Co-Simulation between detailed building energy performance simulation and Modelica HVAC component models

Andreas Nicolai and Anne Paepcke

andreas.nicolai@tu-dresden.de

Starting point: NANDRAD - Building energy simulation engine

- Multi-zone building model
- Realistic building physics (spatially resolved walls/storage members)
- CVODE time integration engine (error controlled variable-order, variable step method)
- Algebraic relations embedded in ODE equations
- Using Newton-Krylov-method for non-linear system of equations
- Special problem-specific sparse matrix structures and preconditioners
Multi-zone model, embedded equipment loop and sparsity pattern

Instead of hard-coded equipment models rather use Modelica-based equipment models
A lot of code like this!

Symbolic analysis (OpenModelica/SimulationX) **really slow** on large Models
Idea tested: get everything into Modelica via Code-generator

Alternative:
Export building as Functional Mockup Unit (FMU) and connect to Modelica equipment model

A lot of code like this!
Symbolic analysis (OpenModelica/SimulationX) really slow on large Models
Software architecture of NANDRAD engine

**Integrator Implementation**
- `setTime(t)`
- `init(model)`
- `step()`
- `y0()`
- `t0()`
- `tEnd()`
- `n()`
- `yOut(t_out)`
- `writeOutputs(...)`
- `stepCompleted(t,y)`

**Physical Model Implementation**
- `setY(y)`
- `ydot()`
- `query functions`
- `control functions`
- `implmentes physical model equations and the computation of the system function f(t,y)`
- `state of object changes only through calls of control functions`

**Solver Control System**
- `implents core integration loop: calls step() function in Integrator, also manages output schedules`
- `query functions`
- `control functions`
Export ODE system function via FMU for ModelExchange

Integrator Implementation

- solves ODE system of type \( y' = f(t,y) \)
- implicit solver with Newton-Raphson iteration

Physical Model Implementation

- implements physical model equations and the computation of the system function \( f(t,y) \)
- state of object changes only through calls of control functions

Solver Control System

- Implements core integration loop: calls step() function in Integrator, also manages output schedules

Corresponds to FMI ModelInterface
Export physical model and integrator as FMI for Co-Simulation

Corresponds to FMI Co-Simulation interface

**Integrator Implementation**

- `solves ODE system of type \( y' = f(t, y) \)`
- *implicit solver with Newton-Raphson iteration*

**Physical Model Implementation**

- *implements physical model equations and the computation of the system function \( f(t, y) \)*
- *state of object changes only through calls of control functions*

**Solver Control System**

*Implements core integration loop: calls \( \text{step()} \) function in Integrator, also manages output schedules*

---

To be replaced by Co-Simulation Master
FMI Generation Process

1. Starting with BIM \(\rightarrow\) Project file for standalone building simulation
2. Indicate which thermal zones shall be heated/cooled and be used in the FMU-interface
3. Run design day simulations (heating/cooling)
4. Generate report file (zone ID – variable index mapping, design loads)
5. Create ModelDescription.xml
6. Create Modelica-Wrapper and Adapter models (corresponding to project)
7. Compress project file, all database files, pre-compiled FMU-DLL and additional dependent libraries into FMU-archive

Steps 3 to 7 are done automatically by NANDRAD solver via command line:

```
NandradSolver RowHouse_var1.nandrad --fmu-export=RowHouse_var1.fmu
(exported FMI is exported as FMI version 2, for **both ME and CS**)
```
FMI Interface to NANDRAD Building Model FMU
Outputs
- Climate data

FMI Interface to NANDRAD Building Model FMU
Outputs

- Climate data
- Zone air/ radiant temperatures

FMI Interface to NANDRAD Building Model FMU
Outputs

- Climate data
- Zone air/radiant temperatures
- Setpoint temperatures (schedules)

FMI Interface to NANDRAD Building Model FMU
Outputs

- Climate data
- Zone air/radiant temperatures
- Setpoint temperatures
- Electric loads
- Warm water consumption

FMI Interface to NANDRAD Building Model FMU
Outputs
- Climate data
- Zone air/ radiant temperatures
- Setpoint temperatures
- Electric loads

Inputs
- Convective and radiant thermal loads

FMI Interface to NANDRAD Building Model FMU
Outputs
- Climate data
- Zone air/ radiant temperatures
- Setpoint temperatures
- Electric loads

Inputs
- Convective and radiant thermal loads

Zone specific inputs/outputs scale with number of zones! Manual connection ???

FMI Interface to NANDRAD Building Model FMU
Application Scenario 1: Import into Modelica environment

Demonstration Case 1
Application Scenario 1: Import into Modelica environment

Demonstration Case 2
Introduce Modelica-Adapter Models (autogenerated)
Introduce Modelica-Adapter Models (autogenerated)

Connectors match exactly native FMU connectors by name.

