FMI Co-Simulation between 2D/3D component models and HVAC/control models

Andreas Nicolai[1], Andreas Söhnchen[1]
[1]Institut für Bauklimatik, Fakultät Architektur, TU Dresden, Dresden, Germany

Abstract

Detailed construction and building component models, including hygrothermal porous material transport models, can be used to model a large variaty of modern energy transfer and storage systems. These include heated concrete slabs, shallow soil heat collectors, heated wall layers, combined photovoltaic and construction panels etc. The interaction with connected HVAC systems/energy distribution systems and/or control models is, however, often limited in such tools. With the use of the FMI runtime tool coupling standard, it is possible to connect detailed component models with other simulation tools. This is illustrated in the article using the hygrothermal transport model DELPHIN. The article first describes the FMI co-simulation interface of the tool. Then, the generation of control model FMUs is shown using open-source technology for FMI generation (OpenModelica, FMICodeGenerator). Using MASTERSIM as co-simulation master, the setup of an application case (heated concrete slab) is illustrated, including two ways of manually generating control model FMUs.

Key Innovations

- FMI interface of heat and mass transport model
- Illustration of workflow and FMU generation technology

Practical Implications

This article provides practitioners with a detailed overview of the FMI technoligy. It provides a clear description on how to manually create own FMUs and interface them with existing, complex simulation models.

Introduction

Simulation tools with flexible parametrisation and physically fundamental modeling can be used in versatile ways to model different things from quite different physical/engineering domains. Two examples: the calculation toolkit COMSOL MultiPhysics (COMSOL, 2021) includes an own mathematical language to express model variants and provide parameters, centered around a solution engine for partial differential equations in 2D/3D with FEM. The Modelica language (MA, 2021c) and its runtime environments/solvers permit users to express very different physical models. Numerous other generic modeling languages/tools exist, that all strive to be usable by a very broad target audience and for a many different applications. Each of the tools is specialised in some way. For example, Modelica is very good at expressing coupled non-linear problems involving systems of differential and algebraic equations (DAE), yet no partial differential equations (PDE). COMSOL, on the other hand, is centered around solving PDEs, yet the included language cannot express all that fine detail that's possible with Modelica and similar modeling languages. In practical applications, often one tool is not sufficient, yet a combination of multiple tools with runtime data exchange is needed.

The FMI Standard

Before the MODELISAR project (Clauß et al., 2011) and the development of the FMI standard, interfacing of dynamic calculation tools at runtime has been done mostly using propriatory bilateral interfaces, or using some kind of dedicated middleware (see article on the middleware CoSimA+ (Stratbücker et al., 2011) which contains a review). The Functional Mock-Up Interface (FMI) is an abstract description of *what* is being exchanged, *when* and *how*, see Blochwitz et al. (2011) for an introduction. It boils down to:

- different simulation models/tools are packaged into Functional Mock-Up Units (FMUs),
- these FMUs are collectively simulated by a *master program*, that can either integrate all models into one big common solver for all exposed differential equations (a mode called *ModelExchange*), or by orchestrating distributed integration and data exchange at dedicated time points (this mode is called *Co-Simulation*).

The packaging of a model into an FMU involves creating a model-specific description file (XML format, named `modelDescription.xml`), a runtime library and additional model specific resources. These are all

placed in a directory structure, zipped and renamed to have a `.fmu` file extension. Details of this procedure are described in the FMI standard documents (MA, 2021a). Creating such an FMU manually is rather complicated, especially programming the runtime library, but there are toolkits that simplify the process (see discussion below).

The process of running the coupled simulation is handled by simulation master programs. The authors detailed discussion of the pros and cons of either approach (ModelExchange vs. Co-Simulation), can be found in section 6/Activity 1.2 of the IEA Annex60 report (Wetter and van Treeck, 2017).

Specialised simulation models with FMI support

Besides generic simulation tool kits and modeling languages, the vast majority of dynamic simulation tools are specialised to some application case. Since the scope is limited, parameter choices and mathematical formulation are known and fixed, these tools can better optimise their calculation kernels and may almost always achieve (much) better performance than general purpose modeling environments. Also, and that is even more important for engineering users, these tools can provide dedicated user interfaces which simplify model setup significantly, and provide thorough input data checking. Consequently, these types of tools are found way more frequently in practical use than generic/multi-purpose modeling tool kits as mentioned above.

