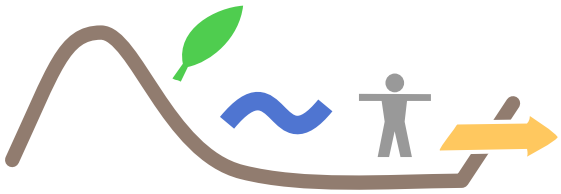

OpenFLUID

Software Environment
for Modelling Fluxes in Landscapes

OpenFLUID in a nutshell

Manual for OpenFLUID v2.1.0

LISAH
Laboratoire d'étude des Interactions
Sol - Agrosystème - Hydrosystème

Jean-Christophe Fabre, 2015

Contents

Foreword

OpenFLUID is a software environment for modelling spatial functionning of 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 simulators for OpenFLUID, either using existing source code or creating source code *de novo*
- an appendix giving useful reference informations

More informations are available on the official OpenFLUID web site, notably detailed informations on scientific concepts underlying the OpenFLUID software and practical informations about how to develop simulators including API documentation.

Official OpenFLUID web site: http://www.openfluid-project.org

Part I

Running simulations with OpenFLUID

Chapter 1

Usage of OpenFLUID applications

OpenFLUID simulations can be run either using the command line interface (`openfluid` program), the graphical user interface (`openfluid-builder` program), or using the R environment through the R-OpenFLUID package.

All these programs and packages use the same input dataset format (See Formats of input datasets), and propose all concepts and features of the OpenFLUID software environment, as they share the same OpenFLUID software framework. A simulation input dataset can be executed using any of the following OpenFLUID software programs (except using the DevStudio development environment).

1.1 Graphical Interface for simulations : OpenFLUID-Builder

The OpenFLUID-Builder user interface proposes a graphical environment to prepare, parameterize, execute and exploit simulations with OpenFLUID. It is a good starting point for beginners who discover the OpenFLUID concepts and environment. It can be run either from the program menu of your system or from a console using the `openfluid-builder` command.

