EDF R&D

FLUID DYNAMICS, POWER GENERATION AND ENVIRONMENT DEPARTMENT
SINGLE PHASE THERMAL-HYDRAULICS GROUP

6, QUAI WATIER
F-78401 CHATOU CEDEX

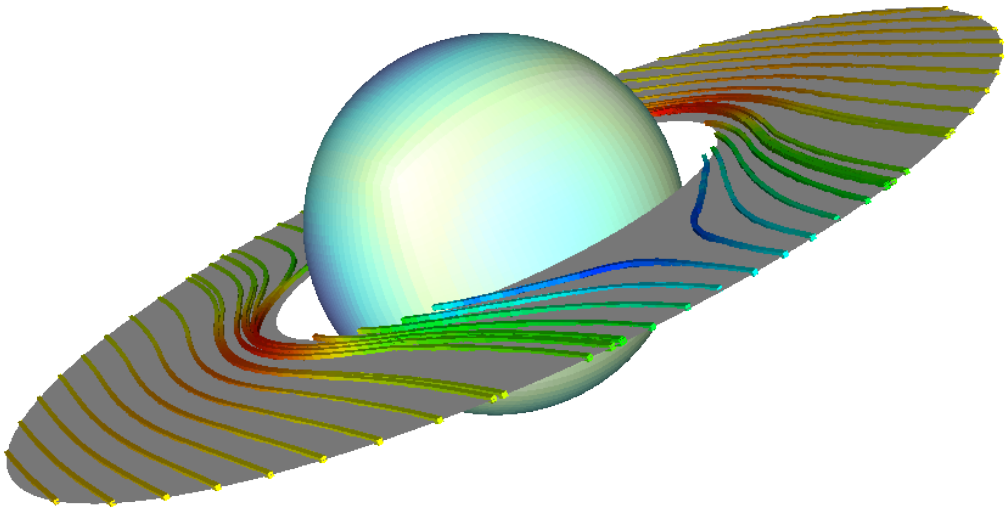
TEL: 33 1 30 87 75 40
FAX: 33 1 30 87 79 16

SEPTEMBER 2014

Code_Saturne documentation

Code_Saturne version 2.0 : manuel informatique du Préprocesseur

contact: saturne-support@edf.fr



SYNTHÈSE

Ce document a été conçu comme un manuel pour développeur : il a pour objectif de permettre à un développeur de trouver certaines informations particulières et d'aider un futur développeur à pénétrer le code.

Cependant, on ne trouvera rien dans ce document de ce qui pourrait être obtenu par un utilitaire de documentation automatique (comme une référence de fonctions par exemple).
Les essais de documentation automatique, qui ont précédé l'écriture de ce document, se sont avérés positifs : avec des utilitaires, on peut générer des documents exploitables [1] qui contiennent tout ce qui serait pénible à transcrire sous traitement de texte, plus ou moins manuellement. L'utilité de ce qui est produit est par contre assez limitée pour un développeur qui a de toutes façons accès à l'ensemble des sources, et qui utilise un environnement de développement performant permettant notamment de suivre les appels de fonctions dans les sources.

Au contraire, ce document comporte, soit des informations synthétiques sur la structure du code, soit des informations précises sur des points particuliers de programmation.

Les deux premiers chapitres donnent une description du contexte dans lequel s'inscrit le Préprocesseur ainsi qu'un aperçu global des fonctionnalités associées.
Les trois chapitres suivants exposent la manière dont le Préprocesseur a été structuré et les principes qui ont été à la base de son architecture.
Les quatre derniers chapitres intéresseront au premier chef les programmeurs, directement concernés par les outils mis en place dans le Préprocesseur, permettant d'intégrer de nouvelles fonctionnalités.

Conventions typographiques

Les conventions typographiques suivantes sont utilisées dans le Préprocesseur :

- le vocabulaire propre au Préprocesseur est écrit dans *la police de caractères penchée* (exemples : *entité de maillage*, *champ*).
  Les mots dans cette police sont automatiquement référencés dans l'index en fin de document.
- le vocabulaire relatif à l'approche « objet » ou au langage *UML* est dans *la police de caractères italique* (exemples : *classe*, *paquetage*).
  Ces mots sont automatiquement référencés dans l'index.
- les noms des catégories « objet » propres au Préprocesseur sont dans la police de caractères « sans sérif » (exemples : EntitéMaillage, Champ).
  Ces mots sont automatiquement référencés dans l'index.
- `la police de caractères « courrier »` est réservée à la transcription du code en langage C ou des instructions UNIX.
  Certains noms de types définis, de fichiers ou de fonctions sont référencés dans l'index.

1. Avec l'utilitaire Doxygen, on obtient une documentation au format HTML très complète avec la prise en compte de certains commentaires des sources. L'utilitaire cxref permet de produire une documentation plus compacte qui peut être sortie sur papier.

SOMMAIRE

Liste des figures

Liste des types définis et des structures C

Liste des exemples

Première partie

Présentation du Préprocesseur

1 Rôle du Préprocesseur dans le *Code_Saturne*

Ce chapitre replace le Préprocesseur dans le contexte du *Code_Saturne* dont elle est une des composantes.

Pour de plus amples informations sur le rôle initialement joué par le Préprocesseur au sein du *Code_Saturne*, on se reportera utilement à la référence [4].

1.1 *Code_Saturne*

Initialement, un premier module, appelé Noyau, regroupait les aspects physico-numériques du *Code_Saturne*, tandis que le second gérait les communications et les échanges de données entre le Noyau et l'extérieur du système et jouait ainsi le rôle d'Enveloppe vis-à-vis du Noyau. Ces modules étaient bien distincts, au point de constituer deux exécutables différents.

Aujourd'hui, ce rôle anciennement dévolu à l'Enveloppe est assuré principalement via la librairie FVM (qui peut être vue comme un successeur à terme de l'Enveloppe), qui est directement appellée par le Noyau. L'Enveloppe assure donc principalement le rôle de Préprocesseur, et a été renomée dans ce sens.

La description géométrique intermédiaire du *maillage*, à fournir en entrée du Noyau, fait abstraction des *types géométriques* des *éléments de maillage*. La géométrie du *maillage*, à fournir en entrée du Noyau, se limite en général à la donnée :

- de la connectivité « faces → cellules » ;
- de la connectivité « faces → sommets » ;
- des coordonnées des sommets ;
- de familles [2] des faces et des cellules.

Ce type de représentation géométrique du *maillage* n'est pas celle que fournissent les mailleurs : ils donnent en général une description en connectivité nodale du *maillage*.

Une première fonctionnalité du Préprocesseur consiste donc à transformer la description géométrique du *maillage* issue d'un mailleur dans la représentation lisible par le Noyau.

Le Préprocesseur pourra lire plusieurs *maillages* éventuellement non conformes et réaliser leur recollement conforme (par découpage de faces non conformes en sous-faces conformes).

Le Préprocesseur doit se charger aussi de la production de données associées à la vérification des maillages qui soient visualisables avec des outils externes de post-traitement.

1.2 Échanges avec le Préprocesseur

On énumère ici les différents échanges possibles avec le Préprocesseur :

- échange avec les mailleurs : le Préprocesseur lit les *maillages* dans les formats spécifiques aux mailleurs ;
- échange avec le Noyau : le Préprocesseur transmet au Noyau les données du *maillage* dans la représentation spécifique du Noyau ;
- échange avec les outils de post-traitement : le Préprocesseur peut transcrire ces mêmes données dans les formats lisibles par les outils de post-traitement (à des fins de vérification) ;

Le Préprocesseur n'est pas lié *a priori* à un ensemble figé de systèmes externes (mailleurs et outils de post-traitement). Il doit donc être en mesure d'intégrer de nouveaux formats d'échange de données avec d'autres systèmes externes.

2. Description compactée des couleurs et groupes

2 Cas d'utilisation du Préprocesseur

2.1 Acteurs

La seule entité externe qui peut interagir directement avec le Préprocesseur est l'utilisateur. On précise ci-dessous le rôle qu'il joue par rapport au Préprocesseur.

L'utilisateur doit nécessairement spécifier au Préprocesseur les fichiers de *maillage* qui devront être traités. Si plusieurs fichiers de *maillage* sont spécifiés, il doit indiquer s'il est nécessaire de les recoller.

L'utilisateur reçoit en échange les fichiers destinés au Noyau, ainsi éventuellement que des fichiers de post-traitement.

2.2 Cas d'utilisation

Il s'agit dans cette section de répertorier les services rendus par le Préprocesseur.

Cas d'utilisation 1 : Lecture des données des *maillages* Le cas d'utilisation est déclenché par l'utilisateur qui doit spécifier les noms des fichiers de maillage à lire (ainsi que leur format si l'extension n'est pas connue).

Plusieurs formats de fichier de maillage sont reconnus, comme indiqué dans le manuel d'utilisation. Chaque *maillage* lu est stocké dans une représentation interne au Préprocesseur.

La lecture du *maillage* proprement dite est dissociée de la mise en forme des données du *maillage* dans la représentation interne au Préprocesseur. La prise en compte dans le *Code_Saturne* d'un nouveau format de fichier de maillage peut dès lors facilement s'intégrer dans le Préprocesseur (cf. § 1.2, page 9).

Les données à lire peuvent provenir de plusieurs fichiers, dont les formats peuvent être différents. Les maillages sont alors concaténés.

Cas d'utilisation 2 : Concaténation de *maillages* Puisque chaque *maillage* lu est converti dans la représentation interne au Préprocesseur, des *maillages* issus de fichiers ayant des formats différents peuvent être concaténés.

La concaténation de plusieurs *maillages* ne suppose aucune opération sur le *maillage* résultant de la concaténation.

Ce cas d'utilisation est une extension du premier cas d'utilisation.

Cas d'utilisation 3 : Recollement de *maillages* ou périodicité Le recollement permet de souder des *maillages* concaténés, en découpant des faces de bord non conformes en sous-faces correspondant à leurs intersections, puis en fusionnant les sous-faces issues de ce découpage (et donc conformes) en vis-à-vis.

Si les *maillages* à recoller partagent des interfaces conformes, le recollement revient à fusionner les *sommets* ayant les mêmes coordonnées, ainsi que les *faces* s'appuyant sur ces *sommets*.

Le recollement périodique est une extension du recollement de maillages consistant à dupliquer les faces sélectionnées, de leur appliquer la transformation géométrique correspondant au pas périodique, puis d'effectuer un recollement entre les faces d'origine et les faces dupliquées, en reproduisant le découpage des faces recollées sur les faces dont elles sont issues. Les faces temporaires dupliquées puis translatées ne participant pas recollement sont supprimées.

C'est en général l'utilisateur qui déclenche ce cas d'utilisation,[3] sauf lorsque le maillage contient des informations sur le recollement (cas des maillages *Igg Hexa*).

3. En présence de plusieurs fichiers de maillage, le Préprocesseur pourrait réaliser systématiquement le recollement de *maillages* mêmes si les *maillages* sont déjà conformes. Mais le recollement de faces appartenant à des parois minces n'étant pas désiré, et les tolérances à appliquer au recollement pouvant varier selon les zones, il est préférable qu'il soit déclenché par l'utilisateur, qui sait si le recollement est nécessaire.

Cas d'utilisation 4 : Vérification de maillage Certains contrôles de cohérence peuvent être effectués par le Préprocesseur.

Cas d'utilisation 5 : Écriture du *maillage* pour le post-traitement La description géométrique du *maillage* est écrite sur fichier pour la vérification avec un outil de post-traitement. Ce cas d'utilisation est une extension du cas précédent.

Trois formats de fichier de post-traitement sont possibles : *EnSight Gold*, *MED*, ou CGNS. Il est possible d'effectuer les sorties dans plusieurs formats simultanément.

Cas d'utilisation 6 : Écriture des données spécifiques au Noyau Les données spécifiques au Noyau sont écrites par le Préprocesseur, pour le Noyau.

On rappelle que le rôle du Préprocesseur vis-à-vis du Noyau consiste justement à lui fournir les données du *maillage* dans une représentation spécifique. Ces données spécifiques sont décrites au § 1.1.

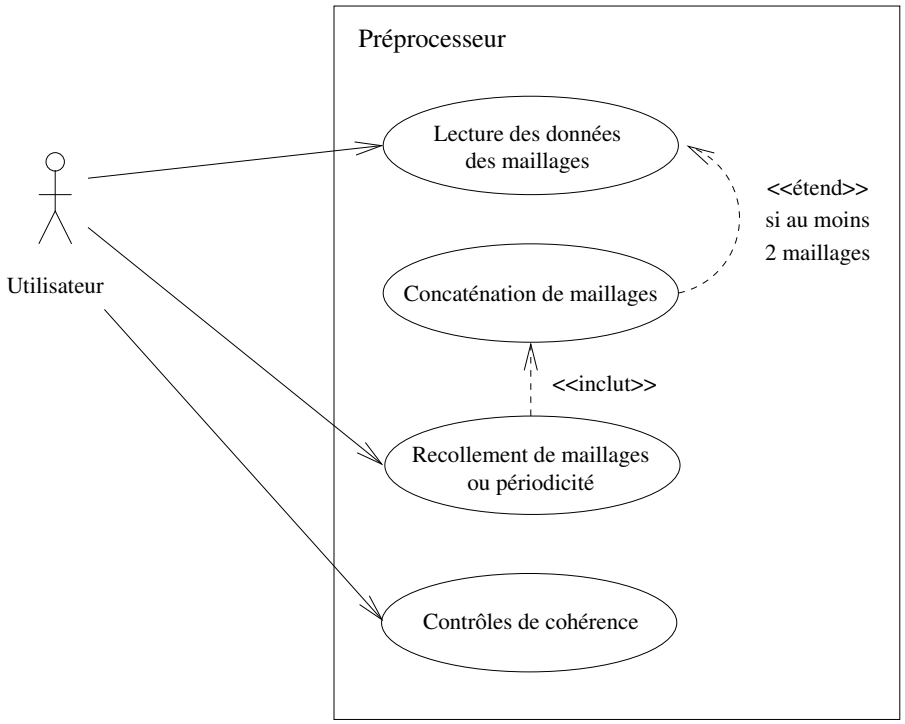


Figure 2.1: Cas d'utilisation

Deuxième partie

Modélisation objet

3 Définition des concepts clés

Ce chapitre définit les principaux concepts qui sont utilisés par le Préprocesseur et auxquels on fera constamment référence dans la suite du rapport.

Ces définitions n'ont pas une valeur générale et s'appliquent uniquement dans le cadre du module Préprocesseur.

On note que ces notions ne correspondent pas pour la plupart à une classe, mais sont représentées sous forme de *champs*. Parmi les notions décrites ci-dessous, seuls le *maillage*, l'*entité de maillage*, et le *champ* sont actuellement représentés par des classes.

Il est important de bien noter que les notions définies ci-dessous correspondent à une analyse du problème à traiter, et que le choix de représenter telle ou telle entité sous forme de classe ou non ne vient qu'après.

3.1 Maillage

On appelle *élément de maillage*, un *sommet*, une *face* ou une *cellule* :

- *élément de maillage* de dimension 0 : un *sommet* est défini par ses coordonnées ;
- *élément de maillage* de dimension 2 : une *face* est une surface fermée, pleine, définie par au moins 3 *sommets* (non alignés) ;
- *élément de maillage* de dimension 3 : une *cellule* est un volume fermé, plein, définie :
  - soit par au moins 4 *faces* (non coplanaires) ;
  - soit par au moins 4 *sommets* (non coplanaires).

Un *maillage* est un ensemble d'*éléments de maillage*, dont chaque *élément de maillage* :

- est défini en fonction d'*éléments de maillage* de dimension inférieure ;
- possède ou non des *propriétés*.

3.1.1 Connectivité

La relation permettant de définir des *éléments de maillage* en fonction d'*éléments de maillage* de dimension inférieure est appelée *connectivité*.

On distingue 2 types de *connectivité* :

- la *connectivité nodale*, dans laquelle tous les *éléments de maillage* (de dimension supérieure ou égale à 1) sont définis en fonction des *sommets* ;
- la *connectivité descendante*, dans laquelle tous les *éléments de maillage* (de dimension supérieure ou égale à 1) sont définis en fonction des *éléments de maillage* de dimension immédiatement inférieure (encore appelés *sous-éléments*).

