COMMUNICATION & SYSTEMES

Amélioration du post-traitement dans Trio_U
Lot 1 : Formats de postraitement
Lot 2 : Modification du framework Trio_U
Lot 3 : Framework de postraitement

Rapport de conception et réalisation

N° Identification : CS SI/311-1/AB06B028/NRD/1.1
Date : 15/10/2007

Rédigé par : M. Elmo	Le : 15/10/07	Signature :
Vérificateur technique : P. Ledac	Le : 15/10/07	Signature :
Vérificateur qualité : C. Pernot	Le : 15/10/07	Signature :
Approuvé par : C. Coutelou	Le : 15/10/07	Signature :

LISTE DE DIFFUSION

Diffusion Externe

Nom des destinataires	Adresse	Nb. Ex.
B. MATHIEU	CEA / DEN / DTP / SSTH / LMDL Av. des Martyrs 38054 GRENOBLE CEDEX	1

Diffusion Interne

Nom des destinataires	Adresse	Nb. Ex.
M. ELMO	CS Fontaine	1

SUIVI DES MODIFICATIONS

Version / Révision		Références		Description de la modification	Nom de l'auteur
Indice	*Date*	*N°page*	*N°§*		
1.0	19/04/07			Version initiale	M. Elmo
1.1	15/10/07			Ajout § 3 (Framework de post-traitement)	M. Elmo

SOMMAIRE

1. LOT 1 : FORMATS DE POSTRAITEMENT

Le logiciel Trio_U supporte divers formats pour le post-traitement de valeurs d'un champ discret. Les formats supportés sont : lml, meshtv, med et lata V1. La gestion de ces formats présente des défauts. D'une part, les sources mis en jeu pour une opération de post-traitement sont répartis dans de nombreuses classes non dédiées spécifiquement à cette fonction. D'autre part, les méthodes utilisées ne sont pas homogénéisées au travers d'une interface. Ces aspects rendent difficiles la compréhension et l'exploitation par un développeur des mécanismes d'écriture de données.
Dans le cadre de ce lot une révision des sources est effectuée et en particulier un objet format de post-traitement est mis en œuvre avec deux objectifs majeurs :
-Les instructions d'écriture de données actuellement dispersées dans des classes non dédiées au post-traitement sont déplacées dans une nouvelle hiérarchie de classes spécifiques. La mise en œuvre de ces classes permet de supprimer les méthodes d'écriture dans des classes telles que Domaine et Zone.
-La nouvelle hiérarchie présente une interface de méthodes communes à chacun des formats, la spécificité d'un format étant reportée dans une méthode de plus bas niveau de la classe correspondante.
Cette conception offre la possibilité de synthétiser les instructions d'écriture dans la classe Post et de supprimer dans celle-ci toute distinction de format.
Dans la suite de ce document, les éléments de conception sont présentés. On décrit premièrement les classes constitutives de la nouvelle hiérarchie. L'interface de méthodes virtuelles communes aux différents formats est alors présentée en indiquant le rôle de chacune d'entre elles, puis on précise les spécificités de chaque format. Ensuite on récapitule les modifications apportées à des classes de base. Enfin on indique quels tests de validation ont été effectués.
Dans le cadre de ce premier lot, les développements sont réalisés sous la branche Evolution_Post dans une vue datée du 23/01/07, en s'appuyant sur une maquette fournie par le CEA.

1.1 HIERARCHIE DES CLASSES DE FORMAT DE POSTRAITEMENT

La nouvelle hiérarchie des classes de format de post-traitement a pour objectif de rassembler les méthodes d'écriture de données et de les supprimer de classes non dédiées à cet effet. On énumère brièvement les classes qui constituent cette hiérarchie ainsi que la mise en œuvre d'un objet format dans la classe de post-traitement. On précise ensuite l'interface de méthodes communes et les spécificités des différents formats.

1.1.1 Classes constitutives de la hiérarchie

La hiérarchie des formats a pour classe mère Format_Post_base (Kernel/Framework). Les classes qui héritent de cette classe sont les suivantes : Format_Post_Lml, Format_Post_Meshtv, Format_Post_Med, Format_Post_Lata (Kernel/Utilitaires). Cette dernière classe possède elle-même une classe fille Format_Post_Lata_V1.
Ces classes disposent d'une interface de méthodes qui sont décrites au paragraphe 1.1.2. Notons qu'une classe générique Format_Post (Kernel/Framework) a aussi été créée.

La classe Post est dotée d'un attribut de type Format_Post qui est typé dynamiquement dans le readOn du post-traitement en fonction du format sélectionné par l'utilisateur. Cet objet permet d'appeler les méthodes de l'interface sans avoir à faire une distinction de format.

Remarque : On notera que les classes Postraitement_dx et Postraitement_ft_dx sont supprimées.

1.1.2 Interface de méthodes communes aux différents formats de post-traitement

L'interface présentée ci-dessous a été mise en œuvre en s'appuyant sur la maquette fournie par le CEA. On précise pour chacune des méthodes : son nom, ses arguments et son rôle. Chaque méthode appelle elle-même une méthode statique spécifique au format considéré. Ces méthodes statiques sont décrites au paragraphe suivant.

initialize(const Nom& file_basename, const entier& format, const Nom& option_para)
-Initialise l'objet de type Format_Post

test_coherence(const entier& champs, const entier& stat, const double& dt_ch, const double dt_stat,)
-Vérification de l'égalité de dt_post pour les champs et les statistiques et vérification de l'égalité de dt_post pour des postraitements contenus dans un même fichier si le format le nécessite (lml, meshtv)

ecrire_entete(double temps_courant, entier reprise, const int& est_le_premier_post)
-Ecriture de l'entête dans l'index (fichier global pour format lml)
L'entête indique la version du code, le nom du cas et le nom du code.

preparer_post(const Nom& id_du_domaine, const int& est_le_premier_post, const entier& reprise,const double& t_init)
-Effectue des opérations de préparation si le format le nécessite (meshtv, med)
meshtv : lecture de la valeur initiale d'un compteur spécifique à ce format.
med : préparation de trois fichiers pour exploitation après calcul.

completer_post(const Domaine& dom, const entier axi, const Nature_du_champ& nature, const entier nb_compo, const Noms& noms_compo, const Motcle& loc_post, const Nom& le_nom_champ_post)

-Effectue des opérations complémentaires si le format le nécessite (meshtv, med)
meshtv : construction des formules (identifiants)
med : complément pour le fichier à exploiter après calcul

ecrire_domaine(const Domaine& domaine, const int& est_le_premier_post)
-Ecriture du domaine

ecrire_temps(const double temps)
-Ecriture du temps courant dans l'index (ou le fichier de post-traitement)

init_ecriture(double temps_courant, double temps_post, int est_le_premier_post, const Domaine& domaine)
-Préparation à l'écriture dans le fichier de post-traitement si le format le nécessite (meshtv)

ecrire_champ(const Domaine& domaine, const Noms& unite_, const Noms& noms_compo, int ncomp, double temps_, double temps_courant, const Nom& id_du_champ, const Nom& id_du_domaine, const Nom& localisation, const DoubleTab& data)
-Ecriture d'un tableau de valeurs (eventuellement interpolées)

finir_ecriture(double temps_courant)
-Finalise l'écriture dans un fichier de post-traitement si le format le nécessite (meshtv)

ecrire_item_int(const Nom& id_item, const Nom& id_du_domaine, const Nom& id_zone, const Nom& localisation, const Nom& reference, const IntVect& data, const entier reference_size)
-Ecriture d'un tableau d'entier (lata, lml)

finir(int& est_le_dernier_post)
-Ecriture de FIN dans l'index (ou fichier global pour format lml)

1.1.3 Description des méthodes statiques

On indique ici les méthodes spécifiques à chaque format en précisant les arguments qu'elles manipulent.

1.1.3.1 Format med

initialize(« nom_du_fichier_sans_extension »,format,option_parallele)

format : 0 pour formaté et 1 pour binaire
option_parallele : simple

ecrire_entete_med(« nom_fichier.med.index »,est_le_premier_post)

est_le_premier_post : indique si le postraitement traité est le premier pour le fichier de post-traitement considéré (vaut alors 1, 0 sinon)

preparer_post_med(nom_fich1,nom_fich2,nom_fich3,id_du_domaine,est_le_premier_post)
nom_fich1, nom_fich2 et nom_fich3 pour construire un .data à exploiter après calcul

completer_post_med(nom_fich2,nom1,nom2)
nom1 et nom2 servent à compléter le .data à exploiter après calcul

ecrire_domaine_med(domaine, « nom_fichier_numero_processeur.med »,est_le_premier_post, « nom_fichier.med.index »)

domaine : le domaine à post-traiter

ecrire_temps_med(temps, « nom_du_fichier.med.index »)
temps : temps courant du schéma en temps

ecrire_champ_med(domaine,unite_,noms_compo,ncomp,temps,nom_fich_ecrit,
id_champ,id_domaine,localisation,valeurs,nom_fich_index)

Noms unite_ :contient une liste de noms désignant les unités du champ considéré
Noms noms_compo : contient la liste des noms de composante du champ
entier ncomp : rend le numéro de la composante (-1 sinon)
double temps_ : temps du champ
Nom nom_fich_ecrit : nom_de_fichier.med
Nom id_champ : nom_post + « elem »+ « nom_du_domaine » si postraitement aux éléments
nom_post+ « som » + « nom_du_domaine » si postraitement aux sommets
Nom id_domaine : nom_du_domaine
Nom localisation : « ELEM » ou « SOM »
DoubleTab valeurs : tableau de valeurs à écrire
Nom nom_fich_index : « nom_de_fichier.med.index »

finir_med(« nom_fichier.med.index »,est_le_dernier_post)
entier est_le_dernier_post : indique si le post-traitement considéré est le dernier pour le fichier considéré

1.1.3.2 Format lata

initialize_lata(« nom_fichier_sans_extension »,format,option_para)
format : ASCII pour formaté (la valeur d'option correspondante passée à initialize() vaut 0) et BINAIRE pour binaire (la valeur d'option correspondante passée à initialize() vaut 1)
option_para : SINGLE_FILE pour écriture sur un seul fichier en parallèle (la valeur d'option correspondante passée à initialize() vaut « simple »), MULTIPLE_FILES pour écriture sur un fichier par processeur (la valeur d'option correspondante passée à initialize() vaut « multiple »)

ecrire_entete_lata_V1(« nom_de_fichier_sans_extension »,option_para,est_le_premier_post)
est_le_premier_post : indique si le postraitement traité est le premier pour le fichier de post-traitement considéré (vaut alors 1, 0 sinon)

ecrire_domaine_lata_V1(id_domaine,type_elem,dimension,sommets,elements,status,
lata_basename,format,option)
Nom id_domaine : nom du domaine
Motcle type_elem : domaine.zone(0).type_elem.que_suis_je()
entier dimension : dimension du domaine
DoubleTab sommets : les sommets du domaine
IntTab elements : les éléments de la zone
Status status : RESET ou INITIALIZED ou WRITING-TIME
Nom lata_basename: « nom_du_fichier_sans_extension »
Format format : ASCII ou BINAIRE
Options_Para option : SINGLE_FILE ou MULTIPLE_FILES

ecrire_temps_lata(temps,temps_format,lata_base_name,status,option)
double temps : temps courant du schéma en temps
double temps_courant : temps associé au format de postraitement

ecrire_champ_lata_V1(id_champ,id_domaine,localisation,valeurs,temps_,lata_basename,format,
option)
Nom id_champ : nom_post + « elem »+ « nom_du_domaine » si post-traitement aux éléments,
nom_post+ « som » + « nom_du_domaine » si post-traitement aux sommets.
Nom id_domaine : nom du domaine
Nom localisation : « ELEM » ou « SOM »

DoubleTab valeurs : tableau de valeurs à postraiter
double temps_ : temps du champ
Nom lata_basename : nom_du_fichier_sans_extension
Format format : ASCII ou BINAIRE
Options_Para : option SINGLE_FILE ou MULTIPLE_FILES

finir_lata_V1((« nom_de_fichier_sans_extension »,const Options_Para& option,const int& est_le_dernier_post)

entier est_le_dernier_post : indique si le post-traitement considéré est le dernier pour le fichier considéré.

1.1.3.3 Format lml

initialize(« nom_du_fichier_sans_extension »,format,option_para)
format : 0 pour formaté et 1 pour binaire
option_para : simple

test_coherence_lml(const entier& champs,const entier& stat,const double& dt_ch,
const double dt_stat)

entier champs : indique si des champs ont été demandés pour post-traitement (1 si oui , 0 sinon)
entier stat : indique si des statistiques ont été demandées (1 si oui, 0 sinon)
double dt_ch : periode de post-traitement des champs pour ce post-traitement
double dt_stat : période de post-traitement des statistiques pour ce post-traitement

ecrire_entete_lml(« nom_du_fichier.lml »,est_le_premier_post)
est_le_premier_post : indique si le post-traitement traité est le premier pour le fichier de post-traitement considéré.

ecrire_domaine_lml(domaine, « nom_de_fichier.lml »)
domaine : le domaine à post-traiter.

ecrire_temps_lml(temps, « nom_de_fichier.lml »)
temps : temps courant du schéma en temps.

ecrire_champ_lml(domaine,unite_,noms_compo,ncomp,temps_, id_champ,id_domaine,
localisation,valeurs, « nom_de_fichier.lml »)

Noms unite_ :contient une liste de noms désignant les unités du champ considéré
Noms noms_compo : contient la liste des noms de composante du champ
entier ncomp : rend le numéro de la composante (-1 sinon)
double temps_ : temps du champ
Nom id_champ : nom_post + « elem »+ « nom_du_domaine » si postraitement aux éléments,
nom_post+ « som » + « nom_du_domaine » si postraitement aux sommets.
Nom id_domaine : nom_du_domaine
Nom localisation : « ELEM » ou « SOM »
DoubleTab valeurs : tableau de valeurs à écrire

finir_lml(« nom_de_fichier.lml »,est_le_dernier_post)
entier est_le_dernier_post : indique si le post_traitement considéré est le dernier pour le fichier considéré

1.1.3.4 Format meshtv

initialize(« nom_du_fichier_sans_extension », format,option_para)
format : 0 pour formaté et 1 pour binaire
option_para : simple

test_coherence_meshtv(const entier& champs,const entier& stat,const double& dt_ch,
const double dt_stat)

entier champs : indique si des champs ont été demandés pour post-traitement (1 si oui, 0 sinon)
entier stat : indique si des statistiques ont été demandées (1 si oui, 0 sinon)
double dt_ch : periode de post-traitement des champs pour ce post-traitement
double dt_stat : période de post-traitement des statistiques pour ce post-traitement

ecrire_entete_meshtv(« nom_de_fichier.index_meshtv »,double temps_courant,entier reprise,
const int& est_le_premier_post)

temps_courant : temps courant du schéma en temps
reprise : 1 si reprise, 0 sinon
est_le_premier_post : : indique si le postraitement traité est le premier pour le fichier de post-traitement considéré (vaut alors 1, 0 sinon)

preparer_post_meshtv(const Nom& file,int& compteur, const entier& reprise,
entier& reprise_meshtv_non_vide, const double& t_init,double& t_reprise)

file : nom_de_fichier_numero_processeur.meshtv
compteur : pour indiquer le nombre d'écriture dans le fichier meshtv
reprise : pour indiquer si une reprise a été efféctuée
reprise_meshtv_non_vide : vaut 1 si une reprise a été effectuée et le fichier meshtv n'est pas vide
t_init : temps initial du calcul
t_reprise : temps auquel on fait la reprise (si fichier meshtv non vide)

