Acta Polytechnica CTU Proceedings **26**:107–111, 2020
doi:10.14311/APP.2020.26.0107

available online at https://ojs.cvut.cz/ojs/index.php/app

MUPIF WORKFLOW EDITOR AND AUTOMATIC CODE GENERATOR

Stanislav Šulc*, Vít Šmilauer, Bořek Patzák

Czech Technical University in Prague, Faculty of Civil Engineering, Department of Mechanics, Thákurova 7, 166 29 Prague 6, Czech Republic

* corresponding author: stanislav.sulc@fsv.cvut.cz

Abstract. Integrating applications or codes into MuPIF API Model enables easy integration of such APIs into any workflow representing complex multiphysical simulation. This concept of MuPIF also enables automatic code generation of the computational code for given workflow structure. This article describes a 'workflow generator' tool for the code generation together with 'workflow editor' graphical interface for interactive definition of the workflow structure and the inner data dependencies. The usage is explained on a thermo-mechanical simulation.

Keywords: MuPIF, workflow editor, code generator.

1. Introduction

Complex physical tasks often need integration of very different models or solvers. Since there exist models solving the particular tasks, it is a very effective approach to connect the existing software tools together to solve the complex tasks. A MuPIF platform [1] is a Python [2] tool for such interconnection of several models together to compute coupled or linked multiphysical simulation. Note that in our terminology a model can be some software, application or code. Each model is covered in a Python API derived from MuPIF class Model, which has the following functions:

```
__init__() # Creates an instance of the class
    Model.
initialize() # Initializes the Model instance
    for a specific usecase.
solveStep() # Solves a computational step with
    defined length.
terminate() # Cleans all resources of the Model
    instance.
set() # Sets inputs of the model, e.g. MuPIF
    Field or Property.
get() # Returns outputs of the model, e.g.
    MuPIF Field or Property.
getCriticalTimeStep() # Returns maximum time
    step length
```

These seven functions suffice for full control of a basic computation.

The communication between models is realized with exchanging instances of MuPIF classes Field or Property. These instances can be obtained from a model by calling its get() method with appropriate parameters, and can be sent to a model by calling its set() method. Both Field and Property instances keep their values, but they also have attributes describing their type, units and some additional info for unique identification. Then, a Property can be sent to a Model and the Model decides what to do with it. This concept creates environment for easy interconnection of even completely different codes, which can also be based on different programming languages. The value of a Property is a tuple of any value types. A Field has a tuple of nodal or cell values for each entity, according to the field type, while each value can be a tuple. A Field is always linked with a MuPIF Mesh, which gives the Field space definition, interpolation and portability to tasks with different discretization.

Several models together create a workflow, which can be executed or added into another workflow, as it can behave like a model from outer scope, see Fig. 1. The MuPIF class Workflow is derived from class Model and is extended with the following function:

```
solve() # Calls solveStep() until the target
    time is reached.
```

which solves the whole simulation. A simplified Python code of a simulation with two models can look like this:

```
tm = thermal_model()
mm = mechanical_model()
tm.initialize('input_t.in', ...)
mm.initialize('input_m.in', ...)
while time<targetTime:
    tm.solveStep(...)
    tf = tm.get(...)
    mm.set(tf)
    mm.solveStep(...)
    ...
tm.terminate()
mm.terminate()
```

2. MuPIF workflow generator

The concept of MuPIF, especially the unified structure of the APIs allows us to automate generation of the computational steering code when the simulation

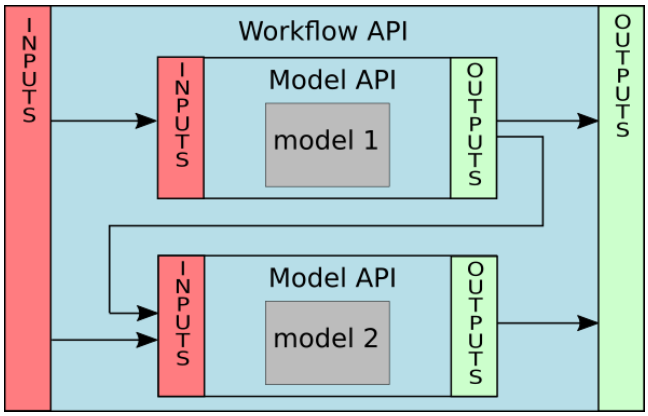

Figure 1. Example of a workflow structure.