  Le tableau ci-dessous donne pour chaque type d'*élément de maillage*, le type de *sous-élément* qui participe à sa définition dans la *connectivité descendante*.

élément de maillage	*sous-élément*
cellule	*face*
face	*sommet*

3.1.2 Type géométrique

On peut déterminer au besoin un *type géométrique* pour chaque *élément de maillage* en connectivité nodale en fonction de son type et du nombre de ses sommets.

On donne dans le tableau ci-dessous la liste des *types géométriques* définis dans le Préprocesseur.

élément de maillage	*type géométrique*
sommet	*sommet*
face	triangle quadrangle polygone
cellule	tétraèdre pyramide prisme (ou pentaèdre) hexaèdre polyèdre

3.1.3 Propriété

Un *élément de maillage* peut avoir plusieurs *propriétés*, de 2 types différents :

— il peut posséder une ou plusieurs *couleurs*[4] ;

— il peut appartenir à un ou plusieurs *groupes*, définis par un nom.

Une *couleur* est représentée par un numéro ; avec tous les mailleurs utilisés jusqu'ici (*I-deas*, *SIMAIL*, *ICEM Hexa*, et *Igg Hexa*), la couleur est un entier positif (ou construite à partir d'un entier positif indiquant un groupe ou une surface CAO pour *ICEM Hexa* et *Igg Hexa*, qui ne possèdent pas directement la notion de couleur). Par convention, une couleur de valeur zéro est considérée dans le Préprocesseur comme équivalente à aucune couleur (ce qui est proche de l'interprétation des couleurs, ou *références* sous *SIMAIL*). Un *groupe* est représenté par un nom.

3.1.4 Filiation

Une *filiation* indique une relation de type « héritage » entre l'*élément de maillage* portant un attribut *filiation* et un *élément de maillage* du même type d'*entité de maillage* (mais pas forcément du même *maillage*). On induit une relation de type *filiation* lorsqu'on construit un maillage par extraction d'un maillage parent.

3.1.5 Connectivité faces

Une *connectivité* entre faces indique une relation de type « faces en vis-à-vis » pour des faces non conformes d'un même *maillage*. La présence de ce type de relation induit un recollement conforme automatique.

3.2 Entité de maillage

On appelle *entité de maillage*, un ensemble d'*éléments de maillage* de même dimension.

Par définition, il y a 3 *entités de maillage* principales,contenant chacune tous les *éléments de maillage* de même dimension d'un maillage :

— l'*entité de maillage des sommets* ;

— l'*entité de maillage des faces* ;

— l'*entité de maillage des cellules*.

4. Les mailleurs ne donnent en général la possibilité d'attribuer qu'une seule *couleur* à un *élément*. Le Préprocesseur intègre une notion plus large de la *couleur*, afin notamment de ne pas perdre d'information sur la *couleur* des éléments en cas de recollement de maillages dont les bords en vis-à-vis sont de *couleurs* différentes.

3.3 Champ

Un *champ* associe à chaque *élément de maillage* d'une même *entité de maillage*, soit une valeur qui caractérise cet *élément* dans la définition du *maillage*, soit une valeur (de nature et utilité quelconque) attachée à cet *élément*.

Il s'agit en fait d'une des notions les plus importantes du Préprocesseur, car presque toutes les données sont représentées en fin de compte sous forme de *champs*.

Les différents types de *champ* sont les suivants :

— la définition des *éléments de maillage* en terme de *connectivité* ;

— les *couleurs* et les *groupes* ;

3.4 Famille

Une *famille* est une classe d'équivalence : deux *éléments* dont toutes les *propriétés* (*couleurs*, appartenances à un *groupe*) sont les mêmes, appartiennent à la même *famille*, alors que dès qu'une *propriété* est différente, les *éléments* appartiennent à une *famille* différente [4].

Une *famille* est identifiée par un numéro et est décrite par les *propriétés* qui la définissent (*couleurs*, appartenances à un *groupe*). Chaque *élément* a un numéro de *famille* associé.

La notion de *famille* est bien adaptée à un stockage compact des diverses propriétés des éléments, mais pas à certaines opérations. Ainsi, si l'utilisation de familles peut être pratique pour une opération de type extraction ou découpage (où il suffit que les éléments issus de l'opération héritent d'un simple numéro de famille pour connaître leurs propriétés), cette utilisation s'avère beaucoup plus problématique pour des opérations de type fusion, où des éléments héritent des propriétés de leurs deux parents, et les classes d'équivalence sont donc à reconstruire.

On notera que le modèle *MED* définit les propriétés via des *familles*, en construisant un groupe de familles sur les propriétés des nœuds (l'équivalent des *sommets* dans le Préprocesseur), et un autre groupe sur les propriétés de l'ensemble des autres *entités de maillage*. Ce découpage semble naturel pour des codes aux éléments finis, mais semblerait arbitraire si on l'appliquait au Préprocesseur, qui sépare clairement les éléments de dimension différente.

De la même façon, si l'on veut fournir des propriétés au noyau sous forme de *familles*, il semble préférable de construire les *familles* sur l'ensemble des entités, afin de réduire le risque de confusion entre les descriptions de *familles* sur les *sommets* et sur les autres entités.

On choisit donc de représenter dans le Préprocesseur les propriétés des divers éléments sous forme de *champs* de *couleurs* et de *groupes*, et de n'utiliser les *familles* que de manière transitoire, en fonction d'une logique locale.

Ainsi, les *familles* lues ou écrites dans des fichiers au format *MED* respectent la logique du modèle associé (i.e. un ensemble de familles de numéros négatifs sur les *sommets*, et un autre ensemble de numéros positifs incluant les *faces* et les *cellules*), alors que les *familles* fournies au noyau regroupent les *attributs* de toutes les *entités de maillage*, et les classes d'équivalence obtenues sont donc différentes.

4 Classes

La construction d'un logiciel, dans une conception structurée, consiste à identifier les fonctions que le programme doit réaliser, et à partitionner, de façon récursive, ces fonctions en sous-fonctions. L'architecture logicielle est alors le reflet de la hiérarchie des fonctions.

Les fonctions sont fortement liées puisqu'elles partagent des données et aucune méthode ne permet d'associer des fonctions à des données. Une modification des données peut impliquer des modifications structurelles importantes dans tout le programme.

L'approche objet consiste à regrouper les fonctions suivant les données qu'elles utilisent ou modifient. L'unité d'agrégation permettant de combiner les fonctions et les données est la *classe*.

Le Préprocesseur est essentiellement un code de structure de données, par conséquent bien adapté à une modélisation objet. Elle manipule les concepts définis dans le chapitre précédent, comme les *maillages*, *entités de maillage*, *champs*, etc., qui trouvent une modélisation évidente en termes de *classes*.

Ce chapitre répertorie les *classes* pertinentes pour appréhender le système Préprocesseur et décrit les relations entre *classes* sous forme de diagrammes de *classes*.

Le langage *UML* (Unified Modeling Language) est utilisé pour décrire les *classes* et leurs relations. Il permet de donner un contenu sémantique précis aux notations graphiques utilisées sur les schémas. Il n'est cependant pas nécessaire de connaître ce langage de modélisation pour lire ce chapitre : on pourra se contenter du texte qui accompagne les schémas.

4.1 Maillage

La classe Maillage est une *classe* principale : elle est la *classe* de plus haut niveau manipulée par le Préprocesseur.

4.1.1 Relations de la classe Maillage

Un *maillage* comporte au plus 3 *entités de maillage* :
- *entité de maillage des sommets* ;
- *entité de maillage des faces* ;
- *entité de maillage des cellules.*

Si le *maillage* est surfacique en connectivité nodale, le *maillage* pourra n'être défini qu'avec les 2 *entités de maillage* : l'*entité de maillage des sommets* et l'*entité de maillage des faces.*

Chacune de ces 3 *entités de maillage* est généralisée par une classe EntitéMaillage. La classe Maillage sera alors une *composition* d'au moins 2 classes EntitéMaillage et d'au plus 3 classes EntitéMaillage.

La définition des *propriétés* du *maillage* en termes de *familles* est facultative et transitoire : les *propriétés* sont généralement définies avec les *champs propriété*. La classe Maillage peut donc ne pas contenir de *classe*Famille.

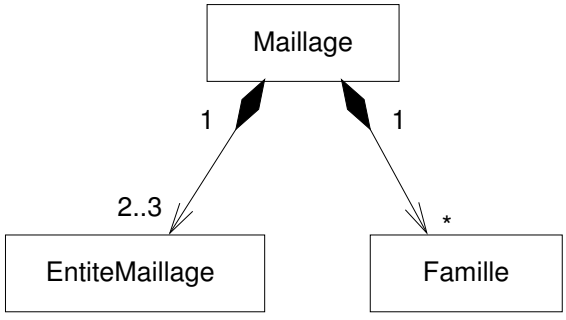


Figure 4.1: Relations de la classe

4.1.2 Attributs et méthodes de la classe Maillage

Le Préprocesseur ne manipule que les *maillages* 3D. Les *maillages* 2D sont transformés en 3D dès leur lecture.

L'outil de post-traitement *EnSight* impose de donner les *connectivités* des *éléments* par *type géométrique* : la *méthode*trieTypeGéométrique() réalise cette tâche.

La plupart des opérations sur un *maillage* sont effectuées dans la représentation en *connectivité descendante* du *maillage*. La transformation d'un *maillage* de la *connectivité nodale* à la *connectivité descendante* est réalisée par la *méthode*creeConnectivitéDescendante().

Si plusieurs *maillages* doivent être traités, ils doivent être concaténés deux à deux (i.e. les entités du second sont ajoutées au premier, ce qui implique un changement de numérotation, sans changer leurs connectivités) et doivent pouvoir être ensuite recollés : les 2 *méthodes* respectives concatène() et recolle() implémentent ces opérations.

Plusieurs *méthodes* sont dédiées aux *familles* :

— création des *familles* à partir des *champs attribut* : créeFamille() ;

— transformation des *familles* en *champs attribut* : créePropriétés() (et détruitFamille()).

La dernière méthode (extrait()) concerne la visualisation pour vérification de faces du maillage.

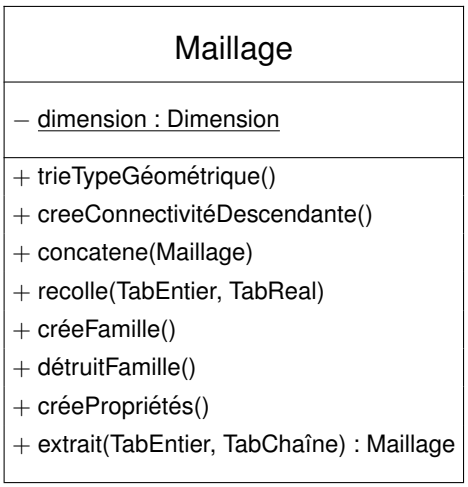


Figure 4.2: Description de la classe Maillage

4.2 EntitéMaillage

4.2.1 Généralisation des entités de maillage

On généralise les 3 *entités de maillage* par la *classe*EntitéMaillage.

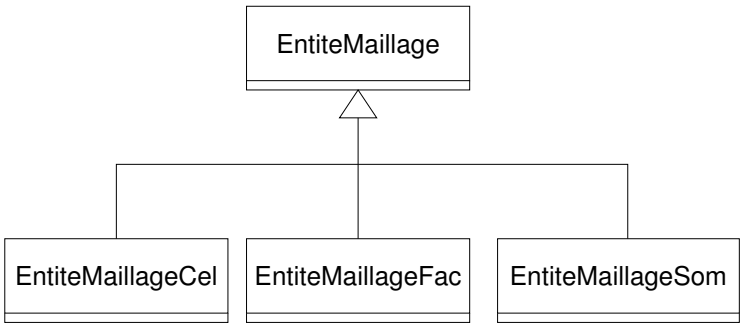


Figure 4.3: Généralisation des *entités de maillage*

4.2.2 Relations de la classe EntitéMaillage

Les *entités de maillage* sont liées entre elles par le type de *connectivité* qui sert à définir le *maillage* : par exemple, en *connectivité nodale*, chaque *face* de l'*entité de maillage des faces* est définie par des *sommets* de l'*entité de maillage des sommets*.

Une *entité de maillage* regroupe en son sein un certain nombre de *champs* :

— le *champ principal* ;

— un nombre quelconque de *champs auxiliaires*.

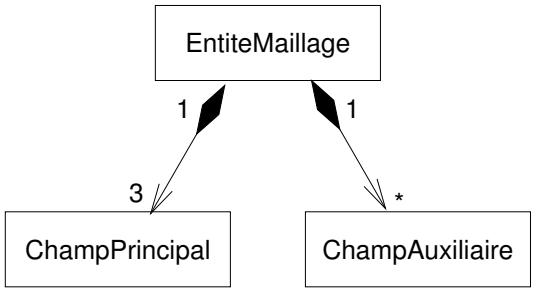


Figure 4.4: Relations de la classe EntitéMaillage

4.3 Famille

Une *famille* est une collection de *descripteurs*, chacun d'entre eux contenant une description de la *propriété* qui participe à la définition de la *famille*.

Un même *descripteur* peut appartenir à plusieurs *familles* ; une *famille* existe si elle a au moins un *descripteur*.

Si les *familles* existent, le *champ famille* contient pour chaque *élément de maillage* son numéro de *famille* (il existe au plus un *champ famille*).
Les *familles* sont donc ordonnées de façon à correspondre aux numéros de *familles* contenus dans le *champ famille*.

S'il existe une *famille*, c'est qu'il existe au moins un *élément de maillage* qui porte le numéro de cette *famille*.
Si le *champ famille* existe, c'est qu'il existe au moins une *famille*.

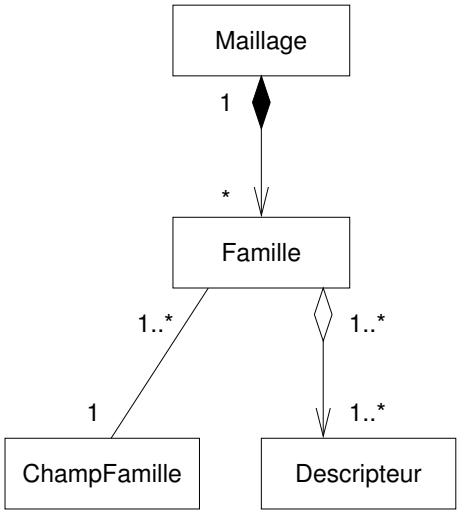


Figure 4.5: Relations de la classe *famille*

4.4 Champ

La *classe*Champ est une *classe* centrale dans la modélisation du Préprocesseur. Elle regroupe les divers *champs* participant à la définition du *maillage*.

La figure 4.6 donne une classification des *champs* et mentionne les *attributs* de la *classe*Champ.

4.4.1 Classification des *champs*

Un *champ* peut être un *champ principal* ou un *champ auxiliaire*.

Le seul *champ principal* est le *champ définition*, qui contient la définition des *éléments de maillage* en fonction d'*éléments de maillage* d'ordre inférieur ; il est aussi utilisé pour donner le nombre d'*éléments* liés à une *entité de maillage* ;

Un *champ auxiliaire* est toujours un *champ attribut*.

Parmi les *champs attribut*, on distingue :
- le *champ famille*, qui contient les numéros de *famille* des *éléments* ;
- les *champs propriété* ;
- le *champ filiation*, qui contient pour chaque *élément*, la référence à un *élément* père dans une transformation.

Les *champs propriété* sont au nombre de 2 : le *champ couleur* et le *champ groupe*, contenant pour chaque *élément*, respectivement, le numéro du *descripteur* de la *couleur* et le numéro du *descripteur* du *groupe*.

4.4.2 Attributs de la classe Champ

Un *champ* est identifié par un nom. On lui attribue un nombre d'éléments et un type.

Les *champs* se comportent différemment en cas de modification du *maillage*. Une modification du *maillage* transforme un *élément de maillage* « père » avant modification en un ou plusieurs *éléments* « fils ».
L'*attribut*statut contient le type de comportement :
- statutIndéfini ;
- statutReferenceÉlément : les valeurs du champ correspondent à des numéros d*éléments* (par exemple pour la description de la périodicité).