The DELPHIN simulation model

The hygrothermal building component/construction detail calculation software DELPHIN (Nicolai, 2020) is one example for such dedicated simulation programs. It is specialised for efficiently solving the highly nonlinear partial differential equation systems used to express coupled heat and mass transport problems in 2D/3D geometries (see Nicolai and Ruisinger (2020) for a discussion on implemented algorithms and numerical parameter optimisation).

Despite the specialisation on building construction details the software includes a very general physical model formulation and offers numerous ways to specify boundary conditions, sources/sinks and auxiliary models. This allows DELPHIN to be used for quite different application cases, such as:

- modeling of soil energy collectors,
- active (heated) carbon-concrete slabs,
- thermal analysis of combined photovoltaic/thermal collectors,
- heating/cooling dynamics of bio-reactors (and other equipment),
- soil ice storage, and many more.

The recent use of DELPHIN for dynamic equipment/system modeling resulted in an increased demand for control model integration. Given the large number of different control strategies and parameters, it is not meaningful to integrate these into the software and native DELPHIN calculation model itself. Also, from the user's point of view, flexible adjustment of control model equations and algorithms should not be slowed down by implementation time and software release schedules. Hence, we opted for a FMI runtime-coupling of flexible control models and standard DELPHIN calculation functionality.

```
├── binaries
│   ├── win32
│   │   └── model.dll
│   ├── win64
│   │   └── model.dll
│   ├── darwin64
│   │   └── model.dylib
│   └── linux64
│       └── model.so
├── modelDescription.xml
└── resources
    ├── climate
    │   └── xxx.c6b
    ├── materials
    │   ├── xxx.m6
    │   └── yyy.m6
    └── Project.d6p
```

Figure 1: DELPHIN FMU directory layout

DELPHIN FMI functionality

The DELPHIN model separates functionality and parametrisation through use of project and resource files (climate data files, material data files). The same principle applies to FMUs, where the runtime library is the same for all models, and only the embedded project parameters change. Hence, the directory layout of DELPHIN FMUs (see Fig. 1) is always the same.

The subdirectory `resources/climate` contains the used climate data file (c6b/wac/epw) and other referenced time series data files (tsv/ccd). The `resources/materials` subdirectory holds all materials referenced by the project. The file `Project.d6p` is the renamed project file - within the FMU the project file always has the name `Project.d6p`. The shared library files are placed in the respective subdirs of `binaries`. Note, on Mac (darwin) and Linux there are no 32-bit variants of DELPHIN, and for Windows, the 32-bit variant is only used for compatibility with older 32-bit FMU exporting tools. Auxilliary shared library files (dll) are placed in the `win32` and `win64` subdirectories as needed. Following the FMI specification, the file name of the shared library files (`"model"` in Fig. 1) matches the FMU name. When exporting the FMU the user has the choice of embedding only the binary library for the current platform (hereby reducing FMU file size), or for all platforms, which will then become a cross-platform FMU.

The separation of model functionality and parameters permits powerful optimisation/parameter analysis variants. First, the FMU is extracted somewhere to create the reference/starting point of the variation. Then a script might be run that (1) modifies project parameters, (2) zips the directory structure into the FMU, (3) copies the FMU into the target directory with the FMI coupling scenario, (4) executes the coupled simulation and (5) evaluates results and generates input for the next iteration.

The DELPHIN FMU currently supports FMI for Co-Simulation Versions 1.0 and 2.0, the latter with the capability of storing/restoring the FMU state (this allows iterative co-simulation master algorithms (Nicolai, 2018)).

Parameters, Input and Output Variables

Since all project-related parameters are contained in the embedded project file and resources, the exported interface defines only one parameter: *ResultsRootDir*. This optional parameter can be used to indicate where DELPHIN shall write its own output data. Generally, for FMUs solving PDE, it is not meaningful to pipe fields of data (e.g. temperature or moisture distributions) over the FMI interface as scalar variables (since data fields are not supported by FMI standard so far). Also, performance-wise it is much more meaningful to write dedicated output file in formats suitable for post-processing applications. Hence, the DELPHIN FMU writes its result data files exactly as in a stand-alone run, only in the directory indicated by the *ResultsRootDir* string parameter.

