OpenFLUID

Software Environment
for Spatial Modelling in Landscapes

Développement de simulateurs

Equipe OpenFLUID

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

Plan

Plan

Structure générale

Un simulateur OpenFLUID est un code source
basé sur un modèle mathématique décrivant le ou les processus à simuler

Le code source d'un simulateur comprend deux parties distinctes:

- La signature
- Le code de calcul

Chaque simulateur

- produit et utilise des variables
- s'appuie sur l'organisation (topologie,connexité) et les différentes propriétés du domaine spatial (attributs)
- peut prendre en compte des paramètres qui lui sont propres

Signature

La signature contient une description du simulateur:

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

La signature doit refléter le comportement du simulateur

Code de calcul du simulateur

Le code de calcul du simulateur est constitué de différentes parties correspondant à différentes phases de la simulation:

- Préparation : traitements préparatoires avant la simulation
- Initialisation : initialisation des variables de couplage produites
- Exécution du calcul : calculs exécutés à chaque pas de temps du simulateur
- Finalisation : opérations de finalisation de la simulation

Code source en C++

Le code source d'un simulateur doit être écrit en C++
et intégré dans un fichier `.cpp`
Il peut être accompagné de fichiers complémentaires si nécessaire

- structuration de codes sources complexes
- intégration native de codes sources compatibles (C, Fortran, ...)
- intéropérabilité dynamique avec des codes sources externes (R, Python, ...)

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<version>-sim`
- extension = `.so` (Unix/Linux), `.dylib` (MacOSX), `.dll` (Windows)

Code source en C++

La signature est déclarée dans le code source à l'aide de directives de signature dédiées

Le code de calcul est regroupé dans une classe C++ comprenant les différentes parties correspondantes à chacune des phases de la simulation

Vue d'ensemble (pseudo-simulateur)

Exemple

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

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

Plan

Identification du simulateur et autres informations

- L'identifiant obligatoire du simulateur
- 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
```

Identifiant de simulateur

L'identifiant de simulateur est unique.
Il permet à OpenFLUID de manipuler le simulateur.

Caractères autorisés: caractères alphanumériques, ., -,_
Expression régulière : `[A-Za-z0-9]+([A-Za-z0-9\.\-]*)`

Nomenclature par convention (non obligatoire):
`thématique.compartiment.processus.méthode`

- water.atm-surf.rain-su.files
- pop.uz-sz.soil-transfer.watsfar
- water-crop.surf-uz.aqyield
- soil.surf.SSC-evolution.logistic-regression
- land.surf.representation.geomhydas
- ...

Voir aussi https://community.openfluid-project.org/scidev/simdatanaming/

Variables de couplage

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

Exemple

```
BEGIN_SIMULATOR_SIGNATURE("formation.exemple");
  // variable produite
  DECLARE_PRODUCED_VARIABLE("exemple.var0","UnitsA","variable␣0","m");
  // variable requise
  DECLARE_REQUIRED_VARIABLE("exemple.var1","UnitsA","variable␣1","kg");
  // variable utilisee si presente
  DECLARE_USED_VARIABLE("exemple.var2","UnitsA","variable␣2","-");
END_SIMULATOR_SIGNATURE
```

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

Nom d'une variable de couplage

La "clé" de couplage OpenFLUID se base sur le nom et la classe d'unité d'une variable

Caractères autorisés: caractères alphanumériques, ., -,_

Expression régulière : `[A-Za-z0-9]+([A-Za-z0-9\.\-]*)`

Nomenclature par convention (non obligatoire):
`thématique.compartiment.grandeur.nom`

- water.surf.H.infiltration
- water.surf.H.runoff
- water.surf.Q.downstream
- plant.canopy.A.lai
- pop.surf.M.liquid-phase
- pop.surf.C.liquid-phase
- ...

Voir aussi https://community.openfluid-project.org/scidev/simdatanaming/

Variables de couplage 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_VARIABLE("exemple.var0","UnitsA","variable␣0","m");
  // variable produite de type matrice
  DECLARE_PRODUCED_VARIABLE("exemple.var4[matrix]","UnitsA","variable␣4","-");
  // variable requise
  DECLARE_REQUIRED_VARIABLE("exemple.var1","UnitsA","variable␣1","kg");
  // variable requise de type double
  DECLARE_REQUIRED_VARIABLE("exemple.var3[double]","UnitsA","variable␣3","?");
END_SIMULATOR_SIGNATURE
```