Cette méthode sert à lire la valeur du compteur à l'initialisation du post-traitement

completer_post_meshtv(const Domaine& dom,const entier axi,const Nature_du_champ,const entier nb_compo,const Noms& noms_compo,const Motcle& loc_post,const Nom le_nom_champ_post)
Cette méthode sert à construire les « formules » (identifiants)

ecrire_domaine_meshtv(domaine, « nom_de_fichier_numero_de_processeur.meshtv »,compteur, « nom_de_fichier_index.meshtv »)
domaine : domaine à post-traiter

init_ecriture_meshtv(nom_fich_ecrit,temps_courant,temps_post,est_le_premier_post,compteur,
reprise_meshtv_non_vide,t_reprise,domaine,formule_meshtv,formule_visit,format_post)

Nom nom_fich_ecrit : nom_de_fichier.mestv
double temps_courant : temps courant du schéma en temps
double temps_post : temps du post-traitement
int est_le_premier_post : indique si le postraitement traité est le premier pour le fichier de post-traitement considéré (vaut alors 1, 0 sinon)
int compteur : pour indiquer le nombre d'écriture dans le fichier meshtv
entier reprise_meshtv_non_vide : vaut 1 si une reprise a été effectuée et le fichier meshtv n'est pas vide
double t_reprise : temps auquel on fait la reprise (si fichier meshtv non vide)
Domaine domaine : domaine de post-traitement
Nom formule_meshtv, formule_visit : expressions construites à partir des identifiants
Format_Post_Meshtv format_post : objet format de post-traitement

ecrire_champ_meshtv(domaine,unite_,noms_compo,ncomp,temps_,nom_fich_ecrit,compteur,
id_champ,id_domaine,localisation,valeurs,nom_fich_index)

Noms unite_ :contient une liste de noms désignant les unités du champ considéré
Noms noms_compo : contient la liste des noms de composante du champ

entier ncomp : rend le numéro de la composante (-1 sinon)
double temps_ : temps du champ
Nom nom_fich_ecrit : nom_de_fichier.mestv
entier compteur : incrément à chaque écriture de champs sur le fichier
Nom id_champ : nom_post + « elem »+ « nom_du_domaine » si postraitement aux éléments
nom_post+ « som » + « nom_du_domaine » si postraitement aux sommets
Nom id_domaine : nom_du_domaine
Nom localisation : « ELEM » ou « SOM »
DoubleTab valeurs : tableau de valeurs à écrire
Nom nom_fich_index : « nom_de_fichier.index_meshtv »)

finir_ecriture_meshtv(temps_courant,reprise_meshtv_non_vide,t_reprise)
temps_courant : temps courant du problème
reprise_meshtv_non_vide : vaut 1 si une reprise a été effectuée et le fichier meshtv n'est pas vide
t_reprise : temps auquel on fait la reprise (si fichier meshtv non vide)

finir_meshtv(nom_fich_index,est_le_dernier_post)
nom_fich_index : « nom_de_fichier.index_meshtv »
est_le_dernier_post : indique si le post-traitement considéré est le dernier pour le fichier considéré

Remarque : La méthode initialize() citée pour les formats med, lml et meshtv est une méthode virtuelle qui devrait en toute rigueur appeler une méthode statique qui n'utiliserait que l'argument « nom_du_fichier_sans_extension ».

1.2 DESCRIPTION DES MODIFICATIONS DE LA CLASSE POST

On présente dans cette partie les principales modifications réalisées dans la classe Post qui utilise maintenant un objet de type Format_Post.

Un nouvel attribut de type Format_Post est déclaré.

```
#include <derivFormat_Post_base.h>
class Postraitement : public Postraitement_base
{
 Format_Post format_post;
}
```

format_post est typé dynamiquement dans le readOn puis est initialisé.
Notons qu'une option « Parallele » est ajoutée pour le post-traitement en parallèle pour le format lata : simple pour écrire sur un seul fichier, multiple pour écrire sur un fichier par processeur.

```
Entree& Postraitement::readOn(Entree& s )
{
les_motcles[10] = "Parallele";

format_post.typer_direct(type_format);
format_post->initialize(base_name,binaire,option_para);
}
```

Des opérations d'initialisation et éventuellement de compléments sont effectuées dans la méthode init() puis le domaine est post-traité.

```
void Postraitement::init()
{
format_post->ecrire_entete(temps_courant,reprise,est_le_premier_postraitement_pour_nom_fich_);
format_post-
>preparer_post(nom_du_domaine,est_le_premier_postraitement_pour_nom_fich_,reprise,tinit);
```

```
  {
    LIST_CURSEUR(Nom) curseur1 = noms_champs_post_;

     while(curseur1) {
format_post->completer_post(dom,axi,nature,nb_compo,composantes,loc_post,le_nom_champ_post) ;
          ++curseur1;
    }
  }

 format_post->ecrire_domaine(dom,est_le_premier_postraitement_pour_nom_fich_);

}
```

La distinction de format est supprimée dans la méthode postraiter_champ() :

```
entier Postraitement::postraiter_champs()
{
if (temps < temps_courant)
  {
   if (est_le_premier_postraitement_pour_nom_fich_)
     format_post->ecrire_temps(temps_courant);
  }

 format_post->init_ecriture(temps_courant,temps,est_le_premier_postraitement_pour_nom_fich_,dom);

 if (temps < temps_courant)
    temps=temps_courant;

 LIST_CURSEUR(Nom) curseur1 = nom_champs_a_post_;
 while(curseur1){
         postraiter(…) ;
  ++curseur1;
 }

 format_post->finir_ecriture(temps_courant);
}
```

1.3 CLASSES DE BASE MODIFIEES

On décrit dans cette partie les principales modifications des classes de base concernées par la nouvelle conception des formats.

Les deux méthodes suivantes ont été ajoutées à la classe Champ_base :
-La méthode postraiter_champ() effectue l'interpolation des valeurs du champ (aux sommets ou aux éléments) puis utilise l'objet format_post pour écrire les valeurs du champ dans un fichier. L'interpolation est effectuée par les méthodes calculer_valeurs…() de la classe Champ_base. Toutefois notons que, pour tous les formats, la méthode calculer_valeurs_som_compo_post() est dorénavant appelée sans tenir compte d'une correction liée aux conditions limites. Les précédentes méthodes de post-traitement liées à chaque format sont supprimées. Elles ont aussi été supprimées dans les classes Champ, Champ_Inc, Champ_Inc_base et Champ_Post.
-La méthode completer_post_champ() utilise l'objet format_post pour lancer des opérations complémentaires si le format utilisé le nécessite, ce qui est le cas pour les formats meshtv et med.
Les deux méthodes ci-dessus sont appelées dans la classe Post par l'intermédiaire du champ concerné.

Les deux méthodes suivantes ont été ajoutées à la classe Operateur_Statistique_tps_base :

-La méthode postraiter_statistiques() est déclarée en virtuelle pure et surchargée dans ses classes filles Moyenne, Ecart_type et Correlation. Cette méthode effectue le calcul (division par dt de l'intégrale du champ) puis lance le post-traitement en appelant postraiter_champ() par l'intermédiaire de l'intégrale du champ et ré-effectue l'intégration (multiplication par dt). Les précédentes méthodes de post-traitement liées à chaque format sont supprimées. Elles ont aussi été supprimées dans les classes Operateur_Statistique_tps, Operateur_Statistique_tps_base.
-La méthode completer_post_statistiques() appelle completer_post_champ() par l'intermédiaire de l'intégrale du champ. La méthode est surchargée dans Correlation de manière à ce qu'elle soit sans effet pour ce type d'opérateur.
Les deux méthodes citées ci-dessus sont appelées dans la classe Post par l'intermédiaire de l'opérateur statistique.

Notons que les méthodes de post-traitement spécifiques à chaque format ont aussi été supprimées des classes Domaine, Zone et Zones. La méthode postraiter_lml() de la classe DomaineCut et l'appel de cette méthode dans la classe Decouper sont aussi supprimés.

1.4 TESTS DE VALIDATION

La nouvelle conception a été testée sur l'ensemble des cas de non-régression pour le format lml.
Pour les autres formats (meshtv, med et lata_V1), les tests ont été réalisés en séquentiel et parallèle sur un ensemble de cas plus restreint.
Format meshtv :
Axi_2D, Axi_3D, bi_dom_2D, bi_dom_3D, es (cas identique à keps mais post-traitement de la composante vitessex), Hexaedre, keps, M3DLAM, Multi_dom_fluide, reprise, stat, Tetra, Triangle.
Pour ce format, un script de comparaison des fichiers avant et après développement a été fourni par le CEA.

Format med :
Canal_perio_VDF_2D, Conduite_NSComp_BM, Hyd_Axi, Hyd_Axi_RZ, Marche_quadra, Quasi_Comp_Cond_GP_VDF, Quasi_Comp_Obst_GP_VDF_BasReyn,
Quasi_Comp_Obs_GP_VEF_Pmoy, ThHYd_C_K_Eps, ThHyd_C_K_Eps_VEF, VAHL_DAVIS, WALE_Canal_Hyd_VEF, coude_irreg, docond, keps_1D_VEF, reprise_QC, smago_VEF
Les résultats au format med sont transformés au format lml pour être testés avec l'outil lance_test_seq_par. La transformation au format lml est faite en utilisant le jeu de donnés créé en préparation du post-traitement, et qui est constitué d'un Pb_Med.

Format lata_V1 :
Cas identiques à ceux testés pour format med et aussi Post_Eclate, stat_reprise
Pour ce format, on utilise un script similaire à celui utilisé pour le format meshtv.

En outre, on note qu'avec la nouvelle conception, les statistiques en série ont été désactivées car ce type de statistiques génère un arrêt avec le format meshtv.

2. LOT 2 : MODIFICATION DU FRAMEWORK

La version actuelle du logiciel Trio_U possède un mécanisme d'accès, depuis les classes de post-traitement, aux champs discrets du problème traité. Le principe de fonctionnement repose sur l'appel par le problème de la méthode le_champ() qui appelle elle-même les méthodes a_pour_champ_inc(), a_pour_champ_fonc() et creer_champ(). L'appel de ces méthodes est délégué aux équations qui renvoient une référence au champ considéré si l'identifiant passé en paramètre est reconnu par la classe interrogée.
La mise en œuvre actuelle de ce mécanisme de recherche présente un défaut majeur qui réside dans la couverture partielle des champs discrets d'un problème. En effet, les classes d'équation sont quasiment les seules à pouvoir identifier un champ ou ses composantes. Les champs qui constituent l'inconnue d'une équation sont par conséquent systématiquement pris en compte. En revanche les équations gèrent moins rigoureusement les champs portés par d'autres classes, comme par exemple les propriétés physiques d'un milieu.

Dans le cadre de ce lot, une modification du framework de Trio_U est mise en œuvre pour atteindre trois objectifs :
-Le mécanisme de requête des champs actuellement gérés par les équations est élargi aux autres classes du framework. Les hiérarchies de classes de milieu, loi d'état, opérateur, source, modèle de turbulence, modèle bas Reynolds, loi de paroi et traitement particulier seront dotées des méthodes nécessaires à la recherche des champs qui leur sont attributs.
-Une option « description » sera à la disposition de l'utilisateur pour lister l'intégralité des champs post-traitables pour un problème donné.
-La gestion des « champs calculés » qui sont dédiés uniquement au post-traitement, comme par exemple la vorticité, sera clarifiée et rendue cohérente avec l'interface de requête des champs.

Dans la suite de ce document, on présente la révision de la conception du framework du logiciel Trio_U. Le nouveau mécanisme de requête des champs compris est premièrement présenté. On clarifie ensuite la notion de « champs calculés » dans le logiciel. L'interface de méthodes qui gère une requête ou assure la création d'un « champ calculé », est alors décrite. On précise notamment comment implémenter le principe de requête dans une classe de base et ses classes filles. Enfin on indique les nouveaux éléments de conception de la classe de post-traitement.
Dans le cadre de ce lot, les développements ont été réalisés dans la branche Evolution_Post_Lot_2 par rapport à une version datée du 06/04/07, de manière indépendante de ceux du premier lot.

2.1 MECANISME DE REQUETE D'UN « CHAMP COMPRIS »

2.1.1 Description des champs compris

Dans le processus de recherche généralisé d'un champ, chaque classe doit être capable d'identifier ses champs attributs et de renvoyer une référence à l'un d'entre eux. D'autre part, chaque classe doit être en mesure de fournir la liste de noms de ses champs ou de leurs composantes pour renseigner l'utilisateur. Ces considérations nous ont conduit à développer une nouvelle classe *Champs_compris*. Chaque classe portant des champs susceptibles d'être post-traités sera dotée d'un attribut de type Champs_compris.

2.1.1.1 Classe des champs compris

La classe *Champs_compris* (Kernel/Framework) possède l'interface suivante :

```
class Champs_compris : public Objet_U
{

  declare_instanciable(Champs_compris)

    public :

    virtual const Champ_base& get_champ(const Motcle& nom) const;
    virtual void ajoute_champ(const Champ_base& champ);
    virtual const Noms& liste_noms_compris() const;
    virtual Noms& liste_noms_compris();

    protected :

    LIST(REF(Champ_base)) liste_champs_;
    //Contient le nom de « postraitement » des champs d'une classe (ex : vitesse, vitesseX)
    Noms liste_noms_;

};
```

L'attribut liste_champs_ contient une liste de référence aux champs d'une classe. L'attribut liste_noms_ contient la liste de noms de ces champs ou de leurs composantes.
La méthode get_champ() permet de lancer une requête sur liste_champs_ et la méthode ajoute_champ() permet d'ajouter une référence à un champ à liste_champ_. Des méthodes liste_noms_compris() donnent accès à liste_noms_.

Le détail du codage des méthodes ajoute_champ et get_champ() est donné ci-dessous.

```
void Champs_compris::ajoute_champ(const Champ_base& champ)
{
 REF(Champ_base) champ_ref;
 champ_ref = champ;

 assert(!liste_champs_.contient(champ_ref));
   liste_champs_.add(champ_ref);
}
```

//La méthode donne accès à la référence à un champ si celui-ci est identifié par son nom

```
const Champ_base& Champs_compris::get_champ(const Motcle& nom) const
{
 REF(Champ_base) ref_champ;
 LIST_CURSEUR(REF(Champ_base)) curseur = liste_champs_;
   Motcle nom_champ;
   while (curseur) {
    nom_champ = Motcle(curseur.valeur().valeur().le_nom());
    if (nom_champ==nom) {
     return curseur.valeur();
    }
    else {
     entier nb_composantes = curseur.valeur().valeur().nb_comp();
     for (entier i=0;i<nb_composantes;i++)
      {
        nom_champ = Motcle(curseur.valeur().valeur().nom_compo(i));
        if (nom_champ==nom) {
             return curseur.valeur();
            }
      }
    }
    ++curseur;
   }
 throw Champs_compris_erreur();

 return ref_champ;

}
```

2.1.1.2 Classe Champs_compris_erreur

Une classe Champs_compris_erreur a été créée (Champs_compris.h). Cette classe permet d'utiliser le procédé try-catch dans la méthode get_champ(). Ce procédé donne à la méthode get_champ() la possibilité d'envoyer un message « d'exception » si le champ recherché n'est pas trouvé. L'objet qui reçoit le message poursuit alors la recherche.

2.1.1.3 Construction d'un objet de type Champs_compris

L'attribut champs_compris_ d'une classe donnée sera déclaré en private, ce qui obligera chaque classe d'une hiérarchie à tester ses propres champs. Cet attribut sera rempli de la manière suivante :
list_noms_ est rempli dans le constructeur de la classe qui porte l'attribut champs_compris_. Toutefois une exception à cette règle provient d'une catégorie de problème qui porte une liste d'équations de scalaire passif pour lesquelles le nom de l'inconnue est modifié en lui concaténant le numéro de son équation dans la liste. Dans ce cas list_noms_ est modifié dans le readOn de l'équation concernée. On relève aussi le même type d'adaptation pour l'équation Transport_Interfaces_FT_Disc.
list_champs_ est rempli après la discrétisation de chacun des champs de la classe qui les porte, par un appel à la méthode ajoute_champ(). Dans le cas des équations, la construction de list_champs_ est immédiate du fait qu'elles possèdent l'objet de discrétisation et leur propre méthode discretiser(). En revanche pour la hiérarchie du milieu qui n'accède pas directement à l'objet de discrétisation et ne possède pas de méthode propre de discrétisation, la construction de list_champs_ est réalisée directement dans les classes de discrétisation ce qui implique de dupliquer le codage. D'autre part, cela implique que chaque classe de milieu possède une méthode d'accès à champs_compris_. On notera qu'une méthode d'accès à champs_compris_ a aussi été ajoutée pour la loi d'état.

