A preCICE-FMI Runner to Couple FMUs to PDE-Based Simulations

Leonard Willeke[1] David Schneider[1] Benjamin Uekermann[1]

[1]Institute for Parallel and Distributed Systems IPVS, University of Stuttgart, Germany,
benjamin.uekermann@ipvs.uni-stuttgart.de

Abstract

Partitioned simulation or co-simulation allows simulating complex systems by breaking them up into smaller subsystems. The Functional Mock-Up Interface (FMI) enables co-simulation for models based on ODEs and DAEs, but typically not PDEs. However, only PDE-based models are able to accurately simulate physical aspects requiring spatial resolution, such as heat transfer or fluid-structure interaction.

We present a preCICE-FMI runner software to integrate FMUs with the open-source coupling library preCICE. preCICE couples PDE-based simulation programs, such as OpenFOAM or FEniCS, in a black-box fashion to achieve partitioned multi-physics simulations. The runner serves as an importer to execute any FMU and to steer the simulation. Additionally, it calls preCICE to communicate and coordinate with other programs. The software is written in Python and relies on the Python package FMPy. We showcase two example cases for the coupling of FMUs to ODE- and PDE-based models.

Keywords: Functional Mock-Up Interface (FMI), multiphysics, preCICE, coupling, co-simulation, FMPy, OpenFOAM

1 Introduction

The simulation of complex, dynamic systems is an important task in science and engineering. It includes multiphysics simulations and the simulation of cyber-physical systems. There are two paths to achieve such simulations, the monolithic and the partitioned approach. In the monolithic setup, one software includes all the necessary computations to model the different phenomena. Contrary to that, the partitioned approach relies on multiple independent pieces of software. Each of these programs covers a specific aspect of the simulation. The programs are then coupled or co-simulated to achieve the correct outcome. This approach allows splitting complicated systems in smaller, simpler subsystems and re-using them in different scenarios. Examples include climate modeling (Gross et al. 2018), but also engineering applications such as the modeling of wind turbines (Sprague, J. M. Jonkman, and B. J. Jonkman 2015).

The *Functional Mock-Up Interface FMI* (Blochwitz et al. 2011) follows a so-called *framework approach* for co-simulation. The simulation models are implemented as standardized Functional Mock-Up Units (FMUs). FMUs are zip-archives with a pre-defined structure and content. They contain the simulation model as library (*.dll, *.so) and meta data about the model such as documentation or reference results. The coupling is done by an additional program, a so-called importer (see Figure 1). The importer loads and executes the FMUs and implements a co-simulation algorithm to ensure communication and data exchange between the models. This approach works well for simple models composed of ODEs and DAEs, but reaches its limits for more complex PDE models. The computation of PDEs often requires legacy software packages and high-performance computing, which are in general not compatible with the regulations for FMUs.

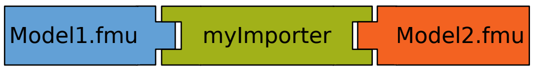

Figure 1. Co-simulation approach of the FMI standard (a framework approach): The standardized FMU models are called and coupled by an importer program. We denote the caller (executable) with a female connector and the callee (library) with a male connector.

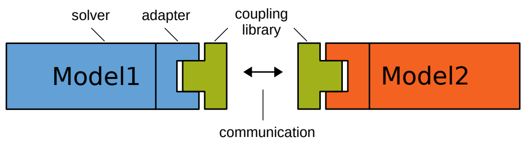

Figure 2. Co-simulation approach of preCICE (a library approach): The simulation programs call preCICE as a library to perform the coupling.

preCICE (Chourdakis et al. 2022), on the other hand, is an open-source coupling library for partitioned multiphysics simulations. It couples PDE-based simulation programs, such as OpenFOAM or FEniCS, in a black-box fashion. preCICE follows a *library approach* for co-simulation: The simulation programs call the coupling (see Figure 2). The programs itself ideally remain untouched and connect to preCICE through an additional software layer called *adapter*. Each simulation program requires a specific adapter. Ready-to-use adapters for many popular simulation programs exist. This setup ensures the re-usability of all components and a decent time-to-solution for new applications.

