



# OpenFLUID

Software Environment  
for Modelling Fluxes in Landscapes

## Développement de simulateurs

Jean-Christophe Fabre

*LISAH*

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



This document is licensed  
under Creative Commons license

# Plan

- 1 Généralités
- 2 Signature de simulateur
- 3 Code de calcul de simulateur
- 4 Kit de développement

# Plan

- 1 Généralités
  - Structure d'un simulateur OpenFLUID
  - Interactions avec le framework OpenFLUID
- 2 Signature de simulateur
- 3 Code de calcul de simulateur
- 4 Kit de développement

# Structure générale

Un simulateur est un **code en C++** comprenant deux parties distinctes

- La **signature**
- Le **corps** intégrant le **code de calcul** ("le simulateur")

L'ensemble est intégré dans un fichier `.cpp` (et éventuellement d'autres fichiers `.cpp` et `.hpp` qui peuvent être inclus).

Ce(s) fichier(s) est(sont) compilé(s) sous la forme d'un **plugin indépendant** pouvant être reconnu et utilisé par le framework OpenFLUID

- nom = identifiant du simulateur
- suffixe = `_ofware-sim`
- extension = `.so` (Unix/Linux), `.dylib` (MacOSX), `.dll` (Windows)

# Signature

La signature contient une **description du simulateur** :

- identifiant du simulateur
- variables utilisées
- variables produites
- paramètres de simulateur utilisés
- attributs spatiaux nécessaires
- évènements pris en compte
- méta-information : version, auteur(s), nom, description, ...

La signature doit **refléter le comportement** du simulateur

La signature peut être consultée en ligne de commande (`openfluid -r`), ou via l'interface graphique.

# Corps du simulateur

Le corps du simulateur est constitué d'une **classe C++** unique.

Cette classe implémente des **méthodes** qui doivent être **obligatoirement présentes** (même si elles sont vides).

Ces méthodes correspondant à différentes **phases de simulation**

- Préparation
- Initialisation
- Exécution du calcul
- Finalisation

# Vue d'ensemble (pseudo-simulateur)

## Exemple

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

DECLARE_SIMULATOR_PLUGIN

BEGIN_SIMULATOR_SIGNATURE("my.own.simulator")
  // ici la signature a completer
END_SIMULATOR_SIGNATURE

class MySimulator : public openfluid::ware::PluggableSimulator
{
  // ici la classe a completer
  // avec les methodes obligatoires
  // incluant le code de calcul
}

DEFINE_SIMULATOR_CLASS(MySimulator);
```

# Interactions avec le framework OpenFLUID

Le framework charge **dynamiquement** les simulateurs constituant le modèle couplé lors des simulations.

La signature de chaque simulateur est utilisée par le framework pour

- connaître les **entrées/sorties** des simulateurs
- **vérifier la cohérence** entre simulateurs, et du modèle couplé dans son ensemble

Les méthodes définies par chaque simulateur sont exécutées (appelées) à différentes phases de simulation.



# Plan

- 1 Généralités
- 2 Signature de simulateur
  - Définition du simulateur
  - Déclaration des données manipulées
- 3 Code de calcul de simulateur
- 4 Kit de développement

# Identification du simulateur et autres informations

- L'identifiant du simulateur est obligatoire
- D'autres informations facultatives peuvent être ajoutées pour mieux décrire le simulateur

## Exemple

```
BEGIN_SIMULATOR_SIGNATURE("formation.exemple");  
  DECLARE_NAME("Simulateur_exemple");  
  DECLARE_DESCRIPTION("Ce_simulateur_est...");  
  
  DECLARE_VERSION("10.04");  
  DECLARE_STATUS(openfluid::ware::EXPERIMENTAL);  
  
  DECLARE_DOMAIN("formation");  
  DECLARE_PROCESS("exemple");  
  DECLARE_METHOD("simple");  
  DECLARE_AUTHOR("Bruce_Wayne", "bruce@waynecorp.com");  
  DECLARE_AUTHOR("Alfred_Pennyworth", "alfred@waynecorp.com");  
END_SIMULATOR_SIGNATURE
```

# Variables

Les variables déclarées **produites**, **requises** ou **utilisées** doivent comporter

- un **nom**
- la **classe d'unité** à laquelle elles sont attachées
- une description et une unité SI

Pour une classe d'unité donnée, un nom de variable doit être **unique**

## Exemple

