Portable runtime environments for Python-based FMUs: Adding Docker support to UniFMU

Thomas Schranz[1] Christian Møldrup Legaard[2] Daniella Tola[2] Gerald Schweiger[1]

[1]Graz University of Technology, Austria, {thomas.schranz,gerald.schweiger}@tugraz.at
[2]DIGIT, Department of Electrical and Computer Engineering, Aarhus University, Denmark, {cml,dt}@ece.au.dk

Abstract

Co-simulation is a means to combine and leverage the strengths of different modeling tools, environments and formalisms and has been applied successfully in various domains. The Functional Mock-Up Interface (FMI) is the most commonly used standard for co-simulation. In this paper we extend UniFMU, a tool that allows users to build Functional Mock-Up Units (FMUs) in virtually any programming language, to support execution within Docker. As a result the generated FMUs can be distributed in an environment containing all runtime dependencies. To describe the process of creating Dockerized FMUs using UniFMU, we show how to model and co-simulate a robotic arm and a controller using two Python-based FMUs.

Keywords: FMI, Co-Sim, Python, Tool-Coupling, Docker

1 Introduction

Complex, heterogeneous systems can be found throughout all fields of science and industry. Due to increasing complexity, market competition and specialization, system evaluation and simulation-based analysis has become more and more difficult (G. Schweiger et al. 2019). However, there often exist partial models for different parts of these systems, albeit in different domains and developed using different tools (Gomes et al. 2018). Co-simulation is a means to combine and leverage the strengths of different modeling tools, environments and formalisms (Cremona et al. 2019) and has been applied successfully in various domains (Gerald Schweiger et al. 2018; Pedersen et al. 2017; Nageler et al. 2018). The Functional Mock-Up Interface (FMI) was found to be the most promising standard for continuous time, discrete event, and hybrid co-simulation in a survey by (G. Schweiger et al. 2019). FMI is maintained by the Modelica association (Modelica Association 2021); it can be used to co-simulate components packaged as Functional Mock-Up Units (FMUs), each of which can be built using a different FMI-enabled modeling tool.

1.1 Co-Simulation Tools

With Open Modelica (Asghar and Tariq 2010), Simulink[1] or 20-sim[2] users can generate FMUs based on common modeling languages such as Modelica or MATLAB/Simulink using a graphical interface. The Universal Functional Mock-up Unit (UniFMU) (Legaard et al. 2021) tool allows users to build FMUs from arbitrary code in any programming language; it supports Python, C# and Java out-of-the-box. It uses a precompiled binary wrapper that implements the methods specified in the FMI standard's C-headers to spawn a process that executes the FMU's actual code. This way the FMU can be built from code written in an interpreted language or a language that uses automatic garbage collection. However, this setup, allowing for this kind of flexibility, requires the host machine to provide the process with all runtime dependencies which limits portability, especially between different host machines, and potentially necessitates a complicated setup procedure.

There exists a number of distributed, FMI-based co-simulation tools, many of which were analyzed in (Hatledal et al. 2019). However, all of them require a tight coupling between the co-simulation components and the master algorithm. ProxyFMU, a tool developed by the authors of (Hatledal et al. 2019) decouples the FMUs, in a way that they become independent of the master algorithm in a client/server solution that supports JavaScript, Python, C++ and the JVM on the client side.

The authors of (Hinze et al. 2018) propose a method for running FMUs inside Docker containers by placing the entire FMU archive inside the container and extending the master algorithm with a *remote procedure call* protocol. A distinction between their work and our approach is that FMUs generated using UniFMU work with any FMI-enabled master algorithm without the need to implement any additional protocols.

