2023

AUGUSTO MOURA (DCS)

DCS | Industrizeile 35, 4020 Linz, Austria

Restricted

WP3- D3.4- SIMULATION PRE/POST-PROCESSING WRAPPERS

D3.4 – SIMULATION PRE/POST PRO-CESSING WRAPPERS

DOCUMENT CONTROL

Document Type	Other
Status	Final
Version	1.0
Responsible	Augusto Moura (DCS)
Author(s)	Giovanni Viciconte (DCS), Anna Matveeva (SISW), Sven Berger (Hereon), Valerio Lunardelli (AMAT), Fabrizio Buscemi (AMAT) Jesper Friis (SINTEF), Augusto Moura (DCS)
Release Date	2023-12-24

ABSTRACT

This deliverable is part of *Task 3.3 – Wrappers for third party visualization, pre- and post- processing tools based on OpenModel SDK*. The aim is to develop and integrate pre- and post-processing, and visualization tools which are required to visualize, prepare, and post process simulation workflows for the models supported in *Task 3.2-Wrappers for third party models and solvers based on OpenModel SDK* based on information passed via *D3.2-Wrappers Roadmap*. The wrappers are based on the ExecFlowSDK, which outlines the requirements for wrappers to connect to the OpenModel platform for workflow execution. For this deliverable, contributions are made by Success Stories 1, 2, 3 and 6. The remaining Success Stories do not have pre- and post-processing software applicable to this deliverable, therefore their contributions will be demonstrated in another deliverable.

CHANGE HISTORY

Version	Date	Comment
0.1	2023-07-19	First Draft
0.2	2023-08-28	Contribution from SS6
0.3	2023-12-04	Contribution from SS2
0.4	2023-12-06	Review and update

0.5	2023-12-12	SS2 complete and overall update
0.6	2023-12-13	Corrections to SS3 and SS4
0.7	2023-12-18	Review by project manager
0.8	2023-12-19	General review
0.9	2023-12-20	Contribution from SS1
1.0	2023-12-24	Final submitted

DISSEMINATION LEVEL

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

D3.4 – SIMULATION PRE/POST- PROCESSING WRAPPERS

1 INTRODUCTION

To be executed in the OpenModel platform, a software requires a wrapper. A wrapper, as the name suggests, operates by wrapping around the software, handling its required inputs and produced outputs and converting those to a common, ontologically described dataset. Wrappers for the software to be utilized in the OpenModel platform are created using the ExecFlowSDK, whose schematic is outlined in Figure 1 and detailed in D3.1. However, please note that since the D3.1 report, the ExecFlowSDK has been refactored and integrated into ExecFlow.

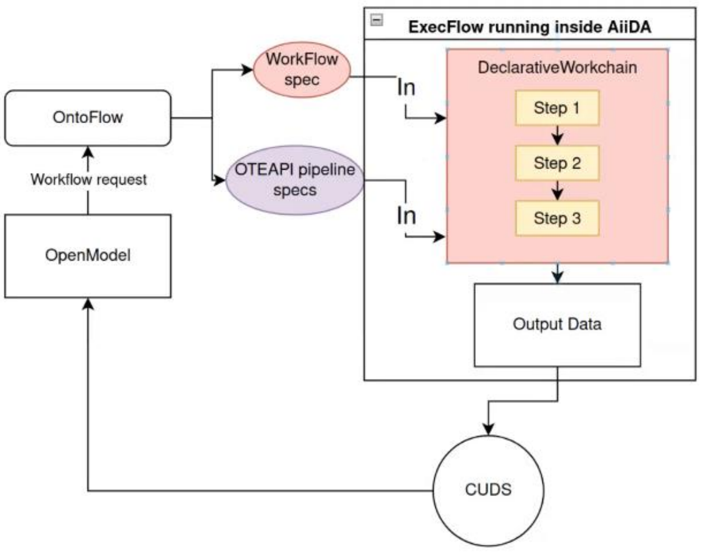


Figure 1: ExecFlow structure (from D3.1).

To be executed on the OpenModel platform, all success stories, and indeed any workflow, need to complete the requirements below:

1. Data documentation (Documentation of the input).
2. Workflow description using the declarative workchain.
3. Parsing of the output before storing into the database (Documentation of the output).

A common modelling workflow relies on one or more modelling software to execute the main modelling tasks. This usually requires some sort of pre-processed data for execution, e.g. as geometry files, meshes, etc, which frequently requires a dedicated pre-processing software to generate. Similarly, the raw output requires post-processing for visualization and analysis, a task which often employs dedicated post-processing software.

As stated above, each of these steps requires a wrapper to operate within OpenModel. A detailed description of modelling wrappers was provided in D3.3, but in summary, the wrapper for a modelling software semantically describes its inputs and outputs. Pre- and post-processing wrappers in the OpenModel OIP are a formal description of a pre- or post-processing software, including what input it takes, what output it provides and how it is executed together with a set of plugins for generating the input from a standardised internal representation and parsing the output to the standardised internal representation. The structures of pre- and post-processing wrappers are illustrated in Figure 2 and Figure 3, respectively.

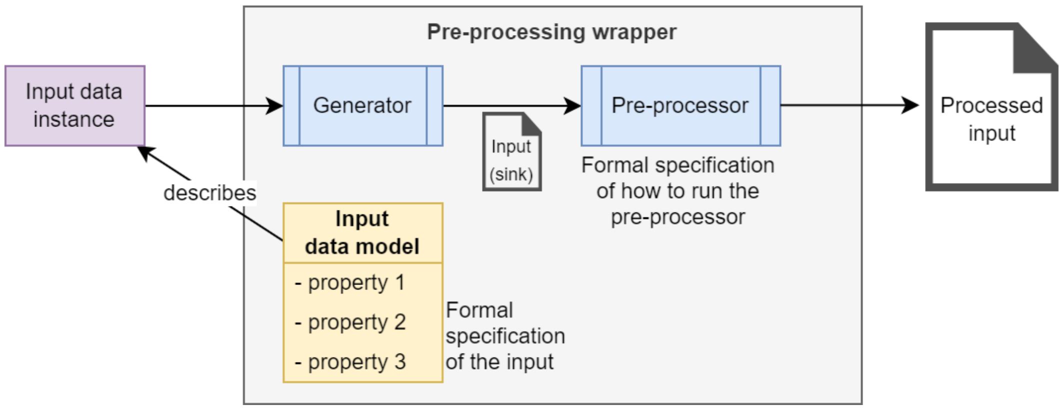


Figure 2: A pre-processing wrapper in OpenModel. The blue boxes represent the executable components, the generators that generates the pre-processor input, the pre-processor itself, and the parsers that parse the pre-processor output. The yellow box represents data models that describe the input. The input data instances are described by the corresponding data models but are not part of the wrapper itself.

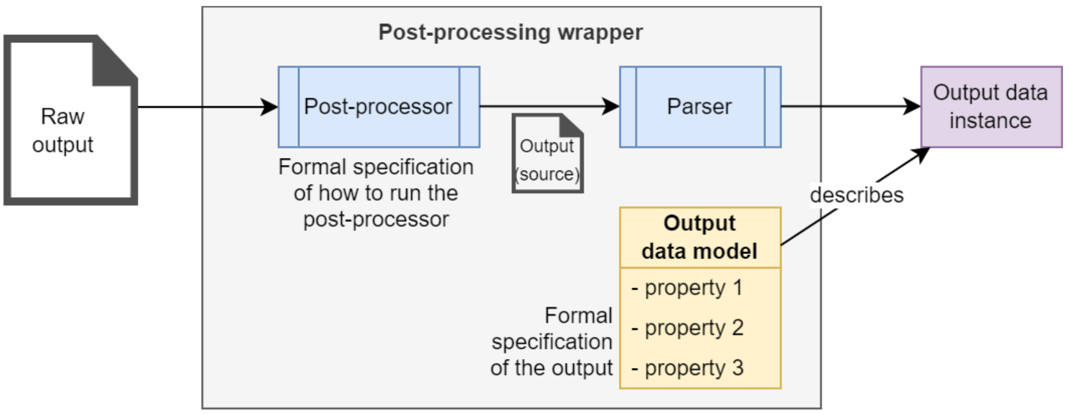


Figure 3: A post-processing wrapper in OpenModel. The blue boxes represent the executable components, the generators that generates the post-processor output, the post-processor itself, and the parsers that parse the post-processor output. The yellow box represents data models that describe the output. The input data instances are described by the corresponding data models but are not part of the wrapper itself.

DLite[1] is used to support data documentation and semantic mappings effectively. As a result, the datasets in OpenModel are described in a manner consistent with DLite data models (also called entities). This ensures a standardized and coherent representation of the data across all software used in the Success Stories and facilitates seamless integration within the OpenModel platform.

The basic requirements are:

- uri: Uniform Resource Identifier uniquely identifying the data model
- description: Describe the data being documented (see Figure 4).
- dimensions: Dimensions (i.e number of element) of the data.
- properties: All the properties associated with the data being documented.

```
"uri": "http://openmodel.eu/meta/0.1/MoltemplateInputPhysicalProperties",
"description": "Entity describing input variables for the simulation",
"dimensions": {
    "nrows": "Number of elements (i.e. number or rows)."
},
"properties": {
    "temperature": {
        "type": "float",
        "shape": ["nrows"],
        "description": "Temperature"
    },
    "pressure": {
        "type": "float",
        "shape": ["nrows"],
        "description": "Pressure."
    }
}
```

Figure 4: Example of data documentation. From the ExecFlow Demo presented in the Second Open OntoTrans Workshop in Bremen, Sep. 7, 2023.

1.1 SUCCESS STORIES

This section outlines the contributions from each Success Story to this deliverable. For this deliverable, contributions are provided from Success Stories 1, 2, 3, and 6, which rely on pre- and post-processing software that fit the description of this task. These will be detailed in the subsections below. The remaining Success Stories, namely 4, and 5, are not included in this report, but will be further elaborated on in deliverables D3.3, D3.7 and D3.6, respectively. The rationale is also outlined below.

The intent is that the combined work from the Success Stories will demonstrate all the capabilities of the platform. Figure 5 shows an overview of features required by the different pre- and post-processing wrappers and the OpenModel component responsible for providing that feature. It also connects the wrappers to the success stories where they are used.

[1]A light-weight data-centric framework for semantic interoperability; https://github.com/SINTEF/DLite

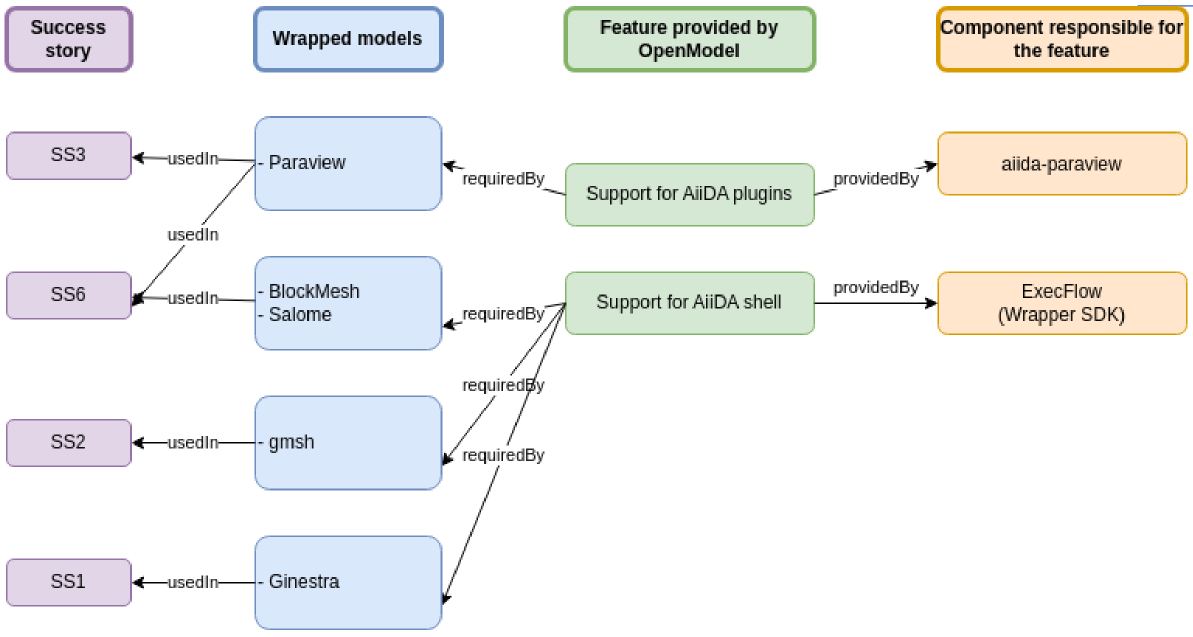


Figure 5: Overview over some selected features of the OpenModel OIP and what wrappers that requires them in what Success Story. The components that provide the selected features are also included.

1.1.1 SUCCESS STORY [1] – SYNAPTIC ELECTRONICS: FROM MATERIALS PROPERTIES TO NEXT-GENERATION MEMORY DEVICES (CNR, AMAT)

The Success story 1 (SS1) is based on the Applied Materials proprietary modelling and simulation software Ginestra®. The ability of the OpenModel platform to run workflows based on Ginestra® enables the user to improve device design and material exploration, linking the material properties to electrical device performances for synaptic electronics, and in general for all semiconductor applications. In the D3.3 it was described the modelling workflows of two fundamental material parameters: the band gap and the effective mass. In this deliverable we describe the development of Ginestra® to make it compliant with the OpenModel platform, together with the required pre-processing and post-processing wrappers for SS1. A schematic overview of the SS1 workflow is shown in Figure 6.

