Module MULTIPR pour Salomé
Partitionneur-Décimateur

Spécifications détaillées et conception

	Nom	Date	Visa
Auteur (s) :	B. PHETSARATH O. LE ROUX	20/02/2007	
Vérificateur(s) :	C. BOURCIER	21/02/2007	
Approbateur :	E. BOUVIER	21/02/2007	

Suivi des modifications

Version/Révision		Références		Description des modifications	Auteur(s)
Indice	Date	Page	N° §		
0.1	28/10/2006			Version initiale	B. PHETSARATH
0.2	17/11/2006		7	Finalisation et ajout des éléments de conception	O. LE ROUX
0.3	01/12/2006			Ajout de la référence [DR4] concernant MED parallèle et [DR5] concernant MED-fichier v2.3	O. LE ROUX
			6.4	Ajout de la spécification du fichier d'aiguillage	
			6.5	Spécification de l'utilisation des données produites par MULTIPR dans le module VISU	
			7	Correction : TETRA10 contient 5 points de Gauss et non pas 6	
			7.3	Ajout d'un paramètre dans Filtre_gradientMoyen pour configurer le boxing Chaque points lié à un observable est indexé dans une cellule unique (boxing) Correction du pseudo-algorithme de filtrage qui était incomplet	
			8.1	Ajout des informations concernant les branches CVS utilisées pour les modules MED et VISU	
1.0	30/01/2007			Version finale	O. LE ROUX
		14	5	Enrichissement des scénarios d'utilisation	
		17	6	Nommage des fichiers	
		18	6	Modification des spécifications du fichier d'aiguillage pour le rendre compatible avec l'outil MEDSPLITTER	
		20	7	Ajout d'un paragraphe sur la gestion des erreurs via les exceptions	
		21-27	7	Modifications dans la description des fonctions implémentées afin de coller aux dernières évolutions du code	
				Correction des spécifications concernant les paramètres de la décimation	
			8	Changement dans les branches CVS utilisées	
1.1	20/02/2007	6	4	Mise à jour des diagrammes de séquence	O. LE ROUX
		12	5	Complément sur la description des fonctions de l'objet MULTIPR_Obj (le nom des fonctions a également légèrement changé) et mise à jour des scénarios Python associés	

Diffusion

DIFFUSION EXTERNE

Guillaume THIBAULT	**EDF/R&D/SINETICS**
Stéphane PLOIX	**EDF/R&D/SINETICS**

DIFFUSION INTERNE

Equipe Projet	**CS/AIC**

DOCUMENTS APPLICABLES

DA1	Cahier des charges du 08/08/2006
DA2	Manuel d'Assurance de la Qualité : MAQ1000 Version 3 du 18/02/2004

DOCUMENTS DE REFERENCE

DR1	Partitionneur-Decimateur – Document de spécifications Réf : IOLS-WP1.2.1/SPEC/001/1.1 du 19/07/2006 Auteur : G. Thibault.
DR2	MEDMEM user's guide – Draft Réf : SFME/LGLS/RT/aa-nnn/A Auteurs : P. Goldbronn, E. Fayolle, N. Benhamou, J. Roy, V. Bergeaud, N. Crouzet.
DR3	MEDSPLITTER user's guide Réf : SFME/LGLS/RT/06-006/A Auteur : V. Bergeaud
DR4	Spécifications détaillées de l'extension de MED – Fichier aux maillages distribués Réf : EDF R&D H-I26-2006-00407-FR Auteur : S. Kortas
DR5	Documentation de la bibliothèque MED-fichier V2.3 : Modèle de données

Table des matières

Table des matières

Chapitre 1.

1. Introduction

1.1 Objectif du document

Ce document fournit la spécification du module Salomé MULTIPR.

Ce composant est un Partitionneur-Décimateur de données pour la visualisation avec le module VISU de Salomé.

2. Rappels

2.1 Définitions

- ✓ Composant (Salomé) : terme désignant dans ce document un module Salomé comme GEOM, SMESH, MULTIPR.... Il est employé ici pour lever les ambiguïtés avec le terme « module Python ». Voir chapitre 2.2.2 pour une description plus détaillée.
- ✓ Module python : terme désignant un fichier de sources Python dont les fonctionnalités peuvent être chargées dynamiquement par le mécanisme d'« import ».
- ✓ Diagramme de séquence : terme désignant un diagramme UML représentant les intéractions entre objets, qui s'ordonnent chronologiquement du haut vers le bas. Un diagramme d'utilisation permet notamment de présenter un cas d'utilisation.
- ✓ Interface IDL : terme désignant un fichier source en IDL. L'IDL est un langage de définition et non un langage de programmation grâce auquel on définit des interfaces et des structures de données et non des algorithmes. Une fois l'IDL définie, une interface dans un langage donné peut être générée (C, C++, Python, Java, ...). La communication lors de l'appel de cette IDL est assurée par le protocole CORBA.
- ✓ NamingService : terme désignant une sorte d'annuaire associant un objet Corba à un nom sous frome d'une chaîne de caractères.
- ✓ Module pylotage: terme désignant le module Python permettant l'exploitation éventuellement distante des services Salomé. Cette exploitation se fait depuis un processus Python ; elle est ainsi possible depuis tout script python.

2.2 Architecture technique de Salomé

2.2.1 Vue générale

On peut schématiser l'application Salomé comme un ensemble de **serveurs**. Selon leurs types, ceux-ci peuvent être exécutés dans un même **processus** ou dans des processus séparés.

Les communications se font à travers l'**ORB CORBA.**

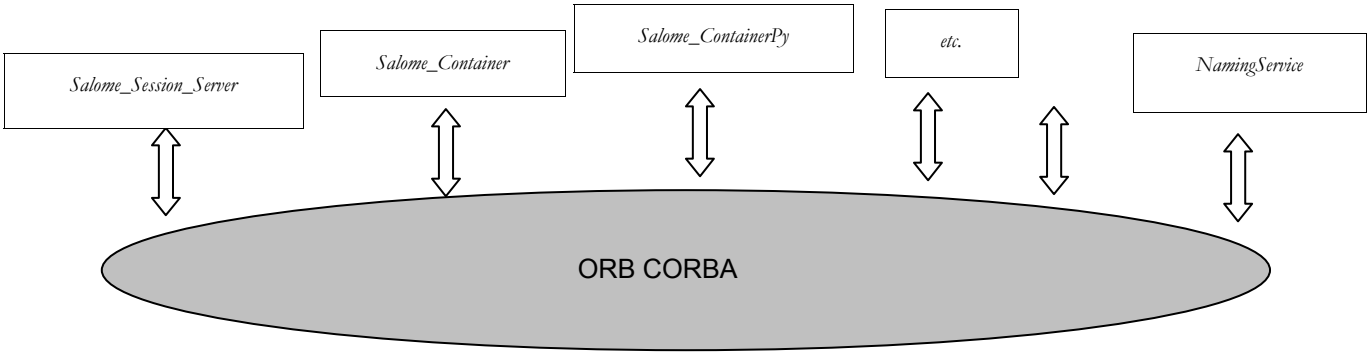


Figure 1 : Les serveurs dans Salomé

Chacun des serveurs fournit des **objets** répondant à des besoins bien spécifiques. Parmi ces objets on retrouve par exemple un gestionnaire d'étude, des containers, des composants Salomé (GEOM, SMESH, MULTIPR, etc.). Cet assemblage d'élément forme la **plate-forme** Salomé.

Par exemple : l'exécution d'une commande dans la partie *Menu->File* de l'IHM Salomé, implique l'appelle à une méthode de l'objet *myStudyManager*. Une bonne partie des actions de l'utilisateur sont ainsi retranscrites en appels sur des méthodes d'objet CORBA. Mais on peut dire que cela ne représente que la partie émergée de l'iceberg : ainsi, il existe des objets dédiés à la création d'autres objets (les *Containers*), des gestionnaires de ces mêmes objets, etc.

2.3 Le composant Salomé

Définition

Un *composant Salomé* est un ensemble logiciel qui comprend :

- ✓ Un moteur (*Engine*) en exécution dans un serveur et qui fournit généralement un service métier. Il correspond à l'implémentation d'un serveur CORBA dont le service est spécifié par une interface idl. Il peut se composer de plusieurs objets ;
- ✓ Une interface graphique (*GUI*) qui correspond aux éléments graphiques clients mis en œuvre dans l'*IAPP* Salomé (interface applicative). Elle peut fournir ou pas les moyens d'accéder au service proposé par le moteur.