— statutHéritable : un *élément* ≪ fils ≫ aura la valeur du champ de l'*élément* ≪ père ≫ ;

4.4.3 Relations de la classe Champ

Un *champ* est composé de 2 tables, la *table des positions* et la *table des valeurs*, qui servent à stocker les valeurs du champ. A des fins d'économie de mémoire, la *table des positions* peut être définie par une *table réglée* dans le cas d'un *champ* entier.

Pour faciliter l'utilisation des valeurs d'un *champ* dans les algorithmes. un *vecteur indexé* est une structure réimplémentant les *tables des positions et valeurs*. Un *vecteur indexé* est donc bien associé à un *champ*, mais a une existence propre (sa *classe* n'est pas une composante de la *classe*Champ).

Les *champs propriété* contiennent les numéros des *descripteurs*, chaque *descripteur* contenant la description de la *propriété* (*couleur* ou *groupe*). Un même *descripteur* ne peut appartenir qu'à un *champ propriété* (on ne mélange pas les *couleurs* et les *groupes* dans un même champ) ; un *champ propriété* existe s'il a au moins un *descripteur*.

De même, le *champ famille* contient les numéros des *familles*, chaque *famille* étant une collection de *descripteurs*. Le *champ famille* existe s'il existe au moins une *famille* ; inversement, s'il existe au moins une *famille*, le *champ famille* existe.

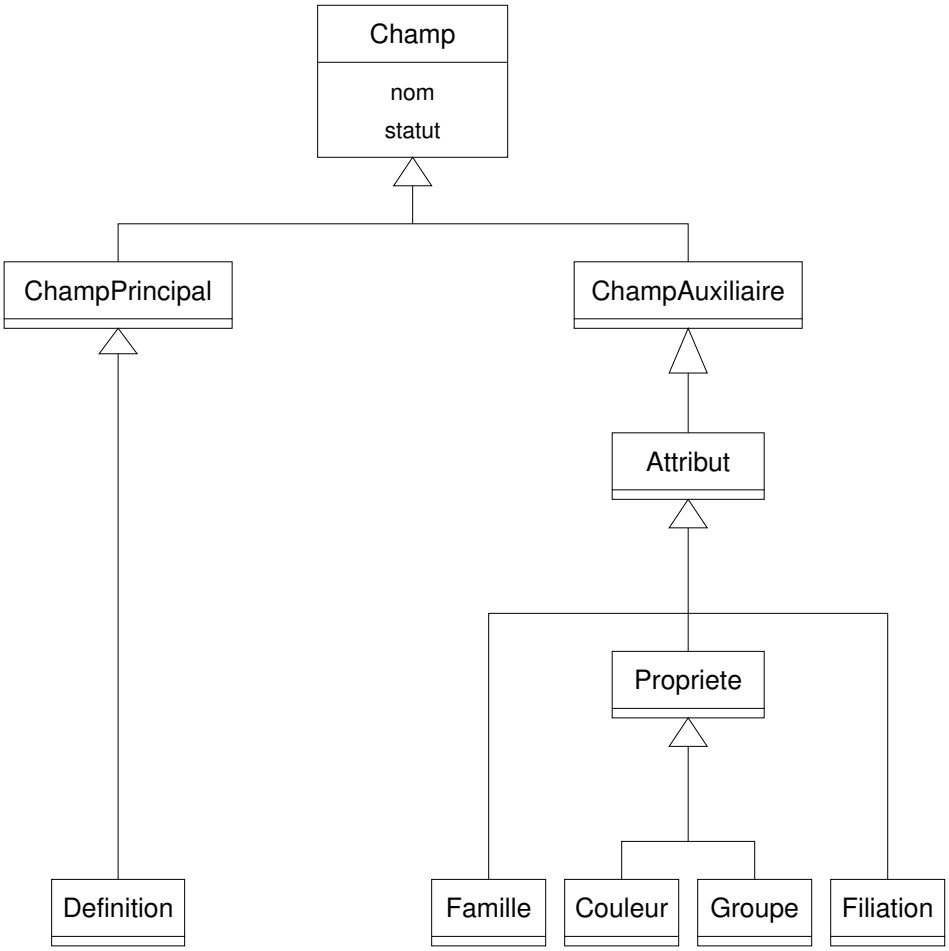


Figure 4.6: Classification des *champs*

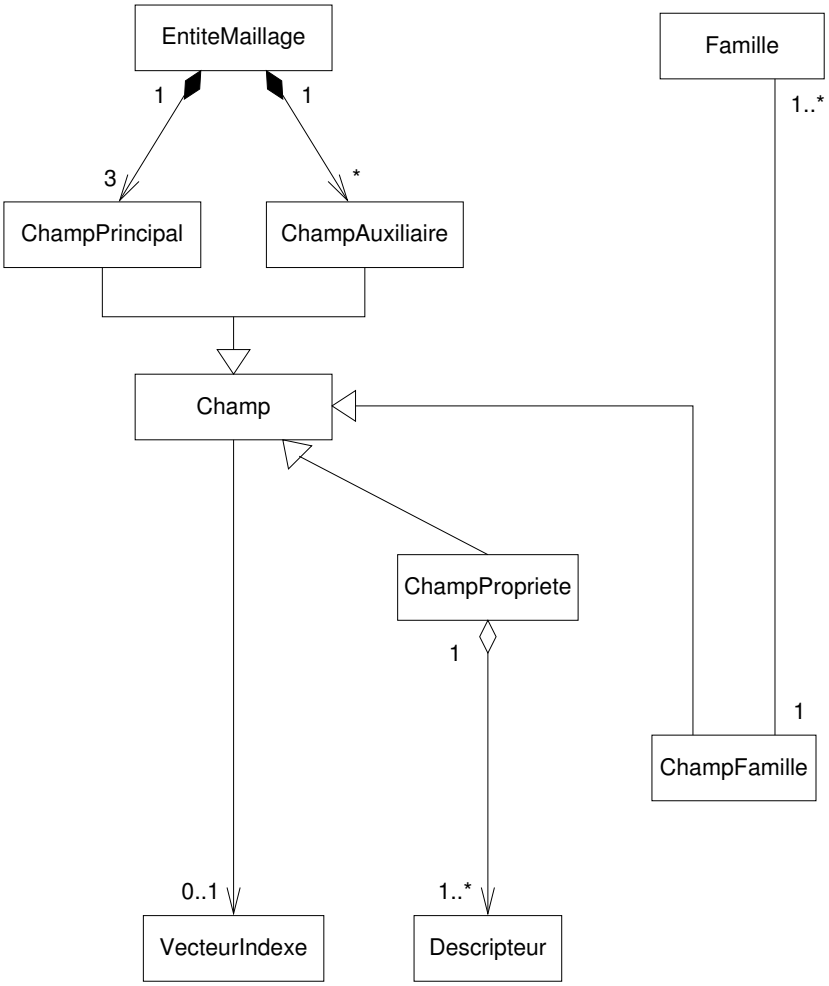


Figure 4.7: Relations de la classe Champ

4.5 Descripteur

Un *descripteur* décrit :

- soit une *propriétécouleur*, représentée par un numéro ;
- soit une *propriétégroupe*, représentée par un nom de *groupe* et éventuellement un numéro.

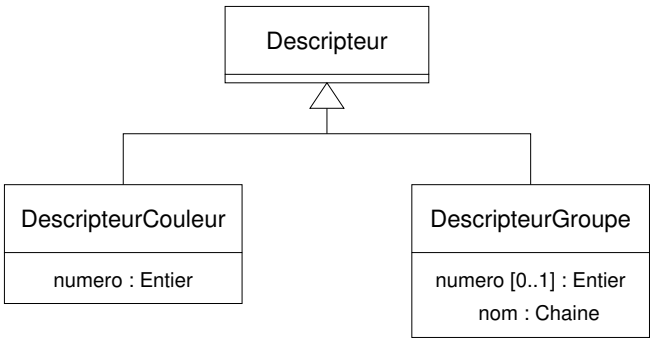


Figure 4.8: Classification des *descripteurs*

La *classe*Descripteur intervient pour décrire les *familles* et pour décrire les *champs propriété*, comme on l'a vu précédemment.

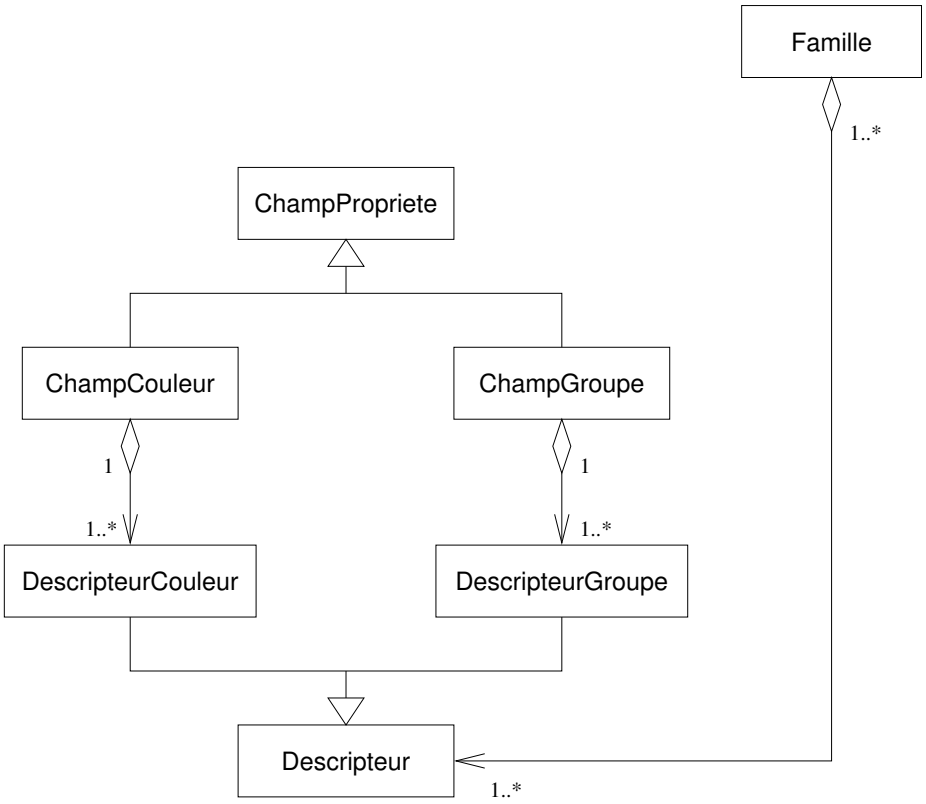


Figure 4.9: Relations de la classe Descripteur

5 Implémentation et configuration logicielle en langage C

Au premier abord, il peut sembler paradoxal d'implémenter une approche objet avec un langage « non objet », comme le C.

En fait, grâce aux macro-instructions et pointeurs de fonctions du C, il est possible d'implémenter la plupart des notions objets (y compris des notions comme l'*héritage*, la généricité, etc.).

Mais à vouloir coller de trop près à une programmation « objet », le codage devient vite lourd à mettre en œuvre. En s'écartant ainsi de la programmation en C traditionnelle, on risque de compromettre la lisibilité et la maintenabilité du programme.

Du coup, le choix du langage C par rapport au langage objet C++, peut paraître surprenant. Cependant, le C dispose encore de plusieurs avantages sur le C++ : son interfaces binaire est mieux standardisée (i.e. des fichiers objets ou librairies compilés sur une même architecture avec des compilateurs différents peuvent être liés), et surtout, il est nettement plus simple, ce qui fait à la fois que les compilateurs C sont souvent plus fiables que certains compilateurs C++. Surtout, son apprentissage est plus rapide et aisé.

Ce qui importe le plus pour nous, c'est la possibilité de regrouper des données dans une même structure. Le langage C possédant la notion de structures, l'encapsulation de données ne pose pas de problèmes. Nous ne cherchons pas à utiliser des relations d'héritage, et utilisons plutôt des attributs pour indiquer quel comportement une méthode doit adopter pour une structure donnée.

Dans le Préprocesseur, on a donc choisi, en toute connaissance de cause, de rester dans le cadre d'une programmation classique en C, et d'implémenter uniquement les notions fondamentales d'une approche objet qui sont utiles ici.

Ce chapitre expose l'architecture logicielle du Préprocesseur, construite à partir de la modélisation objet du chapitre précédent. Il insiste aussi sur la manière dont les concepts objets sont pris en compte dans le Préprocesseur, afin que tout développement à venir respecte la structure objet du Préprocesseur.

5.1 Implémentation en C

5.1.1 Approche objet

Modularité La complexité d'un logiciel peut être réduite en partitionnant le programme en parties relativement indépendantes (appelées *modules*), qui peuvent être appréhendées seules, avec quelques références aux autres parties du programme.

Un *module* exporte vis-à-vis de l'extérieur des informations qui sont contenues dans son interface.

Encapsulation et masquage de l'information À ce niveau de discussion, l'*encapsulation* et le *masquage de l'information* peuvent se définir comme un seul principe :

> un *module* doit rendre inaccessible de l'extérieur les données qu'il manipule et ne peut rendre visibles aux autres *modules* qu'un ensemble de services contenus dans son interface.

Par conséquent, une information gérée par un *module* ne sera jamais modifiable par les autres *modules*, mais pourra être consultable si l'interface du *module* déclare une fonction qui exporte cette information.

L'avantage de la mise en application de ce principe est de dissimuler aux autres *modules* la représentation des données manipulées par un *module*, et donc de forcer un *module* à utiliser une méthode (ou un sous-programme) bien identifiée pour pouvoir accéder aux données d'un autre *module*. Une modification de la représentation des données d'un *module* ne devrait avoir que de faibles conséquences sur les autres *modules*.

Classification L'approche objet est basée sur le regroupement des données et des traitements. Les données et les fonctions qui utilisent ou modifient ces données, sont regroupées dans des unités appelées *classes*.

Les données sont alors représentées par les *attributs* de la *classe*, et les fonctions sont les *méthodes* de la *classe*.

Une *classe* est associée à un *module*, et le principe de *masquage de l'information* appliqué à une *classe*, consiste à rendre les *attributs* inaccessibles à l'extérieur de la *classe* et seules les *méthodes* de la *classe* sont autorisées à manipuler explicitement les *attributs*. L'implémentation des *méthodes* reste cachée et seules les déclarations des *méthodes* sont connues à l'extérieur de la *classe*.

5.1.2 Approche objet en C

Module Le langage C (comme le C++) impose d'identifier la notion de *module* à celle de fichier, celui-ci étant l'unité de compilation du langage.

Une interface est implémentée en C sous la forme de fichier à inclure (fichier avec l'extension « .h »).
Les relations d'inclusion en C (`#include`) sont transitives : il est donc possible d'importer par transitivité une interface de *module* qui n'est pas utile.
Le corps du *module* sera un fichier source (fichier avec l'extension « .c ») n'exportant rien.

Classe Une *classe* définie lors de la conception est transformée en structure C (`struct`).

Chaque *attribut* défini dans la *classe* devient un membre de la structure.

Une référence sur un objet instance d'une *classe* peut être représentée par un pointeur sur une variable de type la structure correspondante à la *classe*.

Une *méthode* de *classe* possède un argument implicite : l'objet sur lequel s'applique la *méthode* (`this` en C++). Dans un langage non objet comme le C, cet argument doit être rendu explicite.

Les *méthodes* relatives à une *classe* peuvent être transformées en C sous la forme de pointeurs de fonction, membres de la structure correspondante à la *classe*. On retrouve de cette façon la notation pointée du C++ : `objet.méthode()`.
La création de pointeurs de fonction pour implémenter les *méthodes* de *classe* peut s'avérer lourde à mettre en place et leur emploi systématique lourd à utiliser.
Une autre solution consiste à utiliser une convention de nommage pour les fonctions : par exemple, en préfixant le nom des fonctions liées à une structure par le nom de la structure.

Encapsulation et masquage de l'information Le langage C offre la possibilité de déclarer un nouveau type structure, sans en donner sa définition.
Dans un fichier, une instruction du type

```
struct _structure;
```

est valide en C, même si le type `_structure` n'est pas défini dans ce fichier.

Dans tous les fichiers où ce type est déclaré mais dont la définition reste inconnue, il ne sera possible que d'utiliser des pointeurs sur des variables ayant ce type.
Cela n'est en rien gênant : on utilisera toujours un pointeur sur une variable de type structure, même dans les fonctions qui extraient simplement de l'information d'une structure, sans la modifier. D'ailleurs, en C, il est toujours plus efficace de passer en argument un pointeur sur une structure plutôt que la structure elle-même.