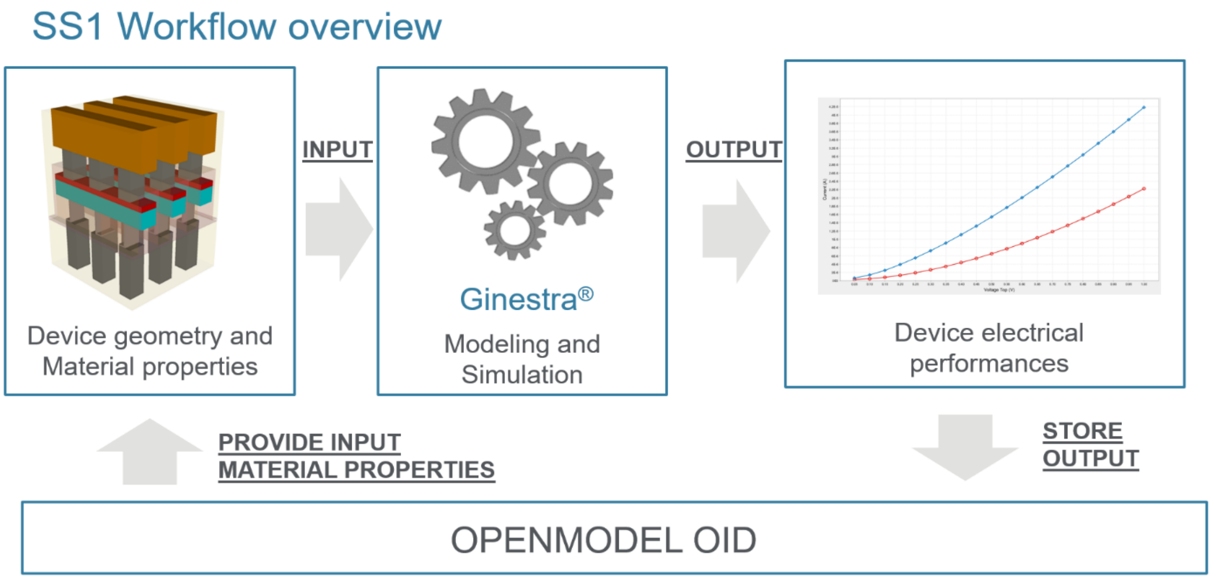

SS1 Workflow overview
Device geometry and Material properties
INPUT
Ginestra®
Modeling and Simulation
OUTPUT
Device electrical performances
PROVIDE INPUT MATERIAL PROPERTIES
STORE OUTPUT
OPENMODEL OID

Figure 6: SS1 Workflow overview

GINESTRA® AS A BLACK BOX:

Ginestra® is a stand-alone TCAD software that, thanks to its graphical user interface (GUI), permits to simulate electronic devices. Everything is done by means of the GUI, and the typical workflow is:

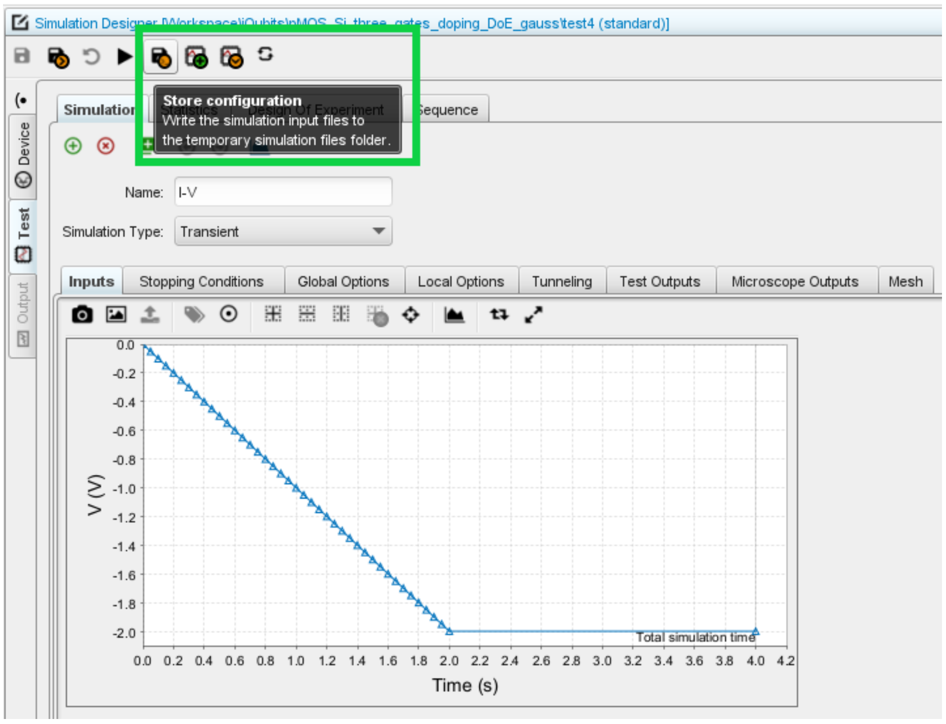
1. Device design
2. Electrical test definition
3. Simulation execution
4. Output data view and analysis.

As it is, Ginestra® is a close environment, not compatible with the OpenModel platform. To enable the interoperability with an external software, a new working condition, called Ginestra® as a black box (Gbb), has been developed.

The purpose of Gbb is to execute simulations on a pre-defined device and test, allowing the specification of some input parameters (geometrical or material properties) from an external dataset, as well as the retrieval of the output electrical characteristics.

This definition of Gbb makes it compatible with workflow automation. It allows to define a specific electrical test that can be integrated in a workflow that explores different configurations and materials, analyses the device performance, and identifies the best solutions. This corresponds exactly to the design of experiment (DoE) process widely applied in the R&D studies of the electrical industry.

- **Points 1 and 2** are complex tasks that still require the assistance of the GUI. They define the simulation, the dataset of input parameters and its outputs. In the GUI, these info are made available to Gbb by a new button called “Store configuration”. It saves all the input files in a simulation folder and defines the subset of the output data to be exported in a text file during Point 4.



Store configuration
Write the simulation input files to the temporary simulation files folder.

Name: I-V

Simulation Type: Transient

Figure 7: Gbb GUI

- **Point 3** has been decoupled from the GUI. In Gbb the simulation can be executed from the command line by the User or from an external driver as AiiDA. The dataset of the input parameters defined at Point 1,2 is exposed to the command line execution, as described later, and they can be defined or modified by the User/driver.
- **Point 4** Gbb stores, in addition to the proprietary Ginestra® output that can only be visualized within the GUI, the selected electrical characteristics. They are saved in a text format that can be easily accessed or imported in other data analysis software.

This new structure (Gbb) has been used to integrate Ginestra® within OpenModel and to develop the workflow of SS1 described in the following.

WORKFLOW SS1

More In details, these are the steps and key components/parameters we are going to describe in the following paragraphs.

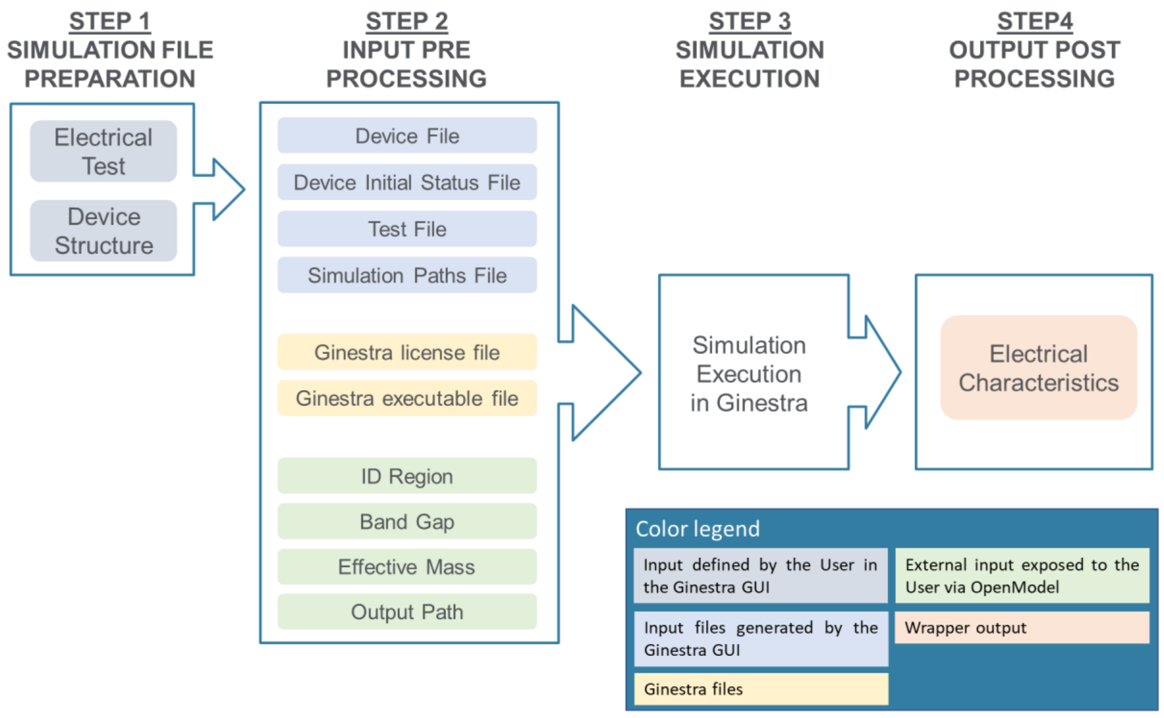


Figure 8: SS1 Ginestra simulation flow

Step 1: SIMULATION FILE PREPARATION.

The simulation input dataset of in Ginestra has two main components: the *device structure* and the *electrical test*. Both should be defined by the User through the Ginestra graphical user interface (GUI).

Device structure creation.

A User can design, by means of the GUI integrated geometry creator, the structure of complex electronic devices, defining several parameters like the shape and dimensions of the regions that compose the device, their number, their material etc. In addition, Ginestra offers an extensive library of standard structures (e.g. MIM, MOSFET, FinFET, GAA), that are stored in predefined “Template” geometries and can be shaped by the User (see Figure 8).

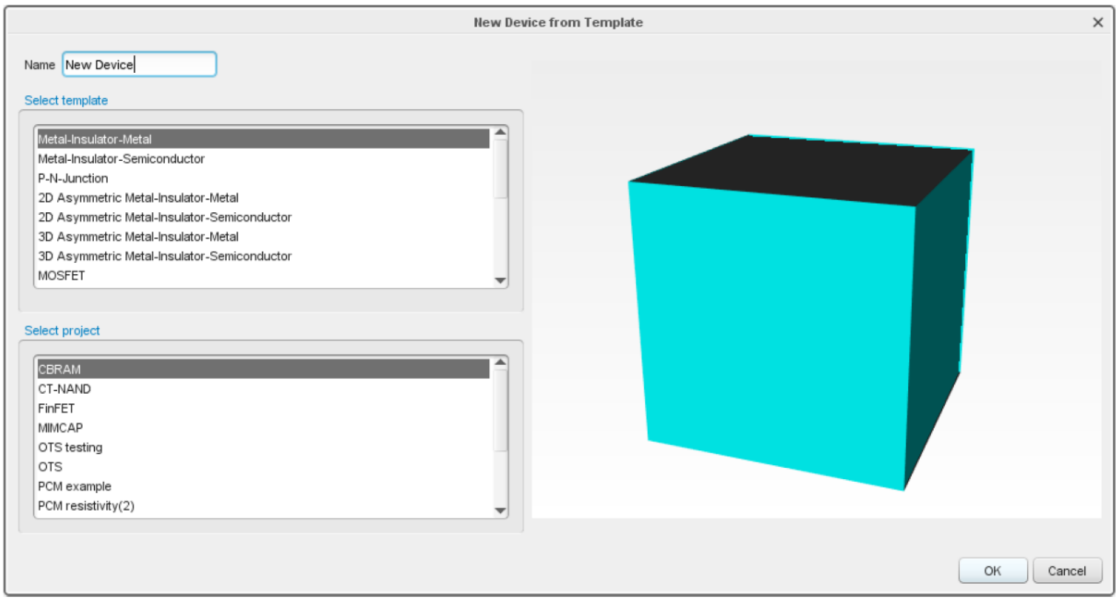

New Device from Template
Name
New Device
Select template
Metal-Insulator-Metal
Metal-Insulator-Semiconductor
P-N-Junction
2D Asymmetric Metal-Insulator-Metal
2D Asymmetric Metal-Insulator-Semiconductor
3D Asymmetric Metal-Insulator-Metal
3D Asymmetric Metal-Insulator-Semiconductor
MOSFET
Select project
CBRAM
CT-NAND
FinFET
MIMCAP
OTS testing
OTS
PCM example
PCM resistivity(2)
OK
Cancel

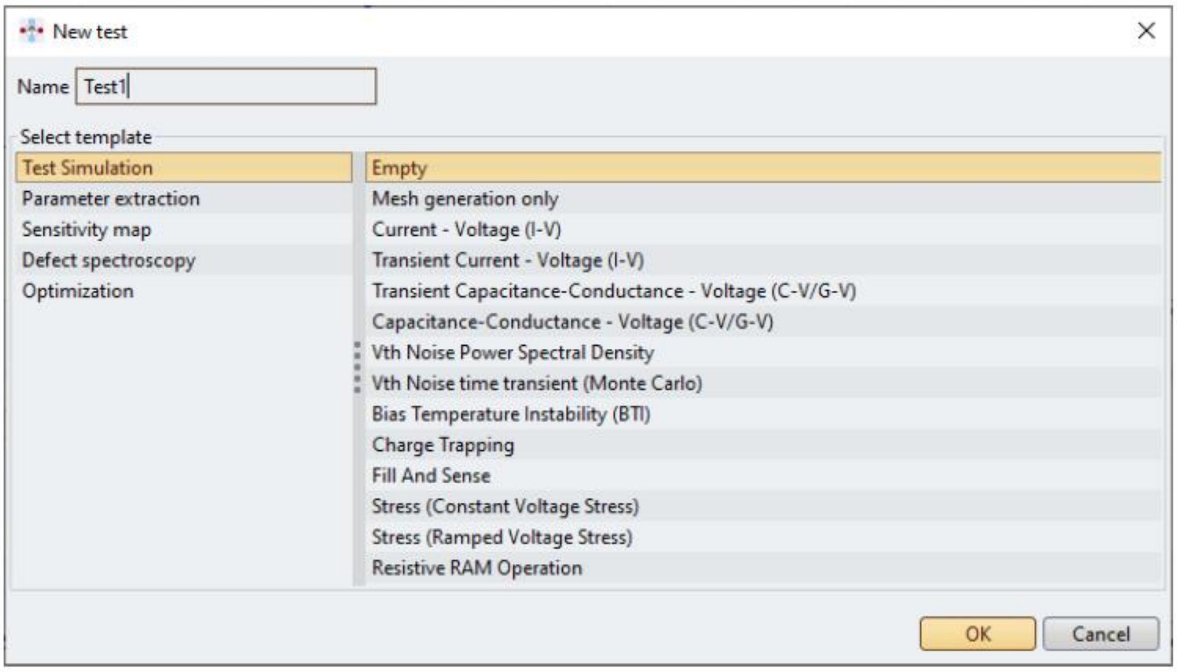
Figure 9: Ginestra GUI- new device from standard template library

The User can also import structures and meshes created by external mesh generators like GMSH [https://gmsh.info/].