L'implémentation de ces deux parties peut être réalisée en C++ ou en python.

Cycle de vie du moteur

Le moteur (*Engine*) du composant est un objet CORBA créé/détruit par un container.

Un *container* n'est rien d'autre qu'un autre objet CORBA. Les composants implémentés en C++ sont gérés par le container *FactoryServer*, ceux en python par le container *FactoryServerPy*.

Tous ces objets résident dans des processus serveurs : *Salome_Session_Server* (C++) et *Salome_ContainerPy.py* (python).

Implémentation

La structure informatique d'un composant suit une spécification bien établie au niveau :

- ✓ de l'organisation des fichiers et répertoires à l'installation. On doit respecter un *layout* type pour la disposition des fichiers ressources (icônes, traduction, idl, ...) et librairies du composant ;
- ✓ de l'implémentation des *callbacks* de la partie *GUI* du composant. Salomé communique avec le composant par ces méthodes qui doivent donc être implémentées au besoin (mais obligatoirement déclarées dans l'interface) ;
- ✓ de l'implémentation du moteur du composant. Celui-ci doit respecter une interface type d'un moteur : des méthodes et attributs doivent figurer absolument. On peut retrouver les définitions de ces interfaces obligatoires dans les fichier `Salome_Component.idl` et `SALOMEDS.idl`.

3. Environnement

3.1 Compilation et installation

Processus de compilation et installation standard OCC.

3.2 Exécution

Le tableau ci-dessous montre les éléments indispensables à la bonne exécution du composant.
On notera notamment la **dépendance** envers un autre composant Salomé : **PAL** .

Pré-requis	Description
`$MULTIPR_ROOT_DIR(*)`	Indique le répertoire d'installation du composant même.
`$PAL_ROOT_DIR(*)`	Indique le répertoire du composant PAL. Celui-ci regroupe des modules pythons (`studyManager.py`, ...) utilisés par le composant MULTIPR.
`SalomeApp.xml(**)`	Configuration du module pour Salomé .
`MULTIPRCatalog.xml(**)`	Définition des services du moteur pour le Superviseur .

Figure 2 : Dépendances à l'exécution.

(*) variable d'environnement.
(**) fichier.

4. Interface cliente

4.1 Réduction

La réduction de la quantité de données d'un maillage est la principale fonctionnalité du composant. On spécifiera dans ce chapitre l'interface offerte à l'utilisateur pour réaliser ce processus de réduction. Des diagrammes de séquences détailleront les opérations qui découlent de chacune des actions.

4.1.1 Sélection fichier MED

Cette opération consiste à sélectionner le fichier MED contenant le maillage à réduire.

L'utilisateur fournit :

- ✓ Le chemin du fichier MED contenant le maillage à réduire. Une boite de dialogue de type « File Browser » l'aide dans sa sélection.

On obtient :

- ✓ La création par l'interface serveur d'un objet CORBA MULTIPR_Obj (cf. 5.3) associé à ce fichier.
- ✓ Au niveau de l'IHM, un item représentant l'objet CORBA MULTIPR_Obj est ajouté dans l'arbre d'étude de Salomé.
- ✓ La liste des maillages contenus dans le fichier MED
- ✓

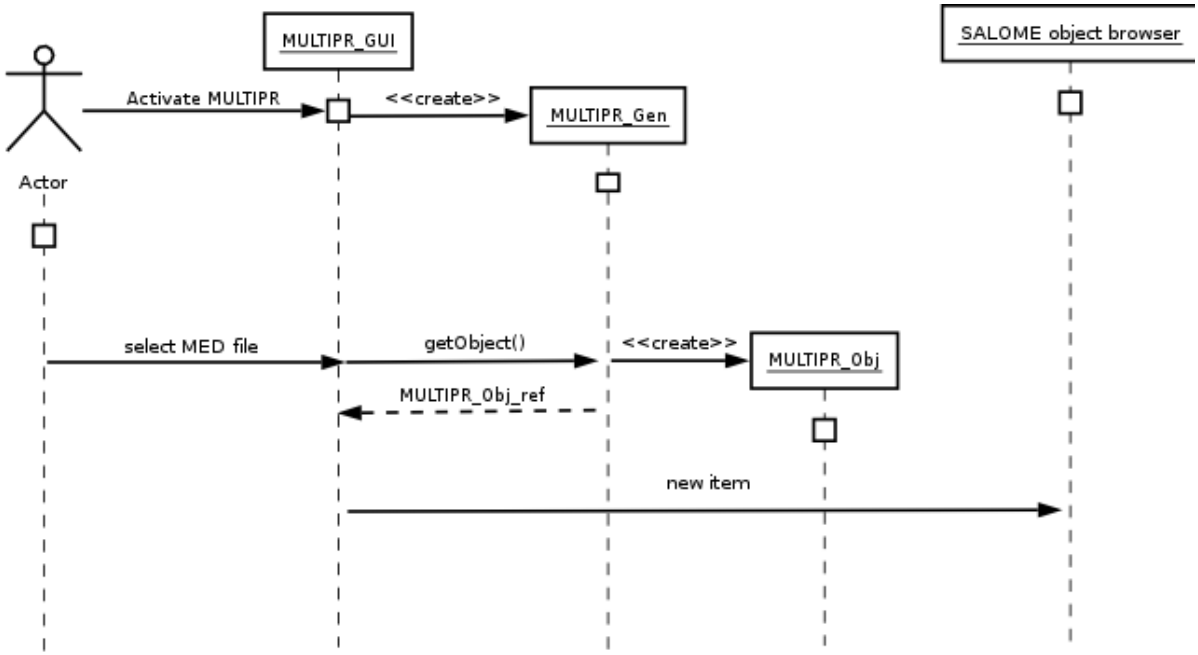


Figure 3 : Diagramme de séquence : sélection fichier MED

Note : Dans ce scénario, la sélection du fichier MED est précédée par l'activation du composant MULTIPR.

4.1.2 Sélection maillage

Cette opération consiste à sélectionner le maillage à réduire.

L'utilisateur fournit :

- ✓ Le nom d'un des maillages contenus dans le fichier MED sélectionné (cf. 4.1.1). Une liste de tous les maillages lui est proposée.

On obtient :

- ✓ L' objet CORBA MULTIPR_Obj (crée en 4.1.1) est mis à jour. Les opérations de partition et décimation porteront dorénavant sur le maillage sélectionné.

4.1.3 Partition

C'est la première étape de réduction. Elle consiste à partitionner le maillage.

L'utilisateur fournit :

- ✓ Le type de partitionnement souhaité : une échelle (en grains) ou à deux échelles (en grains puis intra-grain).
- ✓ Cas une échelle : on a toutes les informations. On peut lancer le partitionnement en grains (Partition_Domaine) du maillage sélectionné (cf. 4.1.2). Les résultats de l'opération (les grains) seront affichés dans une liste pour pouvoir être exploités par l'opération de décimation (cf. 4.1.4).
- ✓ Cas deux échelles : le partitionnement du maillage en grains (Partition_Domaine) est effectué implicitement. L'utilisateur entre en paramètre le nombre de partitions souhaitées des grains. Les résultats de l'opération (les partitions des grains) seront affichés dans une liste pour pouvoir être exploités par l'opération de décimation (cf. 4.1.4)

On obtient :

- ✓ Le résultat de l'opération de partitionnement : une liste de partitions (des grains ou des partitions de grains).
- ✓ L' objet CORBA MULTIPR_Obj (créé en 4.1.1) crée un fichier MED par partition et un fichier MED d'aiguillage.