```
BEGIN_SIMULATOR_SIGNATURE("formation.exemple");  
  // variable produite  
  DECLARE_PRODUCED_VAR("exemple.var0","UnitsA","variable_0","m");  
  // variable requise  
  DECLARE_REQUIRED_VAR("exemple.var1","UnitsA","variable_1","kg");  
  // variable utilisee si presente  
  DECLARE_USED_VAR("exemple.var2","UnitsA","variable_2","-");  
END_SIMULATOR_SIGNATURE
```

# Variables et types de données

- Variable **non typée** : les valeurs peuvent être **de n'importe quel type**
- Variable **typée** : toutes les valeurs doivent être **du type de la variable**

La vérification de cohérence production/utilisation tient compte du typage/non-typage des variables

## Exemple

```
BEGIN_SIMULATOR_SIGNATURE("formation.exemple");  
  // variable produite  
  DECLARE_PRODUCED_VAR("exemple.var0","UnitsA","variable_0","m");  
  // variable produite de type matrice  
  DECLARE_PRODUCED_VAR("exemple.var4[matrix]","UnitsA","variable_4","-");  
  // variable requise  
  DECLARE_REQUIRED_VAR("exemple.var1","UnitsA","variable_1","kg");  
  // variable requise de type double  
  DECLARE_REQUIRED_VAR("exemple.var3[double]","UnitsA","variable_3","?");  
END_SIMULATOR_SIGNATURE
```

# Attributs

Les attributs spatiaux déclarés **produits**, **requis** ou **utilisés** doivent comporter

- un **nom** d'attribut existant
- une **classe d'unité** à laquelle ils sont attachés
- une description et une unité SI

## Exemple

```
BEGIN_SIMULATOR_SIGNATURE("formation.exemple");  
  DECLARE_REQUIRED_ATTRIBUTE("input1","UnitsA",  
                             "attribute_1","m");  
  DECLARE_USED_ATTRIBUTE("input2","UnitsA","attribute_2","-");  
END_SIMULATOR_SIGNATURE
```

# Paramètres

Les paramètres de simulateur déclarés doivent comporter

- un **nom** de paramètre
- une description et une unité SI

## Exemple

```
BEGIN_SIMULATOR_SIGNATURE("formation.exemple");  
  DECLARE_SIMULATOR_PARAM("param1", "", "-");  
  DECLARE_SIMULATOR_PARAM("param2", "", "m/s");  
END_SIMULATOR_SIGNATURE
```

## Exemple complet

```
BEGIN_SIMULATOR_SIGNATURE("formation.exemple");
  DECLARE_NAME("Simulateur_exemple");

  DECLARE_VERSION("13.06");
  DECLARE_STATUS(openfluid::ware::BETA);
  DECLARE_DOMAIN("formation");
  DECLARE_AUTHOR("Anthony_Stark","stark@shield.org");
  DECLARE_AUTHOR("Obi-Wan_Kenobi","kenobi@tatooine.org");

  DECLARE_PRODUCED_VAR("var.exemple.var0","UnitsA","","m");
  DECLARE_REQUIRED_VAR("exemple.var1","UnitsA","","kg");
  DECLARE_USED_VAR("exemple.var2","UnitsA","var_2","-");

  DECLARE_REQUIRED_ATTRIBUTE("input1","UnitsA","","m");
  DECLARE_USED_ATTRIBUTE("input2","UnitsA","input_data_2","-");

  DECLARE_SIMULATOR_PARAM("param2","","m/s");
END_SIMULATOR_SIGNATURE
```

# Plan

- 1 Généralités
- 2 Signature de simulateur
- 3 Code de calcul de simulateur**
  - Structure
  - Manipulation de données
- 4 Kit de développement



# Un simulateur est une classe C++

Une classe définissant un simulateur doit hériter de la classe `openfluid::ware::PluggableSimulator`

Elle doit **redéfinir les méthodes publiques obligatoires** suivantes:

- `initParams()`
- `prepareData()`
- `checkConsistency()`
- `initializeRun()`
- `runStep()`
- `finalizeRun()`

## Classe C++ de simulateur "vide"

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

    MySimulator() : PluggableSimulator() { }
    ~MySimulator() { }

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

    void prepareData()
    { }

    void checkConsistency()
    { }

    openfluid::base::SchedulingRequest initializeRun()
    { return DefaultDeltaT(); }

    openfluid::base::SchedulingRequest runStep()
    { return DefaultDeltaT(); }

    void finalizeRun()
    { }
};
```

