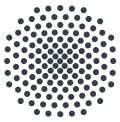


University of Stuttgart
Germany

Sustainable Electrical Power Supply
(Master's Program)
Universität Stuttgart

Master's Thesis

A preCICE-FMI Runner to Couple Controller Models to PDEs

Leonard Willeke

University of Stuttgart
Germany

Sustainable Electrical Power Supply (Master's Program)

Universität Stuttgart

Master's Thesis

A preCICE-FMI Runner to Couple Controller Models to PDEs

Author:	Leonard Willeke
1st examiner:	Univ.-Prof. Dr. Po Wen Cheng
2nd examiner:	Univ.-Prof. Dr. Stefan Tenbohlen
Assistant advisor:	Jun.-Prof. Dr. Benjamin Uekermann
Submission Date:	April 20th, 2023

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen verwendet habe. Alle Stellen dieser Arbeit, die dem Wortlaut, dem Sinn oder der Argumentation nach anderen Werken entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht.

20. April 2023 Leonard Willeke

Acknowledgments

I would like to thank Prof. Stefan Tenbohlen and Prof. Po Wen Cheng for the opportunity to write my Thesis in a different faculty section.
Great thanks to Benjamin Uekermann for his supervision and trust. I enjoyed the work on this Thesis a lot, and it was a great experience to be part of preCICE.
Many thanks to the wider preCICE team, who made me feel welcome during the Coding Days and the Workshop. Special thanks to David Schneider, Ishaan Desai and Benjamin Rodenberg for their support.
Lastly, my deep gratitude to my parents who supported me throughout my study years.

Abstract

Partitioned simulation or co-simulation allows to simulate complex systems by breaking them into smaller, independent subsystems. The Functional Mock-Up Interface FMI enables co-simulation by defining a framework for simulation models. Models adhering to the standard interface (FMUs) are executed and coupled by an importer. This framework approach works well for models based on ODEs and DAEs but reaches its limits for models based on PDEs. Such models require sophisticated, legacy software packages not compatible with the FMI standard. However, only PDE-based models are able to accurately simulate many physical aspects important in engineering like heat transfer or Fluid-Structure interactions. A possible solution to this problem is the open-source coupling library preCICE. preCICE couples PDE-based simulation programs in a black-box fashion to achieve partitioned multi-physics simulations. The coupling of the FMI standard to preCICE would allow the co-simulation of FMI models with the more than 20 simulation programs in the preCICE ecosystem.

This thesis is focused on the development of a preCICE-FMI Runner software to couple FMUs with preCICE. The Runner serves as an importer to execute the FMU and steer the simulation. Additionally, it calls the preCICE library to communicate and coordinate with other solvers. The scope is not to develop a general Runner software, but to couple FMUs that contain control algorithms with PDE-based models as a first step. The software is written in Python and relies on the Python package FMPy as well as the preCICE Python bindings.

Two test cases show the functionality of the preCICE-FMI Runner. The coupling of ODE-based models with FMUs matches the results of a pure Python implementation with an accuracy of 10^{-4}. The coupling of a PDE-based model to a controller FMU proofs the working principle, although the results could not be tested against other implementations. The scope of the implemented abilities restricts the possible simulation scenarios, but does not prohibit a general use for coupling scenarios beyond control applications.

Zusammenfassung

Partitionierte Simulation oder Co-Simulation erlaubt es, die Simulation eines komplexen Systems mit kleineren, unabhängigen Subsystemen durchzuführen. Das Functional Mock-Up Interface FMI ermöglicht Co-Simulation durch die Definition eines Frameworks für Simulationsmodelle. Modelle, die diesem Interface-Standard folgen (FMUs), werden von einem Importer ausgeführt und gekoppelt. Dieser Framework Ansatz zur Co-Simulation funktioniert gut für Simulationsmodelle die auf ODEs und DAEs beruhen, allerdings nicht für Modelle mit PDEs. Diese Modelle werden oftmals durch fortgeschrittene Legacy - Softwarepakete berechnet, die nicht an die normierten Bedingungen des FMI Standards angepasst werden können. Viele physikalische Aspekte wie Hitzeübertragung oder Fluid-Struktur Interaktionen, die eine große Bedeutung für Ingenieursanwendungen haben, können aber nur mit PDE-basierten Modellen präzise berechnet werden. Eine mögliche Lösung für dieses Problem ist die Open-Source Kopplungsbibliothek preCICE. preCICE koppelt PDE-basierte Simulationsmodelle, um Multiphysiksimulationen durchzuführen. Die Kopplung des FMI Standards mit preCICE würde die Co-Simulation von FMUs mit den mehr als 20 Simulationsprogrammen erlauben, die preCICE derzeit verbindet.

Diese Thesis beschäftigt sich mit der Entwicklung einer "preCICE-FMI Runner" Software, um FMUs mit preCICE zu koppeln. Der Runner agiert als Importer für die FMU, um deren Simulation auszuführen und zu steuern. Gleichzeitig ruft er die preCICE Bibliothek auf, um mit anderen Simulationsprogrammen zu kommunizieren und die Simulation zu koordinieren. Der Fokus dieser Arbeit liegt nicht darauf, eine allgemeine Runner Software zu entwickeln. Stattdessen soll eine FMU, die ein Regelungsmodell implementiert, mit einer PDE-basierten Simulation gekoppelt werden. Die Software wird in Python geschrieben und nutzt das Python-Paket FMPy sowie die preCICE Python Bindings.
Zwei Testfälle zeigen die Funktionalität des preCICE-FMI Runners. Die Kopplung von ODE-basierten Modellen mit FMUs zeigt Übereinstimmung mit einer reinen Python - Implementierung bis zu einer Genauigkeit von 10^{-4}. Die Kopplung eines PDE-basierten Modelles mit einer Controller-FMU bezeugt die prinzipielle Machbarkeit dieser Kopplung. Die Resultate konnten allerdings nicht mit anderen Implementierungen verglichen werden. Der Umfang der umgesetzten Funktionalitäten begrenzt die möglichen Simulationsanwendungen, ermöglicht aber dennoch die Nutzung für unterschiedlichste Kopplungsszenarien über Regelungsanwendungen hinaus.

Contents

List of Figures

List of Tables

1 Introduction

Simulation has become the third pillar of scientific discovery, complementing and advancing experiment and theory. An important subgroup are multi-physics simulation, which form the basis for many branches of engineering and the sciences. There are two paths to achieve multi-physics simulations, the monolithic and the partitioned approach. In the monolithic setup, one software includes all the necessary computations to simulate the physical phenomena. Contrary to that, the partitioned approach relies on multiple independent pieces of software. Each of these solvers covers a specific aspect of the simulation. The pieces are then coupled to achieve the correct outcome. This setup allows to re-use the single software components again in different scenarios. The partitioned approach is especially appealing when dealing with large and complicated systems, which can be split into individual subsystems. Examples include climate modeling [3], but also engineering applications such as the modeling of wind turbines [4].

Partitioned simulations can be realized with two different concepts. The Framework approach is based on developing a Master Algorithm which executes the participating simulation programs as needed. The coupled simulation programs will be called participants or solvers throughout this thesis. The coupling software creates a frame to which the participants have to adhere. The Library approach turns this concept around: The participants are executed individually and call the coupling software as a library. This concept is minimally invasive and requests very few changes to the solvers, which do not have to adhere to a special coupling framework. It allows the flexible use and re-use of software components and a decent time-to-solution.

The open-source software preCICE [1] enables partitioned multi-physics simulations using the library approach. Besides the data exchange, preCICE takes care of data mapping, implements different acceleration schemes and can perform time interpolation. These methods are implemented in a black-box fasion to be used with any appropriate solver regardless of the simulation case. To couple a simulation software with preCICE, an additional software called adapter is necessary. The adapter is the middle piece between the preCICE library and the solver.

The Functional Mock-Up Interface (FMI) [5] is an interface standard for the exchange of simulation models. It is developed by the Modelica Association, an industry consortium with members from areas like aerospace and automotive. Due to its great use within these industries, it is currently the de-facto industry standard for the exchange of simulation

models. Many tools, both commercial and open-source, support the use of the standard.

The goal of this thesis is to couple FMI models to other simulation programs with preCICE. To this end, a Runner software [1] was developed. The Runner connects FMI models and preCICE (see Fig. 1.1), similar to the preCICE Adapters that have been written for simulation programs such as OpenFOAM. A general Runner could be used to couple any simulation program in the preCICE ecosystem with FMI models.
This would open up many interesting use cases. preCICE could be used as a Master Algorithm to couple multiple FMI models, exploiting the advanced co-simulation capabilities of the coupling library. The models could be generated from any tool supporting the FMI standard, including programs commonly used in engineering such as Simulink and Dymola. The key idea behind the FMI standard is interoperability, which means opening up to the standard enables coupling to very different applications. But probably most interesting is the prospect of coupling FMI models with simulation programs for advanced physical processes in 3D, such as OpenFOAM or FEniCS, to capture different phenomena of cyber-physical systems.

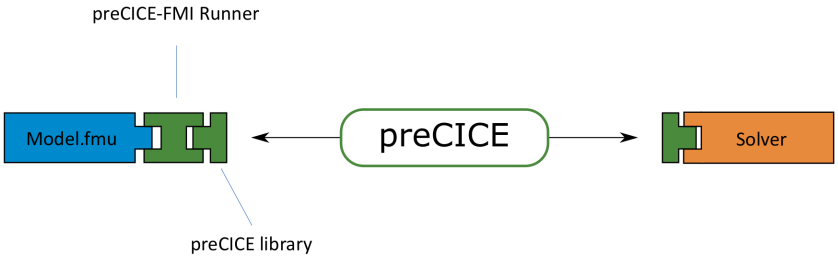


Figure 1.1: A sketch of the preCICE-FMI Runner setup. The Runner enables co-simulation of FMI models by connecting them to preCICE.