The input variables accepted by DELPHIN are always of type fmiReal (double-precision floating point) and continuous variables. They are generated for all time series input data (climate conditions) that are defined as specific FMI input variable. The names of these variables are always automatically generated based on the condition names and type prefix. For example, a climate condition named "ExternalTemperature" will generate an FMI input variable name "ClimateCondition.ExternalTemperature", which is expected always in default SI unit of the respective physical quantity (degree Centigrade for temperatures, Watt for heat fluxes, etc.). To avoid errors, this unit is also written in the `modelDescription.xml` file. Each input variable needs to be given a default value. Using this default value the model behavior can be easily tested in stand-alone runs before being exported. This is particularly useful for engineering workflows, since it speeds up the process of generating a correctly parameterised FMU and co-simulation scenario.

Output variables are also automatically generated for all defined and used *scalar* output quantities. This can be sensor values, or integral flux quantities, or mean values of a quantity in a certain domain of the simulation model. Any output that yields a single value per time point will be exported as an output variable. This way it is possible to export virtually all calculation results (and interium) quantities over the FMI interface.

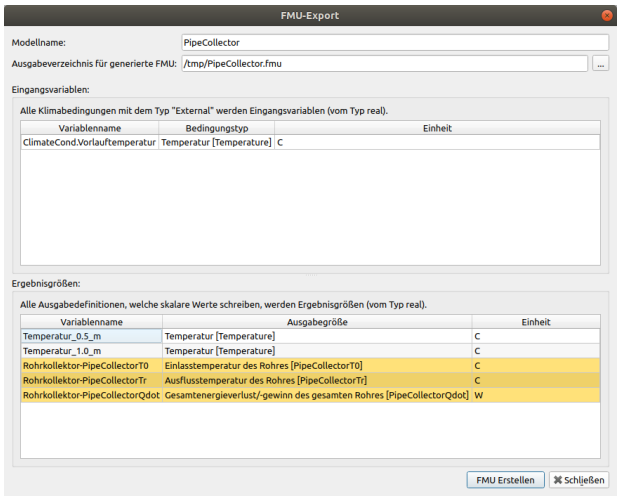

Figure 2: Example model export with one input variable, two regular outputs and further auxilliary model outputs

Lastly, some models may generate auxilliary scalar outputs. For example, the pipe collected model (see below) will provide the outflow media temperature and energy loss/gain over the entire pipe. These are also scalar variables and exported over the model interface.

The interface variables are summarized as part of the export process, whereby the auxilliary scalar outputs are highlighted in yellow (see Fig. 2).

Implementation details related to the variable time-step integrator in the DELPHIN FMU slave

The use of variable step sizes in the FMU slave and co-simulation algorithm needs special consideration. For solvers of PDE problems variable time-step integrators can reduce simulation time substantially. The FMI standard explicitely provides co-simulation masters with the ability to use variable step sizes, provided the slaves are compatible with that (FMI feature `canHandleVariableCommunicationStepSize`).

Consider Fig. 3 where the internal steps of the slave (top line) and the communication intervals of the co-simulation master (bottom line) are illustrated. In between communication intervals the master can exchange FMI input variables. Hence, when the integration commences again inside the slave, it has to begin *exactly* at the start of the next communication interval.

This is achieved by shortening the time step at the end of the communication interval, sometimes substantially. This not only incurs more steps to be taken

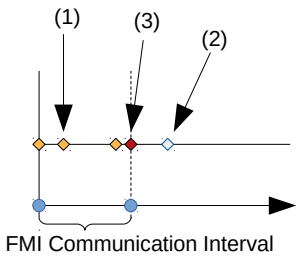

Figure 3: Illustration of variable step synchronisation and the need to artificially shorten steps to hit communication interval end points; (1) regular slave integration steps, (2) step that would be taken without FMI communication, (3) stop-point inserted to hit communicatino end point exactly

by the FMU slave's integrator, but also more overhead for Jacobian matrix update/factorisation. Indeed, drastic changes in time step sizes require a Jacobian matrix update which could otherwise be re-used in modified Newton schemes. In the DELPHIN FMU the use of iterative Krylov-subspace solvers somewhat reduces this penalty at each communication interval end, though the overhead of updating/factorising the ILU-preconditioner remains.

Output handling

FMUs with large number of outputs (as arise from PDE simulations) usually write these outputs to dedicated output files. The FMI standard does not provide any means to tell the slave a suitable target directory for these outputs. Hence, the usual procedure is to write outputs in a hard-coded directory or current working directory. However, this approach is not meaningful when using the same FMU slave *multiple times* in the same co-simulation scenario. This, however, can be a useful feature, especially for DELPHIN FMUs, e.g. when instantiating slaves for several soil heat collector fields.

