OpenFLUID

Software Environment
for Spatial Modelling in Landscapes

OpenFLUID in a nutshell

Manual for OpenFLUID v2.2.0

The OpenFLUID team
March 2025

Contents

Foreword

OpenFLUID is a software environment for spatial modelling in landscapes, mainly focused on fluxes. It is developed by the LISAH (Laboratory of Interactions Soil-Agrosystem-Hydrosystem, Montpellier, France) which is a joint research unit between INRA (French National Institute for Agricultural Research), IRD (French Institute for Research and Development) and Montpellier SupAgro (International Centre for Higher Education in Agricultural Sciences).

This documentation is made of several parts

- a guide for running simulations using OpenFLUID, including the construction of input datasets
- a guide for development of OpenFLUID simulators, either using existing source code or creating source code *de novo*
- an appendix giving useful reference informations

Detailed informations about scientific concepts underlying the OpenFLUID software are available on the official OpenFLUID web site : `https://www.openfluid-project.org`

Practical informations for OpenFLUID users, including usage and development, are available on the OpenFLUID Community site : `https://community.openfluid-project.org`

Part I

Running simulations with OpenFLUID

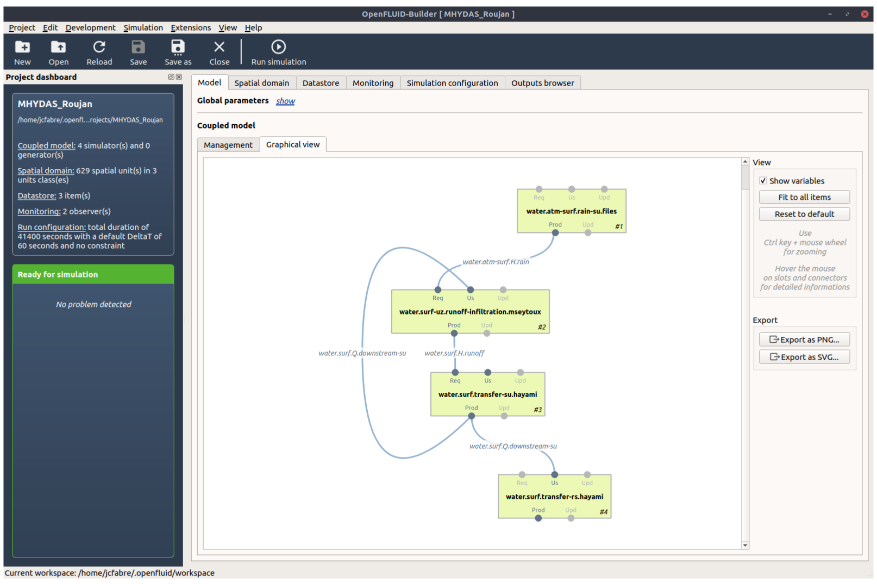
Chapter 1

OpenFLUID software environment

OpenFLUID simulations can be run either using the command line interface (`openfluid` program), the graphical user interface (`openfluid-builder` program), or through bindings with external languages and environments such as R using the ROpenFLUID package.
All these programs and packages use the same input dataset format (See Format of input datasets), and propose all concepts and features of the OpenFLUID software environment, as they share the unique OpenFLUID software framework.

1.1 Graphical Interface for simulations : OpenFLUID-Builder

The OpenFLUID-Builder user interface proposes a graphical environment to prepare, parameterize and execute simulations. It is a good starting point for users who want to discover the OpenFLUID concepts and software environment. It can be run either from the desktop menu of your system or from a console by typing the `openfluid-builder` command.


OpenFLUID-Builder [MHYDAS_Roujan]
Project Edit Development Simulation Extensions View Help
New Open Reload Save Save as Close Run simulation
Project dashboard
Model Spatial domain Datastore Monitoring Simulation configuration Outputs browser
Global parameters show
Coupled model
Management Graphical view
water.atm-surf.rain-su.files #1
water.atm-surf.H.rain
water.surf-uz.runoff-infiltration.mseytoux #2
water.surf.Q.downstream-su
water.surf.H.runoff
water.surf.transfer-su.hayami #3
water.surf.Q.downstream-su
water.surf.transfer-rs.hayami #4
Req Us Upd Prod Upd
View
✓ Show variables
Fit to all items
Reset to default
Use Ctrl key + mouse wheel for zooming
Hover the mouse on slots and connectors for detailed informations
Export
Export as PNG...
Export as SVG...

MHYDAS_Roujan
/home/jcfabre/.openfl...rojects/MHYDAS_Roujan

Coupled model: 4 simulator(s) and 0 generator(s)

Spatial domain: 629 spatial unit(s) in 3 units class(es)

Datastore: 3 item(s)

Monitoring: 2 observer(s)

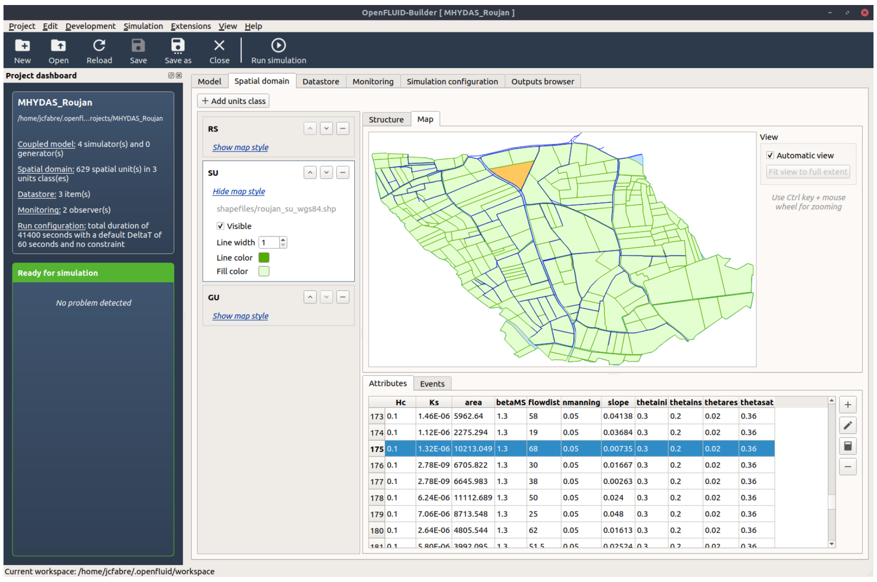
Run configuration: total duration of 41400 seconds with a default DeltaT of 60 seconds and no constraint

Ready for simulation

No problem detected

Current workspace: /home/jcfabre/.openfluid/workspace

Figure 1.1: Screenshot of the model view in OpenFLUID-Builder


OpenFLUID-Builder [MHYDAS_Roujan]
Project Edit Development Simulation Extensions View Help
New Open Reload Save Save as Close Run simulation
Project dashboard
Model Spatial domain Datastore Monitoring Simulation configuration Outputs browser
Add units class
RS
Show map style
SU
Hide map style
GU
Show map style
Structure Map
View
Automatic view
Fit view to full extent
Use Ctrl key + mouse wheel for zooming
Attributes Events

MHYDAS_Roujan

/home/jcfabre/.openfl...rojects/MHYDAS_Roujan

Coupled model: 4 simulator(s) and 0 generator(s)

Spatial domain: 629 spatial unit(s) in 3 units class(es)

Datastore: 3 item(s)

Monitoring: 2 observer(s)

Run configuration: total duration of 41400 seconds with a default DeltaT of 60 seconds and no constraint

Ready for simulation

No problem detected

shapefiles/roujan_su_wgs84.shp

✓ Visible

Line width 1

Line color 🟩

Fill color 🟩

	Hc	Ks	area	betaMS	flowdist	nmanning	slope	thetaini	thetains	thetares	thetasat
173	0.1	1.46E-06	5962.64	1.3	58	0.05	0.04138	0.3	0.2	0.02	0.36
174	0.1	1.12E-06	2275.294	1.3	19	0.05	0.03684	0.3	0.2	0.02	0.36
175	0.1	1.32E-06	10213.049	1.3	68	0.05	0.00735	0.3	0.2	0.02	0.36
176	0.1	2.78E-09	6705.822	1.3	30	0.05	0.01667	0.3	0.2	0.02	0.36
177	0.1	2.78E-09	6645.983	1.3	38	0.05	0.00263	0.3	0.2	0.02	0.36
178	0.1	6.24E-06	11112.689	1.3	50	0.05	0.024	0.3	0.2	0.02	0.36
179	0.1	7.06E-06	8713.548	1.3	25	0.05	0.048	0.3	0.2	0.02	0.36
180	0.1	2.64E-06	4805.544	1.3	62	0.05	0.01613	0.3	0.2	0.02	0.36
181	0.1	5.80E-06	3992.095	1.3	51.5	0.05	0.02524	0.3	0.2	0.02	0.36

Current workspace: /home/jcfabre/.openfluid/workspace

Figure 1.2: Screenshot of the spatial domain map view in OpenFLUID-Builder

OpenFLUID-Builder functionalities can be extended by Builder-extensions which are graphical plugins for this user interface. By default, OpenFLUID is provided with two Builder-extensions: a graph viewer representing the spatial domain as a connected graph, and a spatial data importer to create a spatial domain from standard GIS data file formats (such as Shapefiles) or from a WFS service (Web Feature Service) available from a local or an internet server.

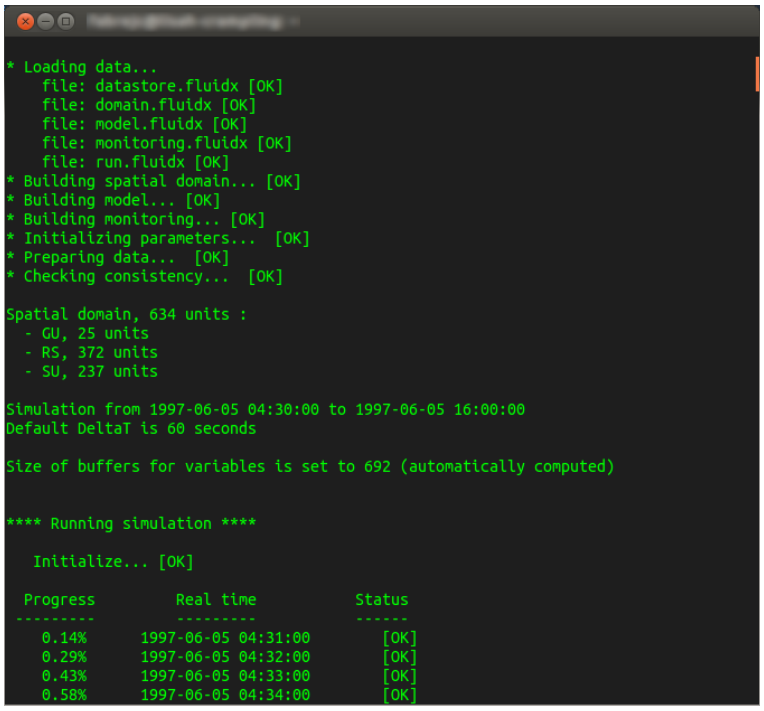
1.2 Command-line interface : openfluid

The OpenFLUID command line interface allows to run OpenFLUID simulations from a terminal, using the `openfluid` program. This usage is particularly useful for running multiple simulations in batch or on compute systems such as compute clusters.
To run the simulation, execute the `openfluid` program with adapted commands and options. You can run a simulation using the `run` command and giving the input dataset path or the project path and the optional results output path:

```
openfluid run (</path/to/dataset>|</path/to/project>) [</path/to/results>]
```

When running a project, the results output path is ignored as it is already defined by the project itself. The project must be a valid OpenFLUID project (see Structure of an OpenFLUID project), usually created using the OpenFLUID-Builder user interface. It can also be created manually.

See Command line usage or run `openfluid --help` to get the list of available commands and options.



```
* Loading data...
    file: datastore.fluidx [OK]
    file: domain.fluidx [OK]
    file: model.fluidx [OK]
    file: monitoring.fluidx [OK]
    file: run.fluidx [OK]
* Building spatial domain... [OK]
* Building model... [OK]
* Building monitoring... [OK]
* Initializing parameters...  [OK]
* Preparing data...  [OK]
* Checking consistency...  [OK]

Spatial domain, 634 units :
  - GU, 25 units
  - RS, 372 units
  - SU, 237 units

Simulation from 1997-06-05 04:30:00 to 1997-06-05 16:00:00
Default DeltaT is 60 seconds

Size of buffers for variables is set to 692 (automatically computed)


**** Running simulation ****

   Initialize... [OK]

  Progress         Real time            Status
 ---------         ---------            ------
    0.14%      1997-06-05 04:31:00        [OK]
    0.29%      1997-06-05 04:32:00        [OK]
    0.43%      1997-06-05 04:33:00        [OK]
    0.58%      1997-06-05 04:34:00        [OK]
```

Figure 1.3: OpenFLUID simulation using command line

1.3 Development environment : OpenFLUID-DevStudio

The OpenFLUID-Devstudio is the part of the software environment dedicated to development of simulators, observers and builder-extensions. It proposes functionnalities for assisted source code creation and development. It can be run either from the desktop menu of your system or from a console by typing the `openfluid-devstudio` command.

The OpenFLUID-DevStudio environment proposes the following facilities:

- Assisted creation of simulators, observers and builder-extensions
- Ware-centered organization of workspace with navigator
- Integrated configuration and build of source code (for debug and install modes)
- OpenFLUID-oriented completion system (as you type and through contextual menu)
- Direct access to online documentation
- Common features of a source code editor

More informations about OpenFLUID-Devstudio are available in the Creation of a simulator part.

1.4 Within the GNU R environment : ROpenFLUID

OpenFLUID can be used from within the GNU R environment through the ROpenFLUID package. This package allows to load an input dataset, parameterize and run a simulation, then use and process simulation results.

It is really useful for taking benefit of all R features and packages for sensitivity analysis, optimization, uncertainty propagation analysis, and more.
Example of a simulation launch in R using the ROpenFLUID package:

```
0
library('ROpenFLUID')

ofsim = OpenFLUID.loadDataset('/path/to/dataset')

OpenFLUID.setCurrentOutputDir('/path/to/results')

OpenFLUID.runSimulation(ofsim)

data = OpenFLUID.loadResult(ofsim,'TestUnits',15,'var.name')
```

More details are available in the dedicated ROpenFLUID documentation, available on the OpenFLUID community site (`https://community.openfluid-project.org/`).