Electrical test definition.

Once the device has been created, the User can define the electrical tests to be simulated. The test describes the working conditions of the device e.g., the applied electric signal, and the properties and output data to be computed. For the electrical tests also, it has been developed a library of predefined "Test Simulation" that can be modified by the User, as well as the possibility to define a test from scratch.

In SS1 we will run an IV test on a MOSFET. A current–voltage characteristic (or I–V) curve is a relationship, typically represented as a chart or graph, between the electric current through a device, and the corresponding voltage applied at its electrodes. In SS1 it will be computed the current at the drain contact of the MOSFET as function of the voltage difference between the drain and the source contacts.



New test

Name Test1

Select template

Test Simulation	Empty
Parameter extraction	Mesh generation only
Sensitivity map	Current - Voltage (I-V)
Defect spectroscopy	Transient Current - Voltage (I-V)
Optimization	Transient Capacitance-Conductance - Voltage (C-V/G-V)
	Capacitance-Conductance - Voltage (C-V/G-V)
	Vth Noise Power Spectral Density
	Vth Noise time transient (Monte Carlo)
	Bias Temperature Instability (BTI)
	Charge Trapping
	Fill And Sense
	Stress (Constant Voltage Stress)
	Stress (Ramped Voltage Stress)
	Resistive RAM Operation

Figure 10: Ginestra GUI- New test creation

Step 2: INPUT PRE-PROCESSING.

After preparing the Simulation files in the Ginestra GUI, during the second step, the pre-processing wrapper will collect the following inputs:

Ginestra files

- *ginestra-core-sim.exe*: the executable file of Ginestra Simulation Software which execute the simulation. In our SS1 we will process the Band Gap and Effective Mass which can be retrieved by the workflows defined in D.3.3 and evaluate the IV characteristics of a MOSFET.
- *ginestra.lic*: the license file necessary to run the Ginestra product. Ginestra is a commercial product own by Applied Materials.
- *input.ini*: a file that is automatically generated by Ginestra GUI at the end of Step 1.
- *input_parameter.ini*: a file specifying the material properties in different regions of the device.

Simulation files automatically generated by the Ginestra GUI at the end of Step 1

- *device.xml*: the file that describes the electrical device. It is automatically generated by Ginestra GUI at the end of Step 1.
- *state.xdf5*: the file that describes the initial state of the device. As an example, a memory cell may be in the fresh state (i.e. not programmed), or it can be programmed in its high or low resistance state. The file is automatically generated by Ginestra GUI at the end of Step 1.
- *test.xml*: the file that describes the electrical test. It is automatically generated by Ginestra GUI at the end of Step 1.
- *input.ini*: a file that collects the paths to the *device.xml*, *state.hdf5* and *test.xml* input files of the simulation, as well as to its output files.

Input dataset that can be set and modified by the User through Openmodel

- *ID Region*: identifier of the region where the user specific materials properties like the band gap and effective mass.
- *Band Gap:* key parameter to distinguishes metals from semiconductor and insulators and provides information about the electronic response of the material to external influences. It will be provided by the workflow described in the D3.3
- *Effective Mass*: core parameter needed to calculate the carrier density and density of states in semiconductors under the approximation of parabolic band dispersion. It will be provided by the workflow described in the D3.3 as well.
- *Output Path*: the path to the output file where the electrical characteristic is saved. In SS1 it is the I-V curve stored in a tab-separated values (TSV) text file.

Step 3: SIMULATION EXECUTION

If the input files are correctly prepared, the simulation can be computed by the Ginestra core simulation engine. The execution does not require the GUI and can be run in a terminal with the following command line:

```
ginestra-core-sim.exe -l “ginestra.lic” -i “input.ini” -p “input_parameter.ini”
```

The files *ginestra-core-sim.exe*, *ginestra.lic* and *input.ini*, have already been described and are directly available from the database. The file *input_parameter.ini* instead collects the parameters ID Region, Band Gap and Effective mass, and it is generated by the wrapper in the pre-processing phase.

Step 4: OUTPUT POST PROCESSING

The output generated by the Ginestra simulation engine can be post processed following two different approaches:

- *.hdf5 format:* the Hierarchical Data Format version 5 (HDF5), is an open source file format that supports large, complex, heterogeneous data. HDF5 uses a "file directory" like structure that allows the user to organize data within the file in many different structured ways, as the user might do with files on user workstation. Using this format the output can be postprocessed in the Ginestra GUI.

- *.tsv format:* a tab-separated values (TSV) file is a text format whose primary function is to store data in a table structure where each record in the table is recorded as one line of the text file. This format is documented in the OPENMODEL project in order to make the resulting data FAIR.

1.1.1.1 PREPROCESSING WRAPPER: GINESTRA WRAPPER

In this paragraph we will describe in detail the workflow of the pre-processing wrapper. Figure 11 shows the main wrapper actions:

1) **PARAMETER DATA PROCESSING**: in this first step, the 3 key parameter data (ID Region, Band Gap and Effective mass), that can be obtained from the existing data on the database o computed as a result of the modeling workflow described in D3.3, are combined by the wrapper in a unique file called *input_parameter.ini*
2) **INPUT COLLECTION:** the files *input.ini* and *input_parameters.ini* are copied to the execution folder. In the same folder, it is created a subfolder named *Inputs* where they are imported the files *device.xml*, *state.hdf5* and *test.xml*. In the same execution folder it is created the *Output* folder where the Ginestra simulation outputs will be collected.
3) **SIMULATION RUN:** the wrapper launch the simulation calling the ginestra-core-sim.exe engine.

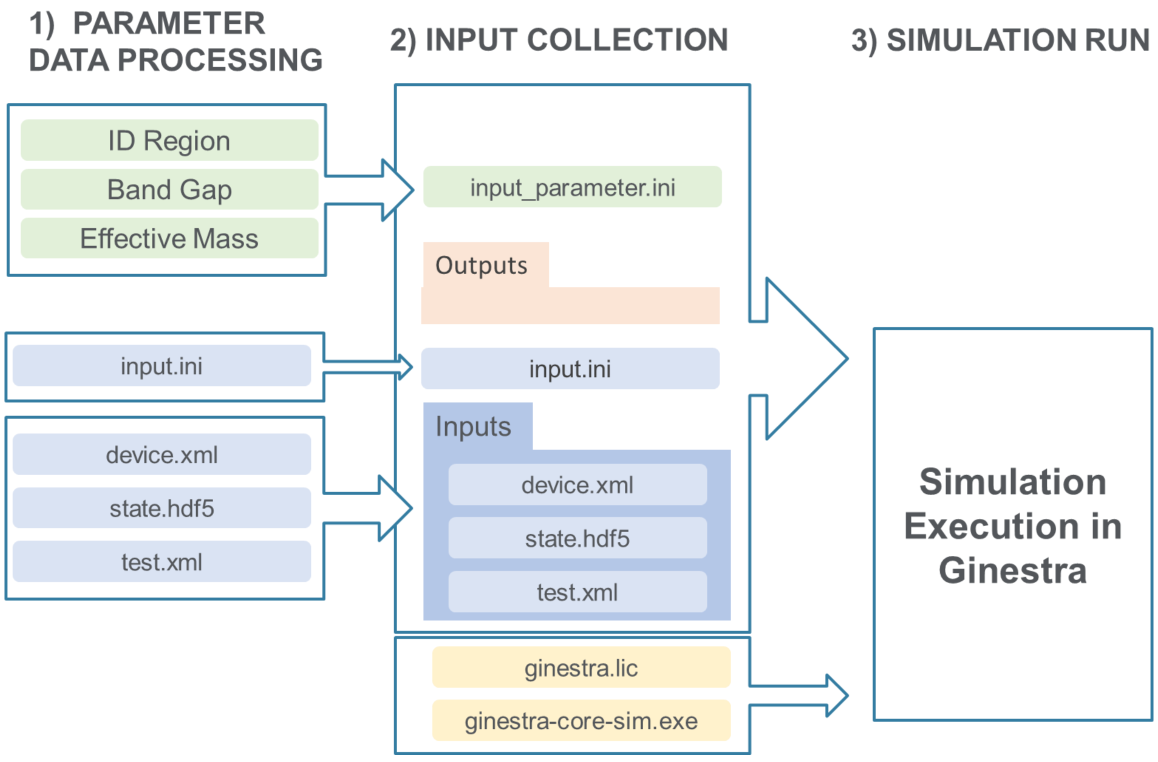


Figure 11: Pre-processing wrapper flow

1.1.1.1.1 DATA DOCUMENTATION

JSON (JavaScript Object Notation) is defined as a file format used in object-oriented programming that uses human-readable language, text, and syntax to store and communicate data objects between applications. In this paragraph we document the json input file for facilitating the data flow between OpenModel OIP and a simulation workflow:

```
{
  "uri": "http://ontotrans.eu/meta/0.1/ginestraSimulationInput",
  "description": "Entity for the simulation of an electrical device.",
  "dimensions": [],
  "properties": {
        "executable": {
            "type": "str",
            "description": "Path to the Ginestra executable file."
        },
        "licence": {
            "type": "str",
            "description": "Path to the Ginestra licence file."
        },
        "g_input": {
            "type": "str",
            "description": "Path to the Ginestra simulation input.ini file."
        },
        "id_region": {
            "type": "int",
            "description": "Identifier of the device region whose material parameters are
set by the wrapper."
        },
        "bandgap": {
            "type": "float",
            "unit": "eV",
            "description": "The material energy band gap."
        },
        "effective_mass": {
            "type": "float",
            "unit": "m0",
            "description": "The density of states effective mass of the conduction band, in
units of the electron mass m0."
        },
        "output": {
            "type": "string",
            "description": "Path to the output .tsv file."
        },
        "m_n_machines": {
            "type": "int",
            "description": "Number of computer nodes."
        },
        "m_n_mpiprocs_pm": {
            "type": "int",
            "description": "Number of MPI processes per node."     }
  }
}
```

1.1.1.1.1.1 INPUT

The input file for enabling the Ginestra simulation are listed in the following table:

Table 1: SS1 Input files

Name	Description	Type	Default values	Available options	Units
licence	Path to the license file	string	N/A	N/A	N/A
executable	Path to the executable file	string	N/A	N/A	N/A
input	Path to the simulation input.ini file	string	N/A	N/A	N/A

device	Path to the device.xml file	string	N/A	N/A	N/A
state	Path to the state.hdf5 file	string	N/A	N/A	N/A
test	Path to the test.xml file	string	N/A	N/A	N/A
id_region	Identifier of the device region where the bandgap and effective_mass parameters are provided	int	1	[0:N) N: number of device regions	Adim.
bandgap	Energy difference between the valence and conduction bands	float	1.16	[0.0:15.0]	eV
effective_mass	Effective mass that allow to describe the band carriers as a free-particle model	float	1	[0:3]	Units of the electron mass (m_0)
output	Path to the output TSV file	string	N/A	N/A	N/A
m_n_machines	Number of computer nodes	int	1	N/A	N/A
m_n_mpiprocs_pm	Number of MPI processes per node.	int	1	N/A	N/A

The YAML file with the wrapper for executing Ginestra in AiiDA follows:

```
# First implementation of the Ginestra task for computing I-V curves.
---
steps:
  # Reading the input dataset for Ginestra BB.
  - workflow: execflow.oteapipipeline
    inputs:
      pipeline:
        $ref: file:pipeline_device1.yaml
      from_cuds:
        - ginestra
        - input_parameters
    postprocess:
      - "{{ ctx.current.outputs.results['ginestra']|to_ctx('ginestra') }}"
      - "{{ ctx.current.outputs.results['input_parameters']|to_ctx('input_parameters') }}"

  - workflow: execflow.exec_wrapper
    inputs:
      shelljob:
        metadata:
          options:
            resources:
              num_machines: "{{ ctx.ginestra.m_n_machines }}"
              num_mpiprocs_per_machine: "{{ ctx.ginestra.m_n_mpiprocs_pm }}"
```

```
   command: "{{ ctx.ginestra.executable }}"
   arguments:
    - "-l"
    - "{{ ctx.ginestra.licence }}"
    - "-i"
    - "{{ ctx.ginestra.g_input }}"
    - "-p"
    - "{parameters}"
   files:
    parameters:
     filename: "input_parameters.in"
     node: "{{ ctx.input_parameters }}"
   outputs:
    - test.log
  postprocess:
   - "{{ ctx.current.outputs['test_log']|to_ctx('ginestra_output') }}"
...
```

1.1.1.1.1.2 OUTPUT

As mentioned above, in the SS1 the main output is the I-V characteristic, which defines the device operation within an electrical circuit. As its name suggests, it shows the relationship between the current flowing through the device and the applied voltage across its terminals.

Below an example of I-V curves plotted in Ginestra.

