The FMI++ Python Interface

A Python package for importing and exporting FMUs

Edmund Widl

AIT Austrian Institue of Technology, Center for Energy, Vienna, Austria

Content of tutorial

- Requirements for running demos and exercises
- Introduction to FMI, FMI++ & FMI++ Python Interface
- Installation of the FMI++ Python Interface on Windows and Linux
- Basic FMU import functionality (ME and CS)
- Advanced FMU import functionality for ME (event prediction, rollbacks, etc.)
- Exporting Python scripts as FMU for CS
- Debugging of Python scripts prior to export
- Hands-on exercises

Running the demos / exercises in this tutorial …

- The full tutorial (presentation, demos, exercises) is available online

  → https://github.com/AIT-IES/py-fmipp-tutorial

- Demos are provided as Jupyter notebooks
  - subfolder *demos* → see below
  - also online → Code Ocean compute capsule: https://doi.org/10.24433/CO.9880202.v2

- All supporting material for demos and exercises in this tutorial are available in the following subfolders:
  - subfolder *demos* → Jupyter notebooks
    - subfolder *demos/scripts* → notebooks as standard Python scripts (in case you don't want to install jupyter)
    - subfolder *demos/modelica* → Modelica models used in the demos
    - subfolder *demos/data* → FMU for model zigzag (Linux 64-bit, Windows 32-bit, Windows 64-bit)
  - subfolder *exercises*
    - subfolder *exercises/import* → import Modelica plant model in Python
    - subfolder *exercise/export* → export Python controller and use it from Modelica

Requirements for running the demos / exercises

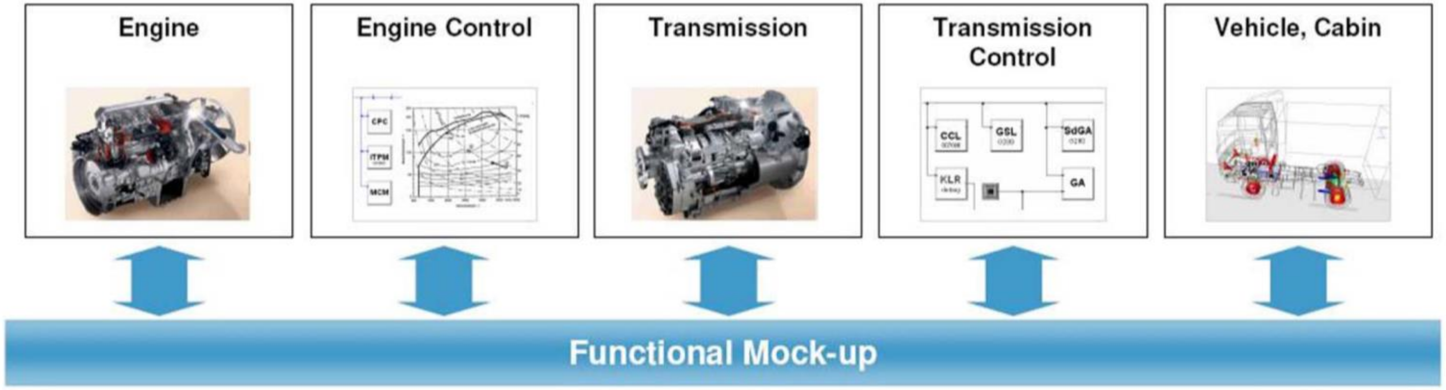
- General requirements:
  - up-to-date version of *Python* installed (version 2.7 or 3.6 and higher)
  - know how to install Python packages via *pip*
- Required Python packages for running demos:
  - *fmipp* → see following slides
  - *jupyter* → `pip install jupyter`
  - *matplotlib* → `pip install matplotlib`
- Alternative to Jupyter notebooks → run standard Python scripts (subfolder *demos/scripts*)
- Requirements for running the exercises:
  - *Modelica compiler* that allows to *export FMUs for Model Exchange* (FMI 1.0 or 2.0)
  - *Modelica compiler* that allows to *import FMUs for Co-Simulation* (FMI 2.0)
  - Modelica compiler and Python version have to be either *both* 32-bit or 64-bit
  - tested with Dymola 2018, but should also work with JModelica, OpenModelica, etc.

Content of tutorial

- Requirements for running demos and exercises
- Introduction to FMI, FMI++ & FMI++ Python Interface
- Installation of the FMI++ Python Interface on Windows and Linux
- Basic FMU import functionality (ME and CS)
- Advanced FMU import functionality for ME (event prediction, rollbacks, etc.)
- Exporting Python scripts as FMU for CS
- Debugging of Python scripts prior to export
- Hands-on exercises

FMI – Functional Mock-up Interface

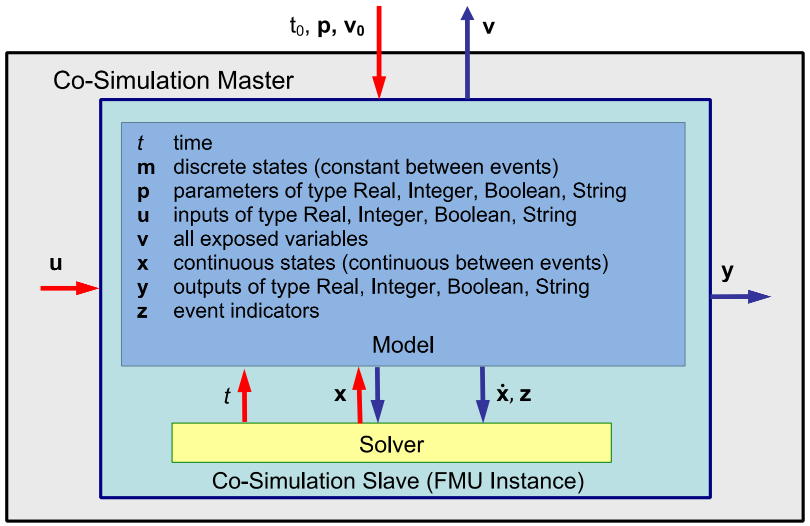
- FMI has been developed to *encapsulate* and *link* models and simulators
  - developed within MODELISAR project
  - driven by a community from *industry and academia*
  - *standardized encapsulation* of *models* and *tools*
  - first version published in 2010, second version published in 2014
  - initially supported by 35 tools, currently *supported by more than 100 tools*
  - see: https://www.fmi-standard.org/



Cosimulation of the behavioral models and the embedded controller software

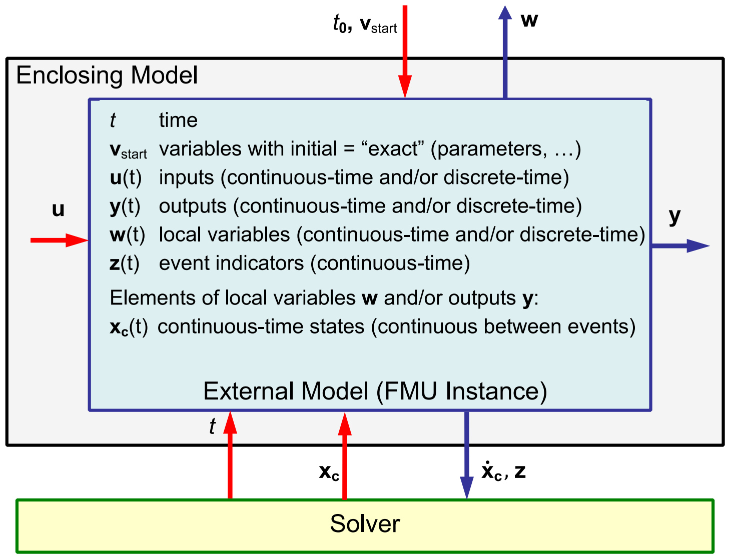
FMI – Functional Mock-up Interface

Co-Simulation (CS)



- stand-alone black-box simulation components
- data exchange restricted to discrete communication points
- between two communication points system model is solved by internal solver
- may call another tool at run-time (tool coupling)

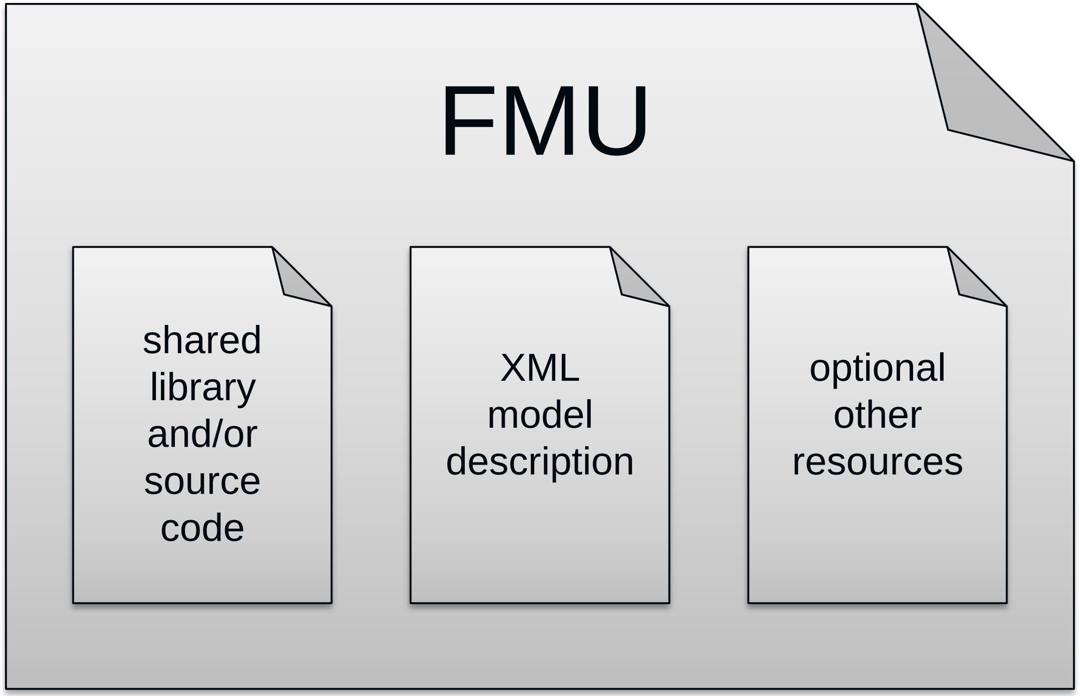
Model Exchange (ME)



- standardized access to model equations
- models described by differential, algebraic and discrete equations
- time-events, state-events and step-events
- solved with integrators provided by embedding environment.

