPython in its dependencies for Modelica FMU exports.

FMPy is a library for simulating FMUs in Python, which also supports co-simulation and model exchange [6]. **SOFIRpy** uses FMPy to perform part of the third workflow step (simulating physical systems), but expands on it with the option of running co-simulations with the custom controllers written in Python. FMPy is included in the dependencies of **SOFIRpy**.

The HILO-MPC package offers a broad range of possibilities for modelling, estimation problems, and control based on optimisation, model predictive control (MPC) as well as machine learning [7]. HILO-MPC addresses the first three workflow steps of **SOFIRpy**. It requires users to explicitly formulate the equations that describe the physical system they are studying, whereas **SOFIRpy** allows users to use FMUs to represent the physical systems. Functionalities for RDM are not included in HILO-MPC.

There are two frameworks that address several of the workflow steps of **SOFIRpy**. ETA Utility provides a framework for optimisation, simulation and communication with physical devices [8]. Similarly to **SOFIRpy**, ETA Utility builds on FMPy to perform simulations. The focus of ETA Utility, however, lies on digital twins of factories and factory operation rather than on fluid and other physical systems. The FMUs in ETA Utility are used to run simulations in parallel to factory operations. For this reason, the framework focuses strongly on communication interfaces commonly used in manufacturing environments for connecting physical devices, which is beyond the scope of **SOFIRpy**.

The most extensive existing tool with similarities to **SOFIRpy** is Mosaik [9]. Mosaik is a framework for co-simulating many models by providing an interface to a variety of existing simulators for the purpose of performing smart grid simulations. Similarly to **SOFIRpy**, the framework organises information exchange between a variety of simulators for the purpose of co-simulation. In contrast to Mosaik, **SOFIRpy** focuses on FMUs and simulation of entities described in Modelica rather than providing interfaces to many different simulators. While Mosaik has also implemented an adapter to couple FMUs, this functionality is just one of many provided by the package. Both Mosaik and **SOFIRpy** allow for users to define custom models for controllers in Python. Mosaik offers comprehensive features for data handling with adapters to various databases, including not only InfluxDB and TimescaleDB, but also HDF5. **SOFIRpy** is limited to HDF5 for storing the results of simulated data. While Mosaik allows users to store metadata and connections related to the simulation results data, the focus of **SOFIRpy** on RDM and reproducibility remains unique to it.

Several software packages perform partial functions of the steps enumerated above, yet **SOFIRpy** unites the functionalities in one overarching framework. Furthermore, none but one of the software packages mentioned above considers the aspect of RDM and no functionalities are provided for this vital step in scientific practice.

2 Approach

2.1 Design choices

SOFIRpy has been developed as a software framework based on the requirements presented in Section 1.1. In the development phase of **SOFIRpy**, several options for each of the functionalities were considered. These options, the selection made and the reasoning behind it will be briefly presented.

Coming from the original use-case, both a modelling and a simulation environment was needed for the fluid system as well as the multi-agent system for controlling it. For the fluid system, options for modelling and simulation available to the authors were: (a) using a modelling software, e.g. Dymola, OpenModelica or Matlab/Simulink, widely used in the community, (b) writing a custom model from scratch in Matlab or Python. Option (a) allows not only the possibility to work with a graphic user interface, significantly improving the user experience when modelling, but also the export of FMUs, a standardised format allowing interoperability with other software tools including the components of the **SOFIRpy** framework. Moreover, it allows creating FMUs of any physical system rather than only fluid systems. Consequently, a solution was chosen that reduces the need to write custom models for fluid systems.

The choice of programming language for the co-simulation of FMU and controller models was made in favour of Python due to its prevalence as a programming language, especially in fields that are interesting for innovative control methods such as machine learning, and because it is non-proprietary. Though the original use-case was for studying multi-agent control of fluid systems, the design of **SOFIRpy** was generalised to allow studying any type of control method.

Accordingly, frameworks specific to agent-based modelling or control were not selected as a fixed choice. Instead, users need to write the custom models for the controllers themselves. This makes it possible to build implemented controllers on a wide range of other Python toolboxes, e.g. for machine learning.

Regarding data storage, there is a great variety of options. The approach chosen for **SOFIRpy** is to store the simulation setup including used models and parameters together with the results of different simulation runs in one HDF5 file. The simulation results are also available as Pandas `DataFrames` [10]. Storing to HDF5 is performed using a wrapper around H5py, a Python package that offers easy-to-use methods for data storage and retrieval with HDF5 files. One advantage of this approach is that all relevant data and information about an experiment are stored in a single location. HDF5 files are versatile enough to store different data, such as the FMU binaries, the serialisations of the classes of the custom models written in Python as well as the simulation setup and the resulting simulation data. Furthermore, HDF5 files allow adding attributes to datasets and (sub-) groups, which is useful for including metadata for describing the simulation setup and runs. Metadata are not only useful for documentation purposes, but also for searching for specific simulation results using a set of parameters. The benefit of considering the RDM requirements in the fundamental design of **SOFIRpy** is that by providing easy-to-use, lightweight RDM features within the workflow software, users are motivated to apply them, ensuring improved scientific data as well as transparent and reproducible results.