The main idea of this work is to combine both worlds: to couple FMU models to other simulation programs via preCICE. To this end, a new software component, the preCICE-FMI runner is developed (see Figure 3). It acts as an importer towards the FMU, loading and executing the model. Additionally, the runner calls preCICE to couple the simulation to other programs and steer the simulation. It allows coupling FMU models to any simulation program in the preCICE ecosystem.

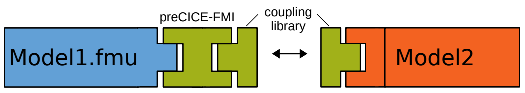

Figure 3. Concept of the preCICE-FMI runner: The new software executes the FMU and calls preCICE to couple the simulation to other programs.

In order to understand how preCICE and the FMI standard can be coupled, Chapter 2 introduces the two existing software components. The chapter continues to explain the concept of the newly developed software, the preCICE-FMI runner, its configuration and functionalities. To showcase the software, Chapter 3 then describes two example cases. First, a partitioned mass-spring oscillator system is used to test the runner against an analytical solution and an alternative Python-based implementation. Second, a FMU model is coupled to a fluid-structure interaction scenario using OpenFOAM as fluid solver and a simple Python script as solid solver. For this scenario, we compare against results from the literature. The final Chapter 4 summarizes the presented ideas and reflects on the impact of the developed software. This paper summarizes and extends the master's thesis of Willeke (2023).

The idea of coupling FMUs to PDE-based programs has already attracted further attention. A recent tool to execute this task is *FMU4FOAM*[1]. It couples FMUs exclusively to the CFD program OpenFOAM. The coupling is realized with a function object, a library that is loaded by OpenFOAM at runtime. The function object embeds a Python interpreter to handle the FMU. FMU4FOAM is configurable on runtime and can be further adapted to specific needs due to its object-oriented structure.

We see the main advantage of the preCICE-FMI runner in its ability to leverage the advanced coupling functionalities of preCICE. The coupling of FMUs is not limited to OpenFOAM, but can be performed with any program in the preCICE ecosystem. Different coupling schemes and topologies are possible. Furthermore, the coupling library implements a toolbox of coupling algorithms to ensure robust and accurate simulations.

2 Software description

Before detailing the concept, functionalities, and limitations of the new runner software, we start this section with describing the existing software components: preCICE and FMI.

2.1 Existing software components

preCICE (Chourdakis et al. 2022) is a widely-used and open-source coupling library for multi-physics simulations. In preCICE terminology, the coupled simulation programs are called *solvers* or *participants*. preCICE implements data mapping and communication between the solvers. The user can choose between explicit and implicit as well as serial and parallel coupling schemes (Gatzhammer 2014). The coupling is not limited to two solvers, but can be easily extended to perform multi-coupling (Bungartz et al. 2015). Acceleration algorithms, such as quasi-Newton methods (Uekermann 2016) are available to stabilize and speed up implicit coupling schemes. Finally, time interpolation allows individual solvers to use different time step sizes. The exact numerical implementation is out of scope here, but can be found in the cited literature. All these coupling configurations are defined in the *precice-config.xml* file, a global file accessed by all participants.

Each solver is connected to preCICE with a specific *adapter*. The adapter allows the solver to access the preCICE API and call the coupling. It can take many forms, depending on the solver itself. For example, the preCICE-OpenFOAM adapter is an OpenFOAM function object, an indepent library, while the preCICE-FEniCS adapter is a Python module. Adapter development is facilitated by the language bindings of preCICE, which are available for C/C++, Python, Fortran, Julia, and Matlab. A more extensive introduction to preCICE is given in Chourdakis et al. (2022).

The *FMI standard* (Blochwitz et al. 2011) defines different types of FMU models. They share a similar interface, but implement different functionalities. *Model Exchange* FMUs hold the model equations and present them to an external solver algorithm. *Co-Simulation* FMUs hold the model equations and the solver algorithms. As such, they can compute the next time step on their own. *Scheduled Execution* FMUs were introduced with FMI v3 (Junghanns et al. 2021) and hold different model partitions, which are accessed by an external scheduler. We only consider the coupling of Co-Simulation FMUs in this work.