MASTERSIM solves the problem of the slave-specific output directories by supporting a special string-type FMU input variable called `ResultsRootDir`. Whenever a slave supports such a variable, MASTERSIM will provide a unique, writeable, slave-specific path for the FMU to use (see Nicolai (2021a), online manual, section *Directory "slaves"*). In turn, the DELPHIN FMU does generate its output files in the provided directory.

Lastly, special care needs to be taken when using FMUs with own output files in an iterative co-simulation algorithm. When repeating a communication interval, it is possible that DELPHIN had appended outputs to files already in the first run through the communication interval. When repeating the communication interval, there will be outputs added again with the same time stamps, yet slightly modified content to the output files. Even when communication step sizes are much smaller than output steps, the problem remains since variably-sized communication steps are usually not synchronized with the output schedules in FMU slaves. Caching outputs until a communication interval is completed (i.e. no further iterations are made) is not meaningful for PDE-type FMUs, since output data tends to be large and a cache would potentially occupy much memory storage.

Instead, we recommend the following pragmatic approach to obtain output files with consecutive outputs in the DELPHIN d6o files:

- in a script, read the output file and in each data line extract the first number (the time stamp)
- process file *back-to-front* and drop all lines whose time stamp is larger than or equal to the time stamp in the previously processed line

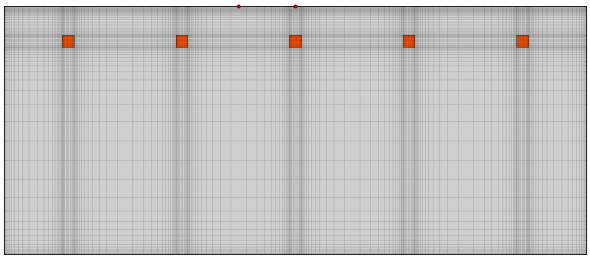

Figure 4: Simulation model of the elevated concrete drive way. Sensors are placed at the surface of the drive way, where the frost-free condition is evaluated.

Application Example

Heating concrete driveways to prevent them from freezing/avoid snow cover may require significant amount of electric energy. The planning of such systems currently involves answering the following questions: where shall the heating elements be placed, and how much *constant* power is needed. Indeed, the current approach of determining necessary heating power involves running a steady-state thermal bridge calculation with constant cold boundary conditions. Optionally, an additional energy demand for freezing of snow pack / falling snow is considered.

While this approach gives a worst case assumption, it also yields the largest dimensioning for the heating elements and gives a *constant heating power* during operation, that is likely to be very much on the safe side and thus waste a lot of energy.

Steady-state analysis

Fig. 4 shows an example of a concrete drive way with 5 distinct heating elements (pipes approximated with square cross section). There are two of these 1m wide slabs side by side, each supporting one side of the train car running on top. Both have the same cross section, so analysis of one supporting concrete slab is sufficient.

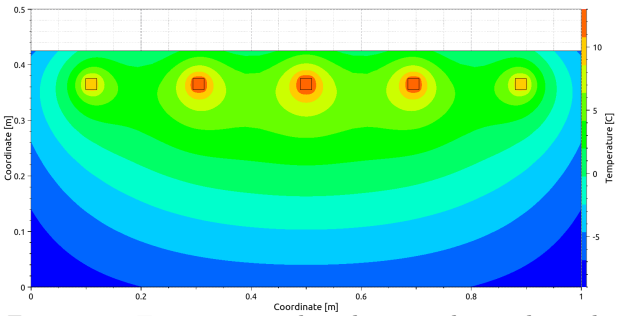

Figure 5: Temperature distribution obtained in the steady-state calculation with 300 W/m (per meter length of the drive way) uniformely distributed into all 5 heating elements.

The steady state analysis with worst-case constant conditions and heating power of 300 W/m (W per length of driveway) yields a temperature distribution as shown in Fig. 5. Ambient conditions were fixed at -10°C with increased exchange coefficient due to elevated position of the driveway. The simulation was done using different uniform heating powers, modeled as energy source inside the heating pipes. For example, 300 W/m devided by total heating element cross section of 5x0.02x0.02 m2 gives a volumentric energy source of 150 kW/m3. This power was found to be sufficient to keep the surface temperature above 2 °C, which was the selected safe limit for operation. Given the length of the driveway (several 100 m) a significant amount of heating power is needed.