Functional Mock-up Unit (FMU)

- FMU ≡ *simulation component* compliant with FMI specification
- *ZIP file* that contains:
  - *shared library* and/or source code
  - XML-based *model description*
  - optional other resources (icon, etc.)
- shared library (or source code) implements *FMI API*
- all static information related to an FMU is stored in an XML text file according to the *FMI Description Schema*

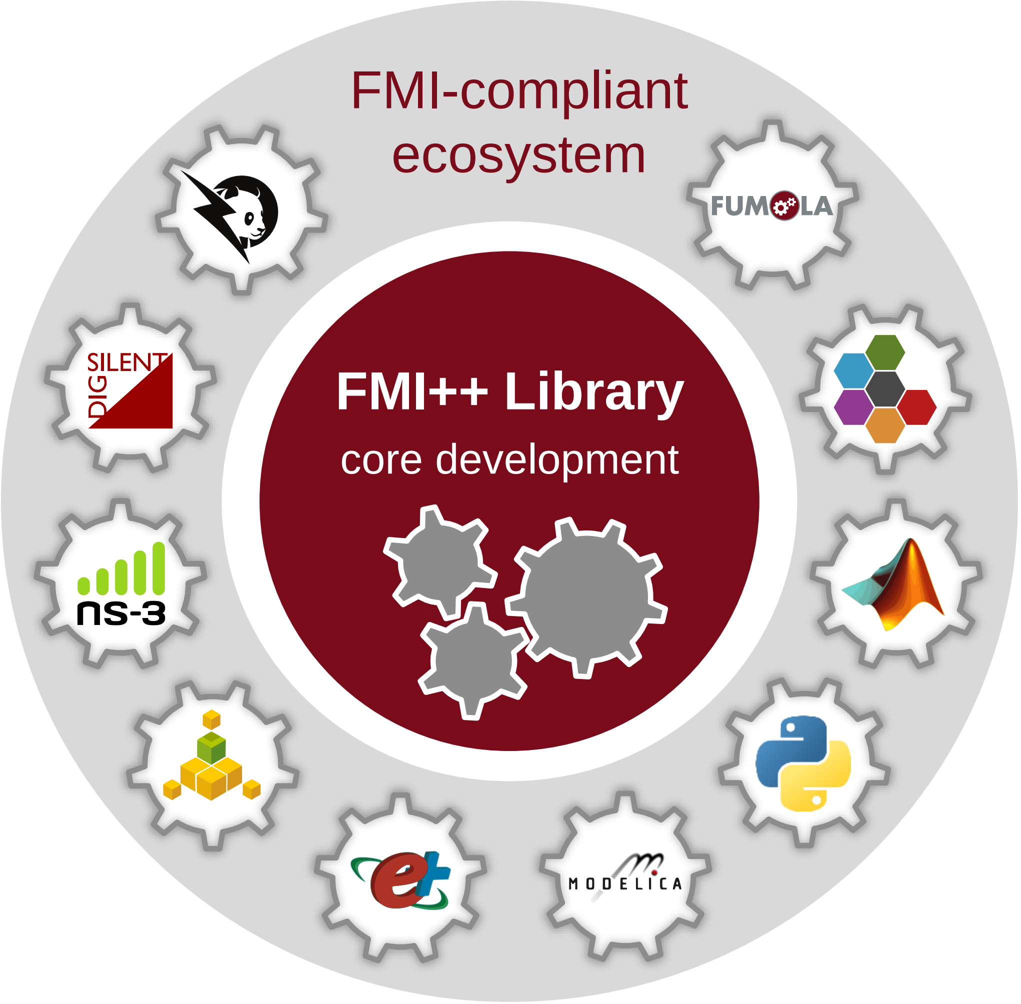


Background: The FMI++ Library

- *software library* based on the *Functional Mock-up Interface* (FMI) specification

- *open-source development* allows application in the context of academia and *industry*

- core development for other tools
  - *FMU import*: enable the use of FMUs in other applications
  - *FMU export*: provide support for developing FMI-compliant co-simulation interfaces
  - *cross-platform* and *cross-language*
  - based on others *state-of-the-art tools* (Boost, SWIG, CVODE, etc.)

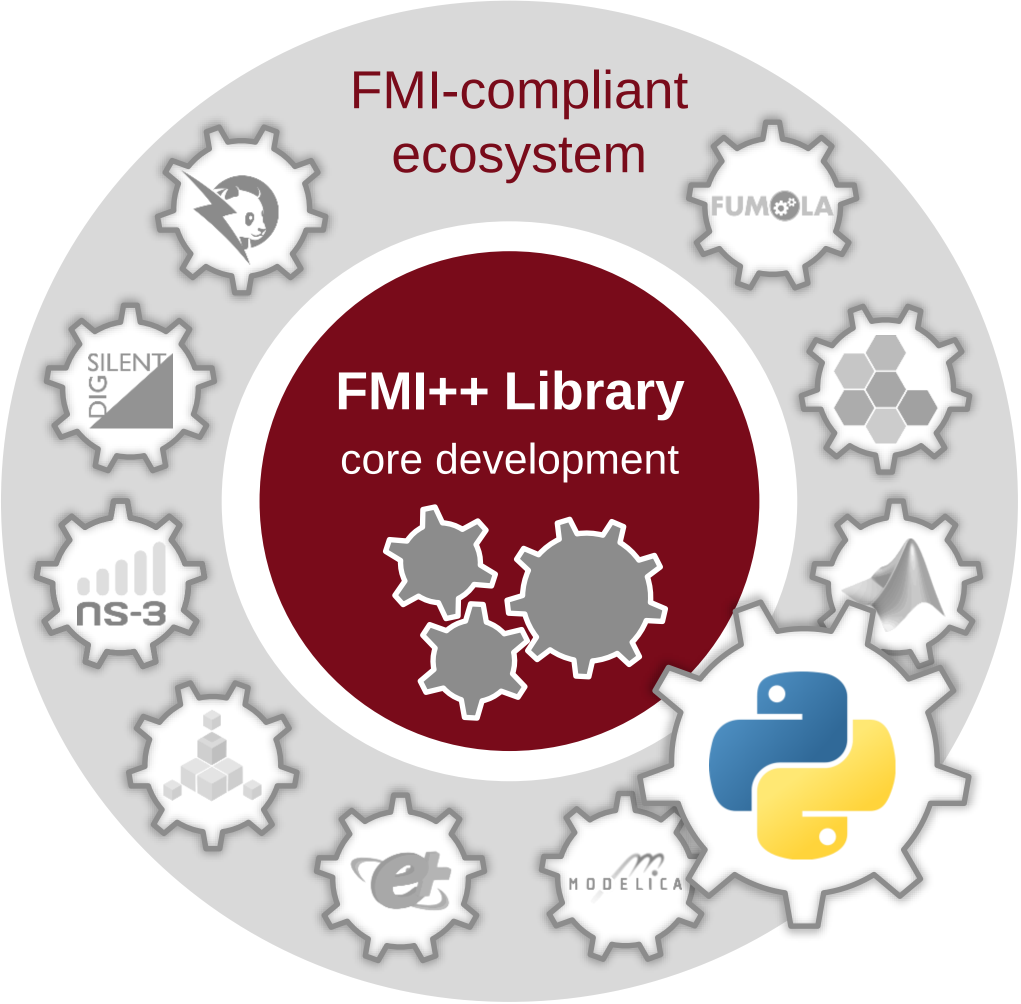


The FMI++ Python Interface

- *Python wrapper* for the FMI++ Library
  - open source
  - freely available via the Python package index

- high-level functionality for *handling* and *manipulation* of FMUs in Python
  - import helper
  - object-oriented representation
  - numerical integration
  - advanced event-handling

- *export Python code* as FMUs for Co-Simulation
  - debugging of Python scripts prior to FMU export



Content of tutorial

- Requirements for running demos and exercises
- Introduction to FMI, FMI++ & FMI++ Python Interface
- Installation of the FMI++ Python Interface on Windows and Linux
- Basic FMU import functionality (ME and CS)
- Advanced FMU import functionality for ME (event prediction, rollbacks, etc.)
- Exporting Python scripts as FMU for CS
- Debugging of Python scripts prior to export
- Hands-on exercises

Installation on Windows

- Use *pip* to install package `fmipp` from the Python package index as pre-compiled binary package (Python wheel):

```
pip install fmipp --prefer-binary
```

- `--prefer-binary` should guarantee that binary distributions (wheels) are chosen over source distributions for the installation

- alternatively, `--only-binary :all:` can be used instead to force installing from binary distribution (old versions of *pip*)

Installation on Linux

- make sure to have installed the following prerequisites (e.g., via *apt-get*):
  - python (*python-dev*) → recommended: version 3.5 (or higher)
  - pip (*python-pip*)
  - distutils (*python-setuptools*)
  - GCC compiler toolchain (*build-essential*)
  - SWIG (*swig*)
  - SUNDIALS library (*libsundials-dev* or *libsundials-serial-dev*)
  - Boost library (*libboost-all-dev*)

- use *pip* to install FMI++ from the Python package index via source distribution:

```
pip install fmipp
```

Checking the installation was successful

- Python command line:

```
>>> import fmipp
>>> fmipp.licenseInfo()
```

- expected output:

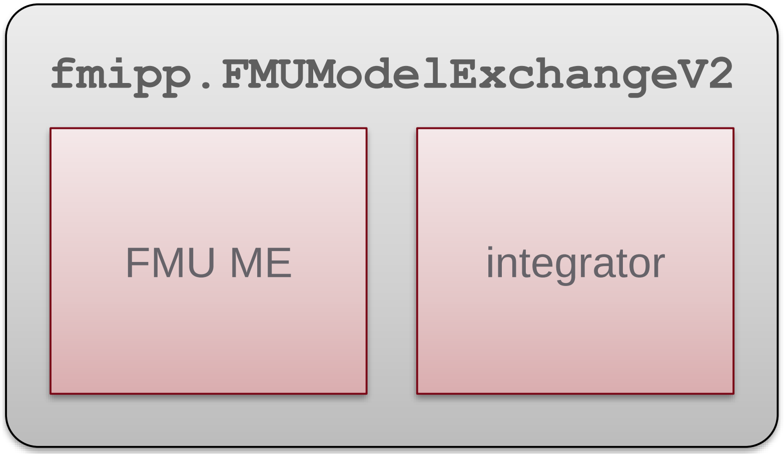
```
The FMI++ Python Interface for Windows is based on code from the
FMI++ Library and BOOST. Also, it includes compiled libraries
implementing the SUNDIALS CVODE integrator.

For detailed information on the respective licenses please refer
to the license files provided here:

C:\path\to\site-packages\fmipp\licenses
```

Content of tutorial

- Requirements for running demos and exercises
- Introduction to FMI, FMI++ & FMI++ Python Interface
- Installation of the FMI++ Python Interface on Windows and Linux
- Basic FMU import functionality (ME and CS)
- Advanced FMU import functionality for ME (event prediction, rollbacks, etc.)
- Exporting Python scripts as FMU for CS
- Debugging of Python scripts prior to export
- Hands-on exercises