The Python library FMPy[2] can be used to load, execute and steer FMU models with Python. We choose to use FMPy over other libraries such as *PyFMI* (Andersson, Akesson, and Führer 2016) and *fmipp* due to its easy installation and usability. An overview of many further FMI tools can be found on the FMI website[3].

2.2 Concept of the preCICE-FMI runner

The *preCICE-FMI runner* is a new piece of software, which connects FMI and preCICE. It uses FMPy to load

[1] `https://dlr-ry.github.io/FMU4FOAM/`

[2] `https://fmpy.readthedocs.io/en/latest/`

[3] `https://fmi-standard.org/tools/`

and execute any FMU and preCICE to couple the simulation. The concept is explained in the simplified code in Figure 4. First, the runner script imports and initializes both FMPy and preCICE (lines 1-9). This includes loading the FMU, setting initial simulation parameters and preparing the communication to other participants. The main loop (lines 11-29) executes the coupled simulation. The program reads data from preCICE (line 16) and writes it to the FMU model (line 18). The model can now compute the next time step (line 20). Afterwards, the program reads the new results (line 22) and writes them to preCICE (line 24). Finally, preCICE advances the simulation as a whole and communicates data with other participant (line 26). Here, preCICE also needs to know how much the FMU model advanced in time to synchronize all participants. This is the main simulation mechanism used for both explicit and implicit coupling. Implicit coupling, however, requires one more feature: the option to repeat a time step. The solver state can be stored (line 14) and reset (line 29) to iterate over a time step. preCICE indicates whether this is necessary or not (lines 12, 27). Finally, preCICE and FMPy are terminated (lines 31-32) to close the communication channels and release the FMU. The software is developed on GitHub[4] and documented in the preCICE user documentation[5].

2.3 Functionalities and limitations of the preCICE-FMI runner

The ideal coupling tool should support the full functionality of both preCICE and FMPy. It should work with any co-simulation FMU and should be configurable at runtime. The software has to be well-documented and include tests to guardrail further development. This section gives insight into the abilities, limitations, and configuration of v0.1 of the preCICE-FMI runner.

The software is available for preCICE v2 and compatible with FMI 1, 2, and 3. It supports explicit and implicit coupling schemes, including acceleration. It is configurable at runtime via two .json files and can be adapted to different FMU models. Time-dependent input signals can be set, output signals are stored as timeseries. The tool is easy to install and comes with a regression test.

A major difference in the coupling of two PDE solvers compared to the coupling of a PDE solver to an FMU is the role of the mesh. For PDE-PDE coupling, the exchanged data is spatial for both solvers and mapped accordingly. However, many FMUs can not deal with spatially resolved data but are designed to receive signal data. To enable the coupling of PDE solvers to such FMUs, we use the mapping capabilities of preCICE: spatial data from one mesh is mapped to a single vertex on another mesh to create signal data and vice versa.

As a result, data exchange is limited to one vertex. Moreover, the exchange of multiple data points and gra-

```
1  import fmpy
2  import precice
3  # FMU Setup
4  fmu = fmpy.fmi3.FMU3Slave(...)
5  fmu.instantiate()
6  # preCICE Setup
7  interface = precice.Interface(...)
8  ...
9  dt = interface.initialize()
10 # main time loop
11 while interface.coupling_ongoing():
12   if interface.action_required(...):
13     # Save state (implicit coupling)
14     state_cp = fmu.getFMUstate()
15   # Get read data from preCICE
16   interface.read_vector_data(...)
17   # Set read_data in FMU
18   fmu.setFloat64(...)
19   # Compute next time step
20   fmu.doStep(t,dt)
21   # Get write_data from FMU
22   write_data = fmu.getFloat64(...)
23   # Send write_data to preCICE
24   interface.write_vector_data(...)
25   # Advance preCICE in time
26   dt = interface.advance(dt)
27   if interface.action_required(...):
28     # Load state (implicit coupling)
29     fmu.setFMUstate(state_cp)
30
31 interface.finalize()
32 fmu.terminate()
```

Figure 4. Concept of the FMI runner: The script utilizes the library FMPy to execute the FMU and calls the preCICE API to couple the simulation. For conciseness, API calls are simplified.

dient information is not yet supported. Also, no internal errors of the FMU model are logged. Finally, the runner software can currently only be executed in serial.