Chapter 2

Format of input datasets

The FluidX file format is used to define a simulation dataset. An OpenFLUID input dataset includes different informations, defined in one or many files:

- the **coupled model** definition
- the **spatial domain** definition, including spatial connectivity, attributes and events
- the **datastore** content
- the **monitoring** configuration
- the **run** configuration

All files must be placed into a directory that can be reached by the OpenFLUID program used. As all OpenFLUID apps use the FluidX format natively, the entire input dataset can be created using OpenFLUID-Builder. Alternatively, these FluidX files can be created by hand or using external tools such as text editors, scientific environments (e.g. R, Matlab), geographic information systems (GIS), ...

2.1 Overview

The FluidX file format is an `XML` based format defined for OpenFLUID input datasets. An OpenFLUID dataset can be provided by a one or many files using this FluidX format. The file (s) name(s) must use the `.fluidx` file extension.
Whatever the input information is put into one or many files, the following sections must be defined in the input file(s) set:

- The model section defined by the `<model>` tag
- The spatial domain section defined by the `<domain>` tag
- The datastore section defined by the `<datastore>` tag
- The monitoring section defined by the `<monitoring>` tag
- The run section defined by the `<run>` tag

The order of these sections is not significant. All of these sections must be inclosed into an *openfluid* section defined by the `<openfluid>` tag.
The `<openfluid>` tag should contain a `format` attribute giving information about the format and version of the file (currently, the format value is `fluidx 4`).
Summary view of the XML structure of FluidX files:

```
0
<?xml version="1.0" standalone="yes"?>

<openfluid format="fluidx 4">
```

```
  <model>
    // here is the model definition
  </model>

  <domain>
    // here is the spatial domain definition
    // with connectivity, attributes and events
  </domain>

  <datastore>
    // here is the datastore content
  </datastore>

  <monitoring>
    // here is the monitoring definition
  </monitoring>

  <run>
    // here is the run configuration
  </run>

</openfluid>
```

2.2 Sections

2.2.1 Model section

The coupled model is defined by an ordered set of simulators and/or data generators that will be automatically plugged in and run by the OpenFLUID environment. It can also include a section for global parameters which apply to all simulators and generators. The global parameters may be overridden by local parameters of simulators or generators.
The coupled model must be defined in a section delimited by the `<model>` tag, and must be structured following these rules:

- Inside the `<model>` tag, there must be a set made of at least one `<simulator>` or `<generator>` tags, and an optional `<gparams>` tag.
- Each `<simulator>` tag must bring an `ID` attribute giving the identifier of the simulator; the value of the `ID` attribute must match the ID of an available and pluggable simulator. It also brings an `enabled` tag giving the active state of the simulator (if missing, default value is *1* for active).
- Each `<simulator>` tag may include zero to many `<param>` tags giving parameters to the simulator. Each `<param>` tag must bring a `name` attribute giving the name of the parameter and a `value` attribute giving the value of the parameter.
- Each `<generator>` tag must bring a `varname` attribute giving the name of the produced variable, a `unitsclass` attribute giving the unit class of the produced variable, a `method` attribute giving the method used to produce the variable (`fixed` for constant value, `random` for random value in a

range, `interp` for a time-interpolated value from given data series, `inject` for an injected value -no time interpolation- from given data series). An optional `<varsize>` attribute can be set in order to produce a vector variable instead of a scalar variable. It also brings an `enabled` tag giving the active state of the simulator (if missing, default value is *1* for active).

- Each `<generator>` tag may include zero to many `<param>` tags giving parameters to the generator. Each `<param>` tag must bring a `name` attribute giving the name of the parameter and a `value` attribute giving the value of the parameter.
- A generator using the `fixed` method must provide a parameter named *fixedvalue* for the value to produce.
- A generator using the `random` method must provide a parameter named *min* and a parameter named *max* delimiting the random range for the value to produce.
- A generator using the `inject` or `interp` method must provide a parameter named *sources* giving the data sources filename and a param named *distribution* giving the distribution filename for the value to produce (see also Single-column value generator).
- Each `<gparams>` tag may include zero to many `<param>` tags giving the global parameters. Each `<param>` tag must bring a `name` attribute giving the name of the parameter and a `value` attribute giving the value of the parameter.

```
0
<?xml version="1.0" standalone="yes"?>

<openfluid format="fluidx 4">

  <model>


    <gparams>

      <param name="gparam1" value="100" />

      <param name="gparam2" value="0.1" />

    </gparams>


    <simulator ID="example.simulatorA" enabled="1" />


    <generator varname="example.generator.fixed" unitsclass="EU1"

               method="fixed" varsize="11" enabled="1">

      <param name="fixedvalue" value="20" />

    </generator>


    <generator varname="example.generator.random" unitsclass="EU2" method="random" enabled="1">

      <param name="min" value="20.53" />

      <param name="max" value="50" />

    </generator>


    <simulator ID="example.simulatorB" enabled="0">

      <param name="param1" value="strvalue" />

      <param name="param2" value="1.1" />

      <param name="gparam1" value="50" />

    </simulator>
  </model>

</openfluid>
```

Warning

There must be only one model definition in the input dataset.

The order of the simulators and data generators in the `<model>` section is important : this order will be the call order at initialization time and during simulations in synchronized coupled model (not applicable for variable time coupled models)

2.2.2 Spatial domain section

Definition and connectivity

The spatial domain is defined by a set of spatial units that are connected each others. These spatial units are defined by a numerical identifier (ID) and a class. They also include informations about the processing order of the unit in the class. Each unit can be connected to zero or many other units from the same or a different unit class. The spatial domain definition must be defined in a section delimited by the `<definition>` tag, which is a sub-section of the `<domain>` tag, and must be structured following these rules:

- Inside the `<definition>` tag, there must be a set of `<unit>` tags
- Each `<unit>` tag must bring an `ID` attribute giving the identifier of the unit, a `class` attribute giving the class of the unit, a `pcsorder` attribute giving the process order in the class of the unit
- Each `<unit>` tag may include zero or many `<to>` tags giving the *to* connections to other units. Each `<to>` tag must bring an `ID` attribute giving the identifier of the connected unit and a `class` attribute giving the class of the connected unit
- Each `<unit>` tag may include zero or many `<childof>` tags giving the *child-parent* connections to other units. Each `<childof>` tag must bring an `ID` attribute giving the identifier of the parent unit and a `class` attribute giving the class of the parent unit

```
0
<?xml version="1.0" standalone="yes"?>

<openfluid format="fluidx 4">

  <domain>

    <definition>


      <unit class="PU" ID="1" pcsorder="1" />


      <unit class="EU1" ID="3" pcsorder="1">

        <to class="EU1" ID="11" />

        <childof class="PU" ID="1" />

      </unit>


      <unit class="EU1" ID="11" pcsorder="3">

        <to class="EU2" ID="2" />

      </unit>
```

```
        <unit class="EU2" ID="2" pcsorder="1" />


      </definition>

    </domain>

  </openfluid>
```

Attributes

The spatial attributes are static data associated to each spatial unit, usually properties and initial conditions. The spatial domain attributes must be defined in a section delimited by the `<attributes>` tag, which is a sub-section of the `<domain>` tag, and must be structured following these rules:

- The `<attributes>` tag must bring an `unitsclass` attribute giving the unit class to which the attributes must be attached, and a `colorder` attribute giving the order of the contained column-formatted data
- Inside the `<attributes>` tag, there must be the attributes as row-column text. As a rule, the first column is the ID of the unit in the class given through the `unitsclass` attribute of `<attributes>` tag, the following columns are values following the column order given through the `colorder` attribute of the `<attributes>` tag. Values for the data can be double, integer, boolean, string, vector, matrix or map formatted as strings (see part String representation of values).

```
0
<?xml version="1.0" standalone="yes"?>

<openfluid format="fluidx 4">

  <domain>


    <attributes unitsclass="EU1" colorder="indataA">

      3 1.1

      11 7.5

    </attributes>


    <attributes unitsclass="EU2" colorder="indataB1;indataB3">

      2 18 STRVALX

    </attributes>


  </domain>

</openfluid>
```

Discrete events

The discrete events are events occurring on units, and can be processed by simulators. The spatial events must be defined in a section delimited by the `<calendar>` tag, which is a sub-section of the `<domain>` tag, and must be structured following these rules:

- Inside the `<calendar>` tag, there must be a set of `<event>` tags
- Each `<event>` tag must bring an `unitID` and an `unitsclass` attribute giving the unit on which occurs the event, a `date` attribute giving the date and time of the event. The date format must be "YYYY-MM-DD hh:mm:ss".

- Each <event> tag may include zero to many <info> tags.
- Each <info> tag give information about the event and must bring a key attribute giving the name (the "key") of the info, and a value attribute giving the value for this key.

```
0
<?xml version="1.0" standalone="yes"?>

<openfluid format="fluidx 4">

  <domain>

    <calendar>


      <event unitsclass="EU1" unitID="11" date="1999-12-31 23:59:59">

        <info key="when" value="before" />

        <info key="where" value="1" />

        <info key="var1" value="1.13" />

        <info key="var2" value="EADGBE" />

      </event>

      <event unitsclass="EU2" unitID="3" date="2000-02-05 12:37:51">

        <info key="var3" value="152.27" />

        <info key="var4" value="XYZ" />

      </event>

      <event unitsclass="EU1" unitID="11" date="2000-02-25 12:00:00">

        <info key="var1" value="1.15" />

        <info key="var2" value="EADG" />

      </event>


    </calendar>

  </domain>

</openfluid>
```

2.2.3 Datastore section

The datastore lists external data which is available during the simulation. The datastore content must be defined in a section delimited by the <datastore> tag, and must be structured following these rules:

- Inside the <datastore> tag, there must be a set of <dataitem> tags
- Each <dataitem> tag must bring an ID attribute giving the unique identifier of the dataitem, a type attribute giving the type of the dataitem (only the geovector and georaster types are currently available), and a source attribute giving the source of the dataitem. An optional unitsclass attribute is possible for giving the spatial unit class associated to the data.

```
0
<?xml version="1.0" standalone="yes"?>

<openfluid format="fluidx 4">

  <datastore>
```

```
    <dataitem id="TUlayer" type="geovector" source="TestUnits_wgs84.shp"
              unitsclass="TestUnits" />
    <dataitem id="Ground" type="geovector" source="data/ground.shp" />
    <dataitem id="Ground" type="georaster" source="data/DEM.tiff" />

  </datastore>
</openfluid>
```

2.2.4 Monitoring section

The monitoring is defined by a set of observers that will be automatically plugged and executed by the OpenFLUID environment. Observers are usually used for exporting formatted data from the simulation or performs continuous control during the simulation.

Note

OpenFLUID provides observers for exporting data to CSV formatted files, KML formatted files (for use with Google Earth), and DOT formatted files (for graph representations).

The monitoring must be defined in a section delimited by the `<monitoring>` tag, and must be structured following these rules:

- Inside the `<monitoring>` tag, there may be a set of `<observer>` tags
- Each `<observer>` tag must bring an `ID` attribute giving the identifier of the observer; the value of the `ID` attribute must match the ID of an available and pluggable observer. It also brings an `enabled` tag giving the active state of the simulator (if missing, default value is *1* for active).
- Each `<observer>` tag may include zero to many `<param>` tags giving parameters to the observer. Each `<param>` tag must bring a `name` attribute giving the name of the parameter and a `value` attribute giving the value of the parameter.

Note

Refer to observers signatures for details about specific parameters for each observer.

```
0
<?xml version="1.0" standalone="yes"?>
<openfluid format="fluidx 4">
  <monitoring>

    <observer ID="export.vars.files.csv" enabled="1">
      <param name="format.f1.header" value="colnames-as-comment" />
      <param name="format.f1.date" value="%Y-%m-%d %H:%M:%S" />
      <param name="format.f1.precision" value="8" />

      <param name="format.f2.header" value="full" />

      <param name="set.s1.unitsclass" value="TestUnits" />
      <param name="set.s1.unitsIDs" value="*" />
      <param name="set.s1.vars" value="*" />
      <param name="set.s1.format" value="f1" />


      <param name="set.s2.unitsclass" value="TestUnits" />

      <param name="set.s2.unitsIDs" value="5;3;11" />

      <param name="set.s2.vars" value="tests.double;tests.string" />

      <param name="set.s2.format" value="f2" />

    </observer>


    <observer ID="export.vars.files.kml-anim" enabled="1">

      <param name="layers.anim.unitsclass" value="TestUnits" />

      <param name="layers.anim.varname" value="tests.double" />

      <param name="layers.anim.sourcetype" value="file" />

      <param name="layers.anim.sourcefile" value="TestUnits_wgs84.shp" />

      <param name="layers.anim.linewidth" value="4" />

      <param name="layers.anim.colorscale"

        value="ff00ff00;14;ff00ff76;18;ff00ffdc;22;ff00faff;26;ff0099ff;28;ff001cff"/>


      <param name="layers.static.1.unitsclass" value="OtherUnits" />

      <param name="layers.static.1.sourcetype" value="file" />

      <param name="layers.static.1.sourcefile" value="OtherUnits_wgs84.shp" />

      <param name="layers.static.1.linewidth" value="3" />

      <param name="layers.static.1.color" value="ffffffff" />

    </observer>


  </monitoring>

</openfluid>
```

Warning

> There must be only one monitoring definition in the input dataset.

2.2.5 Run configuration section

The configuration of the simulation gives the simulation period, the default coupling time step and the optional coupling constraint. The run configuration must be defined in a section delimited by the `<run>` tag, and must be structured following these rules:

- Inside the `<run>` tag, there must be a `<scheduling>` tag giving the scheduling informations of the model coupling.
- The `<scheduling>` tag must bring a `deltat` attribute giving the number of second for the default DeltaT time step, and a `constraint` attribute giving an optional constraint applied to the coupling. The values for the `constraint` attribute can be `none` for no constraint, `dt-checked` to check that coupling is synchronized with the default DeltaT time step, `dt-forced` to force coupling at the default DeltaT time step ignoring the scheduling requests from simulators or generators.
- Inside the `<run>` tag, there must be a `<period>` tag giving the simulation period.