Attributs spatiaux

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 de simulateur

Les paramètres de simulateur déclarés requis ou utilisés doivent comporter

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

Exemple

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

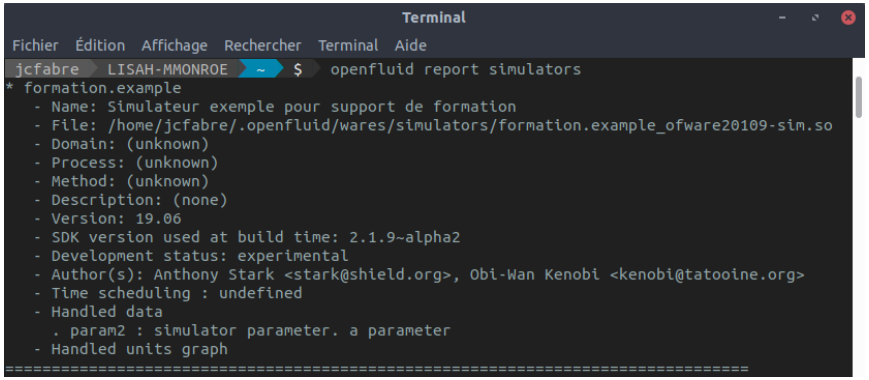
Exemple complet

```
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_VARIABLE("var.exemple.var0","UnitsA","","m");
  DECLARE_REQUIRED_VARIABLE("exemple.var1","UnitsA","","kg");
  DECLARE_USED_VARIABLE("exemple.var2","UnitsA","var␣2","-");

  DECLARE_REQUIRED_ATTRIBUTE("input1","UnitsA","","m");
  DECLARE_USED_ATTRIBUTE("input2","UnitsA","input␣data␣2","-");

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

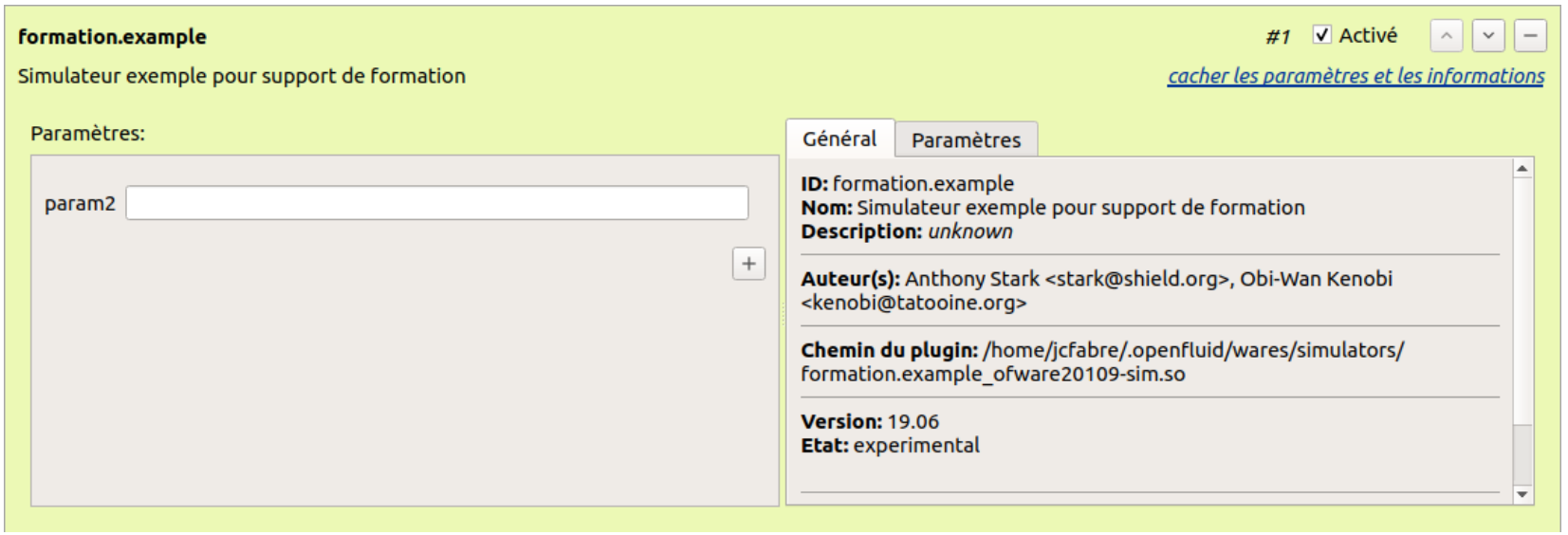
Consultation de la signature

en ligne de commande:

```
openfluid report simulators
```



ou dans l'interface graphique OpenFLUID-Builder



Plan

Code de calcul d'un simulateur

Un simulateur est une classe C++
(groupe de fonctionnalités rassemblées)

Cette classe C++ doit hériter de (être liée à)
la classe prédéfinie `openfluid::ware::PluggableSimulator`

La classe C++ du simulateur doit proposer les méthodes 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

phases de préparation des calculs

`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 d'un simulateur

phases de 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, ...

Planification du temps des simulateurs

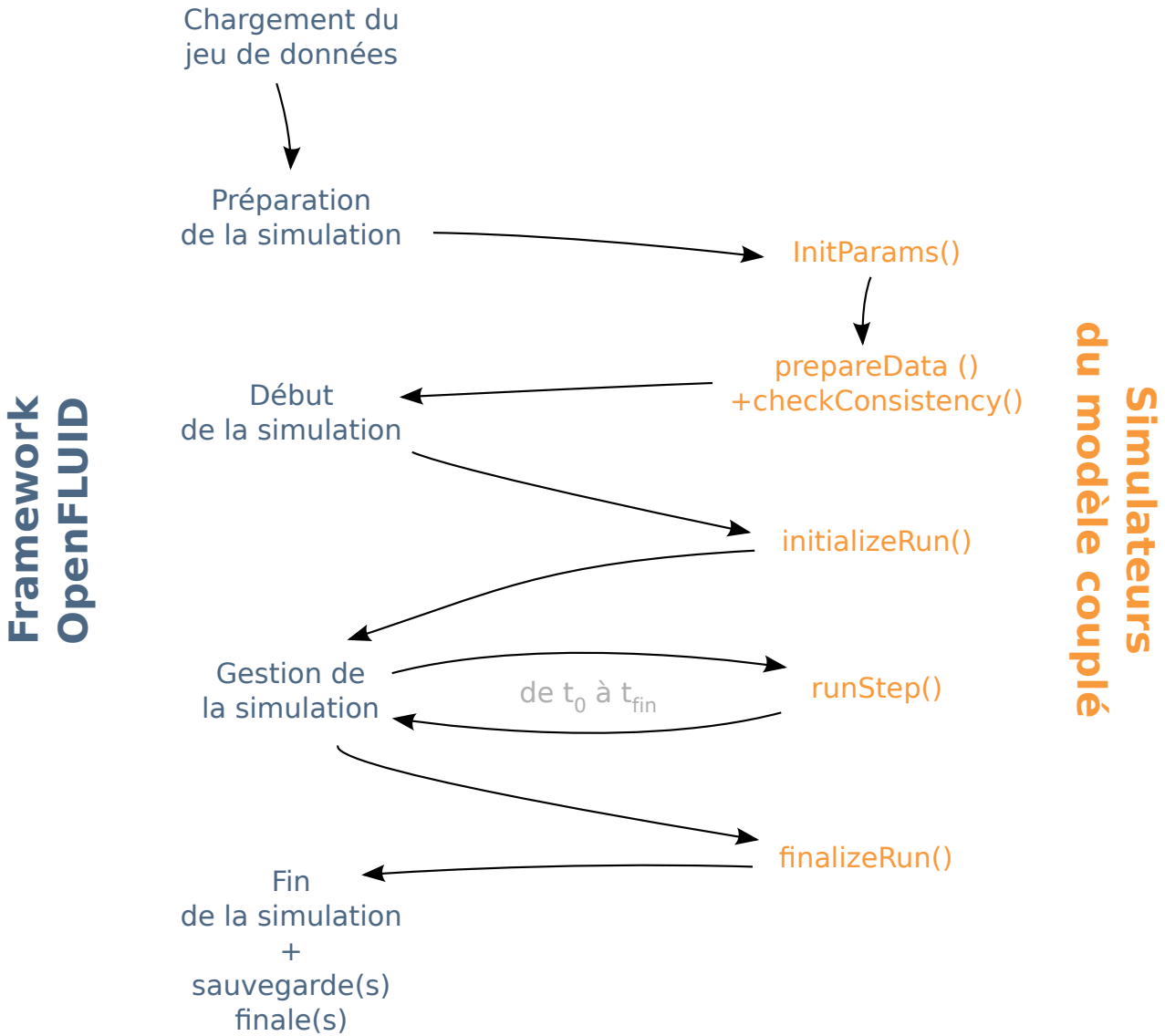
Chaque simulateur précise le nombre de secondes avant sa prochaine ré-exécution sous la forme d'une valeur de retour:

- pour initializeRun() : nombre de secondes avant le premier runStep()
- pour runStep() : nombre de secondes avant le prochain runStep()

Possibilités de planification:

- DefaultDeltaT() : pas de temps par défaut du modèle couplé
- Duration(n) : une durée *n* en secondes, arbitraire ou calculée
- AtTheEnd() : programme une dernière exécution en toute fin de simulation
- Never() : aucune nouvelle exécution, le simulateur est mis en sommeil

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::SpatialUnit* 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 obtenues 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::SpatialUnit* pUA;

  OPENFLUID_UNITS_ORDERED_LOOP("UnitsA",pUA)
  {
    OPENFLUID_InitializeVariable(pUA,"example.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::SpatialUnit* 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::SpatialUnit* 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::SpatialUnit* 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::SpatialUnit* 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();
}
```

Accès aux évènements discrets

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

- `OPENFLUID_GetEvents()`

Exemple

```
openfluid::base::SchedulingRequest runStep()
{
  openfluid::core::SpatialUnit* pUA;
  openfluid::core::DoubleValue Value,Data;
  openfluid::core::DateTime EventPeriodEnd = OPENFLUID_GetCurrentDate();
  openfluid::core::DateTime EventPeriodBegin = EventPeriodEnd - 86399;

  OPENFLUID_UNITS_ORDERED_LOOP("UnitsA",pUA)
  {
    openfluid::core::EventsCollection EvColl;
    OPENFLUID_GetEvents(pUA,BeginDate,EndDate,EvColl);
    // ici des calculs bases sur les evenements
    OPENFLUID_AppendVariable(pUA,"example.varevents",Value);
  }

  return DefaultDeltaT();
}
```

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 spatiales

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

- liste des unités connectées "vers" : `toSpatialUnit()`
- liste des unités connectées "depuis" : `fromSpatialUnit()`

Exemple

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

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

Messages d'erreurs et d'avertissements

Il est possible d'émettre des messages pendant la simulation

- informations : `OPENFLUID_LogInfo()`
- avertissements non-bloquants : `OPENFLUID_LogWarning()`
- erreurs stoppant la simulation : `OPENFLUID_RaiseError()`

Exemple

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

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

    if (Value < 0)
      OPENFLUID_RaiseError("Error message from example simulator");
  }