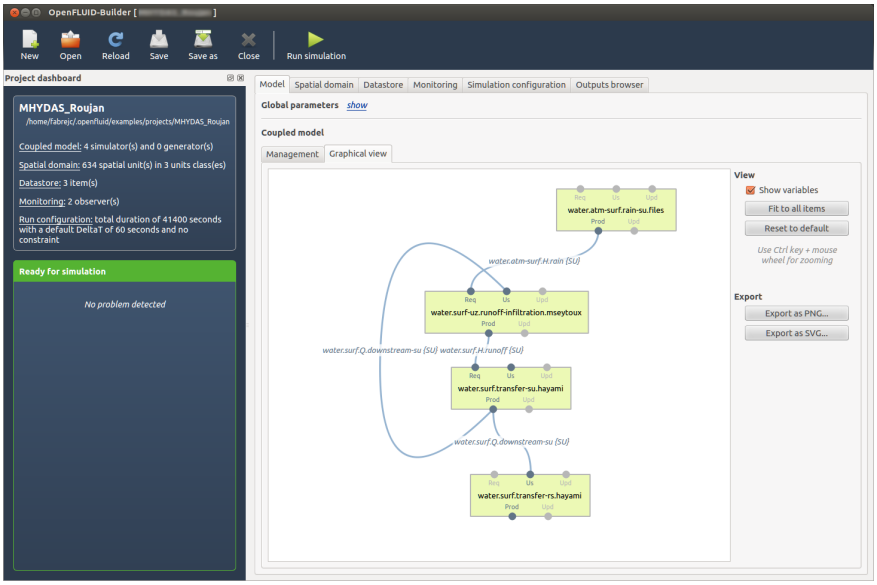

Figure 1.1: Screenshot of the model view in OpenFLUID-Builder

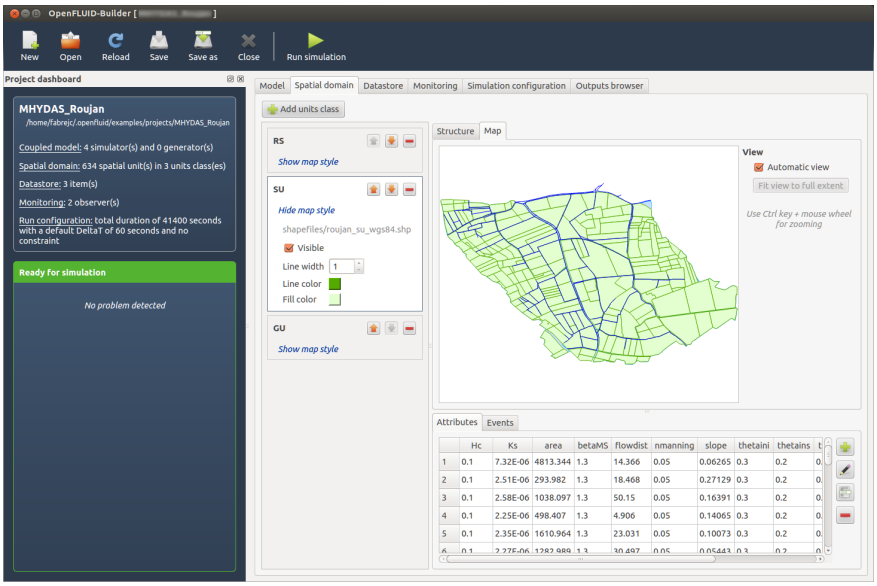

Figure 1.2: Screenshot of the spatial domain map view in OpenFLUID-Builder

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 in the project itself. The project must be a valid OpenFLUID project, usually created using the OpenFLUID-Builder user interface. It can also be created by hand.

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

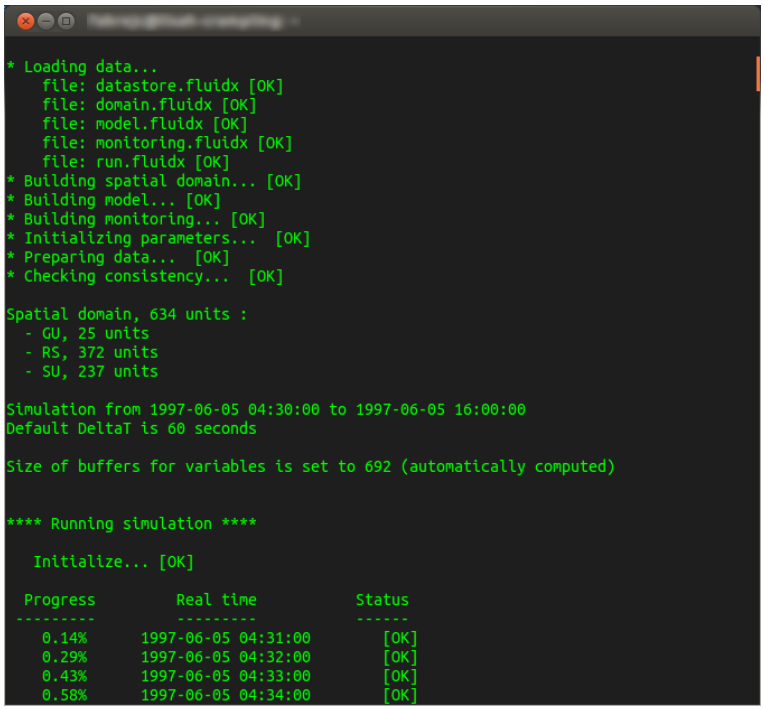

Figure 1.3: OpenFLUID simulation using command line

1.3 Within the GNU R environment : ROpenFLUID

OpenFLUID can be used from within the GNU R environment with the ROpenFLUID package. This package allows to load an input dataset, parameterize and run a simulation, exploit 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:

```
library('ROpenFLUID')
ofsim = OpenFLUID.loadDataset('/path/to/dataset')
OpenFLUID.runSimulation(ofsim)
data = OpenFLUID.loadResult(ofsim,'TestUnits',15,'var.name')
```

More details are available in the specific ROpenFLUID documentation, available on the OpenFLUID web site.

1.4 Development environment : OpenFLUID-DevStudio

The OpenFLUID-Devstudio is the environment for development of simulators, observers and builder-extensions for OpenFLUID. It proposes a complete environment for assisted source code creation and development. It can be run either from the program menu of your system or from a console using the `openfluid-devstudio` command.

