Transformation of a building energy simulation tool with variable time step solver into an FMU for CoSimulation v.2 with serialization capabilities

Andreas Nicolai and Anne Paepcke

andreas.nicolai@tu-dresden.de

Overview

1. Some background on our tools
2. Principle functionality of variable time stepping integrators
3. Setting/Getting FMU States
4. Serialization into sequential memory
5. Output handling when iterating over FMUs
Simulation tools – NANDRAD and THERAKLES

Multi-zone building energy simulation model

Single-zone hygrothermal simulation model

Using the IBK Integration Framework (also used by DELPHIN 6):
Variable time-step, error controlled time integration with preconditioned Newton-Krylov-iteration for solution of sparse linear equation systems
Basic functionality of a variable time step solver

1. Initialization
2. While \( t < t_{\text{end}} \)
3. Do step: do while error test not passed
   4. Setup linear system
   5. Newton iteration: do while not converged
   6. Compute Newton right-hand-side
   7. Solve linear equation system
   8. Convergence test
   9. Do error test and adjust time step
10. Outputs: for all output times scheduled within last step
   11. Interpolate outputs
   12. Append outputs to file
13. End
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.

Integration time frame:

- `t0` to `t_commStart`
- `t_commStart` to `t_commEnd`

Master->doStep(...)
computed solutions after completed integration step

Integrate step-by-step and adapt time step size based on local error control

Master->doStep(...)
Transformation of a building energy simulation tool into an FMU for CoSimulation

Computation of solutions after completed integration step

Integrate until communication interval has been completed

Integration time frame

Time steps

Master->doStep(...)
Limit last time step to hit communication interval end time exactly

computed solutions after completed integration step

Integration time frame

Time steps

Limit time step to match interval end

Master->doStep(...)
Outputs may be requested at different times than time step or communication interval end time.

Integration time frame

Time steps

Outputs

Computed solutions after completed integration step

\[ y \]

Integration time frame:
- \( t_0 \)
- \( t_{\text{commStart}} \)
- \( t_{\text{commEnd}} \)

Time steps:
- \( dt \)

Outputs:
- \( \text{output}_dt \)

Master->doStep(...) Limit time step to match interval end
computed solutions after completed integration step

Outputs are interpolated in last completed integration step (can be several outputs)

Integration time frame

Time steps

Outputs

Limite time step to match interval end

Master->doStep(...)
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 straight forward ...
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():

```
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.}
21. // now copy FMU state into memory array
22. modelInstance->serializeFMUstate(*FMUstate);
23. 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 → 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. }

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. `\
7. `#define DESERIALIZE(type, storageDataPtr, value)\
8. {\
9.   (value) = *(type *)(storageDataPtr);\
10.  (storageDataPtr) = (char *)(storageDataPtr) + sizeof(type);\
11. }`\

1. `#define CVODE_SERIALIZE_A(op, type, storageDataPtr, value, mem_size)\
2.   switch (op) {\
3.   case SUNDIALS_SERIALIZATION_OPERATION_SERIALIZE:\
4.     SERIALIZE(type, storageDataPtr, value)\
5.     break;\
6.   case SUNDIALS_SERIALIZATION_OPERATION_DESERIALIZE:\
7.     DESERIALIZE(type, storageDataPtr, value)\
8.     break;\
9.   case SUNDIALS_SERIALIZATION_OPERATION_SIZE:\
10.    mem_size += sizeof(type);\
11.   break;\
12. }`
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)
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. 
4. /* current order */
5. CVODE_SERIALIZE_A(op, int, *storageDataPtr, 
   cv_mem->cv_q, memSize)
6. /* order to be used on the next step = q-1, q, or q+1 */
7. CVODE_SERIALIZE_A(op, int, *storageDataPtr, 
   cv_mem->cv_qprime, memSize)
Output file handling and iterating over FMUs

- FMI for CoSimulation v2 allows iterations
- Complex models may write many output files (impossible to pass all data as output)

Need to decide when to write outputs!

**Best practice:** whenever master signals in doStep() that *no resetting is done* prior to communication start:

```plaintext
noSetFMUStatePriorToCurrentPoint == fmi2True
```
Output file handling and iterating over FMUs

- FMI for CoSimulation v2 allows iterations
- Complex models may write many output files (impossible to pass all data as output)

Need to decide when to write outputs!

**Alternative method:** write binary output files and overwrite records in repeated communication intervals (may be less coding effort)
Issues when restoring FMU states

- FMI for CoSimulation v2 allows deserialization
- Simulation may be continued at stored state

What about output files written so far?

Good case/error handling needed:
- **Re-open files and continue writing (nominal case)**
- What if simulation is continued sometime in between a completed simulation \( \rightarrow \) truncate output files?
- What if output files are missing? What if gaps are in output files? ...
Summary – or is there an easier path to an FMU?

• No iterating master algorithms → no get/set FMU state needed
• No simulation restarting → no serialization functionality needed
• Communication intervals (~ 1 min) always shorter than model output steps (~ 30 min.. 1h) → no output backsetting needed
• Output steps even multiple of constant communication steps → no output interpolation needed

All of that: → EnergyPlus-FMU and similar
Transformation of a building energy simulation tool with variable time step solver into an FMU for CoSimulation v.2 with serialization capabilities

Andreas Nicolai and Anne Paepcke

andreas.nicolai@tu-dresden.de