However, to develop a general Runner that captures all the different simulation cases is complex and beyond the scope of this thesis. This thesis is focused on developing a Runner with a specific application in mind, which can serve as a starting point for generalization. This application is the coupling of a control algorithm written in the FMI standard with a solver based on Partial Differential Equations (PDE). An example would be a Fluid-Structure Interaction with the controller optimizing a system parameter. Many applications of this type can be found in the engineering sciences. For example, in the field of Wind Energy one might want to simulate the air flow around a wind turbine rotor blade and control the pitch angle of the blade to reduce fatigue or maximize lift. This idea is especially interesting because many control algorithms in Wind Energy are currently being developed in Simulink, a tool that supports the FMI standard.

[1] https://github.com/precice/fmi-runner (accessed: 2023-04-13)

In order to understand how preCICE and the FMI standard can be coupled, chapters 2 and 3 give an overview of the two software components. I then continue to describe the newly developed Runner software in chapter 4. The concept behind the Runner is explained, as well as it's interface, abilities and limitations. Chapter 5 takes the reader through two test cases: First, the partitioned simulation of a mass-spring oscillator system compares the results of the Runner with those of a Python implementation, giving confidence that the Runner works correctly. The second case considers the coupling of the Runner to the simulation of a Fluid-Structure Interaction. Finally, Chapter 6 provides a summary and outlines further possible developments. I conclude if the Runner can be used in a generalized way or if it remains a software for specialized applications.

2 Functional Mock-up Interface

The Functional Mock-up Interface (FMI) [6] is a framework for model exchange and co-simulation. The aim of this interface standard is to "simplify the creation, storage, exchange and (re-) use of dynamic system models"[1]. Models created after the FMI standard can be run black-box in a black-box fashion, independent of their content. The origins of the framework stem back to the MODELISAR project[2] in 2008, an industry cooperation which set out to improve the design of systems and embedded software in vehicles. Since then, the standard has evolved to become the de-facto industry standard for co-simulation. Today, the development of the Functional Mock-up Interface is organized within the Modelica Association[3].

As a framework, the FMI standard sets out to define rules and regulations to standardize the way simulation models are structured, simulated, and coupled. It is not a software in itself, but regulates how a class of software tools is being developed and used. The FMI standard is implemented via so called Funcional Mock-up Units (FMU). These units serve as structured containers for simulation models and will be examined in section 2.1. FMUs are the building blocks for the co-simulation setup, but they need another external tool to run and steer the simulation (see section 2.2). Over time, many software tools (commercial and open-source) have been adapted to be able to simulate FMUs or export their internal models to the FMU format. A selected list of tools is presented in section 2.3.

[1] https://fmi-standard.org/about/ (accessed: 2023-04-13)
[2] https://itea4.org/project/modelisar.html (accessed: 2023-04-13)
[3] https://modelica.org/ (accessed: 2023-04-13)

2.1 Functional Mock-up Units

Within the FMI standard, Functional Mock-Up Units (FMU) serve the purpose of encapsulating simulation models, code, and functionalities. Because these FMUs are standardized, they make interoperability possible: All FMUs share a similar structure and many common functions. However, three different types of FMUs have been developed to satisfy different demands. This section will explore the common structure and types of FMUs, before moving on to their simulation in the next section.

Structure

On the highest level, a FMU is a zip archive that contains specific files and binaries (see Fig. 2.1). The zip folder is unpacked before the simulation gets access to the internal files. The FMU folder always contains *binaries* that encode the model functions. Depending on the operating system, this code will come in the form of a shared object (*.so) or dynamically linked library (*.dll or *.dylib). Crucially, the FMU does not contain any executables. Instead, it always depends on an "importer" program to call and run the binaries (see 2.2). Another mandatory item is the *ModelDescription* file. This XML file contains a high level overview of the simulation model. It introduces the parameters used within the model, gives insight into the implementation by providing capability flags, and defines a default simulation case.
Next to these mandatory items, more information can be added to the zip archive voluntarily to simplify the use of the FMU. For example, a detailed documentation of the model formulas or reference data may be included. Sometimes, it is also desirable to share the source code alongside the binaries. However, in many cases the source code is omitted to protect intellectual property. It is still possible to use the binaries without constraints as the interface is standardized and independent of the model content.

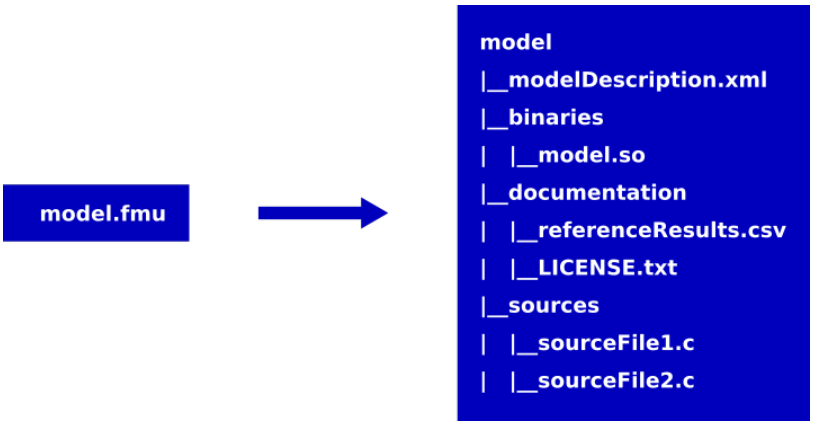


Figure 2.1: A FMU model is a *zip* file containing specific files and binaries.

Types

Three different interface types are currently available for the Functional Mock-up Units. They build on a base of common functions and conventions, but extend it for different use cases. Depending on the type, the interaction with the importer is different.

A **Co-Simulation** FMU implements the model functions and a the solver algorithm. Therefore, it has the ability to compute the next timestep. The importer only tells the FMU when to do so, but doesn't implement a solver on its own. Fig. 2.2 sketches the interaction of FMU and importer.

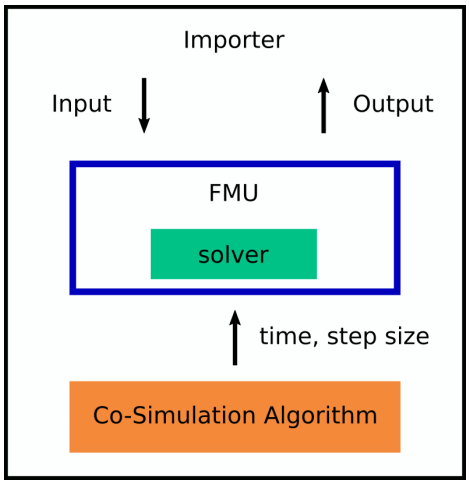


Figure 2.2: Interaction of the FMU type "Co-Simulation" with the importer. The model includes the solver algorithm, while the importer steers the Co-Simulation

A **Model Exchange** FMU implements the model functions, but not a solver algorithm. That means, it can expose the simulation model to another program but not solve the model equations. Crucially, it cannot advance the time in a numerical simulation. Instead, this task is delegated to the importer.

A **Scheduled Execution** FMU differs from the two types above. It is split into several model partitions, which can be executed by a scheduler provided by the importer. Again, the FMU implements the solver algorithm, but with a different timing concept. The motivation here lies in the connection to real hardware with individual timing schemes, where events may occur unpredictably. A more detailed explanation can be found in [5].

2.2 Simulation

To run a simulation with an FMU model, we need an importer. The importer is an external program that loads, unpacks, and calls the FMU model. It uses the standardized FMI-API and can therefore work with any FMU. The importer can be a rather sophisticated tool, such as Mathwork Simulink, or a simple Python script. The latter is possible due to Python packages that have been developed for this task. The library used for the Runner development is FMPy[4]. An exemplary script for the simulation flow with FMPy is given in Fig. 2.3.

I want to highlight some of the functionalities that FMPy provides and which are important for the Runner:

- `fmpy.extract("model.fmu")` [line 6]: The *zip* folder of the FMU is extracted and stored at a local directory. This makes the internal files and the binary *model.so* accessible.
- `fmpy.fmi3.FMU3Slave(...)` [lines 8-11]: Returns an `fmu` object which is used to interact with the FMU.
- `fmu.instantiate()` [line 13]: Instantiates the model. Now model variables and internal functions can be accessed.
- `fmu.setFloat64([vr_parameter], [value])` [line 17]: Sets the value of an internal parameter. Here, a parameter of type `Float64` is set, but the same is possible with other data types. The parameter is not accessed by its name, but by a value reference number `vr_parameter`. Every internal parameter has a value reference, but not all parameters are accessible. The value references and names as well as accessibility can be read from the ModelDescription file.
- `fmu.getFMUstate()` [line 24]: Returns a snapshot of the full state of the model. All parameters and internal states are stored.
- `fmu.doStep(time, step_size)` [line 26]: Computes the next time step. This function calls the `doStep()` method internal to the FMU. Note that this is only possible with co-simulation FMUs, because they include the solver algorithm. The results of the computation are stored internally in the FMU
- `fmu.getFloat64([vr_result])` [line 28]: Reads the value of the result with value reference number `vr_result` from the FMU.
- `fmu.setFMUstate(state_checkpoint)` [line 31]: Sets the full state of the model to an earlier point in time as stored in `state_checkpoint`. This is important for co-simulation, where a master algorithm decides whether `modelConverged` is true or not.