2.1.2 Principe du mécanisme de requête

Le principe de requête généralisée d'un champ est basé sur l'utilisation d'une nouvelle méthode get_champ() décrite ci-dessus pour la classe Champs_compris et qui est aussi implémentée pour diverses hiérarchies de classe détaillées dans la suite de ce document. Le post-traitement appelle la méthode du problème en lui passant un nom en argument. Le problème délègue l'appel au milieu et à ses équations. Le milieu interroge sa classe mère de manière récursive ce qui permet de consulter toute la hiérarchie amont de cette classe. Chacune des classes interrogées teste ses champs compris puis transmet l'appel à ses attributs si le champ recherché n'a pas été identifié. Enfin le milieu consulte sa propre liste de champs compris puis transmet l'appel à ses attributs. En cas de réponse négative du milieu, la requête est transmise aux équations avec un mécanisme identique à celui décrit précédemment. Ainsi pour une équation donnée toute la hiérarchie amont sera consultée et l'appel sera aussi transmis aux opérateurs puis aux termes sources si cela s'avère nécessaire.

2.2 SPECIFICATION DES « CHAMPS CALCULES »

On désigne par « champ calculé » un champ qui n'intervient pas dans la résolution du problème. Ces champs sont uniquement évalués pour être post-traités (ex : vorticité). Dans l'ancienne conception du post-traitement, la lecture d'un mot clé désignant un champ calculé déclenche la création de ce champ par l'intermédiaire de l'équation qui le reconnaît (Navier_Stokes pour la vorticité). L'équation renvoie alors le champ au post-traitement qui le stocke dans une liste et se charge par la suite de sa mise à jour.
Dans la nouvelle conception du post-traitement chacun des « champs calculés » est attribut de type Champ_Fonc d'une classe spécifique. La classe qui porte un « champ calculé » se charge comme précédemment de sa discrétisation. La mise à jour du champ est réalisée lors de chaque appel par la méthode get_champ() si le temps du « champ calculé » est différent du temps du champ duquel il dépend. Dans le cas où le « champ calculé » ne dépendrait pas d'un champ inconnu, sa mise à jour est effectuée si son temps est différent du temps courant. Dans le cadre de cette nouvelle conception, seules des équations portent actuellement un « champ calculé ». Ce choix est justifié par le fait que la discrétisation d'un champ, puis son ajout à champs_compris_ est aisé pour une équation.

La liste des « champs calculés » est présentée ci-dessous en précisant pour chacun de ces champs la classe qui le porte.

Equation_base : volume_maille
Navier_Stokes_std : vorticite, critere_Q, porosite_volumique, combinaison_champ
Navier_Stokes_Turbulent : y_plus
Convection_Diffusion_Temperature : temperature_paroi, gradient_temperature, h_echange
Euler : vitesse
Systeme_SG_base : TxCP, gradrho

Remarque : la version actuelle ne permet de post-traiter qu'un seul coefficient h_echange_T_utlisateur.

2.3 INTERFACE POUR REQUETE D'UN CHAMP ET CREATION DE « CHAMPS CALCULES »

2.3.1 **Description de l'interface**

Le mécanisme de requête d'un champ post-traitable et la création d'un « champ calculé » sont gérés en s'appuyant sur une interface de méthodes déclarées dans une nouvelle classe *Champs_compris_interface.* Cette interface est mise en œuvre dans les classes de base suivantes :

Probleme_base, Equation_base, Milieu_base, Operateur_base, Source_base, Traitement_particulier_NS_base, Traitement_particulier_Solide_base, Mod_turb_hyd_base, Modele_turbulence_scal_base, Loi_Etat_base, Turbulence_paroi_base, Turbulence_paroi_scal_base, Modele_Fonc_Bas_Reynolds_base, Modele_Fonc_Bas_Reynolds_Thermique_Base

L'interface est également propagée dans les classes dérivées de ces classes de base si elles possèdent des champs discrets.

La classe *Champs_compris_interface* possède l'interface suivante :

```
enum Option { NONE, DESCRIPTION} ;

class Champs_compris_interface
{
   public :

   virtual void creer_champ(const Motcle& motlu) =0;
   virtual const Champ_base& get_champ(const Motcle& nom) const=0;
   virtual void get_noms_champs_postraitables(Noms& nom, Option opt=NONE) const=0;

  protected :

};
```

La méthode get_noms_champs_postraitables() permet de lister l'ensemble de noms des champs et de leurs composantes. Dans le cas où l'option opt est fixée à DESCRIPTION, le format nom_de_classe :: nom de champs post-traitables est utilisé pour imprimer dans le fichier de sortie d'extension .err les champs post-traitables pour un problème donné. Dans le cas où l'option opt est fixée à NONE, l'ensemble des noms est affecté dans l'argument nom.
La méthode get_champ() permet de gérer la recherche d'un champ et de renvoyer la référence à celui-ci lorsqu'il est identifié. Dans le cas où le champ recherché est un « champ calculé », la méthode get_champ lance si nécessaire la mise à jour du champ. La signature const de la méthode implique alors l'utilisation de ref_cast_non_const. La méthode creer_champ() permet de lancer la création d'un « champ calculé ».
La mise en œuvre de ces méthodes est détaillée ci-dessous pour la classe Pb_base et pour un exemple type d'une hiérarchie de classes A_mere et A_fille.

Remarque : Les précédentes méthodes de requête a_pour_Champ_Inc, a_pour_Champ_Fonc, comprend_champ et comprend_mot ont été supprimées.

2.3.2 Mise en oeuvre de l'interface dans Pb_base

On présente ici l'implémentation de l'interface de méthodes de requête et de création de champ pour la classe Probleme_base. Ces méthodes seront en particulier appelées dans la classe de post-traitement.

```
class Probleme_base : public Champs_compris_interface, public Probleme_U {
public :
//Methodes de l interface des champs postraitables
  virtual void creer_champ(const Motcle& motlu);
  virtual const Champ_base& get_champ(const Motcle& nom) const;
  virtual void get_noms_champs_postraitables(Noms& nom,Option opt=NONE) const;
};
void Probleme_base::creer_champ(const Motcle& motlu)
{
  milieu().creer_champ(motlu);
  entier nb_eq = nombre_d_equations();
  for (entier i=0; i<nb_eq; i++)
     equation(i).creer_champ(motlu);
}
```

```
const Champ_base& Probleme_base::get_champ(const Motcle& nom) const
{
  REF(Champ_base) ref_champ;
  try {
    return milieu().get_champ(nom);
  }
  catch (Champs_compris_erreur) {
  }

  entier nb_eq = nombre_d_equations();
  for (entier i=0; i<nb_eq; i++)
   {
     try {
      return equation(i).get_champ(nom);
     }
     catch (Champs_compris_erreur) {
     }
   }

  Cerr<<"Le nom indique "<<nom<<" ne correspond pas à un champs compris par le probleme"<<finl;
  Cerr<<"Verifier le nom du champ à indiquer dans le bloc de postraitement des champs"<<finl;
  exit(-1);

  //Pour compilation
  return ref_champ;
}


void Probleme_base::get_noms_champs_postraitables(Noms& nom,Option opt) const
{
 milieu().get_noms_champs_postraitables(nom,opt);
 entier nb_eq = nombre_d_equations();
 for (entier i=0; i<nb_eq; i++)
   equation(i).get_noms_champs_postraitables(nom,opt);
}
```

L'implémentation de l'interface est « particulière » pour la classe Probleme_base car celle-ci ne porte pas d'attribut champs_compris_ et par conséquent aucune requête ne peut être lancée sur celui-ci. Cette interface a été surchargée dans les classes Probleme_Interface_base, Probleme_Interface_Th_base qui portent des champs, Probleme_SG qui possède un systeme_SG et Pb_MED.

En revanche la syntaxe des méthodes de l'interface est systématique pour la hiérarchie de classe des équations, milieu, opérateur, source, traitement_particulier, modèle de turbulence, loi d'état, loi de paroi et modèle bas Reynolds. Cette syntaxe devra être respectée par la suite s'il est nécessaire de propager l'interface dans des classes dérivées ou pour de nouvelles classes de base.
On décrit ci-dessous la syntaxe générale à respecter pour une hiérarchie « type ». On trouvera aussi à titre d'exemple le codage de l'interface pour les classes Eqn_base et Navier_Stokes dans l'annexe 1.

Dans cet exemple, on considère le cas d'une classe A_fille héritant d'une classe A_base, cette dernière héritant elle-même de Champs_compris_interface. A_fille possède un constructeur permettant d'initialiser le nom de ses champ_compris et de leurs composantes. A_fille porte aussi un champ inconnu inco nommé nom_inco, un champ calculé ch_A nommé nom_A dont on admettra pour l'exemple qu'il dépend de la vitesse et un attribut (attribut_1).

2.3.3 Mise en œuvre de l'interface dans une hiérarchie de classe type

```
Class A_fille ::A_base {
declare_instanciable_sans_constructeur(A_fille)
 public :
 A_fille();
   ….
 protected :
 Champ_Inc inco;
 Champ_Fonc ch_A;
 Type_B attribut_1;
 private :
 Champs_compris champs_compris_;
}

A_fille::A_fille()
{
 Noms& nom=champs_compris_.liste_noms_compris() ;
 nom.dimensionner(2) ;
 nom[0] = « nom_inco » ;
 nom[1] = « nom_A » ;
}

void A_fille::creer_champ(const Motcle& motlu)
{
                A_base ::creer_champ(motlu) ;
                if (motlu= = « nom_A ») {
                  if ( !ch_A.non_nul()) {
                          dis.creer_champ_A(…) ;
                          champs_compris_.ajoute_champ(ch_A) ;
                   }
                }
                if (attribut_1.non_nul())
                  attribut_1.creer_champ(motlu) ;
}
const Champ_base& A_fille::get_champ(const Motcle& nom) const
{
                REF(Champ_base) ref_champ ;

                if (nom=="nom_A")
                {
                  double temps_init = schema_temps().temps_init();
                    Champ_Fonc_base&
                ch_fonc=ref_cast_non_const(Champ_Fonc_base,ch_A.valeur());
                if     (((ch_fonc.temps()!=la_vitesse->temps())     ||     (ch_fonc.temps()==temps_init))&&
                (la_vitesse->mon_equation_non_nul())) {
                ch_fonc.mettre_a_jour(la_vitesse->temps());
                }
                return champs_compris_.get_champ(nom);
                }

                try { return A_base::get_champ(nom); }
                catch ( Champs_compris_erreur ) {
                }

                try { return champs_compris_.get_champ(nom); }
                catch (Champs_compris_erreur) {
                }
```

```
                if (attribut_1.non_nul())
                  try { return attribut_1.get_champ(nom) ; }
                  catch ( Champs_compris_erreur )  {
                }
                throw Champs_compris_erreur()
                return ref_champ ;
}
void A_fille::get_noms_champs_postraitables(Noms& nom,Option opt) const
{
                A_base::get_noms_champs_postraitables(nom,opt) ;
                 if (opt=DESCRIPTION)
                Cerr<< « A_fille : »<<champs_compris_.liste_noms_compris()<<finl ;
                else
                  nom.add(champs_compris_.liste_noms_compris()) ;
                if (attribut_1.non_nul())
                 attribut_1.get_noms_champs_postraitables(nom,opt) ;
}
```

Remarque : Dans le cas d'une classe de base située en amont d'une hiérarchie de classe, la syntaxe sera identique à l'exception de l'appel à la méthode de la classe mère.

2.4 NOUVELLE CONCEPTION DE LA CLASSE DE POSTRAITEMENT

La requête d'un champ à post-traiter se fait par l'intermédiaire de son identifiant. La classe Post ne possède donc plus pour attribut une liste de champs à post-traiter précédemment appelée champs_a_postraiter_, elle est dorénavant dotée d'une liste de Noms nommée noms_champs_a_postraiter_. Pour cela, une nouvelle clase List(Noms) a été créée. Le premier élément d'un Noms contient l'identifiant du champ et le second contient la localisation du post-traitement. D'autre part l'attribut champs_crees_ précédemment constitué de l'ensemble des « champs calculés » est supprimé, étant donné que ces champs sont maintenant portés par des classes spécifiques.

La méthode ReadOn a été modifiée pour assurer la fonctionnalité d'affichage des champs post-traitables pour un problème donné. L'option « description » peut être activée par l'utilisateur après la spécification du mot clé « champs » dans le jeu de données. Dans ce cas le problème appelle la méthode get_noms_champs_postraitable() avec l'option description, ce qui permet d'afficher la liste des champs post-traitables dans le fichier d'extension .err.

```
Entree& Post ::readOn()
//cas des champs
if (motlu == "description") {
          Noms liste_noms;
          Option opt=DESCRIPTION;
          mon_probleme->get_noms_champs_postraitables(liste_noms,opt);
          s>>motlu;
        }
```

La méthode lire_champs_a_postraiter() est modifiée pour construire la liste noms_champs_a_postraiter_. Dans un premier temps, on construit list_noms_ qui contient les identifiants de champs post-traitables, en appelant la méthode get_noms_champs_postraitables(). Ensuite les mots clés spécifiés dans le jeu de données sont parcourus. On vérifie que le mot clé correspond à un champ post-traitable ou à l'une de ses composantes sinon le logiciel sort en erreur. Si l'identifiant correspond à un « champ calculé », le problème délègue la création du champ à la classe qui le porte sinon la méthode creer_champ() ne fait rien. Ensuite nom_champ est rempli à partir de l'identifiant et de la localisation précisés par l'utilisateur, puis ajouté à noms_champs_a_postraiter_.

```
entier Post ::lire_champs_a_postraiter()
{
  s >> motlu ;
  Noms nom_champ;
  nom_champ.dimensionner(2);
  Noms liste_noms;

  mon_probleme->get_noms_champs_postraitables(liste_noms);
  Cerr<<"liste_noms="<<liste_noms<<finl;
  entier mot_ok = 0;
  while (motlu != accolade_fermee)
   {
     for (entier i=0; i<liste_noms.size();i++)
       if (Motcle(liste_noms[i])==motlu) {
            mot_ok=1;
            i=liste_noms.size();
            }
      if (mot_ok) {
         mon_probleme->creer_champ(motlu);
             nom_champ[0]=motlu;
      } else {
         Cerr<<"Le mot lu "<<motlu<<" ne correspond pas à un champ postraitable ou à une de ses
composantes"<<finl;
             Cerr<<"Vérifier votre jeu de donnees"<<finl;
             exit(-1);
      }

      // Lecture de la localisation eventuelle du postraitement:
      s >> motlu2;
           if ( (motlu2 == elem) || (motlu2 == som) ) {
              nom_champ[1]=motlu2;
              s >> motlu;
           }
           else
             {
              nom_champ[1]=som;
              motlu = motlu2;
             }

     // Ajout du nom du champ (ou d une composante d un champ) et de sa localisation à la liste
noms_champs_a_postraiter_
     assert(!noms_champs_a_postraiter_.contient(nom_champ));
     noms_champs_a_postraiter_.add(nom_champ);
   }
}
```

La méthode postraiter_champs() a été adaptée pour parcourir une liste de Noms en remplacement d'une liste de champs. La méthode get_champ() est appelée pour récupérer la référence au champ considéré et celle-ci est exploitée pour post-traiter les valeurs du champ.

```
entier Post ::postraiter_champs()
{ …
 LIST_CURSEUR(Noms) curseur1 = noms_champs_a_postraiter_;
  while(curseur1) {
     Nom nom_post = curseur1.valeur()[0];;
     Nom localisation = curseur1.valeur()[1];
     const REF(Champ_base)& champ = mon_probleme->get_champ(nom_post);
     entier ncomp=composante(nom_post,champ);
```

```
   switch(rang)
    {
    case 0:
         {
         if (ncomp!=-1)
           champ->postraiter_lml_dom(os,dom,ncomp,localisation);
         else
           champ->postraiter_lml_dom(os,dom,localisation);
         }
         break;
…
}
```

2.5 IDENTIFICATION DES NOUVEAUX CHAMPS POST-TRAITABLES

On résume ici l'ensemble des nouveaux champs post-traitables pour un milieu en spécifiant la classe concernée ainsi que le mot clé associé.