Bien entendu, dans les fichiers qui contiennent les fonctions créant ou modifiant une variable de ce type, il faudra nécessairement que ce type soit défini.

Le *masquage de l'information* peut donc être assuré en C : dans un fichier qui contient les implémentations des *méthodes* de la *classe*, on inclura la définition de la nouvelle structure implémentant la *classe* ; inversement, la définition de la nouvelle structure ne sera incluse que dans les fichiers contenant les implémentations des *méthodes*.
Dans le fichier à inclure jouant le rôle d'interface et contenant les déclarations des *méthodes*, ne figurera que la déclaration de la nouvelle structure.

5.1.3 Implémentation de l'approche objet dans le Préprocesseur

Masquage de l'information L'implémentation d'une classe et du principe de *masquage de l'information*, suit une méthode identique pour toutes les classes répertoriées au chapitre 4.

Par conséquent, on décrit ci-dessous la méthode d'implémentation pour une classe de nom générique Classe. À cette classe, correspond la structure C : `ecs_classe_t`.

1. On déclare la structure `ecs_classe_t` dans un fichier « .h » dit public (`ecs_classe_t` est en fait défini sous la forme d'un alias sur la structure proprement dite (`_ecs_classe_t`) grâce à l'instruction `typedef`, de manière à s'affranchir du mot clé `struct`). On a dans `ecs_classe.h` :

```
#ifndef _ECS_CLASSE_H_
#define _ECS_CLASSE_H_
/*
 * Declaration de la structure
 */
typedef struct _ecs_classe_t ecs_classe_t;

#endif /* _ECS_CLASSE_H_ */
```

2. On définit la structure `ecs_classe_t` dans un fichier « .h » dit privé (`ecs_classe_priv.h`) :

```
#ifndef _ECS_CLASSE_PRIV_H_
#define _ECS_CLASSE_PRIV_H_
#include "ecs_classe.h"
/*
 *  Definition de la structure
 */
struct _ecs_classe_t {
   type_membre1 membre1;
   type_membre2 membre2;
    /* etc. */
};
#endif /* _ECS_CLASSE_PRIV_H_ */
```

3. Tout fichier incluant `ecs_classe_priv.h` aura accès aux membres de la structure `ecs_classe_t` : ce fichier doit donc être forcément un fichier « .c » implémentant les *méthodes* de la *classe*Classe.
4. Tout fichier incluant `ecs_classe.h` pourra utiliser un pointeur sur une variable de type `ecs_classe_t`.
5. Une structure ou classe simple ne comportant pas d'élément privé sera définie dans un fichier « .h » global (`ecs_classe_glob.h`).

Relations entre classes On donne ici un bref aperçu sur la manière dont sont implémentées les relations entre *classes*. Dans la section 5.3 on explicitera pour chaque *classe*, les implémentations des relations de la *classe*.

Relation de composition

Exemple : relations entre la *classe*Maillage et les *classes*EntitéMaillage et Famille.
C'est le type de relation le plus fréquemment utilisé entre les *classes* du Préprocesseur.

Implémentation : la *classe* composite référence explicitement la *classe* composante (la structure composite référence explicitement le pointeur sur la structure composante).

Relation d'agrégation

Les seules relations d'*agrégation* sont les relations qui mettent en jeu la *classe*Descripteur.

Elles seront implémentées sous forme de relations de *composition* comme on l'indique dans la section 5.3.6.

Relation d'association

Deux relations d'*association* ont été mentionnées dans les diagrammes de relations de *classes* :

— la relation entre la *classe*Famille et la *classe*ChampFamille ;

— la relation entre la *classe*Champ et la *classe*VecteurIndexé.

Ces relations d'*association* lient faiblement les *classes* concernées et ne sont pas implémentées.
La contrepartie de ce choix impose de s'assurer que les *classes* associées n'existent que si la *classe* ayant la multiplicité 1 existe.

Relation de généralisation

L'*héritage* ne sera pas implémenté en tant que tel dans le Préprocesseur, pour les raisons déjà évoquées dans l'introduction du chapitre.

On précisera, pour chaque structure C comment les relations de *généralisation* entre super-*classe* et sous-*classes* ont été résolues.

La méthode consistera :

— soit à supprimer un niveau de *généralisation*,

— soit à remonter les *attributs* des sous-*classes* dans la super-*classe* ;
(Dans ce cas, si l'*attribut* ne concerne pas une *classe* concrète, il lui sera affecté une valeur par défaut. Par exemple pour les pointeurs, les *attributs* non utilisés dans une structure C se verront affecter un pointeur NULL.)

— soit à créer un nouvel *attribut* permettant de différencier les sous-*classes*.

5.2 Configuration logicielle

5.2.1 Organisation du Préprocesseur en sous-systèmes

Le système Préprocesseur est organisé en quatre sous-systèmes :

— Application : contient essentiellement le point d'entrée de l'application (programme principal `main()`);

— Base : contient l'ensemble des fichiers servant à modéliser la représentation interne des données du Préprocesseur ;

— Pré-Post : contient l'ensemble des fichiers servant, soit à convertir la représentation interne des données du Préprocesseur dans un format de sortie, soit, inversement, à convertir un format de données d'entrée dans la représentation interne des données du Préprocesseur ;

— Utilitaire : contient l'ensemble des fichiers implémentant les fonctions utilitaires de bas-niveau.

Dans le schéma ci-dessous, un sous-système est représenté sous forme de *paquetage UML*.

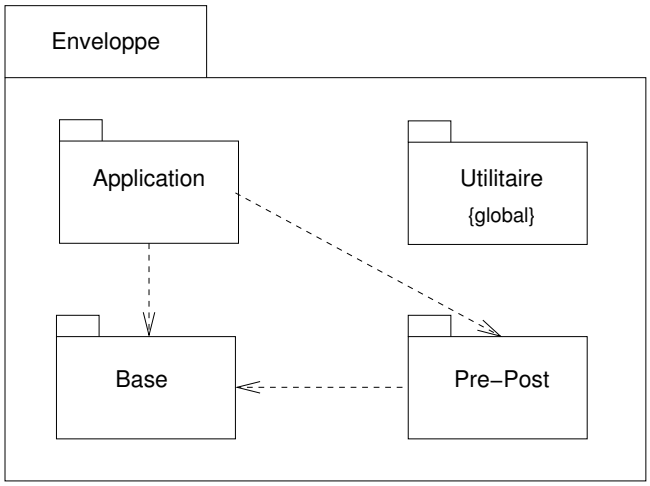


Figure 5.1: Organisation du Préprocesseur en sous-systèmes

Les relations de dépendance entre sous-systèmes sont des relations stéréotypées « importe ».

Les sous-systèmes ont des relations de visibilité, dans le sens où un fichier du sous-système qui importe un autre sous-système a la possibilité d'inclure des fichiers de déclaration du sous-système importé.

Le cas du *paquetage*Utilitaire est particulier, sa valeur marquée {global} indique qu'il est visible de tous les autres sous-systèmes.

5.2.2 Visibilité des classes

Afin d'affiner l'architecture logicielle du Préprocesseur, chacun des deux sous-systèmes Base et Pré-Post est organisé de façon à ce qu'une structure n'ait qu'une visibilité restreinte sur les autres structures.

A chaque structure on associe un *paquetage* contenant les fichiers qui déclarent et définissent les structures, ainsi que les fichiers qui implémentent les *méthodes*.

Le sous-système Base se décompose de la manière indiquée sur la Figure 5.2, page 30. Les *paquetages* sont nommés de manière à rappeler qu'ils contiennent les fichiers implémentant les *classes* définies au chapitre 4.

Les dépendances entre *paquetages* sont matérialisées par des relations de dépendance **non transitives** (stéréotypées par défaut avec le stéréotype prédéfini par *UML*, « accède »).
Certaines sont explicitement stéréotypées « association ». La signification de ce stéréotype non prédéfini est donné plus loin.

Chaque *paquetage* englobant les fichiers liés à une structure n'a la connaissance que des interfaces des *paquetages* auxquels il accède : par exemple, un fichier du *paquetage*EntitéMaillage ne pourra inclure que les fichiers déclarés visibles des *paquetages*Famille et Champ.

Le *paquetage*Maillage est quant à lui directement relié au sous-système Application, par l'intermédiaire du programme principal (cf. Figure 5.1, page 29).

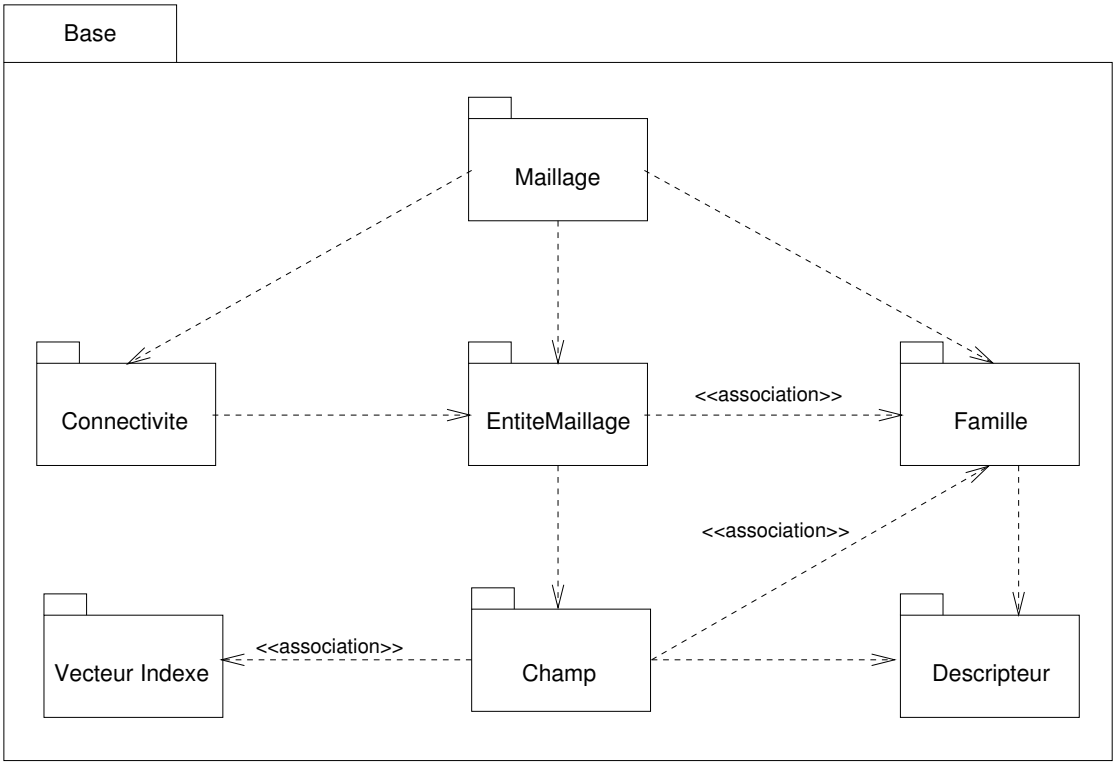


Figure 5.2: Relations de dépendances entre structures

Règles de visibilité intra-*paquetage* Un *paquetage* lié à une structure de nom générique `ecs_classe_t`, est composé des fichiers types suivants :

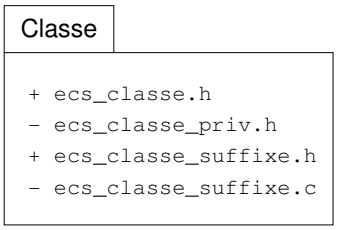


Figure 5.3: Fichiers types d'un *paquetage*

Les fichiers du *paquetage*Classe, visibles par un *paquetage* qui accède au *paquetage*Classe, sont précédés du caractère $+$. Les fichiers précédés du caractère $-$ ne sont jamais visibles à l'extérieur du *paquetage*.

Le fichier `ecs_classe_`*`suffixe`*`.c` représente de manière générique l'ensemble des fichiers « .c » implémentant les fonctions liées à la structure `ecs_classe_t` (les *méthodes* de la *classe*Classe). Son fichier `include` associé, `ecs_classe_`*`suffixe`*`.h`, déclare les prototypes des fonctions du fichier `ecs_classe_`*`suffixe`*`.c` (et implémente l'interface de la *classe*).

On donne ci-dessous les règles précises de visibilité entre fichiers du *paquetage*.

Règle 1 : le fichier `ecs_classe.h` n'inclut aucun fichier

Règle 2 : le fichier `ecs_classe_priv.h` peut inclure `ecs_classe.h` (si `ecs_classe.h` contient des énumérations utiles à `ecs_classe_priv.h` et utilisées en dehors du *paquetage*)

Règle 3 : les fichiers du type `ecs_classe_`*`suffixe`*`.h` n'incluent que `ecs_classe.h`

Règle 4 : les fichiers du type `ecs_classe_`*`suffixe`*`.c` incluent :
- `ecs_classe.h`,
- `ecs_classe_priv.h`,
- `ecs_classe_`*`suffixe`*`.h`.

Règles de visibilité inter-*paquetages* Considérons le cas suivant d'un *paquetage*Classe qui accède au *paquetage*ClasseAccede :

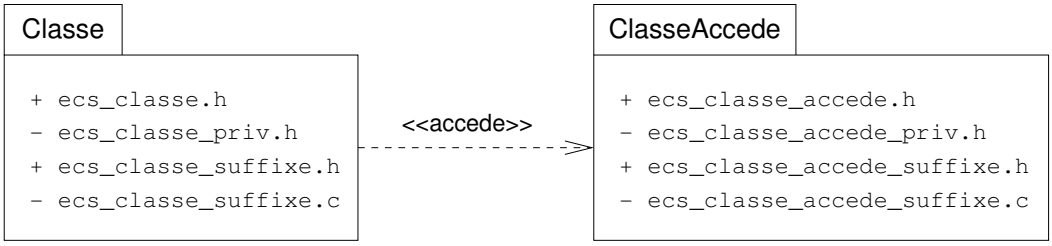


Figure 5.4: Relation type de dépendance entre *paquetages*

La figure 5.2 montre 2 types de relation entre *paquetages* :
- une relation de dépendance non stéréotypée représente une relation forte entre les 2 *paquetages* : la structure `ecs_classe_t` contient explicitement une référence à la structure `ecs_classe_accede_t` ;
- une relation de dépendance stéréotypée « association » représente une relation faible entre les 2 *paquetages* : la structure `ecs_classe_t` ne référence pas `ecs_classe_accede_t`, seules les fonctions liées à `ecs_classe_t` référencent en argument la structure `ecs_classe_accede_t`.

Les règles d'inclusion pour les fichiers de Classe par rapport aux fichiers de ClasseAccede sont plus ou moins fortes suivant l'un ou l'autre type de relation de dépendance.

Les règles d'inclusion pour les fichiers de Classe par rapport aux fichiers de ClasseAccede sont les suivantes :

Règle 1 : le fichier `ecs_classe.h` n'inclut aucun fichier de ClasseAccede

Règle 2 : le fichier `ecs_classe_priv.h` n'inclut
- pour une relation de dépendance non stéréotypée : que `ecs_classe_accede.h`
- pour une relation de dépendance stéréotypée « association » : aucun fichier de ClasseAccede

Règle 3 : les fichiers du type `ecs_classe_`*`suffixe`*`.h` n'incluent que `ecs_classe_accede.h`

Règle 4 : les fichiers du type `ecs_classe_`*`suffixe`*`.c` n'incluent que :
- `ecs_classe_accede.h`
- `ecs_classe_accede_`*`suffixe`*`.h`

Certaines relations indiquées sur la figure 5.2 représentent une dépendance très faible entre les *paquetages*. C'est le cas des 2 relations stéréotypées « association » suivantes :
- *paquetage*Champ dépendant du *paquetage*Famille ;
- *paquetage*EntitéMaillage dépendant du *paquetage*Famille.

Dans ces 2 cas, seules 2 fonctions des *paquetages* dépendants utilisent une structure `ecs_famille_t` en argument.

5.3 Correspondance entre structures C et classes

Cette section décrit les structures C qui ont été choisies pour implémenter les *classes*.

Pour établir le parallèle entre *classe* et structure C, on se référera constamment au chapitre 4.