Finally, it was decided to package and publish **SOFIRpy** as a Python package. This facilities incorporating it in other software setups. Registering **SOFIRpy** on the package index PyPI also addresses the issues of findability, accessibility and easy installation.

2.2 Scope

SOFIRpy is aimed at researchers, planners and operators of fluid and other physical systems who want to analyse novel control approaches. The purpose of **SOFIRpy** is to provide a framework for co-simulating FMUs and custom-written models with integrated research data management.

Of the workflow steps enumerated in Section 1.1, step 1 (modelling physical systems) was not included in the scope of **SOFIRpy**, restricting the package to interactions with FMU representations of the systems. Step 2 (creating custom models for controllers) is the core of the work of the researchers, planners and operators, for which **SOFRIpy** offers a framework with as few restrictions as possible. For step 4 (results analysis), plenty of tools already exist. Therefore, this step lies in the responsibility of the user and is not entailed in the scope of the presented tool. **SOFIRpy** provides three main functionalities:

(i) Export Modelica models and necessary solvers as an FMU

(ii) Co-simulate FMUs with custom written controllers in Python (workflow step 3)

(iii) Store data and metadata of the simulation inside a HDF5 file (workflow step 5)

The first of the functionalities allows users to convert models written in the Modelica modelling language to FMUs. Exporting models to FMUs is also possible in the OpenModelica and Dymola GUI as well as many other software tools used in engineering practice. With **SOFIRpy**, however, this step can be integrated into the code-based workflow. This also allows users to set parameters for the models and exporting them to FMUs from within their scripts. Separate methods are provided for OpenModelica and Dymola. This ensures **SOFIRpy** remains independent of commercial software while also providing full functionality for users of Dymola.

For the second functionality, each entity, be it an FMU or a custom written model, is represented as an instance of the same abstract class `SimulationEntity`. The data class `System` contains the data and the name of each simulation entity. Further data classes are defined for containing system parameters, logged parameters, connection points of each `System` and the actual connections between them. These connections are key for co-simulation, as they are used for defining the reciprocal influence of the controllers and FMUs. Finally, the `Simulation` class allows for a simulation run to be performed of all systems considering their interdependencies. This class also provides a method for storing the simulation results in a Pandas `DataFrame`.

The third functionality concerns the RDM features. **SOFIRpy** wraps the package H5py to store the simulation results in HDF5 files. It also provides the option to store units of the time series in the attributes of the corresponding dataset in the HDF5 file. Beyond this basic data storage function, **SOFIRpy** provides further features for RDM. The HDF5 files used for storing results are structured on the highest level into two or more groups: one for the simulated entities, referred to as models, and one for each simulation run of these models stored in the file. In the models group, the binary files of the FMUs and the classes of custom controllers can be stored along with the source code of the classes. The group for one run contains the resulting time series of a simulation run, as well as datasets with the data for the configuration of the run and software dependencies for the simulation run. This is illustrated in Figure 1.

Furthermore, each run group contains subgroups for each simulated model, both FMU and custom controller. These subgroups contain datasets with information on the connections of this model to other models, which parameters of the model are logged and the start values of the model for that particular run. Another dataset in the subgroup references the binary of the model. In the case of the Python models, there are two datasets: one referencing the binary and the other the source code of the class. This structure is illustrated in Figure 2.

This structure and the stored information allows users to recreate the simulated models along with the run parameters and start values from the data stored in the HDF5 file. In this way, each simulation run can be fully reproduced from the stored data.

2.3 Limitations

The main functionality of **SOFIRpy** is the co-simulation of FMUs and custom written models. Accordingly, limitations mainly concern the FMU export and the RDM features. These will be discussed in the following.

The first limitation is that **SOFIRpy** does not implement its own solver for FMUs. This means that it depends on either open source solvers, such as CVode, or proprietary solvers, such as Dassl from Dymola, being available and exported with the FMU for the simulation to work.