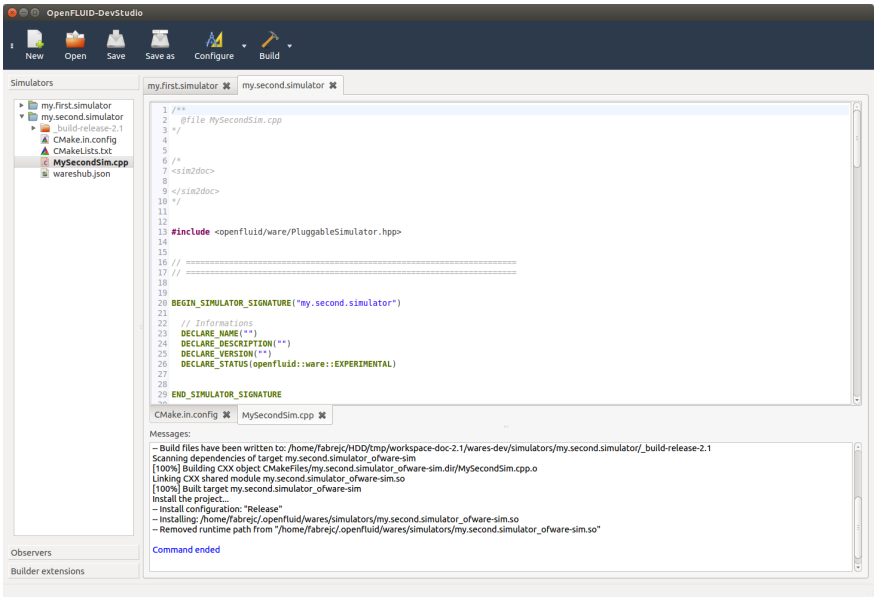

Figure 1.4: Screenshot of OpenFLUID-DevStudio workspace

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)
- Help shortcut to online documentation
- Other usual features of source code editor

Chapter 2

Format of input datasets

This part of the manual describes the FluidX file(s) format used to define a simulation dataset. Refer to the Overview of OpenFLUID applications part of this manual to run the simulations.

An OpenFLUID input dataset includes different informations, defined in one or many files:

- the spatial domain definition
- the flux model definition
- the spatial domain attributes
- the monitoring configuration
- the discrete events
- the run configuration

All files must be placed into a directory that can be reached by the OpenFLUID program used.

As OpenFLUID-Builder uses the FluidX format natively, the entire input dataset can be created through the OpenFLUID-Builder software.

Out of OpenFLUID-Builder, these FluidX files can be created by hand or using external tools. In this case, it is encouraged to write custom scripts in dedicated software, such as Geographic Information Systems (GIS) or scientific environments such as R.

2.1 Overview

The FluidX file format is an XML based format for OpenFLUID input datasets. The OpenFLUID input information can be provided by a one or many files using this FluidX formatn with 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 *run configuration* section defined by the `<run>` tag
- The *monitoring* section defined by the `<monitoring>` 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.

Summary view of the XML structure of FluidX files:

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

  <model>
    <!-- here is the model definition -->
  </model>

  <domain>
    <!-- here is the spatial domain definition, associated data and events -->
  </domain>

  <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 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 of <simulator>, <generator> and <gparams> tags
- 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.
- 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. These parameters can be scalar or vector of integer values, floating point values, string values. In case of vector, the values of the vector are separated by a ; (semicolon).
- 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.

- 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 param named fixedvalue for the value to produce.
- A generator using the random method must provide a param named min and a param named max delimiting the random range for the value to produce.
- A generator using the inject or interp method must provide a param named sources giving the data sources filename and a param named distribution giving the distribution filename for the value to produce (see also File formats for "interp" or "inject" generators).
- 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.

```
<?xml version="1.0" standalone="yes"?>
<openfluid>
  <model>

    <gparams>
      <param name="gparam1" value="100" />
      <param name="gparam2" value="0.1" />
    </gparams>

    <simulator ID="example.simulatorA" />

    <generator varname="example.generator.fixed" unitsclass="EU1" method="fixed" varsize="11">
      <param name="fixedvalue" value="20" />
    </generator>

    <generator varname="example.generator.random" unitsclass="EU2" method="random">
      <param name="min" value="20.53" />
      <param name="max" value="50" />
    </generator>

    <simulator ID="example.simulatorB">
      <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 the permanent call order in synchronized coupled model (not applicable for variable time coupled models)

2.2.2 Spatial domain section

2.2.2.1 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 information 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 outgoing 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 parent 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

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

2.2.2.2 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 real, integer, string, vector or matrix.

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

2.2.2.3 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.

```
<?xml version="1.0" standalone="yes"?>
<openfluid>
  <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.

```
<?xml version="1.0" standalone="yes"?>
<openfluid>
  <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.
- 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. These parameters can be scalar or vector of integer values, floating point values, string values.

Note

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

```
<?xml version="1.0" standalone="yes"?>
<openfluid>
  <monitoring>

    <observer ID="export.vars.files.csv">
      <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" >
      <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.