To explain the configuration of the runner, we assume a FMU model `Suspension.fmu`, which contains the model equations for a spring-damper system. It should be coupled via preCICE to receive the variable `force` from another participant, calculate the displacement internally, and send the variable `position` back. The configuration file *fmi-settings.json*, shown in Figure 5, holds the settings for FMPy. The dictionary `simulation_params` (lines 2-9) is used to choose the FMU model and set the read and write variables. A CSV file can be set to store results. The dictionaries `model_params` and `initial_conditions` (lines 10-16) allow setting model parameters before the simulation start. Time-dependent input signals are set in `input_signals` (line 17-23). In this case, the variable `damping_coeff` is initialized with value 0.0 and

[4] `https://github.com/precice/fmi-runner`
[5] `https://precice.org/tooling-fmi-runner.html`

increases to value 5.0 at $t = 2.0$.

To execute the coupling, the runner needs some information on how to interact with preCICE. The file *precice-settings.json* (Figure 6) points the runner to the global configuration file of preCICE shared with all participants. The following entries define which participant the runner is, which mesh it owns, and which preCICE variables are accessed.

```
{
"simulation_params": {
  "fmu_file": "../Suspension.fmu",
  "fmu_read_data": ["force"],
  "fmu_write_data":["position"],
  "fmu_instance": "suspension_1",
  "output_file": "./output.csv",
  "output": ["position"]
  },
"model_params": {
  "apply_filter":         true,
  "spring_coeff":         65.0
},
"initial_conditions": {
  ...
},
"input_signals": {
  "names":["time", "damping_coef"],
  "data": [
          [0.0, 0.0],
          [2.0, 5.0]
  ]
  }
}
```

Figure 5. Example for fmi-settings.json

```
{
"coupling_params": {
  "config_file": "../config.xml",
  "participant": "Suspension",
  "mesh_name": "Suspension-Mesh",
  "read_data": {"name": "Force"},
  "write_data": {"name": "Position"}
  }
}
```

Figure 6. Example for precice-settings.json

3 Example cases

Two example cases show the functionality of the preCICE-FMI runner. First, the simulation of a partitioned mass-spring oscillator system demonstrates the coupling of two ODE systems. The obtained results are compared to a numerical simulation in Python and an analytical solution. Second, the fluid-structure interaction of a moving cylinder is adapted to include a FMU model for controlling the movement and compared to reference results from literature. The case files are available for reproduction.[6]

3.1 Partitioned mass-spring oscillator system

We assume an ideal mass-spring system (Schüller et al. 2022) as shown in Figure 8. Three springs k_1, k_{12}, and k_2 connect two masses m_1 and m_2 with each other and two fixed walls. The variables u_1 and u_2 denote the positions of the masses. To create a partitioned system, spring k_{12} is cut in the middle. The resulting subsystems exchange interface forces F_1 and F_2. The system is adaptable to perform different kinds of oscillations. We follow Schüller et al. (2022) and set the spring stiffness to $k_1 = k_2 = 4\pi^2 N/m$ and $k_{12} = 16\pi^2 N/m$, while the two masses are set to $m_1 = m_2 = 1kg$. Additionally, the initial conditions are chosen as

$$\begin{aligned} u_1(0) &= 1 \quad & \dot{u}_1(0) &= 0 \\ u_2(0) &= 0 \quad & \dot{u}_2(0) &= 0 \end{aligned}$$

This setup has the analytical solution depicted in Figure 7. The two masses oscillate with a period of $T_P = 1s$.