- The `<period>` tag must bring a `begin` and an `end` attributes, giving the dates of the beginning and the end of the simulation period. The date format for these attributes must be `YYYY-MM-DD hh:mm:ss`
- Inside the `<run>` tag, there may be a `<valuesbuffer>` tag for the number of produced values kept in memory. The number of values is given through a `size` attribute. If not present, all values are kept in memory.

```
0
<?xml version="1.0" standalone="yes"?>

<openfluid format="fluidx 4">

  <run>


    <scheduling deltat="3600" constraint="none" />

    <period begin="2000-01-01 00:00:00" end="2000-06-30 23:59:00" />


    <valuesbuffer size="10" />


  </run>

</openfluid>
```

2.3 Runtime variables in parameters

Parameters of simulators and observers can include variables that will be replaced by corresponding values at runtime. These variables are :

- `${dir.input}` is replaced by the complete path to the input dataset directory
- `${dir.output}` is replaced by the complete path to the output results directory
- `${dir.temp}` is replaced by the complete path to the directory dedicated to temporary files

```
0
<?xml version="1.0" standalone="yes"?>

<openfluid format="fluidx 4">

  <model>


    <gparams>

      <param name="globaldata" value="${dir.input}/data/global" />

    </gparams>


    <simulator ID="example.simulatorA" >

      <param name="temppath" value="${dir.temp}/simA" />

    </simulator>


  </model>

</openfluid>
```

2.4 Example of an input dataset as a single FluidX file

```
0
<?xml version="1.0" standalone="yes"?>
<openfluid format="fluidx 4">


  <model>
    <gparams>
      <param name="gparam1" value="100" />
      <param name="gparam2" value="0.1" />
    </gparams>
    <simulator fileID="example.simulatorA" enabled="1"/>
    <generator varname="example.generator.fixed" unitsclass="EU1"
               method="fixed" varsize="11" enabled="1">
      <param name="fixedvalue" value="20" />
    </generator>
    <generator varname="example.generator.random" unitsclass="EU2"
               method="random" enabled="1">
      <param name="min" value="20.53" />
      <param name="max" value="50" />
    </generator>
    <simulator fileID="example.simulatorB" enabled="0">
      <param name="param1" value="strvalue" />
      <param name="param2" value="1.1" />
      <param name="gparam1" value="50" />
    </simulator>
  </model>



  <domain>


    <definition>
      <unit class="PU" ID="1" pcsorder="1" />
      <unit class="EU1" ID="3" pcsorder="1">
        <to class="EU1" ID="11" />
        <childof class="PU" ID="1" />
      </unit>
      <unit class="EU1" ID="11" pcsorder="3">
        <to class="EU2" ID="2" />
      </unit>
      <unit class="EU2" ID="2" pcsorder="1" />
    </definition>

    <attributes unitsclass="EU1" colorder="indataA">
      3 1.1
      11 7.5
    </attributes>

    <attributes unitsclass="EU2" colorder="indataB1;indataB3">
      2 18 STRVALX
    </attributes>

    <calendar>
      <event unitsclass="EU1" unitID="11" date="1999-12-31 23:59:59">
        <info key="when" value="before" />
        <info key="where" value="1" />
        <info key="var1" value="1.13" />
        <info key="var2" value="EADGBE" />
      </event>
      <event unitsclass="EU2" unitID="3" date="2000-02-05 12:37:51">
        <info key="var3" value="152.27" />
        <info key="var4" value="XYZ" />
      </event>
      <event unitsclass="EU1" unitID="11" date="2000-02-25 12:00:00">
        <info key="var1" value="1.15" />
        <info key="var2" value="EADG" />
      </event>
    </calendar>

  </domain>


  <run>
    <scheduling deltat="3600" constraint="none" />
    <period begin="2000-01-01 00:00:00" end="2000-06-30 23:59:00" />
    <valuesbuffer size="10" />
  </run>


  <monitoring>
    <observer ID="export.vars.files.csv" enabled="1">
```

```
        <param name="format.f1.header" value="colnames-as-comment" />
        <param name="format.f1.date" value="%Y-%m-%d %H:%M:%S" />
        <param name="format.f1.precision" value="8" />
        <param name="format.f2.header" value="full" />
        <param name="set.s1.unitsclass" value="TestUnits" />
        <param name="set.s1.unitsIDs" value="*" />
        <param name="set.s1.vars" value="*" />
        <param name="set.s1.format" value="f1" />
        <param name="set.s2.unitsclass" value="TestUnits" />
        <param name="set.s2.unitsIDs" value="5;3;11" />
        <param name="set.s2.vars" value="tests.double;tests.string" />
        <param name="set.s2.format" value="f2" />
      </observer>
      <observer ID="export.vars.files.kml-anim" enabled="1">
        <param name="layers.anim.unitsclass" value="TestUnits" />
        <param name="layers.anim.varname" value="tests.double" />
        <param name="layers.anim.sourcetype" value="file" />
        <param name="layers.anim.sourcefile" value="TestUnits_wgs84.shp" />
        <param name="layers.anim.linewidth" value="4" />
        <param name="layers.anim.colorscale"
          value="ff00ff00;14;ff00ff76;18;ff00ffdc;22;ff00faff;26;ff0099ff;28;ff001cff"/>
        <param name="layers.static.1.unitsclass" value="OtherUnits" />
        <param name="layers.static.1.sourcetype" value="file" />
        <param name="layers.static.1.sourcefile" value="OtherUnits_wgs84.shp" />
        <param name="layers.static.1.linewidth" value="3" />
        <param name="layers.static.1.color" value="ffffffff" />
      </observer>
    </monitoring>

  </openfluid>
```

Part II

Development of OpenFLUID simulators

Chapter 3

Overview of an OpenFLUID simulator

Technically, an OpenFLUID simulator is made of two main parts: the signature (in a .json file) and a C++ class containing the computational code in a C++ file (.cpp). They must be compiled before being used in the OpenFLUID software environment.

3.1 Simulator signature

The signature of a simulator contains meta-informations about the simulator. These informations will be mainly used by the OpenFLUID framework for automatic coupling and consistency checking of the simulators of the coupled model. To get more informations about the simulators signatures and how to declare them, see part Declaration of the simulator signature.

3.2 Simulator C++ class

The computational part of a simulator is defined by a class, inherited from the **openfluid::ware::PluggableSimulator** class. The simulation code have to be written into the different methods provided by the **openfluid::ware::PluggableSimulator** class. You can also develop other methods in order to organize your source code.
To get more information about the C++ class of a simulator, see part Creation of a simulator.

3.2.1 Constructor and destructor

The constructor of the simulator is called when the simulator is loaded. You may put here the initialization of the private members of the simulator C++ class.
The destructor of the simulator is called when the simulator is released after simulation. You may put here instruction to free any dynamic memory allocated for the needs of the computational code (dynamic variables or objects, ...).

3.2.2 Mandatory methods to be defined

The class of a simulator must define the following methods:

- **initParams**
- **prepareData**
- **checkConsistency**
- **initializeRun**
- **runStep**
- **finalizeRun**

The **initParams** method is used to retreive the parameters of the simulator, read from the model.fluidx file or filled from the OpenFLUID-Builder interface (See Model section). Once read, the values should be stored into private attributes to be accessed by other methods.

The **prepareData** method can be used to perform data pre-processing before the consistency checking.

The **checkConsistency** method is called during the global consistency checking phase. It can be used to add specific consistency checking for the simulator.

The **initializeRun** method is used for initialization of simulation variables, or to compute initialization data.

The **runStep** method is called at each exchange time step. It should contain the main computational code.

The **finalizeRun** method should be used to do post-processing after simulation. It is the last method ran.

Note

Any of theses simulators methods can be empty if there is no relevant code to add for the method.

Chapter 4

Creation of a simulator

The minimal source code of an OpenFLUID simuilator is made of a C++ file and a build configuration for CMake tool. Using the CMake build tool, the simulator source code is built into a binary plugin for OpenFLUID and automatically installed in the dedicated location to be usable by the OpenFLUID platform.

See also the Organization of an OpenFLUID workspace appendix for sources codes location in workspaces.

4.1 Required tools for development environment

In order to build and develop a simulator, the following tools are required:

- GCC as the C++/C/Fortran compiler (version 7 or later for C++17 compatibility)
- CMake as the build configuration tool (version 3.10 or later). OpenFLUID provides a CMake module to ease the build of simulators.

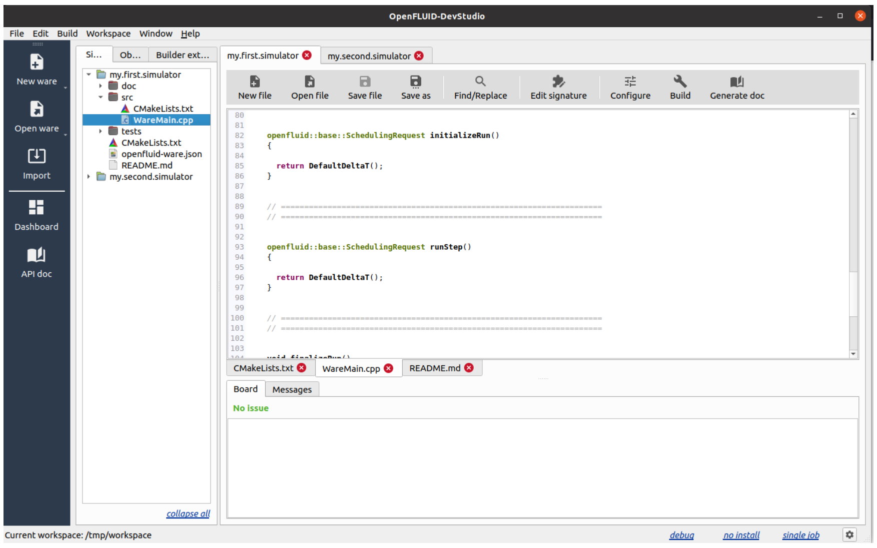
Detailed instructions for installation of these tools are available on the OpenFLUID Community web site (`http://community.openfluid-project.org`).
Even if simulators can be developped using any text editor, the OpenFLUID-DevStudio is the recommended environment for simulators development.

4.2 Creation of a simulator using OpenFLUID-DevStudio

Note

> As the OpenFLUID-DevStudio UI is multilingual, the items cited below such as menu names or labels can be in another language than english for you installation.

The OpenFLUID-DevStudio application is made of a main toolbar located on left, a file navigator on the left side and a file editor on the right side.



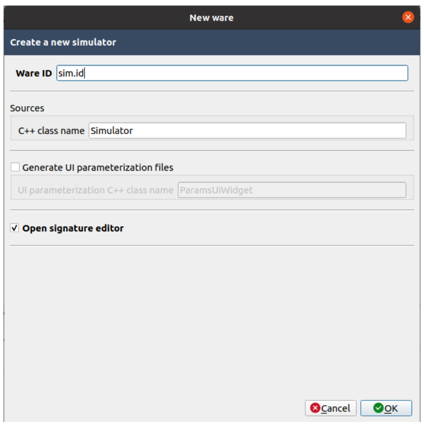
```
  openfluid::base::SchedulingRequest initializeRun()
  {

    return DefaultDeltaT();
  }


  // =====================================================================
  // =====================================================================


  openfluid::base::SchedulingRequest runStep()
  {

    return DefaultDeltaT();
  }


  // =====================================================================
  // =====================================================================
```

No issue

Current workspace: /tmp/workspace

Figure 4.1: Screenshot of OpenFLUID-DevStudio workspace

To create a new simulator, go to menu *File* > *New ware* > *Simulator...* This opens the new simulator dialog. In this dialog, set the simulator ID and source files names then click *OK*. The Source code of a new simulator is created.


New ware

Create a new simulator

Ware ID: sim.id

Sources

C++ class name: Simulator

☐ Generate UI parameterization files

UI parameterization C++ class name: ParamsUiWidget

☑ Open signature editor

Figure 4.2: Screenshot of new simulator dialog

4.2.1 Configuration phase

Once created, the configuration phase must be performed at least once. Click on the *Configure* button of the main toolbar. This phase checks the dependencies (tools and libraries) required to build the simulator. It can be performed either in *Release* mode for performance optimization (mode by default, recommended) or in *Debug* mode to be used with an external debugger.

4.2.2 Build phase

The build phase must be performed each time the source code has been modified. Once the configure process is completed, click on the *Build* button to effectively build the simulator.
This phase builds thes simulator source code into a binary plugin for the OpenFLUID platform. It can be performed either in *Build and install* mode to make the simulator immediately available for simulations (mode by default, recommended) or in *Build only* mode for intermediate builds for example.

4.3 Complete source code example

The example below show a complete example of an empty simulator, including source code and build configuration using the OpenFLUID CMake module.

4.3.1 File ExampleSimulator.cpp containing the simulator source code

```
0
#include <openfluid/ware/PluggableSimulator.hpp>



class ExampleSimulator : public openfluid::ware::PluggableSimulator

{

  private:



  public:



    ExampleSimulator(): PluggableSimulator()

    {

      // Here is source code for constructor

    }



    // ======================================================================

    // ======================================================================



    ~ExampleSimulator()

    {

      // Here is source code for destructor

    }



    // ======================================================================

    // ======================================================================
    void initParams(const openfluid::ware::WareParams_t& /*Params*/)
    {
      // Here is source code for processing simulator parameters
    }


    // =====================================================================
    // =====================================================================


    void prepareData()
    {
      // Here is source code for data preparation
    }


    // =====================================================================
    // =====================================================================


    void checkConsistency()
    {
      // Here is source code for specific consistency checking
    }


    // =====================================================================
    // =====================================================================


    openfluid::base::SchedulingRequest initializeRun()
    {
      // Here is source code for initialization

      return DefaultDeltaT();
    }


    // =====================================================================
    // =====================================================================



    openfluid::base::SchedulingRequest runStep()

    {

      // Here is source code for each time step


      return DefaultDeltaT();

    }




    // =====================================================================

    // =====================================================================



    void finalizeRun()

    {

      // Here is source code for finalization

    }


};



DEFINE_SIMULATOR_CLASS(ExampleSimulator)
```

4.3.2 File CMakeLists.txt containing the ware configuration

```
0
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)


PROJECT("my.simulator.id")


FIND_PACKAGE(OpenFLUID REQUIRED)


ADD_SUBDIRECTORY(src)

ADD_SUBDIRECTORY(doc)

ADD_SUBDIRECTORY(tests)
```

4.3.3 File src/CMakeLists.txt defining the plugin build

```
0
# set this to add include directories
# ex: SET(WARE_INCLUDE_DIRS /path/to/include/A/ /path/to/include/B/)
#SET(WARE_INCLUDE_DIRS )

# set this to add libraries directories
# ex: SET(WARE_LIBRARY_DIRS /path/to/libA/ /path/to/libB/)
#SET(WARE_LIBRARY_DIRS )

# set this to add linked libraries
# ex: SET(WARE_LINK_LIBS libA libB)
#SET(WARE_LINK_LIBS )

# set this to add definitions
# ex: SET(WARE_DEFINITIONS "-DDebug")
#SET(WARE_DEFINITIONS )

# set this to list the extra files or directories to scan for strings to translate
#SET(I18N_FILES_EXTRASCANS )



OPENFLUID_ADD_WAREPLUGIN(
  # list of C++ files
  CPP_FILES WareMain.cpp
  # list of Fortran files
  #FORTRAN_FILES
  # list of UI files (mainly for builderexts)
  #UI_FILES
  # list of RC files (mainly for builderexts)
  #RC_FILES
  # list of languages for translation
  #I18N_LANGS
  # list of supplementary OpenFLUID libraries (e.g. tools, ...)
  #OPENFLUID_COMPONENTS
  # custom target name, automatically generated if not provided
  #TARGET
  # custom installation path, standard path is used if not provided
  #INSTALL_PATH
  # parametrized signature and/or documentation (ON/OFF, OFF is default)
  CONFIGURED_SIGNATURE OFF

  # enable build of parameterization UI (ON/OFF, OFF is default)
  WITH_PARAMSUI OFF

  # list of C++ files for parametrization UI

  #PARAMSUI_CPP_FILES

  # list of UI files for parametrization UI

  #PARAMSUI_UI_FILES

  # list of RC files for parametrization UI

  #PARAMSUI_RC_FILES

  # list of supplementary OpenFLUID libraries (e.g. tools, ...) for parametrization UI

  #PARAMSUI_OPENFLUID_COMPONENTS

  # custom target name for parametrization UI, automatically generated by default

  #PARAMSUI_TARGET

  # custom installation path for parametrization UI, standard path is used by default

  #PARAMSUI_INSTALL_PATH

)
```

Chapter 5

Declaration of the simulator signature

Since OpenFLUID 2.2.0, the signature is defined in a dedicated file called ”openfluid-ware.json”. It is located at the root of the simulator directory. **It is advised to use DevStudio dialog via ”Signature” button to edit its information instead of working directly with json content.** This file contains the same information than before but in a tree structure:

```
0
{
  "id": "traffic.surf.car-transfer",
  "name": "Road Unit (RU) function transfert and stockage for cars",
  "description": "",
  "version": "1.0",
  "status": "experimental",
  "authors": [
    {"name": "MR", "email": "m.r@inrae.fr"}
  ],
  "tags": ["examples"],
  "simulator": {
    ...
  }
}
```

Note

From OpenFLUID 2.1 to 2.2: In most cases, the migration of a simulator can be done automatically through DevStudio ”Try to migrate” button or command line ”openfluid migrate-ware”. This migration will automatically convert the signature from the C++ file into a corresponding json file. For more information, see the `page dedicated to ware migration on Community`.