[4] https://fmpy.readthedocs.io/en/latest/ (accessed: 2023-04-13)

```python
import fmpy

# read information about model
model_description = fmpy.read_model_description("model.fmu")
# extract the FMU
unzipdir = fmpy.extract("model.fmu")
# Simulation object is defined
fmu = fmpy.fmi3.FMU3Slave(
        guid=model_description.guid,
        unzipDirectory=unzipdir,
        modelIdentifier=model_description.coSimulation.modelIdentifier)
# model is initialized
fmu.instantiate()
fmu.enterInitializationMode(startTime=start_time)
fmu.exitInitializationMode()
# Set model parameter
fmu.setFloat64([vr_parameter], [value])
time = start_time
results = []

# main time loop
while time < stop_time:
        # save current model state
        state_checkpoint = fmu.getFMUstate()
        # model computes next time step
        fmu.doStep(time, step_size)
        # get results
        output = fmu.getFloat64([vr_result])
        # Decide whether to repeat the timestep or not
        if modelConverged == False:
                fmu.setFMUstate(state_checkpoint)
        elif modelConverged == True:
                time += step_size
                results.append((time, output))

fmu.terminate()
fmu.freeInstance()
```

Figure 2.3: An example script for the simulation of a FMU model with the Python package FMPy

- `fmu.terminate()` [line 36]: Terminates the simulation.

During co-simulation, multiple FMUs are coupled to simulate a bigger system. Hence, the functionalities presented for data exchange between FMU and importer as well as the ability to repeat a time step when convergence criteria are not met become important. These actions have to be coordinated by a Master Algorithm. One can imagine an importer that loads multiple FMUs, executes them and handles the data exchange and numerical stability (see Fig. 2.4).

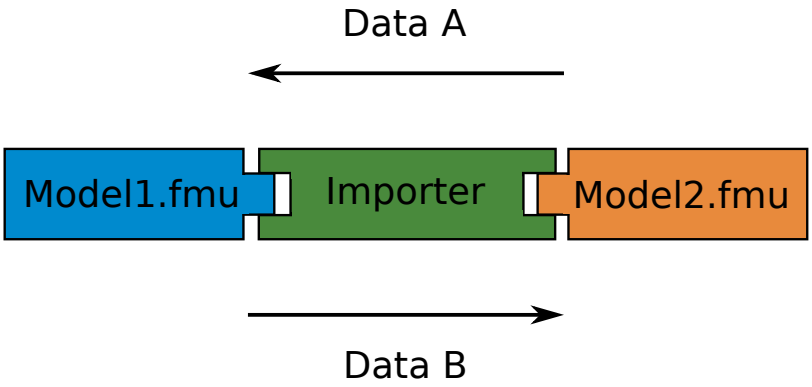


Figure 2.4: The importer can include a master algorithm to execute and steer multiple FMUs.

2.3 Tools

A great benefit of the FMI standard is its interoperability with many established software tools. Over 170 tools so far have adopted the standard and allow for the import of FMU models, the export of their internal models to the FMU format or act as master algorithms. The following list of tools is incomplete, but can serve as a starting point. A complete list of tools supporting the FMI standard can be found on the FMI website[5].

PyFMI is a python package for the simulation of coupled dynamic models with the Functional Mock-up Interface [7]. Like FMPy, it was written to enable the simulation of FMUs with Python. Simulation can be done for single and multiple FMUs, as PyFMI includes a master algorithm. The software is closely linked to another Python package called **Assimulo** [8]. Assimulo implements a variety of solver algorithms for ordinary differential equations and unifies them under a high-level interface. For example, Euler's and Runge Kutta methods are implemented. From the SUNDIALS suite ([9], [10]), it wraps the popular solvers CVODE and IDA. This is very useful when working with Model Exchange

[5] https://fmi-standard.org/tools/ (accessed: 2023-04-14)

FMUs which don't contain a solver algorithm, but depend on the importer to solve the exposed equations. Here lies the strength of PyFMI in combination with Assimulo: One provides the problem in terms of mathematical equations, one provides the numerical solver.

Next to these standalone packages for the simulation of FMUs, many existing softwares have been adapted to include the FMI standard in their ecosystem. One such software is the **OpenModelica Editor**, developed in the broader OpenModelica Environment [11]. The editor is a graphical user interface for the Modelica programming language. Simulation models for a variety of engineering and scientific problems can be created and edited by "drag and drop" elements. Also, a simulation environment with debugging and post-processing facilities is given, which makes the editor a great tool for fast prototyping. FMUs can be integrated into such simulations or internal models can be exported as FMUs. If a terminal-based solution is needed, the **OpenModelica Shell** can be used. However, the FMU models exported from Modelica tools lack some functionalities crucial to this thesis like the ability to reset the state of the FMU to an earlier point in time. The same is true for FMUs created with **Mathwork Simulink**. This is unfortunate, as most control applications in the engineering sciences depend heavily on Simulink for development. However, at this point, FMU models created with Simulink can only be used in explicit coupling schemes.

An alternative to the aforementioned tools to create FMU models is the direct compilation of C-code. To this end, the **Reference-FMUs**[6] of the Modelica Association form a good basis for anyone willing to try it out. This way is the most flexible and gives the most insight into the implementation as needed for some development processes.

[6] https://github.com/modelica/Reference-FMUs (accessed: 2023-04-13)

3 Coupling Library preCICE

The open-source coupling library preCICE [1] enables partitioned multi-physics simulations. It makes it possible to couple different simulation softwares to simulate multi-physics scenarios, for example fluid-structure interactions. Due to the black-box fashion of the coupling, applications range from aerospace engineering to biomechanics. preCICE is developed at the Technical University of Munich and the University of Stuttgart. Great attention is given to the usability, code sustainability and community interaction of preCICE, an important success factor for open source software libraries [12].

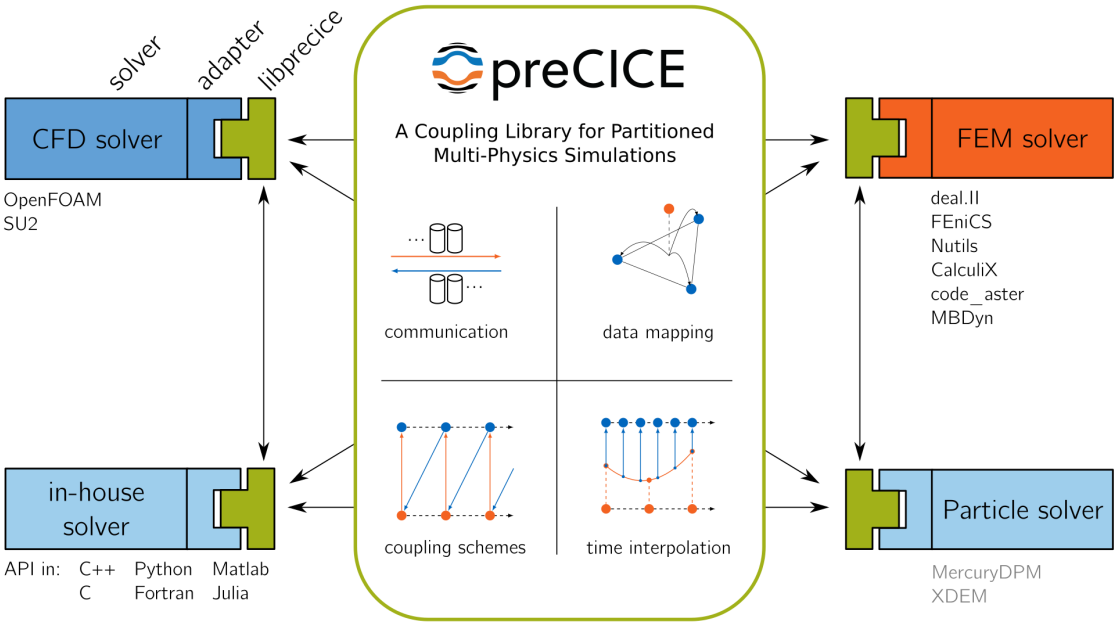


Figure 3.1: An overview of preCICE [1]. The coupling library takes care of the communication, data mapping, coupling schemes and time interpolation during the simulation. To interact with the solvers, an additional piece of software called adapter is necessary.

Figure 3.1 shows an overview of how preCICE achieves co-simulation. The coupling library is called by the participants at every time step. It takes care of communication,

data mapping, coupling schemes and time interpolation. To interact with the solvers, an additional piece of software called adapter is necessary. This software allows the solver to remain untouched and acts as an intermediate between preCICE and the solver. Currently, adapters are available for OpenFOAM, FEniCS and many more simulation tools.

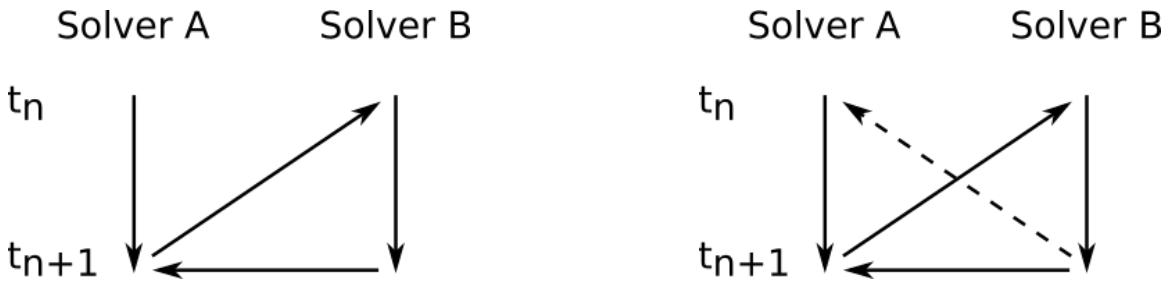
Within the adapter, preCICE is called with a specific API. Although the preCICE core is written in C++, bindings are available for eg. Matlab and Python to simplify the development of adapters. Figure 3.2 shows an exemplary script for coupling a solver in Python. The functions shown here don't cover the whole possibilities that preCICE offers but were selected to introduce the most important functionalities for this project.