## Méthodes d'un simulateur (préparation)

`void initParams(openfluid::ware::WareParams_t Params)`

- **récupération des paramètres de simulateur** depuis la section `<model>`, via le paramètre `Params`

`void prepareData()`

- **préparation des données** avant vérification de cohérence

`void checkConsistency()`

- **vérification** de la cohérence interne du simulateur

## Méthodes du simulateur (simulation)

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

- phase d'**initialisation de la simulation** : initialisation des variables produites, calcul d'invariants, mise en place de "lookup tables", chargement de fichiers propres, ...

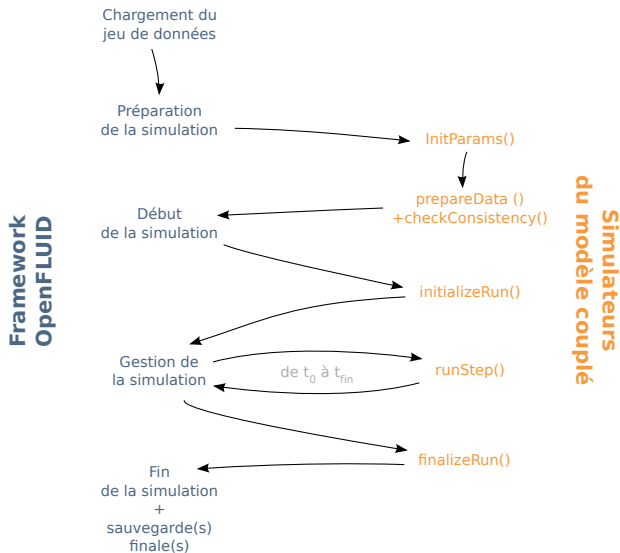
`openfluid::base::SchedulingRequest runStep()`

- exécution des **calculs à un index de temps donné**

`void finalizeRun()`

- phase de **finalisation de la simulation** : calcul de bilans, sauvegarde de fichiers propres, ...

# Séquence d'interaction avec le framework OpenFLUID



# Boucles spatiales

Une boucle spatiale permet de **parcourir** l'ensemble des **unités d'une même classe d'unité**, selon leur **ordre de traitement**

- définition de boucle: `OPENFLUID_UNITS_ORDERED_LOOP()`
- délimitation de la boucle par un bloc d'instructions: `{ }`

## Exemple

```
{
  openfluid::core::Unit* pUA;

  // boucle sur les unites de classe "UnitsA"
  // a chaque tour de boucle, pUA pointe sur l'unite courante
  OPENFLUID_UNITS_ORDERED_LOOP("UnitsA", pUA)
  {
    // effectuer ici les calculs sur l'unite courante
    // pointee par pUA
  }
}
```

# Informations sur l'avancée de la simulation

Les **informations temporelles** sur la simulation peuvent être obtenue via les fonctions suivantes

## Informations **invariantes**

- OPENFLUID\_GetBeginDate() : Date de début de la période de simulation
- OPENFLUID\_GetEndDate() : Date de fin de la période de simulation
- OPENFLUID\_GetSimulationDuration() : Durée de la simulation en secondes
- OPENFLUID\_GetDefaultDeltaT() : DeltaT par défaut

## Informations **évolutives** au cours de la simulation

- OPENFLUID\_GetCurrentTimeIndex() : Index de temps courant
- OPENFLUID\_GetCurrentDate() : Date courante
- OPENFLUID\_GetPreviousRunTimeIndex() : Index de temps lors de la précédente exécution du simulateur