Note

For advanced users: It is possible to use template inside signature, see Parametric signature annex

5.1 Identification

The identification part of the signature must contain at least the ID of the simulator. This ID will be used by the framework to load simulators. It is declared in the signature with key **id**. Other optional informations can be included for better description of the simulator:

- the simulator name as string, with key **name**. Allowing to assign a long name to the simulator
- the simulator description as string with key **description**. Allowing to provide a detailed description of what the simulator actually does
- the name(s) of the author(s) and corresponding email address(es), as list with key **authors**
- the software version of the simulator, as string with key **version**
- the software status of the simulator, as string with key **status**. The value can be *experimental*, *beta* or *stable*

See the Complete signature example part for detailed example.

5.2 Dependencies

A specific field `dependencies` can be filled (it was previously known as `external-deps` in `wareinfo.json`). This field should contain any system dependency required by the ware to be built or run.
This field value must be a dictionary, eg `{"lib1":">=1.2", "tool2":"*"}`. For now, it is purely informative since there is not any control is done internally to check that these requirements are filled by the compilation environment.
It is advised to use the same version syntax than here: `https://docs.npmjs.com/cli/v10/configuring-npm/p`

5.3 Informations about scientific application

The informations about scientific applications are only indicative. It has no effects on simulator consistency or computational code. They are stored as string list with key **tags**. These informations can be:

- the domain in which the simulator can be applied, eg `"domain::urban"`
- the processes simulated by the simulator, eg `"process::traffic"`
- the numerical methods used by the simulator, eg `"method::mseytoux"`

or other relevant tags.

5.4 Data

The data used by the simulators can be:

- Parameters that are attached to the simulator
- Spatial attributes that are attached to spatial units, giving properties about the spatial units
- Simulation variables that are attached to spatial units, representing the resulting dynamics of modeled processes over the spatial units

- Discrete events that are attached to spatial units, representing the events occurring at a given date and time on a given spatial unit
- Specific file(s) loaded by the simulator

These data can be accessed, appended and/or modified by the simulator.

The spatial graph representing the landscape can also be accessed or modified by simulators during simulations.

The declarations of spatial data access include constraint levels:

- *required*, this means that the data must be available or already produced
- *used*, this means that the data are used only if they are available or already produced

Simulator parameters, variables and attributes are declared in the ”simulator” part of the json file, in a ”data” dictionary:

```
0
"data": {

  "parameters": {

    "used": [

      {

        "name": "MultiCapacity",

        "description": "Multiplying factor for capacity",

        "siunit": "-", "type": ""

      }

    ]

  },

  "attributes": {

    ...

  },

  "variables": {

    "produced": [

      {

        "name": "examples.RU.S.stock",

        "unitsclass": "RU",

        "description": "number of cars stocked on RU",

        "siunit": "",

        "type": ""

      }

    ],

    "used": [

      {

        "name": "examples.TLU.S.state",

        "unitsclass": "TLU",

        ...

      }
      ]
    }
    ...
  }
```

5.4.1 Simulator parameters

Simulator parameters are values provided to each simulator, and are declared using the **DECLARE_REQUIRED_PARAMETER** or **DECLARE_USED_PARAMETER** instructions. These instructions takes 3 arguments

- the name of the parameter
- the description of the parameter (may be empty)
- the SI unit of the parameter (may be empty)

Example of a declaration of a required simulator parameter:

```
0
"used": [
          {
            "name": "MultiCapacity",
            "description": "Multiplying factor for capacity",
            "siunit": "-",
            "type": ""
          }
        ]
```

5.4.2 Spatial attributes

Spatial attributes are constant properties attached to each spatial units, and are declared in three blocks: ”required”, ”used” or ”produced”.
These instructions take 4 arguments:

- the name of the attribute
- the units class
- the description of the attribute (may be empty)
- the SI unit of the attribute (may be empty)

5.4.3 Simulation variables

Simulation variables are attached to spatial units. They are produced, accessed and modified by simulators during simulations.
Accessed variables are declared in 4 blocks: ”produced”, ”required”, ”used” and ”updated”.
These instructions take 4 arguments:

- the name of the variable
- the concerned unit class

- the description of the variable (may be empty)
- the SI unit of the variable (may be empty)

These variables can be typed or untyped. When they are declared in the signature, the variable names suffixed by the [] symbol with a type name enclosed are typed, i.e. each value for the variable must match the type of the variable. If no type is mentioned, values for the variable can be of any type.
In case of typed variables, the type of a required or used variable by a simulator must match the type of the variable set when it is produced.

The type name for a declaration of a variable can be:

- `boolean` for boolean values
- `integer` for long integer values
- `double` for double precision values
- `string` for string values
- `vector` for vector data
- `matrix` for matrix data
- `map` for associative key-value data
- `tree` for hierarchical key-value data

These scenarios of variable exchanges between two A and B simulators run successfully:

- simulator A produces an *untyped* variable *var1*, simulator B requires/uses/updates an *untyped* variable *var1*
- simulator A produces a *typed* variable *var1*, simulator B requires/uses/updates an *untyped* variable *var1*
- simulator A produces a *typed* variable *var1* of type `double`, simulator B requires/uses/updates a *typed* variable *var1* of type `double`

These scenarios of variable exchanges between two simulators are failing:

- simulator A produces an *untyped* variable *var1*, simulator B requires/uses/updates a *typed* variable *var1*
- simulator A produces a *typed* variable *var1* of type `double`, simulator B requires/uses/updates a *typed* variable *var1* of type `matrix`

Example of variable declarations:

```
0
"produced": [

          {

            "name": "examples.RU.S.stock",

            "unitsclass": "RU",

            "description": "number of cars stocked on RU",

            "siunit": "",

            "type": ""

          }

        ]
```

5.4.4 Discrete events

Discrete events are attached to spatial units, They are accessed or appended by simulators during simulations, and are declared in the ”events” block.
Example of events declaration:

```
0
"events": [

  "ZU",

  "YU"

]
```

5.4.5 Extra files

Simulators can declare files that they load and manage. This helps users to provide the needed files, and also
notifies the OpenFLUID framework to check the presence of the file if it is required.
These files are declared in the ”extrafiles” block.
The declaration instruction takes 1 argument: the file name with relative path to the dataset path.
Example of extra file declarations:

```
0
"extrafiles": {

  "required": [

    "SUraindistri.dat",

    "rainsources.xml"

  ],

  "used": []

}
```

5.4.6 Spatial units graph

The spatial units graph representing the landscape can be modified by simulators. These modifications are declared in the signature function using two instructions.

The ”spatial_graph” block is used for declaration of the global units graph modification that will occur during simulation.
Example of declarations for spatial units graph:

```
0
"spatial_graph": {

  "description": "The spatial graph of LU is modified according to wind directions at each time step",

  "details": [

    {

      "unitsclass": "LU",

      "description": "The LU connections are modified according to wind directions, using the neighbours cells attribute

    }

  ]

}
```

5.5 Complete signature example

The signature code below shows an example of a signature for a simulator.

```
0
{
  "id": "traffic.surf.car-transfer",
  "name": "Road Unit (RU) function transfert and stockage for cars",
  "description": "",
  "version": "1.0",
  "status": "experimental",
  "authors": [
    {
      "name": "MR",
      "email": "m.r@inrae.fr"
    }
  ],
  "contacts": [],
  "license": "",
  "tags": [
    "examples",
    "process::traffic",
    "domain::urban"
  ],
  "links": [],
  "issues": [],
  "dependencies": {
    "gdal": ">=3.2"
  },
  "simulator": {
    "data": {
      "parameters": {
        "required": [],
        "used": [
          {
            "name": "MultiCapacity",
            "description": "Multiplying factor for capacity",
            "siunit": "-",
            "type": ""
          }
        ]
      },
      "attributes": {
        "required": [],
        "used": [
          {
            "name": "stockini",
            "unitsclass": "RU",
            "description": "",
            "siunit": "-",
            "type": ""
          },
          {
            "name": "capacity",
            "unitsclass": "RU",
            "description": "",
            "siunit": "car/min",
            "type": ""
          }
        ],
        "produced": []
      },
      "variables": {
        "produced": [
          {
            "name": "examples.RU.S.stock",
            "unitsclass": "RU",
            "description": "number of cars stocked on RU",
            "siunit": "",
            "type": ""
          }
        ],
        "required": [],
        "used": [
          {
            "name": "examples.TLU.S.state",
            "unitsclass": "TLU",
            "description": "traffic light unit state",
            "siunit": "",
            "type": ""
          }
        ],
        "updated": []
      },
      "events": [],
      "extrafiles": {
        "required": [],
        "used": []
      }
    },
    "spatial_graph": {
      "description": "",
      "details": []
    },
    "scheduling": {
      "type": "default",
      "min": 0,
      "max": 0
    }
  }
}
```

Chapter 6

Development of the simulator source code

6.1 General information about simulators architecture

6.1.1 Simulator methods sequence and framework interactions

As previously mentioned, a simulator is a C++ class which defines mandatory methods (see Mandatory methods to be defined). These methods are called by the OpenFLUID framework at the right time during the simulation, following the interactions sequence in the figure below.

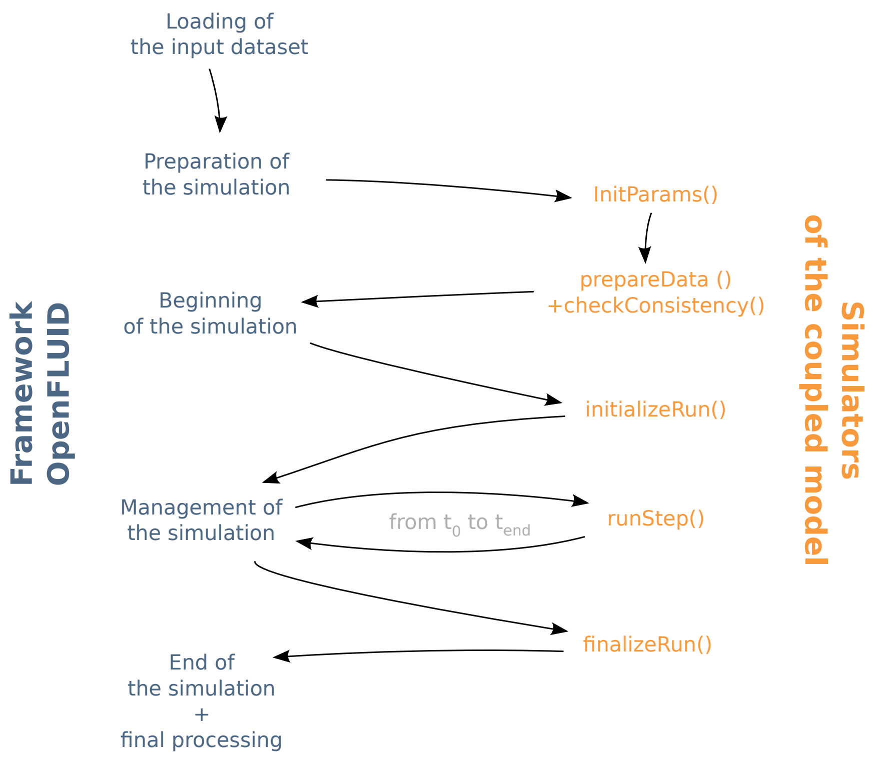


Figure 6.1: Interactions sequence between the OpenFLUID framework and the simulators

Among these methods, the **initializeRun()** and **runStep()** methods have a special behaviour: these two methods must return the simulation duration after which the simulator will be executed again.

This duration can returned using the following instructions :

- **Duration()** to be executed in a number of seconds given as a parameter
- **DefaultDeltaT()** to be executed in a number of seconds given as default DeltaT in the input dataset
- **MultipliedDefaultDeltaT()** to be executed in a number of seconds given as default DeltaT in the input dataset, multiplied by a the value given as parameter
- **AtTheEnd()** to be executed only once at the end of the simulation
- **Never()** to never be executed again

Example for a fixed time step simulator, with a time step equal to the default DeltaT value given in the input dataset:

```
0
openfluid::base::SchedulingRequest initializeRun()

{

  return DefaultDeltaT();

}




openfluid::base::SchedulingRequest runStep()

{

  return DefaultDeltaT();

}
```

Example for a variable time step simulator, based on the internal computation of the simulator:

```
0
openfluid::base::SchedulingRequest initializeRun()

{

  // do something here


  return DefaultDeltaT();

}




openfluid::base::SchedulingRequest runStep()

{

  double TmpValue = 0.0;


  // do something here with TmpValue


  if (TmpValue < 1.0)

  {

    return DefaultDeltaT();

  }

  else
```

```
    {
      return Duration(10);
    }
  }
```

For fully synchronized coupled simulators, all simulators must return the same duration for the next execution, usually **DefaultDeltaT()** .