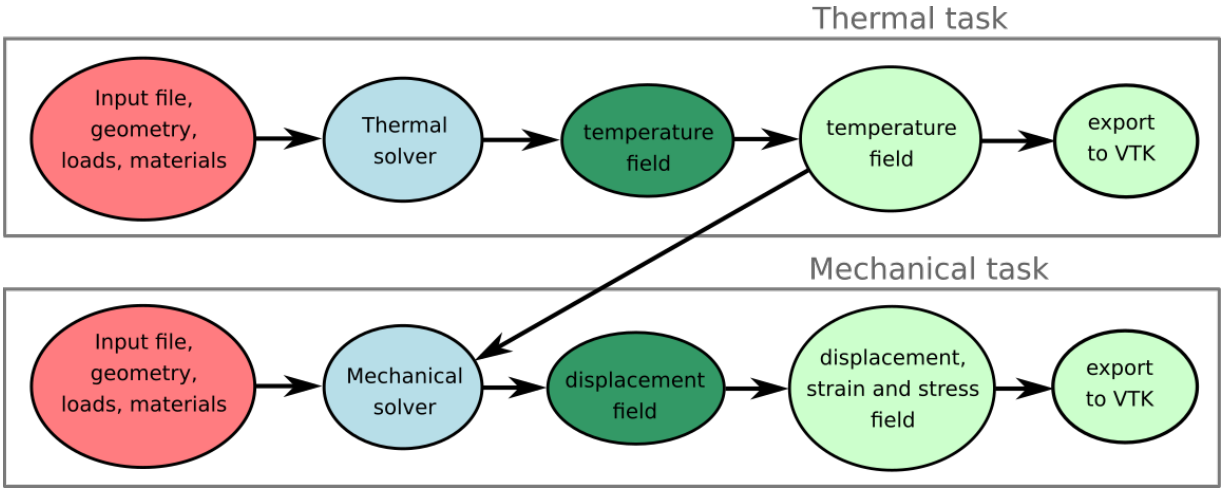

Figure 2. Workflow diagram of a thermo-mechanical simulation.

workflow structure is defined. Each API contains list of inputs and outputs which can be interconnected according to the data flow in the chain of models. This list also contains detailed description of each input or output.

The MuPIF workflow generator project [3] provides Python code environment to define the hierarchical structure of the simulation. It contains classes of blocks, where each block represents an item in the workflow diagram. There are blocks for following entities: MuPIF Model, MuPIF Property, MuPIF Physical quantity, time loop and the whole workflow. There are also some other blocks and also a possibility of implementing a new block by user.

Workflow and time loop blocks can contain a sequence of another blocks. All blocks can have input and output data slots. In case of Model block they represent the Model inputs and outputs. In case of a Property or Field the block has one output slot representing its value. Two data slots (output and input) can be connected, which represents the data exchange between models or setting of the input parameters.

When all the blocks and the data dependencies are defined, the code can be generated. The MuPIF Workflow code can be generated in two ways. The first option is that the workflow structure defines function solveStep(). In this case we call it a class workflow. The second option is that the workflow has a timeloop in itself and the structure defines the function solve(). In this case we call it an execution workflow.

In case of a class workflow there is a possibility of setting some inputs or outputs, just like a Model instance has some. It is realized using external data slots of the workflow. These slots are connected to some input or output slots inside the workflow, which transfers data from or to the outer scope, see this logic in Fig. 1. This is further explained in Section 4. The execution workflow cannot have such slots because the generated code behaves like a script to be run, not like a module to set some parameters and call its specific functions as in case of class workflow.