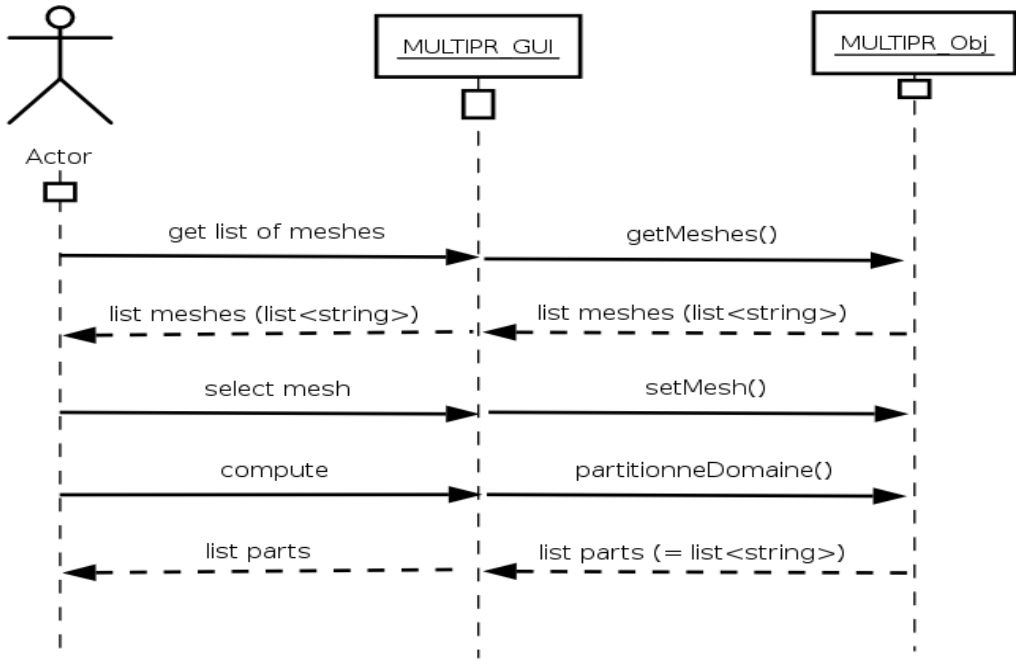


Figure 4 : Diagramme de séquence : une échelle

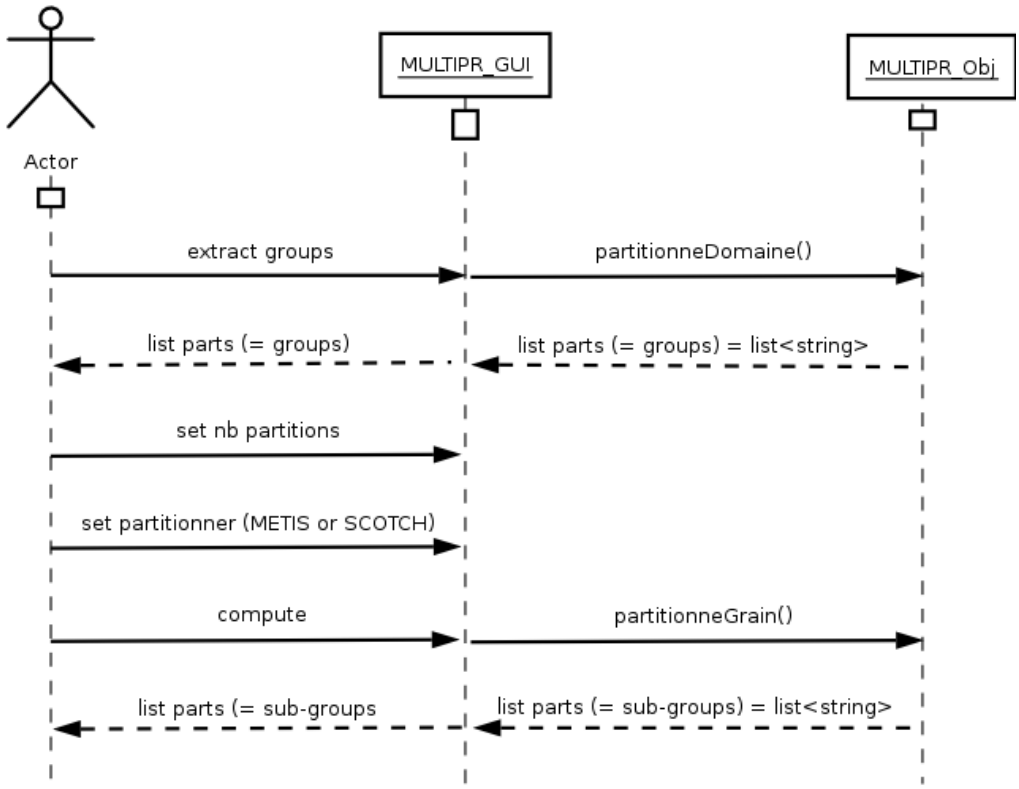


Figure 5 : Diagramme de séquence : deux échelles

4.1.4 Décimation

C'est la deuxième étape de réduction. Elle consiste à produire différents niveaux de détails aux partitions.

L'utilisateur fournit :

- ✓ Le type de filtre : par défaut (et pour l'instant le seul), Filtre_GradientMoyen.
- ✓ Le nom du champ sur lequel on applique le filtre.
- ✓ Le pas de temps du champ à considérer.
- ✓ Le rayon de recherche des voisins (optionnel).
- ✓ Le ratio de la boîte englobante utilisé pour le boxing (optionnel, 1% par défaut).
- ✓ La valeur du seuil bas (resp. haut) permettant de générer la résolution MEDIUM (resp. BASSE)
- ✓ La liste des partitions à décimer parmi celles fournies par l'opération de partition.

On obtient :

- ✓ Le fichier d'aiguillage décrivant les opérations de partition et décimation.
- ✓ L' objet CORBA MULTIPR_Obj (crée en 4.1.1) crée un fichier MED par partition et niveaux de détails et un fichier MED d'aiguillage.

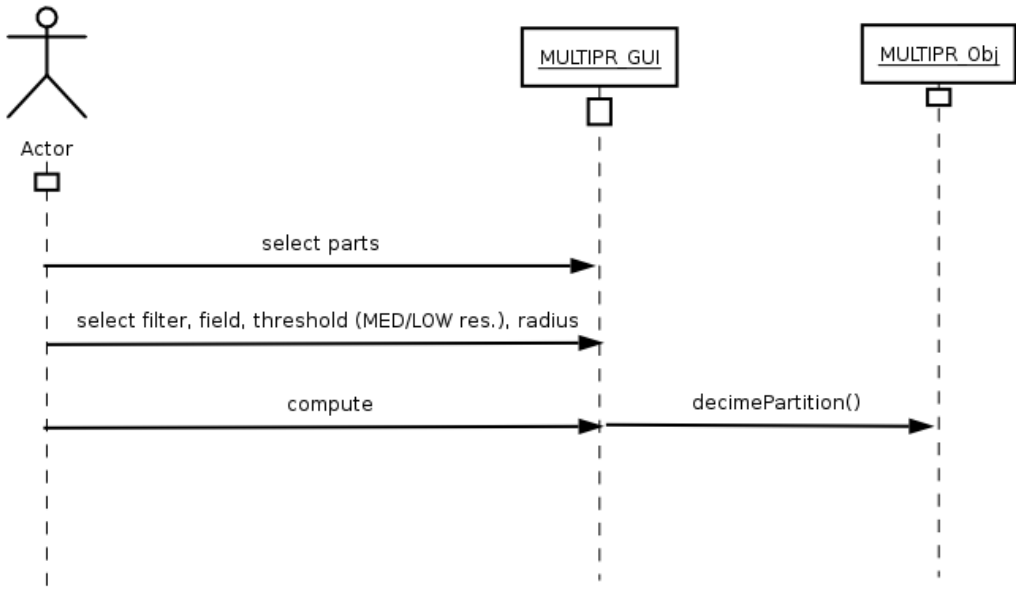


Figure 6 : Diagramme de séquence : décimation

Note : dans ce scénario, on suppose qu'un scénario de partition à une ou deux échelles se soit déroulé auparavant : la sélection des partitions correspond aux grains (une échelle) ou aux partitions d'un grain (deux échelles).

4.2 Exportation des résultats

L'utilisateur peut sauvegarder la trace des opérations et résultats d'une réduction qu'il a effectué en l'exportant dans un fichier d'aiguillage ou les exploiter directement dans le composant Post-Pro de Salomé : VISU.

4.2.1 Fichier d'aiguillage

Le fichier MED d'aiguillage contient toute les informations de localisation des fichiers MED résultats produits par l'opération de réduction.

Ce fichier est exploité par le composant Post-Pro de Salomé pour visualisation des partitions et niveaux de résolution (cf. 4.2.2).

L'utilisateur fournit :

- ✓ Après avoir sélectionné par clic-droit l'élément réduit dans l'Object Browser de Salomé, une boite de dialogue de type « File Browser » apparaitra pour assister l'utilisateur dans le choix du chemin et du nom du fichier.

On obtient :

- ✓ Le fichier d'aiguillage.