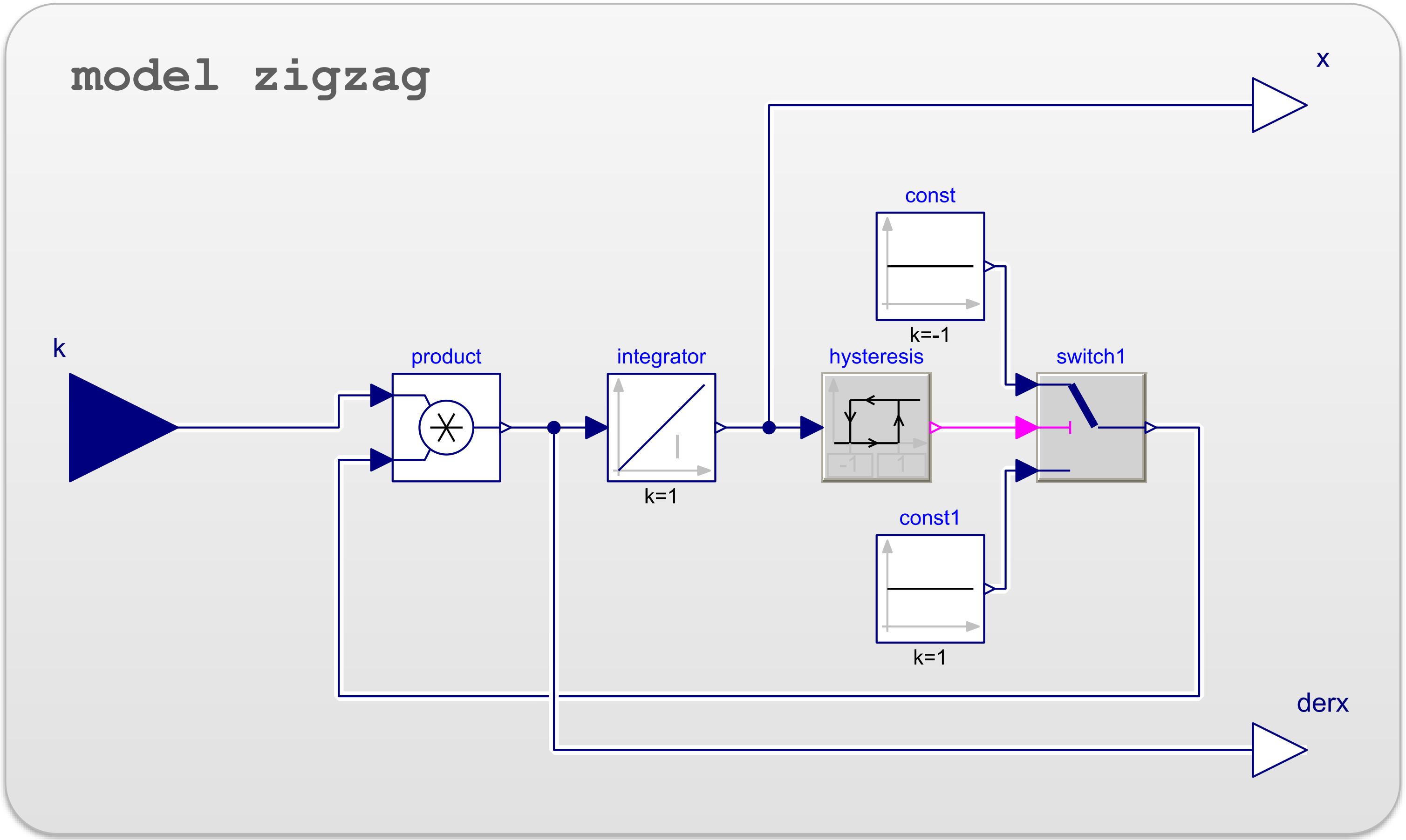
Basic FMU import functionality

- Python package `fmipp` provides functionality that allows to *manipulate FMUs for ME and CS*
- FMUs are represented by *instances* of dedicated *classes*:
  - `fmipp.FMUCoSimulationV1` → FMI CS 1.0
  - `fmipp.FMUCoSimulationV2` → FMI CS 2.0
  - `fmipp.FMUModelExchangeV1` → FMI ME 1.0
  - `fmipp.FMUModelExchangeV2` → FMI ME 2.0

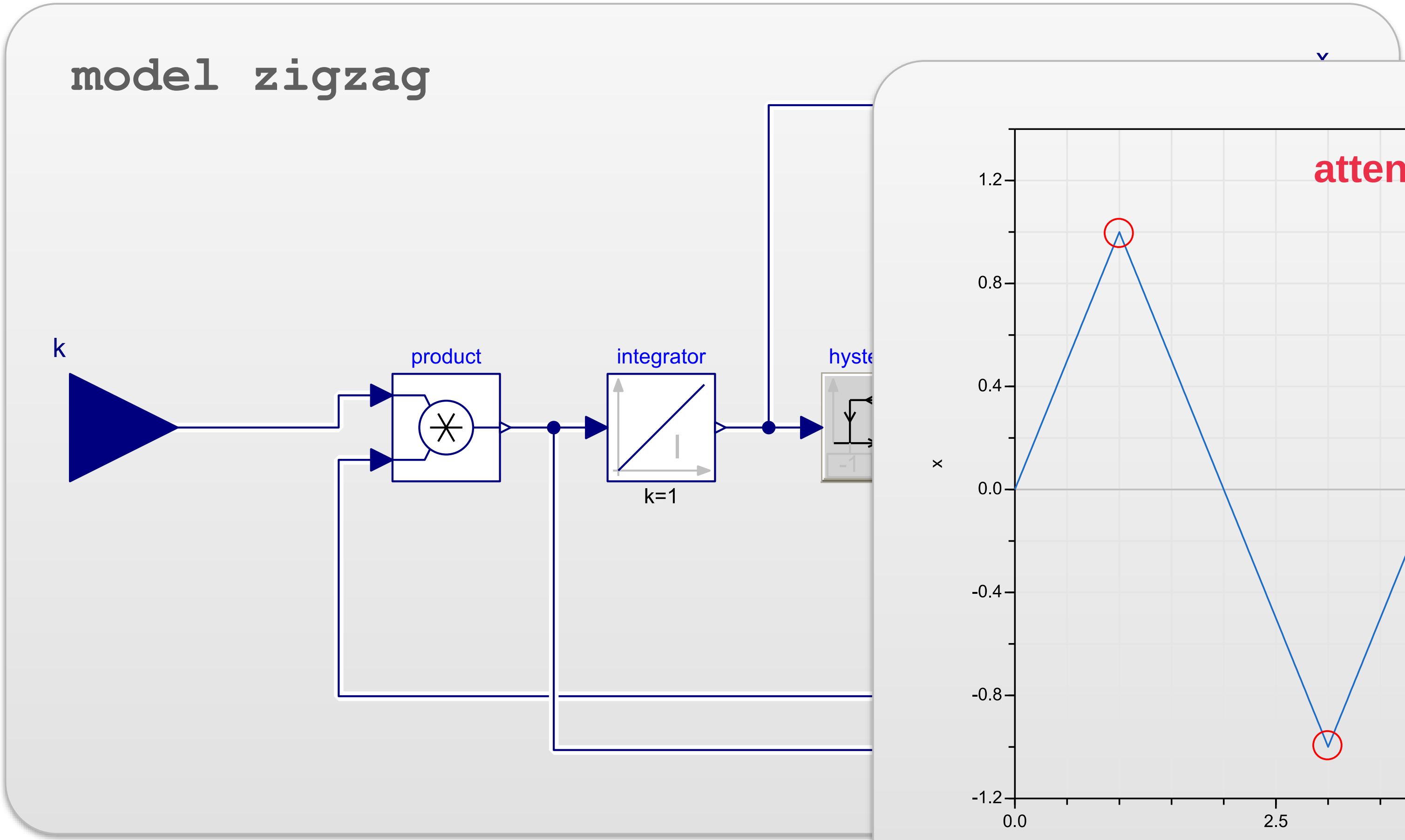


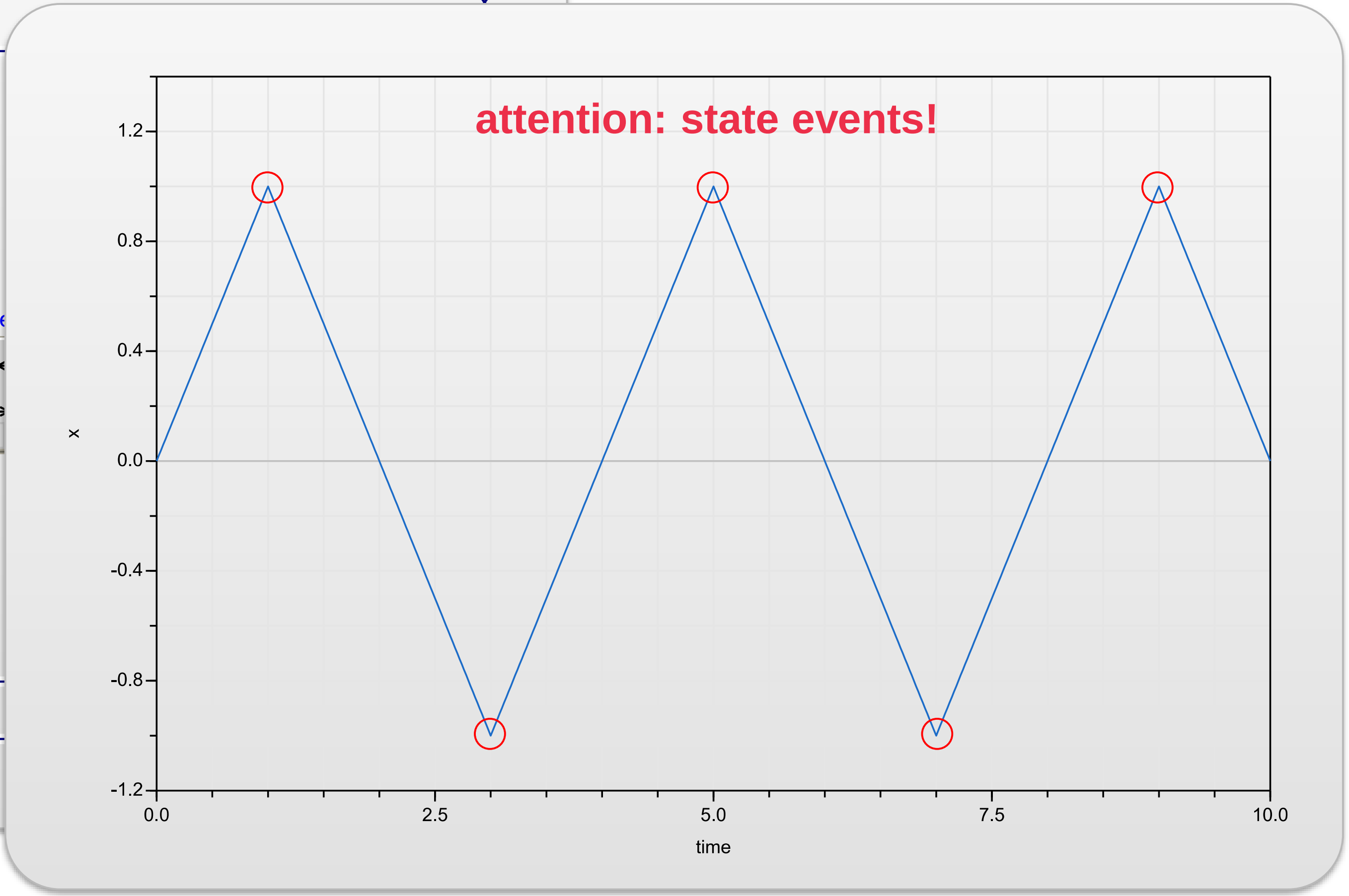
- These classes provide:
  - model description parsing
  - functions for instantiation and initialization
  - getter / setter functions using variable names (not value references)
  - functions for step-wise simulation of model
  - FMI ME: provide integrators (SUNDIALS CVODE, BOOST odeint)
    - detection and handling of internal state events

Modelica model for demo



Modelica model for demo





Demo: Importing and using an FMU for ME

- see Jupyter notebook *TestFMUModelExchange.ipynb*

jupyter TestFMUModelExchange (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Importing and using an FMU for Model Exchange in Python

The most obvious obstacle for using a bare FMU for ModelExchange is its lack of an integrator. For this reason, classes **FMUModelExchangeV1** and **FMUModelExchangeV2** provide generic methods for the integration of FMUs for ModelExchange for FMI Version 1.0 and 2.0, respectively. Instances of these classes own the actual FMU instance and are able to advance the current state up to a specified point in time, including the proper handling of FMU-internal events. The classes also provide functionality for convenient input and output handling.