- `precice.Interface("solver_name", "config.xml", ...)` [line 3]: Returns a `SolverInterface` object for the solver "solver_name" which points to the defined coupling interface. This interface object is from there on used to interact with preCICE. The coupling is defined in the file "config.xml".
- `interface.initialize()` [line 16]: Initializes preCICE, which includes the setup of parallel communication and the exchange of meshes between coupling partners.
- `interface.read_block_vector_data(...)` [line 23]: Reads vector data from the coupling partner from one or more mesh vertices.
- `interface.write_block_vector_data(...)` [line 30]: Sends the write data to the coupling partner.
- `interface.advance(...)` [line 32]: Advances preCICE after the solver has computed one timestep.

preCICE implements many methods, the description of which is out of scope for this thesis. Nevertheless, I want to give a brief glance on the available coupling schemes. Figure 3.3 shows the serial coupling schemes between two solvers, where solver A is executed first and the results are then communicated to solver B. In an explicit coupling schemes, this happens once every time step. Explicit coupling schemes are easy to implement, but can lead to numerical instabilities during the simulation. Such instabilities can be avoided with implicit coupling schemes. Implicit schemes iterate over a single time step until a stability criterium is met. The iteration process can be enhanced by acceleration methods, which act on the results exchanged between the solvers and can lead to faster convergence. Beyond this basic coupling scenario, preCICE implements parallel coupling methods and is able to couple multiple solvers.

```python
solver = Solver() # Coupled participant (not part of preCICE)
# Define the coupling interface
interface = precice.Interface("solver_name", "config.xml", rank, size)
mesh_id = interface.get_mesh_id("coupling_mesh")
dimensions = interface.get_dimensions()

vertices = np.zeros((num_vertices, dimensions))
read_data = np.zeros((num_vertices, dimensions))
write_data = np.zeros((num_vertices, dimensions))
vertex_ids = interface.set_mesh_vertices(mesh_id, vertices)
read_data_id = interface.get_data_id("read_variable", mesh_id)
write_data_id = interface.get_data_id("write_variable", mesh_id)

dt = solver.dt # solver timestep size
precice_dt = float() # maximum precice timestep size
precice_dt = interface.initialize()

while not interface.is_coupling_ongoing(): # time loop
  if interface.is_action_required(...):
    # Save checkpoint for implicit coupling schemes, if needed
    interface.mark_action_fulfilled(...)

  interface.read_block_vector_data(read_data_id, vertex_ids)
  solver.set_read_data(read_data)
  dt = begin_time_step() # compute adaptive time step
  dt = min(precice_dt, dt)
  solver.compute_time_step(dt)
  solver.compute_write_data(write_data)
  write_data = solver.get_write_data()
  interface.write_block_vector_data(write_data_id, vertex_ids, write_data)

  precice_dt = interface.advance(dt) # Advance time in precice
  solver.end_time_step() # update variables, increment time, ...

  if interface.is_action_required(...):
    # Reload checkpoint for implicit coupling schemes, if needed
    interface.mark_action_fulfilled(...)
# Free data structures and close communication channels
interface.finalize()
solver.destroy()
```

Figure 3.2: An exemplary coupling script to showcase the preCICE API (v2.5)



(a) serial-explicit (b) serial-implicit

Figure 3.3: Explicit and implicit serial coupling scheme

4 Development of the Runner software

4.1 Concept

FMUs can be used to package simulation models in a standardized way with the specific goal to enable co-simulation. They are not executables on their own, but require a so-called importer to call them. During co-simulation, the importer has to fullfill two critical roles: exchange data between FMUs and implement a master algorithm to sustain a stable simulation. Writing such an importer in Python is greatly facilitated by the software package FMPy.
The coupling library preCICE is a software specialized on the coupling of simulation programs. To this end, it implements different coupling schemes and provides data mapping and acceleration methods. Importantly, preCICE follows the library approach to partitioned simulations. This means preCICE is called by the participants to advance the simulation instead of preCICE calling the participants.

The idea behind the Runner is to provide the connection between preCICE and FMI. In this scenario, preCICE steers the simulation. It decides which actions have to be taken by each participant to reach a stable and fast-converging simulation. The Runner acts as the importer for the FMU, executing the FMU model and steering it according to the commands from preCICE. This functionality separates the Runner from other software components used to connect simulation programs to preCICE, the adapters. These adapters are usually a function or module that communicates between preCICE and a full scale simulation program. The adapters don't execute the simulation programs, but transfer data and execution commands. The Runner, on the other hand, executes the FMU which implements the simulation model. Simultaneously, it communicates to preCICE. Therefore it is a mixture between importer and adapter.

In its simplest form, the Runner is a merge of the two exemplary scripts shown in Fig. 2.3 and 3.2. This thought experiment is shown in Fig. 4.1. All commands used have already been introduced.

But how would an ideal Runner software look like? What functionalities would we wish for? In the following, I want to give this some consideration to motivate my implementations.
The Runner should be as general as possible. We want to use it with any FMU, regardless of the FMI version or the model equations it implements. Furthermore, the Runner software should be easy to install and use. To check local installations and guardrail further

```python
import fmpy
import precice
# FMU Setup
fmu = fmpy.fmi3.FMU3Slave(...)
fmu.instantiate()
# preCICE Setup
interface = precice.Interface("solver_name", "config.xml", rank, size)
mesh_id = interface.get_mesh_id("coupling_mesh")
dimensions = interface.get_dimensions()
vertices = np.zeros((num_vertices, dimensions)) # Set coordinates for vertices
read_data = np.zeros((num_vertices, dimensions))
write_data = np.zeros((num_vertices, dimensions))
vertex_ids = interface.set_mesh_vertices(mesh_id, vertices)
read_data_id = interface.get_data_id("read_variable", mesh_id)
write_data_id = interface.get_data_id("write_variable", mesh_id)

dt = interface.initialize() # preCICE timestep size

while not interface.is_coupling_ongoing(): # time loop
  if interface.is_action_required(precice.action_write_iteration_checkpoint()):
    state_checkpoint = fmu.getFMUstate() # Save FMU state
    interface.mark_action_fulfilled(precice.action_write_iteration_checkpoint())
  # Get read_data from preCICE
  interface.read_vector_data(read_data_id, vertex_ids)
  # Advance FMU in time
  fmu.setFloat64([vr_read], [read_data]) # Set read_data in FMU
  fmu.doStep(t,dt) # Compute next time step with FMU
  write_data = fmu.getFloat64([vr_write_data]) # Get write_data from FMU
  # Send write_data to preCICE
  interface.write_vector_data(write_data_id, vertex_ids, write_data)
  # Advance preCICE in time
  dt = interface.advance(dt)

  if interface.is_action_required(precice.action_read_iteration_checkpoint()):
    fmu.setFMUstate(state_checkpoint) # Reload FMU state if necessary
    interface.mark_action_fulfilled(precice.action_read_iteration_checkpoint())

interface.finalize() # Frees data structures and closes communication channels
fmu.terminate()
```

Figure 4.1: The concept behind the Runner software: The functionalities of FMPy and preCICE are combined in one script to handle both the simulation of the FMU model and the coupling with other simulation programs.

changes, a test suite should be developed alongside the program itself.
Besides these user-focused requirements, an ideal Runner would support the full functionality of preCICE:

- *Explicit and implicit coupling schemes*: Implicit coupling schemes require solvers that are able to repeat the calculation of a timestep. For this, the state of the solver needs to be stored and retrieved. The FMI standard introduced respective functionalities for FMUs with version 2. For models with the FMI version 1, this is not possible.
- *Exchange of full meshes*: In the past, the equations implemented in FMUs have been focused on Ordinary Differential Equations (ODE) and not on more complex PDEs. The data formats available in FMUs were restricted to scalars, which made discretization in space unfeasible. This is about to change, as FMI version 3 implements vectors and matrices as new data types. Ideally, one could communicate a whole mesh from preCICE to a FMU. However, for any legacy model, the exchange of single vertices is enough.
- *Acceleration*: preCICE implements various acceleration schemes to reduce the convergence time for implicit coupling schemes. They play an important role in conducting simulations in an acceptable timeframe.
- *Data logging*: This is not typically associated with preCICE, as the solvers are responsible for storing the simulation results. However, as the Runner is a mixture of importer and adapter, it also has to implement data logging for the FMU results.
- *Error logging*: Error logs are an invaluable tool to debug the simulation. The FMU can assert internal errors and communicate them to the Runner, which should export them to log files.

In the following I want to give insight into developed software by describing its abilities and limitations. Additionally, the configuration options are discussed to show how the functionalities of the Runner can be accessed. Use cases with possible simulation setups can be found in chapter 5.