5.3.1 Types de base

On donne ci-dessous les types de base définis dans le Préprocesseur.

Ils serviront à typer les membres des structures C.

Le premier cadre regroupe les types de base qui ne sont qu'une redéfinition des types primitifs du C, ainsi que les énumérations globales au code.

Le second cadre introduit un type *tableau* pour chacun des 4 types fondamentaux (entier, réel, chaîne de caractères et booléen) qui permet d'associer à un tableau de valeurs le nombre de valeurs du tableau.

```
typedef int         ecs_int_32_t;   /* Entier sur 4 octets */
typedef double      ecs_real_32_t;  /* Réel (virgule flottante) sur 4 octets */
typedef int         ecs_int_t;      /* Entier (taille par défaut) */
typedef unsigned    ecs_size_t;     /* Taille pour les index */
typedef double      ecs_real_t;     /* Réel (virgule flottante,
                                              double par défaut) */
typedef char        ecs_byte_t;     /* Octet (unité de mémoire non typée) */
typedef ecs_real_t ecs_point_t[3]; /* Points ou vecteurs en 3D */

typedef enum { ECS_TYPE_char,
               ECS_TYPE_bool,
               ECS_TYPE_ecs_int_t,
               ECS_TYPE_ecs_int_32_t,
               ECS_TYPE_ecs_point_t
               ECS_TYPE_ecs_real_t,
               ECS_TYPE_ecs_real_32_t,
               ECS_TYPE_void,
             } ecs_type_t;          /* Énumération des différents types */
```

Type défini 1: Types de base (1) (fichier source `ecs_def_glob.h`)

```
typedef struct { size_t        nbr;
                 ecs_int_t    *val;
} ecs_tab_int_t;

typedef struct { size_t        nbr;
                 ecs_real_t   *val;
} ecs_tab_real_t;

typedef struct { size_t        nbr;
                 char        **val;
} ecs_tab_char_t;

typedef struct { size_t         nbr;
                 ecs_bool_t   *val;
} ecs_tab_bool_t;
```

Type défini 2: Types de base (2) (fichier source `ecs_tab_glob.h`)

5.3.2 Maillage

La composition de la structure C implémentant un *maillage* est la suivante :

```
struct _ecs_maillage_t {
  ecs_maillage_connect_t   typ_connect;
  ecs_entmail_t           *entmail[ECS_ENTMAIL_FIN];
  ecs_famille_t           *famille[ECS_FAMILLE_FIN];

};
```

Type défini 3: Structure `ecs_maillage_t` (fichier source `ecs_maillage_priv.h`)

On donne ci-dessous une représentation graphique du contenu de la structure `ecs_maillage_t` en développant les tableaux statiques des membres de la structure.

`ecs_maillage_t` :
`typ_connect`
`entmail`

ECS_ENTMAIL_SOM
ECS_ENTMAIL_FAC
ECS_ENTMAIL_CEL

`famille`

ECS_FAMILLE_SOM
ECS_FAMILLE_FAC
ECS_FAMILLE_CEL

Figure 5.5: Tableaux des membres de la structure `ecs_maillage_t`

La structure contient en puissance une référence à toutes les *entités de maillage* : si une *entité de maillage* n'existe pas, la référence est le pointeur NULL.

Par exemple, pour un *maillage* surfacique, on aura :

```
maillage->entmail[ECS_ENTMAIL_CEL] = NULL
```

De la même façon, la structure `ecs_maillage_t` contient en puissance une référence à toutes les *familles*.

5.3.3 Entité de maillage

La composition de la structure C implémentant une *entité de maillage* est la suivante :

```
struct _ecs_entmail_t {
  ecs_champ_t  *champ[ECS_CHAMP_FIN];
};
```

Type défini 4: Structure `ecs_entmail_t` (fichier source `ecs_entmail_priv.h`)

On donne ci-dessous une représentation graphique du contenu du membre `champ` de la structure `ecs_entmail_t` en développant les tableaux statiques.

La structure `ecs_entmail_t` contient les têtes de listes chaînées des différents types de *champ*. Afin de donner une généralité de structure à tous les types de *champ*, même le *champ principal* sont constitués en liste chaînée : les têtes de liste sont, pour ces *champs*, les seuls maillons de la chaîne. Une fin de liste chaînée est toujours spécifiée par un pointeur NULL.

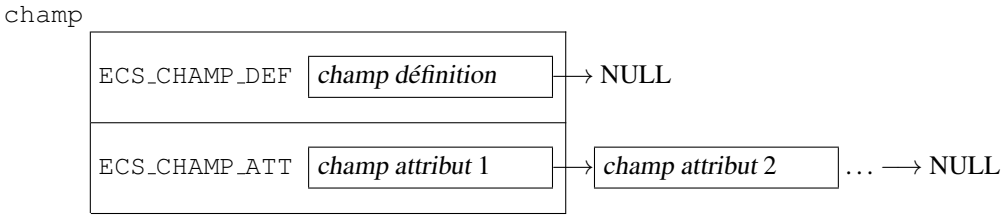


Figure 5.6: Tableau des *champs* d'une *entité de maillage*

On voit bien sur ce schéma que la hiérarchie des *champs* (cf. Figure 4.6, page 22) a été « aplatie » :
- le *champ principal* étant connu, il est directement mentionné dans le tableau ci-dessus ;
- il ne reste dans la hiérarchie des *champs auxiliaires* que catégorie des *champs attribut* ;

5.3.4 Famille

La composition de la structure C implémentant une *famille* est la suivante :

```
struct _ecs_famille_t {
  ecs_descr_t            *descr;
  struct _ecs_famille_t  *l_famille_sui;
};
```

Type défini 5: Structure `ecs_famille_t` (fichier source `ecs_famille_priv.h`)

La collection des *descripteurs* décrivant une *famille* (cf. Figure 4.5, page 20) est implémentée sous la forme d'une liste chaînée de *descripteurs* dont la tête de liste `descr` est référencée dans la structure `ecs_famille_t`.

5.3.5 Champ

Description de la structure La composition de la structure C implémentant un *champ* est la suivante :

```
struct _ecs_champ_t {
  size_t                 nbr_elt;
  ecs_type_t             typ_val;
  ecs_size_t             pos_pas;
  ecs_int_t             *pos_tab;
  void                  *val_tab;
  char                  *nom;
  ECS_CHAMP_STATUT_E     statut_e;
  ecs_descr_t           *descr;
  struct _ecs_champ_t   *l_champ_sui;
};
```

Type défini 6: Structure `ecs_champ_t` (fichier source `ecs_champ_priv.h`)

Les *attributs*nombre d'éléments et type (cf. § 4.4.2, page 20) sont directement impléméntés par les membres respectifs `nbr_elt` et `typ_val`. De même, les *attributs*nom et statut (cf. § 4.4.2, page 20) sont directement implémentés par les membres respectifs `nom` et `statut_e`. Le membre `statut_e` est de type énuméré :

```
typedef enum {
  ECS_CHAMP_STATUT_INDEFINI     = -1,
  ECS_CHAMP_STATUT_REF_ELT,
  ECS_CHAMP_STATUT_HERITABLE
} ECS_CHAMP_STATUT_E;
```

Les *tables des positions et valeurs* qui composent la *classe*Champ (cf. Figure 4.7, page 23) sont directement référencées dans la structure `ecs_champ_t` :

- `pos_tab` est un pointeur sur la *table des positions* ; si celle-ci peut être définie par une *table réglée*, ce pointeur est nul, et on utilise l'attribut `pos_pas`.
- `val_tab` est un pointeur sur la *table des valeurs*.

La collection des *descripteurs* décrivant les *propriétés* du *champ propriété* est implémentée sous la forme d'une liste chaînée de *descripteurs* dont la tête de liste `descr` est référencée dans la structure `ecs_champ_t`. Si le *champ* n'est pas un *champ propriété*, `descr` est le pointeur NULL.

Caractéristiques des champs Dans cette section, on fait pour chaque type de *champ*, un bilan des différentes valeurs que prennent les membres de la structure `ecs_champ_t`.

Caractéristiques du *champ définition*		
nom	`nom`	« definition »
statut du *champ*	`statut_e`	`ECS_CHAMP_STATUT_HERITABLE`
table des positions	`pos_pas`	*table réglée* ou non
	`pos_tab`	
table des valeurs	`val_tab`	valeurs entières ou réelles
descripteur de *champ principal*	`descr`	NULL
lien sur un *champ* suivant	`l_champ_sui`	NULL

Caractéristiques d'un *champ attribut*		
nom	`nom`	« couleur », « groupe », « famille » ou « filiation »
statut du *champ*	`statut_e`	`ECS_CHAMP_STATUT_INDEFINI` ou `ECS_CHAMP_STATUT_HERITABLE` ou `ECS_CHAMP_STATUT_REF_ELT`
table des positions	`pos_tab`	*table réglée* ou non
table des valeurs	`val_tab`	valeurs entières
descripteur de *champ attribut*	`descr`	pointeur sur *descripteur* tête
lien sur un *champ* suivant	`l_champ_sui`	pointeur sur *champ* suivant ou NULL

Répartition des fonctions dans les fichiers Les *méthodes* relatives à la super-*classe*Champ, sont implémentées sous forme de fonctions dans le fichier `ecs_champ.c` : elles s'appliquent à n'importe quel type de *champ*.

Les *méthodes* relatives aux sous-*classes* de Champ, sont implémentées sous forme de fonctions dans un fichier correspondant à la sous-*classe*. Le tableau ci-dessous donne la correspondance entre les types de *champs* et leur fichier associé :

Type de champ	Énumérateur	Fichier associé
champ définition	`ECS_CHAMP_DEF`	`ecs_champ_def.c`
champ attribut	`ECS_CHAMP_ATT`	`ecs_champ_att.c`

Les *méthodes* dépendantes de la *classe* associée VecteurIndexé, sont réparties dans les fichiers suivants :

- les fonctions référençant la structure `ecs_vec_int_t` sont contenues dans le fichier `ecs_vec_int.c` ;
- les fonctions référençant la structure `ecs_vec_real_t` sont contenues dans le fichier `ecs_vec_real.c` ;
- les fonctions référençant à la fois les structures `ecs_vec_int_t` et `ecs_vec_real_t` sont contenues dans le fichier `ecs_vec.c`.

Enfin, les fonctions gérant les listes chaînées de *champs* sont placées dans le fichier `ecs_champ_chaine.c`.

5.3.6 Descripteur

La composition de la structure C implémentant un *descripteur* est la suivante :

```
struct _ecs_descr_t {
  ecs_int_t              num;
  ECS_DESCR_TYP_E        typ_e;
  ecs_int_t              ide;
  char                  *nom;
  struct _ecs_descr_t   *l_descr_sui;
};
```

Type défini 7: Structure `ecs_descr_t` (fichier source `ecs_descr_priv.h`)

Le membre `num` de la structure représente le numéro du *descripteur* qui intervient :
- soit dans le *champ famille* si c'est un *descripteur* intervenant dans la définition d'une *famille* ;
- soit dans le *champ propriété* si c'est un *descripteur* intervenant dans la définition d'une *propriété*.

Une nouvelle fois, la *généralisation* des *descripteurs* (cf. Figure 4.8, page 23) est implémentée en aplatissant la hiérarchie des *classes* : pour différencier le type de *descripteur* (*descripteur* de *couleur* ou *descripteur* de *groupe*) le membre `typ_e` a un type énuméré :

```
typedef enum {
  ECS_DESCR_TYP_COULEUR,
  ECS_DESCR_TYP_GROUPE
} ECS_DESCR_TYP_E;
```

Un *descripteur* de *couleur* n'ayant pas de nom, il aura le membre `nom` à NULL.

Le numéro identifiant la *couleur* (ou le *groupe*) est stocké dans le membre `ide`.

La *classe*Descripteur n'intervient, dans les diagrammes de relations entre *classes*, que dans des relations d'*agrégation*, comme le montre la figure 4.9. Le choix des relations d'*agrégation* sous-entendait qu'un même *descripteur* pouvait appartenir, d'une part à l'ensemble des *descripteurs* participant à la définition d'une *famille*, d'autre part à l'ensemble des *descripteurs* participant à la définition d'un *champ propriété*.
De plus, un même *descripteur* pouvait participer à la définition de plusieurs *familles*.

5.4 Structures Vecteur Indexé

Les *vecteur indexé* n'ont pas fait l'objet d'une description en termes de *classe*, car ce sont des structures d'implémentation : Les valeurs d'un *champ* sont stockées dans la *table des valeurs*, et sont accessibles grâce au couple des *tables des positions et valeurs*. La structure *vecteur indexé* réimplémente le stockage de ces valeurs de façon à ce qu'elles soient plus facilement accessibles.

L'objectif qui a prévalu à la conception de la structure *champ*, a été de limiter au maximum l'encombrement mémoire du stockage de ses valeurs.

Une contrainte pour la construction du stockage des valeurs, était de limiter les appels à l'allocation mémoire. La solution consistant à créer deux tableaux uni-dimensionnels, l'un contenant les valeurs à stocker, l'autre les indices de début des valeurs associées à chaque élément, ne nécessite que deux appels à la fonction système d'allocation dynamique.

Si le rangement des valeurs coïncide avec un tableau rectangulaire $n \times m$, le tableau contenant les indices de début et fin pour chaque élément sera une suite arithmétique de raison m et ne consommera pas de place mémoire.

À une *table des valeurs* sur un ensemble d'éléments, il sera donc toujours associée une *table des positions*, qui contient les indices de début des valeurs associées à l'élément (l'indice de fin correspondant à l'indice de début associé à l'élément suivant -1).

De façon générale, les valeurs d'un champ sont stockées dans un tableau, ainsi que l'index (où tableau des positions) associé à ces valeurs. Cependant, si le tableau des positions constitue une suite arithmétique, plutôt que d'allouer un tableau des positions sur tous les éléments du *champ*, on ne stockera que la raison arithmétique de cette suite (son pas). En procédant de cette manière, le gain de place mémoire peut être très important.

Par exemple, pour un *champ* indiquant le numéro de famille associé à chacun de 999 éléments, on aura une famille et une seule par élément, d'où un pas de la table des positions égal à 1, comme suit :

```
pos_pas  1
pos_tab  NULL
val_tab  <...>
```

On n'utilise les *table des positions* que pour les champs d'entiers, les champs réels rencontrés (coordonnées des *sommets*) ne pouvant que très rarement s'exprimer sous cette forme. Pour un champ réel, on aura toujours une *table réglée* pour les positions.

5.4.1 Vecteur Indexé

La composition de la structure C implémentant un *vecteur indexé entier* est la suivante :

```
struct _ecs_vec_int_t {
  size_t       pos_nbr;
  ecs_size_t  *pos_tab;
  ecs_int_t   *val_tab;
};
```

Type défini 8: Structure `ecs_vec_int_t` (fichier source `ecs_vec_int_priv.h`)

La composition de la structure C implémentant un *vecteur indexé réel* est la suivante :

```
struct _ecs_vec_real_t {
  size_t       pos_nbr;
  ecs_size_t   pos_pas;
  ecs_real_t  *val_tab;
};
```

Type défini 9: Structure `ecs_vec_real_t` (fichier source `ecs_vec_real_priv.h`)

Le *vecteur indexé entier* réimplémente le couple des *tables des positions et valeurs* en une seule structure. Elle contient le nombre de positions, un tableau des positions entier et un tableau des valeurs typé. Si la *table des positions* n'est pas une *table réglée*, le tableau de la *table des positions* correspond à celui du *vecteur indexé*. si la *table des positions* est une *table réglée*, la suite arithmétique est développée en un tableau pour le *vecteur indexé*. De même, le tableau de la *table des valeurs* correspond à celui du *vecteur indexé*, mais ce dernier est typé.

Dans le cas du *vecteur indexé réel*, la *table des positions* est une *table réglée* dans tous les cas envisagés. On ne définit donc pas le tableau des positions pour un *vecteur indexé réel*, mais directement le pas, la première position étant toujours égale à 1.

La structure `ecs_champ_t` a été conçue comme une structure de stockage de données et n'est pas forcément commode à utiliser en pratique, notamment pour l'implémentation des algorithmes :

- elle nécessiterait de différencier le cas où la *table des positions* est une *table réglée* du cas où la *table des positions* n'est pas une *table réglée* ;
- la *table des valeurs* nest pas typée (plus précisément, elle a le type `void *`), il faudrait donc commencer par réaliser une conversion de type sur ce tableau pour l'utiliser.