The following example demonstrates the basic usage of class **FMUModelExchangeV2** (usage of class FMUModelExchangeV1 is analogous).

Loading the library and extracting an FMU

Load the FMI++ library into Python.

In [ ]:
```
import fmipp
```

Specify the FMU's model name.

In [ ]:
```
model_name = 'zigzag'
```

Specify the absolute path of the FMU. In this example, the FMU is supposed to be in the same directory as this notebook.

In [ ]:
```
import os
work_dir = os.getcwd() # get current working directory (contains 'zigzag.fmu')
path_to_fmu = os.path.join( work_dir, 'data', 'win', model_name + '.fmu' ) # path to FMU
```

Content of tutorial

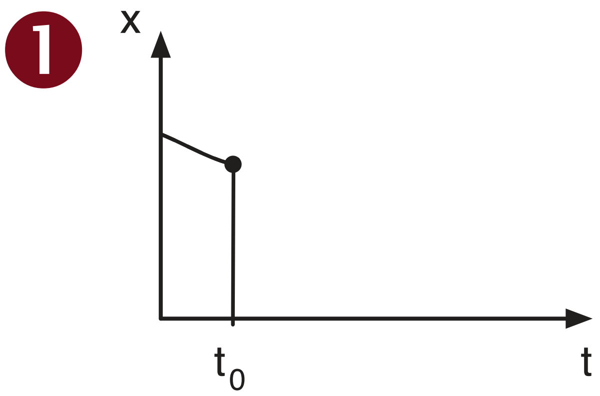
- Requirements for running demos and exercises
- Introduction to FMI, FMI++ & FMI++ Python Interface
- Installation of the FMI++ Python Interface on Windows and Linux
- Basic FMU import functionality (ME and CS)
- Advanced FMU import functionality for ME (event prediction, rollbacks, etc.)
- Exporting Python scripts as FMU for CS
- Debugging of Python scripts prior to export
- Hands-on exercises

Advanced FMU import functionality for ME

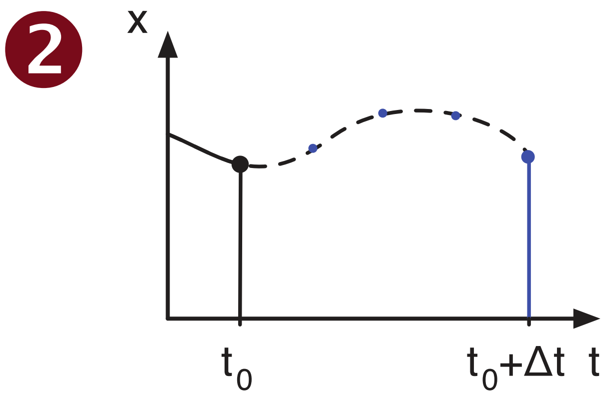
- Python package `fmipp` offers *high-level functionality* that ease the handling of FMUs
  → target the *integration* of FMUs into existing simulation software

- Example: class `IncrementalFMU`
  - combine integration of FMUs for ME with *advanced event handling* capabilities
  - intended for integrating FMUs for ME into *event-based simulations*
  - implements a *look-ahead mechanism*, where predictions of the FMU's state are incrementally computed and stored
  - most important functionality:
    - `predictState`: compute state predictions according to the current inputs
    - `updateState`: updates the state of the FMU to the specified time, i.e., it changes the actual state using previously calculated state prediction(s)
    - `syncState`: set all inputs corresponding to the specified time
    - `sync`: executes `updateState`, `syncState` and `predictState` in one go

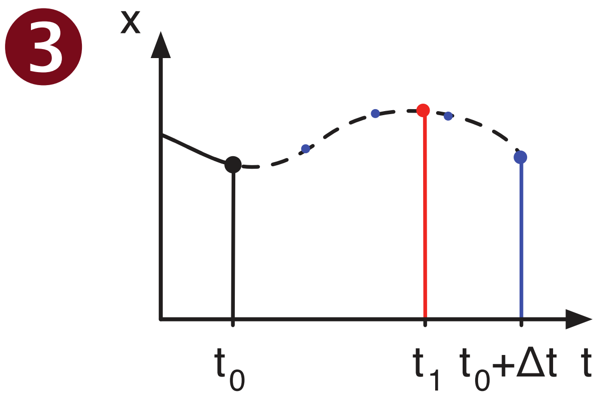
Look-ahead mechanism of class IncrementalFMU



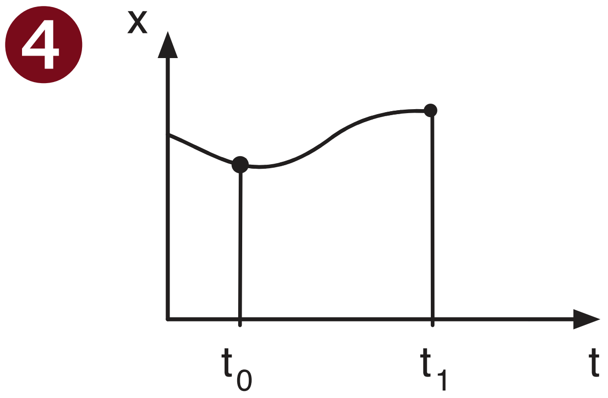
state of FMU after last synchronization



look-ahead: calculate future states

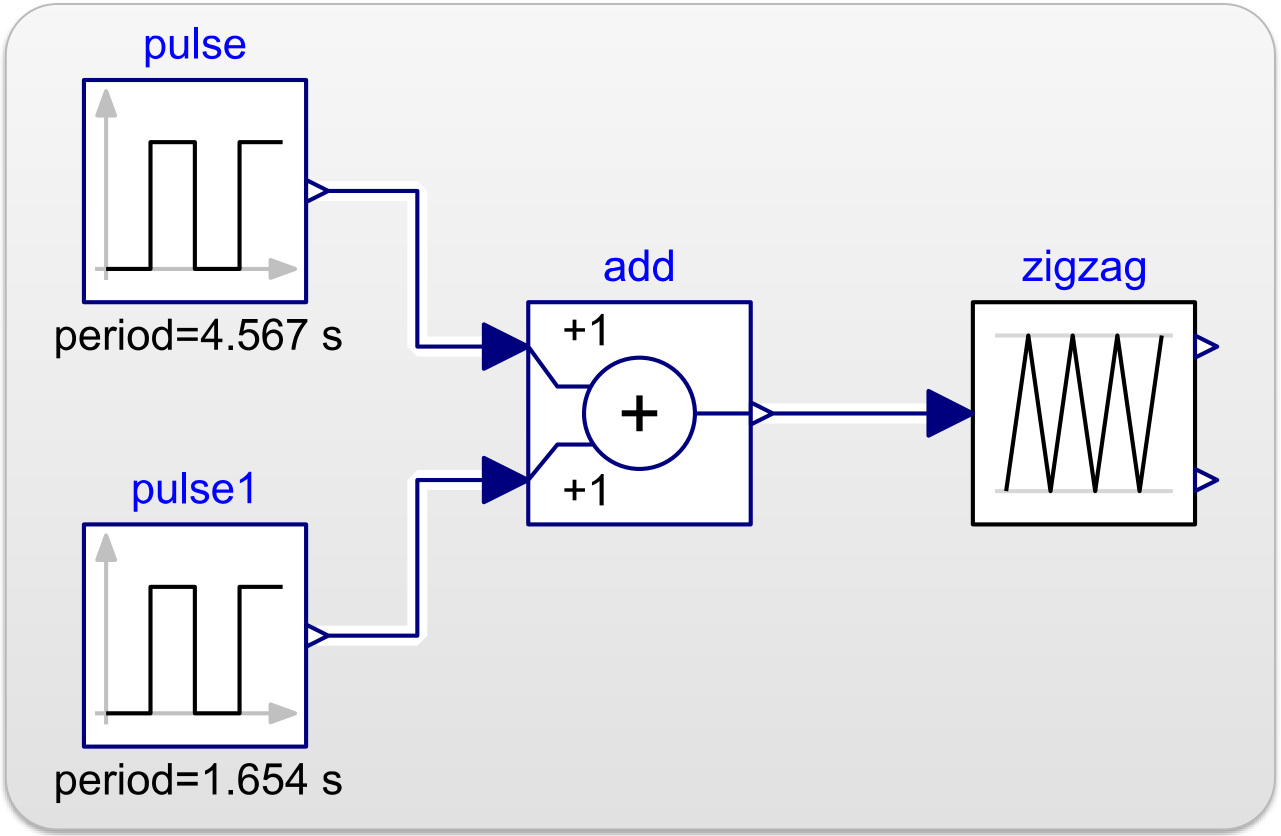


external event within look-ahead horizon

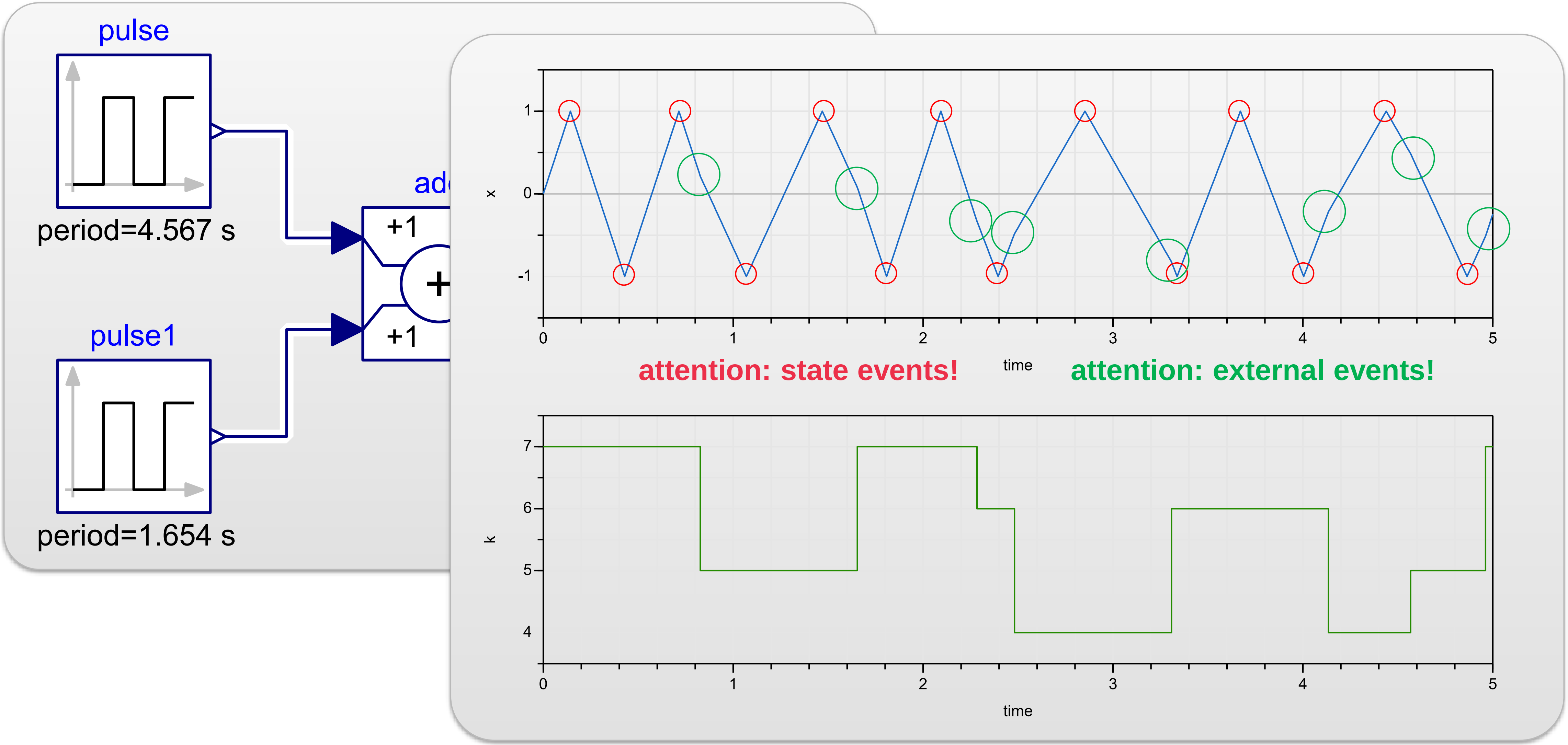


extrapolate FMU state at external event

Demo: Changing the slope (in Modelica)