2.5.1 Milieu

Les champs suivants sont post-traitables :
Milieu_base : lambda (conductivite), alpha (diffusivite), beta_th (dilatabilite), Cp (capacite_calorifique)
Fluide_Incompressible : mu (viscosite_dynamique), nu (viscosite_cinematique)
Fuide_Quasi_Compressible : vitesse_CN_ (vitesse_CN), mu_sur_Sc (mu_sur_Schmidt)
Fluide_Ostwald : K_ (consistance)
Fluide_Compressible_base : mu (viscosite_dynamique)

Remarque : La capacité calorifique Cp de Milieu_base est post-traité sous le nom chaleur_specifique_pression_constante pour un fluide compressible.

2.5.2 Equation

Le champ suivant est post-traitable :
Navier_Stokes_Front_Tracking : Kapa_ (viscosite_dynamique2)

2.5.3 Modèle de turbulence

Le champ suivant est post-traitable :
Hyd_SGE_DSGS_VDF, Hyd_SGE_Smago_Dyn_VDF : coeff_field (dynamic_coefficient)

2.5.4 Traitement particulier

Les champs suivants sont post-traitables :
Trait_Part_NS_pression_VDF, Trait_Part_NS_pression_VEF : ch_p (Pression_porosite)
Trait_Part_NS_Brech_VEF : ch_p (Pression_porosite), ch_ri (Richardson)

2.6 VALIDATION ET CAS TESTS MODIFIES

Les cas tests de non-régression suivants ont été modifiés pour tester les nouveaux champs post-traitables (le ou les champs ajoutés sont indiqués entre parenthèses) :

ThHyd_Cond_K_Eps (viscosite_dynamique, viscosite_cinematique, conductivite)
Nu_var_VEF (viscosite_cinematique)
Phys_var (conductivite, diffusivite)
Quasi_Comp_Cond_GP_VDF (vitesse_CN)
Quas_Comp_Cond_GP_VDF_FM (mu_sur_schmidt, fraction_massique0 déjà post-traitable)
Quasi_Comp_Cond_GR_VDF (capacite_calorifique)
ThHyd_2D_VDF_Ostwald (consistance)
DYN_Canal_ThHyd_VDF (dynamic_coefficient)

Les développements ont été validés en testant l'ensemble des cas de non-régression (format lml). On note un arrêt pour le cas test Champs_fonc_QC_rayo_semi_transp car le post-traitement de l'enthalpie est demandé pour un gaz parfait et la version actuelle du logiciel rend dans ce cas l'inconnue de l'équation d'énergie qui est la température.
Un arrêt probablement lié à un bug mémoire présent dans la v1.5.2 est observé pour les cas Kernel_Post_Med et MED_docond. Par conséquent l'interface ne peut être validée pour Pb_Med.
Enfin les écarts présentés par les cas Def_VEF_impl, U_in_var_impl et PAR_U_in_var_impl étaient déjà observés à l'ouverture de la vue.

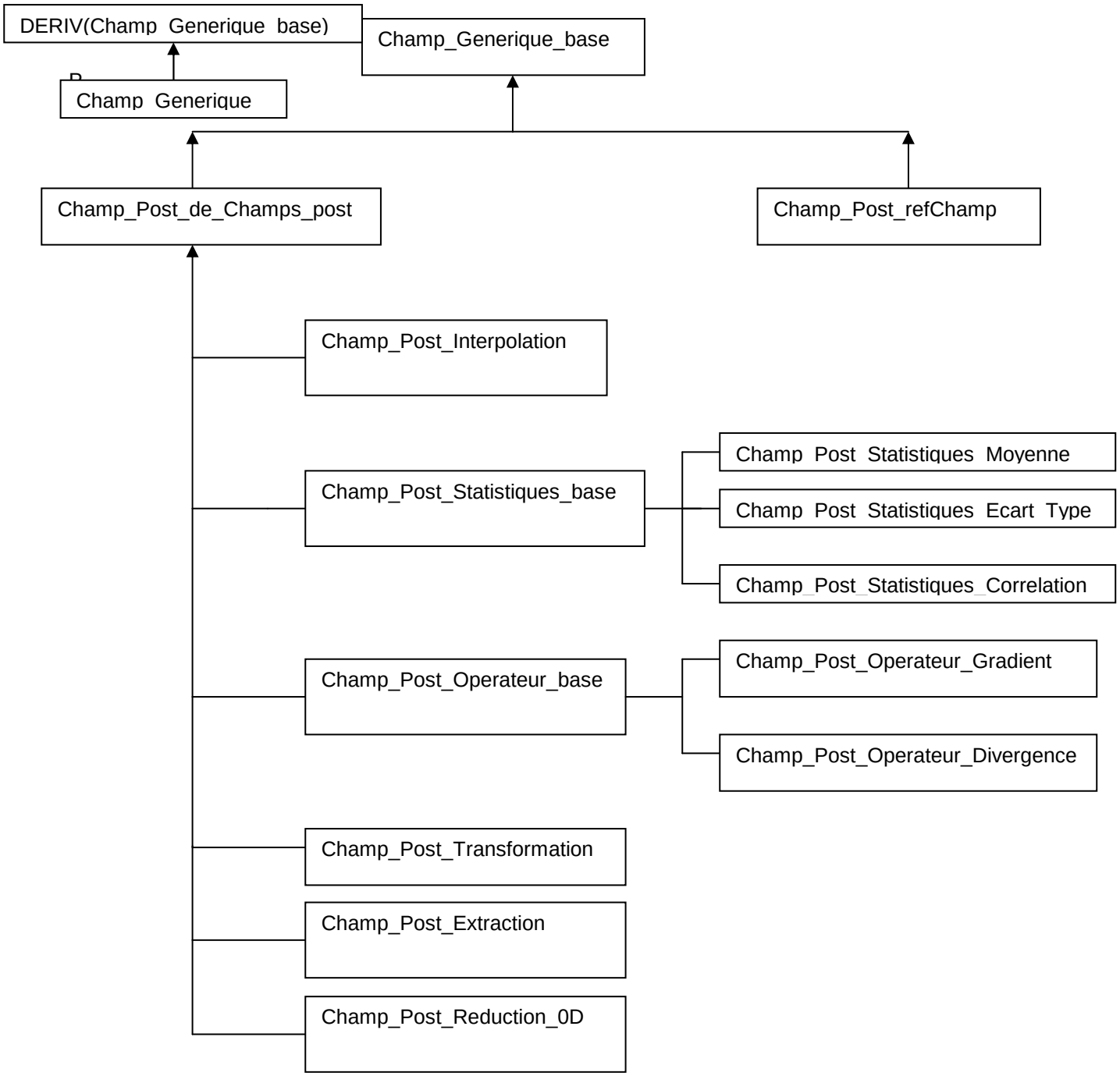
3. LOT 3 : FRAMEWORK DE POSTRAITEMENT

Le module de post-traitement du logiciel Trio_U offre les fonctionnalités suivantes. En premier lieu, le logiciel assure la possibilité d'interpoler les valeurs d'un champ discret aux sommets du domaine de calcul ou au centre de gravité de ses éléments. Notons que l'interpolation peut aussi être réalisée sur un domaine différent de celui considéré pour la simulation. D'autre part, l'intégration temporelle de champ peut être activée. Les statistiques ainsi établies sur des champs du problème sont post-traitables. Enfin, les champs discrets ou leurs statistiques peuvent être sondés en un ou plusieurs points du domaine de calcul.
Dans le cadre de ce lot, une nouvelle conception du framework est proposée. Celle-ci a pour finalité de rendre générique une opération élémentaire appliquée à un champ discret. La généricité rend alors possible la combinaison de divers « opérateurs », accroissant ainsi les fonctionnalités de post-traitement. En particulier, la création d'un champ discret porté par le problème pour répondre à une option spécifique de post-traitement n'est alors plus nécessaire.
Les opérations assurées sont : la requête d'un champ du problème, l'interpolation, la réalisation d'une statistique temporelle, une transformation par application d'un opérateur ou par combinaison, une extraction, la réduction 0D du champ et l'écriture dans un fichier à un format quelconque. D'autre part, une sonde pourra porter sur un champ quelconque du post-traitement.
En terme d'utilisation, l'ancienne syntaxe du jeu de données reste valable pour les fonctionnalités déjà existantes : interpolation de champ et statistiques. Des « macros » sont développées pour assurer la compatibilité de cette syntaxe avec le nouveau framework. En revanche, la requête de fonctionnalités plus fines nécessite l'utilisation d'une nouvelle syntaxe.
Dans la suite de ce document, on présente la conception du framework. On décrit en particulier la nouvelle hiérarchie de classes des champs génériques. Les spécificités de chacun d'entre eux sont détaillées. Les principales modifications de conception concernant les classes déjà existantes du module de post-traitement sont aussi précisées. On décrit en particulier la « macro » permettant la lecture de l'ancienne syntaxe des instructions de post-traitement. Ensuite la nouvelle syntaxe donnant accès à des fonctionnalités élargies est présentée. Enfin on décrit les cas de validation considérés pour tester la nouvelle conception.
Dans le cadre de la mise en oeuvre de ce lot, les développements ont été réalisés dans la branche Evolution_Post_Lot_2, contenant les développements des Lots 1 et 2, actualisés par rapport à une version v1.5.2.

3.1 CONCEPTION DU FRAMEWORK

3.1.1 Description de la hiérarchie des champs génériques

La conception du framework de post-traitement repose sur une nouvelle hiérarchie de champs. Celle-ci a pour objectif, d'une part de permettre l'encapsulation d'un champ discret du problème de type Champ_base, et d'autre part de pouvoir combiner l'action « d'opérateurs » génériques.
On présente ci-dessous le diagramme d'héritage de cette nouvelle hiérarchie. On décrit ensuite dans la suite de ce document les classes constitutives de cette hiérarchie.



On présente dans cette partie l'interface schématique des différentes classes de la hiérarchie des champs génériques et on précise les spécificités de chacune d'entre elles. Les informations relatives à la syntaxe de spécification de chacun des champs génériques dans un jeu de données sont pour leur part fournies aux paragraphes 3.2.2 et 3.2.3.

3.1.1.1 La classe de base : Champ_Generique_base

La classe Champ_Generique_base est la classe de base des champs dédiés au post-traitement. Elle constitue la classe mère d'une classe destinée à encapsuler un champ discret du problème, et de classes « operateurs » ayant pour rôle de réaliser des actions sur celui-ci. On présente l'interface de la classe de façon schématique, le détail de celle-ci étant fourni dans l'annexe 2. Cette classe est sous Kernel/Framework, ses classes filles sont elles sous /Kernel/Champs.

```
enum Entity  { NODE, SEGMENT, FACE, ELEMENT }
class Champ_Generique_base : public Objet_U
{
  declare_base(Champ_Generique_base)
public:
  virtual const Noms   get_property(const Motcle & query) const;
  virtual Entity  get_localisation(const entier index = -1) const;
  virtual double get_time() const;
  virtual const Domaine&    get_ref_domain() const;
  virtual const Zone_dis_base&  get_ref_zone_dis_base() const;
  virtual void completer(const Postraitement_base& post) = 0;
  virtual void  mettre_a_jour(double temps) = 0;
  virtual const Champ_base &  get_champ(Champ & espace_stockage) const = 0;
  virtual const Champ_Generique_base& get_champ_post(const Motcle& nom) const;
  virtual entier comprend_champ_post(const Motcle& identifiant) const;
  virtual entier get_info_type_post() const = 0;
  void fixer_identifiant_appel(const Nom& identifiant) ;
  void nommer(const Nom& nom);
  static inline entier composante(const Nom& nom_test,const Nom& nom, const Noms& composantes);

protected
  virtual Entree &  lire(const Motcle & motcle, Entree & is)

  Nom nom_post_;
  Nom identifiant_appel_;
};
```

La classe possède un attribut *nom_post_* pour identifier le champ et un attribut *identifiant_appel_* qui permet à un champ générique de savoir si une requête le renvoyant a été lancée à partir d'un nom de champ ou de l'une de ses composantes.

Une première série de méthodes est destinée à récupérer des informations spécifiques au champ générique.
-*get_property*() reconnaît les requêtes suivantes :
« nom » : rend le nom
« nom_cible » : rend l'identifiant du champ cible (nom ou composante du champ encapsulé)
« unites » : rend les unités
« composantes » : rend les composantes qui sont établies à partir du nom du champ et du numéro de composantes : nomp_post_+numero_compo
-*get_localisation*() rend la localisation du champ sous la forme d'un nouvel Enum Entity et a pour argument le numero du support (-1 par défaut)
-*get_time*() indique le temps du champ encapsulé
-*get_info_type_post*() indique si l'on post-traite un tableau ou un tenseur pour le champ générique concerné. La valeur rendue est 0 dans le cas d'un tableau et 1 pour un tenseur.
-*composante*() permet de déterminer si l'on manipule une composante du champ ou l'ensemble de ses composantes.

Une seconde série de méthodes donnent accès à des informations géométriques ou de discrétisation.
-*get_ref_domain*() donne accès au domaine de post-traitement du champ.
-*get_ref_zone_dis_base*() donne accès à la zone discrétisée associée.
La série de méthodes suivante a pour but de réaliser des actions sur le champ générique.
-*completer*() complète l'opérateur si le champ en porte un et nomme ce champ par défaut s'il constitue une source pour un autre champ. Dans ce cas, cette instruction sera réalisée par la méthode *nommer_source*() déclarée et codée dans les classes dérivées.
-*mettre_a_jour*() disponible en vue de la mise à jour d'opérateurs qui présentent une dépendance au temps tels que les opérateurs statistiques.

L'estimation des valeurs du champ à post-traiter est réalisée par la méthode *get_champ*() nécessitant un champ constituant un espace de stockage comme argument. L'implémentation de cette méthode dans les classes dérivées est du type :

```
const Champ_base& Champ_Post… ::get_champ(Champ& espace_stockage) const
{
  Champ_generique& source = get_source(0) ;
  const Zone_dis_base& zone_dis = …
  entier nb_composantes = …
  entier nb_ddl = …
  -construction de « type_champ »

  espace_stockage.typer(type_champ) ;
  espace_stockage.associer_zone_dis_base(zone_dis) ;
  espace_stockage.fixer_nb_comp(nb_composantes) ;
  espace_stockage.valeurs()= « operateur ».calculer(source.valeurs()) ;
  espace_stockage.valeurs().echange_espace_virtuel() ;
  return espace_stockage.valeur() ;
}
```

Le nom, les unités et les composantes ne sont pas fixées pour l'espace de stockage car ils sont accessibles par la méthode *get_property*().

Enfin des méthodes de requête du champ *get_champ_post*() et d'identification de celui-ci *comprend_champ_post*() sont disponibles. Le principe de requête est décrit au paragraphe 3.1.2.2.
Remarquons que les méthodes déclarées dans cette classe sont pour la plupart implémentées de façon à renvoyer un message d'erreur car ces méthodes sont surchargées dans les classes filles de Champ_Generique_base.