Obviously, in Germany we do not have -10 °C constant over several months, so for the most time we may be able to save energy by using less heating power. However, we need to define a meaningful control strategy, define sensor points and quantify the savings in energy vs. risk of unsuitable conditions during operation hours. All these variants can be tested in a dynamic simulation, where the concrete slab itself is modeled in a DELPHIN FMU and the control logic is modeled in a control model FMU.

FMUs

Concrete slab FMU

The simulation model of the concrete slab was exported from DELPHIN with the aforementioned export procedure. The FMU has two output variables `TemperatureSensor_1` and `TemperatureSensor_2` for the two sensors at the top of the concrete slab. For control purposes only sensor 2 is being used, that is placed between the heating elements and thus shows always the lower temperature. The volumetric heat source (energy production rate) is the only input variable `EnergySourceHeating` of the `ConcreteSlab` FMU.

```
loadModel(Modelica, {"3.2.1"}); getErrorString();
loadModel(Modelica_DeviceDrivers); getErrorString();
setLanguageStandard("3.3"); getErrorString();
loadFile("HeatingControl.mo"); getErrorString();
setDebugFlags("backenddaeinfo");getErrorString();
translateModelFMU(HeatingControl, fmuType="cs",
version="2.0"); getErrorString();
```

Figure 6: Example OpenModelica script `createFMU.mos` to generate an FMU. The first three lines can be omitted, if no external libraries are needed.

Modelica control model FMU

For the implementation of a typical control model we investigated two possibilities: a) use of Modelica, specifically the OpenModelica environment, b) writing our own control model FMU in C/C++.

Defining a control model in OpenModelica works pretty much like in any other Modelica development environment. The export of the FMU, however, is best done with a script. A simple approach is the use of the OMShell, with the commands:

```
>> loadFile("/path/to/HeatingControl.mo")
>> translateModelFMU(HeatingControl, fmuType="cs")
```

The `translateModelFMU()` function generates the FMU by default in a temporary directory and prints out the export path. By default, if specifying cs (Co-Simulation) as fmuType it will export an FMU with support for FMI for Co-Simulation version 1.0. If FMI 2.0 features are needed, you need to export the model with:

```
>> translateModelFMU(HeatingControl, fmuType="cs",
   version="2.0")
```

For practical purposes it is often much more convenient to use a scripted solution, hereby creating an OpenModelica compiler script, say `createFMU.mos` (Fig. 6) and running it with the command line:

```
> omc createFMU.mos
```

The FMU export generates automatically input and output variables based on the variable declaration prefixes input and output, as well as parameters. For example, the minimalistic controller (Listing 1) will have one FMU input variable `TSurface`, one output `PHeating`, and two parameters.

Listing 1: Minimalistic proportional controller model with limits on temperature and heating power

```
model HeatingControl
  // input temperature in K
  input Real Tsurface;
  // output energy source in W/m3
  output Real PHeating;
  // effective cross section area in m2
  parameter Real area = 0.02*0.02*5;
  // max. available power in W/m
  parameter Real MaxPower = 300;
  // required heating power in W/m
  Real HeatingPower;
equation
  // simple p-controller for this example
  // Compute heating power in W/m,
  // MaxPower at 1K difference
```

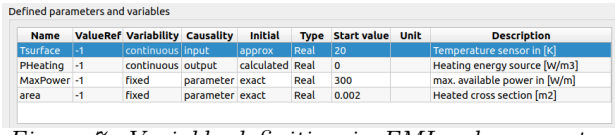

Figure 7: Variable definition in FMI code generator

```
  HeatingPower = max(0, MaxPower*(276.15 - Tsurface));
  // Compute energy source in W/m3
  PHeating = min(HeatingPower, MaxPower)/area;
end HeatingControl;
```

Custom control model FMU

As an alternative to Modelica it is also possible to handcraft an FMU, hereby creating very small and fast models. For simple models such as this heating model, the implementation effort should be minimal. However, the C interface required by the FMI standard is far from trivial to implement. Hence, the author has developed a small open-source toolkit to generate a barebone of an FMU where only the physical equations need to be inserted. The toolkit FMICodeGenerator (Nicolai, 2021b) is a small Python script that collects information about the interface variables (inputs, outputs, parameters and their properties) and then generates a directory structure with compile-ready code. The (engineering) user now only has to fill in the code lines with the actual physics, run a batch/shell script and obtains a fully functional FMU.