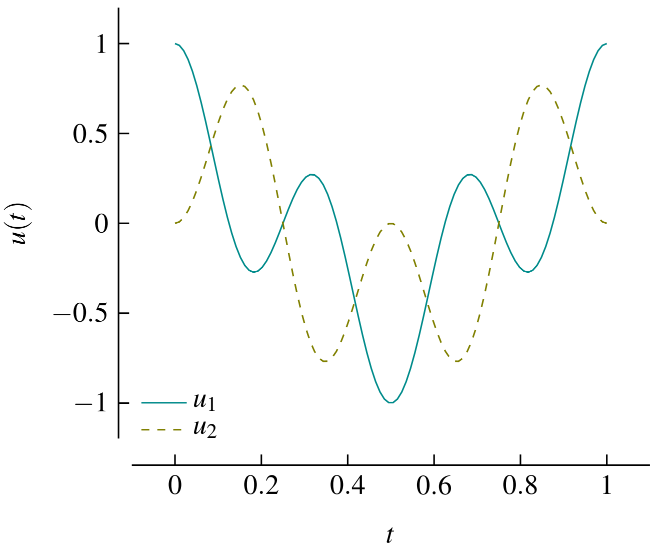

Figure 7. Analytical solution of the mass-spring system (Schüller et al. 2022). The plot shows the displacements u_1 and u_2 for the initial conditions $u_1(0) = 1, \dot{u}_1 = 0$ and $u_2(0) = 0, \dot{u}_2 = 0$.

We use this testcase to compare an FMU-based implementation executed with the preCICE-FMI runner and an existing implementation in Python (Schüller et al. 2022). Both implementations can be coupled to themselves or to one another. We end up with four possible combinations: runner-runner, Python-Python, runner-Python, and Python-runner. Moreover, we compare against the analytical solution.

[6] https://doi.org/10.18419/darus-3549

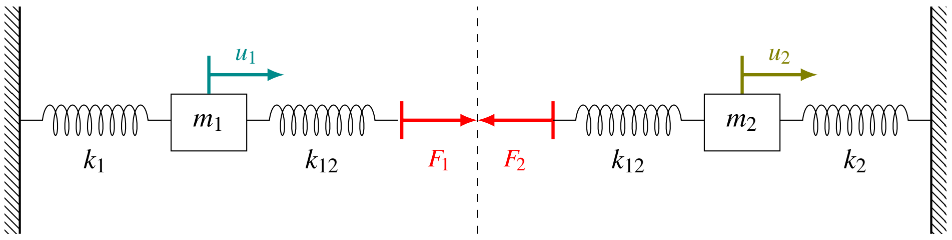

Figure 8. Mass-spring system: The springs k_1, k_{12} and k_2 connect the two masses m_1 and m_2 which have the positions u_1 and u_2. The middle spring k_{12} is cut in half to create a partitioned setup with two subsystems that exchange interface forces F_1 and F_2.

Both implementations use the Newmark-β method (Newmark 1959) with $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$ for time integration. A serial-implicit coupling with Aitken acceleration is used. The simulation time is set to $T = 5s$ with a time step of $\Delta t = 0.005s$. A more extended description is given in Willeke (2023).

The results in Figure 9 show the trajectory of m_1. No differences are visible between all four possible combinations. Furthermore, all combinations match the analytical solution well. For a more accurate comparison, the maximum norm $||e||_\infty$ between analytical solution and numeric result for u_1 is calculated over all time steps. The maximum serves as a worst-case approximation. The calculated error $||e(u_1)||_\infty \approx 3.48 \times 10^{-2}$ is identical for the FMU and the Python implementation up to a precision of 10^{-5}.

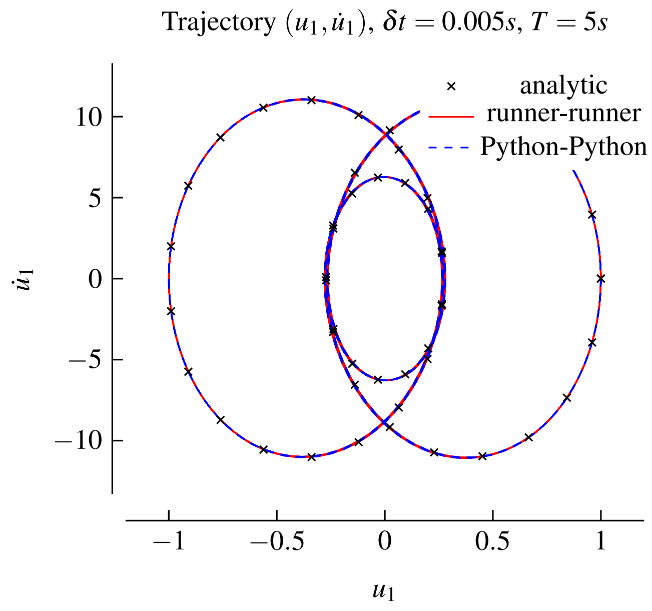

Figure 9. Comparison of different implementations for the mass-spring oscillator system. Velocity $\dot{u}_1$ is plotted over position u_1 to show the trajectory of m_1. The results from a coupling of both the runner and the Python solver to itself have no visible differences and track the analytical solution well. The results for the cross-combinations runner-Python and Python-runner show no visual difference and are omitted for simplicity.