4.2 Abilities and Limitations of the implemented software

The Runner is **compatible with the FMI versions 1, 2 and 3**. As the standard evolved, function calls changed and new features were added. FMI 1 lacks some of the functionalities which are now available in FMI 3. This is most apparent for implicit coupling, which is not possible with FMI 1 models. They lack the ability to store and reload the state of the model, which was introduced with FMI 2 and is necessary for implicit coupling schemes. Even for models adhering to FMI 2 and 3, many models do not implement this feature. The ability to redo a timestep is communicated to the importer by a flag. This makes it

possible to **implement both explicit and implicit coupling schemes** with the Runner and give responding error messages when the configuration of the coupling scheme does not match the abilities of the FMU. As part of the implicit coupling schemes, **acceleration** can be used with FMUs that have an implicit internal solver. No special adaptions were necessary for this, as the acceleration is done in preCICE and affects the solver only in the data that is communicated to it. The data exchanged with the Runner is currently restricted to **one vertex, which can hold a scalar or a vector**. During the simulation, the Runner can **set time-dependent input signals** and **store output data from the FMU**. The results are stored as timeseries in a .csv file for easy post-processing. All of these functionalities, including the coupling parameters with preCICE, are controlled by a **generic interface** accessed by two .json input files. The setting files and the different configuration options are discussed in detail later on. Besides these characteristics directly involved with the simulation process, the Runner also includes a **simple regression test**. Consideration was given to create an **easy installation process** with good documentation using standard tools like the Python Packaging Index pip.

However, there are some limitations to what could be achieved throughout this thesis. Most remarkably, the Runner is currently **not able to exchange multiple vertices**. Although this seems like a harsh restriction, it is not: FMUs are designed and built to run Ordinary Differential Equations. These systems of equations are discretized in time, but not in space like Partial Differential Equations. Therefore, the models implemented in FMUs tend to be simpler and don't require the exchange of big meshes. One sign of the design choice towards ODEs was the limitation to scalar variables within the FMU, which hold up until the introduction of FMI 3 in 2021. With FMI 3, array variables are now available. However, the Runner is currently not adapted for this. Instead, the assumption is that **all FMU variables are scalars.** This does not mean tough that the data exchanged via preCICE is limited to a scalar. Instead, the vector data sent from another participant can be transferred to scalar variables in the FMU. They form a vector through naming conventions. Although not very elegant, this approach has been used by the FMI community for many years before the introduction of FMI 3 and is therefore well established. Another limitation of the Runner is its **lack of the logging of FMU internal errors**. The FMUs can report errors to the importer, and FMPy has functionalities to catch and log these errors, which makes this feature very feasible for future developers. The fact that the software is currently **only available for preCICE v2** falls in the same category. The release of preCICE v3 is scheduled for the summer of 2023 and brings some breaking changes in the API calls. However, it remains to be seen how fast the preCICE community will switch to the new version. Lastly, the Runner **can only be executed in serial.** In general, FMUs can be executed in parallel, which has been used by other tools before [13]. The restriction to serial execution may be a disadvantage when solely coupling FMUs with each other via preCICE. But as soon as the computationally rather light-weight FMUs are coupled with more resource intensive programs like OpenFOAM, the Runner is not likely to be the bottleneck to slow down simulation time.

4.3 Configuration

The configuration of the Runner is organized through two setting files, *precice-settings.json* and *fmi-settings.json*, which are handed to the Runner on execution. They allow to set up the simulation case with the FMU. The files are structured in dictionaries, which are briefly discussed to explain the configuration options.

The file *precice-settings.json* file shown exemplary in Fig. 4.2 contains a dictionary called **coupling_params**. Most entries here consider the standard procedure of setting up a coupling scenario with preCICE: The coupling library needs to know which participant the Runner is going to be, where to find the preCICE configuration file and what mesh belongs to this participant. The last two entries, *Write-Data* and *Read-Data*, concern the data exchanged via preCICE. Data name and type have to be given as they are defined in *precice-config.xml*. Hence, the data type can be either `scalar` or `vector`. This extra classification is necessary to set the correct preCICE API calls within the Runner. The rank and size of the process as well as the number of vertices have been set to set to $rank = 0$, $size = 1$ and $n_{vertices} = 1$. These values reflect the current limitations of the Runner.

```
{
"coupling_params": {
        "participant_name": "Runner",
        "config_file_name": "../precice-config.xml",
        "mesh_name": "Runner-Mesh",
        "write_data": {"name": "Write-Data", "type": "vector"},
        "read_data": {"name": "Read-Data", "type": "vector"}
        }
}
```

Figure 4.2: An exemplary preCICE settings file for the Runner

Let's turn to the second input file *fmi-settings.json* shown in Fig. 4.3. As the name predicts, all entries concern the interaction of the Runner with the FMU. We start of with **simulation_params**. It is a mandatory entry for the settings file. It contains fields to choose the FMU model for simulation, the directory and name of the output file and which internal data from the FMU should be stored there. The fields *fmu_read_data_names* and *fmu_write_data_names* connect the FMU with preCICE. They decide which FMU variables receive the read data from other participants and which variables are retrieved to write data to preCICE. When vector data is exchanged via preCICE, as is the case here, the number of entries in the lists for *fmu_read_data_names* and *fmu_write_data_names* has to match the dimensions of the vector data. The scalar variables in the lists are used to store the vector elements. If, for example, a 2D vector (x, y) is read, the variables are set to $read\_data\_1 = x$

and $read\_data\_2 = y$.
The next entries of the settings file are optional and the dictionaries can be left blank or removed completely. The **model_params** and **initial_conditions** dictionaries are used to set parameters before the start of the simulation. The key has to match the variable name. The variables are set during the initialization phase of the FMU. Doing so allows setting variables which can not be changed during the simulation due to restrictions enforced by the FMU model, such as variables marked as "fixed" in the model description XML. Finally, the dictionary **input_signals** can be used to create changing input signals over the course of the simulation. The first entry of the list always has to be the time at which the new values should be set, followed by the values for the respective variables.
Crucially, all entries concerning variable names in *fmi-settings.json* must match the names of the internal variables of the FMU. The variable names can be checked in the model description XML file of the respective FMU model. Furthermore it is assumed that all variables are scalars, a current limitation of the Runner.

```
{
"simulation_params": {
        "fmu_file_name": "../model.fmu",
        "output_file_name": "./output.csv",
        "output": ["output_1", "output_2"],
        "fmu_read_data_names": ["read_data_1", "read_data_2"],
        "fmu_write_data_names":["write_data_1", "write_data_2"],
        "fmu_instance_name": "model_1"
        },
"model_params": {
        "param_double":     0.0,
        "param_bool":       true,
        "param_string":     "simulation_flag"
        },
"initial_conditions": {
        "variable_1":   0.0,
        "variable_2":   0.0
        },
"input_signals": {
        "names":["time",    "variable_3",   "variable_4"],
        "data": [
                [0.0,       0.0,            0.0],
                [1.0,       2.0,            0.0],
                [3.0,       2.0,            4.0]
                ]
        }
}
```

Figure 4.3: An exemplary FMI settings file for the Runner

5 Test Cases

This chapter introduces two test cases to showcase and test the abilities of the Runner. First, I couple two ODE systems implemented with FMUs in section 5.1. The results are compared with a coupling of Python solvers. Second, I discuss the coupling to a Fluid-Structure Interaction in 5.2. Both test cases are archived for reproduction in the Data Repository of the University of Stuttgart DaRUS.[1]

5.1 Partitioned Mass-Spring Oscillator

Mathematical description

The first test case is made up of an ideal mass-spring oscillator with three springs and two masses. The setup is motivated by [14] and has been adapted for preCICE by [2]. This section closely follows the structure there to introduce the case.
Figure 5.1 shows the system, which we will simulate in a partitioned fashion. This setup has some interesting benefits. It is simple enough for a fast implementation. Because the resulting partitions are described with ODEs, they can be easily computed within FMU models. At the same time, the problem is complex enough to apply different integration methods and coupling schemes (see [2]). This allows the testing of many functionalities of preCICE that the Runner has to implement, like implicit coupling and acceleration.

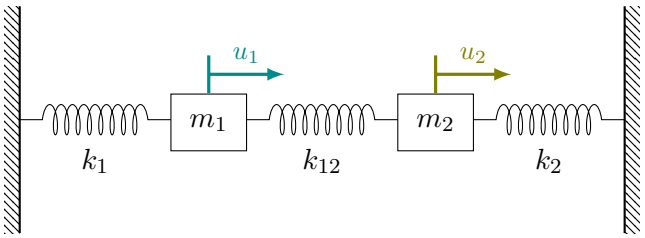


Figure 5.1: Monolithic mass-spring system [2]

[1] https://doi.org/10.18419/darus-3408

The system can be described by the following initial value problem (IVP):

$$\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \begin{pmatrix} \ddot{u}_1 \\ \ddot{u}_2 \end{pmatrix} + \begin{bmatrix} k_1 + k_{12} & -k_{12} \\ -k_{12} & k_2 + k_{12} \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = 0 \tag{5.1}$$

The masses of the two bodies are denoted as m_1 and m_2, while their positions are written as u_1 and u_2 with the respective accelerations $\ddot{u}_1$ and $\ddot{u}_2$. The three spring elements without damping are characterized by their stiffnesses k_1, k_{12} and k_2. The matrix notation of this case reads

$$M\ddot{\mathbf{u}} + K\mathbf{u} = 0$$

with the initial conditions for the positions and the velocities of the masses being

$$\mathbf{u}(0) = \mathbf{u_0}, \qquad \dot{\mathbf{u}}(0) = \mathbf{v_0}$$

The analytical solution of the system can be computed and used to study the numerical results. The total energy of the mass-spring oscillator consists only of kinetic and potential energy and is given as

$$E = \frac{1}{2}\dot{\mathbf{u}}^T M\dot{\mathbf{u}} + \frac{1}{2}\mathbf{u}^T K\mathbf{u}$$