3.1.1.2 Encapsulation d'un champ volumique discret: Champ_Post_refChamp

La classe Champ_Post_refChamp encapsule une référence à un champ discret volumique de Trio_U. L'interface schématique de la classe est la suivante :

```
class Champ_Post_refChamp : public Champ_Generique_base
{
  declare_instanciable(Champ_Post_refChamp)
 public:

  virtual const Noms  get_property(const Motcle & query) const;
  virtual Entity  get_localisation(const entier index = -1) const;
  double get_time() const;
  virtual const Domaine&  get_ref_domain() const;
  virtual const Zone_dis_base&  get_ref_zone_dis_base() const;
  virtual const Champ_base& get_ref_champ_base() const;
  virtual void completer(const Postraitement_base& post);
  virtual void  mettre_a_jour(double temps);
```

```
  virtual const Champ_base& get_champ(Champ & espace_stockage) const;
  void nommer_source();
  entier get_info_type_post() const;

protected:

  virtual Entree &  lire(const Motcle & motcle, Entree & is);

  REF(Champ_base) ref_champ_;
  Nom nom_champ_;
  Nom nom_pb_;
};
```

L'encapsulation d'un champ discret du problème est réalisée à travers l'attribut *ref_champ_*. Ce champ discret constitue la source « cible » pour les divers « operateurs » de post-traitement qui lui sont éventuellement appliqués.
Les attributs *nom_pb_* et *nom_champ_* sont remplis à la lecture du champ générique. Ils indiquent respectivement le nom du problème auquel appartient le champ discret et l'identifiant de ce dernier (nom ou composante).
On décrit maintenant les principales méthodes qui concernent la lecture du champ, l'accès à ses informations spécifiques ainsi qu'à des informations géométriques et de discrétisation, et le calcul des valeurs à post-traiter.

-*lire*() reconnaît le mot clé « Pb_champ » qui déclenche la lecture de *nom_pb_* et *nom_champ_* et par suite l'initialisation de *ref_champ_*.
-*get_ref_domain*() ou *get_ref_zone_dis_base*() exploitent la référence vers un champ du problème pour rendre respectivement le domaine ou la zone discrétisée.
-*get_property*() assure les requêtes présentées dans la description de la classe Champ_Generique_base. Notons l'appel à la méthode corriger_unite_compo() pour la requête « unites ».
-*get_time*() rend le temps du champ encapsulé
-*get_info_type_post*() rend la valeur 0, ce qui signifie que pour ce type de champ on post-traite un tableau.
-*get_champ*() est ici particulière car elle n'exploite pas d'espace de stockage. En effet la méthode rend simplement le champ discret auquel elle fait référence par l'intermédiaire de *get_ref_champ_base*(). Toutefois une actualisation du champ encapsulé est faite dans le cas où celui-ci est un « champ calculé ».

3.1.1.3 Généricité des opérations de post-traitement : Champ_Post_de_Champs_post

La classe Champ_Post_de_Champs_Post est la classe de base des champs « operateurs » de post-traitement qui portent eux-mêmes un ou plusieurs champs sources. Cette structure permet de combiner l'action des différents « opérateurs » tels que l'interpolation, les statistiques …
L'interface schématique de la classe est la suivante :

```
class Champ_Post_de_Champs_post : public Champ_Generique_base
{
  declare_base(Champ_Post_de_Champs_post)
 public:

  virtual entier sauvegarder(Sortie & os) const;
  virtual entier reprendre(Entree & is);
  virtual void completer(const Postraitement_base& post);
  virtual void   mettre_a_jour(double temps);
  virtual const Noms  get_property(const Motcle & query) const;
  virtual Entity  get_localisation(const entier index = -1) const;
  double  get_time() const;
  virtual const Domaine&    get_ref_domain() const;
  virtual const Zone_dis_base&  get_ref_zone_dis_base() const;
```

```
 void nommer_sources();
 virtual void nommer_source();
 virtual entier get_info_type_post() const;
 virtual const Champ_Generique_base& get_champ_post(const Motcle& nom) const;
 virtual entier comprend_champ_post(const Motcle& identifiant) const;

protected:
 Entree & lire(const Motcle & motcle, Entree & is);

 Nom nom_source_ref_;
 REF(Champ_Generique_base) source_ref;

private:
 LIST(Champ_Generique) sources_;
};
```

L'attribut *sources_*, privé de la classe, constitue une liste de champs de type Champ_Generique. Un champ source porte éventuellement une ou plusieurs sources, et ainsi de suite. Cette structure récursive permet d'exploiter la généricité des champs « opérateurs » en combinant leurs actions.
Les attributs *nom_source_ref_* et *source_ref_* sont utilisés pour créer une référence vers un champ générique déjà défini, *nom_source_ref_* désignant l'identifiant de ce dernier et *source_ref_* constituant la référence.

On présente maintenant les principales méthodes qui concernent la lecture du champ, l'accès à ses informations spécifiques et des actions sur celui-ci.

-*lire*() reconnaît :

-le mot clé « source » qui déclenche la lecture d'un champ générique, cette lecture se poursuivant récursivement jusqu'à la lecture d'un champ de type Champ_Post_refChamp portant sur un champ discret « cible ».

-le mot clé « source_reference » qui déclenche la lecture de *nom_source_ref_,* ce qui permettra de construire une source constituant une référence à un champ générique déjà défini.

Les méthodes d'accès à des informations que l'on désigne ici de façon générique par *get_info*(), fonctionnent de façon récursive. La recherche d'information est déléguée à une source « de plus bas niveau » jusqu'à ce que l'une d'entre elles soit capable de la fournir.
Une méthode de requête d'information aura classiquement la forme suivante :

```
….Champ_Post_de_Champs_Post ::get_info()
{
 return get_source(0).get_info(…)
}
```

Les méthodes qui effectuent une action sur le champ, comme par exemple *completer*(), appliqueront l'action récursivement à chacune de leurs sources.

```
…Champ_Post_de_Champ_Post ::completer()
{
 for (entier i=0 ; inb_sources; i++)
 source(i).completer()
}
```

Toutefois chaque classe dérivant de la classe Champ_Post_de_Champs_post est susceptible de surcharger une méthode pour que celle-ci réponde de manière spécifique à la requête d'information ou d'action.
On remarque que la classe Champ_Post_de_Champ_Post ne possède pas de méthode *get_champ*() car celle-ci est spécifique à chaque champ générique. Elle est par conséquent surchargée dans chacune de ses classes filles instanciables. En outre, notons que cette classe est dotée de méthodes complémentaires spécifiques à la gestion des statistiques, que nous ne décrivons pas dans cette partie.

3.1.1.4 Les champs « opérateurs » de post-traitement

3.1.1.4.1 Opérateur d'interpolation : Champ_Post_Interpolation

Un champ générique de type Champ_Post_Interpolation a pour fonction d'effectuer une interpolation des valeurs de son champ source. L'interpolation peut être effectuée aux éléments ou aux sommets du domaine de calcul ou d'un domaine spécifié par l'utilisateur.
L'interface schématique de la classe est la suivante :

```
class Champ_Post_Interpolation : public Champ_Post_de_Champs_post
{
  declare_instanciable(Champ_Post_Interpolation)
 public :
  virtual const Noms  get_property(const Motcle & query) const;
  virtual Entree &  lire(const Motcle & motcle, Entree & is);
  void nommer_source();
  virtual const Champ_base&  get_champ(Champ& espace_stockage) const;
  virtual Entity  get_localisation(const entier index = -1) const;
  virtual const Domaine &  get_ref_domain() const;
  void completer(const Postraitement_base& post);
  const Zone_dis_base& get_ref_zone_dis_base() const;
  void discretiser_domaine(const Postraitement_base& post);
  const Noms& fixer_noms_compo(const Noms& noms);
  //L attribut compo_ n est rempli que pour les Champ_Post_Interpolation
  //crees par macro et cela afin de reproduire les noms de composantes dans les lml
  Noms compo_;

private:
 Motcle   localisation_; // elem, som, etc...
 REF(Domaine) domaine_;  //domaine sur lequel on interpole le champ (domaine natif si reference nulle)
 Domaine_dis le_dom_dis;
};
```

L'attribut *localisation_* indique la localisation où est effectuée l'interpolation (éléments ou sommets). L'attribut *domaine_* fait une référence au domaine de post-traitement du champ si celui-ci est différent du domaine de calcul. Dans ce cas l'attribut *le_dom_dis* représente le domaine discrétisé.

On décrit maintenant les principales méthodes qui concernent la lecture du champ, l'accès à ses informations spécifiques, l'opération de complément et le calcul des valeurs à post-traiter.
-*lire*() reconnaît les mots clé :

- « localisation » qui déclenche la lecture de *localisation_*
- « domaine » optionnel qui déclenche l'initialisation de *domaine._*

-*get_ref_domain*() et *get_ref_zone_dis_base*() sont surchargées pour exploiter la référence *domaine_* dans le cas où celle-ci est initialisée.
-*completer*() réalise la discrétisation du domaine de post-traitement si celui-ci n'est pas le domaine de calcul et par conséquent l'attribut *le_dom_dis* est rempli.
-*get_champ*() est construit sur le modèle présenté dans la description de la classe Champ_Generique_base.

Les remarques suivantes sont à formuler pour cette méthode :
-Dans le cas où l'interpolation est réalisée aux sommets, la méthode fixer_nb_valeurs_nodales() n'est pas appelée pour construire l'espace de stockage car actuellement celle-ci n'est pas codée pour la classe Champ_P1_impl. Par conséquent, l'espace de stockage ne dispose alors pas d'un espace virtuel.

-Dans le cas où l'on procède à l'interpolation d'une composante d'un champ source, l'attribut *identifiant_appel_* indique la composante à interpoler. La méthode *get_champ*() construit alors un espace de stockage possédant le même nombre de composantes que la source elle-même et attribue des valeurs nulles aux composantes non traitées. Cela permet par la suite de pouvoir traiter une sonde de la composante interpolée.

3.1.1.4.2 Opérateur de transformation : Champ_Post_Operateur

Un champ générique de type Champ_Post_Operateur a pour rôle d'appliquer un opérateur aux valeurs d'un champ source. Dans la présente version, les opérateurs disponibles sont les opérateurs gradient et divergence. Les classes concernées héritent de la classe de base Champ_Post_Operateur_base qui hérite elle-même de la classe Champ_Post_de_Champs_post.
L'interface schématique de la classe a la forme suivante :

```
class Champ_Post_Operateur_… : public Champ_Post_Operateur_base
{
   declare_instanciable(Champ_Post_Operateur_…)

  public:

  virtual const Noms get_property(const Motcle & query) const;
  virtual Entity  get_localisation(const entier index = -1) const;
  virtual const Champ_base&  get_champ(Champ& espace_stockage) const;
  void completer(const Postraitement_base& post);
  void nommer_source();

 protected:

  Operateur Op_;
};
```

L'attribut *Op_* est un opérateur gradient ou divergence. Cet attribut est typé et complété dans la méthode *completer*() en fonction du champ source qui est considéré.

-*get_property*() est surchargé pour répondre à la requête « unites ».
-*get_champ*() est construit sur le modèle présenté dans la description de la classe Champ_Generique_base.
Les remarques suivantes sont à formuler pour ce type de champ générique :
-Les champs de type Champ_Post_Operateur_Gradient et Champ_Post_Operateur_Divergence sont applicables uniquement si l'opérateur requis existe pour la discrétisation du champ source considéré. D'autre part le champ source doit posséder des conditions limites.
-En pratique, les champs génériques de type Champ_Post_Operateur_Gradient et Champ_Post_Operateur_Divergence peuvent être appliqués respectivement au champ de pression et de vitesse indépendamment de la discrétisation. La suppression des champs discrets spécifiques gradient_pression et divergence_U est par conséquent envisageable. En outre la suppression du champ discret gradient_temperature, disponible pour une discrétisation de type VEF, est elle aussi possible. En effet le développement d'un nouvel opérateur Op_grad_P1NC_to_P0 permet de généraliser le post-traitement du gradient d'un champ scalaire de type P1NC, en reproduisant celui du gradient de température et en l'étendant par exemple au gradient de concentration ou de fraction massique.

3.1.1.4.3 Les statistiques temporelles : Champ_Post_Statistiques

Un champ générique de type Champ_Post_Statistiques a pour rôle de réaliser une statistique temporelle sur les valeurs de son (ou ses) champ source. Les champs statistiques développés sont destinés à calculer une moyenne, un écart type ou des corrélations. Les classes concernées héritent de la classe de base Champ_Post_Statistiques_base qui hérite elle-même de la classe Champ_Post_de_Champs_post.
L'interface schématique d'une classe Champ_Post_Statistiques a la forme suivante :

```
class Champ_Post_Statistiques_… : public Champ_Post_Statistiques_base
{
  declare_instanciable(Champ_Post_Statistiques_…)

    public:

  virtual const Noms get_property(const Motcle & query) const;
  inline const double temps() const {  return Op_Moyenne_.integrale().temps(); };
  inline const Integrale_tps_Champ& integrale() const {  return Op_Moyenne_.integrale(); };
  void completer(const Postraitement_base& post);
  virtual const Champ_base&  get_champ(Champ& espace_stockage) const;
  void nommer_source();

 protected:

   Operateur_Statistique Op_stat_;
};
```

L'attribut *Op_stat_* est un opérateur statistique du type « Moyenne », « Ecart_Type » ou « Correlation ». Cet opérateur possède lui-même un champ « intégrale » pour réaliser l'intégration temporelle.
Les attributs *tstat_deb_* et *tstat_fin_* de la classe de base, désignent respectivement le temps de début et de fin d'intégration.

On décrit maintenant les principales méthodes qui concernent la lecture du champ, l'opération de complément, la mise à jour de l'opérateur, les opérations de sauvegarde-reprise et le calcul des valeurs à post-traiter.
-*lire*(), de la classe de base, reconnaît les mots clé « t_deb » et « t_fin ». Ces mots-clés déclenchent la lecture de *tstat_deb_* et *tstat_fin_*
-*mettre_a_jour()* assure la mise à jour de l'opérateur porté par le champ statistique.
-*completer*() réalise les opérations nécessaires pour compléter l'opérateur statistique.
-*sauvegarde*r() et *reprendre*(), de la classe de base, gèrent les sauvegarde et reprise du champ « intégrale » porté par l'opérateur statistique.
-*get_champ*() est construit sur le modèle présenté dans la description de Champ_Generique_base.
-*get_info_type_post*() rend la valeur 1 dans le cas où les sources d'un Champ_Post_Statistiques_Correlation constituent un tenseur et 0 sinon.

Notons que la déclaration d'un champ générique de type Champ_Post_Statistiques_Ecart_Type (ou Champ_Post_Statistiques_Correlation) portant sur une (ou des) source donnée implique d'avoir défini au préalable un champ de type Champ_Post_Statistiques_Moyenne sur la (ou les) même source.

3.1.1.4.4 Combinaison de champs : Champ_Post_Transformation

Un champ générique de type Champ_Post_Transformation a pour fonction de manipuler une combinaison dépendante de champs génériques sources et éventuellement des coordonnées de l'espace x, y, z et du temps t.
L'interface schématique de la classe a la forme suivante :

```
class Champ_Post_Transformation : public Champ_Post_de_Champs_post
{
  declare_instanciable(Champ_Post_Transformation)
   public:
```

```
  virtual Entree &  lire(const Motcle & motcle, Entree & is);
  virtual const Noms get_property(const Motcle & query) const;
  virtual const Champ_base&  get_champ(Champ& espace_stockage) const;
  void completer(const Postraitement_base& post);
  void nommer_source();

 protected:

  Noms les_fcts;                  //Contient l expression de la combinaison
  mutable VECT(Parser_U) fxyz; //Parser utilise pour evaluer la valeur prise par la combinaison
};
```

L'attribut *les_fcts* désigne la formule qui est à spécifier par l'utilisateur. Elle dépend du nom des champs génériques sources considérés et des variables x, y, z et t.
L'attribut *fxyz* est le parser qui permet d'évaluer la valeur prise par la formule spécifiée.