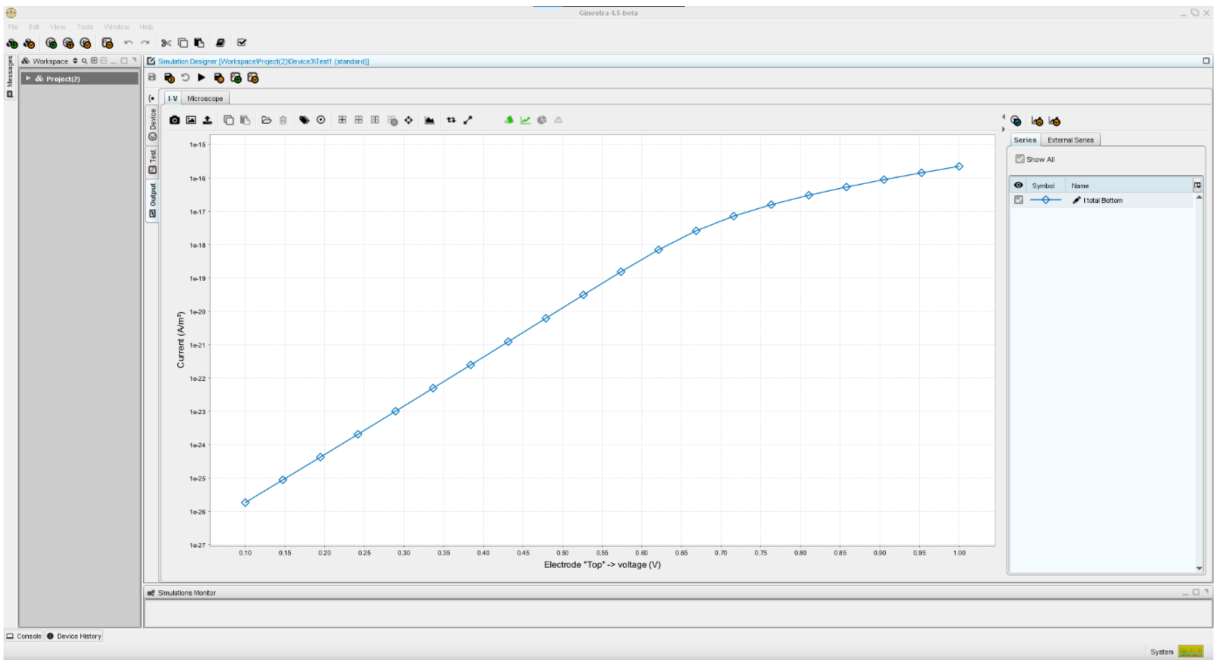


Figure 12: Ginestra GUI I-V output postprocessing

The user can choose different file format to post-process the I-V output:

- *.hdf5:* this is the standard format that is used by Ginestra to store the simulation outputs. HDF5 is a general purpose library and file format for storing scientific data. HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data. HDF5 is portable and is extensible, allowing applications to evolve in their use of HDF5. The HDF5 Technology suite includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format. In general, the HDF5 method significantly outperforms the CSV method on write speed, read speed, and storage size. The best performance in terms of timing occurs for the largest array size of 20,000,000 elements, where the write and read times for HDF5 are only 2.67% and 3.66% of the CSV times, respectively.

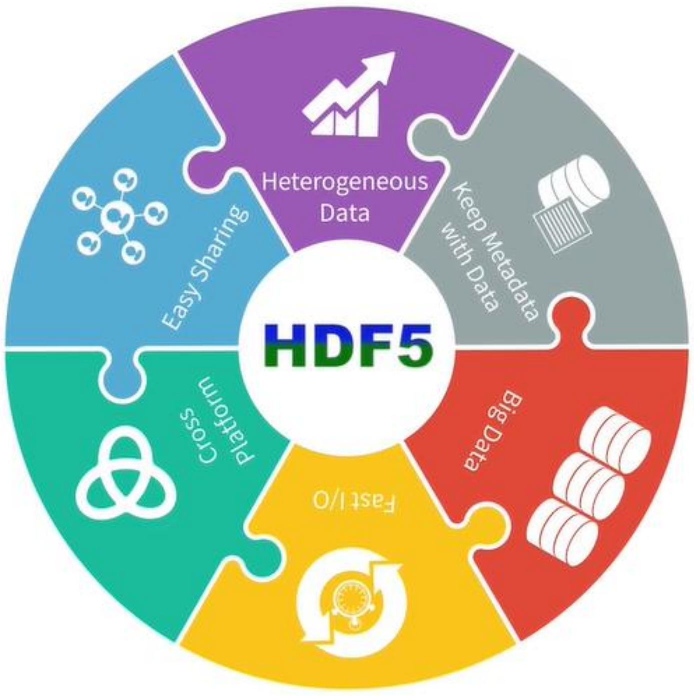


Figure 13: HDF5 main benefits. More information available at https://www.hdfgroup.org/solutions/hdf5/

- *.tsv:* Tab-separated values (TSV) is a simple, text-based file format for storing tabular data.[3] Records are separated by newlines, and values within a record are separated by tab characters. The TSV format is thus a delimiter-separated values format, similar to comma-separated values. TSV is a simple file format that is widely supported, so it is often used in data exchange to move tabular data between different computer programs that support the format. For example, a TSV file might be used to transfer information from a database to a spreadsheet.

1.1.1.1.2 EXECFLOW

The workflow that executes Ginestra as a standalone task uses a single data pipeline to create the *input_parameters.in* file, containing the material parameters to be used for the calculation of the device I-V curve. The workflow is executed by AiiDA with the command:

```
$ ./run_workflow.py workflow05.yaml
```

1.1.2 SUCCESS STORY [2] - COMPOSITE MANUFACTURING SIMULATION (SISW)

This use case involves virtual manufacturing for an automotive woven reinforced composite B-Pillar of a car-body as shown in Figure 14, using Resin Transfer Molding (RTM) as manufacturing technique.

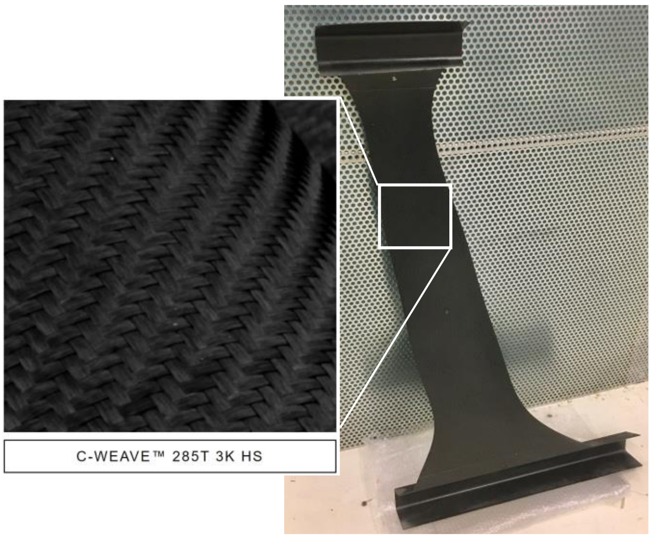
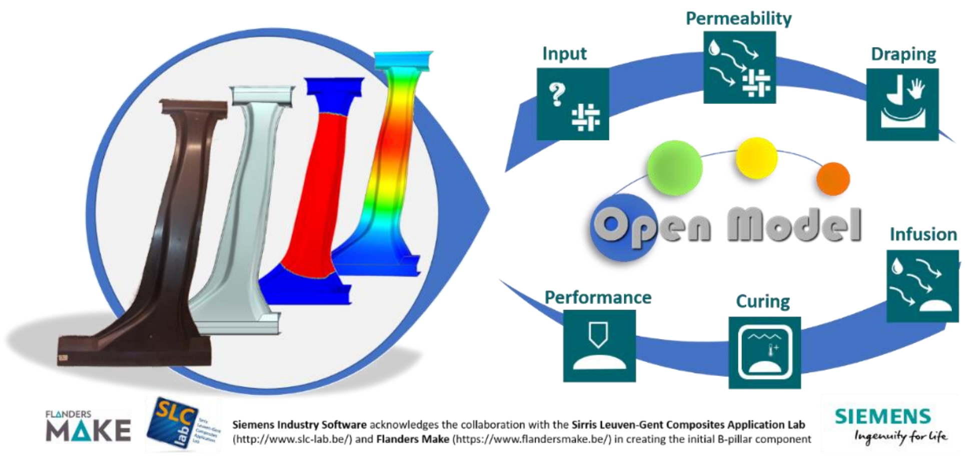


Figure 14: B-Pillar made of Chomarat C-WEAVE™ 285T 3K HS textile.

The RTM technique is a commonly used composite manufacturing process that involves injecting liquid resin into a preform (a dry reinforcement material such as fibers) placed in a closed mold under pressure. RTM allows manufacturing composite parts of complex shapes in a large quantity. The main manufacturing steps include draping of the dry reinforcement onto the mold, resin infusion, curing at elevated temperatures, following by a cooling down and releasing the part from the mold [1]. A multi-scale modelling and simulation workflow, proposed in this Success Story (Figure 15) is extended to performance simulation, including the effect of the manufacturing process (local fiber orientation, residual stresses, residual deformations, ...).



Siemens Industry Software acknowledges the collaboration with the Sirris Leuven-Gent Composites Application Lab (http://www.slc-lab.be/) and Flanders Make (https://www.flandersmake.be/) in creating the initial B-pillar component

Figure 15: Schematic overview of the Success Story 2.

An important parameter for the infusion simulation is the nominal permeability: the ability of a fluid (e.g., epoxy resin) to flow through a fibrous reinforcement when subject to an external force (pressure). An automated work-flow for saturated permeability (steady-state permeability when the reinforcement is fully saturated with a test liquid, and flow is in the steady state) computation has been implemented in Simcenter 3D as depicted in Figure 16 [2].

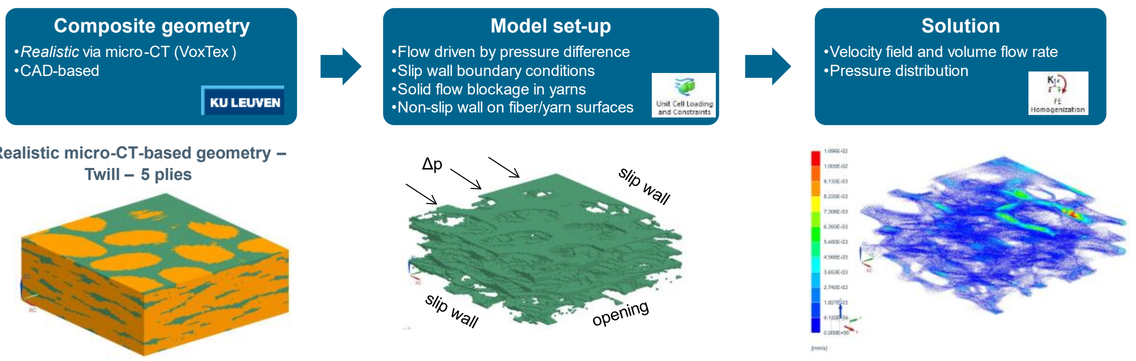

Composite geometry
•*Realistic* via micro-CT (VoxTex)
•CAD-based
KU LEUVEN
Model set-up
•Flow driven by pressure difference
•Slip wall boundary conditions
•Solid flow blockage in yarns
•Non-slip wall on fiber/yarn surfaces
Unit Cell Loading and Constraints
Solution
•Velocity field and volume flow rate
•Pressure distribution
FE Homogenization
Realistic micro-CT-based geometry – Twill – 5 plies
Δp
slip wall
slip wall
opening

Figure 16: Permeability computation workflow (shown on the example of Chomarat C-WEAVE™ 285T 3K HS).

To compute the saturated permeability of a composite unit cell, users are required to provide a meshed unit cell, which can be either microCT-based or CAD-based. The proposed workflow was validated using micro-CT based numerical validation [2]. MicroCT imaging is often impractical due to the destructive nature of sample preparation, limited availability of equipment, and the associated costs of both the equipment and software required for segmentation and finite element model reconstruction. As an alternative, CAD-based models generated based on material characteristics prove to be a viable option, and this approach will be the focus of this deliverable.

The following section 1.1.2.1 describes geometry and mesh generation of a woven composite unit cell using the open-source software Gmsh[2] based on a set of input data describing the textile. It includes a detailed description of coupling this pre-processing step with the OpenModel OIP for efficient input/output data flow. The resulting mesh is then available to the user for the computation in FE (finite element) simulation for further analysis. Deliverable *D3.7 M36 CAE Wrappers - Interface wrappers for OIP to integrated CAE platform, aiming to facilitate a broader connection with industrial end-users* will demonstrate how to import generated mesh into Simcenter 3D for permeability computations.

1.1.2.1 PREPROCESSING WRAPPER: GMSH

Gmsh is an open-source 3D FE mesh generator with a built-in CAD (Computer Aided Design) engine and post-processor. Its design goal is to provide a fast, light, and user-friendly meshing tool with parametric input and flexible visualization capabilities. Gmsh is built around four modules (geometry, mesh, solver and post-processing), which can be controlled with the graphical user interface, from the command line, using text files written in Gmsh's own scripting language (.geo files), or through the C++, C, Python, Julia and Fortran application programming interfaces.

[2] https://gmsh.info/

For this Success Story 2 on “Composite Manufacturing Simulation”, Simcenter Multimech was utilised to generate geometry and mesh of a woven composite unit cell, leveraging C++ application programming interface of Gmsh. Other types of composites such as unidirectional continuous fiber reinforced composites, particulate composites with/without voids, short fiber reinforced composites can be generated too, but their generation is out of scope for this Success Story.

1.1.2.1.1 DATA MODELS

JavaScript Object Notation (JSON[3]) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa). The data model describing the workflow input is a json file with the following structure, ensuring a smooth data flow between OpenModel OIP and a simulation workflow:

```
{
    "uri": "http://openmodel.eu/meta/0.1/MicrostructureInputGeometryMeshProperties",
    "description": "Entity for microstructure geometry and mesh generation.",
    "dimensions": {
    },
    "properties": {
        "weaveType": {
            "type": "string",
            "description": "Weave type."
        },
        "weaveHarness": {
            "type": "int",
            "description": "Number of harness to create the weave microstructure."
        },
        "weaveTowHeight": {
            "type": "float",
            "unit": "mm",
            "description": "Tows Height"
        },
       "weaveTowSpacing": {
            "type": "float",
            "unit": "mm",
            "description": "Spacing between tows."
        },
       "weaveTowWidth": {
            "type": "float",
            "unit": "mm",
            "description": "Tows Width."
        },
       "weaveNumberOfStacks": {
            "type": "int",
            "description": "Number of stacks."
        },
       "meshRefinement": {
            "type": "int",
            "description": "Mesh size: 0 very coarse, 5 very fine."
```

[3] https://developer.mozilla.org/en-US/docs/Glossary/JSON

```
                }
            }
        }
```

1.1.2.1.1.1 INPUT

The input parameters for Simcenter Multimech to generate woven composite unit cell geometry and mesh are listed in Table 4. These define the geometric constraints (weave type, weave harness, spacing between tows, tow widths and height, number of stacks) and the mesh size to discretise geometry.