One can show that

$$\frac{dE}{dt} = 0$$

In the partitioned case, the integration methods and coupling schemes vary in their ability to accurately compute this energy conservation. To create a suitable test case, we consider the system with the parameters shown in Table 5.1.

k_1, k_2	$4\pi^2$	$[N/m]$	Spring stiffness
k_{12}	$16\pi^2$	$[N/m]$	Spring stiffness
m_1, m_2	1	$[kg]$	Mass

Table 5.1: Parameters for the mass-spring system

Furthermore, the initial conditions for displacement and velocity are chosen as:

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

With the system defined, we can now calculate the analytical solution for this special case (see Eq. 5.2). It has a period of $T = 1s$ and is plotted in Figure 5.2.

$$u_1(t) = \frac{1}{2}\left(\cos\left(2\pi t\right) + \cos\left(6\pi t\right)\right) \qquad u_2(t) = \frac{1}{2}\left(\cos\left(2\pi t\right) - \cos\left(6\pi t\right)\right) \tag{5.2}$$

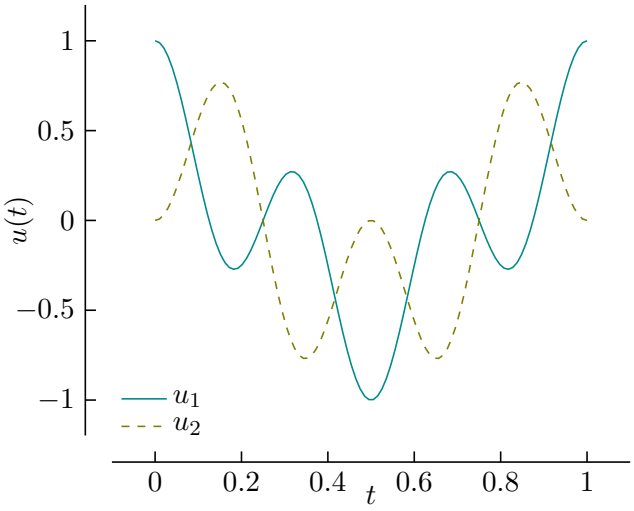


Figure 5.2: Analytical solution of the mass-spring system [2]. The plot shows the displacements u_1 and u_2 for the initial conditions $u_1(0) = 1, \dot{u}_1 = 0$ and $u_2(0) = 0, \dot{u}_2 = 0$

With the monolithic case adequately described, we can now move on to the partitioned case (Fig. 5.3). The system is cut in half at the middle spring k_{12} with the interface forces F_1 and F_2. This way of decomposing the domain into two smaller boundary value problems results in a Schwarz-type coupling: The domains overlap, because both subsystems depend not only on the interface forces, but also on the stiffness of the middle spring k_{12} (see Eq. 5.4). Hence, the resulting subsystems are not fully independent from each other. The benefit of this method is the symmetry of the subsystems which simplifies the implementation.

$$m_1\ddot{u}_1 = -\left(k_1 + k_{12}\right)u_1 + F_2(t) \tag{5.3}$$
$$m_2\ddot{u}_2 = -\left(k_2 + k_{12}\right)u_2 + F_1(t) \tag{5.4}$$

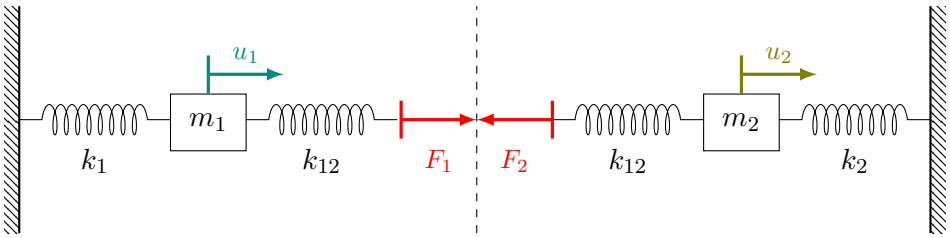


Figure 5.3: Partitioned mass-spring system

Simulation

The mass-spring oscillator system in its partitioned form yields a simulation case that is complex enough to test different coupling schemes, yet easy enough to be fully implemented with FMU models. The Runner is used to load, execute and couple the FMU model computing one of the partitions.

The motivation for this section is to show that the Runner works correctly for this simple case. To this end, the Runner is coupled in different scenarios and the results are compared with simulation results obtained with another solver and the analytical solution. The second solver type used is the Python implementation by [2]. All four coupling options of the two solvers [(*R* - *R*), (*R* - *P*), (*P* - *R*), (*P* - *P*)] are simulated. The methods used for integration and coupling are the same for both solvers. The results for u_1 of all cases are used for comparison. First, they are plotted and compared visually. Second, the maximum norm $\|e\|_\infty$ between analytic and numeric result is calculated over all time steps. The maximum serves as a worst-case approximation for the error.

[2] includes a detailed study about the effect the coupling method and especially the integration method has on the energy conservation for the mass-spring oscillator. However, we will limit ourselves to one integration method, namely the Newmark-β method [15], and a serial-implicit coupling scheme with Aitken acceleration. Acceleration is not necessary to reach acceptable simulation times, but was used for testing. The focus of this example is to show that these methods work with the newly developed Runner software, not to conduct numerical experiments.
The simulations were performed with a timestep size of $\delta t = 0.005s$ for a total of $T = 5s$. A relative tolerance of $r_{tol} = 10^{-6}$ for the difference of computed results in consecutive iterations was chosen as convergence criterion. Both cases were run with the following parameters for the Newmark-β method:

$$\beta = \frac{1}{4} \quad \gamma = \frac{1}{2}$$

Figure 5.4 visualizes the results. The trajectory of m_1 is plotted for the 5 simulated cycles. This visualization makes it easier to spot deviations than a timeseries. No differences are

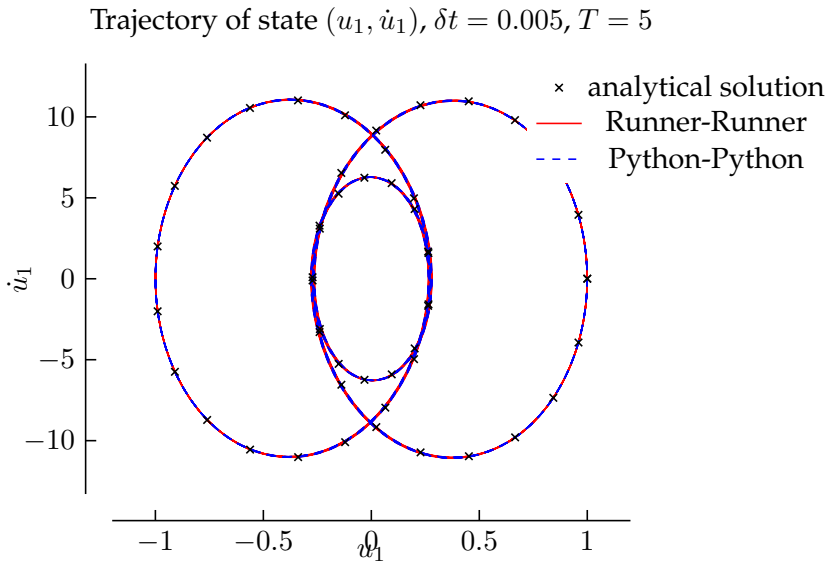


Figure 5.4: Numerical results for serial-implicit coupling with Aitken acceleration. Velocity $\dot{u}_1$ is plotted over position u_1 to show the trajectory of m_1. The results from a coupling of Runner and Python solver to itself have no visible differences and track the analytical solution well. The results for the cross-combinations Runner-Python and Python-Runner are similar (not shown).

visible between the coupling of Runner-Runner and Python-Python solvers. Also, both implementations match the analytical solution well. The same is true for the other two solver combinations, namely Runner-Python and Python-Runner, which are omitted from the plot.
The calculated error confirms this first impression partially. The error $\|e(u_1)\|_\infty \approx 3.48 \times 10^{-2}$ for u_1 is equal for all implementations until a precision of 10^{-5}. For the error $\|e(u_2)\|_\infty \approx 3.43 \times 10^{-2}$, the first divergence occurs at a precision of 10^{-4}. A plausible explanation for this could be the data type conversion that takes place at every time step between the Runner and the FMU model. This may be taken into consideration for applications that rely on a very exact computation.

5.2 Flow around an oscillating cylinder

Setup

We simulate the laminar flow around a cylinder. The cylinder is not fixed, but mounted upon a spring damper system (see Fig. 5.5). During the simulation, vortex shedding occurs behind the cylinder and causes the cylinder to move up and down. To turn this setup into a control case, the root point of the spring can be moved to vary the spring force acting on the cylinder. A control algorithm sets the spring displacement to counteract the cylinder displacement.

Experimental results are available for a similar setup with a fixed spring root [16]. It has received some attention as a test case for numerical simulations, eg. in [17]. [18] provides a comparison of different numerical simulation studies and expanded the setup with a movable spring root and a controller. His explanation of the case forms the basis for the following introduction.

The cylinder is set to $D = 1.6 \times 10^{-3} m$ in accordance with [16]. The experimentalists report lock-in effects for Reynolds Numbers between $104 < Re < 126$ with the highest excitations at the lower limit. The inflow speed v_0 is therefore adjusted to reach a Reynolds number of $Re = 108.83$.