3. MuPIF workflow editor

MuPIF workflow editor [4] is a graphical user interface of the workflow generator for easy definition of the simulation structure according to its MODA [5] representation. Each item in the workflow is represented as a block with list of input slots on the left and list of output slots on the right. These slots can be interactively linked to define the dependencies among

particular models, see Fig. 4. The input slots can also be linked to some constant input data represented with blocks of MuPIF constant Property, Field or Physical quantity. There is also a block to define a time-loop, a block for including custom code or if-else block. When all the necessary dependencies are defined, the Python code of the simulation can be generated and saved or executed instantly.

The workflow editor provides saving of the workflow state into a JSON file to save the work and continue later. At the present time, it takes a part in the EU Horizon 2020 project and thus it contains a list of all available involved models inside, but any model based on MuPIF Model or Workflow can be loaded into this tool from a Python module. It automatically creates the structure and visual representation with the available data slots according to the inputs and outputs specification.

4. Example of use

The usage of the workflow editor and generator is described on a simple nonstationary thermo-mechanical simulation. This task is one of the MuPIF examples and can also be found in the workfloweditor github repository [4]. The geometry and boundary conditions are depicted in Fig. 3.

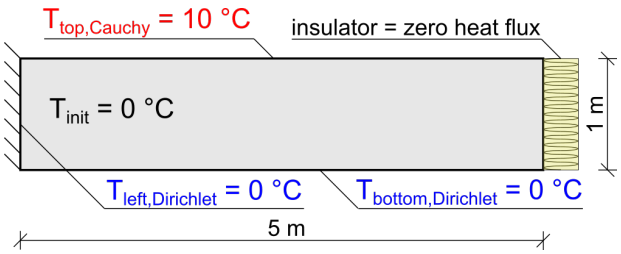

Figure 3. Schema of a thermo-mechanical task.

The material parameters are: conductivity 1.0 $W.m^{-1}.K^{-1}$, capacity 1.0 $J.Kg^{-1}.K^{-1}$, density 1.0 $Kg.m^{-2}$, Young's modulus $E=30\times10^9$ Pa, $\nu=0.25$ and coefficient of thermal expansion 1.2×10^{-6} K^{-1}. The mechanical response is caused only by the thermal expansion, there is no force load. The MODA workflow representation of the task is depicted in Fig. 2.

4.1. Execution workflow

We begin with the option of defining structure of the whole simulation in one execution workflow. The data flow from Fig. 1 is realized in the workflow editor as shown in Fig. 4. The thermal solver solves a nonstationary 2D thermal task on a rectangular domain. The boundary conditions on all edges can be set to the solver, see the thermal_nonstat model inputs in Fig. 4. The left and bottom edge temperature inputs are linked to the output of a block representing MuPIF constant temperature Property of zero value. The top edge temperature is linked analogically to a block representing constant temperature of 10 °C.