6.1.2 OpenFLUID data types

Simulation data exchanged through the OpenFLUID framework should be typed with an OpenFLUID defined type.
The available simple types are:

- **openfluid::core::BooleanValue** for storing boolean values
- **openfluid::core::IntegerValue** for storing long integer values
- **openfluid::core::DoubleValue** for storing double precision values
- **openfluid::core::StringValue** for storing string values

The available compound types are:

- **openfluid::core::VectorValue** for storing vector data
- **openfluid::core::MatrixValue** for storing matrix data
- **openfluid::core::MapValue** for storing associative key-value data
- **openfluid::core::TreeValue** for storing hierarchical key-value data

A specific type is available for storing non-existing or null values:

- **openfluid::core::NullValue**

Simulation data are stored using these types :

- Simulation variables : stored as their native type
- Spatial attributes : stored as their native type
- Simulator parameters : stored as openfluid::core::StringValue, and can be converted to any other type
- Informations associated to events : stored as openfluid::core::StringValue, and can be converted to any other type

Each data type can be converted to and from openfluid::core::StringValue (as far as the string format is correct). String representations of values are described in the String representation of values part.
Simulation variables can be typed or untyped. This is set at the declaration of these variables (see Simulation variables).
In case of typed variables, each value of the variable must be of the type of the variable. In case of untyped variables, values for the variable can be of any type.

6.2 Handling the spatial domain

6.2.1 Parsing the spatial graph

The spatial graph represents the spatial domain where coupled simulators will operate. Parsing this graph in different ways is a common task in simulators. This graph can be browsed using predefined instructions.

Sequential parsing

Spatial units can be parsed following the process order by using the following OpenFLUID instructions:

- **OPENFLUID_UNITS_ORDERED_LOOP** for parsing spatial units of a given units class
- **OPENFLUID_ALLUNITS_ORDERED_LOOP** for parsing of all units in the spatial domain

To parse a specific list of of spatial units, you can use the instruction:

- **OPENFLUID_UNITSLIST_LOOP**

The source code below shows examples of spatial graph parsing. The first part of the source code shows how to browse all units of the SU units class, and how to browse the "From" units for each SU unit. The second part of the source code shows how to browse all units of the spatial domain.

```
0
openfluid::base::SchedulingRequest runStep()

{

  openfluid::core::SpatialUnit* SU;

  openfluid::core::SpatialUnit* UU;

  openfluid::core::SpatialUnit* UpSU;

  openfluid::core::UnitsPtrList_t* UpSUsList;

  openfluid::core::DoubleValue TmpValue;


  OPENFLUID_UNITS_ORDERED_LOOP("SU",SU)

  {

    UpSUsList = SU->fromSpatialUnits("SU");


    OPENFLUID_UNITSLIST_LOOP(UpSUsList,UpSU)

    {

      // do something here

      OPENFLUID_GetVariable(UpSU,"varA",TmpValue);

    }

  }


  OPENFLUID_ALLUNITS_ORDERED_LOOP(UU)

  {

    // do something here

    OPENFLUID_GetVariable(UU,"varB",TmpValue);

  }


  return DefaultDeltaT();

}
```

Parallel processing using multithreading

A processing defined as a method of a simulator class can be applied in parallel to the spatial graph, following the process order, using the following methods:

- **APPLY_UNITS_ORDERED_LOOP_THREADED** for applying a process to a given units class. Extra arguments can be passed (see example below).
- **APPLY_ALLUNITS_ORDERED_LOOP_THREADED** for applying a process to a all units of the spatial domain. Extra arguments can also be passed (see example below).

The first argument of the method passed to the instruction must be a pointer to an **openfluid::core::SpatialUnit** as it represents the currently processed spatial unit.

In order to enable the parallel processing in the spatial graph, the following inclusion must be added at the top of the simulator source code:

```
#include <openfluid/ware/ThreadedLoopMacros.hpp>
```

The code below shows how to apply a method in parallel over the spatial graph:

```
void computeA(openfluid::core::SpatialUnit* U)
{
 // compute something
 // can use/produce variables

 openfluid::core::DoubleValue TmpValue;

 OPENFLUID_GetVariable(U,"varA",TmpValue);
}


void computeB(openfluid::core::SpatialUnit* U,
              const double Coeff)
{
 // compute something else, with extra args
 // can use/produce variables

 openfluid::core::DoubleValue TmpValue;

 OPENFLUID_GetVariable(U,"varA",TmpValue);
 OPENFLUID_AppendVariable(U,"varB",TmpValue*Coeff);
}


openfluid::base::SchedulingRequest runStep()
{
  APPLY_UNITS_ORDERED_LOOP_THREADED("SU",SnippetsSimulator::computeA);
  APPLY_UNITS_ORDERED_LOOP_THREADED("TU",SnippetsSimulator::computeB,2.5);
  APPLY_ALLUNITS_ORDERED_LOOP_THREADED(SnippetsSimulator::computeA);


  return DefaultDeltaT();
}
```

Note

- If a spatial loop is used inside another spatial loop, it is recommended to use multithreading in only one loop.
- In case of concurrent data access, it is strongly recommended to use mutex locks for thread-safe data handling.
- Concurrent parsing using multithreading should improve computing performance, reducing simulations durations. But in case of very short computing durations, the cost of multithreading management may counterbalance the performance improvements of concurrent computing.

6.2.2 Querying the spatial graph

The spatial domain graph can be queried during simulations in order to get informations about spatial units and connections.
The following methods are available:

- **OPENFLUID_IsUnitExist**
- **OPENFLUID_IsUnitsClassExist**
- **OPENFLUID_GetUnit**
- **OPENFLUID_GetUnits**
- **OPENFLUID_GetUnitsCount**
- **OPENFLUID_IsUnitConnectedTo**
- **OPENFLUID_IsUnitConnectedFrom**
- **OPENFLUID_IsUnitChildOf**
- **OPENFLUID_IsUnitParentOf**

6.2.3 Modifying the spatial graph

The spatial graph is usually statically defined through the input dataset. It can also be defined and modified dynamically during simulations, using primitives to create and delete spatial units, and also to add and remove connections between these spatial units.
Although the creation, deletion and modification of connections are allowed at any stage of the simulation, the creation, deletion and modification of spatial units are currently allowed only during the data preparation stage (i.e. in the prepareData() method of the simulator).
For consistent use of simulators which modify the spatial domain graph, please fill the signature with the correct directives. See the Spatial units graph part of the signature declaration.

Creating and deleting spatial units

In order to create and delete units, you can use the following methods:

- **OPENFLUID_AddUnit**
- **OPENFLUID_DeleteUnit**

Adding and removing spatial connections

Connections between spatial units can be of two types:

- ”From-To” connections, linking spatial units topologically. These connections are usually used in ”fluxes-like” processes.
- ”Parent-Child” connections, linking units hierarchically.

In order to add and remove connections, you can use the following methods, whenever during simulations:

- **OPENFLUID_AddFromToConnection**
- **OPENFLUID_AddChildParentConnection**
- **OPENFLUID_RemoveFromToConnection**
- **OPENFLUID_RemoveChildParentConnection**

Example:

```
0
void prepareData()
{

 /*
      TU.1         TU.2
        |            |
       -->  TU.22 <--
               |
              --> TU.18
                    |
          TU.52 --> OU.5 <-- OU.13
                    |
                    --> OU.25

       VU1 <-> VU2

   with:
   TU1, TU2, TU22, TU18 are children of VU1
   TU52, OU5, OU13, OU25 are children of VU2
*/

  OPENFLUID_AddUnit("VU",1,1);
  OPENFLUID_AddUnit("VU",2,2);
  OPENFLUID_AddUnit("TU",1,1);
  OPENFLUID_AddUnit("TU",2,1);
  OPENFLUID_AddUnit("TU",22,2);
  OPENFLUID_AddUnit("TU",18,3);
  OPENFLUID_AddUnit("TU",52,1);
  OPENFLUID_AddUnit("OU",5,4);
```

```
    OPENFLUID_AddUnit("OU",13,1);
    OPENFLUID_AddUnit("OU",25,5);


    OPENFLUID_AddFromToConnection("VU",1,"VU",2);
    OPENFLUID_AddFromToConnection("VU",2,"VU",1);
    OPENFLUID_AddFromToConnection("TU",1,"TU",22);
    OPENFLUID_AddFromToConnection("TU",2,"TU",22);
    OPENFLUID_AddFromToConnection("TU",22,"TU",18);
    OPENFLUID_AddFromToConnection("TU",18,"OU",5);
    OPENFLUID_AddFromToConnection("TU",52,"OU",5);
    OPENFLUID_AddFromToConnection("OU",13,"OU",5);
    OPENFLUID_AddFromToConnection("OU",5,"OU",25);


    OPENFLUID_AddChildParentConnection("TU",1,"VU",1);
    OPENFLUID_AddChildParentConnection("TU",2,"VU",1);
    OPENFLUID_AddChildParentConnection("TU",22,"VU",1);
    OPENFLUID_AddChildParentConnection("TU",18,"VU",1);
    OPENFLUID_AddChildParentConnection("TU",52,"VU",2);
    OPENFLUID_AddChildParentConnection("OU",5,"VU",2);
    OPENFLUID_AddChildParentConnection("OU",13,"VU",2);
    OPENFLUID_AddChildParentConnection("OU",25,"VU",2);
  }
```

Generating spatial domain graphs automatically

A spatial domain graph can be automatically built or extended using a provided method to create a matrix-like graph:

- **OPENFLUID_BuildUnitsMatrix**

6.3 Informations about simulation time

Simulators can access to informations about simulation time. There are constant time informations, such as simulation duration or begin and end date, and evolutive informations such as current time index.
Constant time informations can be accessed from any part of the simulator (except from the constructor), using the following methods:

- **OPENFLUID_GetBeginDate** returns the beginning date of the simulation
- **OPENFLUID_GetEndDate** returns the end date of the simulation
- **OPENFLUID_GetSimulationDuration** returns the duration of the simulation (in seconds)
- **OPENFLUID_GetDefaultDeltaT** returns the default time step of the simulation (in seconds), given in the input dataset

Evolutive time informations can be accessed only from specific parts of the simulator, using the following methods:

- **OPENFLUID_GetCurrentTimeIndex** returns the current time index (in seconds) of the simulation, and is available from the initializeRun(), runStep() and finalizeRun() methods of the simulator
- **OPENFLUID_GetCurrentDate** returns the current date of the simulation, and is available from the initializeRun(), runStep() and finalizeRun() methods of the simulator
- **OPENFLUID_GetPreviousRunTimeIndex** returns the time index corresponding to the previous execution of the simulator, and is available from the runStep() and finalizeRun() methods of the simulator

Example of code:

```
0
openfluid::base::SchedulingRequest runStep()

{

  openfluid::core::Duration_t Duration = OPENFLUID_GetSimulationDuration();


  openfluid::core::TimeIndex_t CurrentIndex = OPENFLUID_GetCurrentTimeIndex();

  openfluid::core::DateTime CurrentDT = OPENFLUID_GetCurrentDate();


  std::cout << Duration << std::endl;

  std::cout << CurrentIndex << std::endl;

  std::cout << CurrentDT.getAsISOString() << std::endl;


  return DefaultDeltaT();

}
```

6.4 Simulator parameters

Simulators parameters can be accessed in the source code from the initParams() method of the simulator. Values of simulators parameters can be retreived using:

- **OPENFLUID_GetWareParameter.**

The requested parameter name must be the same as the one declared in the signature and used in the model.fluidx file (see Model section of the signature declaration).

Example of initParams method:

```
0
void initParams(const openfluid::ware::WareParams_t& Params)

{

  m_MyParam = 0; //default value set to the class member

  OPENFLUID_GetWareParameter(Params,"myparam",m_MyParam);

}
```

To be used in other part of the simulator, the C++ variable storing a simulator parameter should be declared as class member. The types of parameters can be string, double, integer, boolean, vector of string, vector of double (see API documentation of OPENFLUID_GetWareParameter method to get more informations about other available types , available on the `OpenFLUID community site`).

6.5 Spatial attributes

In order to access or update values of spatial attributes, or to test if a spatial attribute is present, you can use the following methods:

- **OPENFLUID_GetAttribute** to get the value of an attribute
- **OPENFLUID_SetAttribute** to set the value of an attribute
- **OPENFLUID_IsAttributeExist** to test if an attribute exists

The methods to test if an attribute exists or to access to an attribute value are usable from any simulators part except from the initParams() part. The methods to update an attribute value are only usable from the prepareData() and checkConsistency() parts of the simulator.
The names of the attributes must match the names in the input dataset (see Spatial domain section), or the name of an attribute created by a simulator.
Example of use:

```
0
openfluid::base::SchedulingRequest runStep()
{
  openfluid::core::SpatialUnit* SU;
  openfluid::core::DoubleValue AreaValue;

  OPENFLUID_UNITS_ORDERED_LOOP("SU",SU)
  {
    OPENFLUID_GetAttribute(SU,"area",AreaValue);

    // continue with source code using the value of the area attribute
  }

  return DefaultDeltaT();
}
```

6.6 Simulation variables

The values for the simulation variables are attached to the spatial units.
The available methods to access to simulation variables are:

- **OPENFLUID_GetVariable** to get the value of a variable at the current time index or at a given time index
- **OPENFLUID_GetVariables** to get values of a variable between two times indexes
- **OPENFLUID_GetLatestVariable** to get the latest available value for the variable
- **OPENFLUID_GetLatestVariables** to get the latest values of a variable since a given time index

The available methods to add or update a value of a simulation variable are:

- **OPENFLUID_AppendVariable** to add a value to a variable for the current time index
- **OPENFLUID_SetVariable** to update the value of a variable for the current time index

The available methods to test if a simulation variable exists are:

- **OPENFLUID_IsVariableExist** to test if a variable exists or if a value for this variable exists at the given time index
- **OPENFLUID_IsTypedVariableExist** to test if a variable exists or if a value for this variable exists at the given time index, and its type matches the given type

These methods can be accessed only from the initializeRun(), runStep() and finalizeRun() parts of the simulator.
Example:

```
0
openfluid::base::SchedulingRequest runStep()
{
  openfluid::core::DoubleValue TmpValue;
  openfluid::core::SpatialUnit* SU;


  OPENFLUID_UNITS_ORDERED_LOOP("SU",SU)
  {
    OPENFLUID_GetVariable(SU,"MyVar",TmpValue);
    TmpValue = TmpValue * 2;
    OPENFLUID_AppendVariable(SU,"MyVarX2",TmpValue);
  }


  return DefaultDeltaT();
}
```

6.7 Events

A discrete event is defined by the **openfluid::core::Event** class. It is made of a date and a set of key-value informations that can be accessed by methods proposed by the openfluid::core::Event class.
A collection of discrete events can be contained in an **openfluid::core::EventsCollection** class.