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

```
  <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

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

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

  <model>
    <gparams>
      <param name="gparam1" value="100" />
      <param name="gparam2" value="0.1" />
    </gparams>
    <simulator fileID="example.simulatorA" />
    <generator varname="example.generator.fixed" unitsclass="EU1"
               method="fixed" varsize="11">
      <param name="fixedvalue" value="20" />
    </generator>
    <generator varname="example.generator.random" unitsclass="EU2"
               method="random">
      <param name="min" value="20.53" />
      <param name="max" value="50" />
    </generator>
    <simulator fileID="example.simulatorB">
      <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">
    <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" >
      <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 speaking, an OpenFLUID simulator is made of two main parts: The signature and a C++ class containing the computational code. These two parts have to be developped in a C++ file (.cpp). They must be compiled before using it in the OpenFLUID environment.

3.1 Simulator signature

The signature of a simulator contains meta-informations about the simulator. These informations will be mainly used for automatic coupling and consistency checking of simulators. To get more informations about the simulators signatures, 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 an empty 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 your private members.

The destructor of the simulator is called when the simulator is released after simulation, at the end of the execution of the OpenFLUID application. You may put here instruction to free the memory you allocated for the needs of the computational code (other objects, pointed vars, ...).

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 should be 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 should be used to do data pre-processing before the consistency checking.

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

The **initializeRun** method must be 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 computational code.

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

Chapter 4

Creation of an empty simulator

As mentioned in the previous section (see Overview of an OpenFLUID simulator), a simulator must contain two parts :

- A signature giving information about the simulator, and used by the OpenFLUID framework to identify and couple simulators
- A C++ class defining essential methods for computational code

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 4.8 or later for C++11 compatibility)
- CMake as the build configuration tool (version 2.8.12 or later).

  OpenFLUID provides a CMake module to ease the build of simulators.

These tools are also required when using the OpenFLUID-DevStudio application. Detailed instructions for installation of these tools are available on the OpenFLUID Community web site (http://www.-openfluid-project.org/community).

The OpenFLUID-DevStudio is the recommended environment for simulators development, and is the only one which is officially supported for OpenFLUID wares development.

4.2 Writing the signature

The simulator signature is a set of informations about the content and behaviour of the simulator source code. With these informations, the OpenFLUID framework can evaluate the simulator, have information on what it does, on what it expects and produces, and can load it dynamically.

The informations included in the signature are :

- Identification
- Development information

- Expected parameters
- Expected variables and spatial attributes
- Produced variables and spatial attributes
- Spatial graph updates
- Time step behaviour

Usually, the signature is declared and implemented at the beginning of the .cpp file. It starts with the **BEGIN_SIMULATOR_SIGNATURE** macro and ends with the **END_SIMULATOR_SIGNATURE** macro. The minimal signature must include identification information. See part Declaration of the simulator signature for details on how to write the signature.

4.3 Writing the C++ class for the simulator

The C++ class integrates the computational code of the simulator, corresponding to successive stages of simulations. You will find a Complete example below, giving an overview of the source code of an empty simulator. The Development of the simulator source code part gives details about how to develop the computational code in the simulator.

4.4 Building the simulator

Any OpenFLUID simulator must be compiled using the GCC C++ compiler (g++) and must be linked to the OpenFLUID libraries and dependencies.

The recommended way to build your simulator is to use the CMake build system with the OpenFLUID CMake module, and provide CMake configuration files (CMakeLists.txt, CMake.in.config).

These operations can be performed automatically using the OpenFLUID-DevStudio application.

These operations can also be performed manually, with the following steps:

1. Create a build directory in your source directory (e.g. `_build`)
2. Go to this build directory
3. Run the `cmake ..` command, with the optional `-DCMAKE_BUILD_TYPE=Debug` directive for debugging mode
4. Run the build command (e.g. `make`)

These steps are for Linux systems, and must be slightly adapted for other systems.

4.5 Complete example

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

4.5.1 File ExampleSimulator.cpp containing the simulator source code

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


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


BEGIN_SIMULATOR_SIGNATURE("example.simulator")

  DECLARE_NAME("Example simulator");
  DECLARE_DESCRIPTION("This simulator is an example");
  DECLARE_VERSION("14.07");
  DECLARE_STATUS(openfluid::ware::EXPERIMENTAL);

  DECLARE_AUTHOR("John","john@foobar.org");
  DECLARE_AUTHOR("Dave","dave@foobar.org");
  DECLARE_AUTHOR("Mike","mike@foobar.org");

END_SIMULATOR_SIGNATURE


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


class ExampleSimulator : public openfluid::ware::PluggableSimulator
{
  private:

  public:


    Example(): PluggableSimulator()
    {
      // Here is source code for constructor
    }


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


    ~Example()
    {
      // 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 step run

      return DefaultDeltaT();
    }


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


    void finalizeRun()
    {
      // Here is source code for finalization
    }

};


DEFINE_SIMULATOR_CLASS(ExampleSimulator);
```