Connectors match complex ports of HVAC library.
Introduce Modelica-Wrapper Models (autogenerated)
Introduce Modelica-Wrapper Models (autogenerated)

The building model from point of view of the user within the Modelica environment
Introduce Modelica-Wrapper Models (autogenerated)

Only need to connect a few ports
Application Scenario 2: Export equipment FMU

Export FMU – with only adapter variables as FMI interface variables
Application Scenario 2: Co-Simulation with CoSim-Master

About MasterSim ...
MasterSim is an FMU Co-Simulation master and programming library. It supports the Functional Mockup Interface for Co-Simulation in Version 1.0 and 2.0. Using the functionality of Version 2.0, it implements various iteration algorithms that rollback FMU slaves and increase stability of coupled simulation.

MasterSim is actively developed at the Technische Universität Dresden, Institut für Bauinformatik (see contact page).

Parts of MasterSim
MasterSim consists of three parts:

• MasterSimulator command line executable
• MasterSimulator user interface
• MasterSim programming library

Why use MasterSim?
First of all, it is free! The complete source code is available on its [SourceForge-project page] under an open source license. But there are many other reasons for using MasterSim:

• MasterSim is a cross-platform development, available for Windows, Linux and MacOS
• Support for script execution
• Support for several FMU instances within the same co-simulation scenario
• Highly configurable master algorithms
• Auto-connection feature in user interface (very useful when connecting FMUs with many matching inputs and outputs)
• Detailed simulation statistics inform about where in your co-simulation most of the simulation time spend; This is very helpful for performance tuning and selection of the optimal algorithm
• Support for parallel executed variation studies (several MasterSimulator processes), where the same FMU file is used concurrently in several simulations running in parallel

MasterSim Software Library

Library Functionality
With the library you can implement co-simulation functionality into your own simulation programs with little effort.

FMU Import functionality
The library supports extraction of FMU modelDescription.xml files, importing them into memory. This is all neatly encapsulated in co-simulation scenarios.

Master Algorithms
Included in the library source code are standard algorithms such as Gauss-Jacobi, Gauss-Seidel and Newton. The algorithms are implemented in master code without any work necessary (numerical parameters, perhaps).

Open-Source Co-Simulation Master developed at TU Dresden, Germany

http://mastersim.sourceforge.net
Co-Simulation: fixed-step size – stability problems
Co-Simulation: Fixed step size – Gauss-Seidel better
Co-Simulation: with adaptive step-size (error controlled)
Summary

1. Building energy simulation NANDRAD can now be used as FMI for ME/CS (version 2, with get/set state capability)

2. Use case 1 (within Modelica environment):
   a. NANDRAD-FMU substitutes simpler building models (when needed)
   b. Modelica-Wrapper-Models significantly simplifies connecting FMUs with hundreds of input/output variables

3. Use case 2 (Equipment FMU + Building FMU running in external Co-Simulation Master)
   a. Use of adapter models simplifies setup of co-simulation szenario
   b. Choice of master algorithm has great impact on stability/accuracy and performance
Co-Simulation between detailed building energy performance simulation and Modelica HVAC component models

Andreas Nicolai and Anne Paepcke

andreas.nicolai@tu-dresden.de

Internal functionality (only for discussion)
Functionality within Co-Simulation Master

1. Model acts as a Co-Simulation slave
2. Interacts with Master through get/set functions
3. Time integration is done within a communication step when Master calls doStep()
Master requests slave to integrate over a communication interval starting from a known solution.

Master $\rightarrow$ doStep(...)
Integrate step-by-step and adapt time step size based on local error control.
Integrate until communication interval has been completed

computed solutions after completed integration step

Integration time frame

Time steps

Master->doStep(...)

t0  t_commStart  t_commEnd

dt
Limit last time step to hit communication interval end time exactly

Limit time step to match interval end

computed solutions after completed integration step
Outputs may be requested at different times than time step or communication interval end time.

Integration time frame:
- $t_0$, $t_{\text{commStart}}$, $t_{\text{commEnd}}$

Time steps:
- $dt$

Outputs:
- $y$
- $\text{computed solutions after completed integration step}$
- $\text{output}_{dt}$
- Limit time step to match interval end

Master $\rightarrow$ doStep(...)

Legend:
- Master
- $t_{\text{commStart}}$
- $t_{\text{commEnd}}$
- $dt$
- $\text{output}_{dt}$
Outputs are interpolated in last completed integration step (can be several outputs)

Computed solutions after completed integration step

Interpolated solution at output time point

Integration time frame

Time steps

Outputs

Master->doStep(...)

Limit time step to match interval end
Things to consider...