1.2 Contributions

In this paper we extend UniFMU using the virtualization environment Docker[3], such that the FMUs can be shipped with all runtime dependencies. We provide a general

[1]http://www.mathworks.com/products/simulink
[2]http://www.20sim.com
[3]https://www.docker.com

mechanism that can be leveraged for all languages supported by the tool. The resulting FMUs have nearly the same portability as compiled FMUs (except for the dependency on Docker) and require no language-specific setup procedure, but still allow the use of non-compiled languages and languages that use automatic garbage collection. To explain the process of creating Dockerized FMUs using UniFMU, we show how to model and co-simulate a robotic arm and a controller using two Python-based FMUs. The UniFMU tool (with the extensions for Dockerization) is available on Github[4]. The FMUs described in this paper can be found in a separate repository[5].

The rest of the paper is structured as follows. First, section 2 introduces a robotic arm and a controller which is used as a case study throughout the paper. Next, section 3 provides an introduction to UniFMU and describe how the robotic arm and controller can be implemented as FMUs using the tool. Then, section 4 describes the extension of UniFMU that allows FMUs and their dependencies to be deployed inside Docker containers. Afterwards, section 6 provides a discussion of the results and outlines future work on the tool. Finally, section 7 provides concluding remarks.

2 Case Study

To exemplify the process of using UniFMU we consider the case of modeling a robotic arm coupled to a controller as depicted in Figure 1. The example is chosen to highlight how different modeling formalisms can be realized by the tool. Specifically, the robotic arm described in subsection 2.1 is inherently continuous, whereas the controller described in subsection 2.2 is discrete.

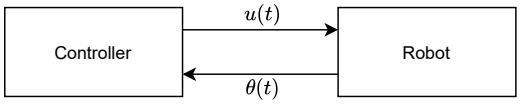

Figure 1. Connection between controller and robot model.

2.1 Robotic Arm

The robotic arm is modeled as a controlled inverted pendulum. The states of the system are its angle θ, the angular velocity ω and the current running through the coils of the electrical motor i. The dynamics of the robotic arm are described by Equation 1. Note that contrary to the visualization shown in Figure 7 the model only considers a single joint that rotates around a single axis.

$$f(x) = \begin{bmatrix} \dot{\theta} \\ \dot{\omega} \\ \dot{i} \end{bmatrix} = \begin{bmatrix} \omega \\ \frac{K \cdot i - b \cdot \omega - m \cdot g \cdot l \cdot cos(\theta)}{J} \\ \frac{u \cdot V_{abs} - R \cdot i - K \cdot \omega}{L} \end{bmatrix} \quad (1)$$

where:

The derivative of the angle $\dot{\theta}$ is, per definition, equal to the velocity of the arm ω. The derivative of the angular velocity $\dot{\omega}$ is determined by the torque coefficient $K = 7.45\ s^{-2}A^{-1}$, the current i, the motor-shaft friction $b = 5.0\ kg \cdot m^2 \cdot s^{-1}$ and the gravity acting on the arm, denoted by $m \cdot g \cdot l \cdot \cos(\theta)$, with $m = 5.0\ kg$, $g = 9.81 ms^{-2}, l = 1.0\ m$. The change in current is determined by the input from the controller u, the voltage across the coils $V_{abs} = 12.0\,V$, the resistance $R = 0.15\ \Omega$ and the motor's inductance $L = 0.036\,H$.

2.2 Controller

A proportional-integral-derivative (PID) controller (Åström and Murray 2010) is used to generate the control signals sent to the robotic arm. The continuous formalization of the controller is given by:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t) \quad (2)$$

where $e(t)$ is a measure of the error of the variable being controlled and K_p, K_i and K_d are coefficients used to tune how the proportional and derivative terms are weighted. In case of the robot, the controller is trying to minimize the error between the desired angle $\theta^*(t)$ and the true angle $\theta(t)$. Thus, the error is defined as $e(t) = \theta^*(t) - \theta(t)$.