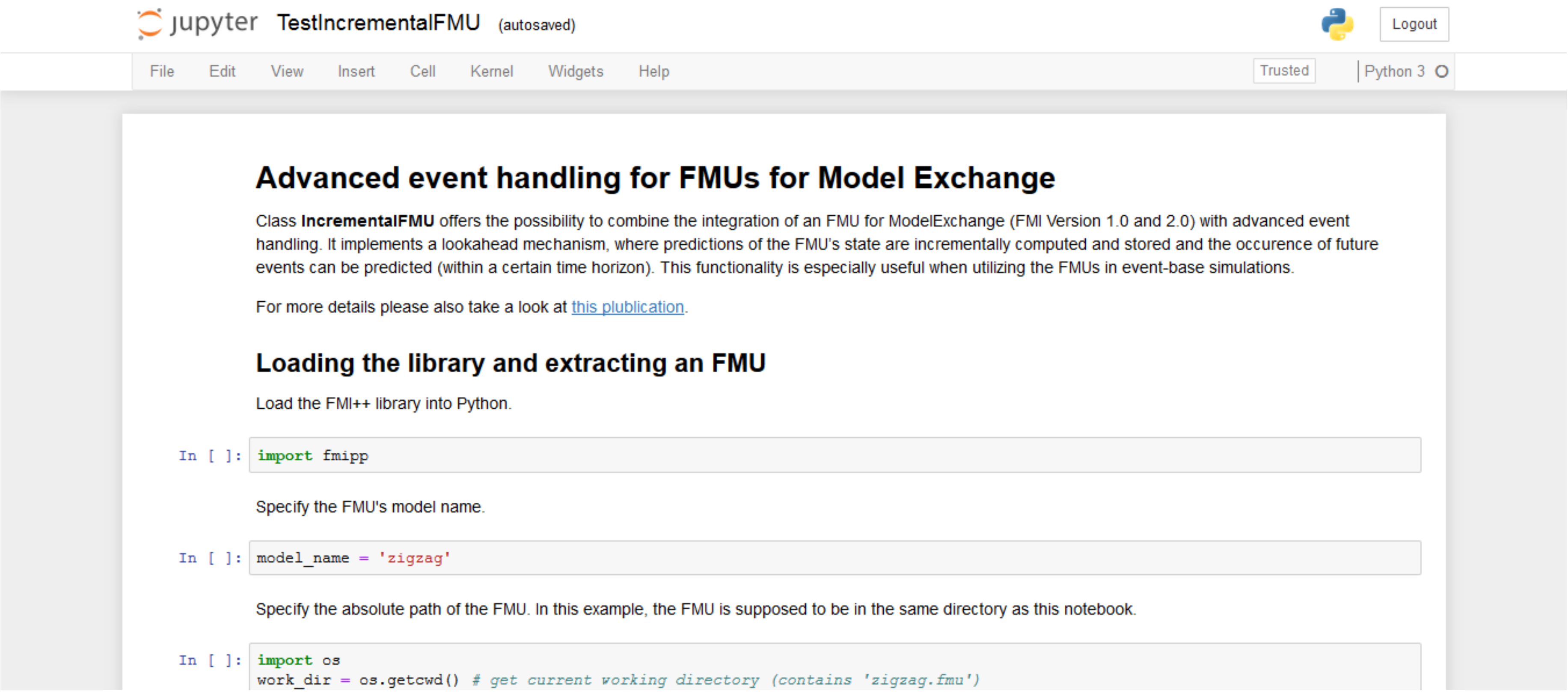


Demo: Changing the slope (in Modelica)



Demo: Advanced event handling for FMUs for ME

- changing the slope → do the same thing in Python, using “random” events
- see Jupyter notebook *TestIncrementalFMU.ipynb*



jupyter TestIncrementalFMU (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3

Advanced event handling for FMUs for Model Exchange

Class **IncrementalFMU** offers the possibility to combine the integration of an FMU for ModelExchange (FMI Version 1.0 and 2.0) with advanced event handling. It implements a lookahead mechanism, where predictions of the FMU's state are incrementally computed and stored and the occurence of future events can be predicted (within a certain time horizon). This functionality is especially useful when utilizing the FMUs in event-base simulations.

For more details please also take a look at this plublication.

Loading the library and extracting an FMU

Load the FMI++ library into Python.

In [ ]:
```
import fmipp
```

Specify the FMU's model name.

In [ ]:
```
model_name = 'zigzag'
```

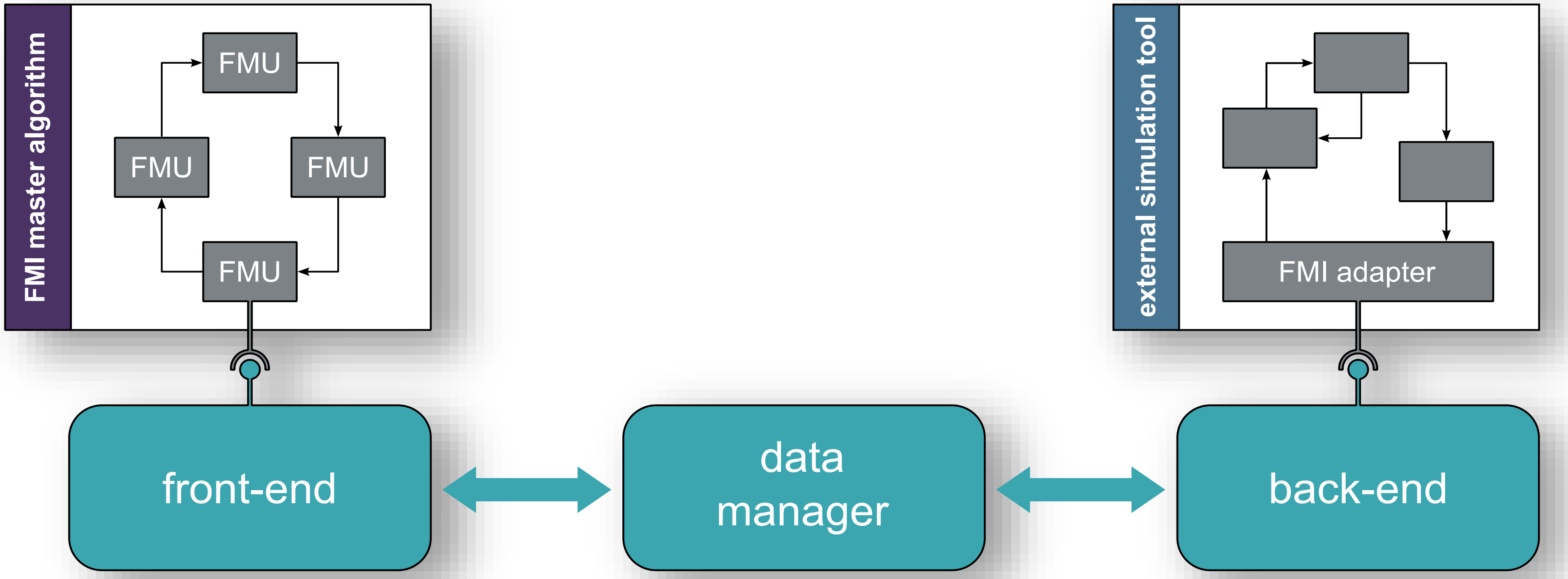
Specify the absolute path of the FMU. In this example, the FMU is supposed to be in the same directory as this notebook.

In [ ]:
```
import os
work_dir = os.getcwd() # get current working directory (contains 'zigzag.fmu')
```

Content of tutorial

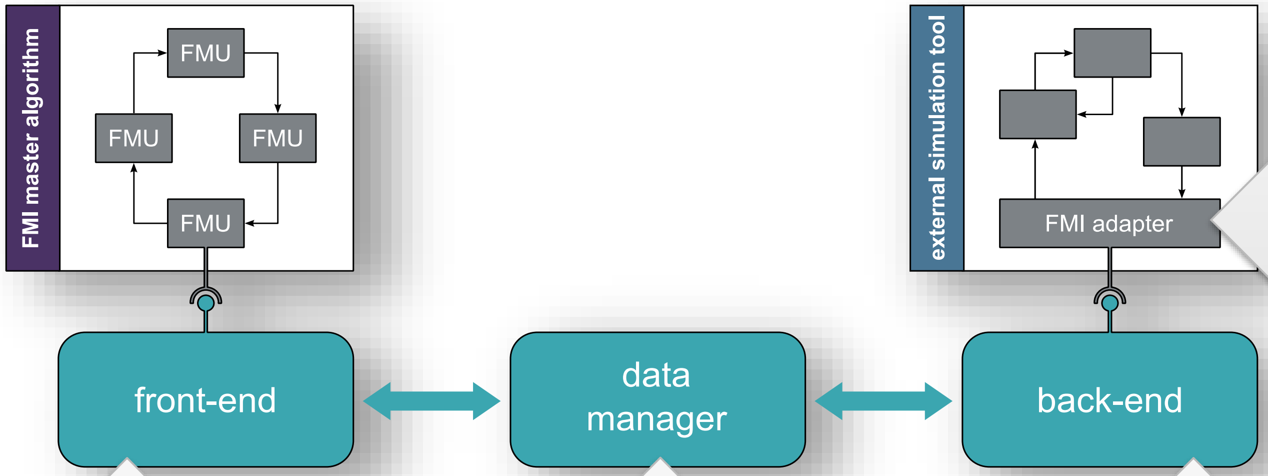
- Requirements for running demos and exercises
- Introduction to FMI, FMI++ & FMI++ Python Interface
- Installation of the FMI++ Python Interface on Windows and Linux
- Basic FMU import functionality (ME and CS)
- Advanced FMU import functionality for ME (event prediction, rollbacks, etc.)
- Exporting Python scripts as FMU for CS
- Debugging of Python scripts prior to export
- Hands-on exercises

The FMI++ approach for tool coupling



E. Widl and W. Müller: "*Generic FMI-compliant simulation tool coupling*", Proceedings of the 12th Int. Modelica Conference, 2017, pp. 1–7.