3.2 Flow around a moving cylinder

As a second example, consider the flow in a channel around a cylinder as shown in Figure 11. The cylinder with diameter D and mass m is not fixed in its position y. Instead, the cylinder is mounted upon a spring-damper system with spring stiffness k and damper coefficient d. The flow with velocity v_0 induces vortex shedding behind the cylinder. This leads to varying lift forces and results in an oscillation of the cylinder position y. This setup has been used as a test case for numerical simulations (Placzek, Sigrist, and Hamdouni 2009) and is especially interesting because experimental reference data is available (Anagnostopoulus and Bearman 1992). The experimentalists report lock-in effects for the cylinder oscillation for Reynolds numbers between $104 < Re < 126$ with the highest excitation at the lower end. For our simulation, we chose a Reynolds number of $Re = 108.83$. The remaining system parameters are set to $m = 0.03575kg$, $d = 0.0043N/s$ and $k = 69.48N/m$. All are in accordance with the referenced literature. We further adapt the case to be able to move the root point of the spring u (Sicklinger 2014). Now, the spring force acting on the cylinder can be actively controlled by adjusting u.

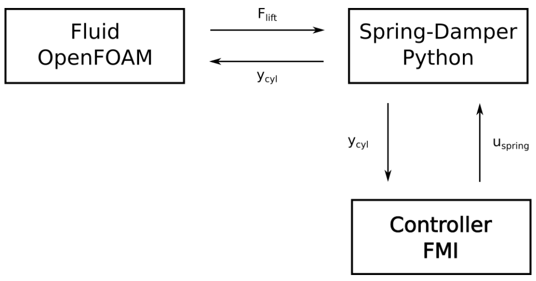

Figure 10. Coupling topology for the flow around a moving cylinder example: Three participants are coupled in two bi-coupling schemes. The fluid participant and the spring-damper participant exchange lift force F and cylinder displacement y. The spring-damper participant and the controller exchange displacement y and spring root displacement u.

The goal of this setup is to couple a controller FMU to a PDE-based simulation. The FMU holds the equations of a Proportional-Integrative-Derivative (PID) controller (Ang, Chong, and Li 2005). It reads the displacement of the cylinder y and tries to minimize it by setting the root point of the spring u.

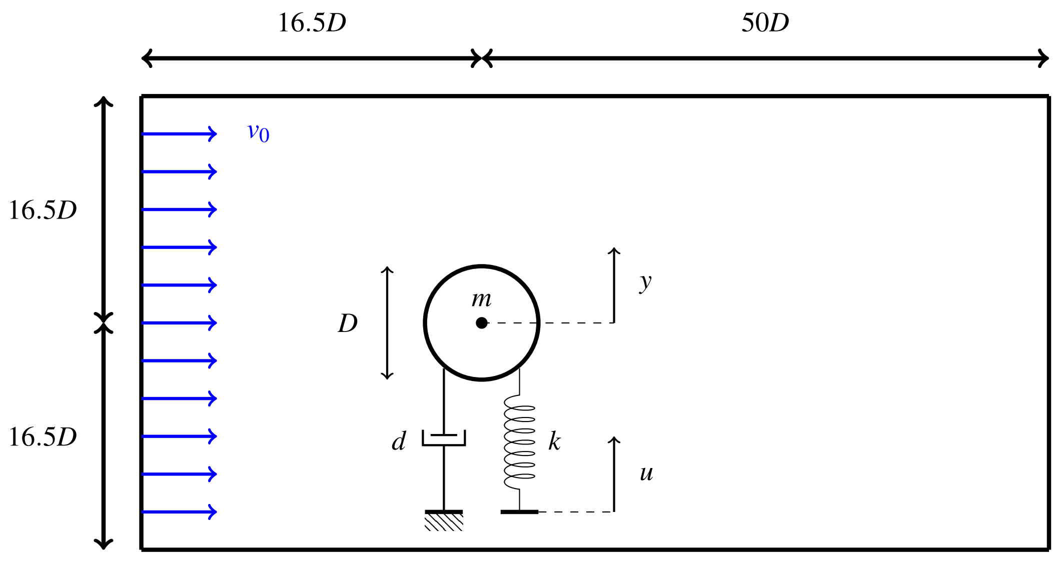

Figure 11. Case setup for flow around a moving cylinder: The object is mounted upon a spring-damper system, which allows it to move in y-direction. The root point of the spring u can be moved to vary the force acting on the cylinder.