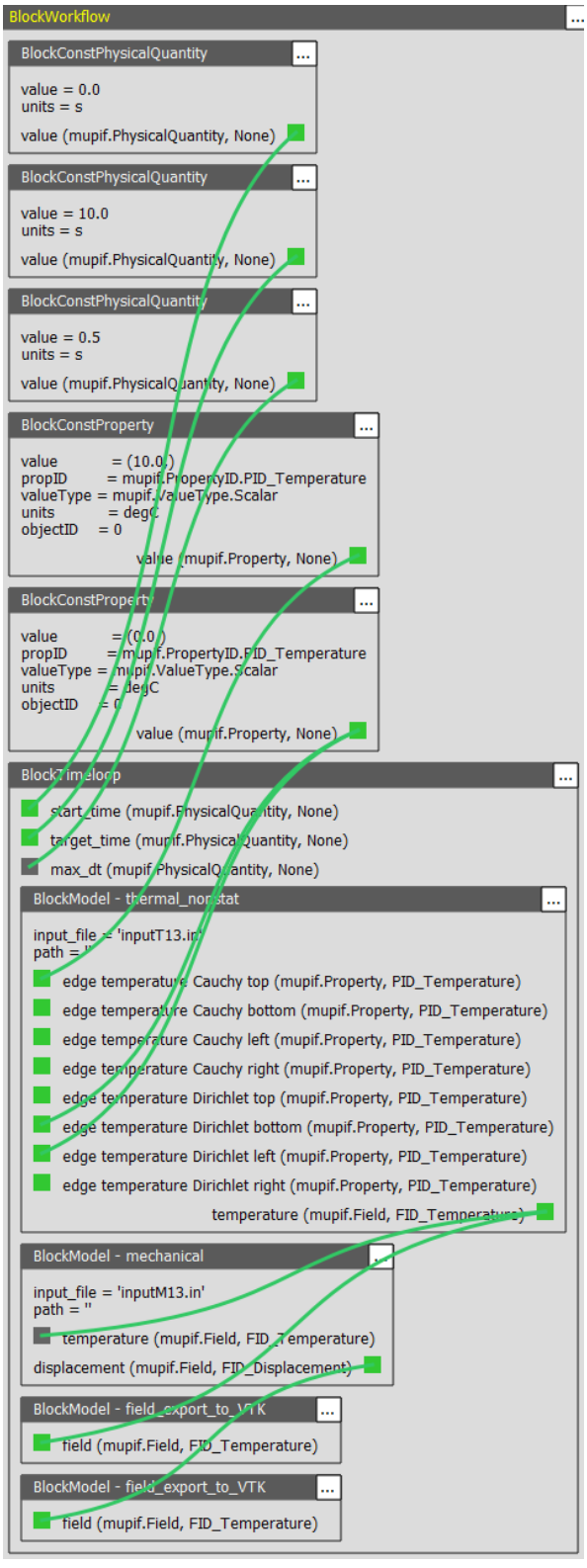

Figure 4. Execution workflow of the thermo-mechanical simulation.

The output temperature field slot of the thermal task is linked to the input temperature field slot of the mechanical task, which yields in the following code:

```
mm.set(tm.get(mupif.FID_Temperature,
    ts.getTime()))
```

where tm is the thermal model instance, mm is the mechanical model instance and ts is the computational time step.

The geometry of both thermal and mechanical task is defined via input file for each task.

The block BlockTimeLoop represents the simulation time loop and contains four model blocks. The last two blocks represent model named field_export_to_VTK, which is a simple tool for saving given MuPIF field into a file. It must have been integrated into the MuPIF class Model to enable such usage in the workflow editor. The MuPIF temperature and displacement fields are exported into files within each computational timestep. The first three blocks of the workflow represent MuPIF time Physical Quantity and define the run of the simulation time loop.

4.2. Class + Execution workflow

We continue with the second option, which begins with definition of a class workflow with the thermal and mechanical models, see Fig. 5.

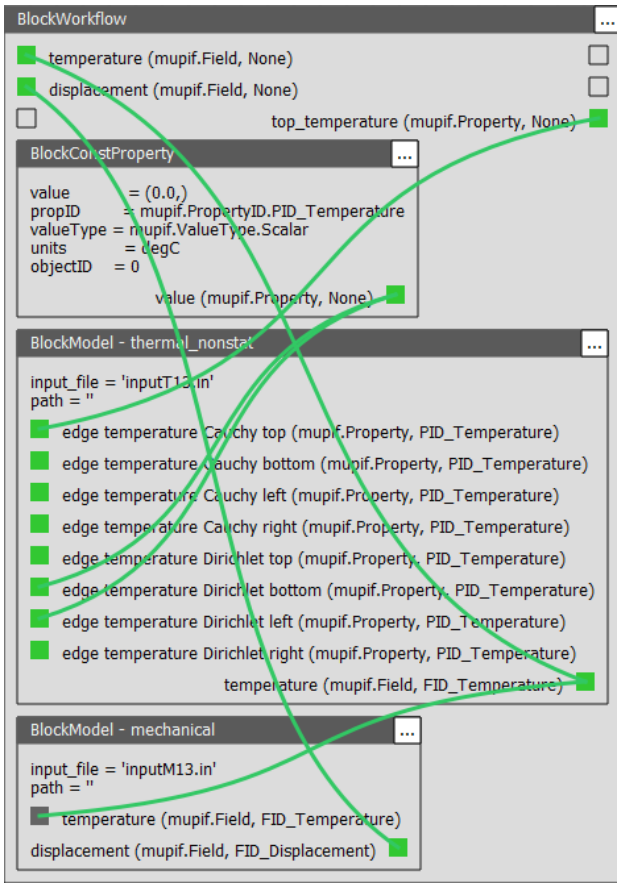

Figure 5. Class workflow of the thermo-mechanical simulation.