The FMI++ approach for tool coupling



- specific to the simulation tool
- part of the model that is loaded in the simulator
- retrieve information about the model at run-time, e.g., names and types of inputs and outputs

- parses model description
- initializes variables
- mapping of internal variable names and FMI value references
- starts simulation tool

- link between the front-end and the back-end
- designed to handle FMI scalar variables
- allocates & synchronizes memory for the front-end and the back-end

- generic interface to the front-end component for the FMI adapter
- establishes synchronized data exchange via the data manager

Exporting Python scripts as FMU for CS

- Python code can be made available as *FMU for Co-Simulation* (version 2.0) with the help of **`class FMIAdapterV2`**

- this class defines *two abstract methods* that have to be *implemented by the user*:

1. **`init( self, currentCommunicationPoint )`**
    - initialize input/output variables and parameters
    - specify fixed simulation time step (optional)

2. **`doStep( self, currentCommunicationPoint, communicationStepSize )`**
    - called at every simulation step (as requested by the master algorithm)

- When using such an FMU, Python is started in the background and synchronized to the master algorithm

Exporting Python scripts as FMU for CS

- For *initializing* input/output variables and parameters of type `fmiReal`, class `FMIAdapterV2` provides the following methods:
  - `defineRealParameters( self, *parameterNames )`
  - `defineRealInputs( self, *inputVariableNames )`
  - `defineRealOutputs( self, *outputVariableNames )`
- For *getting values* of parameters and input variables as well as *setting values* of output variables of type `fmiReal`, class `FMIAdapterV2` provides another set of methods:
  - `realParameterValues = getRealParameterValues( self )`
  - `realInputValues = getRealInputValues( self )`
  - `setRealOutputValues( self, outputValues )`
- Analogous functions exist for `fmiInteger`, `fmiBoolean` and `fmiString`

Exporting Python scripts as FMU for CS

- Fixed time step simulation can be enforced by calling the following method:
  - `enforceTimeStep( self, stepSize )`

Exporting Python scripts as FMU for CS

- Creating an FMU from a class inherited from `FMIAdapterV2` is done by calling this function:
  - `createFMU( fmu_backend, fmi_model_identifier, fmi_version, verbose, litter, start_values, optional_files )`
    - `fmu_backend`: class implementing the abstract base class FMIAdapterV2 (class derived from FMIAdapter)
    - `fmi_model_identifier`: FMI model identifier (`str`)
    - `fmi_version`: FMI version (`str`, `1` or `2`, default: `2`)
    - `verbose`: turn on log messages (`boolean`, default: `False`)
    - `litter`: do not clean-up intermediate files (`boolean`, default: `False`)
    - `start_values`: start values may be specified for paramters and input variables (`None` or `dict`, default: `None`)
    - `optional_files`: additional files (e.g., for weather data) may be specified as extra arguments; these files will be automatically copied to the resources directory of the FMU (`None` or `list` of `str`, default: `None`)

Demo: Exporting Python scripts as FMU for CS

- see Jupyter notebook *TestFMUExport.ipynb*

jupyter TestFMUExport (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3

Exporting Python scripts as FMU for Co-Simulation

The functionality of Python can be made available as FMU for Co-Simulation (version 1.0 or 2.0) with the help of class **FMIAdapterV1** or **FMIAdapterV2**, respectively. These class define two abstract methods that have to be implemented by the user:

- Method *init(self, currentCommunicationPoint)* is intended to initialize input/output variables and parameters needed for co-simulation. Optionally, a fixed simulation time step can be specified.
- Method *doStep(self, currentCommunicationPoint, communicationStepSize)* is called at every simulation step (as requested by the master algorithm).

By deriving a new class from class **FMIAdapterV1** or **FMIAdapterV2** and implementing these two methods, virtually all functionality of Python can be made available via an FMU for Co-Simulation. When using such an FMU, Python is started in the background and synchronized to the master algorithm.

Creating an FMU

Import the requried functionality from the FMI++ library.

In [ ]:
```
from fmipp.export.createFMU import createFMU
```

File *FMUExportTest.py* defines class **FMUExportTest**, which derives from class **FMIAdapterV2**. Import the class, in order to export it.

In [ ]:
```
from TestClassFMUExport import FMUExportTestClass
```

Specify the model name of the FMU.

In [ ]:
```
model_name = 'FMUExportTestCS'
```

Content of tutorial

- Requirements for running demos and exercises
- Introduction to FMI, FMI++ & FMI++ Python Interface
- Installation of the FMI++ Python Interface on Windows and Linux
- Basic FMU import functionality (ME and CS)
- Advanced FMU import functionality for ME (event prediction, rollbacks, etc.)
- Exporting Python scripts as FMU for CS
- Debugging of Python scripts prior to export
- Hands-on exercises

Debugging of Python code prior to export

- Implemented Python code can be *tested* and *debugged* before exporting it as an FMU for Co-Simulation
- *Emulate the master algorithm* and check what the code does *within Python*:
  - Interact with classes inherited from **`FMIAdapterV2`** using functions **`init(…)`** and **`doStep(…)`**
  - Set and retrieve values using these dedicated methods:
    - **`debugSetRealInputValues(…)`**
    - **`debugGetRealOutputValues(…)`**
    - etc.

Demo: Debugging of Python code prior to export

- see Jupyter notebook *TestFMUExportDebug.ipynb*

jupyter TestFMUExportDebug (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Debugging code before exporting it as FMU

File *FMUExportTest.py* defines class **FMUExportTest**, which derives from class **FMIAdapterV2**. Import the class, in order to debug it.

In [ ]:
```
from TestClassFMUExport import FMUExportTestClass
```

Create an instance of class **FMUExportTest** to debug its functionality.

In [ ]:
```
test = FMUExportTestClass()
```

Initialize the new instance (at time *t=0*). When executing this function, a runtime warning will be displayed, saying that the "FMI++ export interface is not active". When debugging a class, this warning can be ignored.

In [ ]:
```
test.init( 0. )
```

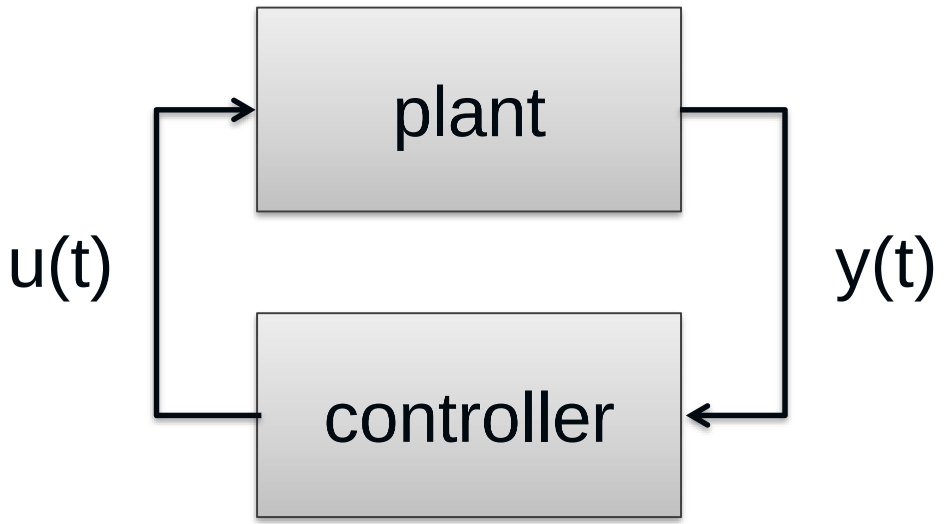
Use functions *debugSetX* to set values for parameters and input variables.

In [ ]:
```
test.debugSetRealParameterValues( { 'pr_y' : 2.2, 'pr_x' : 1.1 } )
test.debugSetRealInputValues( { 'ir_y' : 2., 'ir_x' : 1. } )

test.debugSetIntegerParameterValues( { 'pi_y' : 3, 'pi_x' : 6 } )
test.debugSetIntegerInputValues( { 'ii_y' : 4, 'ii_x' : 5 } )

test.debugSetBooleanParameterValues( { 'pb_y' : True, 'pb_x' : True } )
test.debugSetBooleanInputValues( { 'ib_y' : False, 'ib_x' : True } )
```

Content of tutorial

- Requirements for running demos and exercises
- Introduction to FMI, FMI++ & FMI++ Python Interface
- Installation of the FMI++ Python Interface on Windows and Linux
- Basic FMU import functionality (ME and CS)
- Advanced FMU import functionality for ME (event prediction, rollbacks, etc.)
- Exporting Python scripts as FMU for CS
- Debugging of Python scripts prior to export
- Hands-on exercises

Example application: Rapid prototyping of controls



Example application: Rapid prototyping of controls

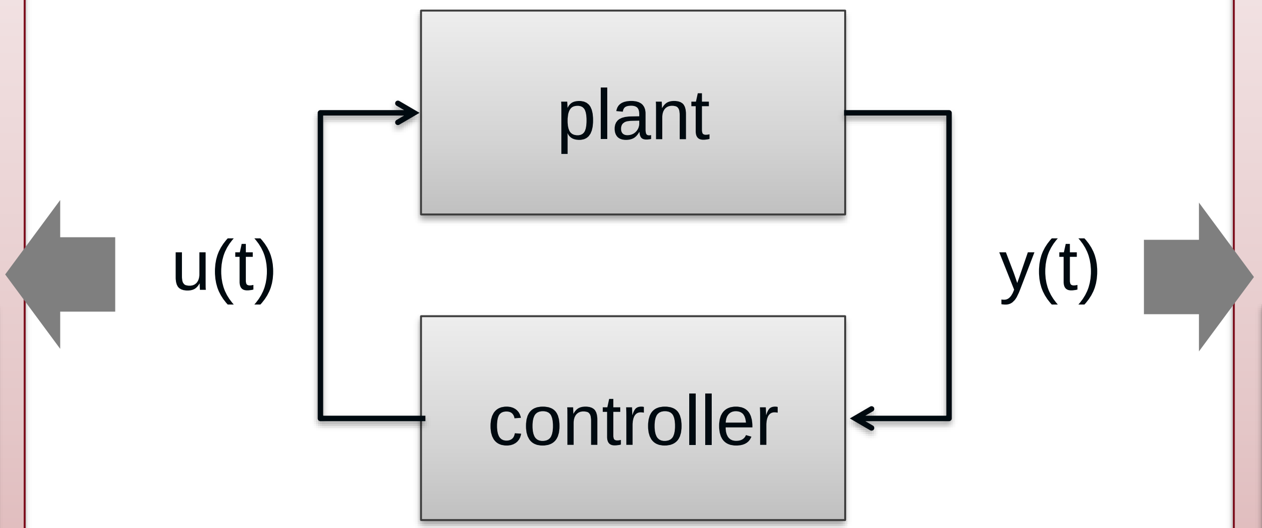
Option 1:

PYTHON script

- implements controller
- executes main simulation loop (master)

MODELICA model

- implements plant model
- imported as FMU for ME



Option 2:

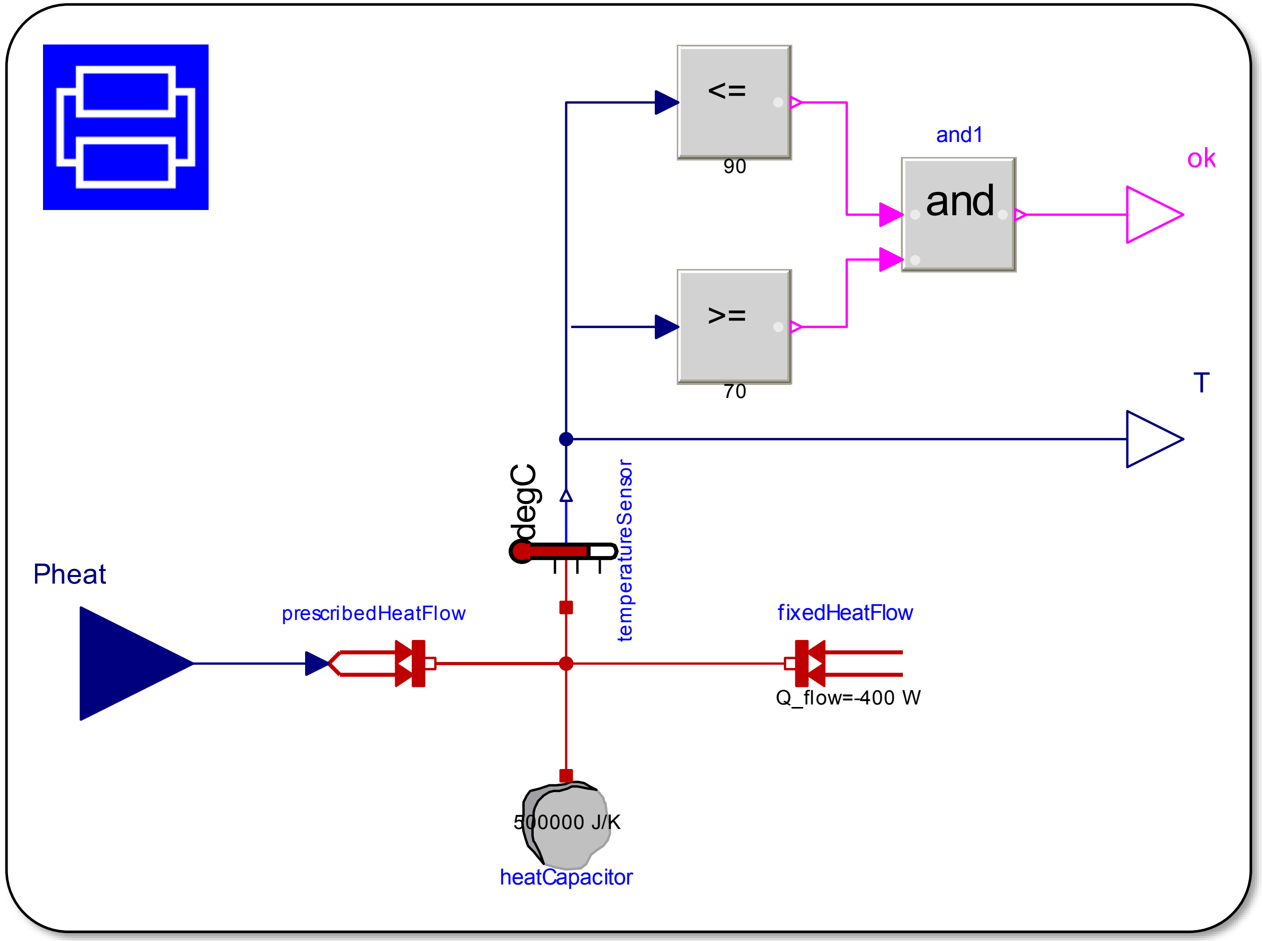
MODELICA model

- implements plant model
- executes main simulation loop (master)