There are several limitations regarding the RDM features. Firstly, the metadata stored with the HDF5 files are not standardised, i.e. they do not follow a specific schema. Selecting and implementing a specific metadata schema is left to the user, since **SOFIRpy** may be used in

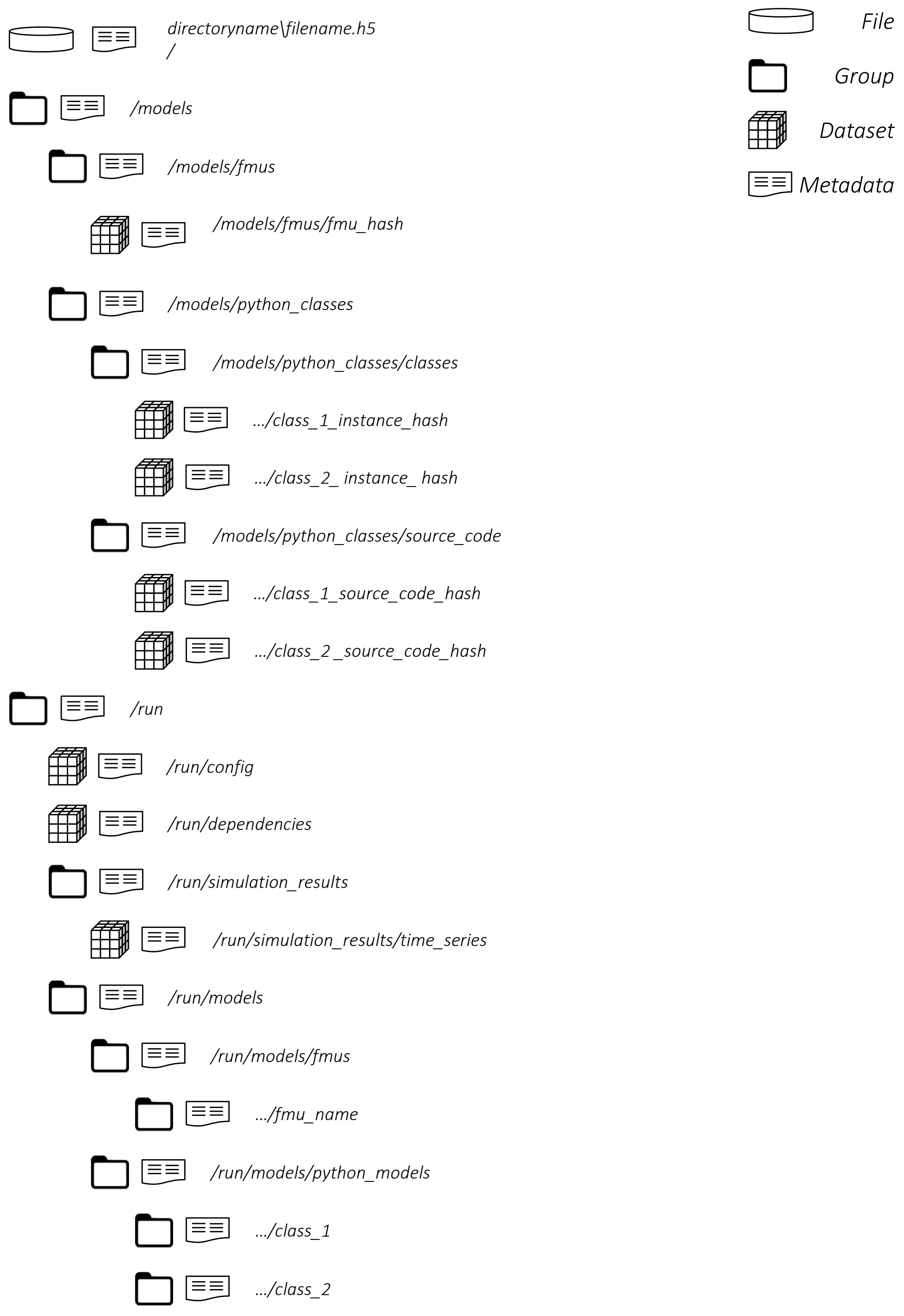
directoryname\filename.h5
/
/models
/models/fmus
/models/fmus/fmu_hash
/models/python_classes
/models/python_classes/classes
.../class_1_instance_hash
.../class_2_ instance_ hash
/models/python_classes/source_code
.../class_1_source_code_hash
.../class_2 _source_code_hash
/run
/run/config
/run/dependencies
/run/simulation_results
/run/simulation_results/time_series
/run/models
/run/models/fmus
.../fmu_name
/run/models/python_models
.../class_1
.../class_2
File
Group
Dataset
Metadata

Figure 1: Structure of the HDF5 file.