# Manipulation de variables: Initialisation

Les variables **doivent être initialisées** depuis la méthode `initializeRun()`

- initialisation: `OPENFLUID_InitializeVariable()`

## Exemple

```
openfluid::base::SchedulingRequest initializeRun()
{
    openfluid::core::Unit* pUA;

    OPENFLUID_UNITS_ORDERED_LOOP("UnitsA", pUA)
    {
        OPENFLUID_InitializeVariable(pUA, "exemple.var", 0.0);
    }

    return DefaultDeltaT();
}
```



# Manipulation de variables: Accès

L'accès aux valeurs de variables n'est possible que depuis la méthode `runStep()`

Plusieurs approches d'accès aux variables sont possibles

- `OPENFLUID_GetVariable()` : Accès à la **valeur à l'index de temps courant** ou à **un index de temps précédent** pour la variable
- `OPENFLUID_GetLatestVariable()` : Accès à la **dernière valeur disponible** pour la variable
- `OPENFLUID_GetLatestVariables()` : Accès aux **dernières valeurs disponibles** pour la variable **depuis un index de temps** donné
- `OPENFLUID_GetVariables()` : Accès aux **valeurs disponibles** pour la variable **entre deux index de temps** donnés

# Manipulation de variables: Accès

## Exemple

```
openfluid::base::SchedulingRequest runStep()
{
    openfluid::core::Unit* pUA;
    openfluid::core::DoubleValue Value, PrevValue;
    openfluid::core::IndexedValue LatestValue;

    OPENFLUID_UNITS_ORDERED_LOOP("UnitsA", pUA)
    {
        OPENFLUID_GetVariable(pUA, "example.var", Value);
        OPENFLUID_GetVariable(pUA, "example.var",
                               OPENFLUID_GetPreviousRunTimeIndex(), PrevValue);

        OPENFLUID_GetLatestVariable(pUA, "example.var", LatestValue);
    }

    return Duration(90);
}
```

# Manipulation de variables: Production

La production de valeurs de variables n'est possible que depuis la méthode `runStep()`

Une valeur de variable produite est **obligatoirement indexée avec l'index de temps courant**

- `OPENFLUID_AppendVariable()`

## Exemple

```
openfluid::base::SchedulingRequest runStep()
{
    openfluid::core::Unit* pUA;
    openfluid::core::DoubleValue Value;

    OPENFLUID_UNITS_ORDERED_LOOP("UnitsA", pUA)
    {
        OPENFLUID_GetVariable(pUA, "example.var", Value);
        Value = Value + 1;
        OPENFLUID_AppendVariable(pUA, "example.varplus", Value);
    }

    return DefaultDeltaT();
}
```

# Manipulation de variables: Mise à jour

La mise à jour de valeurs de variables n'est possible que depuis la méthode `runStep()`

Une valeur de variable ne peut être mise à jour **que pour l'index de temps courant**

- `OPENFLUID_SetVariable()`

## Exemple

```
openfluid::base::SchedulingRequest runStep()
{
    openfluid::core::Unit* pUA;
    openfluid::core::DoubleValue Value;
    const double Adjustment = 0.005;

    OPENFLUID_UNITS_ORDERED_LOOP("UnitsA", pUA)
    {
        OPENFLUID_GetVariable(pUA, "example.var", Value);
        Value = Value + Adjustment;
        OPENFLUID_SetVariable(pUA, "example.var", Value);
    }

    return DefaultDeltaT();
}
```

# Accès aux attributs spatiaux

Les attributs spatiaux ne peuvent être accédés que depuis les méthodes `prepareData()`, `checkConsistency()`, `initializeRun()`, `runStep()`, `finalizeRun()`

- `OPENFLUID_GetAttribute()`

## Exemple

```
openfluid::base::SchedulingRequest runStep()
{
    openfluid::core::Unit* pUA;
    openfluid::core::DoubleValue Value,Data;

    OPENFLUID_UNITS_ORDERED_LOOP("UnitsA",pUA)
    {
        OPENFLUID_GetVariable(pUA,"example.var",Value);
        OPENFLUID_GetAttribute(pUA,"idata",Data);
        Value = Value + Data;
        OPENFLUID_AppendVariable(pUA,"example.varplusdata",Value);
    }

    return Never();
}
```