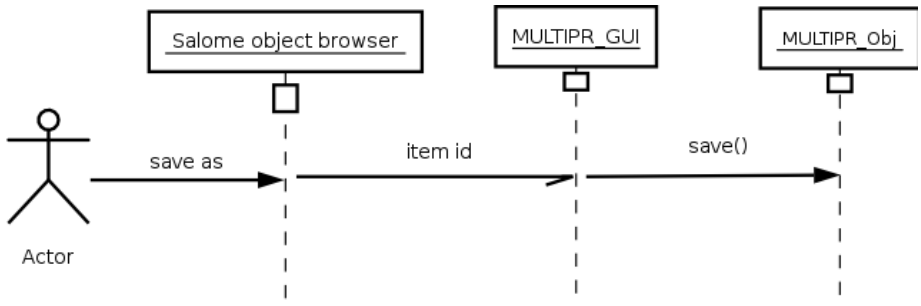


Figure 7 : Diagramme de séquence : sauvegarde fichier d'aiguillage

4.2.2 Post-Pro : VISU

On offre la facilité à l'utilisateur d'envoyer les résulats de sa réduction vers le composant VISU par manipulation simple de l'Object Browser de Salomé.

L'utilisateur fournit :

- ✓ L'élément dans l'Object Browser de Salomé à exporter par clic-droit.

On obtient :

- ✓ Le chargement par le composant VISU de l'élément réduit.

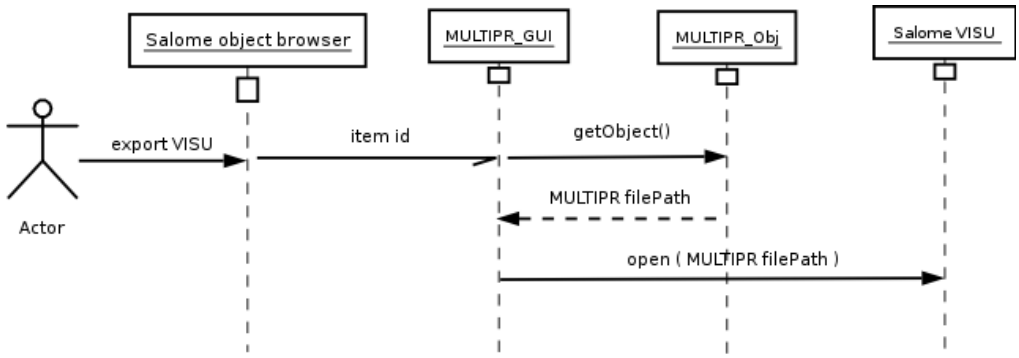


Figure 8 : Diagramme de séquence : export VISU

5. Interface serveur

L'interface serveur du composant MULTIPR fournit un ensemble de services décrits dans les paragraphes suivants.

On trouvera une description informatique dans le fichier `MULTIPR_Gen.idl`.

Ces services sont exploités par l'interface cliente du composant mais sont également accessibles par tout client CORBA .

Chapitre 5.

5.1 Description générale

La partie *engine* (MULTIPR_Gen) du composant MULTIPR est un objet Corba implémenté en C++ et qui réside dans le processus serveur *Salome_Container*. Son cycle de vie est géré par le container *FactoryServer*.

Il est enregistré dans l'annuaire (Naming Service) de Salomé à sa création :

```
ContainerManager.object
Containers.dir
  clp51am.dir
    FactoryServerPy.object
    FactoryServer.object
    FactoryServer.dir
      GEOM_inst_1.object
      SMESH_inst_2.object
      MULTIPR_inst_2.object
Registry.object
Kernel.dir
  ModulCatalog.object
  Session.object
Study.dir
  Study1.object
myStudyManager.object
SalomeAppEngine.object
myGEOM_Gen.object
```

Figure 9 : Salomé Naming Service

Note : un client CORBA (script python par exemple) peut donc en dehors du contexte Salomé accéder aux services du composant MULTIPR en consultant l'annuaire de Salomé.

5.2 Générateur : MULTIPR_Gen

Il représente la partie « Engine » du composant. Il n'existe qu'une seule instance par session Salomé.

Son rôle est de créer des objets « fonctions » MULTIPR_Obj.

5.2.1 getObject

Retourne un objet de type MULTIPR_Obj.

5.2.2 Autres fonctions haut niveau

MULTIPR_Gen propose également 3 fonctions directes (haut niveau) pour réaliser un partionnement/décimation d'un fichier MED (voir exemple de scénarios d'utilisation, ci-après) :

1. partitionneDomaine
2. partitionneGrain
3. decimePartition

5.3 Services de multipartition : MULTIPR_Obj

Un objet de ce type fournit l'ensemble des opérations utiles pour réaliser une partition/décimation.

Chaque objet gère un seul fichier MED à réduire à la fois : pour pouvoir réduire un maillage provenant d'un autre fichier MED, il faudra demander à l'engine un autre objet.

Les fonctions implémentées dans un objet de type MULTIPR_Obj sont les suivantes :

- Les deux fonctions principales du partitionneur, spécifiées dans [DR1] :
  - **partitionneDomaine()**
  - **partitionneGrain()**
- La fonction principale du décimateur, spécifiée dans [DR1] :
  - **decimePartition()**
- Les fonctions annexes suivantes :
  - **setMesh()** : permet d'associer un maillage à l'objet dans le cadre du scénario de partitionnement à 1 échelle (partitionneDomaine)
  - **setBoxing()** : (optionnel) permet de changer le paramètre de boxing pour la décimation, fixé à 100 par défaut (cf. decimePartition)
  - **getMeshes()** : retourne la liste des noms des maillages contenus dans le fichier MED séquentiel associé à l'objet ; cette fonction peut aider l'utilisateur pour configurer partitionneDomaine()
  - **getFieldNames()** : retourne la liste des noms des champs contenus dans le fichier MED associé à l'objet ; cette fonction peut aider l'utilisateur pour configurer decimePartition()
  - **getTimeStamps()** : pour un champ donné, retourne le nombre d'itérations (pas de temps) associé à ce champ ; cette fonction peut aider l'utilisateur pour configurer decimePartition()
  - **getParts** : retourne le nom de toutes les partitions associées à cet objet ; cette fonction n'est disponible qu'une fois le partitionnement effectué
  - **save** : enregistre sur disque un fichier MED distribué, contenant le résultat des différents algorithmes appliqués

Remarque : la fonction open, n'est pas nécessaire, car l'ouverture d'un fichier est inclus dans la création de l'objet.

5.4 Exemples de scénarios d'utilisation

Les exemples suivant se déroulent dans l'un des interprétateurs Python embarqué de Salomé.

5.4.1 Une échelle

```
import LifeCycleCORBA
lcc = LifeCycleCORBA.LifeCycleCORBA(clt.orb)
import MULTIPR_ORB

# récupération des fonctions de partition/ décimation
engine      = lcc.FindOrLoadComponent("FactoryServer", "MULTIPR")
op          = engine.getObject("agregat100grains_12pas.med")
```

```
# récupération de la liste des champs
liste_champs = op.getFields()

# sélection maillage
liste_maillages = op.getMeshes()
op.setMesh( liste_maillages[0] )

# partition en grains
liste_grains = op.partitionneDomaine()

# décimation
nomMethodeDecimation = 'Filtre_GradientMoyen'
nomChamp = liste_champs[2]
iterationChamp = 12
seuilMedium = 10.0     # extrait les points dont la norme du gradient est >= 10.0
seuilBas = 25.0        # extrait les points dont la norme du gradient est >= 25.0
rayonVoisinage = 0.3

# optionnel : op.setBoxing(10)   # le parametre de boxing est fixe à 100 par défaut

for grain in liste_grains[0:4]: # on décime les 5 premiers grains
    op.decimePartition(
       grain,
       nomMethodeDecimation,
       nomChamp,
       iterationChamp,
       seuilMedium,
       seuilBas,
       rayonVoisinage )

op.getParts()

# sauvegarde du fichier d'aiguillage (fichier MED distribué)
op.save()
```