-*lire*() reconnaît le mot-clé « fonction » qui déclenche la lecture de la formule dépendante des noms des champs génériques et des variables x, y, z et t.
-*get_champ*() est construit sur le modèle présenté dans la description de la classe Champ_Generique_base.

3.1.1.4.5 Extraction de champs : Champ_Post_Extraction

Un champ générique de type Champ_Post_Extraction a pour fonction de réaliser l'extraction des valeurs d'un champ sur une frontiere du domaine de calcul.
L'interface schématique de la classe est la suivante :

```
class Champ_Post_Extraction : public Champ_Post_de_Champs_post
{
 declare_instanciable_sans_constructeur(Champ_Post_Extraction)
 public:
 Champ_Post_Extraction();
 virtual Entree &  lire(const Motcle & motcle, Entree & is);
 virtual Entity  get_localisation(const entier index = -1) const;
 virtual const Champ_base&  get_champ(Champ& espace_stockage) const;
 virtual const Noms    get_property(const Motcle & query) const;
 void nommer_source();

 void completer(const Postraitement_base& post);
 virtual const Domaine & get_ref_domain() const;
 virtual void get_copy_domain(Domaine &) const;
 const Zone_dis_base& get_ref_zone_dis_base() const;
 void discretiser_domaine(const Postraitement_base& post);

 protected :

 Nom dom_extrac_;            //Nom du domaine d extraction
 Nom nom_fr_;                //Nom de la frontiere sur laquelle on fait l extraction
 Nom methode_;               //Type de methode pour extraire ("trace" ou "champ_frontiere")
 REF(Domaine) domaine_;      //Reference sur le domaine d extraction
 Domaine_dis le_dom_dis;     //Le domaine discretise correspondant à domaine_.valeur()
                             //La discretisation de ce domaine n est pas faite actuellement
};
```

L'attribut *nom_fr_*désigne le nom de la frontière sur laquelle l'extraction doit etre réalisée. Les attributs *dom_extract_* et *domaine_* désignent respectivement le nom du domaine construit pour réaliser l'extraction du champ et une référence à celui-ci. L'attribut *le_dom_dis* est à considérer si l'on construit une zone discrétisée pour le domaine construit. L'attribut *methode_* est destiné à sélectionner le type d'extraction choisie par l'utilisateur.

-*lire()* reconnaît les mots-clés suivants :
-« domaine » déclenche la lecture du nom du domaine d'extraction
-« nom_frontiere » déclenche la lecture du nom de la frontière considérée pour indiquer la partie de domaine à extraire
-« methode » déclenche la lecture du type de méthode d'extraction, *trace* par défaut ou *champ_frontiere* pour extraire les valeurs du champ le_champ_front.
-*get_localisation*() renvoie une localisation « ELEMENT » car le champ extrait est localisé aux éléments du domaine construit.
-*completer*() réalise la création du domaine d'extraction.
-*get_champ*() est construit sur le modèle présenté dans la description de la classe Champ_Generique_base.

Précisons que la méthode *trace*() n'est actuellement développée que pour les champs inconnus du problème et par conséqent celle-ci ne peut etre appliquée à d'autres champs du type Champ_Fonc … D'autre part la seconde option d'extraction qui rend les valeurs du champ *le_champ_front* n'est valable que pour les champs inconnus du problème auxquels on applique une condition limite utilisant « ce champ frontière ».
En résumé, l'extraction des valeurs de champ n'est actuellement active que pour les champs inconnus du problème. En outre, le post-traitement au format meshtv du domaine extrait présente un arrêt dans le cas où celui-ci est constitué d'éléments triangulaire.

3.1.1.4.6 Réduction de la dimension d'un champ : Champ_Post_Reduction_0D

Un champ de type Champ_Post_Reduction_0D a pour rôle de construire un champ uniforme en espace à partir des valeurs d'un champ discret en fonction d'une spécification de l'utilisateur. Ce dernier peut réduire le champ à sa valeur minimum, maximum ou moyenne.
L'interface schématique de la classe est la suivante :

```
class Champ_Post_Reduction_0D : public Champ_Post_de_Champs_post
{
  declare_instanciable(Champ_Post_Reduction_0D)

  public:

 virtual Entree & lire(const Motcle & motcle, Entree & is);
 virtual const Noms get_property(const Motcle & query) const;
 virtual const Champ_base&  get_champ(Champ& espace_stockage) const;
 void nommer_source();
 void extraire(double& val_extraites, const DoubleVect& val_source) const;

 protected:

  Nom methode_; //Type de reduction : min, max ou moyenne
} ;
```

L'attribut *methode_* est destiné à spécifier le type de réduction à réaliser, celle-ci est à spécifier par l'utilisateur.

-*lire*() reconnaît le mot-clé « methode » qui déclenche la lecture de l'attribut *methode_*.
-*get_champ*() est construit sur le modèle présenté dans le description de la classe Champ_Generique_base.
-*extraire*() est utilisé dans get_champ() pour récupérer la valeur min, max ou moyenne du (ou des) vecteur constituant le tableau de valeurs du champ source.

Notons que la nature du champ réduit est identique à celle du champ source.

3.1.2 Evolution du module Post-Statistiques-Sondes

3.1.2.1 Conception de la classe Postraitement

3.1.2.1.1 Les nouveaux attributs de la classe

Les nouveaux attributs de la classe Post-traitement sont :

*-champs_post_complet_* de type Liste_Champ_Generique. Une classe Liste_Champ_Generique spécifique a été créée. Elle constitue une liste de Champ_Generique et possède des méthodes complémentaires. L'attribut *champs_post_complet_* contient tous les champs génériques du post-traitement y compris ceux qui sont uniquement créés pour les sondes.
*-noms_champs_a_post_* qui constitue une liste de noms identifiant chacun un champ à écrire.
La classe Postraitement est aussi dotée d'attributs spécifiques aux statistiques qui étaient précédemment portés par une liste d'opérateurs statistiques. Ils sont essentiellement utilisés pour la gestion des sauvegarde-reprise et la mise à jour.

3.1.2.1.2 Les nouvelles méthodes de la classe

On décrit maintenant les nouvelles méthodes de la classe Postraitement ainsi que la révision de méthodes déjà existantes précédemment.

La méthode *lire_champs_operateur*() est destinée à lire un champ générique spécifié avec la nouvelle syntaxe, lancer des opérations de complément et ajouter le champ dans la liste *champs_post_complet_*. En résumé, le contenu de la méthode est le suivant :

void Post::lire_champs_operateur(Entree& is)
{
Champ_Generique champ
is>>nom *// lecture du nom*
is>>champ *// lecture du champ*
champ->nommer(nom) *// le champ est nommé*
//ajout du champ à la liste des champ génériques
champ_a_completer=champs_post_complet_.add_if_not(champ)
champ_a_completer->completer(post) // opération de complément (les sources sont nommées ...)
}

Le nom du champ lu sera ajouté à la liste *noms_champs_a_post_* uniquement si ce nom apparaît dans le bloc « champs » du jeu de données.

D'autre part, des méthodes *creer_champ_post*() et *creer_champ_post_stat*() sont développées pour créer un champ générique à partir d'une requête formulée avec l'ancienne syntaxe. En résumé, le contenu de cette méthode *creer_champ_post*() est le suivant :

void Post::creer_champ_post(const Motcle& id_cible, const Motcle& type_loc,Entree& is)
{
Champ_Generique champ
-création d'une chaîne de caractères (ajout) correspondant au type de champ à créer.
soit Cas 1 : Création d'un champ de type Champ_Post_Interpolation
Nom ajout = « Champ_Post_Interpolation { localisation »+ type_loc+
« source Champ_Post_refChamp { Pb_champ »+ nom_pb + « »+ id_cible + « } } »

soit Cas 2 : Création d'un champ de type Champ_Post_refChamp
Nom ajout = « Champ_Post_refChamp { Pb_champ »+mon_pb+ « » + Id_cible + " }";

```
  Entree_complete s_complete(ajout,is);
  s_complete>>champ ; //lecture du champ

  nom = nom_champ_cible+ « _ »+type_loc+ « _ »+nom_domaine //construction du nom du champ
  champ->nommer(nom)                                        //le champ est nommé
  champ->fixer_noms_compo(compo)    //nom des composnates fixees pour testde non-régression
  champ_a_completer=champs_post_complet_.add_if_not(champ) //ajout à la liste
  champ_a_completer->completer(post)                        //opération de complément
  noms_champs_a_post_.add(identifiant)          //ajout de l'identifiant si le champ est à post-traiter
}
```

Ces méthodes « macros » remplissent l'attribut composantes_ du champ générique créé, dans le but de reproduire la syntaxe des composantes utilisée dans les fichiers de référence d'extension .lml. Ainsi la non-régression du logiciel peut être vérifiée. L'appel des « macros » est lancé par la méthode *lire_champs_a_postraiter*() si l'ancienne syntaxe de requête de champs dans les blocs « champs » ou « statistiques » (ex : vitesse elem dom) est utilisé.

L'appel de la méthode a la forme suivante : creer_champ_post(id_cible,type_loc,is)
-id_cible : nom du champ cible ou l'une de ses composantes (ex : vitesse, vitessex …)
-type_loc : localisation de l'interpolation « elem » ou « som » si création d'un champ de type Champ_Post_Interpolation, « natif » si création d'un champ de type Champ_Post_refChamp.
-is : fichier de lecture

La méthode creer_champ_post_stat() est faite sur le même modèle, son appel a la forme suivante :
creer_champ_post_stat(id_cible1,type_loc,type_op,id_cible2,t_deb,t_fin,is)
-id_cible1 : identifiant du champ cible (ou premier s'il y a deux sources)
-type_loc : localisation de l'interpolation
-type_op : type d'opérateur statistique (« Moyenne », « Ecart_Type », « Correlation »)
-id_cible2 : identifiant du second champ cible (s'il y en a deux)
-t_deb : temps de debut des statistiques
-t_fin : temps de fin des statistiques
-is : fichier de lecture

La classe Postraitement est dotée d'une méthode *comprend_champ_post()* qui renvoie 1 si l'identifiant passé en paramètre correspond à celui d'un champ générique contenu dans la liste *champ_post_complet_* ou à celui de l'une de ses sources.
Une méthode *get_champ_post*() est aussi développée pour renvoyer un champ de la liste *champ_post_complet_* si l'identifiant passé en paramètre est reconnu.

En outre, des méthodes de la classe sont révisées pour exploiter la nouvelle conception du framework. En particulier, la méthode *postraiter_champs*() exploite un champ générique récupéré par une requête lancée à partir de son identifiant. Ce champ générique permet de calculer le tableau de valeurs à écrire et d'accéder aux informations nécessaires à son post-traitement telles que le nom du champ, de ses composantes ...
En résumé, le contenu de la méthode est le suivant :

```
entier Post::postraiter_champs()
{
 LIST_CURSEUR(Nom) curseur = noms_champ_a_post_ // parcours de la liste d'identifiant
 while (curseur)
 {
  //requête pour récupérer le champ générique
   const Champ_Generique_base& champ = mon_pb.get_champ_post(curseur.valeur())

  -calcul du  tableau de valeurs à écrire
   Champ espace_stockage
   const Champ_base& champ_ecriture=champ.get_champ(espace_stockage)
   const DoubleTab& valeurs_post =champ_ecriture.valeurs()
```

-les informations nécessaires au post-traitement sont récupérées
localisation, nom_post, unites, composantes, temps sont obtenus par appel à :
champ.get_property(« query ») (ex : champ.get_property(« nom »)

-détection post-traitement d'une composante
int ncomp = Champ_Generique_base ::composante(curseur.valeur(),nom_post,composantes)

-écriture du tableau de valeurs sur fichier
postraiter(domaine,unites,composantes,ncomp,temps,...,curseur.valeur(),localisation,valeurs_post)
}
}

Notons que les méthodes *postraiter_statistiques*() et *postraiter_statistiques_en_serie*() sont supprimées car les champs statistiques sont des champs génériques traités par la méthode *postraiter_champs*().
La méthode *postraiter*() qui réalise l'écriture sur fichier des valeurs du champ considéré est maintenant portée par la classe Postraitement et non plus par la classe Champ_base. Cette méthode distingue l'écriture d'un tableau de valeurs, ou de l'une de ses composantes, de celle d'un tenseur. Dans ce dernier cas, on post-traite les composantes les unes après les autres. Le cas d'un tableau ou vecteur est traité par la méthode *postraiter_tableau*() alors que celui d'un tenseur est traité par la méthode *postraiter_tenseur*().

La méthode *mettre_a_jour*() effectue la mise à jour des champs génériques de la liste *champs_post_complet_* ainsi que celle de leurs sources. Pratiquement, cette mise à jour concerne les opérateurs des champs statistiques. Les méthodes *sauvegarder*() et *reprendre*() sont modifiées pour parcourir la liste *champs_post_complet_* et sont adaptées pour la gestion des sauvegarde-reprise des opérateurs statistiques.
Enfin les méthodes *recherche_op_stat_moyenne*() et *champ_fonc*() sont révisées pour effectuer une requête sur les champs de la liste *champs_post_complet_*.

3.1.2.2 Requête d'un champ de post-traitement

Le problème doit pouvoir répondre à une requête lancée sur un champ générique du post-traitement à partir d'un identifiant. Dans cette optique, la classe Probleme_base est dotée d'une méthode *get_champ_post*(). La méthode délègue la requête à chacun des post-traitements du problème. Pour un post-traitement donné, la liste *champ_post_complet_* est consultée. En premier lieu, un champ générique de la liste est testé et si la recherche échoue, elle est poursuivie récursivement dans ses sources. Si le champ recherché est identifié, une référence à celui-ci est renvoyée, sinon la recherche est menée sur le champ suivant de la liste. Afin de rendre ce principe opérationnel, une méthode *get_champ_post*() est développée dans les classes suivantes : Probleme_base, Postraitement, Champ_Generique_base et Champ_Post_de_Champs_post.
Remarquons que dans le cas où un champ générique serait identifié, la méthode *get_champ_post*() fixe la valeur de son attribut *identifiant_appel_*. Ainsi l'objet qui manipule ensuite le champ générique renvoyé est apte à déterminer s'il doit traiter une composante ou le champ complet. Cet aspect concerne en particulier la méthode *get_champ*() de la classe Champ_Post_Interpolation.

3.1.2.3 Evolution des opérateurs statistiques et des sondes

3.1.2.3.1 Sonde d'un champ générique

Afin de généraliser la création d'une sonde à un champ générique, l'attribut *mon_champ* de la classe Sonde est modifié. Il constitue maintenant une référence vers un champ de type Champ_Generique_base et non plus Champ_base. Par conséquent, la requête d'une sonde sur un champ discret du problème déclenche la création d'un champ générique de type Champ_Post_refChamp par appel à la « macro » *creer_champ_post*(). Le champ créé est ajouté à la *liste champs_post_complet_* et la référence *mon_champ* peut être initialisée.
Les méthodes *ReadOn*() et *completer*() sont révisées :

Entree& Sonde ::ReadOn(Entree& is)
{
…
//On récupère le champ discret à partir de l'identifiant spécifié dans le jeu de données dans le but //d'accéder à son nom
REF(Champ_base) champ = pb.get_champ(nom_champ_lu_)
//On crée le champ générique de type Champ_Post_refChamp par appel à creer_champ_post()
mon_post.creer_champ(champ.le_nom(), « natif »,is)
…
}

void Sonde ::completer()
{
-On fixe l'identifiant pour lancer la requête sur un champ générique et initialiser mon_champ
Si nom_champ_lu_ est -le nom d'un champ du problème
nom_champ_ref = nom_champ_base+ « natif »+nom_dom
-composante
nom_champ_ref = nom_champ_base+ « natif »+nom_dom+num_compo
sinon
*nom_champ_ref = nom_champ_lu_*

//Initialisation de la référence
mon_champ = pb.get_champ_post(nom_champ_ref)

-On détermine s'il s'agit d'une composante par la méthode composante.
nom_champ = mon_champ.get_property(« nom »)
composantes = mon_champ.get_property(« composantes »)
ncomp = Champ_Generique_base ::composante(nom_champ_ref,nom_champ,composantes)
…
}

3.1.2.3.2 Opérateur statistique pour un champ générique

Les opérateurs statistiques dont sont dotés les champs génériques statistiques sont les opérateurs « Moyenne », « Ecart_Type » et « Correlation » déjà existants dans le logiciel. Toutefois, afin de généraliser les champs auxquels peuvent s'appliquer les opérateurs statistiques, les attributs *mon_champ* de la classe Integrale_tps_champ et *mon_scond_champ* de la classe Integrale_tps_produit_champ sont modifiés. Ils constituent maintenant une référence vers un champ de type Champ_Generique_base et non plus Champ_base. On procède à leur initialisation par une requête lancée à partir de l'identifiant du champ générique auquel l'opérateur doit faire référence. En conséquence, les méthodes *associer…*() des opérateurs statistiques sont modifiées.