  return DefaultDeltaT();
}
```

Pour aller plus loin...

Autres fonctionnalités et instructions (non présentées)

- Manipulation de différents types de variables (vecteur, matrice, entiers, ...)
- Tests d'existence de variables, d'attributs spatiaux, ...
- Création d'attributs spatiaux
- 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 spatiales
- ...

Plan

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)

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

Environnement de développement OpenFLUID

Présentation générale

OpenFLUID-DevStudio:

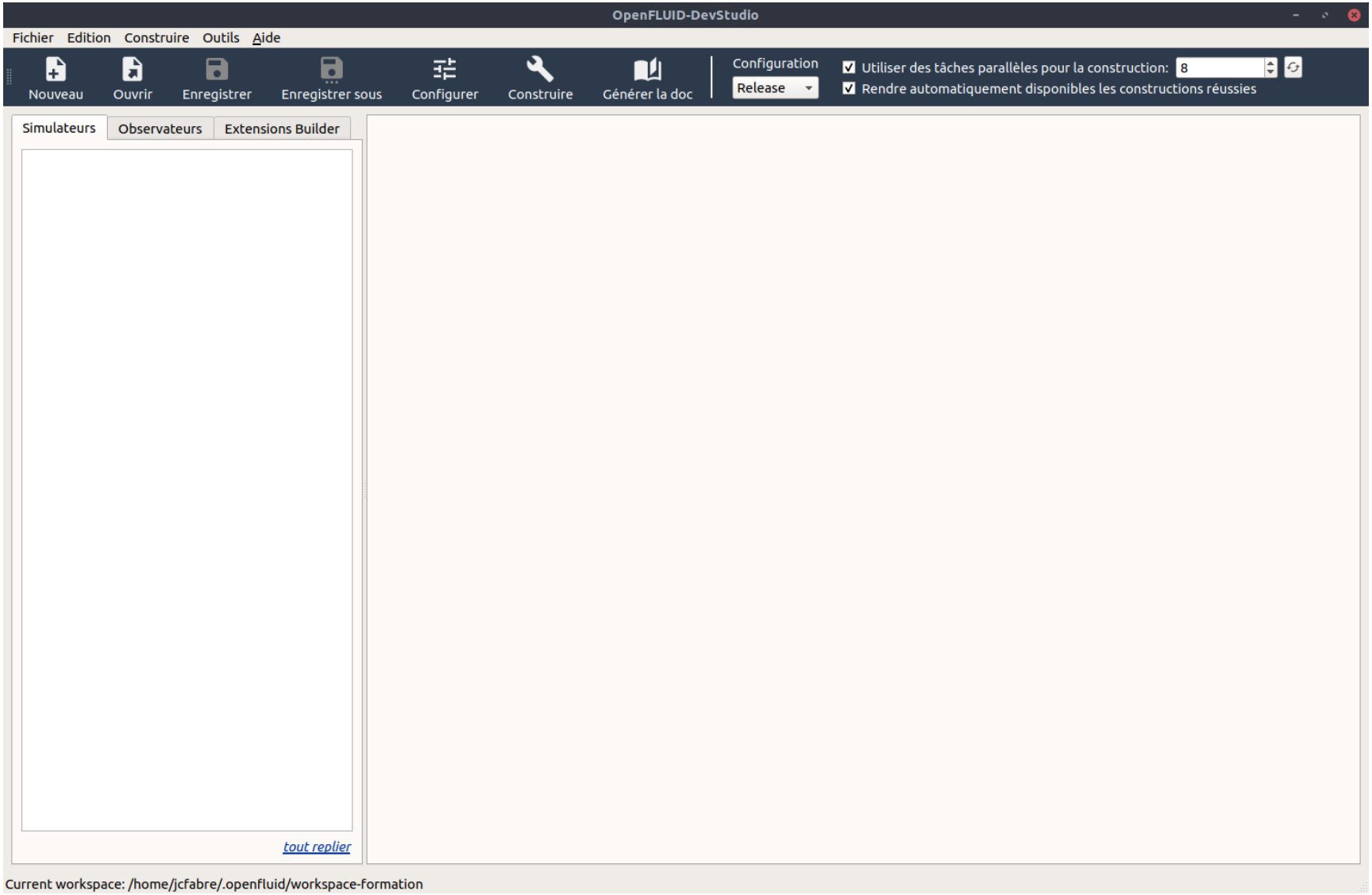
- Environnement de développement pour OpenFLUID
- Création, développement, construction de simulateurs (+ observateurs et builder-extensions)
- Assistance au développement, automatisation du processus de construction/installation
- Intégration possible dans OpenFLUID-Builder

Outils mobilisés:

- Compilateurs : g++, gfortran
- Système de construction/test/packaging : CMake, CTest, CPack

Environnement de développement OpenFLUID

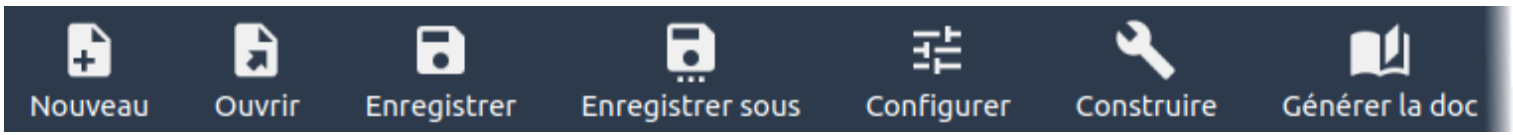
Présentation générale



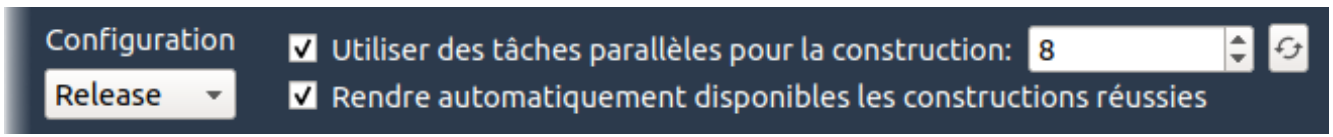