A collection of events occuring during a period on a given spatial unit can be acessed using

- **OPENFLUID_GetEvents**

This method returns an **openfluid::core::EventsCollection** that can be processed.

The returned event collection can be parsed using the specific loop instruction:

- **OPENFLUID_EVENT_COLLECTION_LOOP**

At each loop iteration, the next event can be processed.

An event can be added on a specific spatial unit at a given date using:

- **OPENFLUID_AppendEvent**

Example of process of events occurring on the current time step:

```
0
openfluid::base::SchedulingRequest runStep()
{
  openfluid::core::SpatialUnit* TU;
  openfluid::core::EventsCollection EvColl;
  openfluid::core::Event* Ev;
  openfluid::core::DateTime BTime, ETime;

  BTime = OPENFLUID_GetCurrentDate() - 86400;
  ETime = OPENFLUID_GetCurrentDate();

  OPENFLUID_UNITS_ORDERED_LOOP("TU",TU)
  {
    OPENFLUID_GetEvents(TU,BTime,ETime,EvColl);

    OPENFLUID_EVENT_COLLECTION_LOOP(EvColl.eventsList(),Ev)
    {
      if (Ev->isInfoEqual("molecule","glyphosate"))
      {
        // process the event
      }
    }

  }

  return DefaultDeltaT();
}
```

6.8 Internal state data

In order to preserve the internal state of the simulator between calls (from the run step to the next one for example), internal variables can be stored as class members. The class members are persistant during the whole life of the simulator.
To store distributed values, data structures are available to associate a spatial unit ID to a storedvalue. These data structures exist for different types of data:

- **openfluid::core::IDFloatMap**
- **openfluid::core::IDDoubleMap**
- **openfluid::core::IDIntMap**
- **openfluid::core::IDBoolMap**

- **openfluid::core::IDDoubleValueMap**
- **openfluid::core::IDVectorValueMap**
- **openfluid::core::IDVectorValuePtrMap**
- **openfluid::core::IDSerieOfDoubleValueMap**
- **openfluid::core::IDSerieOfDoubleValuePtrMap**

Example of declaration of ID-map structures in private members of the simulator class:

```
class SnippetsSimulator : public openfluid::ware::PluggableSimulator
{
  private:

    openfluid::core::IDDoubleMap m_LastValue;

  public:

    // rest of the simulator class
```

Example of usage of the ID-map structures:

```
openfluid::base::SchedulingRequest runStep()
{
  int ID;
  double TmpValue;
  openfluid::core::SpatialUnit* SU;

  OPENFLUID_UNITS_ORDERED_LOOP("SU",SU)
  {
    ID = SU->getID();

    TmpValue = TmpValue + m_LastValue[ID];
    OPENFLUID_AppendVariable(SU,"MyVarPlus",TmpValue);

    m_LastValue[ID] = TmpValue;
  }

  return DefaultDeltaT();
}
```

6.9 Runtime environment

The runtime environment of the simulator are informations about the context during execution of the simulation: input and output directories, temporary directory,...
They are accessible from simulators using:

- **OPENFLUID_GetRunEnvironment**

Example:

```
0
openfluid::base::SchedulingRequest initializeRun()

{

  std::string InputDir;


  OPENFLUID_GetRunEnvironment("dir.input",InputDir);


  // the current input directory is now available through the InputDir local variable


  return DefaultDeltaT();

}
```

The keys for requesting runtime environment information are:

- dir.input [string] : the current input directory
- dir.output [string] : the current output directory
- dir.temp [string] : the directory for temporary files
- mode.cleanoutput [boolean] : cleaning output dir before data saving is enabled/disabled

6.10 Informations, warnings and errors

6.10.1 Informations and warnings from simulators

Simulators can emit informations and warnings to both console and files using various methods

- **OPENFLUID_DisplayInfo** to display informative messages to console only
- **OPENFLUID_LogInfo** to log informative messages to file only
- **OPENFLUID_LogAndDisplayInfo** to log and display informative messages simultaneously
- **OPENFLUID_DisplayWarning** to display warning messages to console only
- **OPENFLUID_LogWarning** to log warning messages to file only
- **OPENFLUID_LogAndDisplayWarning** to log and display warning messages simultaneously

Using these methods is the recommended way to log and display messages. Please avoid using std::cout or similar C++ facilities in production or released simulators.
Example:

```
0
openfluid::base::SchedulingRequest initializeRun()

{

  openfluid::core::SpatialUnit* TU;
```

```
    OPENFLUID_UNITS_ORDERED_LOOP("TestUnits",TU)
    {
      OPENFLUID_LogInfo("TestUnits #" << TU->getID());
      OPENFLUID_DisplayInfo("TestUnits #" << TU->getID());

      OPENFLUID_LogWarning("This is a warning message for " << "TestUnits #" << TU->getID());
    }

    return DefaultDeltaT();
  }
```

The messages logged to file are put in the `openfluid-messages.log` file placed in the simulation output directory. This file can be browsed using the OpenFLUID-Builder application (*Outputs* tab) or any text editor.

6.10.2 Errors from simulators

Simulators can raise errors to notify the OpenFLUID framework that something wrong or critical had happened. An error stops the simulation the next time the OpenFLUID framework has the control.

Errors can be raised using **OPENFLUID_RaiseError**
Example:

```
0
void checkConsistency()
{
  double TmpValue;
  openfluid::core::SpatialUnit* SU;

  OPENFLUID_UNITS_ORDERED_LOOP("SU",SU)
  {
    OPENFLUID_GetAttribute(SU,"MyAttr",TmpValue);

    if (TmpValue <= 0)
    {
      OPENFLUID_RaiseError("Wrong value for the MyProp attribute on SU");
    }
  }
}
```

6.11 Debugging

Debugging instructions allow developpers to trace various information during simulations.
They are enabled only when debug is enabled at simulators builds. They are ignored for other build types.

In order to enable debug build mode, the option `-DCMAKE_BUILD_TYPE=Debug` must be added to the cmake command (e.g. `cmake <srcpath> -DCMAKE_BUILD_TYPE=Debug`).

Example of build configuration:

```
cmake .. -DCMAKE_BUILD_TYPE=Debug
```

This debug build mode is disabled using the release build mode, with the option `-DCMAKE_BUILD_TYPE=Release`.

Simulators can emit debug information to both console and files using various methods

- **OPENFLUID_DisplayDebug** to display debug messages to console only
- **OPENFLUID_LogDebug** to log debug messages to file only
- **OPENFLUID_LogAndDisplayDebug** to log and display debug messages simultaneously

Example:

```
openfluid::base::SchedulingRequest runStep()

{

  OPENFLUID_LogDebug("Entering runStep at time index " << OPENFLUID_GetCurrentTimeIndex());


  return DefaultDeltaT();

}
```

Additional instructions are available for debugging, see file debug.hpp:

6.12 Integrating Fortran code

The C++/Fortran interface is defined in the openfluid/tools/FortranCPP.hpp file. It allows to integrate Fortran 77/90 code into simulators.
In order to execute Fortran code from a simulator, the Fortran source code have to be wrapped into subroutines that are called from the C++ code of the simulator.
To help developers of simulators to achieve this wrapping operation, the FortranCPP.hpp file defines dedicated instructions. You are invited to read the FortranCPP.hpp file to get more information about these instructions.

In order to enable the call of Fortran code, the following inclusion must be added at the top of the simulator source code:

```
#include <openfluid/tools/FortranCPP.hpp>
```

Example of Fortran source code (e.g. FortranSubr.f90):

```
subroutine displayvector(Fsize,vect)


implicit none
```

```
integer Fsize,i
real*8 vect(fsize)

write(*,*) 'size',fsize

do i=1,fsize
  write(*,*) vect(i)
end do

return

end
```

Example of declaration block int the .cpp file (e.g. FortranSim.cpp):

```
0
BEGIN_EXTERN_FORTRAN
  EXTERN_FSUBROUTINE(displayvector)(FINT *Size, FREAL8 *Vect);
END_EXTERN_FORTRAN
```

Example of call of the fortran subroutine from the initializeRun method (e.g. FortranSim.cpp):

```
0
openfluid::base::SchedulingRequest initializeRun()
{
  openfluid::core::VectorValue MyVect;

  MyVect = openfluid::core::VectorValue(15,9);
  int Size = MyVect.getSize();

  CALL_FSUBROUTINE(displayvector)(&Size,(MyVect.data()));

  return DefaultDeltaT();
}
```

The compilation and linking of Fortran source code is automatically done when adding fortran source files to the SIM_FORTRAN variable in the CMake.in.config file (See dev_createsim_exmpl_config).

6.13 Embedding R code

Note

> The embedding of R code in simulators is currently an experimental feature.

Thanks to the `RInside package`, It is possible to embed R code in simulators written in C++. It also relies on the `Rcpp package` for handling data from and to the `R environment`.
In order to embed R code using RInside, the following inclusion must be added at the top of the simulator source code:

```
#include <RInside.h>
```

A unique RInside variable is used to run R code, it should be declared as a member of the simulator class (named `m_R` in this example).

```
class SnippetsSimulator : public openfluid::ware::PluggableSimulator
{

  private:

    RInside m_R;

  public:

    // rest of the simulator class
```

The R environment can be acessed through the RInside variable and R commands can be run using its `parseEvalQ()` method.

```
m_R["varA"] = 1.2;
m_R["varB"] = 5.3;
m_R.parseEvalQ("varC = max(varA,varB)");
// value of the R varC variable can be accessed from C++ through the m_R["varC"] variable
```

In this short example, simple variables and commands are used. It is possible to perform complex operations involving external R packages, or call R scripts by executing a `source()` R command through RInside. See the `RInside package` documentation to get more details and examples.
To help configuring the simulator which is using the RInside package, a CMake module is provided with OpenFLUID to setup the configuration variables when building the simulator. It should be used in the `CMake.in.cmake` file of the simulator.

```
# Simulator ID
# ex: SET(SIM_ID "my.simulator.id")
SET(SIM_ID "example.simulator")


# list of CPP files, the sim2doc tag must be contained in the first one
# ex: SET(SIM_CPP MySimulator.cpp)
SET(SIM_CPP ExampleSimulator.cpp)
```

```
# list of Fortran files, if any
# ex: SET(SIM_FORTRAN Calc.f)
#SET(SIM_FORTRAN )



# list of extra OpenFLUID libraries required
# ex: SET(SIM_OPENFLUID_COMPONENTS tools)
SET(SIM_OPENFLUID_COMPONENTS )

# set this to add include directories
# ex: SET(SIM_INCLUDE_DIRS /path/to/include/A/ /path/to/include/B/)
#SET(SIM_INCLUDE_DIRS )

# set this to add libraries directories
# ex: SET(SIM_INCLUDE_DIRS /path/to/libA/ /path/to/libB/)
#SET(SIM_LIBRARY_DIRS )

# set this to add linked libraries
# ex: SET(SIM_LINK_LIBS libA libB)
#SET(SIM_LINK_LIBS )

# set this to add definitions
# ex: SET(SIM_DEFINITIONS "-DDebug")
#SET(SIM_DEFINITIONS )



# unique ID for linking parameterization UI extension (if any)
#SET(WARE_LINK_UID "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}")

# set this to ON to enable parameterization widget
# ex: SET(SIM_PARAMSUI_ENABLED ON)
SET(SIM_PARAMSUI_ENABLED OFF)

# list of CPP files for parameterization widget, if any
# ex: SET(SIM_PARAMSUI_CPP MyWidget.cpp)
SET(SIM_PARAMSUI_CPP )

# list of UI files for parameterization widget, if any
# ex: SET(SIM_PARAMSUI_UI MyWidget.ui)
SET(SIM_PARAMSUI_UI )
```

```
# list of RC files for parameterization widget, if any
# ex: SET(SIM_PARAMSUI_RC MyWidget.rc)
SET(SIM_PARAMSUI_RC )



# set this to ON to enable translations
#SET(SIM_TRANSLATIONS_ENABLED ON)

# set this to list the languages for translations
#SET(SIM_TRANSLATIONS_LANGS fr_FR)

# set this to list the extra files or directories to scan for strings to translate
#SET(SIM_TRANSLATIONS_EXTRASCANS )



# set this to force an install path to replace the default one
#SET(SIM_INSTALL_PATH "/my/install/path/")



# set this if you want to use a specific sim2doc template
#SET(SIM_SIM2DOC_TPL "/path/to/template")



# set this if you want to add tests
# given tests names must be datasets placed in a subdir named "tests"
# each dataset in the subdir must be names using the test name and suffixed by .IN
# ex for tests/test01.IN and tests/test02.IN: SET(SIM_TESTS_DATASETS test01 test02)
#SET(SIM_TESTS_DATASETS )
```

An adjustment of the CMake module path prefix may be required to find the provided R module for CMake

```
0
SET(CMAKE_MODULE_PATH "/prefix/lib/openfluidhelpers/cmake;${CMAKE_MODULE_PATH}")
```

where `prefix` depends on the OpenFLUID installation path and operating system (e.g. `/usr`,`/usr/local`, `C:\OpenFLUID-x.x.x`)

Warning

> Due to limitations of the RInside package, embedding R code in simulators does not support threading.
>
> Simulations including simulators with RInside must be run using Command line mode in OpenFLUID-Builder or using the `openfluid` command line.
>
> Due to limitations of the RInside package, only one simulator embedding R code can be used in a coupled model.

6.14 Miscellaneous helpers

The OpenFLUID API provides miscellaneous functions and classes to help simulators developpers in their setup of data processing or numerical computation. They are available in various namespaces:

- openfluid::tools
- openfluid::scientific
- openfluid::utils

In order to use these helpers, the corresponding headers files must be included in the simulator source code. As they are not detailed here in this manual, more informations about these helpers are available in the provided header files (.hpp), located in the corresponding include directories.

Chapter 7

Handling code redundancy

7.1 Use case

When working on several simulators, it often appears that parts are redundant between them. These identical codes can become a problem when a change is required, it has to be done in each simulator, which generates a real risk of discrepency.

7.2 Git solution: Submodule

To solve these cases, we decided to use a git feature named `Submodule` that allows us to encapsulate this common code in a dedicated location with its own versioning history. This feature fits well our existing use of `git` as versioning control system and it can be integrated in our existing workflows.
For now, OpenFLUID environment can only deal with simple fragments with two constraints:

- they should be *header only*, meaning that it can only contain .h/.hpp files containing all the code and not .c/.cpp files.
- they can not use functions or helpers depending on the PluggableSimulator (but other helper functions remain available)

The current handling of fragments is sufficient for many use cases, such as:

- using the same solver function,
- share global variables (even if the excessive use of global variables is strongly discouraged!)
- provide a class for a common object structure

Example of simulator structure including a fragment:

```
0
this.very.simulator

 .git

 .gitmodules

 README.md

 CMakeLists.txt

 openfluid-ware.json

 doc/

 src

     WareMain.cpp

     CMakeLists.txt

     fragments
```

```
        fragmentName (https://gitlab.abc/fragmentName)

            .git

            myCommonFunctions.h

            openfluid-ware.json

            doc/
```

7.3 Impact on code

Source files import are slightly different when using a fragment, compilation system triggered by the user knows the location `src/fragments/` and syntax such as `#include <fragmentName/myCommonFunctions.hpp>` can be used.