5.4.2 Deux échelles

```
import LifeCycleCORBA
lcc = LifeCycleCORBA.LifeCycleCORBA(clt.orb)
import MULTIPR_ORB

# récupération des fonctions de partition/ décimation
engine      = lcc.FindOrLoadComponent("FactoryServer", "MULTIPR")
op          = engine.getObject("agregat100grains_12pas.med")

# récupération de la liste des champs
liste_champs = op.getFields()

# sélection maillage
liste_maillages = op.getMeshes()
op.setMesh( liste_maillages[0] )

# partition en grains
liste_grains = op.partitionneDomaine()

# partition d'un grain en 16
```

```
liste_pgrain = op.partitionGrain(liste_grains[4], 16 , 0) // 0=METIS ; 1=SCOTCH

# décimation
nomMethodeDecimation = 'Filtre_GradientMoyen'
nomChamp = liste_champs[2]
iterationChamp = 12
seuilMedium = 10.0     # extrait les points dont la norme du gradient est >= 10.0
seuilBas = 25.0        # extrait les points dont la norme du gradient est >= 25.0
rayonVoisinage = 0.5

# optionnel : op.setBoxing(10)   # le parametre de boxing est fixe à 100 par défaut

for grain in liste_pgrain[0:11]: # on décime les 5 premieres partition du grain
    op.decimePartition(
        grain,
        nomMethodeDecimation,
        nomChamp,
        iterationChamp,
       seuilMedium,
        seuilBas,
       rayonVoisinage )

# sauvegarde du fichier d'aiguillage
op.save()
```

5.4.3 Fonctions haut niveau

```
import LifeCycleCORBA
lcc = LifeCycleCORBA.LifeCycleCORBA(clt.orb)
import MULTIPR_ORB

# récupération des fonctions de partition/ décimation
engine = lcc.FindOrLoadComponent("FactoryServer", "MULTIPR")

# isole les grains du maillage
engine.partitionneDomaine("agregat100grains_12pas.med", "MAIL")

# partitionne le 98e grain en 3 partie en utilisant METIS (METIS=0 ; SCOTCH=1)
engine.partitionneGrain("agregat100grains_12pas.med", "MAIL_98", 3, 0)

# décime le 97e grain
engine.decimePartition(
       "agregat100grains_12pas.med",
       "MAIL_97"               # nom de la partie à décimer = le grain 97
       "",
       12,                     # itération du champ
       "Filtre_GradientMoyen", # méthode de décimation
       10.0,                   # seuil medium
       25.0,                   # seuil bas
       0.5,                    # rayon de voisinage
       10)                     # parametre de boxing
```

6. Spécification détaillée

6.1 Arbre d'étude Salomé

Ce chapitre spécifie la sémantique et les valeurs des éléments de l'arbre d'étude Salomé manipulés par le composant.

6.1.1 Attributs

Listes

Nom	Description
`AttributeName`	Nom de l'objet. Visible dans l'arbre d'étude (section Object).
`AttributeFileType`	(cf. 6.1.2.2)
`AttributeExternalFileDef`	(cf. 6.1.2.2)
`AttributeComment`	Chaîne de caractère à vocation diverse. Visible dans l'arbre d'étude (section Value).
`AttributePixMap`	Chaîne de caractère désignant l'icône de l'objet .

Figure 10 : les principaux attributs manipulés par le composant

Remarque : la liste n'est pas exhaustive.

Typage: attributs AttributeFileType et AttributeExternalFileDef

Les valeurs de l'attribut « AttributeFileType » permettent de typer un objet de l'arbre d'étude.

L'attribut « AttributeExternalFileDef » vient souvent s'y ajouter pour stocker la valeur.

Pour chacun de ces couples (type, valeur) on peut ainsi engager des opérations spécifiques (autoriser un clic-droit par exemple). Le tableau ci-dessous donne la liste des objets typés dans l'arbre d'étude.

AttributeFileType	Désigne
`MULTIPR_FILE`	objet multipartition d'un fichier

(*) ne figure pas actuellement dans l'arbre d'étude.

Icônes: attribut AttributePixMap et fichier MULTIPR_icons.po

Il est possible d'associer un icône à un objet de l'arbre d'étude. Celui-ci doit posséder l'attribut *AttributePixMap* de valeur son nom figurant dans le fichier `MULTIPR_icons.po`.

Par exemple :

```
....
msgid "ICON_MULTIPR"
msgstr "Multipr.png"
....
```

Figure 11 : fichier `MULTIPR_icons.po`

6.1.2 Objet Multipartition

Pour le moment la représentation de l'objet dans l'arbre d'étude comportera les attributs suivants :

Nom dans ObjectBrowser	AttributeFileType	AttributeExternalFileDef	AttributeComment	Tag
nom fichier MED d'entrée contenant le maillage à réduire	MULTIPR_FILE	chemin complet fichier MED d'entrée	chemin complet fichier MED d'entrée	

6.2 Gestion des erreurs

6.2.1 Interface Cliente

Les comportements attendus et inattendus mais non gérés par le composant sont caractérisés par des levées d'*exceptions*. La politique est la capture de ces *exceptions* et la notification de celles-ci par boîte de dialogue à l'utilisateur Salomé.

6.2.2 Interface Serveur

Lorsqu'on invoque une fonction du moteur du composant MULTIPR, il est possible que celle-ci lève une exception en cas de dysfonctionnement. L'exception est alors typée *SALOME::SALOME_Exception* (cf. *SALOME_Exception.idl*, *MULTIPR_Gen.idl*).

6.3 Superviseur : `MultiprCatalog.xml`

Ce catalogue est utilisé en premier lieu par le composant superviseur de *Salomé*.

```
<component-service>
    <service-name>Get_Object</service-name>
    <service-author>CS</service-author>
    <service-version>1</service-version>
    <service-comment></service-comment>
    <inParameter-list>
        <inParameter>
            <!-- .med file name including full absolute path -->
            <inParameter-name>theMedFileName</inParameter-name>
            <inParameter-type>string</inParameter-type>
            <inParameter-comment>filename must include full path</inParameter-comment>
        </inParameter>
    </inParameter-list>
    <outParameter-list>
        <outParameter>
            <outParameter-name>return</outParameter-name>
            <outParameter-type>MULTIPR_Obj</outParameter-type>
            <outParameter-comment></outParameter-comment>
        </outParameter>
    </outParameter-list>
</component-service>
```

Figure 12 : extrait du catalogue MULTIPR

Chapitre 6.

Dans ce fichier on définit les services qu'offre le moteur du composant MULTIPR dans l'environnement *Salomé* : il doit donc refléter une sous-partie de l'interface IDL du composant (`MULTIPR_Gen.idl`). Il est écrit en XML et suit une grammaire bien précise : une balise pour déclarer un service `<component-service>`, les paramètres d'entrée `<inParameter>`, de sortie `<outParameter>`, etc.

6.4 Fichier d'aiguillage

A l'origine le module MULTIPR accepte uniquement en entrée un fichier **MED séquentiel** (fichier MED classique). Dès lors que l'utilisateur souhaite enregistrer sur disque le résultat d'un partitionnement ou d'une décimation, MULTIPR engendre un fichier **MED distribué.**

Un fichier MED distribué n'est rien d'autre qu'une série de fichiers MED séquentiels accompagné d'un fichier d'aiguillage (ou fichier maître) au format ASCII et compatible avec l'outil MEDSPLITTER et les spécifications de MED-parallèle [DR4].

6.4.1 Nommage des fichiers engendrés par le partitionneur

Soit « XXX.med » le nom du fichier MED séquentiel original.

Partitionnement du domaine maillé en grain (scénario à une échelle) :

Le fichier qui contient le I^{ème} grain est un fichier MED séquentiel. Il est nommé :

XXX_grainI.med

Partitionnement d'un grain (scénario à deux échelles) :

Le fichier qui contient la J^{ème} partition du I^{ème} grain est nommé :

XXX_grainI_partJ.med

Décimation d'une partition :

Soient YYY le nom de la méthode de décimation et PPP la chaîne de caractère qui représente les paramètres utilisés pour la décimation.

Suivant le scénario (1 ou 2 échelles), les fichiers issus de la décimation portent respectivement les noms :

XXX_grainI_YYY-PPP.med

XXX_grainI_partJ_YYY-PPP.med

Exemple :

L'utilisation du filtre « gradient moyen » pour la décimation de la J^{ème} partition du I^{ème} grain du fichier XXX.med, configurée avec le seuil bas = 2.0 et le seuil medium = 1.0 et un rayon de voisinage égal à 3.0, engendre les deux fichiers suivants :

XXX_grainI_partJ_gradmoy-low-2.0-3.0.med

XXX_grainI_partJ_gradmoy-med-1.0-3.0.med

6.4.2 Spécifications du fichier d'aiguillage

Soit « XXX.med » le nom du fichier original ; soient N le nombre de groupes (i.e. grains) présents dans le fichier et { K_1, ... K_N }, le nombre de sous-partitions de chacun des grains.