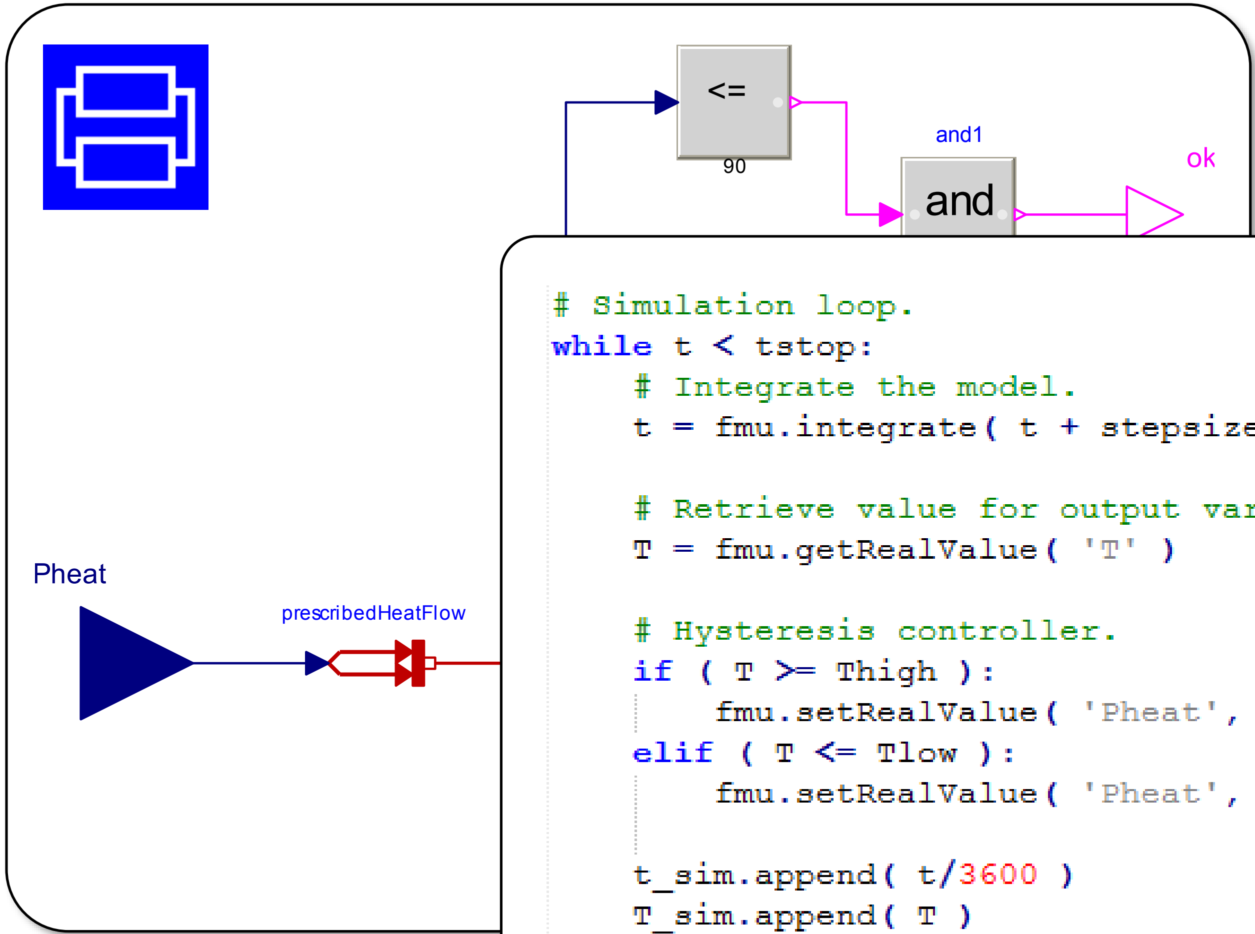
PYTHON script

- implements controller
- imported as FMU for CS

Option 1: Import FMUs in Python code

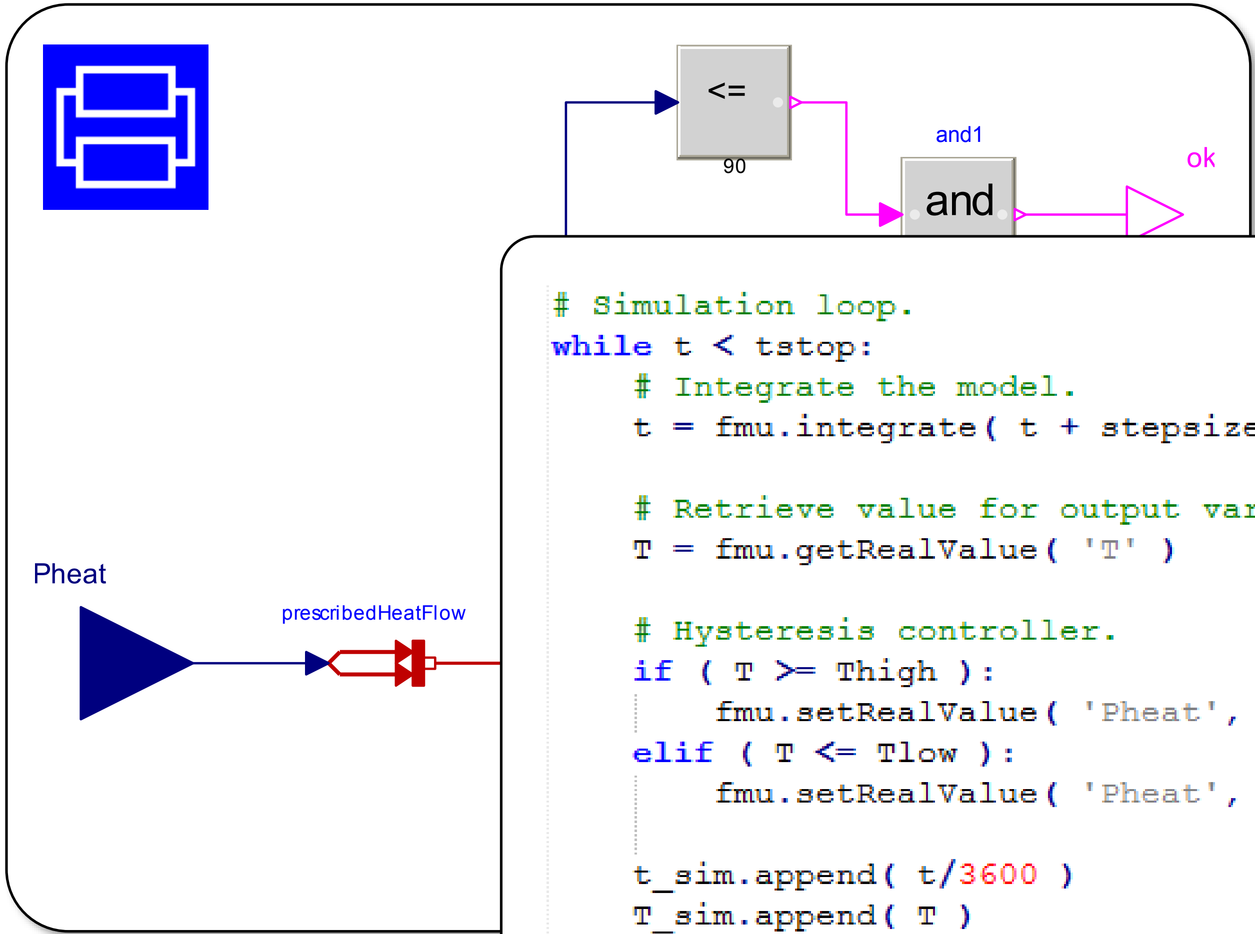


Option 1: Import FMUs in Python code

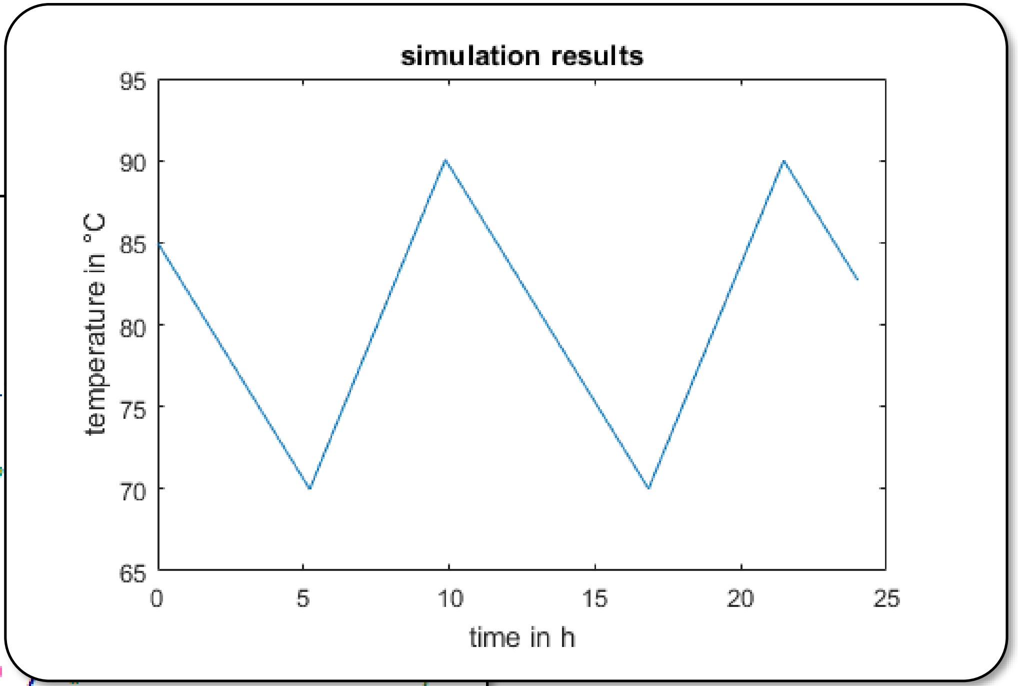



```python
# Simulation loop.
while t < tstop:
    # Integrate the model.
    t = fmu.integrate( t + stepsize, integrator_stepsize )

    # Retrieve value for output variable T.
    T = fmu.getRealValue( 'T' )

    # Hysteresis controller.
    if ( T >= Thigh ):
        fmu.setRealValue( 'Pheat', 0.0 ) # turn off heating
    elif ( T <= Tlow ):
        fmu.setRealValue( 'Pheat', 1e3 ) # turn on heating

    t_sim.append( t/3600 )
    T_sim.append( T )
```

Option 1: Import FMUs in Python code



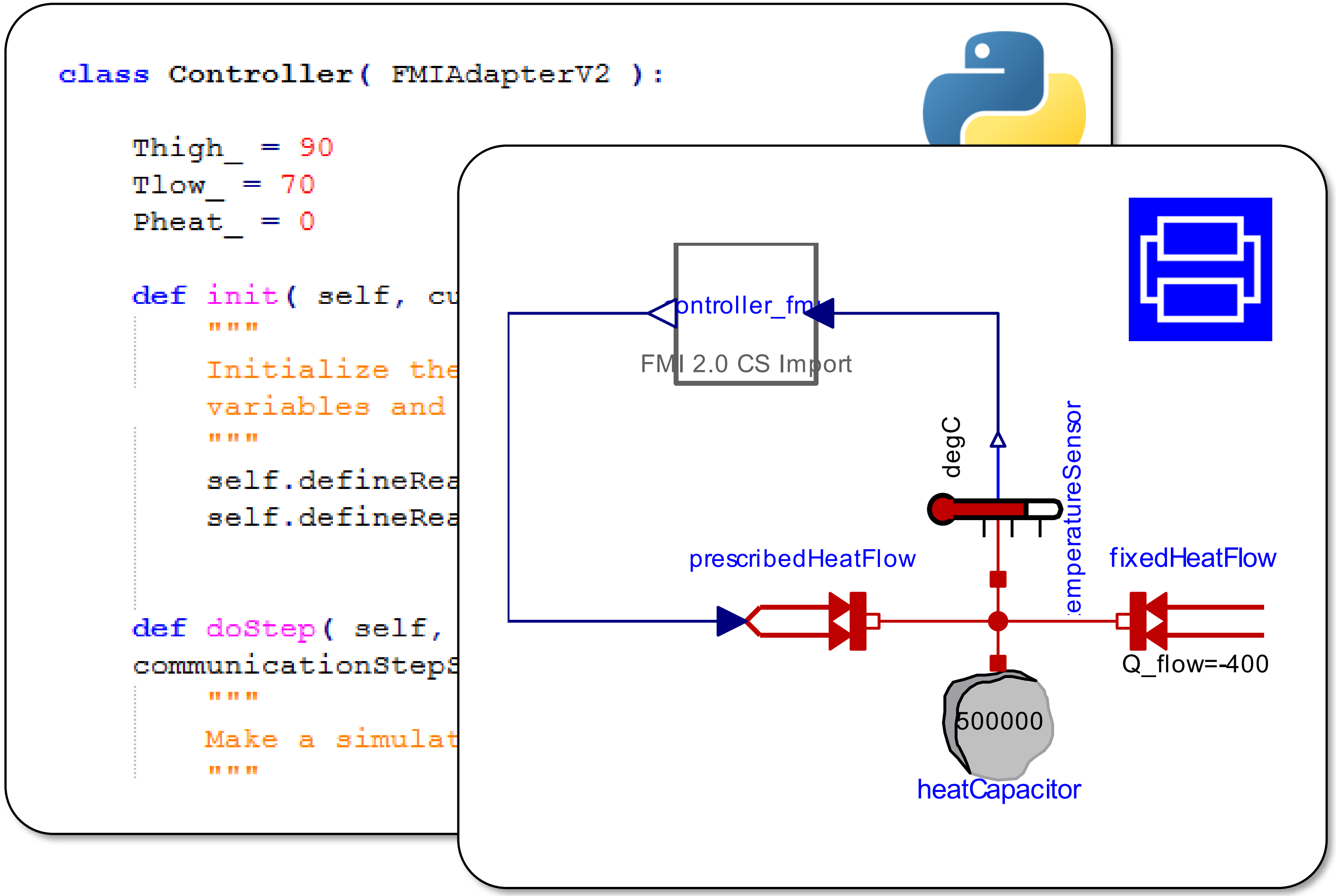

```python
# Simulation loop.
while t < tstop:
    # Integrate the model.
    t = fmu.integrate( t + stepsize, i

    # Retrieve value for output variab
    T = fmu.getRealValue( 'T' )

    # Hysteresis controller.
    if ( T >= Thigh ):
        fmu.setRealValue( 'Pheat', 0.0 )
    elif ( T <= Tlow ):
        fmu.setRealValue( 'Pheat', 1e3 ) # turn on heating

    t_sim.append( t/3600 )
    T_sim.append( T )
```



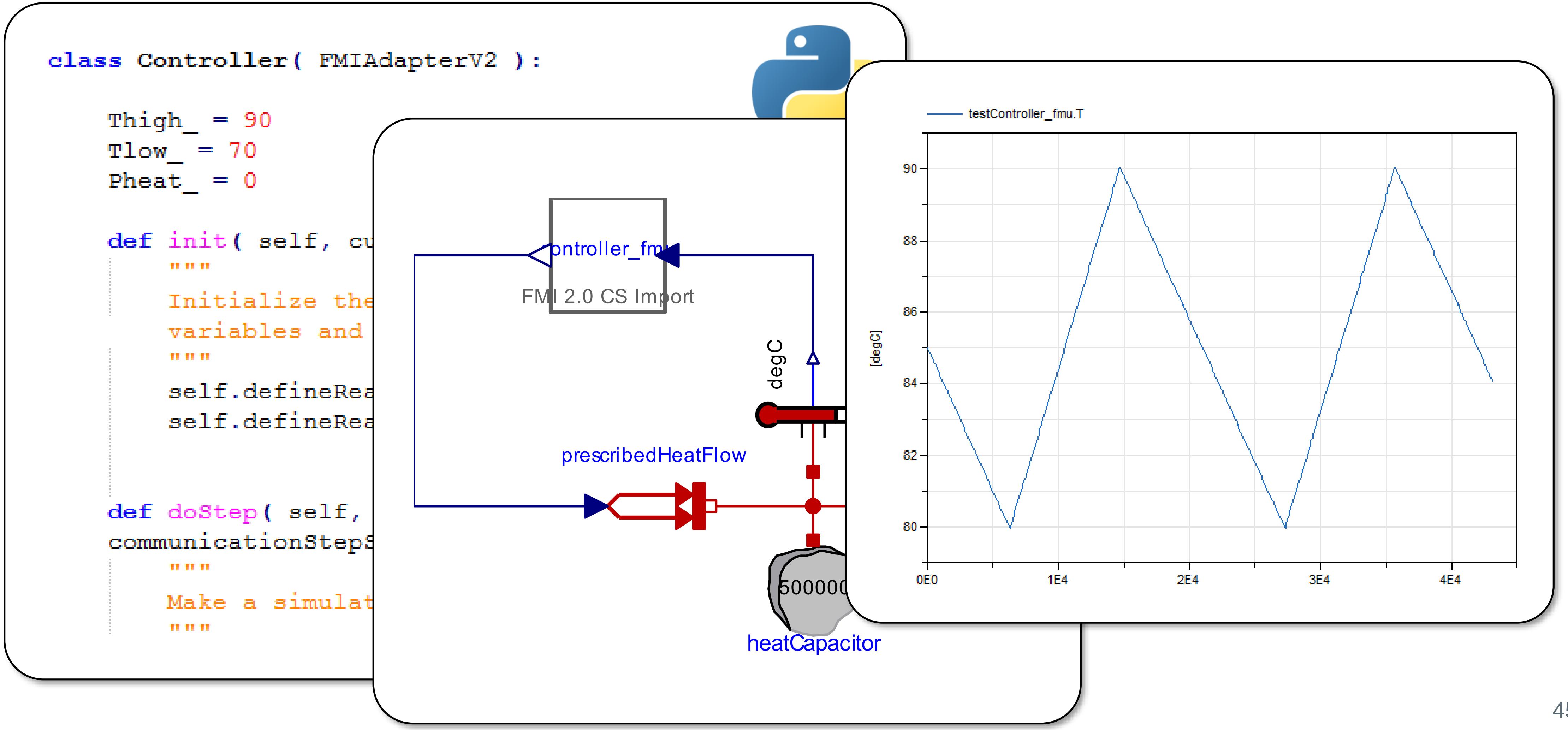
Option 2: Export Python code as FMU

```python
class Controller( FMIAdapterV2 ):

    Thigh_ = 90
    Tlow_ = 70
    Pheat_ = 0

    def init( self, currentCommunicationPoint ):
        """
        Initialize the FMU (definition of input/output
        variables and parameters, enforce step size).
        """
        self.defineRealInputs( 'T' )
        self.defineRealOutputs( 'Pheat' )


    def doStep( self, currentCommunicationPoint,
    communicationStepSize ):
        """
        Make a simulation step.
        """
```

Option 2: Export Python code as FMU



Option 2: Export Python code as FMU



Links

- FMI++ library: http://fmipp.sourceforge.net
- fmipp source code repository: https://github.com/AIT-IES/py-fmipp/
- fmipp on Python package index: https://pypi.org/project/fmipp/
- Code Ocean compute capsule with demos: https://doi.org/10.24433/CO.9880202.v2

Acknowledgements

- The development of the FMI++ Python Interface has been partly supported by the project “IntegrCiTy” in the ERA-NET Cofund Smart Cities and Communities call.

- The development of the FMI++ Python Interface has been partly supported by the European Community’s Horizon 2020 Program (H2020/2014-2020) under project “ERIGrid” (Grant Agreement No. 654113).

Have fun with the fmipp package!!!