Un *vecteur indexé* (de type `ecs_vec_int_t` ou `ecs_vec_real_t`) permet de définir un *champ* mais a une existence propre :

— un *vecteur indexé* associé à un *champ* peut être créé et détruit à tout moment ;

— si le *champ* est détruit, le *vecteur indexé* n'est pas détruit (on prendra garde justement à ce qu'il n'existe plus de *vecteur indexé* associé au *champ* à détruire) ;

— un *vecteur indexé* peut être créé sans être associé à un *champ*.

Autrement dit, la structure `ecs_vec_int_t` a été conçue pour avoir une durée de vie temporaire (le temps de l'exécution de quelques fonctions) alors que la structure `ecs_champ_t` a une durée de vie plus longue.

Les sections 6.3 et 6.4 contiennent des compléments d'information sur l'utilisation d'un *vecteur indexé*.

5.4.2 Lien entre les « positions » et les « valeurs »

Le but de cette section est de préciser le lien entre les « positions » et les « valeurs ». Ce lien existe de manière équivalente :

— soit entre la *table des positions* et la *table des valeurs* ;

— soit entre les tableaux `pos_tab` et `val_tab` des structures `ecs_vec_int_t`.

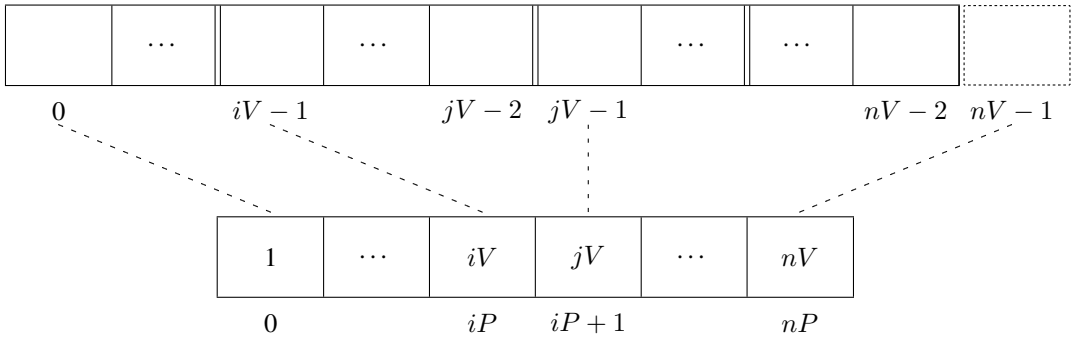
On commence par décrire ce lien dans le cas général, puis on l'illustrera sur 2 exemples représentatifs.

Pour les explications qui suivent, qu'il s'agisse du tableau de la *table des positions* (éventuellement construit à partir de la suite arithmétique) ou du tableau des positions du *vecteur indexé*, on appelle pos_tab le tableau des positions.
Pour connaître les positions des valeurs d'un élément d'indice iE, on cherche l'indice $iP = iE$ du tableau des positions pos_tab. Le nombre de valeurs associées à l'élément iE est donné par $pos\_tab[iP+1] - pos\_tab[iP]$. L'indice, dans le tableau des valeurs, de la 1ère valeur est $pos\_tab[iP] - 1$, et celui de la dernière valeur est $pos\_tab[iP+1] - 2$.

La 1ère valeur du tableau des positions est toujours égale à 1 : $pos\_tab[0] = 1$, et si on a N éléments, le tableau des positions a la dimension $pos\_nbr = N + 1 = nP + 1$.

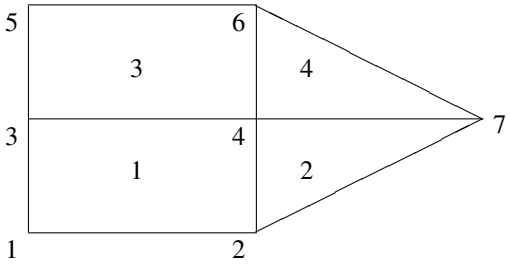
La dimension du tableau des valeurs est donnée par $pos\_tab[pos\_nbr - 1] - 1 = pos\_tab[nP] - 1$.

On représente ci-dessous les liens entre le tableau des positions et le tableau des valeurs : le tableau des positions est placé en dessous du tableau des valeurs, iP est un indice du tableau des positions, iV est un indice du tableau des valeurs.

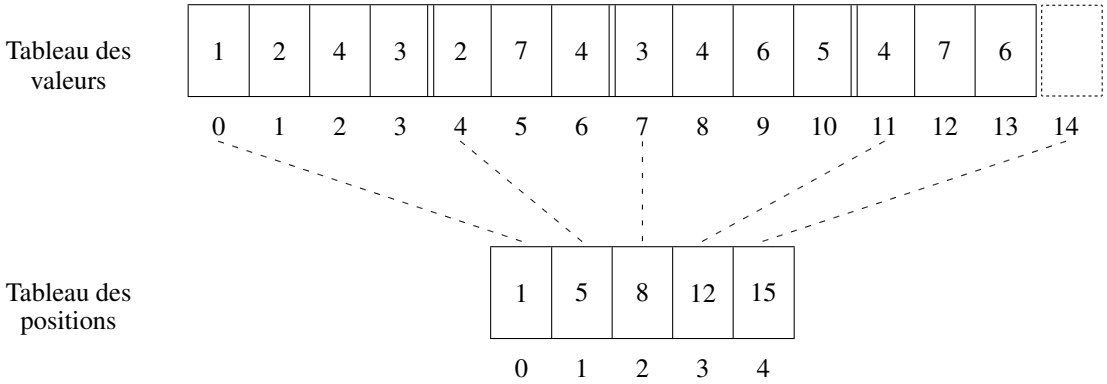


On illustre le lien entre les tableaux des positions et des valeurs dans un cas concret.

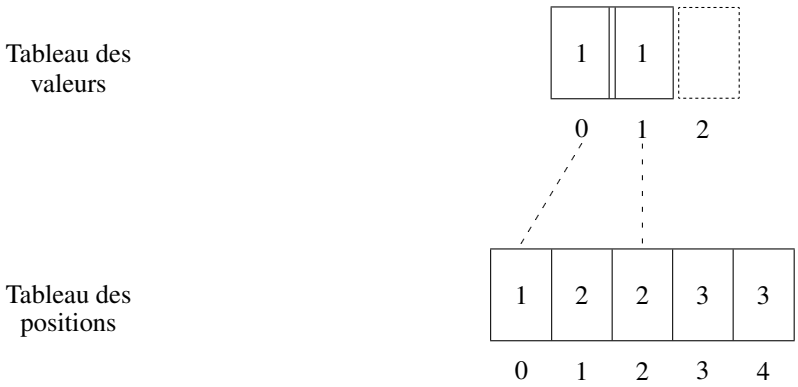
Considérons l'exemple du *maillage* surfacique plan ci-contre :



Ce *maillage* (très simple !) est composé de 2 quadrangles (numérotés 1 et 3) et de 2 triangles (numérotés 2 et 4). On construit les tableaux des positions et des valeurs correspondants à la définition des éléments en fonction des *sommets* (numérotés de 1 à 7) :



Supposons maintenant que les quadrangles appartiennent à un *groupe* nommé « QUAD » et portant le numéro 1. Les triangles sont supposés quant à eux n'appartenir à aucun *groupe*. Les tableaux des positions et des valeurs correspondants au *champgroupe* sont représentés ci-dessous :



Cet exemple montre comment est construit le tableau des positions lorsqu'un élément n'a pas de valeur : les positions de cet élément et de l'élément suivant sont égales.

Pour d'autres exemples de tableaux de position et de valeurs, et plus généralement d'autres exemples de contenus de structure `ecs_champ_t`, on se reportera à l'annexe **??** reproduisant une transcription complète sur fichier de la structure `ecs_maillage_t` et de ses sous-structures correspondant à l'exemple de *maillage* surfacique plan décrit dans la présente section (ici, les *sommets* ont une *couleur* 1 et les *faces* ont une *couleur* 7)

5.5 Autres structures

Les classes principales ne sont pas les seules utilisées. D'autres structures et classes sont définies, mais leur définition est plus liée à l'application qu'au modèle de données interne. Ainsi, on les les décrit pas en détail ici, mais on en énumère quelques unes :

- La classe `ecs_cmd_t` permet de conserver en mémoire les choix effectués au moyen des options de la ligne de commande. Un seul objet de cette classe est instancié.
- La structure `ecs_post_t` relative au choix des formats de post-traitement contient des listes chaînées de structures décrivant pour chaque format de sortie disponible les différents cas actifs. Ces structures contiennent elles-mêmes les descripteurs de fichiers utilisées, la liste des maillages associés, etc.

Troisième partie

Programmation

6 Utilisation de la structure objet

Les grandes lignes de la structure objet du Préprocesseur ont été exposées dans le chapitre précédent.

Dans le présent chapitre, on donne des informations utiles à la programmation dans le Préprocesseur, notamment :

— sur la manière d'utiliser la structure objet ;

— sur l'emploi des fonctions disponibles pour utiliser au mieux la structure objet.

Les informations contenues dans ce chapitre sont très techniques et pour mieux les illustrer, des exemples seront directement sortis des sources du Préprocesseur [5].

6.1 Mise en place des relations de visibilité

La compilation est configurée de telle manière qu'il est toujours possible, a priori, d'inclure n'importe quel fichier en-tête « .h » dans un fichier source.

Afin de respecter l'architecture du Préprocesseur, on appliquera systématiquement les règles de visibilité exposées à la section 5.2.

On reprend le cas générique décrit à la section 5.2.2, où un *paquetage*Classe accède à un *paquetage*ClasseAccede.

Supposons qu'on veuille renseigner la rubrique du fichier `ecs_classe_`*`suffixe`*`.c`

```
*  Fichiers 'include' visibles des paquetages visibles
```

Cette rubrique propose d'inclure, si nécessaire, les fichiers en-tête « .h » visibles du *paquetage*ClasseAccede :

— `ecs_classe_accede.h`

— `ecs_classe_accede_`*`suffixe`*`.h`

Si le *paquetage*Classe correspond en fait au *paquetage*EntitéMaillage (cf. Figure 5.2, page 30), les seuls *paquetages* visibles sont Famille et Champ.

6.2 Utilisation des relations entre paquetages

6.2.1 Fonctions liées à une structure C

Les noms des fonctions liées à une structure C `ecs_classe_t`, afin de rappeler qu'elles sont les implémentations des *méthodes* de la *classe*Classe, sont préfixées par

`ecs_classe__`

et leur nom complet est de la forme

`ecs_classe__`*`xyz`*`()`

Ces fonctions sont contenues dans les fichiers `ecs_classe_`*`suffixe`*`.c`.

Une sous-*classe* de la *classe*Classe, SousClasse, n'est jamais implémentée en tant que structure C : elle est directement définie par la structure `ecs_classe_t`.

Les *méthodes* de la sous-*classe* sont regroupées, quant à elles, dans un fichier dédié à la sous-*classe* et les fonctions mentionnent la sous-*classe* dans leur nom.

Par exemple, la sous-*classe*ChampAttribut de la *classe*Champ est définie directement par la structure `ecs_champ_t` ; les fonctions qui implémentent ses *méthodes* sont contenues dans le fichier `ecs_champ_att.c` et ont un nom de la forme `ecs_champ_att__`*`xyz`*`()`.

5. Les morceaux de programme qui servent d'exemple, sont retranscrits tels qu'ils existent, moyennant un reformatage pour les besoins de mise en page, consistant à la suppression de lignes blanches et d'espaces.

6.2.2 Appel d'une fonction sur une structure visible

Comme conséquence des relations de visibilité, une fonction liée à une structure ne peut appeler que deux types de fonction :

— soit une fonction du même *paquetage* ;

— soit une fonction d'un *paquetage* visible.

Considérons le cas du traitement du tri des *éléments de maillage* suivant leur *type géométrique*.
Au niveau de l'*entité de maillage*, la fonction `ecs_entmail_pcp__trie_typ_geo()` retranscrite ci-dessous, réalise tous les traitements relatifs à la structure `ecs_entmail_t` et délègue la suite des traitements aux fonctions s'appliquant sur les *champs*.

Exemple 6.1 Appel d'une fonction sur une structure visible (1)

```
void
ecs_entmail_pcp__trie_typ_geo(ecs_entmail_t  *entmail,
                              ECS_ENTMAIL_E   typ_entmail)
{
  int            dim_elt;
  ecs_tab_int_t  vect_renum;

  dim_elt = (int)typ_entmail;

  /* Tri des types géométriques des éléments (si nécessaire) */
  /*-----------------------------------------------------------*/

  vect_renum.nbr = 0;
  vect_renum.val = NULL;

  if (entmail->champ[ECS_CHAMP_DEF] != NULL)
    vect_renum = ecs_champ_def__trie_typ(entmail->champ[ECS_CHAMP_DEF],
                                         dim_elt);

  /* Application du vecteur de renumérotation sur les autres champs */
  /*-----------------------------------------------------------------*/

  if (vect_renum.val != NULL) {

    ecs_tab__int__inverse(&vect_renum);

    /* Traitement du champ représentant les définitions */

    ecs_champ_chaine__transforme(entmail->champ[ECS_CHAMP_DEF],
                                 vect_renum.nbr,
                                 vect_renum,
                                 ecs_champ__transforme_pos);

    /* Traitement des champs "attribut" */

    ecs_champ_chaine__transforme(entmail->champ[ECS_CHAMP_ATT],
                                 vect_renum.nbr,
                                 vect_renum,
                                 ecs_champ__transforme_pos);

    BFT_FREE(vect_renum.val) ;

  } /* Fin : si le vecteur de renumérotation n'est pas NULL */

}
```

On ne s'étonnera pas qu'à un niveau de structure donné, aucun traitement ne soit nécessaire : la fonction liée à la structure se contente alors de transférer le traitement à effectuer sur une structure visible de plus bas niveau.

Dans l'exemple ci-dessous, aucun traitement au niveau de la structure `ecs_maillage_t` n'est nécessaire :la fonction `ecs_maillage__calc_coo_ext()` se contente d'afficher les coordonnées minimales et maximales du domaine.

Exemple 6.2 Appel d'une fonction sur une structure visible (2)

```
void
ecs_maillage__calc_coo_ext(ecs_maillage_t  *const maillage)
{
  assert(maillage != NULL);

  if (maillage->dim_e == 3) {

    assert(maillage->entmail != NULL);
    assert(maillage->entmail[ECS_ENTMAIL_SOM] != NULL);

    ecs_entmail_pcp__calc_coo_ext(maillage->entmail[ECS_ENTMAIL_SOM]);

  }
}
```

6.3 Utilisation de la structure *champ*

La structure relative à un *champ* jouant un rôle central dans le Préprocesseur et ayant une constitution assez complexe, son utilisation est décrite en détail dans ce chapitre.

Bien que le contenu d'un champ puisse être utilisé ou rempli indifféremment soit directement soit à partir des structures *vecteur indexé*, pour l'implémentation des algorithmes on utilisera la structure *vecteur indexé* (cf. § 5.4.1, page 37).

Un ensemble de fonctions disponibles permettent de passer directement d'une structure *champ* à la structure *vecteur indexé* associée, et inversement.

Ces fonctions appellent des fonctions de plus bas niveau, qui construisent un *vecteur indexé* à partir du *champ*, en réalisant les actions suivantes :

- si le tableau des positions suit une *table réglée*, un tableau alloué est construit, et la structure *vecteur indexé* référence le pointeur sur ce tableau ;
- si le tableau des positions n'est pas une *table réglée*, la structure *vecteur indexé* référence directement le pointeur sur le tableau des positions du *champ*.
- la structure *vecteur indexé* référence directement le pointeur sur le tableau des valeurs du *champ*, moyennant la conversion de type qui convient.

Les fonctions dédiées à la transformation inverse commencent par vérifier si les valeurs du tableau des positions du *vecteur indexé* peuvent constituer une suite arithmétique (commencant à 1) : si c'est le cas, le tableau des positions du *champ* est défini par la *table réglée* correspondante, et dans le cas contraire, le tableau du *vecteur indexé* sera directement référencé dans le *champ*. Le tableau de valeurs du *vecteur indexé* sera directement référencé dans le *champ*.