# Manipulation des paramètres de simulateur

Les paramètres de simulateur ne peuvent être accédés que depuis la méthode `initParams()`

- `OPENFLUID_GetSimulatorParameter()`

Afin de pouvoir être utilisées depuis d'autres méthodes, il est préférable de stocker les valeurs des paramètres dans des attributs privés de la classe C++

## Exemple

```
void initParams(openfluid::ware::WareParams_t Params)
{
    OPENFLUID_GetSimulatorParameter(Params, "paramA", &m_PA);
    OPENFLUID_GetSimulatorParameter(Params, "paramB", &m_PB);
}
```

# Manipulation des connexions entre unités

Les **connexions** sont **portées par les unités** elles-mêmes, accessibles via les méthodes

- liste des unités connectées "vers" : `getToUnits()`
- liste des unités connectées "depuis" : `getFromUnits()`

## Exemple

```
void prepareData()
{
    openfluid::core::Unit* pUA, pFromUA;
    openfluid::core::DoubleValue Area, AreaSum;

    OPENFLUID_UNITS_ORDERED_LOOP("UnitsA", pUA)
    {
        UpperAreaSum = 0;
        OPENFLUID_UNITS_LIST_LOOP(pUA->getFromUnits("UnitsA"), pFromUA)
        {
            OPENFLUID_GetAttribute(pFromUA, "area", &Area);
            UpperAreaSum = UpperAreaSum + Area;
        }
    }
}
```

# Messages d'erreurs et d'avertissements

Il est possible de faire remonter des **messages d'avertissement**

- avertissements non-bloquants : `OPENFLUID_RaiseWarning()`
- erreurs stoppant la simulation : `OPENFLUID_RaiseError()`

## Exemple

```
openfluid::base::SchedulingRequest runStep()
{
    openfluid::core::Unit* pUA;
    openfluid::core::IntegerValue Value;

    OPENFLUID_UNITS_ORDERED_LOOP("UnitsA", pUA)
    {
        OPENFLUID_GetVariable(pUA, "example.var", &Value);

        if (Value < 0)
            OPENFLUID_RaiseError("simulator.example", "runStep()",
                                "Error_message_from_example_simulator");
    }

    return DefaultDeltaT();
}
```



# Plan

- 1 Généralités
- 2 Signature de simulateur
- 3 Code de calcul de simulateur
- 4 Kit de développement**
  - API et documentation
  - Environnement de développement

# API OpenFLUID

Application Programming Interface = ensemble des **fonctionnalités OpenFLUID disponibles** pour développer des simulateurs, des observateurs, des applications logicielles, des bindings de langage, ...

## Principaux espaces de nommage OpenFLUID

- `openfluid::core` (types et structures de données)
- `openfluid::base` (base de fonctionnement)
- `openfluid::ware` (branchement des plugins)
- `openfluid::tools` (outils de traitement de données)
- `openfluid::fluidx` (gestion des descripteurs de simulation et I/O)
- `openfluid::machine` (moteur de simulation)
- `openfluid::landr` (fonctionnalités de traitement spatial)

⇒ bibliothèques logicielles et fichiers d'en-têtes C++

# Documentation

Disponible via l'espace communautaire OpenFLUID sur le web

<http://www.openfluid-project.org/community/>

- Manuel de l'utilisateur
- Guide de développement, documentation de l'API
- Guides de bonnes pratiques
- Code snippets ("Bouts de code")
- Information de migration (changement de version du moteur de calcul)
- Support
- ...

## Environnement de développement préconisé

### Compilateurs : suite GCC

- g++, gfortran

### Système de construction/test/packaging

- CMake, CTest, CPack

### Environnement de développement intégré (IDE)

- Eclipse + CDT + Plugin Eclipse pour OpenFLUID

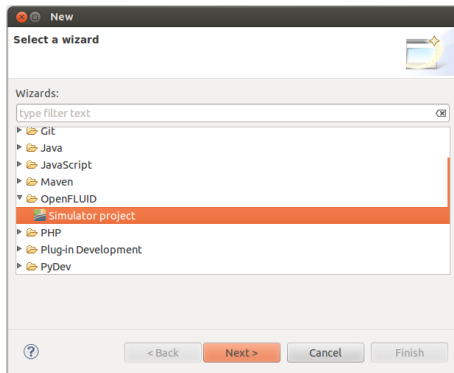
### Support : canal IRC, mailing-list, bug-tracking