In practice, most controllers are implemented digitally, which means that derivatives and integrals must be replaced by discrete approximations. There are several ways to do this, the simplest being to replace derivatives by first-order differences

$$\dot{e}(t_k) \approx \dot{e}_k = \frac{e_k - e_{k-1}}{T},$$

and integrals by sums

$$\int_0^{t_k} e(t_k) \approx E_k = \sum_{n=1}^{N} e_{k_n} \cdot T$$

where $e_k = e(t_k)$, T is the sampling time and $N = t_k/T$ is the number of samples between time 0 and t_k. After replacing the continuous definitions in Equation 2 we obtain an equation that can be implemented on a discrete controller

$$u_k = K_p e_k + K_i E_k + K_d \dot{e}_k \quad (3)$$

This discretization scheme is simple to implement

3 Modeling

In this section, we describe how UniFMU is installed and how it is used to generate an FMU. We provide a brief overview of the resulting FMU's structure and method of operation. Subsequently, we describe the FMUs used to model the robotic arm and the controller. For illustrative purposes both FMUs are implemented in Python, however in the general case they can be implemented in a mix of languages.

[4] https://github.com/INTO-CPS-Association/unifmu

[5] https://github.com/Daniella1/robot_unifmu

DOI 10.3384/ecp21181419

3.1 Creating an FMU using UniFMU

UniFMU is a command line interface (CLI) that can be installed through Python's package installer `pip` or from source following the instructions in the official repository. Installation through `pip` uses a single command:

```
pip install unifmu
```

It should be noted that the FMUs generated with UniFMU do not require Python during runtime, unless the FMUs themselves are implemented in Python. To generate an FMU the tool has to be invoked with the subcommand `generate`, and supplied with the language the FMU is implemented in and the name it should have; for a Python-based FMU with the name `robot` this looks like:

```
unifmu generate python robot
```

This generates an FMU with the structure shown in Figure 2. The `binaries` directory contains a precompiled wrapper for Windows, Linux and MacOS that implements the methods specified in the FMI's C-headers and relays them to the actual implementation of the FMU found in the `resources` directory. `model.py` defines a class that declares a set of methods that correspond to the methods in the FMI standard, such as FMI's `fmi2DoStep` which is implemented by the Python method `do_step`. The actual overwrite used to model the robotic arm can be seen in Listing 1.

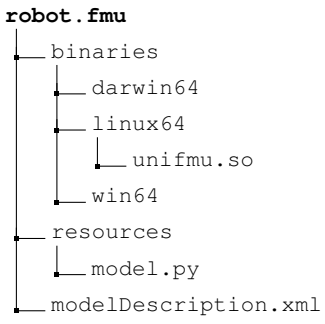

Figure 2. The directory structure for a Python FMU. Note that several files generated by the tool are omitted for simplicity.

3.2 Robotic Arm FMU

The robotic arm FMU is implemented in Python using a numerical solver provided by the *SciPy* (Virtanen et al. 2020) package. The general procedure for solving an ODE using Scipy is to define a function which evaluates the derivative for a given combination of state and time. Using Equation 1 as a reference the function $f(\cdot)$ can be defined as shown in Listing 1.