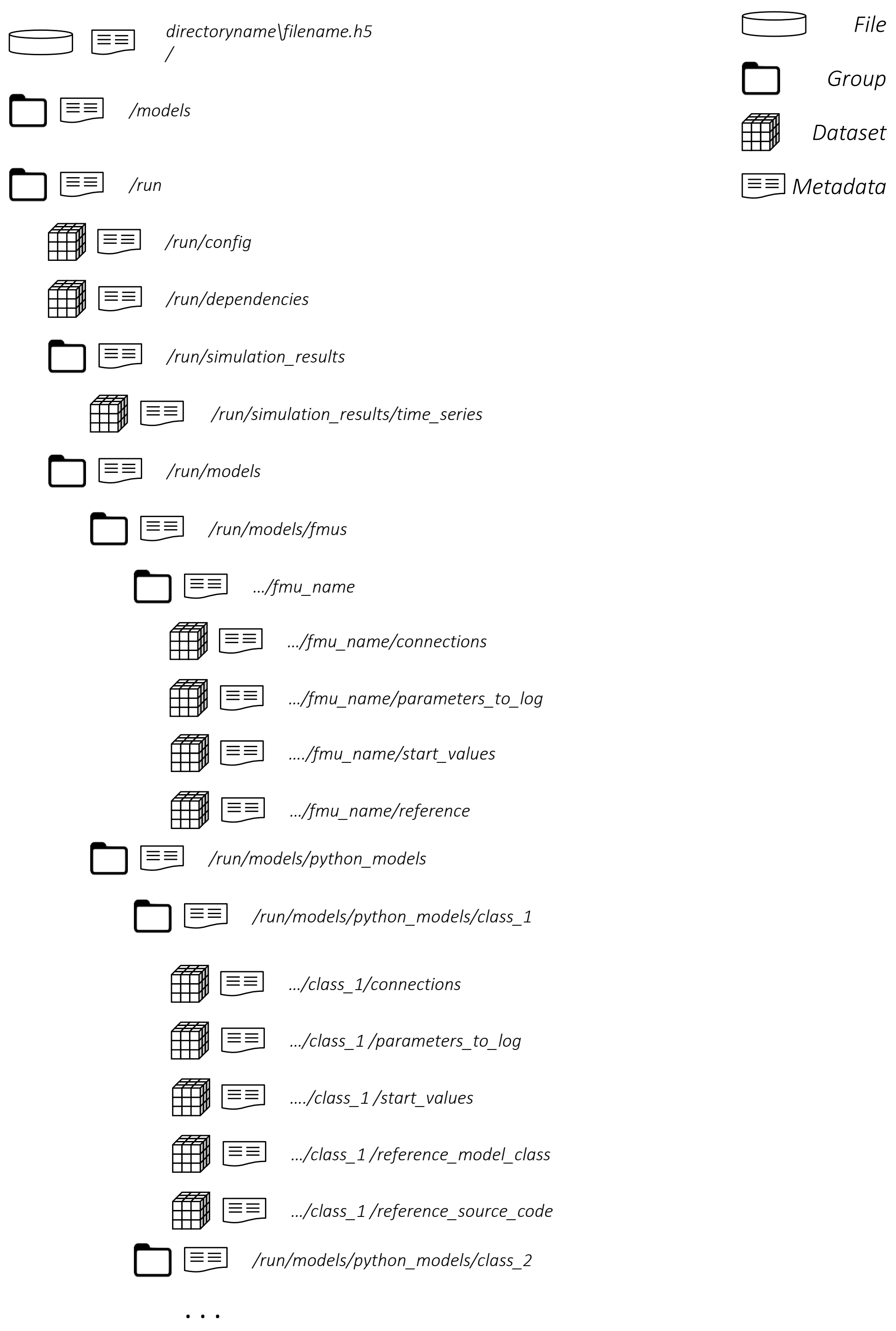
directoryname\filename.h5
/
/models
/run
/run/config
/run/dependencies
/run/simulation_results
/run/simulation_results/time_series
/run/models
/run/models/fmus
.../fmu_name
.../fmu_name/connections
.../fmu_name/parameters_to_log
..../fmu_name/start_values
.../fmu_name/reference
/run/models/python_models
/run/models/python_models/class_1
.../class_1/connections
.../class_1 /parameters_to_log
..../class_1 /start_values
.../class_1 /reference_model_class
.../class_1 /reference_source_code
/run/models/python_models/class_2
. . .
File
Group
Dataset
Metadata

Figure 2: Structure of the HDF5 file and contents of the sub-groups of the `/run` group. Each class sub-group contains the same five datasets.

different contexts, each of which may require a separate schema. Secondly, the metadata are not described using a standardised vocabulary. Vocabularies are context specific and accordingly, **SOFIRpy** leaves it to the users to decide which terms best describe the data they record using the software. Thirdly, the source code of the custom Python models is serialised for storage in the resulting datasets using `cloudpickle` [11]. Deserialising objects with `cloudpickle` requires the use of the same operating system used for serialising them. Deserialisation is only guaranteed for the exact same version of Python, though it has been shown to work when using different patch versions of Python. The latter issue can be addressed by using a tool for environment management, e.g. Poetry [12], VirtualEnvironment [13] or Anaconda [14] and providing the specifications of that environment. Furthermore, `cloudpickle` is not suitable for long-term object storage [11]. Since the source code of the classes is also stored, the code can be executed to recreate the simulation, though binaries of the classes may not be deserialisable in the long term.