- <http://www.openfluid-project.org/community/>

# Création d'un simulateur sous Eclipse (1/4)

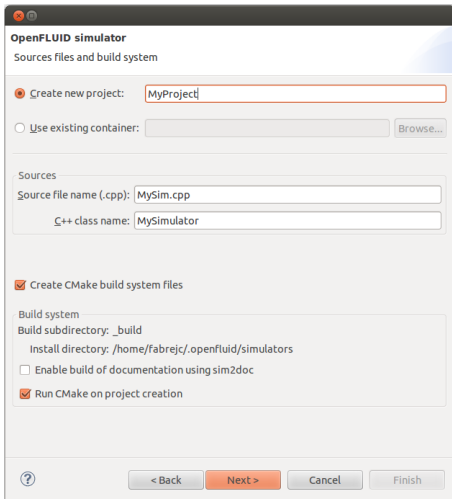
Assistant de création de simulateur OpenFLUID

Menu File > New > Other ...



# Création d'un simulateur sous Eclipse (2/4)

## Définition de la structure du code et de la compilation/construction



# Création d'un simulateur sous Eclipse (3/4)

## Déclaration de l'ID et de la méta information

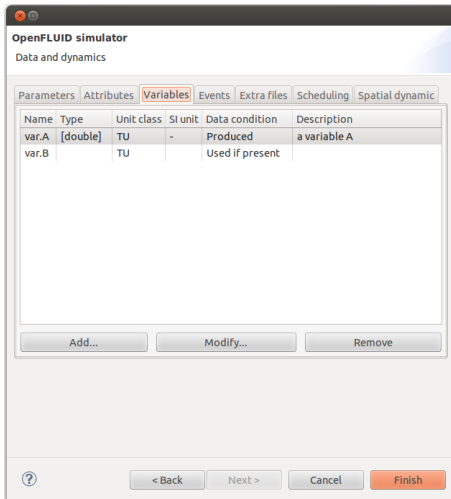
The screenshot shows a dialog box titled "OpenFLUID simulator" with the subtitle "Simulator ID and meta-information". It contains several input fields and a text area:

- Simulator ID:
- Simulator name:
- Simulator version:
- Simulator Domain:
- Description:
- Author's name:
- Author's email(s):

At the bottom of the dialog, there is a help icon (question mark), and four buttons: "< Back", "Next >", "Cancel", and "Finish".

# Création d'un simulateur sous Eclipse (4/4)

## Déclaration du comportement et des données manipulées





# Compilation/construction d'un simulateur sous Eclipse

Une fois l'assistant complété, le code source est automatiquement généré

- **signature** intégrant les informations renseignées dans l'assistant
- **classe C++** du simulateur (à compléter avec le code de calcul)

Le système de compilation/construction est également généré (si option cochée)

Pour **compiler/construire** et **installer le simulateur**, il faut aller dans le menu Project > Build ou cliquer sur le marteau dans la barre d'outils

Cette opération est à effectuer pour que les **modifications du code source soient prises en compte**.

## Pour en savoir plus...

### Fonctionnalités non présentées

- Variables de différents types (vecteur, matrice, entiers, ...)
- Tests d'existence de variables, d'attributs spatiaux, ...
- Création d'attributs spatiaux
- Utilisation d'évènements discrets
- Production d'évènements discrets
- Gestion de fichiers propres aux simulateurs
- Intégration de code en d'autres langages (Fortran, ...)
- Attributs de types élaborés (vecteur, matrice, ...)
- Parallélisation des calculs
- Connectivité de type parents/enfants entre unités
- ...

## Références & Ressources



Jean-Christophe Fabre.  
Guide du développeur LISAH.  
[fabrejc@supagro.inra.fr](mailto:fabrejc@supagro.inra.fr).



Eclipse IDE.  
<http://www.eclipse.org/>.



Site web OpenFLUID.  
<http://www.openfluid-project.org/>.



Site web OpenFLUID Community.  
<http://www.openfluid-project.org/community/>.