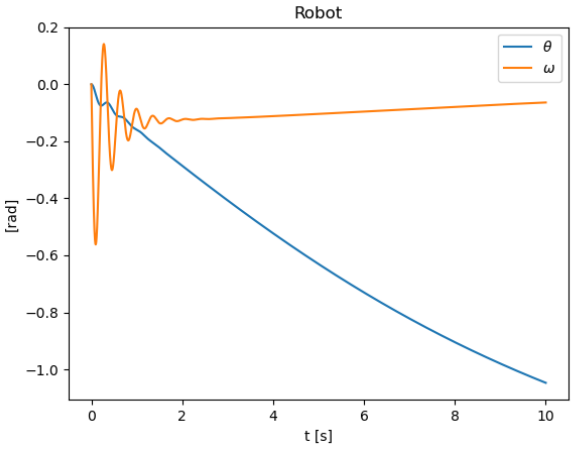

Figure 3. Standalone test of robotic arm, with values $\theta_0 = \omega_0 = i_0 = u(t) = 0$.

```
def do_step(
    self,current_time,step_size,no_step_prior
    ):
    def f(t, y):
        theta, omega, i = y
        tau=self.k1*np.cos(theta)
        domega=(self.K*i-self.b*omega-tau)/
    self.J
        di=(self.V-self.R*i-self.K*omega)/
    self.L
        dtheta=omega
        return dtheta, domega, di
    res=solve_ivp(f,(
    current_time,current_time+step_size),y0)
    self.theta,self.omega,self.i=res.y[:,-1]
    return Fmi2Status.ok
```

Listing 1. Implementation of the `fmi2DoStep` method for the robotic arm FMU.

Given the definition of the derivative, the `solve_ivp` function can be used to obtain the solution for the next step of the FMU and allows users to choose between solvers. However, it is also possible to use any other Python library providing numerical solvers or to implement a custom solver. This is a very flexible solution as it allows users to choose the type of solver that is suitable for the particular ODE. After solving the ODE, the newly estimated state is assigned to the instance, where it can be accessed from other methods and the FMI.

To test the dynamics of the robotic arm FMU, a small test program is written in Python which invokes the *do_step* several times. The results are shown in Figure 3 for initial state and input $\theta_0 = \omega_0 = i_0 = u(t) = 0$. We see that the angle of the robot decreases from 0 to -1 over 10 seconds.

3.3 Controller FMU

The controller-FMU implements a simple control algorithm that determines the signal sent to the motor based on the difference between the desired and the actual current angle. Similarly to how Equation 1 was translated into a

Python function describing the derivative of the state, we use the control policy Equation 3 as a reference to implement the expression shown in Listing 2.

```
def do_step(self, current_time, step_size,
    no_step_prior):
    err=self.setpoint_t-self.measured_t
    self.I=self.I+err*step_size
    D=(err-self.p_err)/step_size
    self.u=self.Kp*err+self.Ki*self.I+self.Kd
    *D
    self.p_err = err
    return Fmi2Status.ok
```

Listing 2. Implementation of the `fmi2DoStep` method for the controller-FMU.

In most practical situations, controllers are implemented on a processing unit where updates to the output would happen at a fixed update rate determined by the controller's clock frequency and the number of operations needed at each update.

An implicit assumption of our model is that the step-size used by the solver matches the update rate of the controller. For small step sizes, the discrete approximation implemented by the model remains relatively accurate. However, for larger step sizes the accuracy of the discretization scheme is reduced, which may ultimately cause the closed-loop system to become unstable. A solution to mitigate the discretization error is to use a more sophisticated discretization scheme, such as Tustin's method (Franklin, Powell, and Emami-Naeini 2020).

As in the robotic arm FMU, we can also use any external library for modeling the controller. For instance, a package such as *python-control*[6], can be used to evaluate the performance of different controllers.

The functionality of the controller can be verified by writing a small test program in Python that invokes the *do_step* method of the FMU. To examine the closed-loop behavior, the robot is replaced with a simple linear model described by the ODE $\dot{\theta} = u$. Executing the Python test program, we obtain the step-response of the closed-loop system (with surrogate model) as depicted in Figure 4.

4 Docker Support

A key contribution of this paper is extending UniFMU so that the generated FMUs can be executed within a virtualization environment using Docker. To create a Dockerized FMU the user can append the *–dockerize* switch to UniFMU's `generate` subcommand:

```
unifmu generate python --dockerize robot
```

The functionality is available for Linux and macOS and all languages that the tool supports. Windows support is under development, but is held back by limitations of Docker's networking capabilities when running on Windows.