3 Code Structure

The following section will present the code structure of **SOFIRpy**: the design concepts, architecture of the package and workflow. **SOFIRpy** is structured into 3 separate sub packages, coinciding with the functionalities mentioned in Section 2.2: `fmu_export`, `simulation` and `rdm`.

3.1 fmu_export

Modelica models created with both OpenModelica and Dymola can be exported to FMUs. For each export, a separate class is implemented that inherits from a base class, `FmuExport`. Both classes define an `export_fmu` method that encapsulates the logic required for exporting a Modelica model to an FMU.

The OpenModelica export uses the OMPython package, which provides a method to do the export. In contrast, the Dymola export is more complex. In **SOFIRpy**, Dymola's native scripting language is used. A script is dynamically generated to set the required parameters and export the Modelica model. This script is then executed using `subprocess`, a standard library in Python.

3.2 simulation

The UML diagram in Figure 3 provides an overview of the design for the simulation subpackage. At its core is the `Simulator` class. The class holds the instance attributes `systems`, `connections` and `parameters_to_log`.

The `systems` attribute is a dictionary that maps the name of the system to an instance of the `System` class. The `System` class is designed to represent an entire physical system. An instance of the `System` class will hold an instance of the `FMU` class or the `CustomPythonModel` class, both of which inherit from the abstract base class `SimulationEntity` to ensure a standardised interface. The `SimulationEntity` class defines three abstract methods `set_parameter`, `get_parameter_value`, and `do_step`, which govern parameter manipulation, value retrieval, and simulation progression, respectively. The optional methods `initialize`, `get_unit`, and

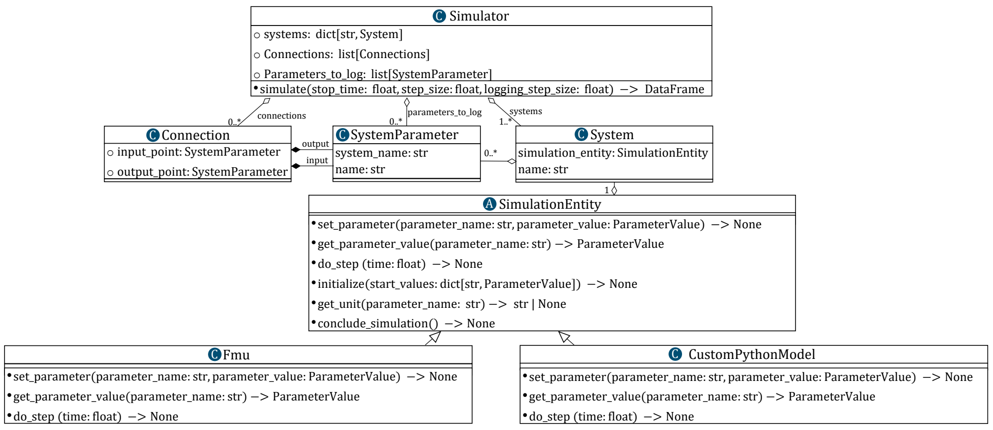
Simulator
systems: dict[str, System]
Connections: list[Connections]
Parameters_to_log: list[SystemParameter]
simulate(stop_time: float, step_size: float, logging_step_size: float) -> DataFrame
connections
0..*
parameters_to_log
0..*
systems
1..*
Connection
input_point: SystemParameter
output_point: SystemParameter
output
input
SystemParameter
system_name: str
name: str
0..*
System
simulation_entity: SimulationEntity
name: str
1
SimulationEntity
set_parameter(parameter_name: str, parameter_value: ParameterValue) -> None
get_parameter_value(parameter_name: str) -> ParameterValue
do_step (time: float) -> None
initialize(start_values: dict[str, ParameterValue]) -> None
get_unit(parameter_name: str) -> str | None
conclude_simulation() -> None
Fmu
set_parameter(parameter_name: str, parameter_value: ParameterValue) -> None
get_parameter_value(parameter_name: str) -> ParameterValue
do_step (time: float) -> None
CustomPythonModel
set_parameter(parameter_name: str, parameter_value: ParameterValue) -> None
get_parameter_value(parameter_name: str) -> ParameterValue
do_step (time: float) -> None

Figure 3: Simulation UML Diagramm.

