Supplementary Material for:

Streamlining Multi-scale Materials Modeling: The MUSICODE Low-code Approach for Simulation Workflows and Executable MODAs

Integrating Materials and Manufacturing Innovation

Dario Campagna[1*], Alan Del Piccolo[1], Konstantinos Kaklamanis[2], Stanislav Šulc[3], Mattia De Bernardi[1], Flavio Ellero[1], Saeideh F. Kalourazi[6], Klaus Reimann[5], Maria Andrea[2], Konstantinos Kordos[2], Michael Selzer[6], Britta Nestler[6], Dimitrios G. Papageorgiou[2], Aron Kneer[5], Davide Di Stefano[4], Bořek Patzák[3], Elefterios Lidorikis[2]

[1]Research and Development Department, ESTECO SPA, Trieste, 34149, Italy.
[2]Department of Materials Science and Engineering, University of Ioannina, Ioannina, 45110, Greece.
[3]Department of Mechanics, Faculty of Civil Engineering, Czech Technical University in Prague, Praha 6, 16000, Czech Republic.
[4]Ansys, Cambridge, CB1 7EG, United Kingdom.
[5]TinniT Technologies GmbH, Karlsruhe, 76131, Germany.
[6]Institute of Applied Materials—Microstructure Modelling and Simulation, Karlsruhe Institute of Technology, Karlsruhe, 76131, Germany.

*Corresponding author(s). E-mail(s): campagna@esteco.com.

A MUSICODE workflow is constructed by linking Model Tasks and defining the Data flow between them in the Workflow Editor. In the MuPIF environment, this executable workflow manages the data operations and executions of individual Model Tasks through standardized Model APIs [Patzák, B., Šulc, S., Šmilauer, V., MuPIF User Manual, https://mupif.readthedocs.io/en/master/index.html, 2024]. These APIs are designed to standardize model integration and data flow, I/O data serialization, and execution scheduling, while also supporting traceability and reusability. Model APIs are the single element requiring coding expertise. In MUSICODE, API development is streamlined by a graphical Model API generation tool whose input is the model metadata, where data mapping is achieved by standardized operations (using get/set services) on semantically structured data. In our demonstrator, the data represent an ensemble of molecules comprising a grain of material. This so-called GRAIN includes structural, physical, computational, and other required general information and is packaged in semantically enriched HDF5 data set having the MuPIF type definition *HeavyStruct*. GRAIN is the sole input required besides simulation-specific user-defined parameters.

As an example, we will present the API for the *DFTsolve* Model Task, which in this embodiment is implemented using the Gaussian solver. This module is intended to perform a standard DFT calculation (at B3LYP/6-31G(d,p) theory) on an arbitrary subset of n molecules, $n = 1, 2, \ldots, N$ where N is the total number of molecules in the GRAIN. The set of n molecules to be simulated is defined in the second input of the API, which is a string serialized as a single line containing a set of n integer quadruplets (ID, T1, T2, T3), where ID is the molecule number in the GRAIN and T1,T2,T3 = 0, ± 1 constitutes a normalized translation vector (i.e., in fractional coordinates) considering the cell structure and periodic boundary. In this example, GRAIN contains an ensemble of molecules relaxed at temperature T and ambient pressure, thus every molecule is at a different deformation state. We need to evaluate each molecule separately, so $n = 1$ and the *DFTsolve* Model will be invoked N times in parallel, one for each molecule in the GRAIN. The global list of executions (i.e., containing a total on N lines, one for each molecule) is prepared in the previous Model API *DFTpre*, while the loop architecture invoked in this workflow iterates over and serves each line to the *DFTsolve*, initiating N DFT instances to the HPC facility. We note here that if a different case scenario requires a different set of calculations, this would involve changes in the *DFTpre* module only, since the *DFTsolve* module is

generic enough to be reused at any scenario. Finally, the last input is a single string denoting a unique directory name to be used as temporary storage space for the I/O files. After execution is completed, the results will be processed to extract useful information (in our example these are the HOMO-LUMO values of each deformed molecule) and update GRAIN with them.

To summarize the *DFTsolve* API I/O metadata, the input contains:

- *GRAIN,* containing the molecular configuration (type: mupif.HeavyStruct)
- *DFT Iteration,* corresponding to a single molecule and their translation (type: mupif.String)
- *Workflow Name,* corresponding to the new execution directory (type: mupif.String)

While a single output item exists:

- *GRAIN*, updated with electronic states (type: mupif.HeavyStruct)

The series of operations performed by *DFTsolve* solveStep API method include:

- Create directory and copy appropriate files
- Extract molecular data and format them into appropriate input files (format: txt)
- Execute Gaussian solver to find molecular states
- Update the Assembly Grain container with HOMO-LUMO states