[6] `https://python-control.readthedocs.io/en/0.8.3/index.html`

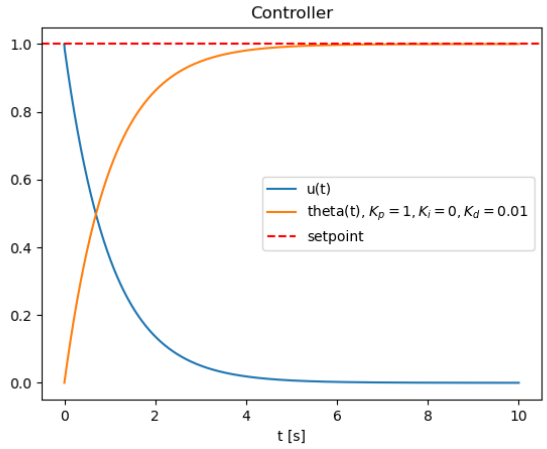

Figure 4. Standalone test of the controller FMU with using linear model for plant $\dot{\theta} = u$, step size = 0.001, setpoint = 1.0

4.1 Setting up the image

A configuration file, referred to as the `Dockerfile`, provides instructions to build the environment on any host machine. An excerpt from the `Dockerfile` used by the robotic arm FMU can be seen in Listing 3. The first line declares that the image for the FMU is assembled ontop a pre-built Python 3.8. image from the Docker container library. The second line invokes the package manager `pip` to install packages required by the model. For simplicity, the three dots represent the dependencies required by the Python backend to communicate with the binary. The third line instructs Docker to copy the `container_bundle` directory into the image. The `container_bundle` contains all files that are needed during runtime, such as the actual model implementation and all user-generated files and dependencies.

```
FROM python:3.8
RUN pip install ... scipy
    roboticstoolbox-python matplotlib
COPY container_bundle resources
...
```

Listing 3. Dockerfile used to assemble the image used by FMU instances.

4.2 Instantiating a Dockerized FMU

The process of creating an instance of a Dockerized FMU is depicted in Figure 5. The steps are as follows: First, the binary will ensure that the image declared in the Dockerfile has been built. If this is not the case, it will automatically invoke the `Dockerfile` to build the image. Next, from the image a container is created. The container has access to all dependencies listed in the `Dockerfile`, such as Python packages that were installed through `pip` and everything inside the `container_bundle`. Note that each instance of an FMU is executed within its own container and removed after use. This ensures that no instances of an FMU share any state or influence each other directly.

DOI 10.3384/ecp21181419

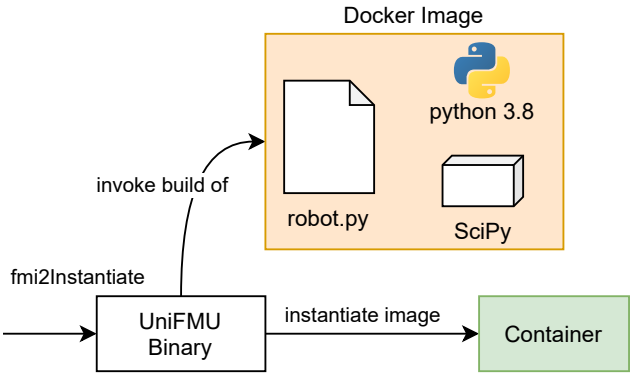

Figure 5. Deployment of a model inside the Docker container.

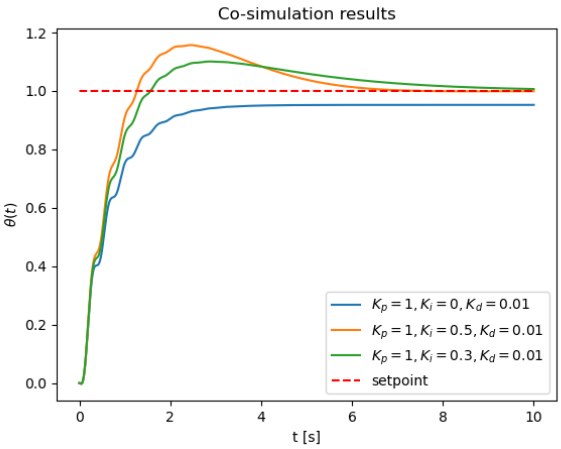

Figure 6. Co-simulation of the controller and the robot with the *set point* $= 1$. Experiments of varying the controller parameters are shown.