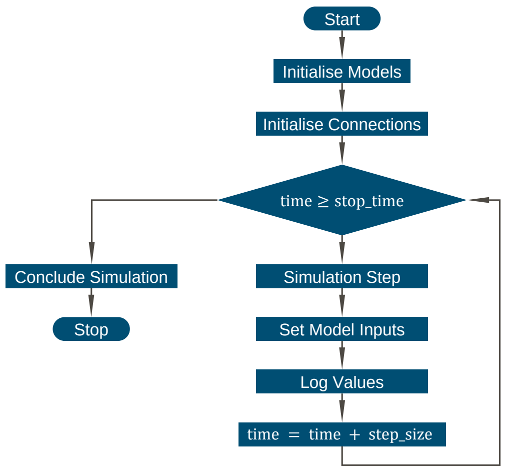

Figure 4: Simulation Flowchart.

`conclude_simulation` can be implemented. These methods define how to set start values, retrieve units, and perform necessary steps after the simulation has finished.

The `connections` attribute contains information on how all simulation models are connected to each other and `parameters_to_log` defines which parameters/values should be logged.

The steps performed during a simulation are shown in Figure 4. Before starting the simulation, the models to be simulated are initialised by calling each model's `initialize` method and all connections between the models are initialised. Then the simulation loop starts. In each loop, one simulation step is performed in each model. Subsequently, the inputs in each model are set to the appropriate value. Next, values that should be logged are logged and finally, the simulation time is increased by the step size. If the simulation time is greater than the stop time, the simulation is concluded and stops.

3.3 rdm

At the core of the `rdm` subpackage is the `Run` class. This class contains attributes for all data relevant to a simulation run and provides methods to manipulate the settings of the run. Additionally, the `Run` class provides the methods `simulate`, `to_hdf5`, and `from_hdf5`. The `simulate` method allows simulating the defined systems based on the current configuration. Upon completion of the simulation, the results and all associated metadata can be stored in a HDF5 file via the `to_hdf5` method. This method stores the data in the format outlined in Section 2.2. Each data entry in the `Run` class is serialised using custom serialisation methods, ensuring that the data is properly converted into a format suitable for storage in the HDF5 file. The `from_hdf5` method, in turn, reconstructs the `Run` class from a specified HDF5 file and run name, deserialising the stored data and enabling the simulation to be rerun using the previously saved configuration.

4 User Notes

The following section will show how to engage with **SOFIRpy**, be it as a user with installation instructions and minimal examples or as a developer with instructions on how to contribute.

4.1 Getting Started

SOFIRpy is hosted on the package index PyPI and can be installed using pip with the following command:

```
    pip install sofirpy
```

The **SOFIRpy** documentation includes an introduction with installation instructions, minimal examples for all functionalities and full API documentation. The documentation is publicly available [15].

4.2 Minimal Example

The code examples below illustrate how to co-simulate an FMU and a custom controller, and how to utilise the RDM features in **SOFIRpy**.

In the first example, a PID controller is implemented as a custom controller for controlling a direct current (DC) motor of which there is already an FMU. After the PID controller is implemented, the connections between the controller and the DC motor are defined, as well as the start values and the parameters to be logged during the simulation. Finally, the simulation run of the PID controller controlling the DC motor is performed using the defined setup.

```python
from sofirpy import simulate, SimulationEntity

# define fmus to be simulated
fmu_paths = {"DC_Motor": "path/to/fmu"}

# define custom models to be simulated
# here a custom PID controller is implemeted
class PID(SimulationEntity):
    """Simple implementation of a discrete pid controller"""
    def do_step(self,  time): ... # To be implemented
    def get_parameter_value(self, output_name): ... # To be implemented
    def set_parameter(self, parameter_name, parameter_value): ... # To be implemented

model_classes = {"pid": PID}

# define how the models are connected
connections_config = {
    "DC_Motor": [
        {
            "parameter_name": "u",
            "connect_to_system": "pid",
            "connect_to_external_parameter": "u",
        }
    ],
    "pid": [
        {
            "parameter_name": "speed",
            "connect_to_system": "DC_Motor",
            "connect_to_external_parameter": "y",
        }
    ],
}

# define start values for models
start_values = {
    "DC_Motor": {"inertia.J": 2},
    "pid": {"K_p": 3,"K_i": 20,"K_d": 0.1},
}

# define which variables to log
parameters_to_log = {
    "DC_Motor": ["MotorTorque.tau"],
    "pid": ["u"],
}
results, units = simulate(
    stop_time=10,
    step_size=1e-3,
    fmu_paths=fmu_paths,
    model_classes=model_classes,
    connections_config=connections_config,
    start_values=start_values,
    parameters_to_log=parameters_to_log,
    logging_step_size=1e-3,
    get_units=True,
)
```