The spring-damper system itself is calculated in a separate Python program, while the fluid flow is computed with the CFD program OpenFOAM. The resulting coupling topology is shown in Figure 10. Two explicit bi-coupling schemes are combined. A fitting mapping configuration ensures the transition from the spatial domain in OpenFOAM to the signal domain in the FMU.

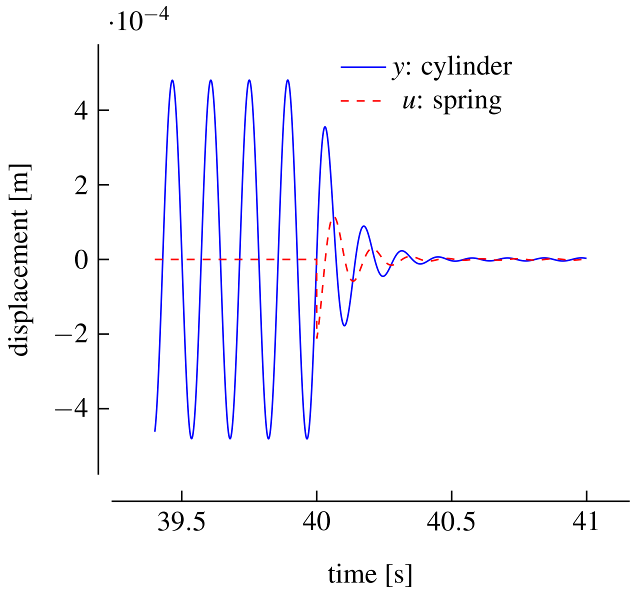

Figure 12. Cylinder displacement with activation of the PID controller: The cylinder oscillates in a stable state until the controller is activated at $T = 40s$. This reduces the displacement y by orders of magnitude.

Figure 12 shows the simulation results obtained with a time step of $\Delta t = 0.0001s$. First, the controller is deactivated until the cylinder has reached a state of stable oscillation. The displacement has an amplitude of $\hat{y} = 0.48mm$, which is close to the reference amplitude of $\hat{y}_{ref} = 0.6mm$. The differences may be attributed to the explicit solver in the spring-damper system and the explicit coupling scheme, both of which are implicit in the reference simulation. The PID controller is activated at $T = 40s$. The control gains are set to $K_P = 0.02$, $K_I = 0.02$ and $K_D = 0.01$ to ensure a robust transient behaviour (Sicklinger 2014). The cylinder displacement is reduced by orders of magnitude with a fast transition phase.

4 Conclusions

We presented the preCICE-FMI runner, a new software to couple FMU models to PDEs. The software loads and executes FMU models and calls preCICE to execute the coupling. It is written in Python to leverage the Python package FMPy and the preCICE Python bindings. The runner software is configured with two settings files, enabling different simulation scenarios. An easy, standard installation process lowers the entry barrier for new users. The preCICE-FMI runner is compatible with co-simulation FMUs of FMI versions 1, 2, and 3 and preCICE version 2. It supports explicit and implicit coupling via preCICE, as well as the use of acceleration methods. Time-dependent input signals can be set for the FMU model during simulation. Output signals from the FMU are stored for post-processing.

Two example cases introduced the functionalities of the runner software. The partitioned simulation of a mass-spring oscillator system showed good agreement with another numerical solver and an analytical solution. The simulation of the flow around a moving cylinder was coupled to a FMU-based control algorithm and showed qualitatively meaningful results.

The new software is focused on providing a general coupling of co-simulation FMUs to PDE-based solvers. It enables plug-and-play coupling to many popular solvers thanks to the preCICE library approach. The software is documented on *precice.org* and the presented test cases are available for reproduction. Some limitations, such as the lack of error logging or full mesh exchange remain.