5 Results

A co-simulation was configured and run using the two FMUs with the INTO-CPS tool-chain (Larsen et al. 2016). We used a fixed step-size solver with a step-size of 0.001 seconds, set the desired angle to 1 radian and plotted θ as a function of time for various values of K_p, K_i and K_d. The corresponding plot can be seen in Figure 6. Fig. 7 shows the robotic arm at an angle of 1 radian. The controller with $K_i = 0$ exhibits a substantial steady-state error, whereas the ones with an integral term converge within 10 seconds. Besides, it can be seen that controllers with an integral term cause the system to overshoot the setpoint. Tweaking the coefficients of the controller allows us to balance the tendency to overshoot and the steady-state error, such that they meet the requirements of the application. Methods based on heuristics exist for tuning PID controllers, which could be applied to tune the controller for the robotic arm. However, we considered applying these to be beyond the scope of this example use case.

Figure 7. A visualization of the robot during the co-simulation. The measured θ is equal to the *set point* $= 1$ *rad*, when controlled with the PID-controller.

6 Discussion

A central objective of the FMI standard is to facilitate the exchange of models generated by different tools. To do so, FMI requires communication through a C-API, which complicates implementing models in languages that cannot be compiled into a C-compatible binary. UniFMU circumvents this issue by providing a generic C-binary that handles all communication between FMI calls to the FMU and the FMU's actual code. Being able to use high-level programming languages such as Python allows developers to leverage a large ecosystem of scientific libraries and thus implement models quickly and efficiently, especially in contrast to writing everything from scratch. Consider the implementation of the `do_step` method for the robotic arm shown in Listing 1. The ODE is declared and solved in eight lines of code. We believe that this has the potential to simplify co-simulation for more modeling applications and engage more developers.

Another aspect to this approach is that the resulting FMUs can be verified and debugged using the development tools of the FMU's language. For instance, it allowed us to write small test programs for verifying the FMUs before performing the co-simulation of the system. In our experience, the ability to effectively test the individual models greatly reduces the number of issues encountered when integrating the models.

Using FMUs that require runtime dependencies to be handled manually is counter-intuitive to the idea of simple, standardized model exchange. Consequently, in this work we addressed this issue by providing a way to automatically virtualize the runtime environment with all dependencies inside a Docker container rather than requiring the host machine to provide a suitable environment. The way this Dockerization was implemented did not affect UniFMU's precompiled binaries and all changes to the language-specific backends are simply additional con-

figuration options instead of hard dependencies on Docker itself. The latter might be reused in the future to implement remote deployment.

7 Conclusion

Co-simulation is a key research interest. The FMI standard is among the most popular interfaces for model exchange and co-simulation. There are various tools to generate FMI-compliant FMUs. UniFMU is one such tool that allows users to build FMUs from arbitrary code written in any language. We used UniFMU to generate two Python-based FMUs in order to co-simulate a robotic arm and a controller. However, the resulting FMUs required the host machine to provide a Python runtime environment with all dependencies preinstalled, effectively limiting portability and ease of deployment. To address this issue we extended UniFMU using the virtualization tool Docker. With our extension, UniFMU is able to generate FMUs shipped with a `Dockerfile` that automatically builds a runtime environment inside a container. This way the FMUs are almost as portable as compiled FMUs (except for the dependency on Docker) but still support the use of non-compiled languages. Our extension is not limited to Python but can be reused for other languages as well. Besides, the changes we implemented can help in developing a configuration for remote deployment of FMUs in the future.

Acknowledgements

The authors would like to thank Thomas Schwengler for their support with designing the Docker-integration.