4.5.2 File CMake.in.config containing the build configuration

```
# 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 to ON or AUTO for build of simulator documentation using sim2doc
SET(SIM_SIM2DOC_MODE ON)

#set to ON to disable installation of sim2doc built documentation
SET(SIM_SIM2DOC_INSTALL_DISABLED OFF)

# 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 )
```

4.5.3 File CMakeLists.txt defining the build process

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

INCLUDE(CMake.in.config)

FIND_PACKAGE(OpenFLUIDHelpers REQUIRED)

OPENFLUID_ADD_SIMULATOR(SIM)
```

Chapter 5

Declaration of the simulator signature

The signature has to be defined between the **BEGIN_SIMULATOR_SIGNATURE** and the **END_SIMULATOR_SIGNATURE** macros.

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 as an argument of the **BEGIN_SIMULATOR_SIGNATURE** macro.

Other optional informations can be included for better description of the simulator:

- the simulator name, declared through the **DECLARE_NAME** macro, allowing to assign a long name to the simulator
- the simulator description, declared through the **DECLARE_DESCRIPTION** macro, allowing to provide a detailed description of what the simulator actually does
- the name(s) of the author(s) and her/his email address, declared through the **DECLARE_AUTHOR** macro. There may be multiple **DECLARE_AUTHOR** macros in the signature in case of multiple authors
- the software version of the simulator, declared through the **DECLARE_VERSION** macro
- the software status of the simulator, declared through the **DECLARE_STATUS** macro. The value can be openfluid::ware::EXPERIMENTAL, openfluid::ware::BETA or openfluid::ware::STABLE

See the Complete signature example part for detailed example.

5.2 Informations about scientific application

The informations about scientific applications is only indicative. It has no effects on simulator consistency or computational code. These informations can be :

- the domain in which the simulator can be applied, declared through the **DECLARE_DOMAIN** macro
- the processes simulated by the simulator, declared through the **DECLARE_PROCESS** macro
- the numerical methods used by the simulator, declared through the **DECLARE_METHOD** macro

5.3 Data and spatial graph

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

5.3.1 Simulator parameters

Simulator parameters are values provided to each simulator, and are declared using the **DECLARE_REQUIRED_PARAMETER** or **DECLARE_USED_PARAMETER** macros. These macros 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 simulator parameter declaration:

```
DECLARE_REQUIRED_PARAMETER("meanspeed","mean speed to use","m/s")
```

5.3.2 Spatial attributes

Spatial attributes are constant properties attached to each spatial units, and are declared using **DECLARE_REQUIRED_ATTRIBUTE** or **DECLARE_USED_ATTRIBUTE** macros

These macros 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)

Example of attributes declaration:

```
DECLARE_REQUIRED_ATTRIBUTE("area","TU","area of the Test Units","m")
DECLARE_USED_ATTRIBUTE("landuse","OU","landuse of the Other Units","")
```

5.3.3 Simulation variables

Simulation variables are attached to spatial units. They are produced, accessed and modified by simulators during simulations.

Accessed variables are declared using **DECLARE_REQUIRED_VARIABLE** or **DECLARE_USED_VARIABLE** macros, produced variables are declared using **DECLARE_PRODUCED_VARIABLE** macro, updated variables are declared using **DECLARE_UPDATED_VARIABLE** macro.

These macros 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:

```
DECLARE_REQUIRED_VARIABLE("varA[double]","TU","","m")
DECLARE_USED_VARIABLE("varB","OU","simple var on Other Units","kg")
DECLARE_PRODUCED_VARIABLE("VarB[vector]","TU","vectorized var on Test Units","kg")
DECLARE_UPDATED_VARIABLE("VarC","TU","","")
```

5.3.4 Discrete events

Discrete events are attached to spatial units, They are accessed or appended by simulators during simulations, and are declared using the **DECLARE_USED_EVENTS** macro.

The declaration macro takes 1 argument: the units class.

Example of events declaration:

```
DECLARE_USED_EVENTS("TU")
```

5.3.5 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 using the **DECLARE_USED_EXTRAFILE** or **DECLARE_REQUIRED_EXTRAFILE** macros.

The declaration macro takes 1 argument: the file name with relative path to the dataset path.

Example of extra file declarations:

```
DECLARE_USED_EXTRAFILE("fileA.dat")
DECLARE_REQUIRED_EXTRAFILE("geo/zone.shp")
```

5.3.6 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 macros.

The **DECLARE_UPDATED_UNITSGRAPH** macro is used for declaration of the global units graph modification that will occur during simulation. It is for information purpose only, and takes a description as a single argument.

The **DECLARE_UPDATED_UNITSCLASS** macro is used for declaration of units classes that will be affected, and how. It takes two arguments:

- the units class
- the description of the update (may be empty)

Example of declarations for spatial units graph:

```
DECLARE_UPDATED_UNITSGRAPH("update of the spatial graph for ...")
DECLARE_UPDATED_UNITSCLASS("TU","")
```

5.4 Complete signature example

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

```
BEGIN_SIMULATOR_SIGNATURE("example.simulator")

  DECLARE_NAME("Example simulator");
  DECLARE_DESCRIPTION("This simulator is an example");
  DECLARE_VERSION("13.05");
  DECLARE_STATUS(openfluid::ware::EXPERIMENTAL);
  DECLARE_AUTHOR("John","john@foobar.org");
  DECLARE_AUTHOR("Dave","dave@foobar.org");
  DECLARE_AUTHOR("Mike","mike@foobar.org");

  DECLARE_USED_PARAMETER("meanspeed","mean speed to use","m/s")

  DECLARE_REQUIRED_ATTRIBUTE("area","TU","area of the Test Units","m")
  DECLARE_USED_ATTRIBUTE("landuse","OU","landuse of the Other Units","")

  DECLARE_REQUIRED_VARIABLE("varA[double]","TU","","m")
  DECLARE_USED_VARIABLE("varB","OU","simple var on Other Units","kg")
  DECLARE_PRODUCED_VARIABLE("VarB[vector]","TU","vectorized var on Test Units","kg")
  DECLARE_UPDATED_VARIABLE("VarC","TU","","")

  DECLARE_USED_EVENTS("TU")

END_SIMULATOR_SIGNATURE
```

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 mentioned in the previous section, 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.

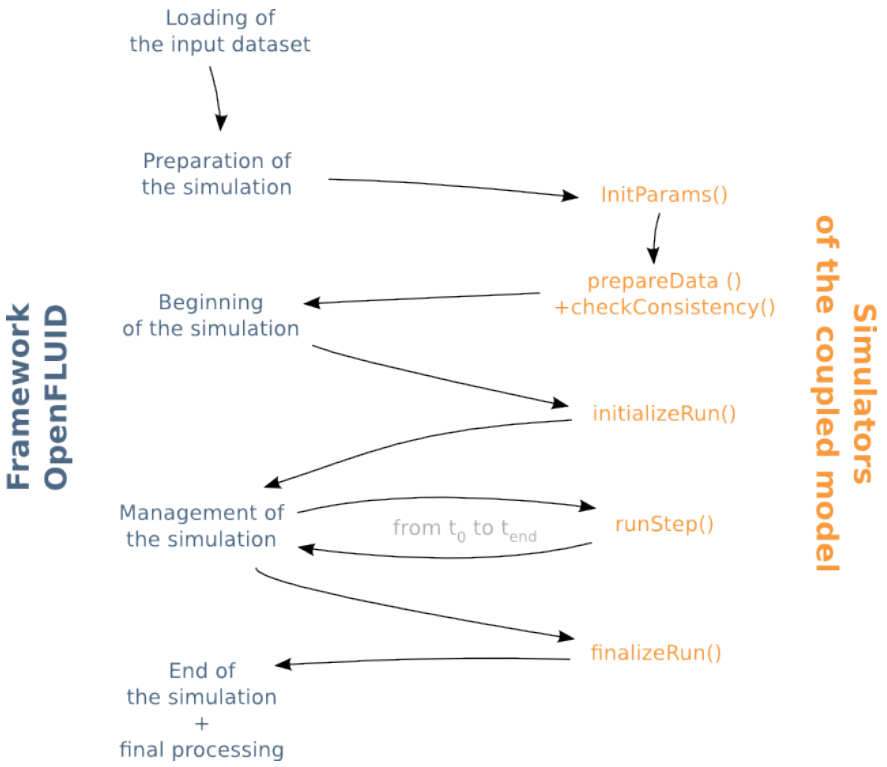

Figure 6.1: Interactions sequence between the OpenFLUID framework and the simulators

Among these methods, the **initializeRun()** and **runStep()** methods have a special behaviour: these two methods must return the simulation time when the simulator will be called again.

This simulation time can be

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

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

```
openfluid::base::SchedulingRequest initializeRun()
{
  // do something here

  return DefaultDeltaT();
}

openfluid::base::SchedulingRequest runStep()
{
  // do something here

  return DefaultDeltaT();
}
```

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

```
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 calling, 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 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 (see String representation of values)

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 macros.

6.2.1.1 Sequential parsing

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

- **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 macro:

- **OPENFLUID_UNITSLIST_LOOP**

The source code below shows spatial graph parsing examples. 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.

```
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_ALLUNITS_ORDERED_LOOP(UU)
  {
    // do something here
  }

  return DefaultDeltaT();
}
```

6.2.1.2 Parallel processing using multithreading

A process 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 macro must be a pointer to an **openfluid::core::SpatialUnit** as it represents the currently processed spatial unit.