Control algorithm

The proportionate-integral-derivative controller (PID) is commonly used in many industrial applications for its simplicity and robustness. The continuous control law reads

$$u(t) = K_P e(t) + K_I \int_0^t e(t)\, dt + K_D \frac{\delta e(t)}{\delta t}$$

with $e(t)$ as the error between desired and measured state. The parameters K_P, K_I and K_D are used to tune the controller for the specific use case and can be found analytically or heuristically. The discretized form of the PID law with timestep size h can be stated as

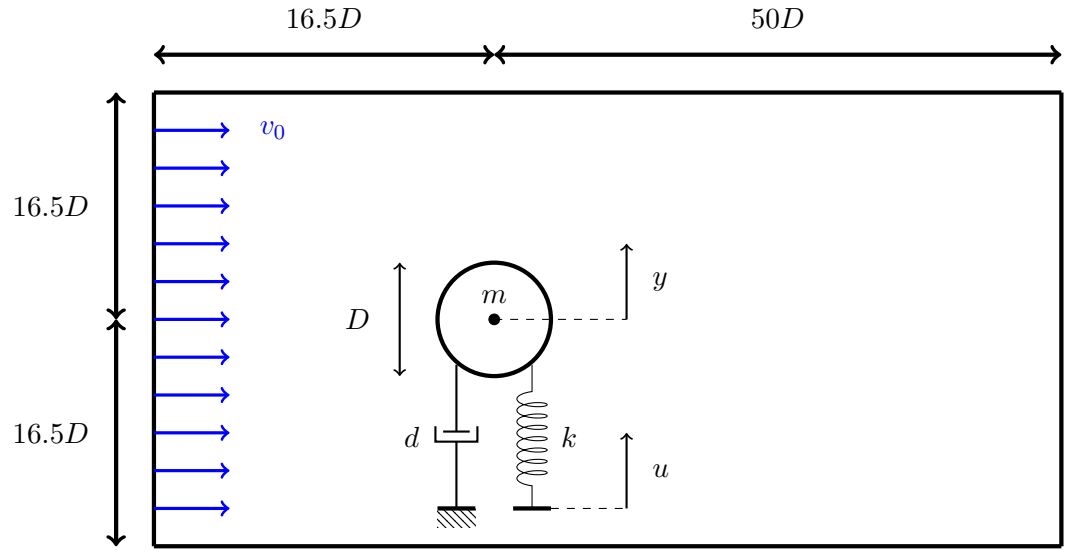


Figure 5.5: The setup for the flow around an oscillating cylinder. The object is mounted upon a spring-damper system which allows it to move in the y-direction. The root point of the spring can be moved to vary the force acting on the cylinder.

$$U^{n+1} = K_P {X_P}^{n+1} + K_I {X_I}^{n+1} + K_D {X_D}^{n+1}$$

with the elements

$$ {X_P}^{n+1} = e^{n+1} \tag{5.5}$$

$$ {X_I}^{n+1} = \frac{h}{2}(e^{n+1} + e^n) + {X_I}^n \tag{5.6}$$

$$ {X_D}^{n+1} = \frac{1}{h}(e^{n+1} - e^n) \tag{5.7}$$

Here, equation 5.6 is derived with a trapezoid time integration and equation 5.7 with the backwards Euler time integration.
For the oscillating cylinder, the error $e(t)$ equals the distance of the cylinder from a given reference point $r(t)$: $e(t) = r(t) - y(t)$. The control goal is therefore to minimize the distance and keep the position of the cylinder at $r(t)$ despite the external forces.

k	69.48	$[N/m]$	Spring stiffness
d	0.0043	$[N/s]$	Damping coefficient
m	0.03575	$[kg]$	Mass

Table 5.2: Parameters for the spring-damper system

Spring-Damper System

The spring-damper system upon which the cylinder is mounted is described by the equation

$$m\ddot{y} + d\dot{y} + ky = F + ku \tag{5.8}$$

y is the position of the cylinder centre, while $\dot{y}$ is its velocity and $\ddot{y}$ its acceleration. The lift force F comes from the fluid flow, while u represents the displacement of the spring root. u is steered by the controller. Equation 5.8 is discretized with the trapezoid method which results in the following set of equations:

$$\begin{bmatrix} \frac{2}{h} & -1 & 0 \\ k & \frac{2m}{h} + d & 0 \\ 0 & \frac{2}{h} & -1 \end{bmatrix} \begin{pmatrix} y^{n+1} \\ \dot{y}^{n+1} \\ \ddot{y}^{n+1} \end{pmatrix} = \begin{bmatrix} \frac{2}{h} & 1 & 0 \\ 0 & \frac{2m}{h} & m \\ 0 & \frac{2}{h} & 1 \end{bmatrix} \begin{pmatrix} y^{n} \\ \dot{y}^{n} \\ \ddot{y}^{n} \end{pmatrix} + \begin{pmatrix} 0 \\ F^{n+1} + ku^{n+1} \\ 0 \end{pmatrix} \tag{5.9}$$

In accordance with the experiments in [16] and the simulations in [18], the system parameters of Table 5.2 are used.

Simulation

To couple the three participants, a multi-coupling scheme is needed. An overview of the necessary communication between the solvers is shown in Figure 5.6. preCICE allows to combine multiple bi-coupling schemes to satisfy this setup. The coupling between Fluid and Solid is serial-implicit and the coupling between Solid and Controller is serial-explicit. This combination of coupling schemes is known to perform well. Not all possible combinations are numerically useful, see [19] for more information.

First, the case is run with a deactived controller, meaning the control gains are set to $K_P = K_I = K_D = 0.0$. The spring root is thereby fixed. This is done to compare the basic setup with the mentioned experimental and numerical results. Figure 5.7 shows the cylinder displacement of this simulation, which was obtained for $T_{Sim} = 20s$ and $\delta t = 5 \times 10^{-3}s$. After a transient phase, the system reaches a stable oscillation with a frequency

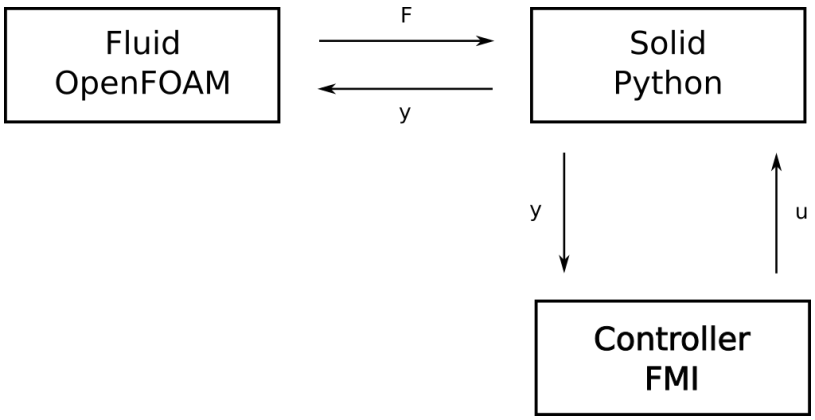


Figure 5.6: The multi-coupling scheme for the oscillating cylinder case. Two different coupling schemes are implemented: The coupling Fluid-Solid is serial-implicit, while the coupling Solid-Controller is serial-explicit.

of $f = 6.5Hz$, which was also reported by [18] and [16]. However, the amplitude results are quite different: Where the cited literature reports an amplitude of $\hat{y}_{lit} = 6 \times 10^{-4}m$, the simulation results show $\hat{y} = 2 \times 10^{-6}m$. This mismatch of two orders of magnitude could not be resolved, but OpenFOAM was allocated as the participant calculating wrong results. Nevertheless, the test case is continued to be used to test the coupling of the FMU controller model with a PDE-based model. The excited oscillation of the cylinder is still a valid control test case for the main focus of this work, the Runner software.

Moving on, the controller is used during the simulation. Deriving from Fig. 5.7, the reference point is set to $r = 4 \times 10^{-6}m$ to keep the cylinder at the equilibrium point of lift and spring force. The simulation starts without controller intervention. After the system has reached a stable phase at $T = 3s$, the controller is activated by setting the control gains to $K_P = 100$, $K_I = 0.0$ and $K_D = 0.0$. The results in Fig. 5.8 show a clear reduction in displacement, indicating that the control algorithm in the FMU model is connected correctly to the simulation by the Runner.

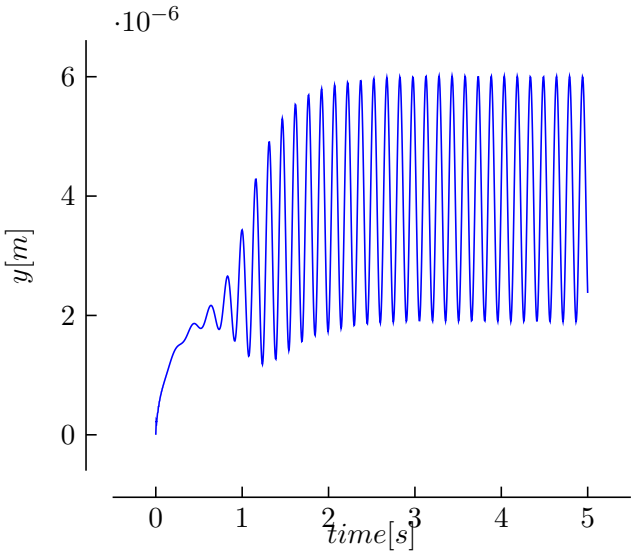


Figure 5.7: Simulated cylinder displacement y over time without controller intervention. The amplitude $\hat{y} = 2 \times 10^{-6} m$ misses the cited literature ([18], [16]) by 2 orders of magnitude. The error could be pointed down to the force calculation in OpenFOAM, without being able to correct it. The frequency of the oscillation $f = 6.5 Hz$ matches the cited values.