On voit donc qu'un *champ* et le *vecteur indexé* associé sont intimement liés, puisqu'ils peuvent référencer le même pointeur sur un tableau. Cela évite d'avoir à allouer un nouveau tableau et d'effectuer une copie pour le remplir. La contrepartie de cette façon de procéder, c'est qu'un même pointeur sur un tableau est référencé dans deux structures différentes. Pour éviter les problèmes de pointeur « perdu », on utilisera toujours utiliser les fonctions de passage du *champ* au *vecteur indexé* (qui effectuent ces conversions et éventuelles libérations de manière fiable).

On décrit dans les sections suivantes les fonctions permettant de créer, modifier et utiliser la structure *champ*, en insistant particulièrement sur les fonctions de passage d'un *champ* à son *vecteur indexé* associé.

L'utilisation de ces fonctions est illustrée dans des cas concrets, directement tirés des sources du Préprocesseur.

Parmi les exemples suivants, ceux qui mettent en jeu un *vecteur indexé* concernent un *vecteur indexé entier*, mais toutes les fonctions mentionnées dans ce cas ont leur équivalent pour un *vecteur indexé réel*.

6.3.1 Utilisation d'un *champ* existant non modifié

Le passage d'un *champ*, dont le contenu ne sera pas modifié, au *vecteur indexé* correspondant repose sur l'utilisation de deux fonctions :

— la fonction qui retourne la structure *vecteur indexé*, remplie à partir du *champ* :

```
ecs_vec_int_t * ecs_champ__initialise_vec_int(ecs_champ_t  *this_champ)
```

— la fonction qui libère la structure *vecteur indexé* et les éventuels tableaux règlés associés au *champ* :

```
void ecs_champ__libere_vec_int(ecs_champ_t     *this_champ,
                               ecs_vec_int_t   *vec_int)
```

Dans l'exemple qui suit, le *tableau* `tab_ref` est construit à partir du *vecteur indexé* `vec_int` correspondant au *champ* `this_champ`. Le *champ* n'étant pas modifié, les 2 fonctions précitées sont utilisées pour passer du *vecteur indexé* au *champ* et inversement.

Exemple 6.3 Utilisation d'un *champ* existant non modifié

```
ecs_tab_int_t
ecs_champ__ret_reference(ecs_champ_t  *this_champ,
                         size_t        nbr_ref)
{
  ecs_tab_int_t  tab_ref;
  ecs_vec_int_t *vec_int;

  vec_int = ecs_champ__initialise_vec_int(this_champ);

  tab_ref = ecs_vec_int__ret_reference(vec_int, nbr_ref);

  ecs_champ__libere_vec_int(this_champ, vec_int);

  return tab_ref;
}
```

6.3.2 Utilisation d'un *champ* existant à modifier

Pour répercuter les modifications d'un *vecteur indexé* au *champ* qui lui est associé, on utilisera la fonction :

```
void
ecs_champ__transfere_vec_int(ecs_champ_t     *this_champ,
                             ecs_vec_int_t   *vec_int)
```

où `this_champ` est le *champ* dans lequel le *vecteur indexé* sera transféré, et `vec_int` est le *vecteur indexé* à transférer.

Dans l'exemple ci-dessous, le *vecteur indexé* initial correspondant au *champ* `this_champ` est construit de la même manière que précédemment avec la fonction `ecs_champ__initialise_vec_int`. Mais cette fois, il s'agit de répercuter les modifications du contenu du *vecteur indexé*=`vec_int` au *champ* qui lui est associé.

Exemple 6.4 Utilisation d'un *champ* existant à modifier

```
void
ecs_champ__incremente_val(ecs_champ_t  *this_champ,
                          ecs_int_t     increment)
{
  ecs_vec_int_t *vec_int;

  vec_int = ecs_champ__initialise_vec_int(this_champ);

  ecs_vec_int__incremente_val_sgn(vec_int, increment);

  ecs_champ__transfere_vec_int(this_champ, vec_int);
}
```

6.3.3 Création d'un champ

Création d'un *champ* à partir d'un *vecteur indexé* On se place dans la situation où un nouveau *champ* doit être créé à partir d'un *vecteur indexé*.

Construire un *champ* à partir d'un *vecteur indexé* nécessite d'utiliser la fonction :

```
ecs_champ_t * ecs_champ__init_avec_vec_int(ecs_vec_int_t   *vec_int,
                                           const char      *nom)
```

qui retourne un *champ* et prend pour arguments son nom et le *vecteur indexé* à partir duquel il sera construit.

Une structure *champ* contient plus d'informations qu'une structure *vecteur indexé*, et cette dernière ne suffit donc pas à en renseigner tous les membres. La fonction `ecs_champ__init_avec_vec_int` impose déjà de fournir en supplément du *vecteur indexé*, le nom du champ. Les autres membres du *champ* sont initialisés à des valeurs indéfinies (les pointeurs sont à NULL). Ces autres membres pourront être renseignés ultérieurement.

Dans l'exemple ci-dessous, le *statut* du *champ* est renseigné explicitement après la création du *champ*. Le *champ* `champ_connect_fac_som` ne sera pas rattaché à la structure *maillage* (ce qui nécessiterait un appel de fonction approprié), les autres membres de la structure gardent leur valeur d'initialisation donnée par la fonction `ecs_champ__init_avec_vec_int`.

Exemple 6.5 Création d'un *champ* à partir d'un *vecteur indexé*

```
void
ecs_champ_def__decompose_cel(ecs_champ_t  *vect_champ_fac[],
                             ecs_champ_t  *champ_def_cel)
{
  ecs_vec_int_t *vec_def_cel;
  ecs_vec_int_t *vec_def_fac;
  ecs_vec_int_t *vec_cel_def_fac;

  size_t nbr_fac;

  /*xxxxxxxxxxxxxxxxxxxxxxxxxx Instructions xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx*/

  /* Construction, pour les cellules,                                  */
  /*  des vecteurs 'ecs_vec_int_t' associés aux champs principaux */
  /*-------------------------------------------------------------------*/

  vec_def_cel = ecs_champ__initialise_vec_int(champ_def_cel);

  ecs_vec_def__decompose_cel(&vec_def_fac,
                             &vec_cel_def_fac,
                             vec_def_cel);

  ecs_vec_int__detruit(vec_def_cel);

  ecs_champ__transfere_vec_int(champ_def_cel, vec_cel_def_fac);

  nbr_fac = ecs_vec_int__ret_pos_nbr(vec_def_fac) - 1;

  vect_champ_fac[ECS_CHAMP_DEF]
    = ecs_champ__init_avec_vec_int(vec_def_fac, ECS_CHAMP_NOM_DEFINIT);

  vect_champ_fac[ECS_CHAMP_DEF]->statut_e = ECS_CHAMP_STATUT_INDEFINI;
}
```

Création complète d'un *champ* Un *champ* peut être aussi construit directement en renseignant en argument les valeurs qui lui seront attribuées (la phase de lecture d'un *maillage* procède de cette façon) :

```
ecs_champ_t *
ecs_champ__cree(size_t               nbr_elt,
                const ecs_size_t     pas_pos,
```

```
                const ecs_size_t    *tab_pos,
                const void          *tab_val,
                ecs_type_t           typ_val_e,
                const char          *nom,
                ecs_descr_t         *descr,
                ECS_CHAMP_STATUT_E   statut_e)
```

Dans l'exemple suivant, cette fonction est appelée pour construire le *champ définition* des *sommets* après lecture d'un fichier de maillage. les *tables des positions et valeurs* sont des suites arithmétiques et il est possible de renseigner directement les valeurs qui seront attribuées au *champ* :

Exemple 6.6 Création complète d'un *champ*

```
ecs_entmail_t *
ecs_entmail_pre__cree__som(size_t       nbr_som,
                           ecs_real_t  *som_val_coord)
{
  /* Debut du corps de la fonction */

  champ_definit = ecs_champ__cree(nbr_som,
                                  3,
                                  NULL,
                                  som_val_coord,
                                  ECS_TYPE_ecs_real_t,
                                  ECS_CHAMP_NOM_DEFINIT,
                                  NULL,
                                  ECS_CHAMP_STATUT_HERITABLE);

  /* Suite du corps de la fonction */
}
```

6.3.4 Rattachement d'un champ à une *entité de maillage*

Une fois le *champ* créé, si c'est un champ qui doit appartenir à la structure *maillage* de base, il doit être rattaché au pointeur sur le type de *champ* correspondant (*champ définition* ou listes chaînées *champ attribut*).

On utilisera pour cela la fonction :

```
void
ecs_champ_chaine__ajoute(ecs_champ_t  **this_champ_tete,
                         ecs_champ_t   *champ_concat_tete)
```

qui rajoute la liste chaînée à concaténer (dont la tête de liste est `champ_concat_tete`) à la fin de la liste chaînée réceptrice (dont la tête de liste est `this_champ_tete`).

Dans l'exemple qui suit, les *champscouleur* et *groupe* sont construits à partir du *champ famille*. Une fois construits, ils sont rattachés à la liste chaînée des *champs attribut* de l'*entité de maillage* `this_entmail`.

Cet exemple montre aussi comment rechercher un *champ* particulier dans une liste chaînée de *champs*, à partir de son nom : le *champ famille* (de nom `ECS_CHAMP_NOM_FAMILLE`) est recherché parmi les *champs attribut* grâce à la fonction :

```
ecs_champ_t *
ecs_champ_chaine__trouve_nom(ecs_champ_t  *this_champ_tete,
                             const char   *nom_champ)
```

qui prend comme 1er argument la tête de la liste chaînée des *champs* dans laquelle est faite la recherche, et comme second argument, le nom du champ à rechercher. La fonction retourne le pointeur sur le *champ* trouvé s'il existe sinon elle retourne le pointeur NULL.

La fonction de suppression d'un champ d'une liste chaînée de *champs* est utilisée dans cet exemple, afin de supprimer parmi les *champs attribut*, le *champ famille*, une fois les *champscouleur* et *groupe* construits. Cette fonction a pour 1er argument le *champ* à supprimer, et pour second argument, la tête de la liste chaînée dans laquelle le champ doit être supprimé :

```
void
ecs_champ_chaine__supprime(ecs_champ_t  **this_champ_noeud,
                           ecs_champ_t  **champ_tete)
```

Le *champ* ainsi supprimé perd bien entendu son lien sur le *champ* suivant qu'il avait lorsqu'il appartenait à la liste chaînée (le lien sur le *champ* suivant est à mis à NULL, et l'ancienne valeur est affectée au *champ* suivant du *champ* précédent celui qui est supprimé, afin de recomposer la liste).

Cette fonction se contente de retirer le *champ* de la liste chaînée, mais ne le détruit pas. Dans cet exemple, le *champ famille* est détruit explicitement après avoir été supprimé de la liste chaînée.

Exemple 6.7 Rattachement d'un *champ* à une *entité de maillage*

```
void
ecs_entmail_pcp__cree_attribut(ecs_entmail_t  *this_entmail,
                               ecs_famille_t  *famille)
{
  ecs_champ_t *champ_couleur;
  ecs_champ_t *champ_famille;
  ecs_champ_t *champ_groupe;

  if (this_entmail->champ[ECS_CHAMP_ATT] != NULL)
    champ_famille = ecs_champ_chaine__trouve_nom
                      (this_entmail->champ[ECS_CHAMP_ATT],
                       ECS_CHAMP_NOM_FAMILLE);
  else
    champ_famille = NULL;

  if (champ_famille != NULL) {

    /* Création des champs "couleur" et "groupe" */
    ecs_champ_att__cree_att_fam(champ_famille,
                                famille,
                                &champ_couleur,
                                &champ_groupe);

    /* Suppression de "famille" de la liste des champs "attribut" */
    ecs_champ_chaine__supprime(&champ_famille,
                               &this_entmail->champ[ECS_CHAMP_ATT]);

    /* Libération du champ "famille" */
    ecs_champ__detruit(champ_famille);

    /* Ajout de "couleur" et "groupe" à la liste des champs "attribut" */
    ecs_champ_chaine__ajoute(&this_entmail->champ[ECS_CHAMP_ATT],
                             champ_couleur) ;
    ecs_champ_chaine__ajoute(&this_entmail->champ[ECS_CHAMP_ATT],
                             champ_groupe) ;
  }
}
```

6.4 Utilisation de la structure *vecteur indexé*

Dans la section précédente on a vu comment créer ou utiliser un *vecteur indexé* à partir d'un *champ*. Un cas d'utilisation de la structure *vecteur indexé* n'a pas encore été décrit, qui ne met pas en jeu un *champ*, lorsqu'un *vecteur indexé* est utilisé simplement comme structure de travail. Deux fonctions seront utiles dans ce cas :

— la fonction qui retourne un *vecteur indexé* en ayant alloué ses tableaux à partir des dimensions des tableaux des positions et des valeurs, données en argument :

```
ecs_vec_int_t *
ecs_vec_int__alloue(size_t  pos_nbr,
                    size_t  val_nbr)
```

— la fonction qui détruit un *vecteur indexé* (en ayant préalablement dés-alloué les tableaux des positions et des valeurs) et qui retourne un pointeur NULL :

```
ecs_vec_int_t *
ecs_vec_int__detruit(ecs_vec_int_t  *this_vec_int)
```

Exemple 6.8 Utilisation d'un *vecteur indexé* de travail

```
void
ecs_vec_int__transforme_pos(ecs_vec_int_t        *this_vec_int,
                            size_t                nbr_elt_ref,
                            const ecs_tab_int_t   vect_transf)
{
  ecs_vec_int_t *vec_int_ref;
  size_t         nbr_val_ref;

  nbr_val_ref = ecs_vec_int__ret_val_nbr(this_vec_int);

  vec_int_ref = ecs_vec_int__alloue(nbr_elt_ref + 1,
                                    nbr_val_ref);

   /* Suite du corps de la fonction */

  ecs_vec_int__detruit(vec_int_ref);
}
```

De nombreuses fonctions de bas niveau permettent d'effectuer des transformations sur une structure *vecteur indexé* (tri, renumérotation, etc.). Les déclarations de ces fonctions sont rassemblées dans le fichier `ecs_vec_int.h` pour un *vecteur indexé entier* et dans le fichier `ecs_vec_real.h` pour un *vecteur indexé réel*.

7 Conventions et règles

7.1 Nommage

7.1.1 Généralités

Les règles suivantes sont appliquées dans le Préprocesseur :

- le nom d'un identificateur comporte au plus 31 caractères (ANSI C) ;
- le nom d'un identificateur est en minuscules sauf le nom d'une macro ou d'une énumération qui est entièrement en majuscules ;
- un identificateur global a un nom préfixé par `ecs_` ou `ECS_` ;
- un identificateur local à un fichier a un nom préfixé par `ecs_loc_` ou `ECS_LOC_` :
  - le nom d'une fonction locale à un fichier (`static`) sera préfixé par `ecs_loc_` ;
  - le nom d'une énumération locale à un fichier sera préfixé par `ECS_LOC_` ;
  - le nom d'une macro locale à un fichier sera préfixé par `ECS_LOC_` ;
  - le nom d'une macro locale à une fonction sera préfixé par `ECS_FCT_` ;

7.1.2 Nommage des énumérations

Dans la version 1.0, les énumérations étaient écrites sous la forme :

```
typedef enum { ECS_ENUMERATION_ENUMERATEUR1,
               ECS_ENUMERATION_ENUMERATEUR2,
               /* etc. */
             } ECS_ENUMERATION_E;
```

On préfère maintenant utiliser la forme :

```
typedef enum { ECS_ENUMERATION_ENUMERATEUR1,
               ECS_ENUMERATION_ENUMERATEUR2,
               /* etc. */
             } ecs_enumeration_t;
```

7.1.3 Nommage des *classes*

La structure objet du Préprocesseur reposant aussi sur des conventions, on a donc déjà eu l'occasion d'exposer les conventions de nommage sur les *classes* : on se reportera aux sections 5.2, 5.3 et 6.2.

On reprend ici le cas d'une *classe* générique dont la structure C a pour nom `ecs_classe_t` (définie comme alias par `typedef` de `struct _ecs_classe_t`).

Les macros et les énumérations liées à la *classe* sont préfixées par `ECS_CLASSE_`.