7.4 Operations

7.4.1 In DevStudio

To add a new fragment or import one from a distant repository, right-click on the simulator and chose the wanted action in the ”add fragment” submenu. If you want to add an existing fragment, you will then be able to add it from a Hub or from a direct git repository.
It is also possible to remove a submodule by doing a right-click on the targetted fragment and select the ”remove fragment” option.

7.4.2 In a console

For operations such as:

- pushing commits on the fragment,
- adding a fragment to a simulator,
- updating the state of a fragment contained in a simulator see the dedicated page for direct console git operations (in french for now: Gestion directe de simulateur avec fragments de code)

7.4.3 Good practices

- Metadata: a `openfluid-fragment.json` file should be added at root directory of the fragment to describe useful information, especially the field `openfluid-components` in `fragment` sub-dictionary to inform the simulator developer to add any required OpenFLUID components to ware `CMakeLists.txt` file.
- Namespace: it is advised to use a two level namespace to encapsulate the code with ”fragment” as first level. For example:

```
0
namespace fragment { namespace hydro {


double myHydroVar = 9000;


int myHydroFunction(int param1, double param2)

{

    ...
}


myHydroClass

{

    ...

};


} }  // end of namespace
```

these function will then be usable in the simulator code as follows:

```
0
double b = fragment::hydro::myHydroVar + 2;

int result = fragment::hydro::myHydroFunction(3, b);
```

Chapter 8

Documenting simulators

The scientific documentation of simulators is important to clearly describe the scientific concepts and methods applied in source code of simulators.

8.1 docalyze (OpenFLUID $>=$ 2.2.0)

In order to facilitate the writing and maintenance of these documentation, OpenFLUID provides the docalyzer system for simulators designers and developers.

The documentation can be provided inside a `README.md` file located at simulator root folder, or inside `doc/` directory. It can handle LateX (as before), but also markdown and Rmarkdown.
These files will be converted into a pdf file by using the *pandoc* tool.
Docalyze operation can be done through DevStudio ware operation "Build doc" or from command-line:

```
0
openfluid docalyze --src-path=<simulator_path> --output-path=<output_path>
```

8.2 sim2doc (OpenFLUID $<$ 2.2)

In order to facilitate the writing and maintenance of these documentation, OpenFLUID provides the Sim2Doc system for simulators designers and developers.

The Sim2Doc system uses the simulator signature and an optional LaTeX-formatted text to build a PDF or HTML document. The LaTeX-formatted text can be placed in the main file of the simulator source code, into a single C++ comment block, and between the `<sim2doc>` and `</sim2doc>` tags.

The final document can be generated using the OpenFLUID Sim2Doc buddy, included in the OpenFLUID command line program. See also apdx_optenv_cmdopt_buddies command line for available options.
Example of OpenFLUID command line to generate the PDF document using the Sim2Doc tool:

```
0
openfluid buddy sim2doc -o inputcpp=MySimFile.cpp,pdf=1
```

Part III

Appendix

Appendix A

Command line options and environment variables

A.1 Environment variables

The OpenFLUID framework takes into account the following environment variables (if they are set in the current running environment):

- `OPENFLUID_INSTALL_PREFIX`: overrides automatic detection of install path, useful on Windows systems.
- `OPENFLUID_USERDATA_PATH`: overrides the default user data home directory (set by default to `$HOME/.openfluid` on Unix systems)
- `OPENFLUID_TEMP_PATH`: overrides the default OpenFLUID temporary directory, used by OpenFLUID software components for temporary data.
- `OPENFLUID_SIMS_PATH`: extra search paths for OpenFLUID simulators. The path are separated by colon on UNIX systems, and by semicolon on Windows systems.
- `OPENFLUID_OBSS_PATH`: extra search paths for OpenFLUID observers.

The path are separated by colon on UNIX systems, and by semicolon on Windows systems.

A.2 Command line usage

Usage : `openfluid [<command>] [<options>] [<args>]`
Available commands:

- About OpenFLUID:
  - `info`: Display information about OpenFLUID
  - `version`: Display OpenFLUID version
  - `install-examples`: Install or reinstall examples in the user directory
  - `prepare-workspace`: Prepare an OpenFLUID workspace
- About wares:
  - `report`: Display report about available wares
  - `show-paths` Show search paths for wares
  - `create-ware`: Create ware sources
  - `check`: Checks ware sources for potential issues

  - `configure`: Configure ware sources for build
  - `build`: Build configured ware sources
  - `docalyze`: Build documentation of a ware
  - `purge`: purge build outputs on ware sources
  - `migrate-ware`: Migrate ware sources to current version
  - `migrate-ghostsim`: Migrate ghost simulator to current version
  - `info2build`: Generate build files from ware information

- About simulation:
  - `run`: Run a simulation
  - `create-data`: Create project or dataset

Available options:

- `--help,-h`: display the help message
- `--version`: display version

A.2.1 Running simulations

Run a simulation from a project or an input dataset.
Usage : `openfluid run [<options>] [<args>]`
Available options:

- `--help,-h` : display this help message
- `--auto-output-dir, -a` : create automatic output directory
- `--clean-output-dir, -c` : clean output directory before simulation
- `--max-threads=<arg>, -t <arg>` : set maximum number of threads for threaded spatial loops (default is 4)
- `--observers-paths=<arg>, -n <arg>` : add extra observers search paths (colon separated)
- `--profiling, -k` : enable simulation profiling
- `--quiet, -q` : quiet display during simulation
- `--simulators-paths=<arg>, -p <arg>` : add extra simulators search paths (colon separated)
- `--verbose, -v` : verbose display during simulation

Example of running a simulation from an input dataset:

```
0
openfluid run /path/to/dataset /path/to/results
```

Example of running a simulation from a project:

```
0
openfluid run /path/to/project
```

A.2.2 Wares reporting

Display informations about available wares
Usage : `openfluid report [<options>] [<args>]`
Available options:

- `--help,-h` : display this help message
- `--format=<arg>` : output format, argument can be text (default) or json
- `--list, -l` : display as simple list of wares IDs
- `--observers-paths=<arg>, -n <arg>` : add extra observers search paths (colon separated)
- `--simulators-paths=<arg>, -p <arg>` : add extra simulators search paths (colon separated)
- `--with-errors, -e` : report errors if any

Example of detailed reporting about available simulators:

```
0
openfluid report simulators
```

Example of reporting as a list about available observers:

```
0
openfluid report observers --list
```

A.2.3 Paths

Show search paths for wares
Usage : `openfluid show-paths [<options>] [<args>]`
Available options:

- `--help,-h` : display this help message
- `--observers-paths=<arg>, -n <arg>` : add extra observers search paths (colon separated)
- `--simulators-paths=<arg>, -p <arg>` : add extra simulators search paths (colon separated)

Appendix B

Datetime formats

OpenFLUID uses the ANSI strftime() standard formats for date time formatting to and from a format string. As an example, this format string can be used in CSV observer in parameters to customize date formats. The format string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a % character and a terminating conversion character that determines the conversion specification's behaviour. All ordinary characters are copied unchanged into the array.

For example, the nineteenth of April, two-thousand seven, at eleven hours, ten minutes and twenty-five seconds formatted using different format strings:

- `%d/%m/%Y %H:%M:%S` will give `19/04/2007 10:11:25`
- `%Y-%m-%d %H.%M` will give `2007-04-19 10.11`
- `%Y\t%m\t%d\t%H\t%M\t%S` will give `2007 04 19 10 11 25`

List of available conversion specifications:

- %a : locale's abbreviated weekday name.
- %A : locale's full weekday name.
- %b : locale's abbreviated month name.
- %B : locale's full month name.
- %c : locale's appropriate date and time representation.
- %C : century number (the year divided by 100 and truncated to an integer) as a decimal number [00-99].
- %d : day of the month as a decimal number [01,31].
- %D : same as %m/%d/%y.
- %e : day of the month as a decimal number [1,31]; a single digit is preceded by a space.
- %h : same as %b.
- %H : hour (24-hour clock) as a decimal number [00,23].
- %I : hour (12-hour clock) as a decimal number [01,12].
- %j : day of the year as a decimal number [001,366].
- %m : month as a decimal number [01,12].
- %M : minute as a decimal number [00,59].

- %n : is replaced by a newline character.
- %p : locale's equivalent of either a.m. or p.m.
- %r : time in a.m. and p.m. notation; in the POSIX locale this is equivalent to %I:%M:%S %p.
- %R : time in 24 hour notation (%H:%M).
- %S : second as a decimal number [00,61].
- %t : is replaced by a tab character.
- %T : time (%H:%M:%S).
- %u : weekday as a decimal number [1,7], with 1 representing Monday.
- %U : week number of the year (Sunday as the first day of the week) as a decimal number [00,53].
- %V : week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1.
- %w : weekday as a decimal number [0,6], with 0 representing Sunday.
- %W : week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.
- %x : locale's appropriate date representation.
- %X : locale's appropriate time representation.
- %y : year without century as a decimal number [00,99].
- %Y : year with century as a decimal number.
- %Z : timezone name or abbreviation, or by no bytes if no timezone information exists.
- %% : character %.

Appendix C

String representations of values

OpenFLUID values can be converted into strings, using the following representations

C.1 Simple values

Representation of simple values is trivial. In OpenFLUID, it is based on classical string representations.

C.1.1 BooleanValue

Values of BooleanValue type are converted into the `true` or `false` string.

C.1.2 IntegerValue

Values of IntegerValue type are converted into their textual representation. As an example, the value 192 will be converted to the `192` string.

C.1.3 DoubleValue

Values of DoubleValue type are converted into their textual representation. As an example, the value 17.37 will be converted to the `17.37` string.

C.1.4 StringValue

Since values of StringValue type are natively stored as string, they are not converted and represented as they are.

C.2 Compound values

Representation of compound values requires a more complex representation schema. In OpenFLUID, it is based on the JSON data format without any space or newline.

C.2.1 VectorValue

Values of VectorValue type are converted using the JSON notation for vectors which is a comma separated list of values enclosed by an opening square bracket and a closing square bracket.
As an example, the following vector

$$\begin{bmatrix} 1.5 & 19.6 & 0.005 & 1.0 & 258.99 \end{bmatrix}$$

will be converted into the `[1.5,19.6,0.005,1.0,258.99]` string.

C.2.2 MatrixValue

Values of MatrixValue type are converted using the JSON notation for matrix which are considered as a vector of vector(s).
As an example, the following matrix

$$\begin{bmatrix} 1.5 & 19.6 & 0.005 \\ 2.0 & 1.0 & 258.99 \end{bmatrix}$$

will be converted into the `[[1.5,19.6,0.005],[2.0,1.0,258.99]]` string.

C.2.3 MapValue

Values of MapValue type are converted using the JSON notation for objects which is a comma separated key-value list enclosed by an opening curly bracket and a closing curly bracket.
As an example, the following map

$$\left\{ \begin{array}{lcl} key1 & = & 0.005 \\ key2 & = & ''a\ word'' \\ key3 & = & \begin{bmatrix} 1.5 & 19.6 & 0.005 & 1.0 & 258.99 \end{bmatrix} \end{array} \right.$$

will be converted into the {`"key1":0.005,"key2":"a word","key3":[1.5,19.6,0.005,1.0,258.99]`} string.

Appendix D

Variabilize ware signature

Even if compiled simulators require an explicit definition of variable and unit class, the underlying code and signature can be templated. Before json signature, such change was possible through precompilation blocks around signature lines.
Since json signature we provide a new workflow to handle parametric information both in code and documentation.

D.1 In openfluid-wareinfo.json

Template variables (in id, unit definition...) should be surrounded by '@'
Example: `water.surf-@foo@`

D.2 In cmake

Affect a value to your variable in the main `CMakeLists.txt`
Example: `SET(foo "SU")`
Uncomment `CONFIGURED_SIGNATURE ON` in the `CMakeLists.txt` of `src`

D.3 In sources

(as before the parametric signature change)
Variable can be transmitted to cpp as precompilation variable by adding a definition in the CMakeLists.txt
Example: `ADD_DEFINITIONS(-DISFOO=1)`
Afterwards the precompilation variable `ISFOO` will be found in cpp code

D.4 In doc (optional)

Patterns in README and README.md in root dir of the ware will also be converted
Example:

```
0
# water.runoff.@foo@




Effect on @foo@ will...
```

Appendix E

Generator variable format and setup

This appendix provides insight about use cases and set up of generators to provide variable data easily. It deals only with single-column csv, for multi-column data files, see Multi-column generators : setup and use

E.1 What data types can be generated?

The available data types depend from the kind of wanted generation:

Generator type	Double (scalar)	Double (vector)	Double (matrix)	Integer	Boolean	String
Fixed values	Yes	Yes∗	Yes∗	Yes	Yes	Yes
Random values	Yes	Yes	Yes	Yes	Yes	No
Values from file interpolation	Yes	No	No	No	No	No
Values from file injection	Yes	No	No	No	No	No

E.1.1 Floating-point variables

∗The non-scalar double generator can be used either with a single value for all, or a value for each cell of the container. Eg, for every time step and every unit of the given unit class, for a generator of vector of size 4 containing doubles:

- ”3” will produce as variable value [3.0, 3.0, 3.0, 3.0]
- ”[1,4,2.2,3]” will produce as variable value [1.0, 4.0, 2.2, 3.0]

An option is available for the random generator, to be able to have identical or different values inside the container. Eg with a vector of 4 double:

- same value activated: t0: [2.1, 2.1, 2.1, 2.1], t1: [43, 43, 43, 43]
- same value disabled: t0: [4.5, 2.0, 22.1, 1.7], t1: [10.2, 2.3, 7,4, 9.9]

E.1.2 Boolean

Currently the random generator for boolean gives a balanced probability between true and false (50% each)

E.2 File formats for generators

Note

> Currently, these files formats are used by *interp* and *inject* generators only.