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
}


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


openfluid::base::SchedulingRequest runStep()
{

  APPLY_UNITS_ORDERED_LOOP_THREADED("SU",MySimulator::computeA);
  APPLY_UNITS_ORDERED_LOOP_THREADED("TU",MySimulator::computeB,2.5);

  APPLY_ALLUNITS_ORDERED_LOOP_THREADED(MySimulator::computeA);

  return DefaultDeltaT();
}
```

Please note:

- If a spatial loop is used inside other 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 speed 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 can be 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 Spatial units graph.

6.2.3.1 Creating and deleting spatial units

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

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

6.2.3.2 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:

```
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);
}
```

6.2.3.3 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:

```
openfluid::base::SchedulingRequest runStep()
{
  long int Duration = OPENFLUID_GetSimulationDuration();

  long int CurrentIndex = OPENFLUID_GetCurrentTimeIndex();
  openfluid::core::DateTime CurrentDT = OPENFLUID_GetCurrentDate();

  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_GetSimulatorParameter**.

The requested parameter name must be the same as the one used in the model.fluidx file (see Model section), or be filled from the OpenFLUID-Builder graphical interface.

Example of initParams method:

```
void initParams(const openfluid::ware::WareParams_t& Params)
{
  m_MyParam = 0; //default value
  OPENFLUID_GetSimulatorParameter(Params,"myparam",m_MyParam);
}
```

To be reused in other part of the simulator, the 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_GetSimulatorParameter method to get more informations about other available types, available on OpenFLUID web 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:

```
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
  }
}
```

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:

```
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 macro:

- **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:

```
openfluid::base::SchedulingRequest runStep()
{
  openfluid::core::SpatialUnit* TU;
  openfluid::core::EventCollection EvColl;
  openfluid::core::Event* Ev;
  std::list<openfluid::core::Event* > *EvList;
  openfluid::core::DateTime BTime, ETime;

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

  OPENFLUID_UNITS_ORDERED_LOOP("TU",TU)
  {
    OPENFLUID_GetEvents(TU,BTime,ETime,EvColl);
    EvList = EvColl.getEventsList();

    OPENFLUID_EVENT_COLLECTION_LOOP(EvColl.getEventsList(),Ev)
    {
      if (Ev->isInfoEquals("molecule","glyphosate"))
      {
```

```
        // process the event
      }
    }

  }

  return DefaultDeltaT();
}
```

6.8 Internal state data

In order to keep the status of the simulation function 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 MySimulator : 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()@tableofcontents
{
  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:

```
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
- mode.saveresults [boolean] : result saving in output directory is enabled/disabled
- mode.writereport [boolean] : simulation report 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:

```
openfluid::base::SchedulingRequest runStep()
{
  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-logexplorer` program or using the OpenFLUID-Builder application.

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:

```
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 distributed property on SU");
      return false;
    }
  }
}
```

6.11 Debugging

Debugging macros 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 macros are available for debugging:

- **OFDBG_LOCATE** for file/line location where the macro is called
- **OFDBG_MESSAGE** adding the message given as an argument
- **OFDBG_UNIT** for information about the unit given as an argument
- **OFDBG_UNIT_EXTENDED** for extended information about the unit given as an argument
- **OFDBG_EVENT** for information about the event given as an argument
- **OFDBG_EVENTCOLLECTION** for information about the event collection given as an argument

Example:

```
openfluid::base::SchedulingRequest runStep()
{
  openfluid::core::SpatialUnit* TU;
  openfluid::core::DateTime BeginDate,EndDate;
  openfluid::core::EventsCollection EvColl;

  OFDBG_LOCATE;

  BeginDate = OPENFLUID_GetCurrentDate();
  EndDate = OPENFLUID_GetCurrentDate() + OPENFLUID_GetDefaultDeltaT() - 1;
```

```
  OPENFLUID_UNITS_ORDERED_LOOP("TU",TU)
  {
     OFDBG_UNIT_EXTENDED(TU);

     EvColl.clear();
     OPENFLUID_GetEvents(TU,BeginDate,EndDate,EvColl);
     OFDBG_EVENTCOLLECTION(EvColl);
  }

  return DefaultDeltaT();
}
```

6.12 Fortran 77/90 source code integration

The C++ - Fortran interface is defined in the openfluid/tools/FortranCPP.hpp file. In order to execute Fortran code from a simulator, this Fortran source code have to be wrapped into subroutines that will be called from the C++ code of the simulation function.

To help developers of simulators to achieve this wrapping operation, the FortranCPP.hpp file defines macros. These macros allows calls of Fortran77 and Fortran90 source code. You are invited to read the FortranCPP.hpp file to get more information about these macros.

Example of Fortran source code (FSubr.f90):