For the example above, the procedure is rather simple:

1. download the FMICodeGenerator repository from github
2. start the code generator wizard (the python script main.py) and specify variables as shown in Fig. 7
3. finish wizard and generate the FMU source directory
4. add model equations
5. generate FMU

Step 4 requires editing the generated source code, in this case a single file `HeatingControl.cpp`. The file is prepared so that access to input variables and output variables is very simple. All that's needed is to insert code, very similar to the Modelica code above, into the marked section of the code (see listing 2).

Listing 2: Own model implementation inside `HeatingControl::integrateTo()` of the generated source code

```
// get input variables
double Tsurface = m_realVar[FMI_INPUT_Tsurface];
double MaxPower = m_realVar[FMI_PARA_MaxPower];
double area = m_realVar[FMI_PARA_area];

// *** own code starts here ***

// Compute heating power in W/m
double HeatingPower =
    std::max(0.0, MaxPower*(276.15 - Tsurface));

// Compute energy source in W/m3
double PHeating =
    std::min(HeatingPower, MaxPower)/area;

// output variables
m_realVar[FMI_OUTPUT_PHeating] = PHeating;

// *** own code ends here ***
```

Step 5 in the generation process involves simply calling the ready-made scripts `build.bat` or `build.sh` (on Mac/Linux) and deployment scripts `deploy.bat`, `deploy.sh` or `deployMac.sh`, depending on the platform.

Co-Simulation setup

There are numerous co-simulation master programs available (see cross-check list (MA, 2021b)). We choose the open-source master program MASTERSIM (Nicolai, 2018). It implements all needed algorithms (including iterative master algorithms, if needed), and provides a good separation between co-simulation scenario setup and the actual FMU. This is particularly useful for engineering workflows that involve parameter variation/sensitivity studies (see Nicolai (2021a), online manual, section *Workflows*). Also, it can be scripted very well.

The co-simulation setup involves essentially 3 steps:

- import of the FMUs and analysis of the interfaces
- definition of FMU connections (input - output variable associations)
- setting simulation and algorithmic parameters

For this example only the `ConcreteSlab.fmu` (DELPHIN model) and the `HeatingControl.fmu` need to be imported. The connection of the FMUs is rather simple: *TemperatureSensor_2* output from the `ConcreteSlab` fmu feeds into *TSurface* from `HeatingControl` fmu. And *PHeating* feeds into *ClimateCond.EnergySourceHeating* (see Fig. 8).

Unit conversions

Particular care has to be taken when connecting variables with different units. Here, the surface temperature is exported in °C, but expected in Kelvin by the Modelica control model FMU. Even if FMUs export units alongside the variable declarations, an automatic value conversion is tricky, if only because of different naming conventions for units in different simulation tools. Hence, our recommendation is to *document units of variables clearly* and configure unit conversions manually when setting up the co-simulation. In MASTERSIM this is done by assigning connection conversion properties. Here we define an offset parameter 273.15 to convert from C to K when computing the value to pass to *HeatingControl.TSurface*.