E.3 Sources file

The sources file format is an XML based format which defines a list of sources files associated to an unique ID.
The sources must be defined in a section delimited by the `<datasources>` tag, inside an `<openfluid>` tag and must be structured following these rules:

- Inside the `<datasources>` tag, there must be a set of `<filesource>` tags
- Each `<filesource>` tag must bring an `ID` attribute giving the identifier of source, and a `file` attribute giving the name of the file containing the source of data. The files must be placed in the input directory of the simulation.

```
0
<?xml version="1.0" standalone="yes"?>

<openfluid>



  <datasources>

    <filesource ID="1" file="source1.dat" />

    <filesource ID="2" file="source2.dat" />

  </datasources>



</openfluid>
```

Note

> As a sources file is not part of the standard input dataset, it must not have a `.fluidx` extension. Using the `.fluidx` extension may lead to an unexpected behaviour (such as deletion of the sources file).
> The recommended extension for this file is `.xml` (e.g. *sources.xml*)

An associated source data file is a two columns text file, containing a serie of values in time. The first column is the date using the ISO format `YYYY-MM-DD'T'HH:MM:SS`. The second column is the value itself.

```
0
1999-12-31T12:00:00 -1.0

1999-12-31T23:00:00 -5.0

2000-01-01T00:30:00 -15.0

2000-01-01T00:40:00 -5.0

2000-01-01T01:30:00 -15.0
```

E.4 Distribution file

A distribution file is a two column file associating a unit ID (first column) to a source ID (second column).

```
0
1 1

2 2

3 1

4 2

5 1
```

Appendix F

Multicolumn generators : setup and use

F.1 Sources file

The format of the source file is a CSV format which defines the values of the desired variables according to a simulation date. The first column defines the simulation dates using the ISO format `YYYYMMDDThhmmss`. The first row of the column is labelled #datetime. All others columns define a variable for a desired spatial unit. The first row of these columns are labelled using the format `UnitClass#UnitID:VarName`.

- `UnitClass` defines the unit class,
- `UnitId` defines the unit ID,
- `VarName` defines the variable name.

If you want to define the same variable for all the units in a given class, you can use the format `UnitClass#*:VarName` as a column label, where $*$ replace all unit classes.

```
0
#datetime;           UnitClass1#1:var1;      UnitClass2#*:var2;      UnitClass2#5:var2;

19910812T090000;     25;                     5;                      10;

19910812T100000;     30;                     NA;                     15;

19910812T110000;     35;                     NA;                     25;

19910812T120000;     40;                     NA;                     40;

19910812T130000;     45;                     NA;                     60;

19910812T140000;     NA;                     20;                     85;
```

Note

Currently, the column separator is the character `;`.

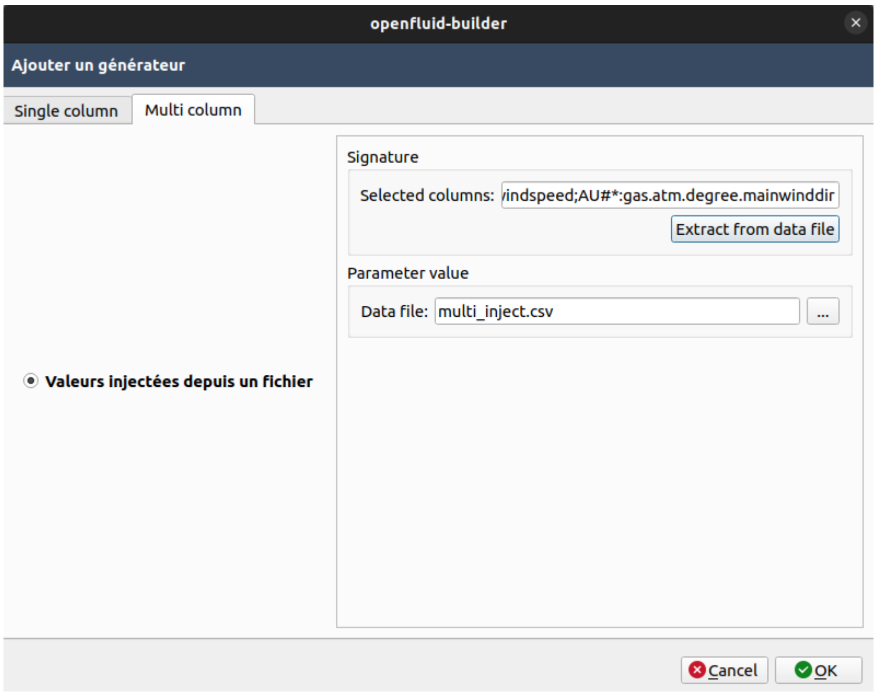
If a variable does not have a value at a specific date, we indicate it using the value *NA*.

This example shows that we inject values for the dates in the first column. The second column indicates that we are injecting values for the *var1* variable for spatial unit *1* of the *UnitClass1* class. The third column indicates that we inject values for the *var2* variable for all the spatial units in the *UnitClass2* class. The last column override the third one for the spatial unit 5 because it is more specific.

F.2 Usage

F.2.1 Builder

In OpenFLUID-Builder, the multi-column CSV can be set with the same dialog than for other generators. It will use a chosen multi-column csv file. You can either use all the variables defined in the file or specify just a selection of them. The generator will produce all selected variables from the csv file.



openfluid-builder

Ajouter un générateur

Single column | Multi column

Signature

Selected columns: /indspeed;AU#*:gas.atm.degree.mainwinddir

Extract from data file

Parameter value

Data file: multi_inject.csv

(●) Valeurs injectées depuis un fichier

Figure F.1: Multi-column generator setup tab

In this example (taken from *Firespread example*), we are importing the file *multi_inject.csv* which defines 2 variables for all spatial units in the unit class *AU* : `AU#*gas.atm.V.windspeed` and `AU#*:gas.atm.degree.mainwinddir`. The *Signature* parameter defines all the selected columns from the file.

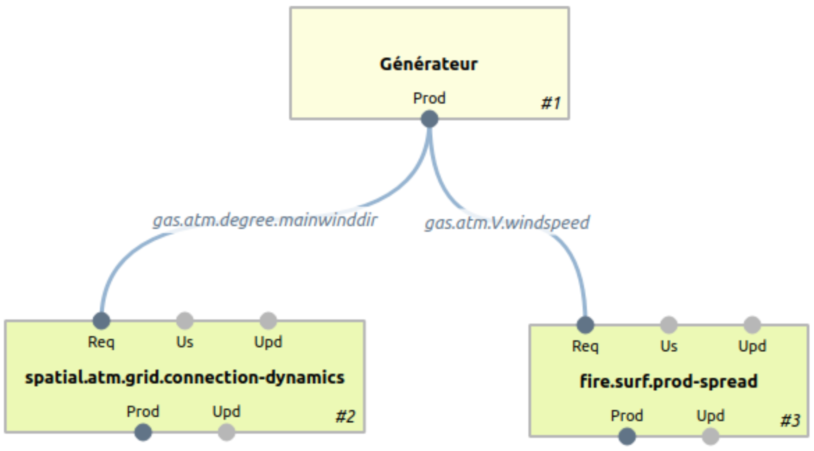


Figure F.2: Generator variables

In this image, we can see that the generators produces the selected variables *gas.atm.V.windspeed* and *gas.atm.degree.mainwinddir*.

F.2.2 FluidX format

Here is an example of model.fluidx file containing a multi-column generator:

```
0
<?xml version="1.0" encoding="UTF-8"?>

<openfluid format="fluidx 4">

    <model>

        <generator variables="AU#*:gas.atm.V.windspeed;AU#*:gas.atm.degree.mainwinddir" method="inject-multicol" enablec

            <param name="datafile" value="multi_inject.csv"/>

        </generator>

        ...

    </model>

</openfluid>
```

The generator `method` is `inject-multicol` and there is also has a mandatory attribute `variables` where the selected columns headers are specified. The multi-column generator block contains a single parameter called `datafile` indicating the path of csv file used for injection.

Appendix G

Structure of an OpenFLUID project

An OpenFLUID project can be run using any of the OpenFLUID programs such as `openfluid` command line, OpenFLUID Builder or ROpenFLUID.

As an example example, to run a simulation based on the the project located in `/absolute/path/to/workdir/a_dummy_p` using the `openfluid` command line program, the command to use is:

```
0
openfluid run /absolute/path/to/workdir/a_dummy_project
```

An OpenFLUID project is made of a directory which includes:

- an `openfluid-project.json` file containing informations about the project,
- an `IN` subdirectory containing the input dataset,
- an `OUT` subdirectory as the default output directory, containing the simulation results if any.

The `openfluid-project.json` contains the name of the project, the description, the authors, the creation date, the date of the latest modification, and a flag for incremental output directory (this feature is currently disabled).

```
0
{

  "name": "a dummy project",

  "description": "",

  "authors": "John Doe",

  "created_at": "2013-09-16 17:00:00",

  "updated_at": "2013-09-16 17:00:00",

  "inc_outdir": false,

  "context": {}

}
```

The `openfluid-project.json` may also contain contextual informations added by OpenFLUID software applications such as OpenFLUID-Builder. These informations can be colors of shapes in map view, placement of models in model view, ...

Appendix H

Organization of an OpenFLUID workspace

An OpenFLUID workspace is a directory organized to contain OpenFLUID projects for simulations and source code of simulators, observers and builder-extensions.

The default workspace is located in `${HOME}/.openfluid/workspace` for Linux and MacOS systems or in `%HOMEPATH%\openfluid\workspace` for Windows systems. Any user can create as many workspaces as needed using the *Environment* section of the *Preferences* dialog.

A workspace directory is organized as following:

- a `project` directory containing OpenFLUID projects
- a `wares-dev` directory containing source code of wares in which there are the `simulators`, `observers` and `builder-extensions` directories for each ware category.
- an `openfluid-waresdev.json` file containing the workspace configuration for wares development

Appendix I

Working with submodules

Cette page explique comment manipuler les simulateurs contenant des fragments en ligne de commande. La partie d'ajout/suppression de fragment peut être faite depuis DevStudio.

Il s'agit ici d'une adaptation des principes de submodule dans le contexte de simulateurs. Aucune vraie spécificité, il s'agit d'opérations faisables dès que l'on possède Git et pour des contextes bien plus variés que la gestion de fragments de code dans un simulateur.
`Page Git de référence pour les commandes Submodule`

I.1 Créer un simulateur avec fragment de code

```
0
git submodule add https://github.com/.../MyFragment.git fragments/MyFragment
```

I.2 Utilisation d'un simulateur contenant un fragment

I.2.1 Clonage du repo Git du simulateur incluant le fragment

```
0
git clone --recurse-submodules MY_SIMULATOR_REPOSITORY.git
```

Cette commande clonera le simulateur ainsi que tout les submodules (tous les fragments) qu'il contient.

I.3 Modification d'un fragment de code

I.3.1 Patch sur le repo Git du fragment

2 façons de faire :

- Sûre, pas de risque de confusion ou d'effet de bord : **Modifier le fragment à part**
    1. cloner le repo du fragment dans un répertoire à part
       `git clone https://abc.org/Codefrag.git`
    2. Effectuer les modifications et commit
       `.../Codefrag$ git add ...` puis `git commit`
    3. Push les changements sur le repo du fragment
       `.../Codefrag$ git push origin main`

- Directe, périlleuse : **Modifier le code directement dans le repo du simulateur**
  Même manipulations que celles pour la façon précédente, mais en allant dans le répertoire contenant le code du fragment récupéré lors de la récupération du code du simulateur au lieu de le cloner dans un endroit indépendant.
  Le principe est que les modifications au niveau du fragment seront ”contenues” par le repo du fragment et que le repo du simulateur pourra juste dire que le repo du fragment a changé lors d'un git status. Il faut donc faire les manipulations (add/commit/push) en se positionnant bien au niveau du dossier du fragment pour que les opérations fonctionnent correctement

I.3.2 Propagation sur les simulateurs concernés

1. Récupérer les changements de fragments dans le repo des simulateurs impactés en utilisant la commande `submodule update`:
   ```
   .../Simulator$ git submodule update --recursive --remote
   ```
2. Puis `.../Simulator$ git add` du dossier contenant le submodule, ce qui donne pour l'exemple utilisé :
   ```
   .../Simulator$ git add fragments/MyFragment
   ```
   Il devrait être visible en tant que changement dans le `.../Simulator$ git status` avant l'opération.
3. Commit ces changements du repo du *simulateur* (`.../Simulator$ git commit`) pour que le dépôt distant récupère ces informations

Si l'on a choisi de modifier le code du fragment dans le dossier du simulateur et qu'on a plusieurs simulateurs concernés, il faut se souvenir dans quel simulateur on a modifié le code et penser à réaligner les autres.

Appendix J

Controlled stochasticity in ware

J.1 Setup stochasticity

OpenFLUID provides tools to use stochasticity inside your wares. To do so, a parameter with the key *seed* must be specified inside a ware tags in the *model.fluidx* file:

```
0
<simulator ID="SimulatorTest">

    <param name="seed" value="1" />

</simulator>
```

A seed is a value used to initialise a random number generator. It is used to produce a pseudo-random sequence of numbers, which will always be identical if the same seed is used again. It allows the reproduction of random results in simulations, for example. If a negative value is specified for this parameter, it will use a random seed value. If it is a positive value, it will use this value as the seed. In both cases, you can track the seed value in the simulation output logs.
Once the parameter *seed* has been added, go to your ware source code and add the include line: `#include <openfluid/ware/WareRNG.hpp>` Create a class variable of type `openfluid::ware::WareRNG` `RNG`. This variable will handle the randomness and provide few utility functions. To construct this variable, call its constructor by passing as argument the ware object

```
0
SimulatorTest() : PluggableSimulator(), Rng(this)

{

    // constructor

}
```

In order to use the seed value provided in the *model.fluidx*, we need to initialize the variable in the `initParams(const openfluid::ware::WareParams_t& Params)` function by adding the line : `Rng.init(Params);`

```
0
void initParams(const openfluid::ware::WareParams_t& Params)

{

    Rng.init(Params);

}
```

Note

If not initialized, the object will use a random seed.

Utility functions can now be used in order to generate random numbers.

J.2 Example

The example below shows the use of stochasticity in ware.

```
0
// HPP


#include <openfluid/ware/WareRNG.hpp>


class SimulatorTest : public openfluid::ware::PluggableSimulator

{

    public:

        SimulatorTest();


    private:

        openfluid::ware::WareRNG Rng;

};




// CPP


SimulatorTest() : PluggableSimulator(), Rng(this)

{

    // constructor

}




// ======================================================================

// ======================================================================




void initParams(const openfluid::ware::WareParams_t& Params)

{

    Rng.init(Params);

}


openfluid::base::SchedulingRequest runStep()

{

    // example of uniform distribution with integer numbers
    std::vector<int> Vector = Rng.runif(5, 0, 10);

}
```

To see all available utility functions, check **openfluid::tools::RandomNumberGenerator** class.

Laboratoire d'étude des Interactions Sol - Agrosystème - Hydrosystème

SCIENCE & IMPACT