Le fichier d'aiguillage est un fichier ASCII qui assure le lien entre les différents fichiers MED issus de la phase de partitionnement. Il adresse ΣK_g fichiers, où g varie entre 1 et N. Tous les fichiers MED produits par le partitionneur sont au format MED-Fichier v2.3 [DR5].

Le nom du fichier d'aiguillage est obtenu en concaténant le suffixe « `_grains_maitre` » au nom du fichier sans l'extension : XXX`_grains_maitre.med`. Sa structure est la suivante :

```
# MED Fichier v2.3 - Master file created by MULTIPR
#
# Nombre de grains
N
Nom_maillage_source   N°  Nom_sous_maillage   localhost   Nom_fichier_MED_associé
...
```

Exemple avec le fichier agregat100grains_12pas.med :

- Ce fichier contient 100 groupes et un unique maillage, MAIL (on ne tient pas compte du groupe qui englobe tous les autres groupes et correspond donc au maillage d'origine)
- On suppose que le 99e groupe a été découpé en 3 parties
- On suppose que le 100e groupe a été décimé

```
# MED Fichier v2.3 - Master file created by MULTIPR
#
104
MAIL 1   MAL_1        localhost agregat100grains_12pas_grain1.med
MAIL 2   MAL_2        localhost agregat100grains_12pas_grain2.med
...
MAIL 98  MAL_98       localhost agregat100grains_12pas_grain98.med
MAIL 99  MAL_99_1     localhost agregat100grains_12pas_grain99_part1.med
MAIL 100 MAL_99_2     localhost agregat100grains_12pas_grain99_part2.med
MAIL 101 MAL_99_3     localhost agregat100grains_12pas_grain99_part3.med
MAIL 102 MAIL_100     localhost agregat100grains_12pas_grain100.med
MAIL 103 MAIL_100_MED localhost agregat100grains_12pas_grain100_gradmoy-med-1.0-1.0.med
MAIL 104 MAIL_100_LOW localhost agregat100grains_12pas_grain100_gradmoy-low-2.0-1.0.med
```

Commentaires :

- Ce format est le même que celui employé par MEDSPLITTER, ce qui rend les deux outils compatibles
- Le symbole # introduit un commentaire
- Les sous-fichiers (i.e. les partitions) sont tous placés dans le même répertoire que le fichier source
- Les 3 résolutions (PLEINE, MEDIUM et BASSE) issues de la décimation du 100e groupe correspondent aux entrées n° 102, 103 et 104 du fichier maître.

6.5 Visualisation des données produites par MULTIPR dans VISU

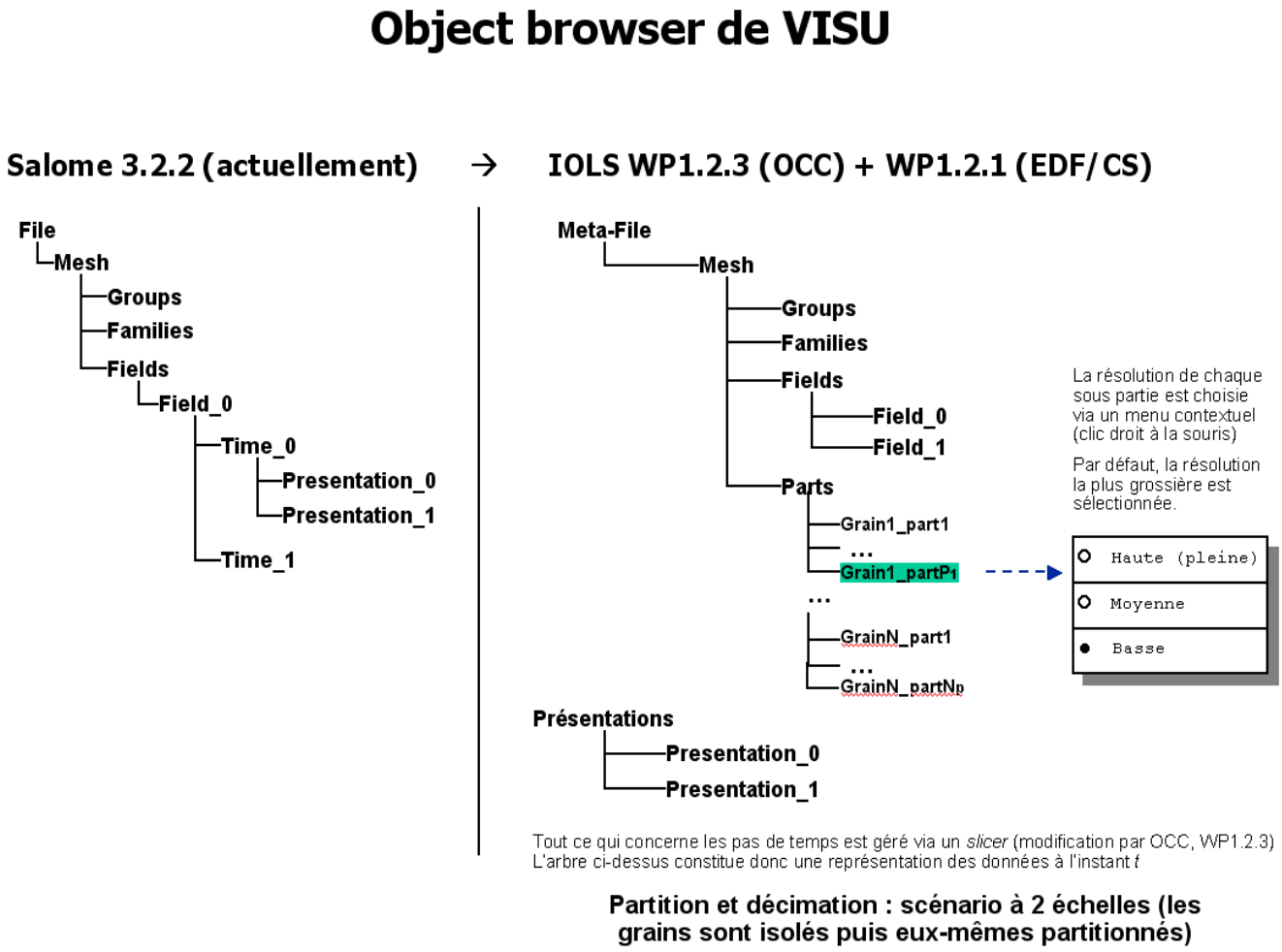
La figure ci-dessous montre, dans sa partie de gauche, la représentation actuelle de l'object browser (SALOME v3.2.2) et, dans sa partie de droite, les modifications envisagées pour l'intégration du composant MULTIPR.

Les modifications apportées à l'*object browser* intègrent les modifications récemment réalisées par OpenCascade (IOLS WP1.2.3) et disponible dans la branche **`WP1_2_3_17-01-2007_Data_format_at_entry_of_visualization_pipeline`** de la base VISU (voir section 8).

D'un point de vue visuel, le chargement d'un fichier partitionné engendre la création de la branche **Parts.** Cette branche donne à l'utilisateur l'accès à toutes les partitions tirées du fichier maître généré par le partitionneur.

L'accès aux trois résolutions calculées par le décimateur se fait via un menu contextuel qui apparaît en cliquant avec le bouton droit de la souris sur une des partitions. Par défaut, à l'ouverture du fichier, toutes les partitions sont affichées à la résolution la plus grossière. Dans cette version du module MULTIPR, ce menu contextuel offre la seule façon de changer la résolution d'une partition.

La sélection d'un pas de temps s'opère via un *slider* en bas la fenêtre. Toutes les informations affichées dans l'object browser se réfèrent donc au pas de temps courant.



Partition et décimation : scénario à 2 échelles (les grains sont isolés puis eux-mêmes partitionnés)

7. Conception des fonctions de partitionnement et de décimation

Cette partie présente la conception des principales fonctions du module MULTIPR, établie à partir de [DR1].

La solution que nous proposons, basée sur des interfaces et des hiérarchies de classes, satisfait aux exigences de généricité et permet d'envisager assez simplement de nombreuses évolutions.

Tous les fichiers sources (*.hxx, *.cxx) du module MULTIPR sont préfixés MULTIPR_.

L'interface principale (haut niveau) de notre solution est **MULTIPR_API.hxx.** Il regoupe la déclaration des principales fonctions haut niveau du module, conformes au cahier des charges.

Compte tenu des outils disponibles pour l'étape de partionnement (MED fichier, MEDSPLITTER), les principales difficultés de conception sont concentrées sur la partie décimation (que nous détaillons donc plus).