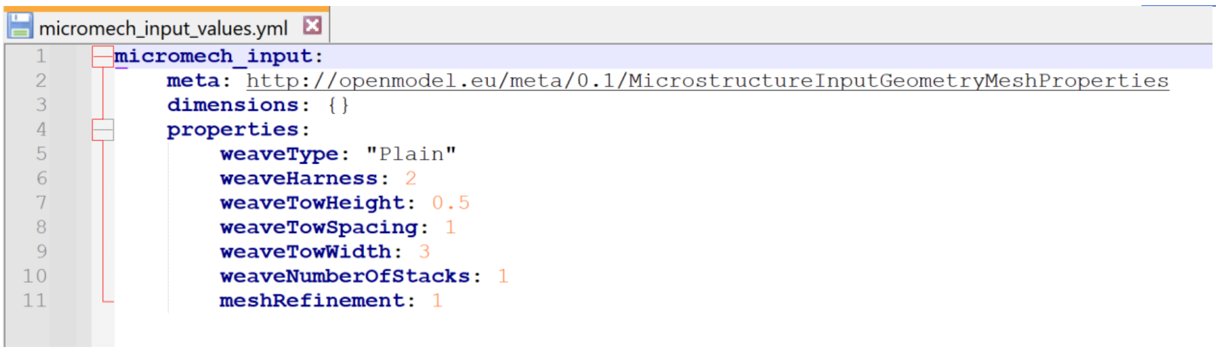
Table 2: Input parameters for Gmsh wrapper

Property name	Property description	Type	Default values	Available options	Units
weaveType	Weave Type	string	twill	plain/ twill/ satin	N/A
weaveHarness	Number of harness to create the weave microstructure	value	2	2,3,4,5	N/A
weave-TowHeight	Tow height	value	0.5	0.5-2	mm
weaveTowSpacing	Spacing between tows	value	1	0-10	mm
weave-TowWidth	Tow width	value	3		mm
weaveNumber-OfStacks	Number of stacks	value	1	0.1-10	N/A
meshRefinement	Mesh refinement	value	2 (coarse)	1-5 (1 = very coarse, 5 = very fine)	N/A

The values of input data are stored in .yaml format. YAML, which stands for "YAML Ain't Markup Language" or sometimes "Yet Another Markup Language," is a human-readable data serialization format. It is often used for configuration files and data exchange between languages with different data structures. YAML is designed to be easy to read and write for humans while being easily converted to and from data structures used by programming languages[4].

Figure 17 shows the Yaml file containing the input values used for the *Gmsh* wrapper.

[4] https://en.wikipedia.org/wiki/YAML



```yaml
micromech_input:
    meta: http://openmodel.eu/meta/0.1/MicrostructureInputGeometryMeshProperties
    dimensions: {}
    properties:
        weaveType: "Plain"
        weaveHarness: 2
        weaveTowHeight: 0.5
        weaveTowSpacing: 1
        weaveTowWidth: 3
        weaveNumberOfStacks: 1
        meshRefinement: 1
```

Figure 17: Yaml file for Gmsh wrapper input.

1.1.2.1.1.2 OUTPUT

The input data from Table 2: Input parameters for Gmsh wrapperTable 2 with the values from Figure 17 are processed by Simcenter Multimech to create geometry and mesh for Gmsh. The output is a text file .msh that contains nodes and elements and can be used as an input in FE simulations. For a more detailed information on file structure, please consult Gmsh documentation: https://gmsh.info/dev/doc/texinfo/gmsh.pdf

The example of .msh file for a woven unit cell is presented in Table 3

Table 3: Output example: .msh file

Identifier	Example	Description
$PhysicalNa-mes	$PhysicalNames 9 2 4 "mmExtBoundary_3_minus" 2 5 "mmExtBoundary_3_plus" 2 6 "mmExtBoundary_1_minus" 2 7 "mmExtBoundary_1_plus" 2 8 "mmExtBoundary_2_minus" 2 9 "mmExtBoundary_2_plus" 3 1 "Warp Tows" 3 2 "Weft Tows" 3 3 "Matrix"	Number of physical names Entry 1: physical dimension Entry 2: physical tag Entry 3: physical name mmExtBounday: groups of opposite faces that can be used by a user to define boundary condi-tions Warp/Weft tows: groups that defines the yarns Matrix: group that define the matrix
$Nodes	$Nodes 1262 1 0 -2 -0.25 2 0 -2 0.25 3 0 -3.5 0 4 0 -0.5 0	Total number of nodes Entry 1: node id Entry 2-4: Nodes coordinates

$Elements	6887
	1 2 2 6 200 380 45 373
	2 2 2 6 200 377 376 47
	3 2 2 6 200 373 374 753
		Total number of elements
		Entry 1: element ID
		Entry 2: element type
		Entry 3: reference to a tag 1 (yarns/matrix)
		Entry 4: reference to a tag 2 (mmEXTBoundary)
		Entry 5 – 8: Nodal connectivity

		Note: element type "2" - 3-node triangles element type "4" - 4-node tetrahedron

A reference to the output .msh file, that can be taken by the user for further FE analysis is provided in a MicrostructureInputGeometryMeshProperties with the following .json representation:

```
{
    "uri": "http://openmodel.eu/meta/0.1/MicrostructureInputGeometryMeshProper-
ties",
    "description": "Entity for generated microstructure mesh.",
    "dimensions": {
    },
    "properties": {
        "msh_file": {
            "type": "string",
            "description": ".msh file containing nodes and elements."
        }
    }
}
```

1.1.2.1.2 EXECFLOW

The *oteapi pipeline execflow*[5] utility is used to retrieve the available input data and make it available for the workflow execution.

```
   - workflow: execflow.oteapipipeline
     inputs:
       pipeline:
         $ref: file:micromech_pipeline_input.yml
       run_pipeline: pipe_input
       from_cuds:
         - micromech_aiida
     postprocess:
       - "{{ ctx.current.outputs.results['micromech_aiida']|to_ctx('micro-
mech_mic') }}"
```

The *oteapi pipeline execflow* utility executes the following pipeline, which is composed by two steps. In the first step, input data are read from the input yaml file. In the second, the DLite plugin *write_micromech* writes out the micromech.mic that will be used by Simcenter Multimech to generate the mesh .msh file.

```
version: 1
path:
```

[5] https://pypi.org/project/ExecFlowSDK/

```
strategies:
  - dataresource: load_data
    downloadUrl: "file:/// SS2wrapper_micromech/micromech_input_values.yml"
    mediaType: application/vnd.DLite-parse
    configuration:
      driver: yaml
      options: "mode=r"
      label: input

  - function: write_micromech_script
    functionType: application/vnd.DLite-generate
    configuration:
      driver: write_micromech
      location: /SS2wrapper_micromech/micromech.mic
      options: "mode=w"
      label: input
```

Once the input pipeline is executed, *micromech.exe* command from Simcenter Multimech is used to generate the mesh. A description of the workflow is given below. The output of this workflow is micromech.msh which contains the mesh generated by the software.

```
 - workflow:  execflow.exec_wrapper
     inputs:
       command: "/mnt/c/Program\\ Files/Siemens/SimcenterMultimech_2306/bin/micromech.exe"
       arguments:
         - "{script}"
         - "-b"
       files:
         script:
           filename: "micromech.mic"
           node: "{{ctx.micromech_mic}}"
           template: "/home/b4xpnt/workdir/SS2wrapper_micromech/micromech.mic"
       outputs:
         - micromech.msh
     postprocess:
       - "{{ ctx.current.outputs['micromech_msh']|to_ctx('micromech_msh') }}"
```

The last step describes the data retrieval. We retrieve the information from the context, in particular the uuid (unique universal into data node, to make it available to the rest of the workflow.

```
  - workflow: execflow.oteapipipeline
    inputs:
      pipeline:
        $ref: file:micromech_pipeline_output.yml
      run_pipeline: pipe_output
      to_cuds:
        - micromech_msh
      micromech_msh: "{{ctx.micromech_msh.uuid}}"
    postprocess:
      - "{{ ctx.current.outputs['collection_id']|to_ctx('collection_uuid') }}"
```

With the corresponding pipeline:

```
version: 1
```

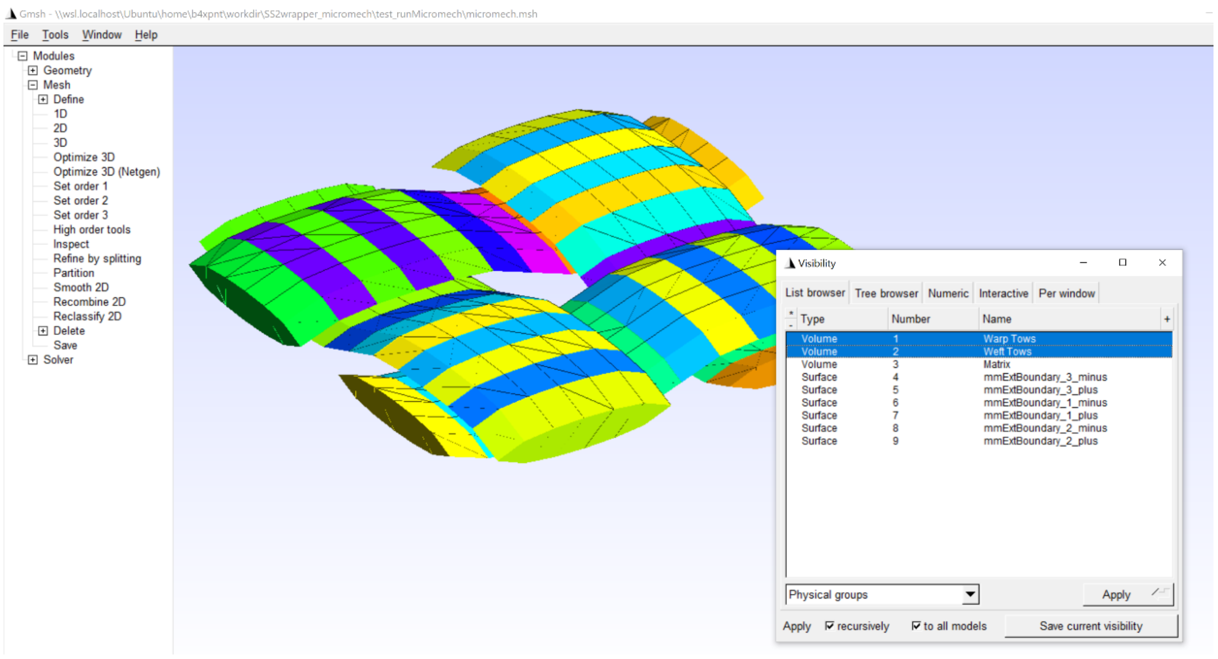
```
path:
strategies:
  - function: datanode2cuds
    functionType: aiidacuds/datanode2cuds
    configuration:
      names: to_cuds

  - function: saveMesh
    functionType: application/vnd.DLite-generate
    configuration:
      driver: save_msh
      location: ./mesh
      label: micromech_msh

pipelines:
  pipe_output: datanode2cuds | saveMesh
```

1.1.2.2 POSTPROCESSING .MSH FILE

Generated .msh file contains nodes and elements that the user can leverage in various FE simulations. Mesh visualisation can be done directly in Gmsh (Figure 18).



Type	Number	Name
Volume	1	Warp Tows
Volume	2	Weft Tows
Volume	3	Matrix
Surface	4	mmExtBoundary_3_minus
Surface	5	mmExtBoundary_3_plus
Surface	6	mmExtBoundary_1_minus
Surface	7	mmExtBoundary_1_plus
Surface	8	mmExtBoundary_2_minus
Surface	9	mmExtBoundary_2_plus

Figure 18: Gmsh GUI with visualised warp and weft tows from a woven unit cell.

Different types of woven unit cells are depicted in Figure 19.

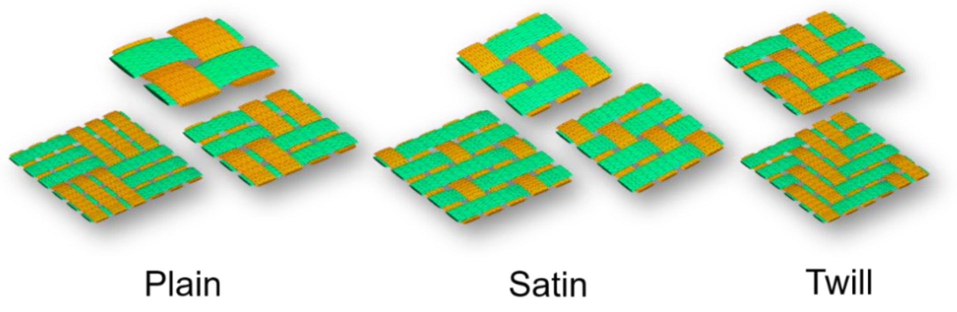


Figure 19: Visualisation of woven unit cell for different weave types.

All data models and workflow descriptions are available in the OpenModel Github project at https://github.com/H2020-OpenModel/SuccessStory2_wrappers.

1.1.3 SUCCESS STORY [3] – CIVIL ENGINEERING – REINFORCED CONCRETE (HYDRO, SINTEF, HEREON)

The workflow in Success Story 3 comprises multiple steps of varying complexity. The two most critical and challenging steps are the Finite Element simulation of the Al reinforced concrete beam and the Finite Element submodel of the layer deformation. These steps were described in detail in D3.3, including the required wrappers for postprocessing of the output files. Therefore, there is not much need for additional pre- and post-processing wrappers in this Success Story.