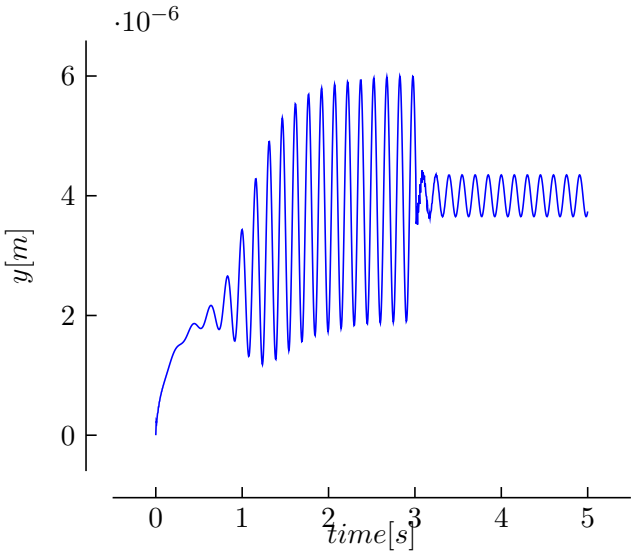


Figure 5.8: Simulated cylinder displacement y over time with controller intervention. The controller is activated at $T = 3s$ with the gains $K_P = 100$, $K_I = 0.0$ and $K_D = 0.0$. This leads to a clear reduction of displacement.

6 Conclusion

6.1 Summary of the Thesis

This thesis developed a first preCICE-FMI Runner to couple controller models to PDEs. The new software loads and executes FMU models and calls preCICE to couple the simulation with other simulation programs. 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. Concern was given to create an easy, standard installation process to make the program accessible for users. Next to the Runner software, this thesis included the creation of a regression test to guardrail further development. Well-documented test cases were implemented which show that the software works correctly and can be used as tutorials for future users.

The preCICE-FMI Runner is compatible with Co-Simulation FMUs of the FMI versions 1, 2 and 3. It supports explicit and implicit coupling via preCICE, as well as the use of acceleration methods. Time-dependent input signals can be set to the FMU model during simulation. Output signals from the FMU are stored for post-processing.
The work of this thesis is limited to an implementation for preCICE v2. Furthermore, the Runner can only exchange data on one vertex. It does not support vector variables within the FMU, but can exchange vector variables from preCICE elementwise as scalars to the FMU model. The logging of internal FMU errors could not be implemented, as well as parallel execution.

The developed software and the test cases files are archived on DaRUS. [1]

6.2 Conclusion

Can the preCICE-FMI Runner be used as a general tool to couple FMUs to PDE-based models or is it confined to a special use case? The new software can be used to couple FMUs in a black-box fashion, but the scope of possible coupling scenarios is restricted by its limited abilities. Some FMU models may require the exchange of more variables than is currently possible. Other models will rely on more interaction with the importer than implemented at this point in time. However, given these limitations, the configuration options of the Runner are general enough to enable many interesting co-simulation scenarios

[1] https://doi.org/10.18419/darus-3408

already.
Beyond control applications, FMUs are often used as system and component models for cyber-physical systems. These simulations can now be enhanced by high-fidelity models through preCICE. It remains to be seen how this will be used. But the two ecosystems that have grown around preCICE and the FMI standard contain some very different tools, the combination of which could lead to interesting new simulation scenarios.

6.3 Outlook

This thesis is the first approach towards a coupling of FMUs with preCICE. It was written with a specific use case in mind, but can serve as a starting point for different implementations. Some possible routes for future work on the preCICE-FMI Runner are:

- **Overcome minor limitations:** Many of the limitations described here can be overcome and would lead to a more versatile Runner. This includes the adaption for preCICE v3, as this major update includes changes in the API and is scheduled for 2023. Moreover, the handling of FMU vector variables and FMU error messages should be included.
- **Explore the coupling of FMUs exported from other simulation tools:** Many tools export their internal models to the FMU format. This gives them a standardized interface, but not necessarily standardized abilities. Exported FMU models may lack vital functionalities like rollback in time for the co-simulation with preCICE. To detect these shortcomings and address them either in the preCICE-FMI Runner or the exporting tools will be an interesting venue.
- **Get user feedback:** The preCICE-FMI Runner should be introduced to both the preCICE and the FMI community to start a discussion and steer future development. What is useful to the communities will decide whether the new tool will be used or not.

Glossary

FMI Functional Mock-Up Interface. 1

FMU Functional Mock-Up Unit. 6

IVP Initial Value Problem. 26

ODE Ordinary Differential Equation. 19

PDE Partial Differential Equation. 2

Bibliography

[1] G Chourdakis, K Davis, B Rodenberg, M Schulte, F Simonis, B Uekermann, G Abrams, HJ Bungartz, L Cheung Yau, I Desai, K Eder, R Hertrich, F Lindner, A Rusch, D Sashko, D Schneider, A Totounferoush, D Volland, P Vollmer, and OZ Koseomur. preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]. *Open Research Europe*, 2(51), 2022.

[2] Valentina Schüller, Benjamin Rodenberg, Benjamin Uekermann, and Hans-Joachim Bungartz. A simple test case for convergence order in time and energy conservation of black-box coupling schemes. 07 2022.

[3] Markus Gross, Hui Wan, Philip J. Rasch, Peter M. Caldwell, David L. Williamson, Daniel Klocke, Christiane Jablonowski, Diana R. Thatcher, Nigel Wood, Mike Cullen, Bob Beare, Martin Willett, Florian Lemarié, Eric Blayo, Sylvie Malardel, Piet Termonia, Almut Gassmann, Peter H. Lauritzen, Hans Johansen, Colin M. Zarzycki, Koichi Sakaguchi, and Ruby Leung. Physics-dynamics coupling in weather, climate, and earth system models: Challenges and recent progress. *Monthly Weather Review*, 146(11):3505–3544, 2018.

[4] M.A. Sprague, J.M. Jonkman, and B.J. Jonkman. FAST Modular Framework for Wind Turbine Simulation: New Algorithms and Numerical Examples. 2015.

[5] A. Junghanns, T. Blochwitz, C. Bertsch, T. Sommer, K. Wernersson, A. Pillekeit, I. Zacharias, M. Blesken, P.R. Mai, K. Schuch, C. Schulze, C. Gomes, and M. Najafi. The Functional Mock-up Interface 3.0 - New Features Enabling New Applications. 2021.

[6] M. Arnold C. Bausch H. Elmqvist A. Junghanns J. Mauß M. Monteiro T. Neidhold D. Neumerkel T. Blochwitz, M. Otter and al. The functional mockup interface for tool independent exchange of simulation models.

[7] Christian Andersson, Johan /AAkesson, and Claus Führer. PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface. LUTFNA-5008-2016(2), 2016.

[8] Führer C. Andersson, C. and J. /AAkesson. Assimulo: A unified framework for ODE solvers. *Mathematics and Computers in Simulation*, 116:26–43, 2014.

[9] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[10] David J Gardner, Daniel R Reynolds, Carol S Woodward, and Cody J Balos. Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 2022.

[11] Peter Fritzson, Adrian Pop, Karim Abdelhak, Adeel Ashgar, Bernhard Bachmann, Willi Braun, Daniel Bouskela, Robert Braun, Lena Buffoni, Francesco Casella, Rodrigo Castro, Rüdiger Franke, Dag Fritzson, Mahder Gebremedhin, Andreas Heuermann, Bernt Lie, Alachew Mengist, Lars Mikelsons, Kannan Moudgalya, Lennart Ochel, Arunkumar Palanisamy, Vitalij Ruge, Wladimir Schamai, Martin Sjölund, Bernhard Thiele, John Tinnerholm, and Per Östlund. The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development. *Modeling, Identification and Control*, 41(4):241–295, 2020.

[12] Wolfgang Bangerth and Timo Heister. What makes computational open source software libraries successful? *Computational Science and Discovery*, 6(1):015010, nov 2013.

[13] Virginie Galtier, Stephane Vialle, Dad Cherifa, jean-philippe Tavella, Jean-Philippe Lam-Yee-Mui, and Gilles Plessis. Fmi-based distributed multi-simulation with daccosim. *Simulation Series*, 47, 04 2015.

[14] A. Prakash and K. D. Hjelmstad. A feti-based multi-time-step coupling method for newmark schemes in structural dynamics. *International Journal for Numerical Methods in Engineering*, 61(13):2183–2204, 2004.

[15] N. M. Newmark. A method of computation for structural dynamics. *J. Eng. Mech. Div.-ASCE*, 85(3):67–94, 1959.

[16] P. Anagnostopoulus and P.W. Bearman. Response characteristics of a vortex-excited cylinder at low reynolds numbers. *Journal of Fluids and Structures*, 6(1):39–50, 1992.

[17] A. Placzek, J.F. Sigrist, and A. Hamdouni. Numerical simulation of an oscillating cylinder in a cross-flow at low reynolds number: Forced and free oscillations. *Computers and Fluids*, 38(1):80–100, 2009.

[18] S.A. Sicklinger. *Stabilized Co-Simulation of Coupled Problems including Fields and Signals*. PhD thesis, Technical University of Munich, 2014.

[19] Hans-Joachim Bungartz, Florian Lindner, Miriam Mehl, and Benjamin Uekermann. A plug-and-play coupling approach for parallel multi-field simulations. 2015.