3.2 EVOLUTION DE LA SYNTAXE DU JEU DE DONNEES

3.2.1 Déclaration des champs génériques et requête pour post-traitement

La syntaxe du bloc de post-traitement dans le jeu de données se présente désormais de la manière suivante :

Postraitement
{
*[**Format** …]*
*[**Sondes** { …]]*
*[**Domaine** dom_post]*

*[**definition_champs** {… }]*
__champs dt_post__ dt {…}
*[**Statistiques** {… }]*
}

Le mot-clé « definition_champ » déclenche la lecture d'un nouveau bloc réservé à la déclaration des champs génériques. Chacun des champs lus dans ce bloc est ajouté à la liste *champ_post_complet_* de l'objet de post-traitement correspondant. On rappelle que les champs contenus dans cette liste ne sont pas systématiquement considérés pour une écriture complète des valeurs, mais peuvent être définis uniquement pour la construction d'une sonde, ou pour constituer une source d'un autre champ générique. Les requêtes de champ à écrire sont toujours spécifiées dans les blocs annoncés respectivement par les mots-clés « champs » et « statistiques ». L'ancienne syntaxe de requête reste valable. Toutefois soulignons que celle-ci ne concerne que l'interpolation de champs discrets du problème ou de statistiques portant sur ces champs. La requête d'un champ spécifié dans le bloc « definition_champs » nécessite simplement d'indiquer son nom dans le bloc « champs».
On présente ci-dessous un exemple de spécification d'un objet de post-traitement dans un jeu de données pour illustrer la déclaration de champs génériques et préciser la formulation des requêtes d'écriture.

```
Postraitement
{
        Sondes
        {
                sonde_pression pression periode 0.01 points 1 0.0045 0.0045
                sonde_vitesse vitesse periode 0.01 segment 10 0.0005 0.005 0.0095 0.005
                sonde_vit vitesse periode 0.01 segment 11 0.0005 0. 0.0005 0.01
                sonde_temp temperature periode 0.01 segment 21 0. 0.0055 0.01 0.0055
                sonde_moyenne_vitesse Moyenne_vitesse_som_dom periode 0.00001 points 1 0.0045 0.0045
                sonde_seg_moy_vit Moyenne_vitesse_som_dom periode 0.01 segment 10 0.0005 0.005 0.0095 0.005
        }

        Definition_champs
        {

                Moyenne_temperature_som_dom Champ_Post_Interpolation {
                        localisation som
                        source Champ_Post_Statistiques_Moyenne
                        {
                         t_deb 0. t_fin 0.011
                source Champ_Post_refChamp { Pb_champ pb temperature }
                        }
                }

                Moyenne_vitesse_som_dom Champ_Post_Interpolation {
                        localisation som
                        source Champ_Post_Statistiques_Moyenne
                        {
                         t_deb 0. t_fin 0.011
                source Champ_Post_refChamp { Pb_champ pb vitesse }
                        }
                }

        }
Champs dt_post 0.01
         {
                        pression elem
                        pression som
                        vitesse som
                        temperature som
                        vorticite elem
                        Moyenne_vitesse_som_dom
                        Moyenne_temperature_som_dom
        }
        Statistiques dt_post 0.01
         {
                 t_deb 0. t_fin 0.011
                 moyenne pression
```

```
                moyenne vorticite
                ecart_type vitesse
                ecart_type temperature
                ecart_type pression
                ecart_type vorticite
        }
}
```

Dans cet exemple, les champs génériques de noms Moyenne_temperature_som_dom et Moyenne_vitesse_som_dom sont déclarés dans le bloc « definition_champs » et leur requête d'écriture est réalisée dans le bloc « champs » en précisant leur nom. En revanche, les requêtes de post-traitement des champs de pression, vitesse, temperature et vorticité sont effectuées par l'ancienne syntaxe dans le bloc « champs » ainsi que des statistiques temporelles dans le bloc « statistiques ».
Notons que le logiciel distingue la requête concernant un champ discret du problème pour lequel on ne précise pas la localisation (elle est alors fixée par défaut à « som ») d'un champ générique déclaré dans le bloc « definition_champs ».
Au sujet des statistiques, notons que l'on peut toujours effectuer des statistiques en série. Toutefois l'utilisation du mot-clé « Statistiques_en_serie » est nécessaire même si le champ statistique a été déclaré dans le bloc « definition_champs ». En outre, il faut spécifier dans ce bloc des valeurs de début et de fin des statistiques en plus de la période d'intégration.

3.2.2 Syntaxe de spécification des différents champs génériques

De façon générale, pour spécifier un champ générique, on indique :
-le nom du champ (au choix de l'utilisateur)
-le type du champ
-les paramètres spécifiques au champ
-sa (ses) source(s)

On précise ici la syntaxe à respecter pour chacun des champs génériques disponibles actuellement.

nom_champ Champ_Post_refChamp { **Pb_champ** *nom_pb nom_champ* }
-*nom_pb* désigne le nom du problème auquel appartient le champ discret
-*nom_champ* désigne l'identifiant du champ cible(nom ou composante)

nom_champ Champ_Post_Interpolation { [**domaine** nom_dom]
localisation *type_loc*
source Champ_Post … { … }
}
-*nom_dom* nom du domaine sur lequel est réalisée l'interpolation (domaine de calcul par défaut)
-*type_loc* indique la localisation où est faite l'interpolation (« elem » ou « som »)

nom_champ Champ_Post_Operateur_Gradient { **source** Champ_Post.. { … } }
nom_champ Champ_Post_Operateur_Divergence { **source** Champ_Post.. { … } }

nom_champ Champ_Post_Statistiques_Moyenne { **t_deb** *val1* **t_fin** *val2*
source Champ_Post… {…} }
nom_champ Champ_Post_Statistiques_Ecart_Type { **t_deb** *val1* **t_fin** *val2*
source Champ_Post… {…} }

nom_champ Champ_Post_Statistiques_Correlation { **t_deb** *val1* **t_fin** *val2*
source Champ_Post… { … }
source Champ_Post… { … }
}
-*val1* désigne la valeur du temps de début d'intégration temporelle
-*val2* désigne la valeur du temps de fin d'intégration temporelle

nom_champ Champ_Post_Transformation { **fonction** *formule*
source Champ_Post … { … }
source Champ_Post … { … }
…
}
-*formule* expression de la combinaison basée sur le nom des champs sources, de x,y,z et t.

nom_champ Champ_Post_Extraction { **domaine** *nom_dom* **nom_frontiere** *nom_fr*
[**methode** *type_methode*]
source Champ_Post… { … }
}
-*nom_dom* nom du domaine d'extraction (celui-ci doit être déclaré dans le jeu de données)
-*nom_fr* nom de la frontière sur laquelle on va extraire les valeurs du champ
-*type_methode* désigne la méthode d'extraction (*trace* par défaut ou *champ_frontiere*)

nom_champ Champ_Post_Reduction_0D { **methode** *type_methode*
source Champ_Post…{ … }
}
-*type_methode* désigne la méthode de réduction (*min*, *max* ou *moyenne*)

On rappelle d'autre part que pour un champ de type de Champ_Post_de_Champ_Post, il est possible de spécifier une source constituée par un champ déjà déclaré en utilisant le mot-clé source_reference :
nom_champ1 Champ_Post… { … }
nom_champ2 Champ_Post… { **source_reference** *nom_champ1* }
-*nom_champ1* désigne le nom d'un champ générique déclaré avant d'utiliser source_reference.

Enfin notons qu'un utilisateur peut nommer un champ source par l'identifiant de son choix en spécifiant le mot-clé nom_source.
nom_champ Champ_Post… { source Champ_Post… { nom_source nom} }
sinon ce champ source est nommé par défaut.

3.2.3 Nomenclature d'identification des champs génériques

Un champ générique déclaré dans le bloc « definition_champs » est nommé par l'identifiant fixé par l'utilisateur. Dans le cas où le champ est de type Champ_Post_de_Champs_post, ses sources sont nommées par défaut (appel à la méthode *nommer_source*()), de la manière suivante :

Champ_Post_refChamp : nom = « nom_champ_base »+ « _ »+ « nom_dom_natif »
-nom_champ_base : nom du champ discret du problème
-nom_dom_natif : nom du domaine de calcul

Champ_Post_Interpolation :nom=« nom_source »+ « _ »+« localisation »+ « _ »+ « nom_dom_interp »
-nom_source : nom du champ générique source
-localisation : localisation de l'interpolation (« elem » ou « som »)
-nom_dom_interp : nom du domaine d'interpolation

Champ_Post_Statistiques_Moyenne :	nom = « Moyenne_ »+ »nom_source
Champ_Post_Statistiques_Ecart_Type :	nom = « Ecart_Type_ »+ nom_source
Champ_Post_Statistiques_Correlation :	nom = « Correlation_ »+ nom_source1+nom_source2
Champ_Post_Operateur_Gradient :	nom = « Gradient_ »+ nom_source
Champ_Post_Operateur_Divergence :	nom = « Divergence_ »+ nom_source
Champ_Post_Transformation :	nom = « Combinaison_ »+ nom_source
Champ_Post_Extraction :	nom = « Extraction_ »+nom_source
Champ_Post_Reduction_0D :	nom = « Reduction_0D_ »+nom_source

Les composantes d'un champ générique identifié par « nom » sont désignées par :
nom +numero_de_composante

Rappelons que les champs de type Champ_Post_Interpolation créés par macro sont nommés de façon spécifique et portent un attribut *composantes_* rempli lui aussi spécifiquement, dans le but de pouvoir tester la non-régression du logiciel à partir des fichiers de référence.

3.3 NOUVELLES FONCTIONNALITES DE POST-TRAITEMENT

La restructuration du module de post-traitement donne la possibilité de traiter certains champs discrets sans effectuer nécessairement une procédure d'interpolation. Cet aspect concerne les champs discrétisés aux éléments (P0) ou aux sommets (P1). Toutefois le champ générique correspondant doit être déclaré avant d'être sélectionné parmi les champs à écrire. En revanche les champs ne possédant pas une discrétisation P0 ou P1 nécessitent une interpolation pour être écrits sur fichier.
L'aspect générique des « opérateurs de post-traitement » accroît les combinaisons possibles avec par exemple la possibilité de réaliser une statistique temporelle sur un champ interpolé. La généricité se trouve éventuellement limitée par les caractéristiques de l' « opérateur » lui-même. En particulier, les Champ_Post_Operateur_Gradient et Champ_Post_Operateur_Divergence ne peuvent actuellement s'appliquer qu'à des champs dotés de conditions limites. Par conséquent leur application est restreinte à certains champs inconnus du problème.
La création du Champ_Post_Operateur_Gradient combinée avec celle de l'opérateur de type Op_Grad_P1NC_to_P0 donne accès au post-traitement du gradient de champs scalaires de type P1NC possédant des conditions limites. Le gradient d'un champ de concentration devient par exemple post-traitable pour une discrétisation de type VEF. On peut aussi envisager le post-traitement d'un gradient d'une fraction massique. Les autres « opérateurs », tels que l'interpolation ou une statistique, sont bien entendu applicables ensuite à ces nouveaux champs post-traitables.
Chacun des champs statistiques possède son temps de début et de fin d'intégration. Ainsi l'influence de la période d'intégration peut être analysée.
Parmi les nouvelles fonctionnalités, on note la possibilité d'extraire les valeurs d'un champ sur une frontière d'un domaine. La réduction 0D d'un champ, dans le sens où l'on construit un champ uniforme en espace à partir d'une valeur spécifique (min, max ou moyenne) du champ source constitue aussi une nouvelle fonctionnalité de post-traitement.
Notons d'autre part qu'un champ constituant l'espace de stockage pour un champ générique est parallèle.
Remarquons enfin que tous les champs génériques sont désormais susceptibles d'être sondés.

3.4 TESTS REALISES ET CREATION DE NOUVEAUX CAS

3.4.1 Tests de non-régression

Les tests de validation sont d'une part effectués en vérifiant la non régression du logiciel. En effet, dans le cas des tests de non-régression, les champs post-traités constituent maintenant des champs génériques créés par « macro » et cette nouvelle formulation doit générer les mêmes résultats que précédemment.
Le cas Kernel_Post_MED traitant un Pb_Med présente un arrêt, car un développement réalisé au cours de l'intégration des Lot 1 et Lot 2 dans la version standard n'est pas disponible dans la présente version.
Le cas MED_docond ne peut être testé car l'interprète LireMed ne fonctionne pas.
Le cas Echangeur_VF présente un arrêt qui n'est pas analysé car il devient obsolète dans la version v1.5.3_beta.
Le cas Champs_fonc_function qui effectue une reprise et traite le post-traitement de statistiques, présente des écarts sur l'écart type de certains champs. Ces écarts ne sont pas analysés car un comportement semblable est déjà observé pour le cas Obstacle_reprise dans la version standard du logiciel.

Enfin le cas Post_Eclate est annoncé comme présentant des écarts d'une part en séquentiel et d'autre part en parallèle. Après analyse, ces écarts ne sont plus annoncés si la casse (majuscule-minuscule) du nom de certains champs post-traités est modifiée. Par conséquent les valeurs calculées ne présentent pas d'écart.

3.4.2 Nouveaux cas tests

Des tests de validation sont aussi effectués à travers l'utilisation de la nouvelle syntaxe du post-traitement de façon explicite, pour tester de nouvelles combinaisons ou fonctionnalités générées par la nouvelle conception.
Parmi les nouvelles fonctionnalités disponibles, le gradient d'un champ de concentration pour une discrétisation de type VEFPreP1b est testé. L'extraction d'un champ sur une frontière est testée pour la méthode « trace » et « champ_fontiere ». La réduction 0D de champs vectoriel et scalaire est validée pour les options « min », « max » et « moyenne ».
Les fonctionnalités de post-traitement déjà existantes tels que le gradient de temperature ou de pression ou encore la divergence de vitesse sont reproduites avec la nouvelle architecture. La reproduction de la combinaison de champs est elle aussi vérifiée.
Des combinaisons « d'operateurs » tels que l'interpolation, les statistiques, une transformation … sont aussi testées.
Enfin des sondes sont réalisées sur diverses champs génériques pour s'assurer que celles-ci ne sont pas limitées aux champs discrets du problème.
Les jeux de données de ces cas tests sont fournis au CEA avec les développements réalisés.