However, these two steps in the workflow do output mesh files in the XDMF format, which can be visualized with ParaView. A ParaView post-processing wrapper has been developed for Success Story 6 and is described in more detail in Section 1.1.6.

To read the new file format, the wrapper requires a data model for the XDMF file format used in Success Story 3, as shown in Figure 12. This necessitates only a slight modification of the wrapper to accommodate reading the new file. This demonstrates the potential for interoperability of the OpenModel platform, as the wrapper developed for one Success Story can be easily adapted for use in another Success Story and its software requirements.

```
{
  "uri": "http://openmodel.eu/meta/0.1/xdmfFile",
  "description": "Entity to represent a set of xdmf files",
  "dimensions": {
    "number_of_files": "Number of xdmf files."
  },
  "properties": {
    "files": {
      "type": "string",
      "dims": ["number_of_files"],
      "description": "List of xdmf file names."
    }
  }
}
```

Figure 20: Data model for a set of xdmf files for visualisation and post-processing with ParaView.

1.1.4 SUCCESS STORY [4] – METAL FORMING: RESOURCE EFFICIENT PROCESSING AND MANUFACTURING (HEREON)

Success Story 4 uses a generalizable machine learning (ML) approach that utilizes experimental data to predict material properties and guide metal forming process parameters. At the heart of this workflow is an ensemble of shallow artificial neural networks. These networks have been fine-tuned through hyperparameter optimization to accurately predict material behaviors. This method represents a significant shift from traditional linear regression models, enabling more effective use of experimental data and leading to improved predictive capabilities in metal forming. The ML-driven approach outlined in Success Story 4 offers a scalable solution, promising substantial advancements in optimizing resource allocation and refining manufacturing processes. In this instance, the workflow uses material properties and process parameters as both input and output, obviating the need for pre- and post-processing software. A detailed description of the modeling wrappers used in Success Story 4 is provided in Deliverable D3.3.

1.1.5 SUCCESS STORY [5] – DIGITAL POWDER TESTING (CMCL)

Success Story 5 leverages CMCL’s physics-based proprietary tools *k*inetics and SRM Engine Suite along with the advanced data analytics tool MoDS. Via introduction of an end-to-end digital workflow, all the internal interactions between these toolkits, and ensuing connection to the OpenModel OIP, is managed through the MoDS interface. Dealing with the execution of the Success Story in this way, would simplify the workflow as far as the OIP is concerned, and requires a dedicated data analytics and modelling wrapper for MoDS toolkit. Such a wrapper is currently under development and will be reported as a separate deliverable (D3.6) in the project. To avoid duplication, further details about this wrapper will only be covered in that deliverable.

1.1.6 SUCCESS STORY [6] – FUEL CELL MODELLING (DCS, TOYOTA, HEREON)

Success Story 6 involves the modelling of hydrogen fuel cells for use in the automotive industry. The fuel cell works by supplying hydrogen from one end and oxygen from the other, with the electrolysis reaction between them happening in the catalyst. The reaction produces water, which must be removed through the same porous media through which oxygen is supplied. These are, therefore, competing processes, and it is important to determine the flow properties through the porous media to establish the performance of the fuel cell.

The modelling approach for this process is a complex one: it requires modelling flow through porous media of different scales, while capturing the electrical conductivity through the porous material.

The model contains three steps:

- Calibration: in this step, the electrical properties of the porous material are calibrated from experimental measurements of the overall fuel cell resistivity.
- Generation of the porous media: the porous media itself is created using the Discrete Element Method (DEM) with the software Aspherix.
- Flow modelling: the flow through the porous media is simulated with a coupled simulation using Computational Fluid Dynamics and DEM. This calculates the flow of oxygen and water vapour through the porous media, while simultaneously simulating the electrical current through the material. This step uses the software Aspherix and CFDEMcoupling.

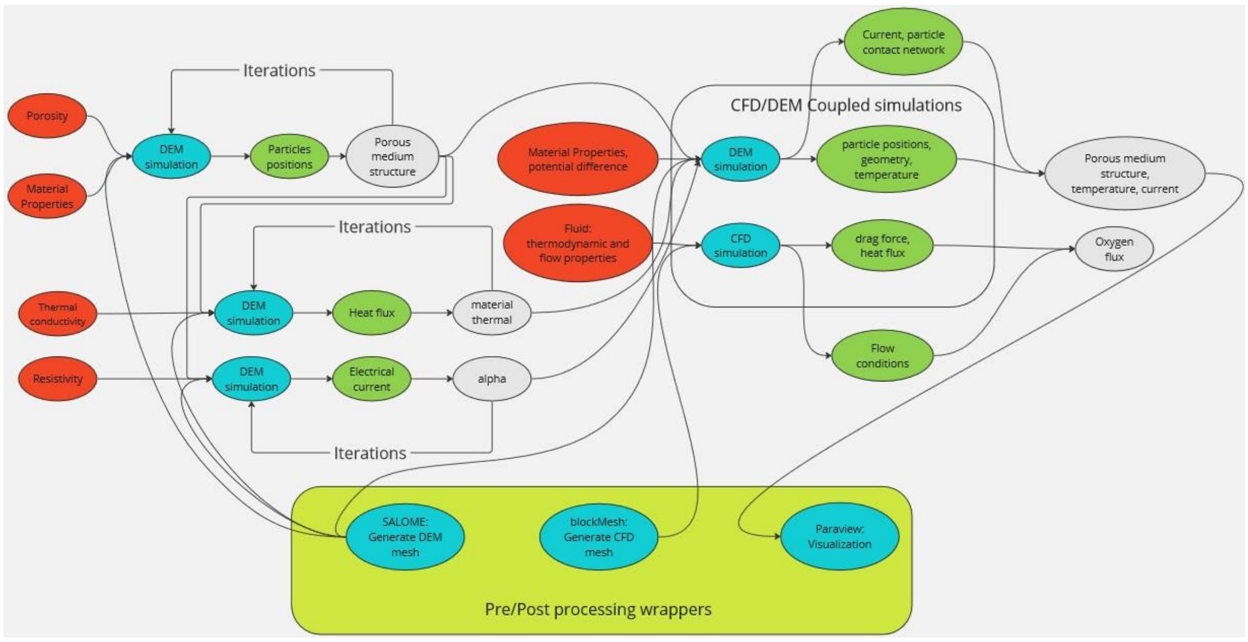


Figure 21: Success story workflow description.

As with any modelling workflow, this setup requires some preprocessing, in particular the mesh generation for the CFD calculations, and postprocessing for visualization of the results. For Success Story 6, mesh generation is performed with *blockMesh*, while postprocessing and visualization are done with ParaView. Wrappers have been developed for both applications and are discussed below. On top of this, a wrapper has been developed for the pre-processor SALOME, which allows for geometry and mesh generation.

PRE-PROCESSING WRAPPER: BLOCKMESH

blockMesh is a utility distributed with OpenFOAM. It enables the creation of structured meshes featuring parametric design, grading options, and curved edges.

When executed, *blockMesh* reads a dictionary, which specifies the geometry and discretization, processes the mesh specifications, and outputs mesh data into:

- points,
- faces,
- cells,
- boundaries

within the same directory.

The fundamental concept of *blockMesh* involves dividing the geometric domain into a series of three-dimensional hexahedral blocks. These blocks' edges can take the form of straight lines, arcs, or splines. By defining the desired cell count in each block direction, blockMesh can generate the corresponding mesh data.

1.1.6.1.1 INPUT

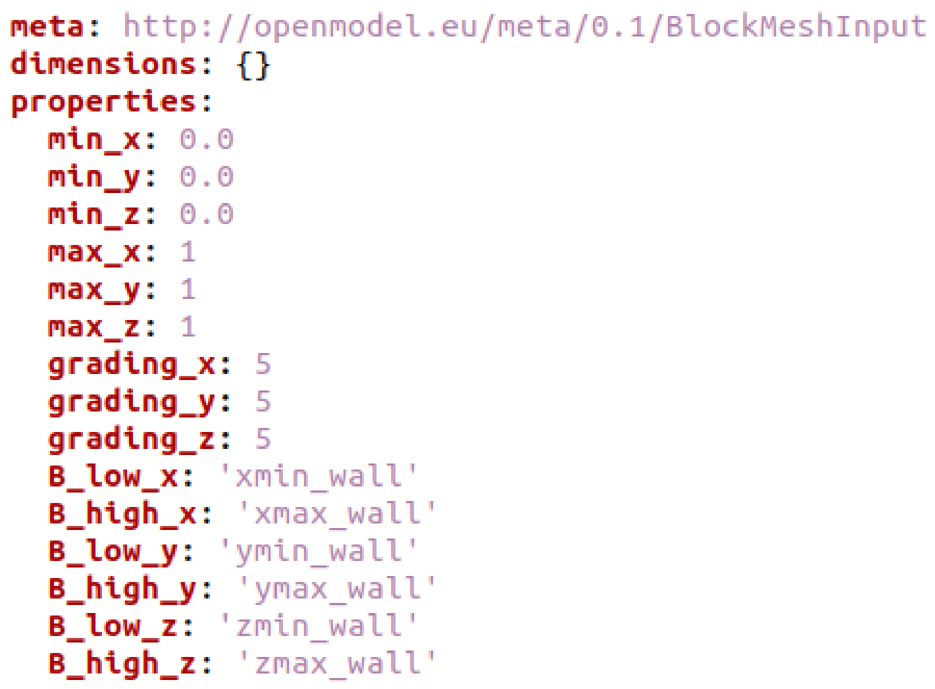
The input parameters for *blockMesh* are listed in Table 4. These define the geometric constraints (the size of the volume), the grading in each direction (how many mesh cells are distributed in that direction) and the name of each domain boundary.

Figure 22 shows the YAML file containing the inputs in the table, used as input for the *blockMesh* wrapper.

Table 4: Input parameters for blockMesh wrapper

Name	Type	description
min x	float	Minimum dimension of the domain in x direction
min y	float	Minimum dimension of the domain in y direction
min z	float	Minimum dimension of the domain in z direction
max x	float	Maximum dimension of the domain in x direction
max y	float	Maximum dimension of the domain in y direction
max z	float	Maximum dimension of the domain in z direction
grading_x	int	Discretisation in x direction
grading_y	int	Discretisation in y direction

grading_z	int	Discretisation in z direction
B_low_x	string	Name of the boundary at x min
B_high_x	string	Name of the boundary at x max
B_low_y	string	Name of the boundary at y min
B_high_y	string	Name of the boundary at y max
B_low_z	string	Name of the boundary at z min
B_high_z	string	Name of the boundary at z max



```yaml
meta: http://openmodel.eu/meta/0.1/BlockMeshInput
dimensions: {}
properties:
  min_x: 0.0
  min_y: 0.0
  min_z: 0.0
  max_x: 1
  max_y: 1
  max_z: 1
  grading_x: 5
  grading_y: 5
  grading_z: 5
  B_low_x: 'xmin_wall'
  B_high_x: 'xmax_wall'
  B_low_y: 'ymin_wall'
  B_high_y: 'ymax_wall'
  B_low_z: 'zmin_wall'
  B_high_z: 'zmax_wall'
```

Figure 22: Yaml file for blockMesh wrapper input.

1.1.6.1.2 OUTPUT

The output of *blockMesh* can be parsed to a CFDmesh entity, which is defined as below, and whose properties are summarized in Table 5.

```
"uri": "http://openmodel.eu/meta/0.1/CFDMesh",
  "description": "Entity to represent a CFD mesh",
  "dimensions": [
    {
      "name": "npoints",
      "description": "Number of points."
    },
    {
      "name": "ninternalfaces",
      "description": "Number of internal faces."
    },
    {
      "name": "nfaces",
      "description": "Number of faces."
    },
    {
      "name": "ncells",
      "description": "Number of cells."
    }
  ],
```

Table 5: Properties of a CFDmesh entity. The point, cell_face and cell types are so-called ref-types referring to the http://openmodel.eu/meta/0.1/point, http://openmodel.eu/meta/0.1/cell_face and http://openmodel.eu/meta/0.1/cell data models listed below.

Name	**Type**	**dimension**	**description**
points	point	npoints	Mesh points
faces	cell_face	nfaces	Connectivity: which points constitute the face.
cells	cell	ncells	which faces constitute the cell
owner	float	nfaces	List containing which cell owns the face.
neighbour	float	ninternalfaces	List containing which cell is considered neighbour

In Table 5, new data types are used, namely point, cell_face and cell. These are defined below.

Point

```
"uri": "http://openmodel.eu/meta/0.1/point",
"description": "Entity to represent a point",
"dimensions": [{
  "name": "space_dim",
  "description": "space dimensionality."
}],
"properties": [
  {
    "name": "coordinates",
    "shape": ["space_dim"],
    "type": "double",
    "description": "point coordinates (x,y,z)."
}
]
```

Face

```
"uri": "http://openmodel.eu/meta/0.1/cell_face",
"description": "Entity to represent points composing a face",
"dimensions": [{
  "name": "num_of_points",
  "description": "number of points that compose the face."
}],
"properties": [
  {
    "name": "points",
    "shape": ["num_of_points"],
    "type": "int",
    "description": "list of points that compose the face."
}
]
```

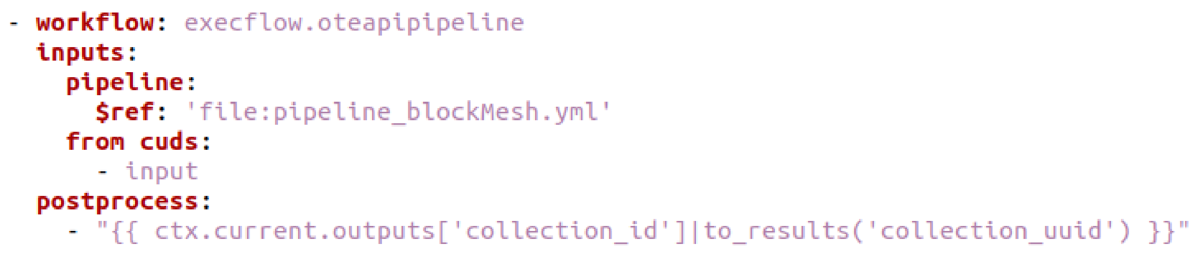
Cell

```
"uri": "http://openmodel.eu/meta/0.1/cell",
"description": "Entity to represent a cell",
"dimensions": [{
  "name": "num_of_faces",
  "description": "number of faces that compose the cell."
}],
"properties": [
  {
    "name": "faces",
    "shape": ["num_of_faces"],
    "type": "int",
    "description": "list of faces that compose the cell."
}
]
```

1.1.6.1.3 EXECFLOW

The reading of the required inputs, described in Section 1.1.6.1.1, is described in Figure 23 and Figure 24.