The data slots are connected analogically, but the top edge temperature slot is linked to the external input data slot and thus this data requirement will be satisfied from outer scope. Note that this input of the workflow is called 'external input data slot' although inside the workflow its value is represented with an output data slot to be linked with another block's input data slot. In case of external outputs it works analogically, see the two external output data slots, creating the outputs of this workflow.

Now we generate the class code and save it into a file. It contains a definition of a class derived from class Workflow. Then we create another workflow and insert a BlockTimeLoop into it, see Fig. 6.

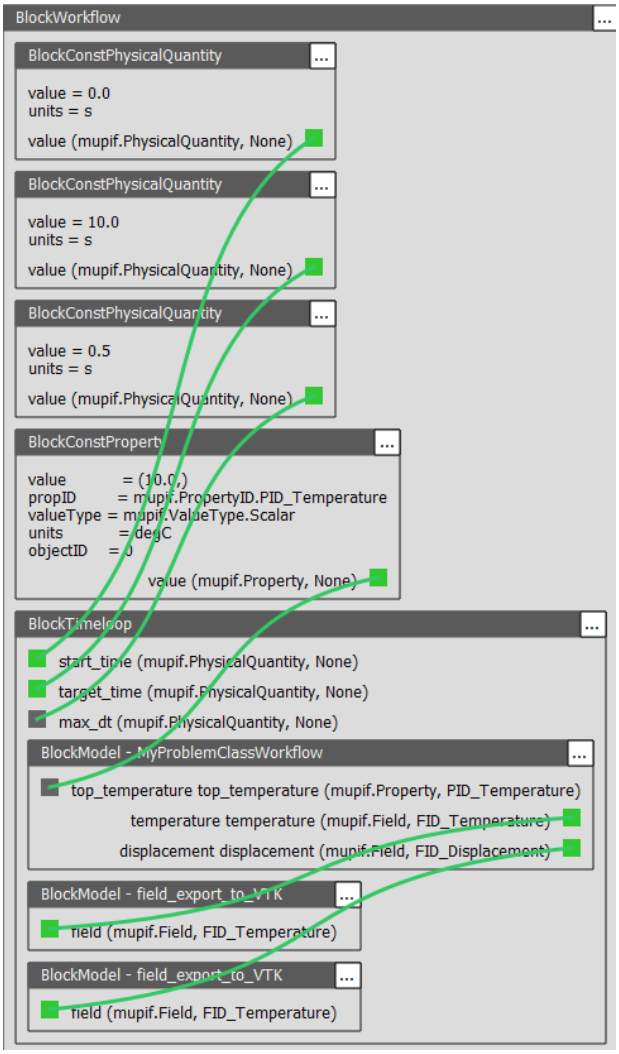

Figure 6. Execution workflow of the thermo-mechanical simulation with thermo-mechanical workflow used as a model.

This block contains the thermo-mechanical Workflow as a Model and illustrates the encapsulation of a complex simulation into a Workflow which provides easier usage from outer scope.

4.3. Task solution and results

Both approaches from Section 4.1 and Section 4.2 yield into an execution workflow for which we can generate the Python code. It can be saved into a file or executed instantly from the workflow editor. The execution will produce the VTK [6] files, which can be easily visualized, see Figs. 7 and 8.