Environnement de développement OpenFLUID

Barre d'outils

Partie gauche : gestion de fichiers et construction



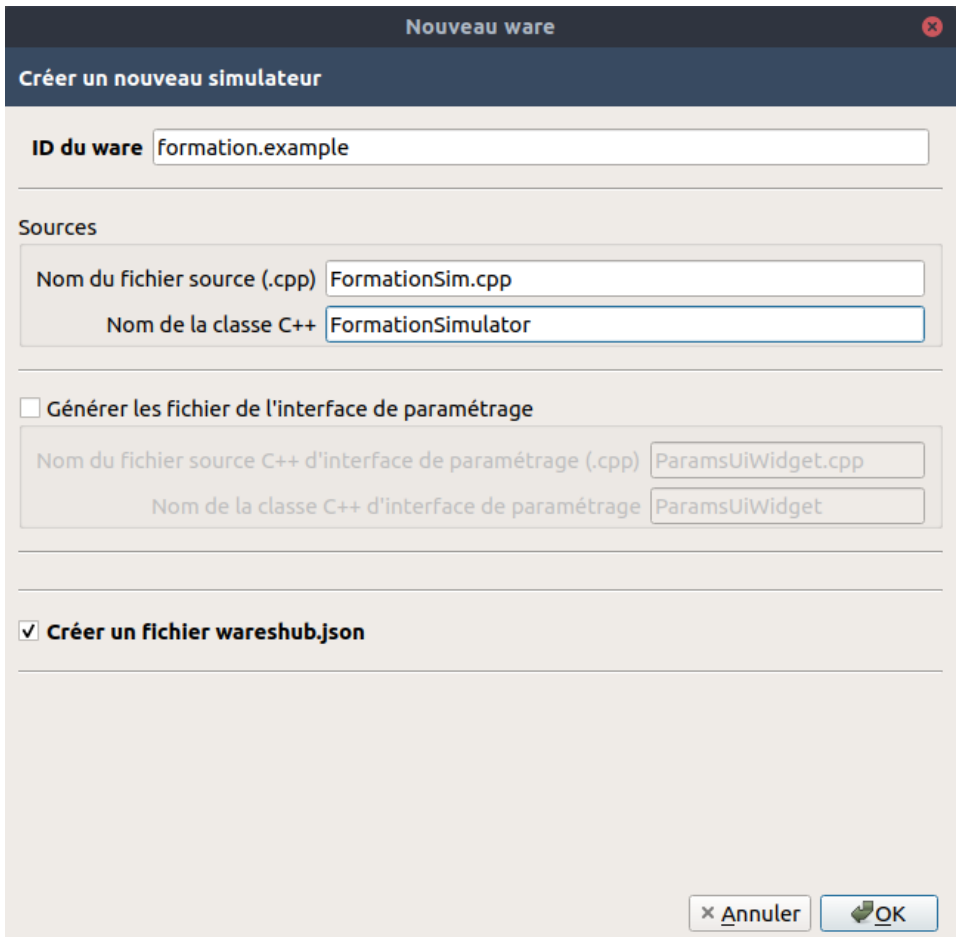
Partie droite : accès rapide aux préférences



Environnement de développement OpenFLUID

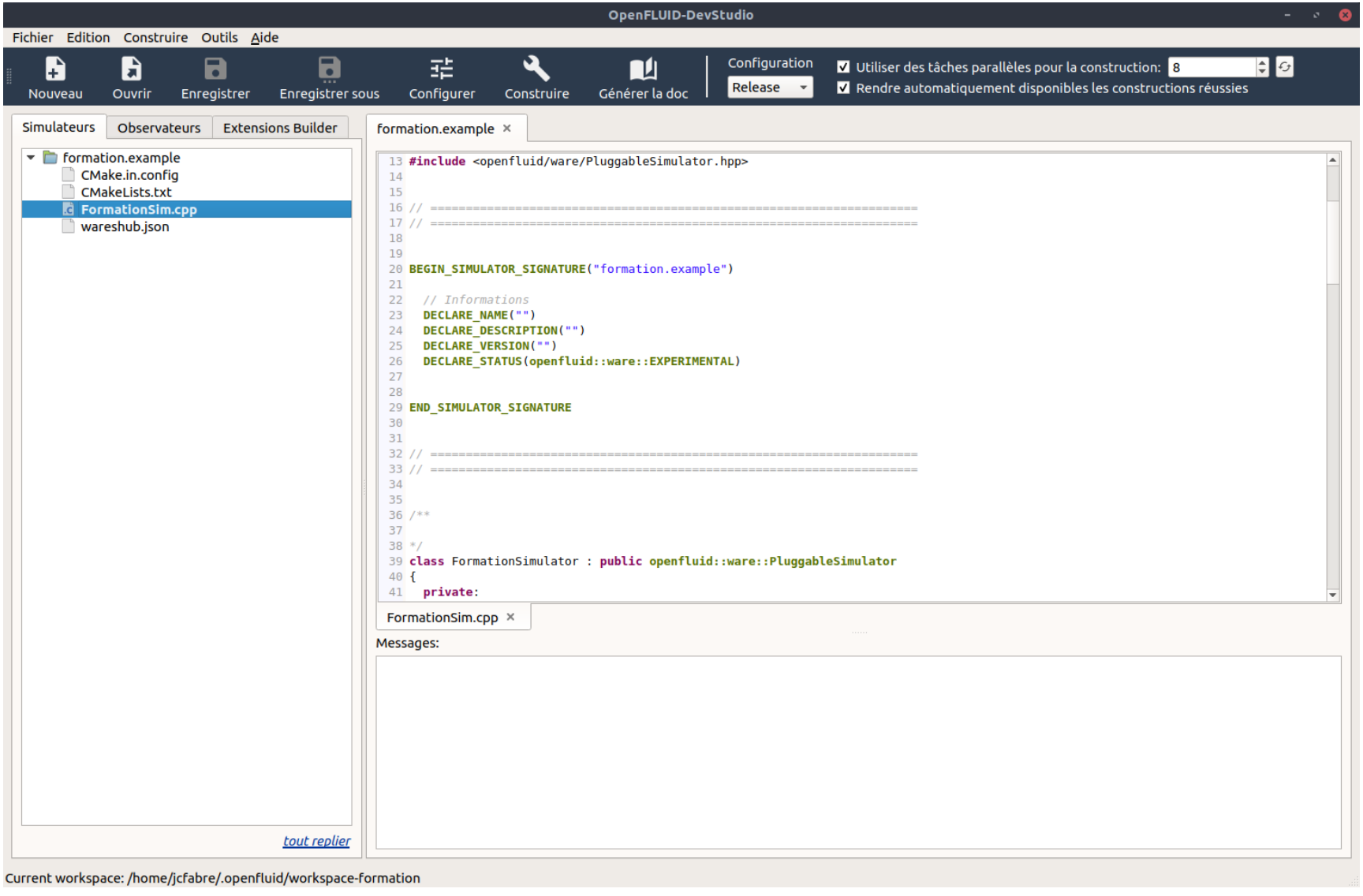
Création d'un nouveau simulateur

Menu Fichier > Nouveau ware > Simulateur...



Environnement de développement OpenFLUID

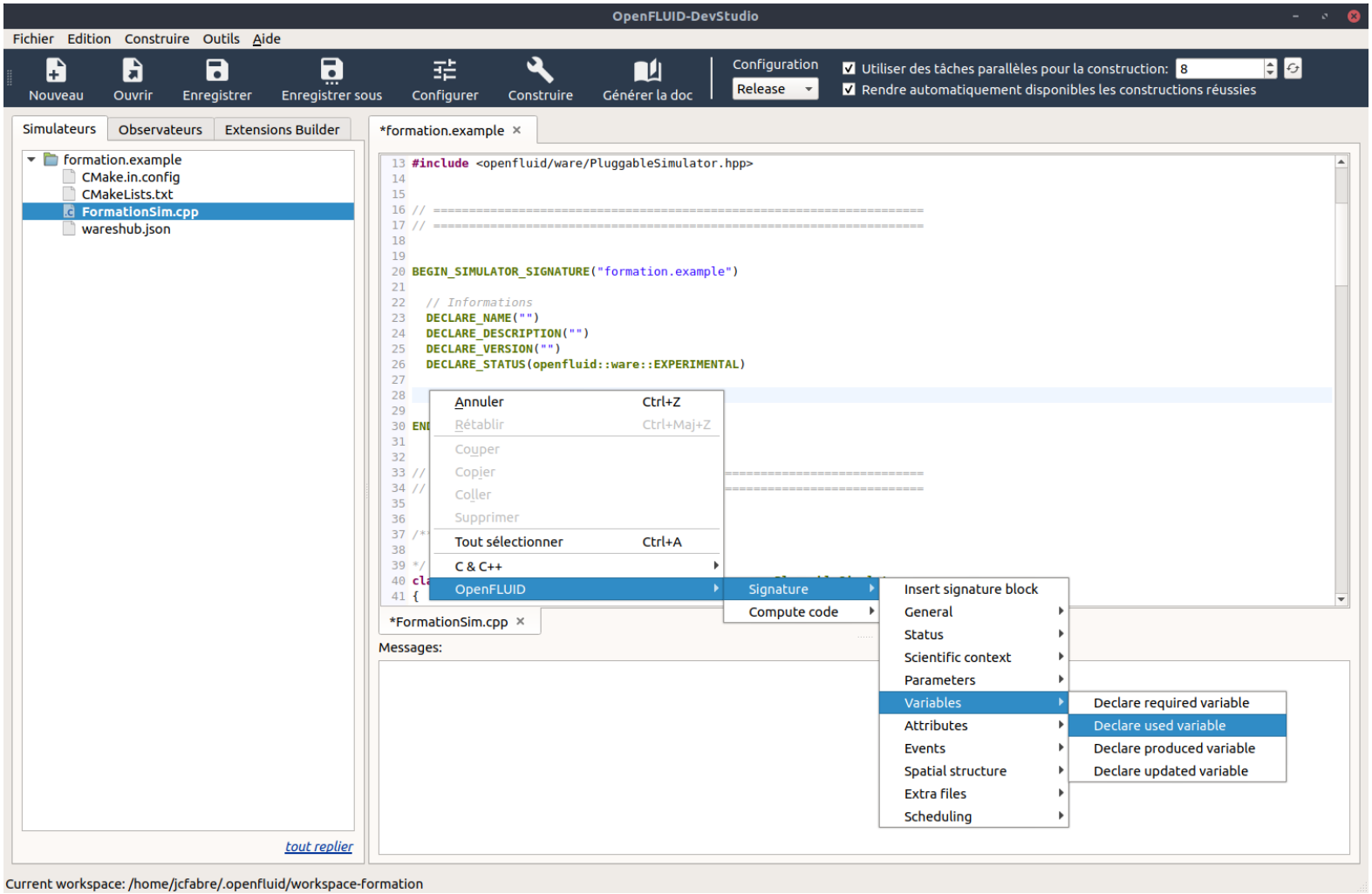
Edition du code source



Environnement de développement OpenFLUID

Assistance au développement - Menu contextuel