```
subroutine displayvector(Fsize,vect)

implicit none

integer Fsize,ifrom
real*8 vect(Fsize)

write(*,*) 'size',Fsize
write(*,*) (vect(i),i=1,Fsize)

return
end
```

Example of declaration block int the .cpp file (MySim.cpp):

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

Example of call of the fortran subroutine from the initializeRun method (MySim.cpp):

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

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

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

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

  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 File CMake.in.config containing the build configuration).

6.13 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

Documenting your simulators

The scientific documentation of simulators is very important to clearly understand the scientific concepts and methods applied in source code of simulators. 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. If present, the LaTeX-formatted text is placed in the main file of the simulator source code, into a single C++ comment block, and between the <sim2doc> and </sim2doc> tags.

Example of a part of source code including sim2doc informations:

```
/*
<sim2doc>
This part of the documentation will be included in the
It can be formatted using \LaTeX and is fully compatible with all \LaTeX commands,
including tables, scientific formulae, figures, and many more.
</sim2doc>
*/

BEGIN_SIMULATOR_SIGNATURE("example.simulator")

  DECLARE_NAME("Example simulator");
  DECLARE_DESCRIPTION("This simulator is an example");
  DECLARE_VERSION("13.05");
  DECLARE_STATUS(openfluid::ware::EXPERIMENTAL);
  DECLARE_AUTHOR("John","john@foobar.org");
  DECLARE_AUTHOR("Dave","dave@foobar.org");
  DECLARE_AUTHOR("Mike","mike@foobar.org");

  DECLARE_REQUIRED_PARAMETER("meanspeed","mean speed to use","m/s")

  DECLARE_REQUIRED_ATTRIBUTE("area","TU","area of the Test Units","m")
  DECLARE_USED_ATTRIBUTE("landuse","OU","landuse of the Other Units","")

  DECLARE_REQUIRED_VARIABLE("varA[double]","TU","","m")
  DECLARE_USED_VARIABLE("varB","OU","simple var on Other Units","kg")
  DECLARE_PRODUCED_VARIABLE("VarB[vector]","TU","vectorized var on Test Units","kg")
  DECLARE_UPDATED_VARIABLE("VarC","TU","","")

  DECLARE_USED_EVENTS("TU")

END_SIMULATOR_SIGNATURE
```

The final document can be generated using the OpenFLUID Sim2Doc buddy, included in the OpenFLUID command line program. See also Buddies for available options.

Example of generation of the PDF document using Sim2Doc tool:

```
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:

- `buddy` : Execute a buddy. Available buddies are newsim, newdata, sim2doc, examples
- `report` : Display informations about available wares
- `run` : Run a simulation from a project or an input dataset
- `show-paths` : Show search paths for wares

Available options:

- `-help,-h` : display this 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:

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

Example of running a simulation from a project:

```
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
- `-list, -l` : display simple list instead of report
- `-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` : show errored wares during search

Example of detailed reporting about available simulators:

```
openfluid report simulators
```

Example of reporting as a list about available observers:

```
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)

A.2.4 Buddies

Execute a buddy. Available buddies are newsim, newdata, sim2doc, examples

Usage : `openfluid buddy [<options>] [<args>]`

Available options:

- `-help,-h` : display this help message
- `-buddy-help` : display specific buddy help
- `-options=<arg>, -o <arg>` : set buddy options

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

$$\left[\begin{array}{ccccc} 1.5 & 19.6 & 0.005 & 1.0 & 258.99 \end{array}\right]$$

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

$$\left[\begin{array}{ccc} 1.5 & 19.6 & 0.005 \\ 2.0 & 1.0 & 258.99 \end{array}\right]$$

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

$$\left\{\begin{array}{lcl} key1 & = & 0.005 \\ key2 & = & \text{"}a\ word\text{"} \\ key3 & = & \left[\begin{array}{ccccc} 1.5 & 19.6 & 0.005 & 1.0 & 258.99 \end{array}\right] \end{array}\right.$$

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

C.2.4 TreeValue

The string format for TreeValue is not stable and will be updated in further versions to match the philosophy of string formats for other compound OpenFLUID values.

Appendix D

File formats for generators

D.1 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.

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

 <datasources>
    <filesource ID="1" file="source1.dat" />
    <filesource ID="2" file="source2.dat" />
  </datasources>

</openfluid>
```

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.

```
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
```

D.2 Distribution file

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

```
1 1
```

```
2 2
3 1
4 2
5 1
```