The following minimal example of the RDM features shows how a simulation can be configured, run and stored in an HDF5 file. In a second step, the simulation run is loaded from the HDF5 file and a parameter is changed. After this, a second simulation is run with the changed parameter.

```python
from sofirpy import Run

run_name = "Run_1"
model_classes = {"pid": PID}
fmu_paths = {"DC_Motor": "path/to/fmu"}
run = Run.from_config(
    run_name=run_name,
    stop_time=10,
    step_size=0.1,
    fmu_paths=fmu_paths,
    model_classes=model_classes,
)
run.simulate() # calls the simulate function with the defined config
# accessing the results
results = run.time_series

# storing the run inside the hdf5
hdf5_path = "path/to/hdf5"
run.to_hdf5(hdf5_path)

# loading the run from the hdf5
run_loaded = Run.from_hdf5(run_name, hdf5_path)

# manipulating the config of a run, e.g. changing the stop time
run_loaded.stop_time = 100
# simulating the run with changed config
run_loaded.simulate()
```

4.3 Ensuring Code Quality

Aside from being a tool for co-simulation, **SOFIRpy** is a demonstration example of how to write and document research software in the field of mechanical engineering for good scientific practice in RDM. Accordingly, great emphasis was placed on adhering to best practices in software development and ensuring comprehensive documentation. The used methods will be briefly presented.

4.3.1 Version Control and CI/CD

Throughout the development of the package **SOFIRpy**, Git was used as a version control system. The package is hosted on a remote repository on GitHub. An automated CI/CD pipeline using GitHub Actions is implemented that contains static code analysis, testing, building the documentation, as well as publishing and releasing the package.

Testing **SOFIRpy** is equipped with parametrised unit tests implemented with Pytest. The test routine is a part of its CI/CD pipeline. **SOFIRpy** currently has a test coverage of 80 %.

Static Code Analysis Linting is a method for ensuring compliance of source code with style conventions and for detecting programmatic errors. **SOFIRpy** includes the linting tool Ruff among its optional dependencies for development, as well as in the automated testing pipeline. The tool is also used for automatic formatting of code, which facilitates reading the code for users and other developers.

For improved documentation and for facilitating development, **SOFIRpy** uses the tool Mypy in strict mode for static type checking. **SOFIRpy** is fully typed.

Pre-commit To ensure compliance with style guidelines in writing code, the **SOFIRpy** repository comes with a configuration file for pre-commit hooks. The package Pre-commit is a tool for configuring pre-commit hooks for version control with git. Before each commit, the source code gets checked for violations of the configured commit hooks.

4.3.2 Documentation

Comprehensive use is made of docstrings throughout the source code of the package for documenting the functionality of classes and methods. Docstrings are consistently written in the Google format.

The docstrings are used for automatically building the API documentation. The documentation of the package **SOFIRpy** is publicly available [15]. Apart from the API documentation, it also includes installation instructions and the minimal examples presented above, as well as further demonstrations of the features of **SOFIRpy**. The documentation is built using the package Sphinx.

4.3.3 Packaging and Publishing

The project **SOFIRpy** has been packaged and released. It is published on the public repository PyPI [16]. Not only does this make the package easily installable through Pip, it also ensures compliance with the same standards in metadata, declaration of dependencies, version history, etc., that the most common Python packages follow. The release of new versions of **SOFIRpy** is also automated through GitHub actions.

To summarise, **SOFIRpy** is a fully developed software package that follows common practices in software development. It is a package of research software with a typical application in the field of mechanical and control engineering. These two aspects together show how **SOFIRpy** can serve as an orientation example for engineering scientists without a background in software development who frequently need to write research software.

4.4 Maintenance and How to Contribute

SOFIRpy is maintained in the publicly accessible GitHub repository. It is possible to submit issues and contribute to the project. Detailed guidelines on how to contribute are provided in the project's documentation [15].

5 Current Status and Outlook

SOFIRpy is currently a stable Python package that adheres to common domain and language best practices. It offers its three core functionalities: (i) exporting Modelica models as an FMU, (ii) co-simulating FMUs with custom written controllers in Python, (iii) storing data and metadata of the simulation in a HDF5 file and reproducing simulations from the stored data.

As discussed in Section 2.3, the RDM functionalities are not as feature-rich as may be expected considering the current state of development in the RDM community. However, this is a design choice made in the development of **SOFIRpy**, as this allows greater flexibility for users. Further developments in this area may be to ensure a seamless integration with other tools that focus on RDM methods for HDF5 files and use of standardised metadata schemas and controlled vocabularies, e.g. H5RDMtoolbox [17].

An aspect for future development is to further generalise the application of **SOFIRpy**. Currently, the package supports co-simulation of FMUs as models of physical systems, e.g. fluid systems, and custom written models for controlling them. The package may be extended to enable users to substitute the FMUs with actual physical systems. This will broaden the utility of the software, allowing its use as a framework for conducting physical experiments as well as simulations. A further possible extension is to expose more functions to the user with the purpose of facilitating the use of **SOFIRpy**, e.g. when using reinforcement learning methods for controllers.