Figure 23 depicts the use of the oteapi pipeline ExecFlow utility to execute the pipeline and recover information, which is then sent to the global workflow context.



```
- workflow: execflow.oteapipipeline
  inputs:
    pipeline:
      $ref: 'file:pipeline_blockMesh.yml'
    from_cuds:
      - input
  postprocess:
    - "{{ ctx.current.outputs['collection_id']|to_results('collection_uuid') }}"
```

Figure 23: blockMesh wrapper: Input reading.

Figure 24 shows the pipeline, which is composed by two steps. In the first step, input data are read from the input yaml file. In the second, the DLite plugin *write_blockMeshDict* uses the read data and writes out the dictionary that will be used by blockMesh to generate the mesh data.

```yaml
version: 1
path:
strategies:
  - dataresource: load_data
    downloadUrl: "file://tmp/blockMesh_input.yml"
    mediaType: application/vnd.dlite-parse
    configuration:
      driver: yaml
      options: "mode=r"
      label: input

  - function: write_blockMeshDict
    functionType: application/vnd.dlite-generate
    configuration:
      driver: write_blockMeshDict
      location: /tmp/blockMeshCase/system/blockMeshDict
      options: "mode=w"
      label: input
```

Figure 24: Input pipeline with the loading of the data and the preparation of the blockMesh dictionary.

Once the input pipeline is executed, the shell_job utility is used to run blockMesh. A description of shell_job is given in Figure 25. From the execution of blockMesh we recover a folder called *constant* which contains the dictionary file generated by the software.

```yaml
- workflow: execflowdemo.shell_job
  inputs:
    executable:
      value: "blockMesh"
      type: core.str
    arguments:
    nodes:
      value:
        folder:
          value: "/tmp/blockMeshCase"
          type: core.str
      type: core.dict
    filenames:
    outputs:
      value:
      - constant
      type:
        core.list
- postprocess:
  - "{{ ctx.current.outputs['constant'] | to_ctx('constant') }}"
```

Figure 25: Shell job execution of the blockMesh wrapper.

In the last step, described in Figure 26, data is parsed to create an instance of the CFDmesh entity described in Section 1.1.6.1.2.

```
- workflow: execflow.oteapipipeline
  inputs:
    pipeline:
      $ref: 'file:pipeline_blockMesh_out.yml'
    to_cuds:
      - parse_mesh
    from_cuds:
      - cfd_mesh

    parse_mesh: "{{ ctx.constant.uuid }}"

- postprocess:
    - "{{ ctx.current.outputs['results']['cfd_mesh']|to_ctx('cfd_mesh') }}"
```

Figure 26: Results parsing of the blockMesh wrapper.

The pipeline which creates the CFDmesh instance is shown in Figure 27. We recover the information from the context, particularly the uuid, to make it available to the rest of the workflow. In fact, the *cfd_mesh* created and the end of the pipeline is then sent to the context in the postprocessing step.

```
version: 1
strategies:
  - function: datanode2cuds
    functionType: aiidacuds/datanode2cuds
    configuration:
      names: to_cuds

  - function: parseMesh
    functionType: application/vnd.dlite-generate
    configuration:
      driver: parse_mesh
      location: ./mesh
      label: parse_mesh

  - dataresource: inst_parseMesh
    downloadUrl: "file://tmp/mesh/all_info"
    mediaType: application/vnd.dlite-parse
    configuration:
      driver: parse_mesh_i
      options: "mode=r"
      label: cfd_mesh

  - function: cuds2datanode
    functionType: aiidacuds/cuds2datanode
    configuration:
      names: from_cuds

pipelines:
  pipe: datanode2cuds | parseMesh |inst_parseMesh | cuds2datanode
```

Figure 27: Pipeline to parse the mesh files.

GENERATE GEOMETRY AND MESH: SALOME WRAPPER

SALOME is a multi-platform open-source scientific computing environment, allowing the realization of industrial studies of physics simulations.

In this context, we are interested only in two SALOME modules which handle geometry and meshes:

- **GEOM**: this component provides multiple functionalities for creating, viewing, and modifying geometric CAD models.
- **SMESH**: mesh generator, compatible with the UNV, MED, STL, CGNS, SAUV and GMF formats. It contains the NetGen algorithms, mesh handling functionalities, and mesh quality control operations.

Since the creation of geometry is a complex task, in this deliverable we will demonstrate the wrappers for SALOME for simple 2D and 3D shapes, namely:

- Square plate.
- Cube.
- Sphere.
- Cylinder.

1.1.6.1.4 DATA MODELS

The file documenting the workflow input is a json file, with the following structure:

```
{
 "uri": "http://openmodel.eu/meta/0.1/salomeInputShape",
 "description": "Input for salome script",
 "dimensions": {},
 "properties": {
   … properties describing the shapes …

   "max_size": {
     "type": "float",
     "description": "maximum size of the mesh elements"
   },
   "min_size": {
     "type": "float",
     "description": "minimum size of the mesh elements"
   },
   "discr_type": {
     "type": "string",
     "description": " discretization type (hexa or tetra)"
   },
   "mesh_type": {
     "type": "string",
     "description": " Mesh type surface or volume"
   },
   "name": {
     "type": "string",
     "description": "name stl file"
   }
 }
}
```

Figure 28: Data model describing SALOME script

All the shapes follow the same structure, with a series of properties that describe the geometry, followed by the discretization and mesh properties.

1.1.6.1.5 INPUT

The input parameters for generating shapes and meshes are contingent upon the user's desired outcome, as briefly outlined in the preceding section. Table 6 delineates the parameters governing the mesh. Table 7 to Table 10 detail the inputs necessary for creating the various shapes.

Table 6: Input parameters for mesh.

Name	Type	description
max_size	float	Maximum size allowed for mesh elements.
min_size	float	Minimum size allowed for mesh elements.
discr_type	string	The mesh can be generated using tetrahedra of hexahedra.
mesh_type	string	The mesh can be the result of surface or volume discretization.
name	string	Name of the generated mesh.

In Figure 29 we show the same geometry with different mesh discretization, hexahedral and tetrahedral. Typically, the choice of mesh discretization stems from the application of different numerical methods; the former is prevalent in the finite volume method, while the latter is characteristic of the finite element method.

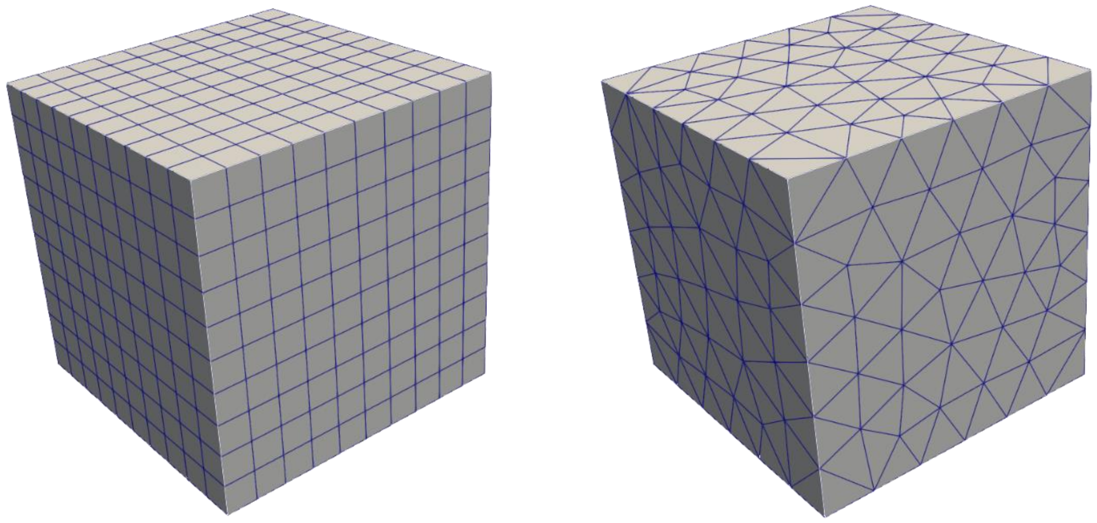


Figure 29: Example of the same geometry (cube) with different discretization type. Left: hexahedral mesh. Right: Tetrahedral mesh.

Figure 30 shows a single geometry with tetrahedral mesh, but in one case we want a surface (2D) mesh, while in the other we want a volume (3D) mesh.

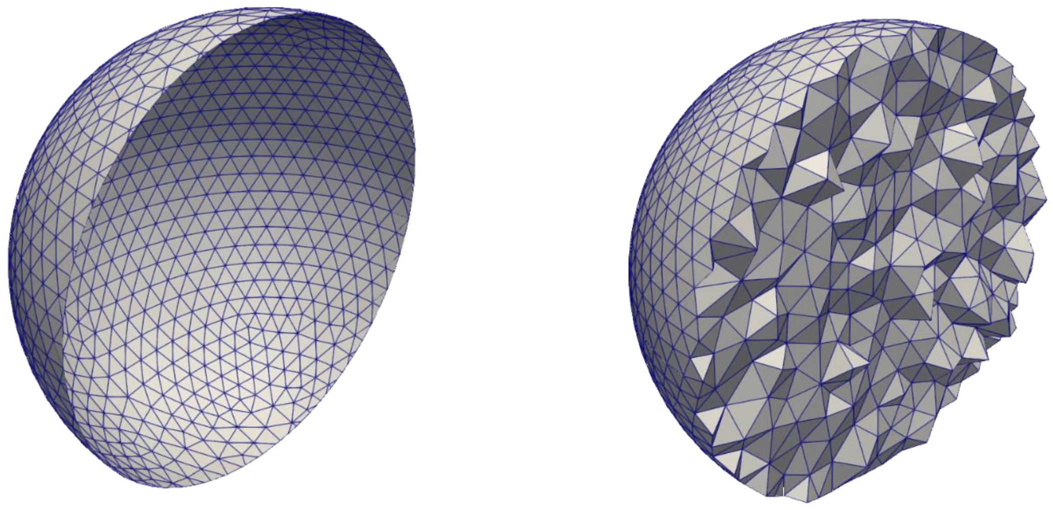


Figure 30: Example of the same geometry (sphere) with different mesh type. Left: surface mesh. Right: volume mesh.

Table 7 : Input parameters to generate a plate.

Name	Type	description
min x	float	Minimun dimension of the plate in x direction
min y	float	Minimun dimension of the plate in y direction
min z	float	Minimun dimension of the plate in z direction
max x	float	Maximun dimension of the plate in x direction
max y	float	Maximun dimension of the plate in y direction
max z	float	Maximun dimension of the plate in z direction (should be equal to min z)

Table 8 : Input parameters to generate a sphere.

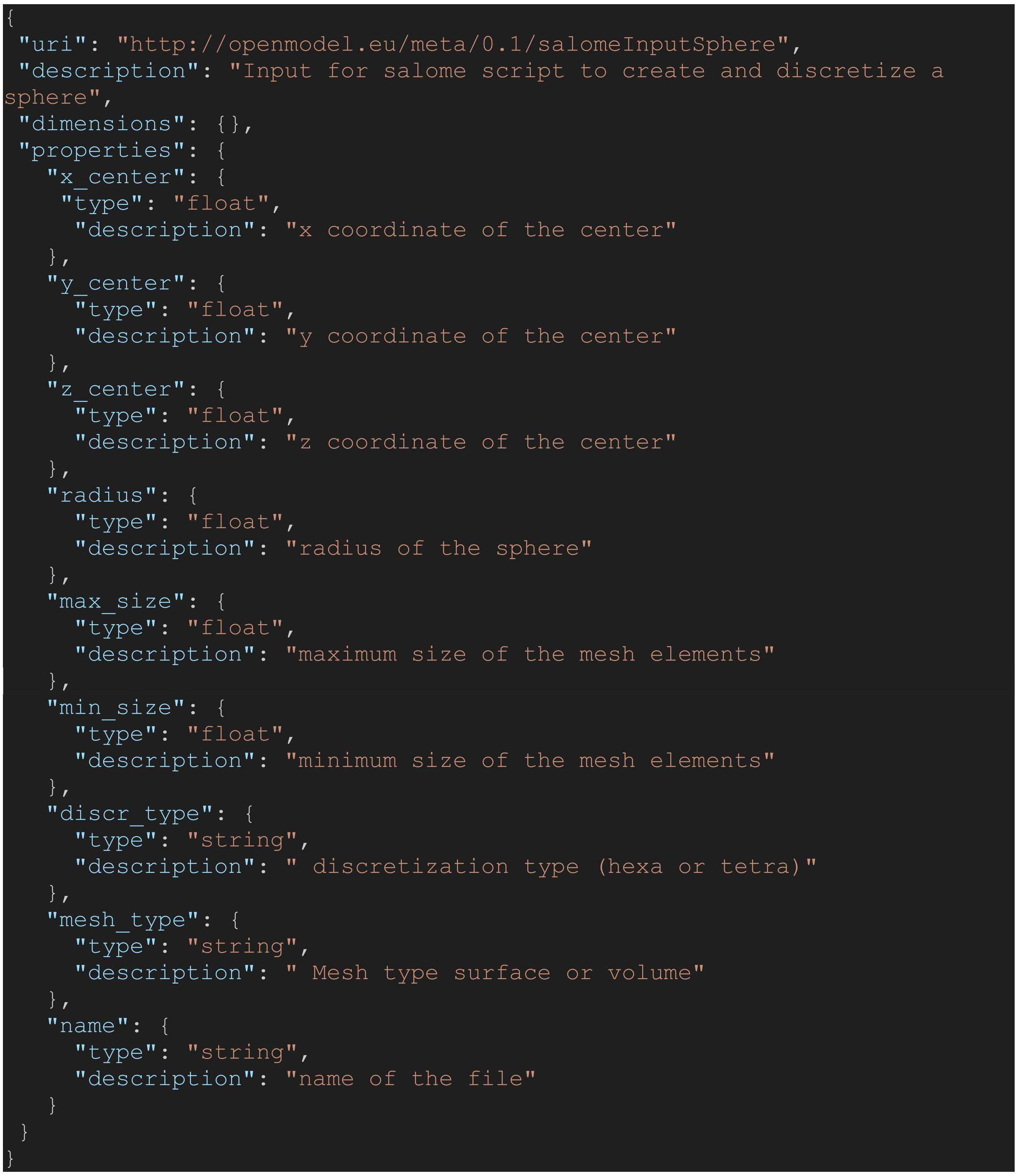
Name	Type	description
x center	float	x coordinate of the center
y center	float	y coordinate of the center
z center	float	z coordinate of the center
radius	float	Radius of the sphere

Table 9 Input parameters to generate a cylinder.