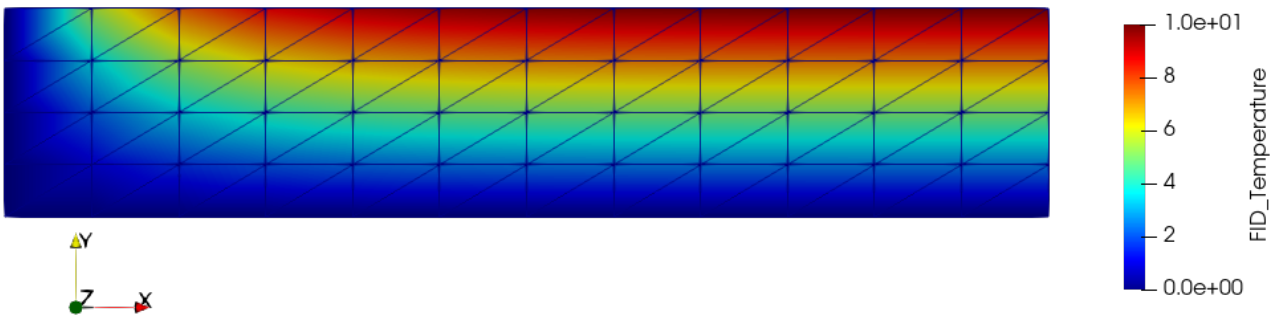

FIGURE 7. Resulting temperature at time=10 s.

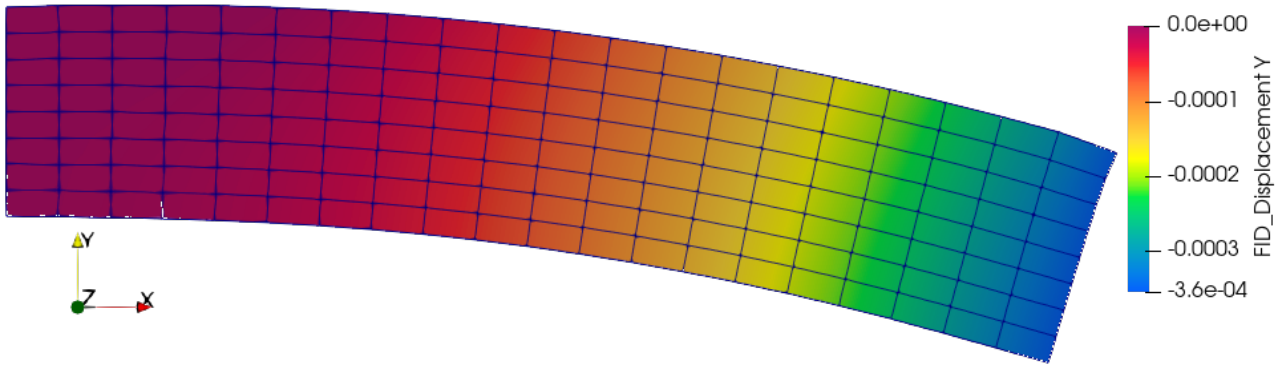

FIGURE 8. Resulting displacement at time=10 s.

5. CONCLUSIONS

This article describes a graphical software tool for user-friendly definition of a multiphysical simulation structure and the following generating of a Python code of the simulation. The usage of the workflow editor tool was explained on a thermo-mechanical task in two modes - The first one is the whole task defined in one execution workflow. The second one is defining a class workflow to be imported into an execution workflow as a model. The example is available in the workflow editor github repository.

In the future, a web-based version of the workflow editor will be created to provide easier access to this tool without any installation.

ACKNOWLEDGEMENTS

We gratefully acknowledge financial support from EU Horizon 2020 Project, contract number: 721105. We also gratefully acknowledge the financial support from Czech Technical University in Prague, grant number SGS19/032/OHK1/1T/11.

REFERENCES

[1] MuPIF https://github.com/mupif/mupif, 2019.

[2] Python https://www.python.org/.

[3] MuPIF Workflow Generator project repository. https://github.com/mupif/workflowgenerator, 2019.

[4] MuPIF Workflow Editor project repository. https://github.com/mupif/workfloweditor, 2019.

[5] The European Materials Modelling Council - MODA, homepage. https://emmc.info/moda/.

[6] VTK https://vtk.org/.