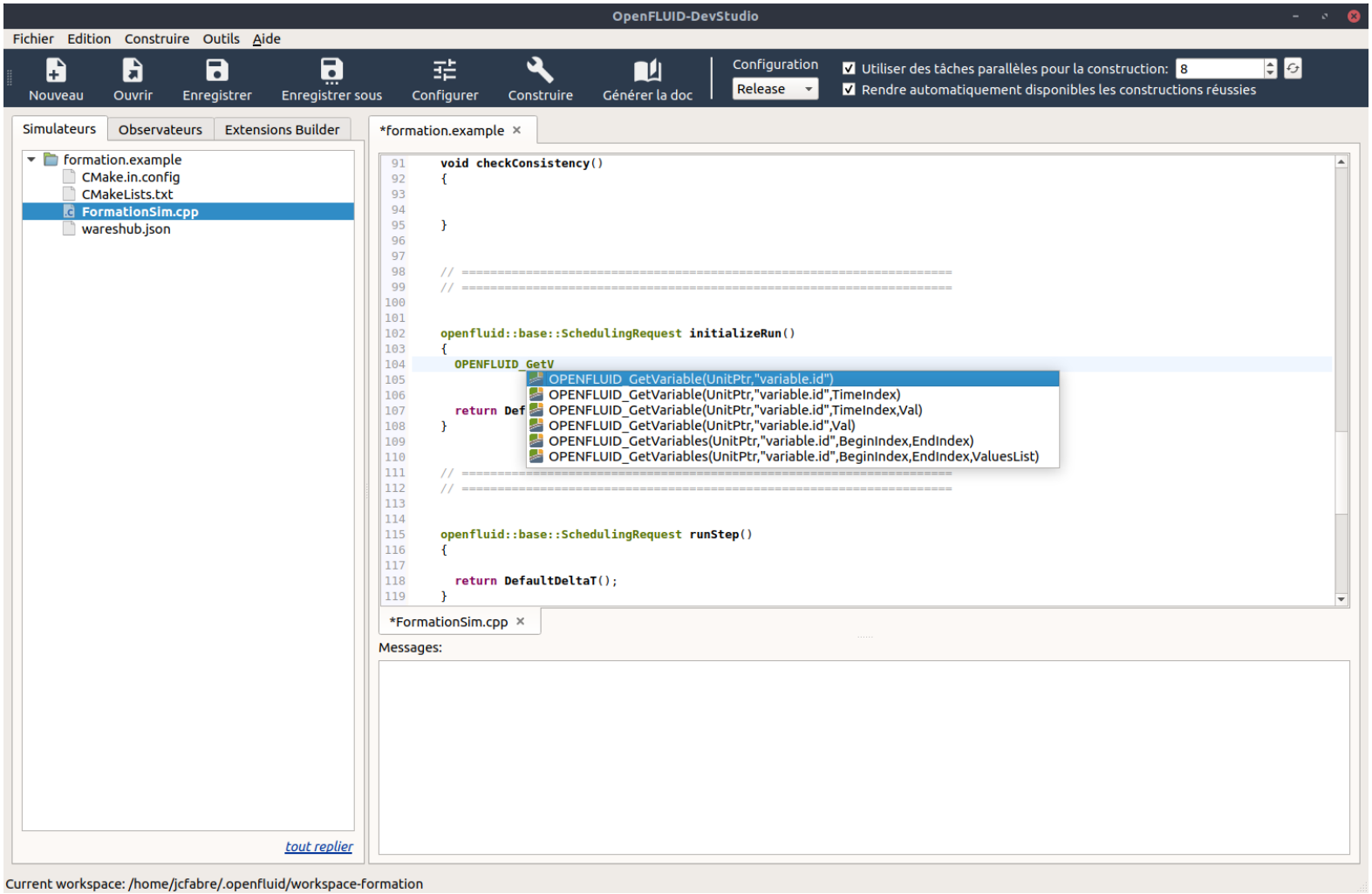
Clic droit dans l'éditeur > OpenFLUID



Environnement de développement OpenFLUID

Assistance au développement - Completion

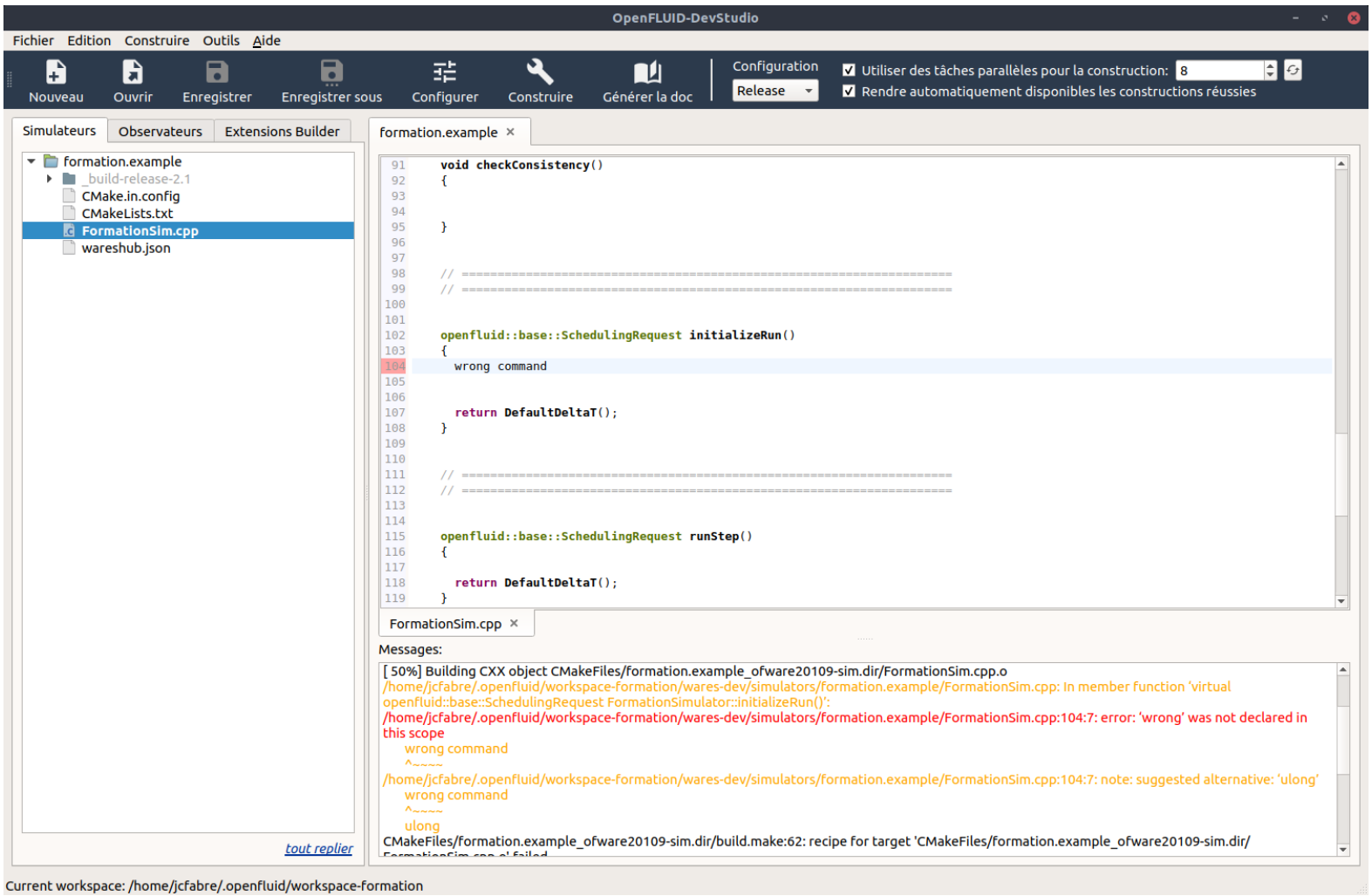
Combinaison clavier : Control + Espace



Environnement de développement OpenFLUID

Configuration / construction d'un simulateur

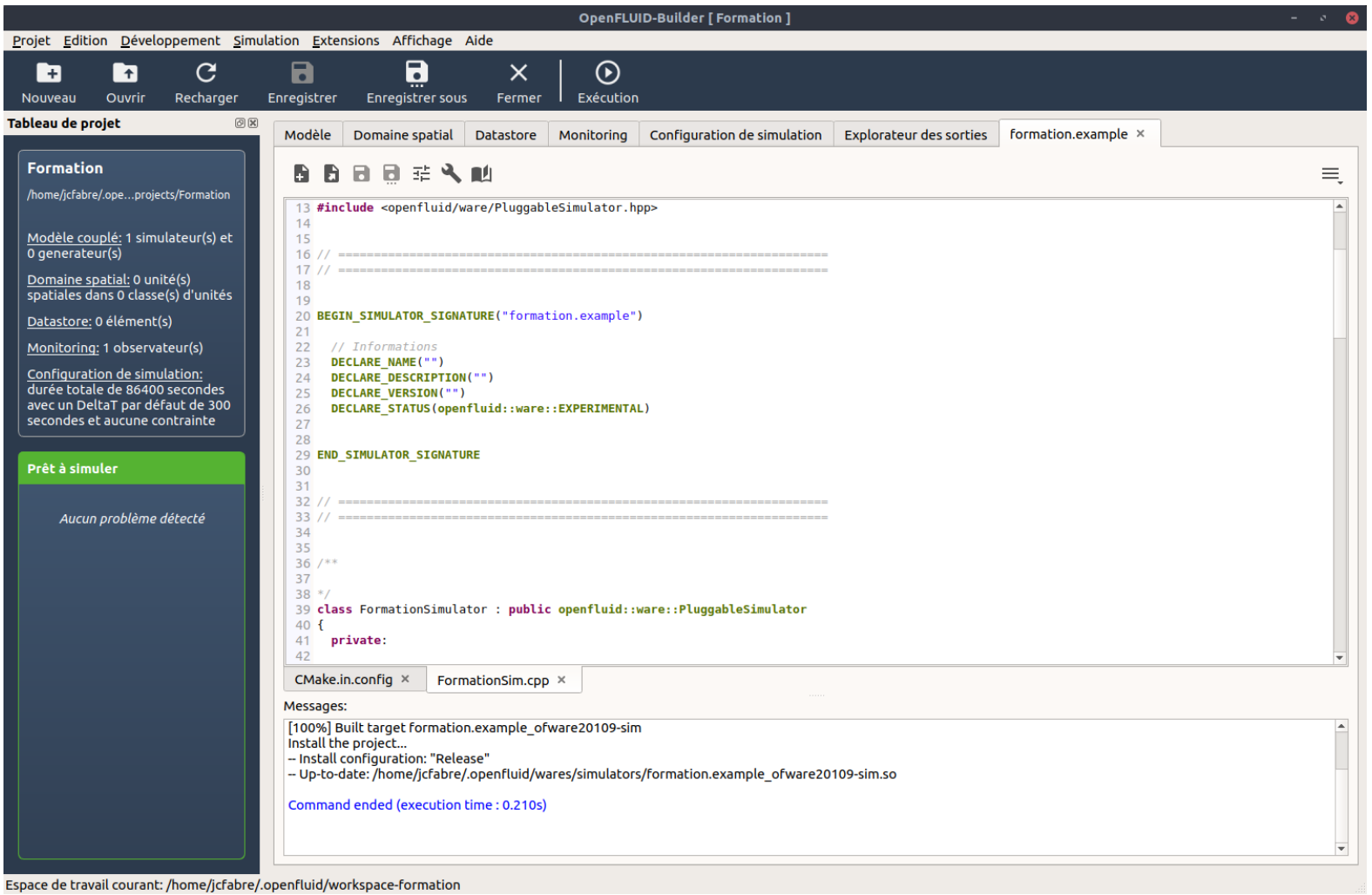
Menu Construire ou barre d'outils



Environnement de développement OpenFLUID

Intégration dans OpenFLUID-Builder

Doucle-clic sur un simulateur



Environnement de développement OpenFLUID

Divers

Edition:

- Touche F1 pour l'aide en ligne
- Marquage des instructions OpenFLUID dépréciées

Configuration/Construction:

- Configuration une seule fois au début du développement
- Sauvegarde du fichier et construction pour appliquer les modifications
- A la construction: erreurs et avertissements notifiés dans la marge du code source, messages d'erreurs cliquables

Documentation

Disponible via l'espace communautaire OpenFLUID sur le web

http://community.openfluid-project.org/

- 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)
- Assistance aux utilisateurs
- ...