6 Acknowledgements

The Authors would like to thank the Federal Government and the Heads of Government of the Länder, as well as the Joint Science Conference (GWK), for their funding and support within the framework of the NFDI4Ing consortium. Funded by the German Research Foundation (DFG) - project number 442146713.

This work has been co-funded by the LOEWE initiative (Hesse, Germany) within the emergenCITY center.

The authors would further like to thank Jonas Bodner, Julius Breuer, Lara Doß, Hendrick Gockel, Tim Buchert, Tim Schimkat and Johanna Spegg for their contributions.

7 Roles and contributions

Daniele Inturri: Conceptualization, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing

Kevin T. Logan: Conceptualization, Writing – original draft, Writing – review & editing, Project Administration, Supervision

Michaela Leštáková: Conceptualization, Writing – review & editing, Project Administration, Supervision

Tobias C. Meck: Writing – review & editing, Validation, Supervision

Peter F. Pelz: Supervision, Resources, Funding Acquisition

References

[1] European Union, *Ecodesign Pump Review - Extended report: Study of Commission Regulation (EU) No 547/2012 (Ecodesign requirements for water pumps)*, 2018.

[2] eurostat. “Electricity production, consumption and market overview: Net electricity generation, EU-27, 1990-2018.” (2021), [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Electricity_production,_consumption_and_market_overview (visited on 10/11/2024).

[3] Directorate-General for Energy. “Energy Efficiency Directive.” (2024), [Online]. Available: https://energy.ec.europa.eu/topics/energy-efficiency/energy-efficiency-targets-directive-and-rules/energy-efficiency-directive_en (visited on 04/29/2024).

[4] B. Went, “The systems approach to reducing carbon emissions,” *World Pumps*, vol. 2008, no. 500, pp. 52–55, 2008.

[5] “OMPython: A Python interface to OpenModelica communicating via CORBA or ZeroMQ.” (2024), [Online]. Available: https://github.com/OpenModelica/OMPython (visited on 05/30/2024).

[6] T. Sommer. “FMPy: Simulate Functional Mockup Units (FMUs) in Python.” (2024), [Online]. Available: https://github.com/CATIA-Systems/FMPy (visited on 05/30/2024).

[7] J. Pohlodek, B. Morabito, C. Schlauch, P. Zometa, and R. Findeisen, *Flexible development and evaluation of machine-learning-supported optimal control and estimation methods via HILO-MPC*, 2022. DOI: 10.48550/ARXIV.2203.13671.

[8] B. Grosch, H. Ranzau, B. Dietrich, *et al.*, “A framework for researching energy optimization of factory operations,” *Energy Informatics*, vol. 5, no. 1, 2022. DOI: https://doi.org/10.1186/s42162-022-00207-6.

[9] C. Steinbrink, M. Blank-Babazadeh, A. El-Ama, *et al.*, “Cpes testing with mosaik: Co-simulation planning, execution and analysis,” *Applied Sciences*, vol. 9, no. 5, 2019, ISSN: 2076-3417. DOI: 10.3390/app9050923. [Online]. Available: https://www.mdpi.com/2076-3417/9/5/923.

[10] “pandas - Python Data Analysis Library.” (2024), [Online]. Available: https://pandas.pydata.org/ (visited on 08/23/2024).

[11] “cloudpipe/cloudpickle: Extended pickling support for Python objects.” (2024), [Online]. Available: https://github.com/cloudpipe/cloudpickle (visited on 08/23/2024).

[12] “Poetry - Python dependency management and packaging made easy.” (2024), [Online]. Available: https://python-poetry.org/docs/ (visited on 08/23/2024).

[13] Python documentation. “venv — Creation of virtual environments.” (2024), [Online]. Available: https://docs.python.org/3/library/venv.html (visited on 08/23/2024).

[14] “Managing environments — conda 24.7.2.dev47 documentation.” (2024), [Online]. Available: https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html (visited on 08/23/2024).

[15] D. Inturri. “SOFIRpy documentation.” (2024), [Online]. Available: https://fluid-systems.github.io/SOFIRpy/index.html (visited on 09/26/2024).

[16] PyPI. “Sofirpy.” (2024), [Online]. Available: https://pypi.org/project/sofirpy (visited on 08/23/2024).

[17] M. Probst and B. Pritz, “h5RDMtoolbox - A Python Toolbox for FAIR Data Management around HDF5,” *ing.grid Preprints*, [Online]. Available: https://preprints.inggrid.org/repository/view/23/ (visited on 08/23/2024).