The reported research was conducted within the project NextHyb2 (881150) and project DigitalEnergyTwin (873599), which received funding in the framework of ”Stadt der Zukunft” and "Energieforschung", a research and technology program of the Austrian Ministry for Transport, Innovation and Technology (BMVIT).

References

Asghar, Syed Adeel and Sonia Tariq (2010). *Design and Implementation of a User Friendly OpenModelica Graphical Connection Editor*. eng.

Åström, Karl Johan and Richard M Murray (2010). *Feedback Systems: An Introduction for Scientists and Engineers*. In English. ISBN: 978-1-4008-2873-9.

Cremona, Fabio et al. (2019-06). “Hybrid Co-Simulation: It’s about Time”. en. In: *Software & Systems Modeling* 18.3, pp. 1655–1679. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-017-0633-6.

Franklin, Gene F., J. David Powell, and Abbas Emami-Naeini (2020). *Feedback Control of Dynamic Systems*. eng. Eighth edition, global edition. Harlow, United Kingdom: Pearson Education Limited. ISBN: 978-1-292-27452-2.

Gomes, Cláudio et al. (2018-07). “Co-Simulation: A Survey”. en. In: *ACM Computing Surveys* 51.3, pp. 1–33. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3179993.

Hatledal, Lars Ivar et al. (2019-02). “FMU-proxy: A Framework for Distributed Access to Functional Mock-up Units”. In: pp. 79–86. DOI: 10.3384/ecp1915779. URL: https://ep.liu.se/en/conference-article.aspx?series=ecp&issue=157&Article_No=8 (visited on 2021-05-09).

Hinze, Christoph et al. (2018). “Towards Real-Time Capable Simulations with a Containerized Simulation Environment”. In: *2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pp. 1–6. DOI: 10.1109/M2VIP.2018.8600827.

Larsen, Peter Gorm et al. (2016). “Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project”. In: *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pp. 1–6. DOI: 10.1109/CPSData.2016.7496424.

Legaard, Christian Møldrup et al. (2021). “A Universal Mechanism for Implementing Functional Mock-up Units”. In: *11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. SIMULTECH 2021. Virtual Event, to appear.

Modelica Association (2021). *Functional Mock-up Interface for Model Exchange and Co-Simulation*. https://www.fmi-standard.org/downloads.

Nageler, P. et al. (2018-08). “Novel method to simulate large-scale thermal city models”. en. In: *Energy* 157, pp. 633–646. ISSN: 03605442. DOI: 10.1016/j.energy.2018.05.190. URL: https://linkinghub.elsevier.com/retrieve/pii/S0360544218310363 (visited on 2021-05-06).

Pedersen, Nicolai et al. (2017). “Distributed Co-Simulation of Embedded Control Software with Exhaust Gas Recirculation Water Handling System using INTO-CPS:” in: *Proceedings of the 7th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. Madrid, Spain: SCITEPRESS - Science and Technology Publications, pp. 73–82. ISBN: 9789897582653. DOI: 10.5220/0006412700730082. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006412700730082 (visited on 2021-05-06).

Schweiger, G. et al. (2019-09). “An empirical survey on co-simulation: Promising standards, challenges and research needs”. en. In: *Simulation Modelling Practice and Theory* 95, pp. 148–163. ISSN: 1569190X. DOI: 10.1016/j.simpat.2019.05.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S1569190X1930053X (visited on 2021-05-06).

Schweiger, Gerald et al. (2018-12). “District energy systems: Modelling paradigms and general-purpose tools”. en. In: *Energy* 164, pp. 1326–1340. ISSN: 03605442. DOI: 10.1016/j.energy.2018.08.193. URL: https://linkinghub.elsevier.com/retrieve/pii/S0360544218317274 (visited on 2021-05-06).

Virtanen, Pauli et al. (2020). “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17, pp. 261–272. DOI: 10.1038/s41592-019-0686-2.