Concernant les maillages, seules les mailles de type TETRA10 seront exploitables dans cette première version.
Chaque élément de type TETRA10 est composé de 10 nœuds (les 4 sommets du tétraèdre et les 6 milieux des arêtes) et de 5 points de Gauss.

7.1 Gestion des erreurs

La gestion des erreurs repose sur le mécanisme C++ des exceptions qui offre une grande souplesse et permet un traitement simple et relativement exhaustif des problèmes.

Toutes les erreurs à l'exécution du module MULTIPR engendre une **RuntimeException** qui contient les informations suivantes :

- Message d'erreur (chaîne de caractères)
- Nom du fichier dans lequel a été levé l'exception (chaîne de caractères)
- Numéro de la ligne de code qui a levé l'exception.

Afin de différencier les causes d'erreur, plusieurs classes dérivent de RuntimeException :

- **NullArgumentException** : pour signaler un pointeur NULL malvenu
- **IllegalArgumentException** : pour signaler une valeur erronée d'un paramètre
- **IndexOutOfBoundsException** : pour signaler un index hors de sa plage de validité
- **IllegalStateException** : pour signaler qu'un objet est dans un état erroné
- **IOException** : pour signaler un problème d'entrée/sortie
- **UnsupportedOperationException** : pour signaler qu'une opération est invalide ou non supportée

7.2 Fonction de partitionnement du domaine (Partitionne_Domaine)

Description :

Cette fonction découpe un domaine maillé issu d'un fichier MED séquentiel en autant de partitions que de groupes présents dans les données d'origine. Elle constitue l'élément central du scénario de partition à « une échelle ».

Signature :

```
void partitionneDomaine(
      const char* medFilename,
      const char* meshName);
```

En sortie :

- La fonction engendre un fichier MED distribué. Elle génère autant de partitions que de groupes dans le maillage dont le nom est `meshName`. Chaque sous maillage est enregistré sur disque au format MED, dans un fichier séparé. L'ensemble des fichiers est référencé dans un fichier maître (ou « fichier d'aiguillage »), également au format ASCII et compatible avec MEDSPLITTER.
- En cas d'erreur, une RuntimeException est levée (voir gestion des erreurs).

Commentaires :

- Le travail consiste à utiliser la bibliothèque MED fichier v2.3 pour lire un fichier et isoler les groupes.
- Comme les grains du domaine maillé ne sont pas connexes, le recours aux objets de type joint (MED-Fichier v2.3 [DR4, DR5]) n'est pas nécessaire pour le scénario à une échelle.
- Afin de limiter le volume des fichiers en sortie, les champs du maillage d'origine sont projetés sur chacun des goupes (la notion de profil n'est pas utilisée). Cette approche est la même que celle retenue pour le développement de MEDSPLITTER.

7.3 Fonction de partitionnement d'un grain (Partitionne_Grain)

Description :

Cette fonction opère sur un fichier MED distribué. Elle partitionne un grain selon le nombre de fragments spécifiés par l'utilisateur. Cette fonction intervient en deuxième passe du scénario de partition à « deux échelle ».

Signature :

```
void partitionneGrain(
      const char* medFilename,
      const char* partName,  // nom du sous-maillage a partitionner (3e colonne fichier maitre)
      int nbParts,           // nombre de fragments à générer
      int partitionner);     // partitionneur à utiliser (METIS ou SCOTCH)
```

En sortie :

- La fonction engendre un nouveau fichier MED distribué. Elle génère `nbPart` partitions du grain spécifié. Chaque sous maillage est enregistré sur disque au format MED v2.3, dans un fichier séparé. L'ensemble des fichiers est référencé dans un fichier maître (ou « fichier d'aiguillage »).
- En cas d'erreur, une RuntimeException est levée (voir gestion des erreurs).

Commentaires :

- Le travail consiste principalement à utiliser MEDSPLITTER [DR3] qui contient toutes les fonctions nécessaire à l'accomplissement de la tâche. En particulier, MEDSPLITTER encapsule les outils METIS et SCOTCH préconisés pour cette opération de partitionnement et produit des fichiers au format MED v2.3 en sortie. En outre, précisons que MEDSPLITTER réalise les joints nécessaires entres les différentes partitions (les objets de type joint sont une des nouveautés de MED v2.3).

7.4 Fonction de filtrage suivant le gradient de l'observable (Filtre_GradientMoyen)

Description :

Cette fonction assure la décimation des données et sert à produire les maillages à différentes résolutions. Elle est utilisée par Multi-Res_Partition.

Signature :

```
Mesh* apply(                          // retourne le maillage décimé
       Mesh* mesh,                    // maillage d'origine (la source)
       const char* argv,              // paramètre de la décimation sous forme générique
       const char* nameNewMesh)       // nom du maillage décimé
```

Commentaires :

- D'un point de vue conceptuel, cette fonction est implémentée dans la classe `DecimationFilterGradAvg` qui dérive de l'interface `DecimationFilter`. Techniquement DecimationFilter est une classe virtuelle pure qui ne contient que la signature de la méthode `apply`. Elle sert uniquement à abstraire la technique de filtrage afin de faciliter de futures évolutions.

- L'utilisateur peut choisir la dimension des cellules utilisées pour le boxing en définissant le nombre de cellules suivant un des axes de la boîte englobante. Par défaut, cette valeur est fixée à 100 (suivant chacune des 3 dimensions de l'espace), ce qui produit une grille composée de 1.000.000 de cellules.

- L'utilisateur a la possibilité de fixer une valeur quelconque pour le rayon de la sphère de proximité. Lorsque qu'aucune valeur n'est passée en paramètre, la valeur par défaut (0.0) est utilisée. Cette valeur engendre le calcul automatique d'une valeur « raisonnable » pour ce paramètre.

  Plus précisément, le rayon de la sphère est évalué de façon à ce que le nombre moyen de voisins d'un point soit égal à m=8 ; le calcul est réalisé en supposant une répartition uniforme des N points du champ dans k=80% du volume V de la boîte englobante du champ.

Soit :

$$(m.k.V) / N = 4/3.\pi.radius^3$$

D'où, la valeur par défaut du rayon de la sphère :

$$\mathbf{radius = (3.m.k.V /4.\pi.N)^{1/3}}$$

- Sans structure accélératrice, la complexité algorithmique temporelle de ce filtrage est quadratique en fonction du nombre de points dans le champ scalaire initial : pour chaque point du champ, la recherche des points du voisinage nécessite de tester tous les autres points.

  Afin de réduire le temps de calcul, la technique de boxing a donc été retenue. Elle consiste à plonger le champ scalaire dans une grille régulière afin d'accélérer la recherche des points voisins.

  Bien que non optimale, cette technique permet de considérablement réduire le temps nécessaire pour identifier les points voisins tout en conservant une certaine simplicité de l'algorithmique. De futures évolutions pourront considérer l'utilisation d'un kd-tree, plus performant mais sensiblement plus complexe à mettre en oeuvre.

  Dans ce cadre, nous proposons donc le recours à une structure accélératrice générique, dont la conception est la suivante :

```
class DecimationAccel // interface
{
public:

        // configure la structure accélératrice
        // une chaîne de caractère est utilisée pour la généricité
        virtual void configure(const char* argv) = 0;

        // construit une structure accélératrice à partir d'un champ scalaire
        virtual void create(const std::vector<Points>) = 0;

        // retourne la liste des voisins du point (x,y,z) inclus dans une sphère
        // de centre (x,y,z) et de rayon r
        virtual std::vector<Point> findNeighbours(real x, y, z, r) = 0;
}
```

Le boxing apparaît alors comme une spécialisation de ce shéma :

```
class DecimationAccelGrid  : public DecimationAccel // boxing
{
public:
        ...
private:
        // grille régulière 3D
        // chaque cellule de la grille indexe les points qui sont contenus
        // dans cette cellule
        vector<Point> *grid;   // <=> grid[NX*NY*NZ] <=> grid[NX][NY][NZ]
        int size_x;            // nombre de cellules suivant la dimension X
        int size_y;            // nombre de cellules suivant la dimension Y
        int size_z;            // nombre de cellules suivant la dimension Z
}
```

Accès à la cellule d'indices (x, y, z) :

```
int getCellIndex(int x, int y, int z) const
{
        return x + size_x * (y + z * size_y);
}
```

Pseudo-code :