1. Limit last time step
2. When interpolating model state backwards in last step, restore model state to end of communication interval before returning control to master

...otherwise pretty straightforward...
Iterative Master algorithms

- Gauss-Jacobi, Gauss-Seidel, Newton
- Master requests Slave to store its state
- Master can request Slave to store multiple states
- Master requests Slave to reset its state back to a previously stored state (roll-back)
- Related functions:
  - fmi2Serialize(), fmi2Deserialize(), fmi2SerializedFMUstateSize(), fmi2GetFMUState(), fmi2SetFMUState()
Getting and Setting FMU State

State is defined by:

- Integrator (conservation variables of ODEs)
- LES solver (for direct solvers factorized Jacobian)
- Jacobian matrix (unfactorized)
- Preconditioner (unfactorized and in case of ILU also factorized)
- Model state (e.g. hysteresis variables)
Getting and Setting FMU State

Fragmented memory structures
→ Direct copy not meaningful (and not fast)

→ Copy into sequential memory first

Implement first serialize and deserialize functionality
Getting FMU State via Serialization

Typical implementation of fmi2GetFMUState():

```c
1. // check if new alloc is needed
2. if (*FMUstate == NULL) {
3.   // alloc new memory
4.   fmi2FMUstate fmuMem = malloc(modelInstance->m_fmuStateSize);
5.   // remember this memory array
6.   modelInstance->m_fmuStates.insert(fmuMem);
7.   // store size of memory in first 8 bytes of fmu memory
8.   *(size_t*)(fmuMem) = modelInstance->m_fmuStateSize;
9.   // return newly created FMU mem
10.  *FMUstate = fmuMem;
11. }
12. else {
13.   // check if FMUstate is in list of stored FMU states
14.   if (modelInstance->m_fmuStates.find(*FMUstate) == modelInstance->m_fmuStates.end()) {
15.     modelInstance->logger(fmi2Error, "error", "fmi2GetFMUstate is called with invalid "
16.       "FMUstate (unknown or already released pointer).");
17.     return fmi2Error;
18.   }
19. }
20. // now copy FMU state into memory array
21. modelInstance->serializeFMUstate(*FMUstate);
22. return fmi2OK;
```
Serialization into sequential memory

Fragmented memory structures

Sequential memory

"Append"-pointer
Serialization into Sequential Memory

Fragmented memory structures

Sequential memory

"Append"-pointer
Serialization into Sequential Memory

Fragmented memory structures

Sequential memory

"Append"-pointer
Serialization into Sequential Memory

Fragmented memory structures

Sequential memory

"Append"-pointer
Serialization into Sequential Memory

Fragmented memory structures

Sequential memory

"Append"-pointer
Serialization into Sequential Memory

Maintainable code $\rightarrow$ Single code for:

- Size calculation
- Copy into sequential memory
- Copy back from sequential memory

For C compatibility (e.g. for CVODE integrator) use Macros
Serialization into Sequential Memory

1. #define SERIALIZE(type, storageDataPtr, value)\ 
2. {\ 
3.   *(type *)(storageDataPtr) = (value);\ 
4.   (storageDataPtr) = (char *)(storageDataPtr) + sizeof(type);\ 
5. } \ 
6. \ 
7. #define DESERIALIZE(type, storageDataPtr, value)\ 
8. {\ 
9.   (value) = *(type *)(storageDataPtr);\ 
10.  (storageDataPtr) = (char *)(storageDataPtr) + sizeof(type);\ 
11. }
Serialization into Sequential Memory

1. `#define SERIALIZE(type, storageDataPtr, value) \
2. { \
3. *(type *)(storageDataPtr) = (value); \
4. (storageDataPtr) = (char *)(storageDataPtr) + sizeof(type); \
5. } 

6. `#define DESERIALIZE(type, storageDataPtr, value) \
7. { \
8. (value) = *(type *)(storageDataPtr); \
9. (storageDataPtr) = (char *)(storageDataPtr) + sizeof(type); \
10. }` 

11. `#define CVODE_SERIALIZE_A(op, type, storageDataPtr, value, mem_size) \
12. \
13. switch (op) { \
14. case SUNDIALS_SERIALIZATION_OPERATION_SERIALIZE: \
15. SERIALIZE(type, storageDataPtr, value) \
16. break; \
17. case SUNDIALS_SERIALIZATION_OPERATION_DESERIALIZE: \
18. DESERIALIZE(type, storageDataPtr, value) \
19. break; \
20. case SUNDIALS_SERIALIZATION_OPERATION_SIZE: \
21. mem_size += sizeof(type); \
22. break; \
23. }`
Serialization into Sequential Memory

Single code for mapping memory depending on selected operation type (op):

1. CVODE_SERIALIZE_A(op, booleanType, *storageDataPtr, 
   cv_mem->cv_tstopset, memSize)
2. CVODE_SERIALIZE_A(op, realType, *storageDataPtr, 
   cv_mem->cv_tstop, memSize)
3. /* current order */
4. CVODE_SERIALIZE_A(op, int, *storageDataPtr, 
   cv_mem->cv_q, memSize)
5. /* order to be used on the next step = q-1, q, or q+1 */
6. CVODE_SERIALIZE_A(op, int, *storageDataPtr, 
   cv_mem->cv_qprime, memSize)
Aspects of creating a coupled building energy simulation

1. Adding FMI ME+CS interface to a stand-alone simulation engine
2. Implementing FMU v2 setting/getting-state capability
3. FMU generation process + resource handling
4. Application scenarios
5. Interface conventions
6. Usability (Modelica-Wrapper/Adaptors)
7. Demonstration cases (findings and experience gained)