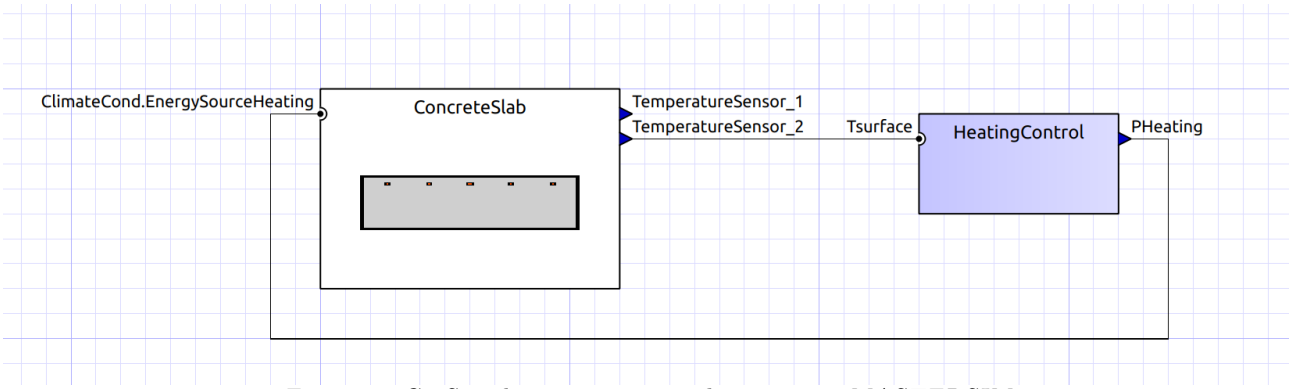

Figure 8: Co-Simulation scenario schematics in MASTERSIM

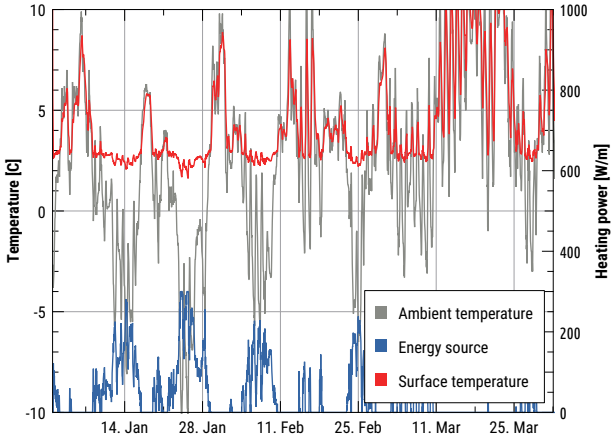

Figure 9: Temperatures and controlled energy source for the first 3 months of the transient simulation

Co-Simulation algorithms and parameters

For most applications, a simple non-iterating Gauss-Seidel algorithm with fixed step sizes is a good starting choice. We selected constant 10 min data exchange intervals and ran the simulation for the first cold months of the year. Indeed, as shown in Fig. 9, the heating power is adjusted based on the surface temperature computed with the DELPHIN model.

Performance considerations

A Co-Simulation is generally slower than a stand-alone simulation, not only because of the additional overhead of transferring data from one FMU to the next. Also, the feedback of one model onto the other may be causing more iterations, potentially more convergence and error test failures and result generally in more steps to take. Furthermore, the requirement to stop an integration at the end of each communication interval increases the work for variable step size solvers, who cannot optimally adjust time steps anylonger. In the test above, the performance overhead is significant: a stand-alone DELPHIN simulation took about 2 minutes for 4 months of calculation, whereas the coupled simulation (with a 10 min communication step) took about 15 min. Admittingly, this is an unfair comparison because the stand-alone model did not include a controlled heating source.

It is generally believed that using a larger communication interval size will increase performance, yet reduce accuracy. Surprisingly, the first assumption does not always hold when it comes to error-controlled, adaptive time step integrators, such as used in DELPHIN. It turns out that *taking larger communication steps* with a type of feedback control as used in this example will in fact lead sometimes to *longer simulations*. The longer a communication step lasts, the stronger are the step-changes of the computed heating powers passed on to the DELPHIN model. These large jumps (in the first derivative of the energy balances) will lead to many error test failures and cause the solver to effectively restart each communication interval with very small time steps. In a variation test using 10, 20 and 30 min as communication step size, the simulation lasted 15 min, 19 min and 17 min, respectively. Alas, this observation is very problem dependent and thus individual testing of the optimal communication step is advised.

Fortunately, varying communication step sizes in a meaningful size range (i.e. at least smaller than the typical expected frequency of input signal changes) did not impact simulation results much. This is illustrated in Fig. 10, which shows a comparison of computed surface temperatures for three different co-simulation step sizes for the first few hours. The communication step size effectively determines, how fast the heating control can react on changes in the sensor temperature. As usual for delay-type controls, the longer the delay and unmodified control regimes, the larger will be the deviations of the sensor value from the setpoint. Hence, the variants with larger communication step sizes show the observed larger oscillations in the calculated sensor temperatures.

Despite that, the total energy consumption in the first 4 months of the year varied only from 94.9 kWh/m (for 10 min) to 95.1 kWh/m (30 min), an insignificant reduction in accuracy.