Name	Type	description
min x	float	x coordinate of the center (lower circle)
min y	float	y coordinate of the center (lower circle)
min z	float	z coordinate of the center (lower circle)
max x	float	x coordinate of the center (upper circle)
max y	float	y coordinate of the center (upper circle)
max z	float	z coordinate of the center (upper circle)
radius	float	Radius of the cylinder
height	float	Height of the cylinder

Table 10 : Input parameters to generate a rectangular cuboid.

Name	Type	description
min x	float	Minimun dimension of the rectangular cuboid in x direction
min y	float	Minimun dimension of the rectangular cuboid in y direction
min z	float	Minimun dimension of the rectangular cuboid in z direction
max x	float	Maximun dimension of the rectangular cuboid in x direction
max y	float	Maximun dimension of the rectangular cuboid in y direction
max z	float	Maximun dimension of the rectangular cuboid in z direction



```
{
 "uri": "http://openmodel.eu/meta/0.1/salomeInputSphere",
 "description": "Input for salome script to create and discretize a
sphere",
 "dimensions": {},
 "properties": {
   "x_center": {
    "type": "float",
     "description": "x coordinate of the center"
   },
   "y_center": {
     "type": "float",
     "description": "y coordinate of the center"
   },
   "z_center": {
     "type": "float",
     "description": "z coordinate of the center"
   },
   "radius": {
     "type": "float",
     "description": "radius of the sphere"
   },
   "max_size": {
     "type": "float",
     "description": "maximum size of the mesh elements"
   },
   "min_size": {
     "type": "float",
     "description": "minimum size of the mesh elements"
   },
   "discr_type": {
     "type": "string",
     "description": " discretization type (hexa or tetra)"
   },
   "mesh_type": {
     "type": "string",
     "description": " Mesh type surface or volume"
   },
   "name": {
     "type": "string",
     "description": "name of the file"
   }
 }
}
```

Figure 31: Example of data description file for a sphere.

1.1.6.1.6 OUTPUT

The final result of the workflow varies based on the type of mesh requested by the user. In the case of surface meshes, exported as a stl file, the output is documented in Figure: 32, whereas in the case of volume meshes, which are exported as cgns files, it is described in Figure 33.

```
{
 "uri": "http://openmodel.eu/meta/0.1/stlMesh",
 "description": "Entity to represent a stl mesh",
 "dimensions": [{
   "name": "number_of_files",
   "description": "Number of stl files."
 }],
 "properties": {
   "files": {
     "type": "string",
     "dims": ["number_of_files"],
     "description": "list of stl file"
   }}}
```

Figure: 32 Data description for the surface mesh (stl file).

```
{
 "uri": "http://openmodel.eu/meta/0.1/cgnsMesh",
 "description": "Entity to represent a cgns mesh",
 "dimensions": [{
   "name": "number_of_files",
   "description": "Number of cgns files."
 }],
 "properties": {
   "files": {
     "type": "string",
     "dims": ["number_of_files"],
     "description": "list of cgns file"
   }}}
```

Figure 33: Data description for the volume mesh (cgns file).

1.1.6.1.7 EXECFLOW

Figure 34 depicts the use of the oteapi pipeline execflow utility to execute the pipeline and recover information, which is then sent to the global workflow context.

```
 - workflow: execflow.oteapipipeline
   inputs:
     pipeline:
       $ref: 'file:pipeline_salome.yml'
     from_cuds:
       - salome_input
   postprocess:
     -"{{ctx.current.outputs['collection_id']|to_ctx('collection_uuid')}}"
```

Figure 34: Input reading for the SALOME wrapper.

Figure 35 shows the pipeline, which is composed of two steps. In the first step, input data are read from the input yaml file. In the second, the DLite plugin *write_salome_script*, uses the data and writes out the script that will be used by SALOME to generate the mesh data.

```
strategies:
 - dataresource: load_data
   downloadUrl: "file:////tmp/salome_input.yml"
   mediaType: application/vnd.dlite-parse
   configuration:
     driver: yaml
     options: "mode=r"
     label: input

 - function: write_salome_script
   functionType: application/vnd.dlite-generate
   configuration:
     driver: write_salome_shape
     location: tmp/salome_script.py
     options: "mode=w"
     label: input
pipelines:
 pipe: load_data | write_salome_script
```

Figure 35: Input pipeline for SALOME wrapper.

Once the input pipeline is executed, the exec_wrapper utility is used to run SALOME. A depiction of the wrapper structure is given in Figure 36. From the execution of SALOME we recover the remote folder which contains the mesh files. Information is retrieved from the context, in particular the data node uuid (unique universal identifier), and made available to the rest of the workflow.

```
- workflow:  execflow.exec_wrapper
    inputs:
     command: "/home/SALOME_9_9_0/salome"
     arguments:
       - "-t"
       - "{script}"
     files:
       script:
         filename: "salome_script.py"
         template: "/tmp/salome_script.py"
     outputs:

 - postprocess:
   - "{{ ctx.current.outputs['remote_folder'] |
to_ctx('remote_folder')}}"
```

Figure 36: SALOME wrapper execution using exec_wrapper.

In the last step, depicted in Figure 37, data is parsed to create an instance of the stlMesh or cgnsMesh entity described in 1.1.6.1.6.

```
- workflow: execflow.oteapipipeline
   inputs:
     pipeline:
       $ref: 'file:pipeline_salome_out.yml'
     to_cuds:
       - path
     from_cuds:
       - mesh
     path: "{{ ctx.remote_folder.uuid }}"
 - postprocess:
     -"{{ ctx.current.outputs['results']['mesh']|to_ctx('mesh')}}"
```

Figure 37: Execution of the output pipeline in the SALOME wrapper.

The pipeline which creates the mesh instance is shown in Figure 38. Using a combination of DLite plugins we can create an instance of the mesh data, which can be then recovered and used by other part of the workflow. For example, we can use the ParaView wrapper to visualize the mesh.

```yaml
version: 1
strategies:
 - function: datanode2cuds
   functionType: aiidacuds/datanode2cuds
   configuration:
     names: to_cuds

 - function: parseMesh
   functionType: application/vnd.dlite-generate
   configuration:
     driver: parse_mesh
     location: ./mesh
     label: path

 - dataresource: inst_parseMesh
   downloadUrl: "file:///tmp/mesh/uuid_meshes"
   mediaType: application/vnd.dlite-parse
   configuration:
     driver: parse_mesh_i
     options: "mode=r"
     label: stl_mesh

 - function: cuds2datanode
   functionType: aiidacuds/cuds2datanode
   configuration:
     names: from_cuds

pipelines:
 pipe: datanode2cuds | parseMesh | inst_parseMesh | cuds2datanode
```

Figure 38: Output pipeline of the SALOME wrapper.

POSTPROCESSING WRAPPER: PARAVIEW

ParaView is an open-source data visualization and analysis software designed for use with scientific and engineering data. It is commonly used to visualize complex datasets, perform simulations, and analyze results across various domains such as physics, engineering, and medical research. ParaView provides a graphical user interface that allows users to explore and visualize data using a variety of rendering techniques, including volume rendering, contour plots, and surface plots. It supports a wide range of data formats and can handle large datasets efficiently.

In this context, we will limit the wrapper scope to the data used or generated by the success stories. The meshes, CFD and granular data from Success Story 6 and the xdmf file from Success Story 3.

1.1.6.1.8 AIIDA PLUGIN

An AiiDa plugin was developed for ParaView, which among other things execute the code shown in Figure 39.

This plugin receives as input the uuid of the data instance to visualize. The data is read and then, based on the data type, a custom reader is used to transform the data from the instance into information that can be visualized in ParaView.

The readers for each data type involves the use of the vtk library and they must be customized for every data type.

1.1.6.1.9 EXECFLOW

Since ParaView is a visualization software, it only makes sense to use it at the end of other workflows.

Figure 40 shows an example of the use of the ParaView plugin in execflow to visualize the mesh. The mesh itself is shown in Figure 41.

```python
import vtk
import paraview.web.venv
from pv_readers import pv_readers
import aiida
aiida.load_profile()
from aiida import orm


data = orm.load_node(uuid="{uuid}")

def pv_show(x, chosen_data="invalidData"):
  wrappers = {
    "CFDMesh": pv_readers.CFDMeshReader(),
    "granularMedium": pv_readers.granularReader(),
    "stlMesh": pv_readers.stlReader(),
    "cgnsMesh": pv_readers.cgnsReader(),
    "xdmfFile": pv_readers.xdmfReader()
  }

  chosen_wrapper = wrappers.get(chosen_data, pv_readers.notImplemented())

  return chosen_wrapper.show(x)

pv_show(data, data['type'])
```

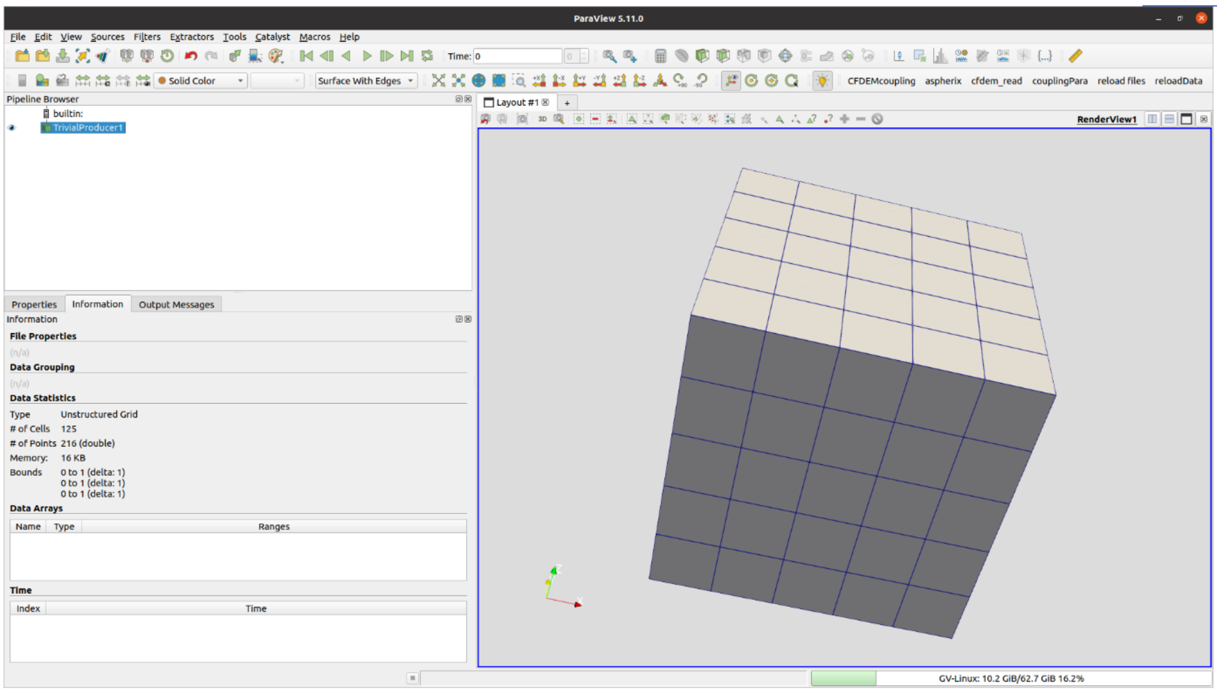
Figure 39: Python script executed by the ParaView Aiida plugin.

```yaml
  - calcjob: aiida_pv
    inputs:

      code: paraview@localhost-test

      uuid:  "{{ ctx.mesh.uuid }}"
```

Figure 40: ParaView in execflow.


ParaView 5.11.0
File Edit View Sources Filters Extractors Tools Catalyst Macros Help
Time: 0
Solid Color
Surface With Edges
CFDEMcoupling aspherix cfdem_read couplingPara reload files reloadData
Pipeline Browser
builtin:
TrivialProducer1
Layout #1
RenderView1
Properties Information Output Messages
GV-Linux: 10.2 GiB/62.7 GiB 16.2%

Information

File Properties

(n/a)

Data Grouping

(n/a)

Data Statistics

```
Type        Unstructured Grid
# of Cells  125
# of Points 216 (double)
Memory:     16 KB
Bounds      0 to 1 (delta: 1)
            0 to 1 (delta: 1)
            0 to 1 (delta: 1)
```

Data Arrays

Name	Type	Ranges

Time

Index	Time

Figure 41: Example of a CFDMesh visualized in ParaView using the OpenModel wrapper.

2 BIBLIOGRAPHY

[1] "Closing the loop on composites. Using the digital twin to virtually predict andoptimize product performance.," *Siemens Digital Industries Software*, 2019.

[2] F. Shishkina; Martine Wevers; Stepan V. Lomov; Laszlo Oxana, "Micro-CT-based numerical validation of the local permeability map for the B-Pillar infusion simulation," in *Composites Meet Sustainability – Proceedings of the 20th European Conference on Composite Materials, ECCM20. 26-30 June, 2022, Lausanne, Switzerland*.

3 ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 953167.