Annexe 1 : Codage de l'interface pour Eqn_base et Navier_Stokes
Voir remplacement par les sources
Class Eqn_base

```
        public :

        //Faire sortir si motlu pas reconnu
        //Methodes de l interface des champs postraitables
         virtual void creer_champ(const Motcle& motlu);
        virtual const Champ_base& get_champ(const Motcle& nom) const;
        virtual void get_noms_champs_postraitables(Noms& nom,Option opt=NONE) const;

        protected :
        Champ_Fonc volume_maille;
        private :
        Champs_compris champs_compris_ ;
```

//Le nom des champs compris par la classe est initialisé dans le constructeur de cette classe

```
Equation_base ::Equation_base()
{
  Noms& nom=champs_compris_.liste_noms_compris();
  nom.dimensionner(1);
  nom[0]="volume_maille";
}

void Equation_base::creer_champ(const Motcle& motlu)
{
  if (motlu == "volume_maille")
     {
       if (!volume_maille.non_nul()) {
            discretisation().volume_maille(schema_temps(),zone_dis(),volume_maille);
            champs_compris_.ajoute_champ(volume_maille);
       }
     }

  entier i;
  entier nb_op = nombre_d_operateurs();
  for (i=0;i<nb_op;i++) {
   if (operateur(i).op_non_nul())
        operateur(i).l_op_base().creer_champ(motlu);
  }


  LIST_CURSEUR(Source) curseur = les_sources;
  while(curseur) {
  if (curseur.valeur().non_nul())
      curseur.valeur()->creer_champ(motlu);
  ++curseur;
  }

}
```

//Equation base teste ses champs compris puis délègue à ses opérateurs et ses termes sources la recherche

```
const Champ_base& Equation_base::get_champ(const Motcle& nom) const
{
  REF(Champ_base) ref_champ;
  if (nom=="volume_maille")
   {
```

```
  Champ_Fonc_base& ch_vol_maille=ref_cast_non_const(Champ_Fonc_base,volume_maille.valeur());
  if (ch_vol_maille.temps()!=inconnue()->temps())
    ch_vol_maille.mettre_a_jour(inconnue()->temps());
  return champs_compris_.get_champ(nom);
 }
 try {
  return champs_compris_.get_champ(nom);
 }
 catch (Champs_compris_erreur) {

 }

 entier i;
 entier nb_op = nombre_d_operateurs();
 for (i=0;i<nb_op;i++) {
  if (operateur(i).op_non_nul())
   try {
    return operateur(i).l_op_base().get_champ(nom);
   }
   catch (Champs_compris_erreur) {
   }
 }

 LIST_CURSEUR(Source) curseur = les_sources;
 while(curseur) {
 if (curseur.valeur().non_nul())
   try {
    return curseur.valeur()->get_champ(nom);
   }
   catch (Champs_compris_erreur) {
   }
 ++curseur;
 }
 throw Champs_compris_erreur();

 return ref_champ;
}

void Equation_base::get_noms_champs_postraitables(Noms& nom,Option opt) const
{
  if (opt==DESCRIPTION)
    Cerr<<"Equation_base : "<<champs_compris_.liste_noms_compris()<<finl;
  else
    nom.add(champs_compris_.liste_noms_compris());

  entier i;
  entier nb_op = nombre_d_operateurs();
  for (i=0;i<nb_op;i++) {
   if (operateur(i).op_non_nul())
    operateur(i).l_op_base().get_noms_champs_postraitables(nom,opt);
  }

  LIST_CURSEUR(Source) curseur = les_sources;
  while(curseur) {
   if (curseur.valeur().non_nul())
    curseur.valeur()->get_noms_champs_postraitables(nom,opt);
  ++curseur;
  }
```

```
}
Class Navier_Stokes_std
public :
 //Methodes de l interface des champs postraitables
  virtual void creer_champ(const Motcle& motlu);
  virtual const Champ_base& get_champ(const Motcle& nom) const;
  virtual void get_noms_champs_postraitables(Noms& nom,Option opt=NONE) const;

  Champ_Fonc la_vorticite;
  Champ_Fonc critere_Q;
  Champ_Fonc porosite_volumique;
  Champ_Fonc combinaison_champ;
Navier_Stokes_std ::Navier_Stokes_std()
{
 Noms& nom=champs_compris_.liste_noms_compris();
  nom.dimensionner(12);
  nom[0]="vitesse";
  nom[1]="pression";
  nom[2]="gradient_pression";
  nom[3]="divergence_U";
  nom[4]="vitesseX";
  nom[5]="vitesseY";
  nom[6]="vitesseZ";
  nom[7]="pression_pa";
  nom[8]="vorticite";
  nom[9]="critere_Q";
  nom[10]="porosite_volumique";
  nom[11]="combinaison_champ";

}

void Navier_Stokes_std::creer_champ(const Motcle& motlu)
{
 Equation_base::creer_champ(motlu);

 if (motlu == "vorticite")
    {
     if (!la_vorticite.non_nul()) {
      const Discret_Thyd& dis=ref_cast(Discret_Thyd, discretisation());
      dis.creer_champ_vorticite(schema_temps(),la_vitesse,la_vorticite);
      champs_compris_.ajoute_champ(la_vorticite);
     }
    }
 else
    if (motlu == "critere_Q")
     {
      if (!critere_Q.non_nul()) {
           const Discret_Thyd& dis=ref_cast(Discret_Thyd, discretisation());
           dis.critere_Q(zone_dis(),zone_Cl_dis(),la_vitesse,critere_Q);
           champs_compris_.ajoute_champ(critere_Q);
      }
     }

  else
    if (motlu == "porosite_volumique")
    {
     if (!porosite_volumique.non_nul()) {
       const Discret_Thyd& dis=ref_cast(Discret_Thyd, discretisation());
       dis.porosite_volumique(zone_dis(),schema_temps(),porosite_volumique);
```

```
      champs_compris_.ajoute_champ(porosite_volumique);
    }
   }
  else
   if (motlu == "combinaison_champ")
    {
     if (!combinaison_champ.non_nul()) {
         combinaison_champ.typer("Champ_Fonc_Combinaison");
      combinaison_champ->nommer("combinaison_champ");
         combinaison_champ->fixer_unite("??");
         combinaison_champ->fixer_nb_valeurs_nodales(inconnue().nb_valeurs_nodales());
         combinaison_champ->valeurs().structure().identificateur() = "Champ_Fonc_Combinaison";
         Champ_Don_Fonc_Combinaison& ch_fonc =
ref_cast(Champ_Don_Fonc_Combinaison,combinaison_champ.valeur());
         Nom chaine="";
         chaine+=probleme().le_nom();
         chaine+=" ";
         chaine+=chaine_champ_combi;
         chaine+="\n";
         //Cout << "Chaine champ combi = " << chaine << finl;
         EChaine ec(chaine);
         ec >> ch_fonc;
         champs_compris_.ajoute_champ(combinaison_champ);
      }
     }

  if (le_traitement_particulier.non_nul())
    le_traitement_particulier->creer_champ(motlu);
}


const Champ_base& Navier_Stokes_std::get_champ(const Motcle& nom) const
{
 REF(Champ_base) ref_champ;

 if (nom=="vorticite")
   {
    double temps_init = schema_temps().temps_init();
    Champ_Fonc_base& ch_W=ref_cast_non_const(Champ_Fonc_base,la_vorticite.valeur());
     if (((ch_W.temps()!=la_vitesse->temps()) || (ch_W.temps()==temps_init)) && (la_vitesse-
>mon_equation_non_nul())) {
      ch_W.mettre_a_jour(la_vitesse->temps());
      }
    return champs_compris_.get_champ(nom);
   }
 if (nom=="critere_Q")
   {
    double temps_init = schema_temps().temps_init();
    Champ_Fonc_base& ch_Criter_Q=ref_cast_non_const(Champ_Fonc_base,critere_Q.valeur());
    if (((ch_Criter_Q.temps()!=la_vitesse->temps()) || (ch_Criter_Q.temps()==temps_init)) && (la_vitesse-
>mon_equation_non_nul()))
     ch_Criter_Q.mettre_a_jour(la_vitesse->temps());
    return champs_compris_.get_champ(nom);
   }
 if (nom == "porosite_volumique")
    {
     double temps_courant = schema_temps().temps_courant();
     double temps_init = schema_temps().temps_init();
```

```
    Champ_Fonc_base&
ch_porosite=ref_cast_non_const(Champ_Fonc_base,porosite_volumique.valeur());
     if ((ch_porosite.temps()!=temps_courant) || (ch_porosite.temps()==temps_init))
      ch_porosite.mettre_a_jour(temps_courant);
    return champs_compris_.get_champ(nom);
   }
 if (nom == "combinaison_champ")
   {
    double temps_courant = schema_temps().temps_courant();
    double temps_init = schema_temps().temps_init();
    //temporaire ajouter test de temps
    Champ_Fonc_base&
ch_combinaison=ref_cast_non_const(Champ_Fonc_base,combinaison_champ.valeur());
    if ((ch_combinaison.temps()!=temps_courant) || (ch_combinaison.temps()==temps_init))
    ch_combinaison.mettre_a_jour(temps_courant);
    return champs_compris_.get_champ(nom);
   }

 try {
  return Equation_base::get_champ(nom);
 }
 catch (Champs_compris_erreur) {
 }

 try {
  return champs_compris_.get_champ(nom);
 }
 catch (Champs_compris_erreur) {
 }

 if (le_traitement_particulier.non_nul())
   try {
   return le_traitement_particulier->get_champ(nom);
   }
   catch (Champs_compris_erreur) {
   }
  throw Champs_compris_erreur();

 return ref_champ;
}

void Navier_Stokes_std::get_noms_champs_postraitables(Noms& nom,Option opt) const
{
  Equation_base::get_noms_champs_postraitables(nom,opt);
  if (opt==DESCRIPTION)
    Cerr<<"Navier_Stokes_std : "<<champs_compris_.liste_noms_compris()<<finl;
  else
    nom.add(champs_compris_.liste_noms_compris());

  if (le_traitement_particulier.non_nul())
     le_traitement_particulier->get_noms_champs_postraitables(nom,opt);
}
```

Annexe 2 : Interface de la classe Champ_Generique_base

```
//////////////////////////////////////////////////////////////////
//
// File: Champ_Generique_base.h
// Directory : $TRIO_U_ROOT/Kernel/Framework
//
//////////////////////////////////////////////////////////////////

#ifndef Champ_Generique_base_inclus
#define Champ_Generique_base_inclus

#include <Objet_U.h>
#include <Motcle.h>
#include <DoubleTab.h>
#include <Domaine.h>
#include <Ch_base.h>
#include <Champ.h>

class ArrOfBit;
class Postraitement_base;
enum Entity { NODE, SEGMENT, FACE, ELEMENT };

// Description:
// Classe de base des champs generiques pour importation d un champ discret
// et actions elementaires sur ce champ
//  (postraitement, etc)
// Attention: toutes les methodes sont PARALLELES, il faut les appeler
//  simultanement sur tous les processeurs (get_domain() peut par exemple
//  construire le domaine parallele avant de le renvoyer).
class Champ_Generique_base : public Objet_U
{
  declare_base(Champ_Generique_base)
public:
  virtual entier           get_dimension() const;
  virtual double           get_time() const; //rend le temps du champ encapsule
  void nommer(const Nom& nom);
  const Nom& get_nom_post() const;
  //rend -1 si identifiant est le nom, numero de composante sinon
  static inline entier composante(const Nom& nom_test,const Nom& nom, const Noms& composantes);

  virtual void             get_property_names(Motcles & list) const;
  virtual const Noms               get_property(const Motcle & query) const; //repond aux requetes nom, nom_cible, unites et
composantes
  virtual Entity           get_localisation(const entier index = -1) const; //localisation ELEMENT ou SOM sont post-traitables
  virtual entier           get_nb_localisations() const;
  virtual const DoubleTab&  get_ref_values() const;
  virtual void             get_copy_values(DoubleTab &) const;
  virtual void             get_xyz_values(const DoubleTab & coords, DoubleTab & values, ArrOfBit & validity_flag) const;
  virtual const Domaine&    get_ref_domain() const; //rend une reference au domaine associe au champ
  virtual void             get_copy_domain(Domaine &) const;
  virtual const Zone_dis_base&  get_ref_zone_dis_base() const; //rend la zone discretisee liee au domaine
   virtual const DoubleTab&  get_ref_coordinates() const;
  virtual void             get_copy_coordinates(DoubleTab &) const;
  virtual const IntTab&     get_ref_connectivity(Entity index1, Entity index2) const;
  virtual void             get_copy_connectivity(Entity index1, Entity index2, IntTab &) const;

   // Remet l'objet dans l'etat obtenu par le constructeur par defaut
  virtual void  reset() = 0;
  virtual void completer(const Postraitement_base& post) = 0; //Complete l operateur eventuellement porte par le champ
                                                              //et nomme les sources par defaut
   virtual void  mettre_a_jour(double temps) = 0;          //Mise à jour d un operateur eventuellement porte par le champ

   //  La methode get_champ() est en particulier appelee par la classe de postraitement lorsque
   //  le champ doit etre postraite (dt_post ecoule).
   //  L'appelant doit fournir un champ "espace_stockage" non type comme parametre
   //   "espace_stockage".
   //
   //  Soit elle renvoie un champ existant (voir Champ_Post_refChamp)
   //   et elle n'utilise pas espace_stockage.
   //  Soit elle construit un nouveau champ qu'elle stocke dans espace_stockage,
   //   et la valeur de retour est espace_stockage.valeur()
```

```
  //  L'appelant recupere le resultat du calcul dans la valeur de retour,
  //   sachant qu'elle peut eventuellement referencer espace_stockage
  //   (donc, ne pas detruire espace_stockage trop tot).

  // Les etapes de creation de l espace de stockage sont :
  // espace_stockage.typer(type_champ)
  // espace_stockage.associer_zone_dis_base(une_zone_dis)
  // espace_stockage->fixer_nb_comp(nb_comp);
  // espace_stockage->fixer_nb_valeurs_nodales(nb_ddl);
  // Calcul des valeurs par instruction de la forme
  // espace_stockage.valeurs() = Operateur.calculer(source.valeurs())
  // espace_stockage.valeurs().echange_espace_virtuel()
  //return espace_stockage.valeur()
   virtual const Champ_base &  get_champ(Champ & espace_stockage) const = 0;
  //get_champ_post() renvoie le champ si l identifiant passe en parametre designe
  //le nom du champ ou l une de ses composantes
  virtual const Champ_Generique_base& get_champ_post(const Motcle& nom) const;
  //renvoie 1 si le champ est identifie, 0 sinon
  virtual entier comprend_champ_post(const Motcle& identifiant) const;
  //get_info_type_post() renvoie 0 si l on doit postraiter un tableau, 1 pour un tenseur
  virtual entier get_info_type_post() const = 0;
  //fixe l attribut identifiant_appel_ du champ pour indiquer si la requete
  //a ete lancee par le nom ou une composante du champ (ch Champ_Post_Interpolation)
  void fixer_identifiant_appel(const Nom& identifiant);

protected:
  static void               assert_parallel(entier);
  virtual Entree &  lire(const Motcle & motcle, Entree & is);
  static const char *  description();
  Nom nom_post_;
  Nom identifiant_appel_;
};
inline void Champ_Generique_base::fixer_identifiant_appel(const Nom& identifiant)
{
  identifiant_appel_ = identifiant;
}
inline entier Champ_Generique_base::composante(const Nom& nom_test,const Nom& nom,const Noms& composantes)
{
  Motcle motlu(nom_test);
  if (motlu == Motcle(nom))
    return -1;
  entier n = composantes.size();
  Motcles les_noms_comp(n);
  for (entier i=0; i<n; i++)
     les_noms_comp[i] = composantes[i];
  entier ncomp = les_noms_comp.search(motlu);
    if (ncomp == -1)
      {
      Cerr << "Erreur Trio_U, l identifiant : " << nom_test << finl
           << "ne designe ni le nom du Champ testé ni l'une de ses composantes\n";
       Cerr <<" les composantes du champ de nom "<<nom <<" sont :"<< finl;
      for (entier ii=0; ii<n; ii++)
               Cerr << composantes[ii] << " ";
      Cerr << finl;
      exit(-1);
      }
  return ncomp;
}
class Champ_Generique_erreur {
 public:
  Nom mot1;
  Champ_Generique_erreur(const Nom mot2)
  {
   mot1 = mot2;
   Cerr<<"Erreur de type : "<<mot1<<finl;
   exit(-1);
  }
};
#endif
```

FIN DE DOCUMENT