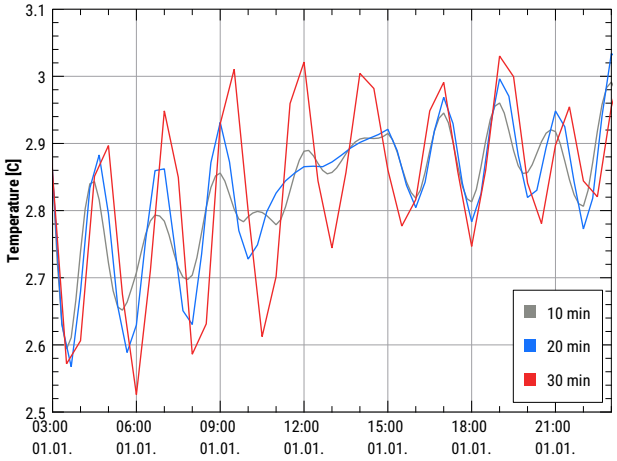

Figure 10: Computed sensor temperatures for co-simulations with 10, 20 and 30 min synchronisation interval length

Based on the presented co-simulation setup, many variants of the model and control parameters can be analysed to optain optimal design parameters. However, a discussion of these findings is beyond the scope of this article.

Conclusions

The FMI simulation coupling standard makes it relatively simple to extend comprehensive simulation models with external functionality. As illustrated in the article, such additional models can often be crafted with open-source tools such as OpenModelica or even compiled manually with the help of code-generators such as the FMICodeGenerator. Similarly, open-source technology exists for running the coupled simulation. The *FMI for Co-Simulation* is used in this article, as it allows use of the optimised numerical integration engine in the DELPHIN model. Also, this coupling method appears to be more commonly supported in building engineering tools than *FMI for ModelExchange*.

The setup of the co-simulation scenario with tools like MASTERSIM is fairly straight forward and even possible within an engineering workflow (and time budget). Care should be taken when connecting variables from different models with potentially different unit conventions. The value conversion feature of MASTERSIM may be of help here.

Regarding the performance of coupled simulation, the optimal choice of co-simulation step size is problem specific. Initially it should be selected according to the expected frequency of input signal changes.

The error introduced to the model by larger communication intervals my be interpreted, for example, as deliberately choosen delay time between operation model adjustments. In the presented example, a communcation interval of 10 min would mark the minimum time delay between an adjustment of the heating power.

The presented example and all input data can be accessed on the MASTERSIM website: `https://sourceforge.net/projects/mastersim/files/examples`

References

Blochwitz et al. (2011). The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *8th International Modelica Conference 2011*.

Fraunhofer IIS (2011). *MODELISAR : from system modeling to S/W running on the Vehicle ; von der System-Modellierung zur Software im Fahrzeugeinsatz ; Schlussbericht*.

COMSOL (2021). COMSOL Multiphysics® v. 5.4. COMSOL AB, Stockholm, Sweden. www.comsol.com.

MA (2021a). FMI Standard Webpage, The Modelica Association, accessed Jan. 2021. fmi-standard.org.

MA (2021b). FMI Tool Listing, website, accessed 2021, The Modelica Association. fmi-standard.org/tools.

MA (2021c). Modelica® - A Unified Object-Oriented Language for Systems Modeling, Language Reference, Version 3.4, The Modelica Association. www.modelica.org.

Nicolai, A. (2018). Co-Simulations-Masteralgorithmen - Analyse und Details der Implementierung am Beispiel des Masterprogramms MASTERSIM. *Qucosa http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa2-319735*.

Nicolai, A. (2020). DELPHIN 6 Webseite. https://bauklimatik-dresden.de/delphin.

Nicolai, A. (2021a). FMI Co-Simulation Masterprogram MASTERSIM, online manual, accessed 2021. https://bauklimatik-dresden.de/mastersim.

Nicolai, A. (2021b). FMICodeGenerator, project website, accessed 2021. https://github.com/ghorwin/FMICodeGenerator.

Nicolai, A. and U. Ruisinger (2020). Performanceoptimierung hygrothermischer Simulationen durch Parameteroptimierung. *Bauphysik 42*(6), 289–299.

Stratbücker, S., C. van Treeck, S. R. Bolineni, D. Wölki, and A. Holm (2011). A co-simulation framework for scale-adaptive coupling between heterogeneous computational codes. In *Roomvent 2011*.

Wetter, M. and C. van Treeck (2017, September). *IEA EBC Annex 60: New Generation Computing Tools for Building and Community Energy Systems*.