We next give an example of the Model API without going into full detail. We start with Model metadata documenting model characteristics, I/O data, solver and execution requirements. Here they are serialized in JSON format:

```
MD = {
    "Name": "DFTsolve",
    "ID": "DFTsolve",
    "Description": "Gaussian API implementation",
    "Version_date": "01/01/2025",
    "Geometry": "Cubic",
    "Boundary_conditions": "Periodic",
    "Inputs": [
        {
            "Name": "Iteration",
            "Type": "mupif.String",
            "Required": True,
            "Type_ID": "mupif.DataID.ID_None",
            "Units": "",
            "Obj_ID": "iteration_string",
            "ValueType": 'Scalar',
            "Set_at": "timestep"
        },
        {
            "Name": "Workflow name",
            "Type": "mupif.String",
            "Required": True,
            "Type_ID": "mupif.DataID.ID_None",
            "Units": "",
            "Obj_ID": "wname_string",
            "ValueType": 'Scalar',
            "Set_at": "timestep"
        },
        {
            "Name": "Grain",
            "Type": "mupif.HeavyStruct",
            "Required": True,
```

```
            "Type_ID": "mupif.DataID.ID_GrainState",
            "Units": "",
            "Obj_ID": "input_grain",
            "Set_at": "timestep"
        }
    ],
    "Outputs": [
        {
            "Name": "Grain",
            "Type": "mupif.HeavyStruct",
            "Type_ID": "mupif.DataID.ID_GrainState",
            "Units": "",
            "Obj_ID": "output_grain"
        },
    ],
    "Solver": {
        "Software": "Gaussian16",
        "Type": "Electronic",
        "Accuracy": "High",
        "Sensitivity": "High",
        "Complexity": "High",
        "Robustness": "High",
        "Estim_time_step_s": 1,
        "Estim_comp_time_s": 1000,
        "Estim_execution_cost_EUR": 0.01,
        "Estim_personnel_cost_EUR": 0.01,
        "Required_expertise": "Expert",
        "Language": "Fortran",
        "License": "xxxxxxxxxxxxxxx",
        "Creator": "Gaussian",
        "Version_date": "2022",
        "Documentation": "https://gaussian.com/g16main/"
    },
    "Physics": {
        "Type": "Electronic",
        "Entity": "Electron",
        "Equation": [Schrodinger],
        "Equation_quantities": [wavefunction, charge density]
        "Relation_description": [],
        "Relation_formulation": [],
        "Representation": " "
    },
    "Execution_settings": {
        "Type": "Distributed",
        "jobManName": "UOI.DFTsolve",
        "Class": "DFTsolve",
        "Module": "dftsolve"
    }
}
```

As an example of data operations performed in the API, we show a utility function for extracting atomic info and coordinates from the GRAIN and serialize them in appropriate .com files which are standard input files for the Gaussian. This is constructed using the set/get data services provided by MuPIF on the HeavyStruct data structure (details on the data schema and data services will appear in a follow-up publication):

```
def _extract_molecule_coordinates(self, molID, vector):

    grain = self.input_grain.openData(mode = 'readwrite')

    cell = grain.getStructure().getCellVectors().value
    molecules = grain.getParts()

    coordinates = ''
    for atom in molecules[molID].getParts():
        element = atom.getGeneralInfo().getElement()
        position = atom.getStructure().getPosition().value
        position += np.matmul(cell, vector)
        coordinates+='{:>3s}\t{:>13.6f}\t{:>13.6f}\t{:>13.6f}\n'
                                                .format(element, *position)

    self.input_grain.closeData()

    return coordinates
```

An example of a standard .com file created is shown below:

```
%NProcShared=4
%NProcLinda=1
%mem=100MW
%chk=FILE.chk
#B3LYP/6-31G(d,p) SCF=(tight,maxcycle=256) pop=full nosymm punch=mo iop(3/33=1) Test

 FILE

0 1
  S         1.226000    -8.030000    23.186000
  O        -3.373000    -6.968000    23.953000
  N        -0.411000   -12.369000    22.367000
  N        -4.400000   -13.631000    23.098000
  C        -3.562000    -8.192000    23.947000
  C        -2.583000    -9.231000    23.777000
  C        -3.167000   -10.581000    24.082000
  C        -4.428000   -10.239000    24.704000
  C        -4.710000    -8.869000    24.572000
  C        -5.902000    -8.358000    25.121000
  H        -6.227000    -7.323000    25.147000
  C        -6.858000    -9.244000    25.581000
  H        -7.830000    -8.897000    25.921000
  C        -6.592000   -10.618000    25.714000
  .
  .
  .
```

Similarly, updating the output GRAIN with the newly calculated HOMO-LUMO values for a molecule is also straightforward.

```
def _update_with_orbitalHL(self, molID, LUMO, HOMO):

    grain = self.output_grain.openData(mode = 'readwrite')
    molecule = grain.getParts()[molID]

    molecule.getProperties().getSiteEnergy().setOrbitalL(LUMO*mupif.U.eV)
    molecule.getProperties().getSiteEnergy().setOrbitalH(HOMO*mupif.U.eV)

    self.output_grain.closeData()
```