FMI is a widely used industry standard for the co-simulation of cyber-physical systems. preCICE, on the other hand, has a growing user base in academia and industry focused on high-fidelity multi-physics applications. The preCICE-FMI runner connects these two communities. With our work, we hope to spark a discussion about the specific needs of both communities to guide further developments.

Acknowledgements

Many thanks to Mosayeb Shams from Heriot-Watt University, UK, who shared his expertise on OpenFOAM with us. Great thanks to Ishaan Deshai, University of Stuttgart, for his manifold support throughout the project.
Funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2075 – 390740016. We acknowledge the support by the Stuttgart Center for Simulation Science (SimTech).

References

Anagnostopoulus, P. and P.W. Bearman (1992). "Response Characteristics of a vortex-excited cylinder at low Reynolds numbers". In: *Journal of Fluids and Structures* 6.1, pp. 39–50. DOI: 10.1016/0889-9746(92)90054-7.

Andersson, Christian, Johan Akesson, and Claus Führer (2016). "PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface". In: Technical Report in Mathematical Sciences LUTFNA-5008-2016.2.

Ang, Kiam Heong, G. Chong, and Yun Li (2005). "PID control system analysis, design, and technology". In: *IEEE Transactions on Control Systems Technology* 13.4, pp. 559–563. DOI: 10.1109/TCST.2005.847331.

Blochwitz, T. et al. (2011). "The functional mockup interface for tool independent exchange of simulation models". In: Technical University; Dresden; Germany.

Bungartz, Hans-Joachim et al. (2015). "A plug-and-play coupling approach for parallel multi-field simulations". In: *Computational Mechanics* 55, pp. 1119–1129. DOI: 10.1007/s00466-014-1113-2.

Chourdakis, G et al. (2022). "preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]". In: *Open Research Europe* 2.51. DOI: 10.12688/openreseurope.14445.2.

Gatzhammer, B. (2014). "Efficient and Flexible Partitioned Simulation of Fluid-Structure Interactions". PhD thesis. Technical University of Munich, pp. 1–183.

Gross, Markus et al. (2018). "Physics-dynamics coupling in weather, climate, and Earth system models: Challenges and recent progress". English (US). In: *Monthly Weather Review* 146.11, pp. 3505–3544. DOI: 10.1175/MWR-D-17-0345.1.

Junghanns, A. et al. (2021). "The Functional Mock-up Interface 3.0 - New Features Enabling New Applications". In: DOI: 10.3384/ecp2118117.

Newmark, N. M. (1959). "A method of computation for structural dynamics". In: *J. Eng. Mech. Div.-ASCE* 85.3, pp. 67–94. DOI: 10.1061/JMCEA3.0000098.

Placzek, A., J.F. Sigrist, and A. Hamdouni (2009). "Numerical Simulation of an oscillating cylinder in a cross-flow at low Reynolds number: Forced and free oscillations". In: *Computers and Fluids* 38.1, pp. 80–100. DOI: 10.1016/j.compfluid.2008.01.007.

Schüller, Valentina et al. (2022-07). "A Simple Test Case for Convergence Order in Time and Energy Conservation of Black-Box Coupling Schemes". In: DOI: 10.23967/wccm-apcom.2022.038.

Sicklinger, S.A. (2014). "Stabilized Co-Simulation of Coupled Problems including Fields and Signals". PhD thesis. Technical University of Munich, pp. 126–135. DOI: 10.13140/2.1.1103.7762.

Sprague, M. A., J. M. Jonkman, and B. J. Jonkman (2015). "FAST Modular Framework for Wind Turbine Simulation: New Algorithms and Numerical Examples". In: *SciTech 2015: 33rd Wind Energy Symposium*.

Uekermann, B. (2016). "Partitioned Fluid-Structure Interaction on Massively Parallel Systems". PhD thesis. Technical University of Munich, pp. 62–70. DOI: 10.14459/2016md1320661.

Willeke, Leonard (2023). *A preCICE-FMI Runner to Couple Controller Models to PDEs*. Master thesis. DOI: 10.18419/opus-13130.