Construction de la structure accélératrice (grille régulière) :

```
Déterminer la boite englobante b (alignée sur les axes) du champ scalaire
           (une simple boucle => O(n) ou n est le nombre de points dans le champ)

Pour chaque point p du champ scalaire
        c <- Identifier la cellule de la grille contenant p
        // L'identification de la cellule est réalisé de la façon suivante :
        //      int x = int((p.x - b.xmin) / b.dimx)
        //      seuiller x pour régler les éventuels problèmes d'arrondi
        //      idem pour y et z
        // Remarque : chaque point p est référencé dans une cellule unique
        Ajouter une référence à p dans c
```

Recherche des voisins de p inclus dans une sphère centrée en p et de rayon r :

```
Déterminer la boîte englobante b de la sphère s centrée en p et de rayon r :
           b_min = (p.x-r, p.y-r, p.z-r) ; b_max = (p.x+r, p.y+r, p.z+r)

Plonger cette boîte b dans la grille régulière afin de déterminer les cellules
intersectées (trivial : quelques additions et divisions)

// soit L la liste des voisins recherchés
L <- vide
Pour chaque cellule c de la grille régulière intersectée par b
// ces cellules sont déterminées par une triple boucle sur (X, Y, Z)

       Pour chaque point p' référencé dans c
              Si p' est contenu dans s (<=> norme du vecteur pp' < r)
                     Alors Ajouter p' à L
Retourner L
```

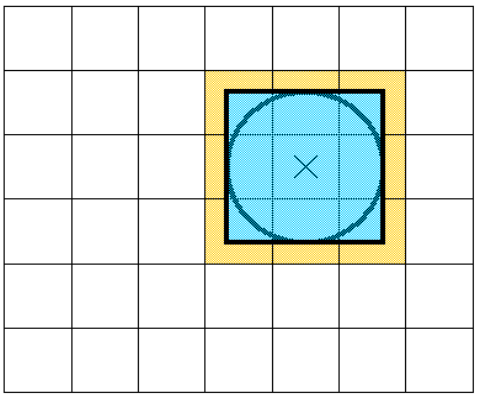


Illustration 1: grille régulière (boxing)

L'illustration 1 présente une vue bidimensionnelle de la structure spatiale utilisée pour accélérer la recherche des proches voisins. Etant donné un point p du champ, on recherche tous les voisins compris dans une sphère centrée en p et de rayon r. Grâce à la grille, on peut se contenter de tester les points qui sont dans des cellules intersectées par la sphère (9 cellules sur le schéma).

Filtrage par gradient moyen :

```
Champ C <- Récupérer le champ scalaire à partir de (nomChamp, iterationChamp)
DecimationAccel* accel <- new DecimationAccelGrid()
accel->create(C)
res <- liste de maille, initialement vide
Pour chaque maille M du maillage d'origine // M est supposée être une maille TETRA10
        bool conserveMaille = false;
        Pour chaque point de Gauss p de M
                ListePoint L <- g->findNeighbours(p, radius)
                Vecteur3D gradient = (0, 0, 0)
                Pour chaque point p' de L
                        Vecteur3D vecPP' = Vecteur3D(p,p')
                        real q = (valeurObservable(p') - valeurObservable(p))
                        q /= vecPP'.getNorm()
                        gradient +=  vecPP' * q
                gradient /= |L|
                real normeGrad = gradient.getNorm()
                Si normeGrad >= seuil
                        conserveMaille = true;
                        // optimisation : 1 pt valide => toute la maille est valide
                        Quitter la boucle (break)
        Si conserveMaille
                Ajouter M dans res

Supprimer accel
Retourner res (= la liste des mailles non filtrés)
...
Res est ensuite écrit sur disque au format MED, via l'API MEDMEM.
```

7.5 Fonction de création de la multi-résolution d'une partition (Multi-Res_Partition)

Description :

Cette fonction opère sur une partition et produit deux nouvelles résolutions, plus grossières, de cette partition (médium et basse) en procédant à la décimation des données.

Techniquement, la décimation des données repose sur un filtre. Dans cette première version du module MULTIPR, un seul filtre est disponible : Filtre_GradientMoyen, cf. [DR1].

Signature :

```
void decimePartition(
        const char* medFilename,  // nom du fichier maître (MED distribué)
        const char* partName,     // nom de la partition à décimer
        const char* fieldName,    // nom du champ de référence (utiliser pour la décimation)
        int fieldIt,              // numéro d'itération du champ à utiliser (pas de temps)
        const char* filterName,   // nom du filtre pour la décimation
        real thresholdMedRes,     // seuil du gradient pour la résolution moyenne
        real thresholdLowRes,     // seuil du gradient pour la résolution basse
        real radius = 0.0,
        int boxing = 100);
```

En sortie :

- La fonction met a jour le fichier MED distribué et crée deux nouveaux fichiers MED qui contiennent les deux résolution MEDIUM et BASSE de la partittion considérée.
- En cas d'erreur, une RuntimeException est levée (voir gestion des erreurs).

Pré-condition :

- $0 \leq$ `thresholdMedRes` $<$ `thresholdLowRes` (la fonction vérifie cette condition avant de commencer les traitements).
- `radius > 0`
- `2 < boxing` ≤ 200

Commentaires :

- Le type `real` correspond au type MED med_float (double).
- Afin de préserver la généricité de l'opération, la fonction de décimation est passée en paramètre sous la forme d'une chaîne de caractères. L'objet qui accomplit le filtrage est construit dynamiquement (*pattern factory*) à partir de son nom.
- La décimation correspond à l'extraction des points du maillage d'origine pour lesquels la variation du champ est la plus importante. La variation du champ est établie en calculant la norme du gradient en chaque point. Plus cette norme est importante, plus la variation du champ est forte en ce point. La résolution BASSE (qui comprend le moins de points) correspond aux zones de plus forte variation du champ.

Pseudo-code :

```
Vérifier l'existance de la partition à partir de son nom
Vérifier la validité des paramètres numériques
Si erreur dans au moins un paramètre => Retourner false
// création du filtre de décimation (factory)
DecimationFilter* filtre <- DecimationFilter::create(decimationMethodName)
Si filtre erroné Alors Lever une exception
// décimation : calcul de la résolution moyenne
Mesh* meshMedium = filtre->apply(
        currentMesh, paramMed, currentMesh->getName() + « _MED »);
// décimation : calcul de la résolution basse
Mesh* meshLow = filtre->apply(
        currentMesh, paramLow, currentMesh->getName() + « _LOW »);
Supprimer filtre
```

Fabrication du filtre à partir de son nom (*factory*) :

```
Filtre* filtre instancierFiltre(decimationMethodName)

        Si (decimationMethodName == « Filtre_GradientMoyen »)
                Retourner new FiltreGradientMoyen()
        // futures évolutions :
        // sinon si (decimationMethodName == XXX) ...
        Sinon Lever une exception
```

8. Annexe

8.1 Base CVS

La gestion de configurations est assurée par l'outil CVS.

8.1.1 Administration

L'administration des bases est assurée par OpenCascade et gérée par Mikhail KAZAKOV : `mikhail.kazakov@opencascade.com`.

8.1.2 Server

Le serveur CVS est hébergé à l'adresse : `cvs.opencascade.com/home/server/cvs`

8.1.3 Bases

La base CVS utilisée dans le cadre du développement du Partitionneur-Décimateur est : `MULTIPR`.
CS dispose également un accès en lecture/écriture aux bases `VISU` et `MED`.

L'accès est réalisé via le protocole de connexion `pserver`.

Exemple, pour l'accès à la base MED :

```
> export CVSROOT=:pserver:cs@cvs.opencascade.com:/home/server/cvs/MED
> cvs login
> cvs checkout -r WP1_2_3_17-01 2007_Data_format_at_entry_of_visualization_pipeline MED_SRC
> cvs logout

ou encore

> cvs -d":pserver:salome2anonymous@www.opencascade.com:/home/server/cvs/MED" login
> rntlrntl   (<-- mot de passe)
> cvs -d":pserver:salome2anonymous@www.opencascade.com:/home/server/cvs/MED" checkout -kk
-RP -r  WP1_2_3_17-01 2007_Data_format_at_entry_of_visualization_pipeline
> cvs -d":pserver:salome2anonymous@www.opencascade.com:/home/server/cvs/MED" logout
```

8.1.4 Branches / balises

Les versions de MED et VISU utilisées sont balisées
`WP1_2_3_17-01-2007_Data_format_at_entry_of_visualization_pipeline`

La version incluse de MED inclue le composant MEDSPLITTER.