Les fonctions implémentant les *méthodes* ont un nom de la forme :

`ecs_`[`loc_`]*classe* [_mot-clé]`__xyz()`

où :

- `loc_` est présent s'il s'agit d'une fonction locale à un fichier ;
- « mot-clé » désigne, pour le sous-système Base :
  - soit une sous-*classe* ;
  - soit une classe associée ;

— soit un type de collection de la classe (`chaine` ou `arbre`).

Les fichiers contenant ces fonctions sont de la forme :

`ecs_`*classe* [_mot-clé].c

Dans la section 5.3.5 on a donné les fichiers qui contiennent les fonctions de la structure `ecs_champ_t`.

Ci-dessous, on trouve deux exemples concernant la structure `ecs_descr_t` :
l'exemple 7.1 retranscrit le fichier `ecs_descr.h` tandis que 7.2 donne l'exemple d'une fonction déclarée dans `ecs_descr_chaine.h`.

Exemple 7.1 Nommage pour un *descripteur* (1)

```
/*-----------------------------------------------------------------------------
 *  Déclaration de la structure
 *----------------------------------------------------------------------------*/

typedef struct _ecs_descr_t ecs_descr_t;


/*-----------------------------------------------------------------------------
 *  Définition d'énumération
 *----------------------------------------------------------------------------*/

typedef enum { ECS_DESCR_COULEUR,
               ECS_DESCR_GROUPE
             } ecs_descr_typ_t;


/*-----------------------------------------------------------------------------
 *  Définition de macros
 *----------------------------------------------------------------------------*/

#define ECS_DESCR_NUM_NUL -1
#define ECS_DESCR_IDE_NUL -1
#define ECS_DESCR_ENT_NUL -1
```

Exemple 7.2 Nommage pour un *descripteur* (2)

```
/*-----------------------------------------------------------------------------
 *  Fonction qui recherche dans une liste chaînée de descripteurs
 *   dont la tête est donnée,
 *   un numéro de descripteur donné
 *
 *  La fonction renvoie :
 *  -    le pointeur du descripteur si le numero de descripteur a ete trouve
 *  - ou NULL                        sinon
 *----------------------------------------------------------------------------*/

ecs_descr_t *
ecs_descr_chaine__cherche_num(ecs_descr_t  *descr_tete,
                              ecs_int_t     num);
```

7.2 Présentation des sources

7.2.1 Généralités

Le codage des sources du Préprocesseur respecte les conventions de présentation suivantes :

— pas d'indentation : 2 caractères ;

— longueur maximale des lignes : 80 caractères ;

— emploi systématique des minuscules pour les instructions et les identificateurs, sauf pour les identificateurs des macros et des énumérations qui sont entièrement en majuscules.

7.2.2 Présentation des fonctions

L'exemple 7.2 donne un exemple de présentation des fonctions dans le Préprocesseur.

7.3 Programmation

7.3.1 Langage

Le langage C est utilisé dans sa version ANSI 1989.

Afin de donner un certain confort à l'utilisateur du Préprocesseur, quelques extensions à la norme ANSI ont été introduites dans le code. Ces extensions ne sont intégrées que si des variables d'environnement ont été effectivement positionnées au moment de la compilation (cf. § 8.1.3, page 57). Autrement dit, il est toujours possible d'exclure ces extensions sans que le fonctionnement du Préprocesseur en soit perturbé.

Certains aspects douteux du langage C ont été conservés par la norme ANSI pour des raisons historiques de compatibilité avec les versions diffusées du langage avant normalisation. On pense surtout à la possibilité de définir une fonction sans en avoir préalablement défini un prototype (i.e. déclaré son interface). Ces aspects du langage (qui n'ont d'ailleurs pas été retenus en C++) ne seront pas utilisés dans le cadre du module Préprocesseur. Les versions récentes des compilateurs C préviennent généralement par un « warning » en cas d'utilisation de ces aspects du langage. La compilation en C++ du Préprocesseur permet de les repérer à coup sûr.

La compilation systématique sur différentes plates-formes (avec les options de compilation appropriées) et la compilation en C++, devrait garantir le respect de la norme ANSI, ainsi que l'absence d'utilisation d'aspects douteux du langage C.

7.3.2 Règles de programmation

Les règles de programmation utilisées dans le Préprocesseur sont classiques.

Citons quelques règles en vrac :

- chaque variable doit être initialisée avant d'être utilisée
  (les pointeurs seront systématiquement initialisés à NULL)
- l'usage des variables globales doit être évité en dehors des fonctions utilitaires de base de la librairie ou des chaînes de caractère statiques représentant les divers messages affichés.
  Si la création d'une variable globale à un fichier est justifiée, elle doit être déclarée `static`.
  De même, une variable globale à une fonction doit être déclarée `static`.
- un type `const` ne doit pas être converti en un type non-`const`
- chaque fois que la valeur d'une expression doit être testée parmi un ensemble de constantes, une construction `switch` doit être préférée à une construction
  `if ... else if ... else ...`
- les clauses `case` d'un `switch` doivent être des énumérateurs
- chaque `switch` doit avoir une clause `default`
- tous les paramètres des macros doivent être entourés de parenthèses
- les structures qui ne sont utilisées qu'à travers des pointeurs ne doivent pas être incluses dans les fichiers d'en-tête (« .h »)
- lorsqu'une fonction est définie dans un fichier source « .c », le fichier d'en-tête « .h » qui contient sa déclaration doit être inclus
- un fichier d'en-tête « .h » doit se suffire à lui-même
- un fichier d'en-tête « .h » doit avoir un mécanisme pour prévenir les inclusions multiples
- le type de retour d'une fonction doit toujours être indiqué

Deux règles concernant les arguments des fonctions, n'ont pas toujours, à tort, été appliquées :

- l'attribut `const` sera utilisé si un pointeur ou une variable n'est pas modifié
- si un paramètre pointeur est utilisé en tant que tableau dans la fonction, le paramètre doit être déclaré avec des crochets

7.3.3 Assertions

Les assertions sont des conditions qui doivent toujours être remplies.
Elles servent :

- à la mise au point du programme ;
- à la vérification du programme ;
- à l'auto-documentation du programme.

Les assertions jouent un rôle important dans la mise en œuvre de la programmation par contrat.
Chaque fonction doit répondre à un contrat. Une fonction reçoit des paramètres en entrée qui doivent répondre à certaines contraintes, et doit effectuer les traitements attendus par la fonction appelante.
On peut donc définir trois types de conditions devant être remplies :

- pré-condition : condition devant toujours être remplie à l'entrée de la fonction ;
- post-condition : condition devant toujours être remplie à la fin d'une fonction ;
- invariant : condition devant toujours être remplie à n'importe quel endroit de la fonction.

Grâce aux assertions, le programme se teste de lui-même lors de son exécution !

Le C ANSI déclare une macro-instruction `assert()` permettant de vérifier si une condition est vraie. Si la condition n'est pas vérifiée, le programme est interrompu et un message d'erreur indique le fichier et la ligne où la macro `assert()` a été appelée. Le Préprocesseur fait un usage fréquent des assertions, en particulier pour vérifier les pré-conditions.

Si la macro `NDEBUG` est définie à la compilation, tous les `assert()` se réduisent à une instruction nulle. C'est le cas pour les options de compilation utilisées par défaut. Par conséquent, pour la version « principale » installée, l'utilisateur ne sera pas pénalisé par les tests d'assertion, mais il pourra en bénéficier en cas de problème avec une version « DEBUG ».

7.3.4 Constantes nommées

L'utilisation de constantes numériques nommées en lieu et place de constantes littérales rend les modifications des valeurs des constantes plus fiables et aisées. Elle facilite donc largement la maintenance ultérieure du programme, ainsi que son extension.

Leur définition en C prendra l'une des formes suivantes :

- constante énumérée, pour une constante appartenant à un ensemble de constantes liées sémantiquement et pouvant ainsi former une liste d'énumérations
- une macro du pré-processeur.

En particulier, les constantes nommées apparaîtront systématiquement dans les `case` des `switch`. Les constantes énumérées sont préférées, car elles apparaissent sous leur nom (et non valeur) sous un débogueur, et certains compilateurs (tels que GCC) peuvent afficher des messages d'avertissement lorsque certaines valeurs d'une constante énumérée ne son pas traitées dans un `switch`.

7.3.5 Internationalisation

L'internationalisation des messages se fait au moyen du mécanisme de type `gettext()`. Les messages on été passés en Anglais dans le code source, et la version Française est disponible via un catalogue de traductions `pofr.po`. Ceci a l'avantage avec la version GNU de `gettext()` de fonctionner aussi bien avec un environnement Latin-1 (ou Latin-9 ou Latin-15) qu'avec une environnement « Unicode » de type UTF-8 (à condition que l'encodage utilisé par le terminal soit consistant avec la variable d'environnement `LANG`, soit en général `LANG=fr_FR` ou `LANG=fr_FR.UTF-8`.

On doit donc encapsuler les déclarations de chaînes de caractère immédiatement traduisibles dans une macro `_()` (correspondant à une abbréviation pour `gettext()` si ce mécanisme est disponible sur la machine cible,

ou à une macro vide sinon). On encapsulera les chaînes dont la traduction ne peut se faire à l'initialisation (i.e. déclarées via des constantes) dans une macro `N_( )` (macro vide permettant le repérage de ces chaînes par les outils de gestion des catalogues de traduction), en n'ommettant pas d'encapsuler la variable résultante dans la macro `_( )` lors de son utilisation.

Si une chaîne n'est pas traitée de la sorte ou que sa traduction n'est pas disponible, la chaîne non traduite sera utilisée. Les conséquences ne sont donc pas catastrophiques.

On notera qu'avec des chaînes de caractères de type UTF-8, les caractères accentués sont représentés sur plusieurs octets, alors que les caractères ASCII de base sont représentés sur un seul octet. La fonction C `strlen()` permet donc de connaître la taille d'un chaîne en octets, mais pas de dire combien de colonnes elle occupe pour l'affichage. On utilisera la fonction `ecs_print_padded_str()` pour imprimer une telle chaîne en la complétant automatiquement par le bon nombre de caractères blancs lorsqu'on souhaite occuper une largeur de colonne donnée.

7.3.6 Fonctions utilitaires

Aux fonctions de la librairie C standard, on préférera, quand elles existent, les fonctions (ou les macros) de la librairie BFT ou du Préprocesseur qui les encapsulent.
Ces fonctions ont l'avantage d'intégrer les tests de retour des fonctions de la librairie, en produisant les messages d'erreur appropriés en cas d'erreur.

Le tableau ci-dessous recense les principales fonctions de la librairie C standard qui ont une fonction utilitaire correspondante dans le Préprocesseur ou dans la librairie BFT. Cette librairie propose ou « améliore » (au sens de l'écriture d'un code scientifique) d'autres services, et dispose de sa propre documentation en ligne.

Fonction de la librairie C	Fichier en-tête système	Macro ou fonction utilitaire	Fichier Préprocesseur
`exit()`	`stdlib.h`	`ecs_exit()`	`ecs_exit.c`
`fopen()`	`stdio.h`	`bft_file_open()`	`bft_file.h`
`fclose()`	`stdio.h`	`bft_file_free()`	`bft_file.h`
`fprintf()`	`stdio.h`	`bft_printf()`	`bft_printf.h`
`malloc()`	`stdlib.h`	`BFT_MALLOC()`	`bft_mem.h`
`realloc()`	`stdlib.h`	`BFT_REALLOC()`	`bft_mem.h`
`free()`	`stdlib.h`	`BFT_FREE()`	`bft_mem.h`

8 Compilation et aide au débogage

8.1 Compilation

8.1.1 Configuration pour la compilation

Le Préprocesseur utilise maintenant le système de configuration et de compilation GNU (Autoconf / Automake / Libtool), qui possède sa propre documentation.

Pour obtenir la liste des options de configuration disponible, dans le répertoire racine du Préprocesseur, il suffit de taper la commande : `./configure --help`.

La plupart des systèmes actuels disposent d'une commande `make` capable de gérer la compilation depuis un répertoire différent de celui des sources (support `VPATH`). Si c'est le cas sur le système utilisé, il est conseillé de configurer le Préprocesseur depuis un répertoire différent de son arborescence, afin de laisser celle-ci inchangée par rapport à la référence. Sinon, on peut toujours installer `gmake`, le `make` du système GNU (le standard sous Linux) pour disposer de cette fonctionnalité.

Par exemple, si l'arborescence du Préprocesseur se situe sous
`/home/saturne/Enveloppe/ecs-2.0.0.src`
les commandes suivantes permettront sa configuration en mode « debug » et son installation sous
`/home/saturne/Enveloppe/ecs-2.0.0/arch/Linux` :

```
> cd /home/saturne/Enveloppe
> mkdir ecs_build
> cd ecs_build
> ../ecs-2.0.0.src/configure \
 --prefix=/home/saturne/Enveloppe/ecs-2.0.0/arch/Linux \
 --enable-debug
> make
> make install
> make clean
```

Les options de `./configure` les plus utiles pour le Préprocesseur sont les suivantes :

Option	Fonction
`--enable-debug`	compilation en mode « debug » plutôt qu'en mode optimisé
`--with-bft-prefix=`<chemin>	chemin d'installation de la librairie BFT (en général nécessaire si différent d'un chemin système)
`--with-hdf5-prefix=`<chemin> `--with-med-prefix=`<chemin> `--disable-med`	chemin d'installation des librairies HDF5 et MED pas de support MED même si les librairies sont détectées
`--with-cgns-prefix=`<chemin> `--with-cgns-exec-prefix=`<chemin> `--disable-cgns`	chemin d'installation de la librairie CGNS chemin d'installation de la partie dépendante du système de la librairie CGNS (si différent du précédent) pas de support CGNS même si la librairie est détectée
`--with-metis-libdir=`<chemin> `--disable-metis`	chemin auquel se trouve la librairie METIS (`libmetis.a`) pas de support METIS même si la librairie est détectée

Si la version de `make` utilisée ne supporte pas la logique `VPATH`, alors on sera limité à la compilation depuis l'arborescence source (dans quel cas on pourra utiliser `make distclean` plutôt que `make clean` pour ramener le répertoire à l'état le plus proche possible de son état initial :

```
> cd /home/saturne/Enveloppe/ecs-2.0.0.src
> ./configure --prefix=/home/saturne/Enveloppe/ecs-2.0.0/arch/Linux \
 --enable-debug
> make
> make install
> make distclean
```

Les plates-formes sur lesquelles le Préprocesseur est régulièrement compilé sont mentionnées dans le tableau ci-dessous.

Plate-forme	`'uname -s'`
IBM Power 5 (Frontal Blue Gene/L)	`Linux`
Station Linux (Debian Sarge et Etch)	`Linux`
Cluster AMD Opteron (Red Hat EL 4, Chatou, CCRT)	`Linux`
Cluster Itanium II (Bull Novascale, CCRT)	`Linux`
SUN Sparc (Solaris 8)	`SunOS`

Le Préprocesseur a été utilisé régulièrement mais n'est plus maintenu sur les plateformes suivantes, qui ne sont plus utilisées ou disponibles :

Plate-forme	`'uname -s'`
Alpha EV68 sous Tru64 Unix (HP Alphaserver, CCRT)	`OSF1`
SGI (Irix 6.5)	`IRIX64`
HP 9000 (HP-UX 10.2, HP-UX 11)	`HP-UX`
PC Linux (Red Hat 7.2, Debian 2.2)	`Linux`
Cluster PC Linux MFEE (Red Hat 8.0)	`Linux`
Cluster PC Linux MFEE (Red Hat 7.2 avec)	`Linux`
Fujitsu VPP5000	`UNIX_System_V`

Finalement le *Code_Saturne* a été compilé et testé à l'occasion sur de nombreuses architectures, dont :

- de nombreuses variantes de Linux sous PC (Red Hat 8.0, Fedora Core 4, SuSE 7.1 à 9.1, SUSE 9.2 à 10.2 Mandrake 9 et 10, etc.) avec diverses versions du compilateur GCC (2.95, 2.96, 3.1 à 3.4, 4.0.2 à 4.3.1) ;
- sur un machine SGI Altix (Linux avec patchs SGI, processeur Intel Itanium II, compilateur Intel).

8.1.2 Options de compilation

Il est possible de choisir le compilateur C et l'éditeur de liens au moyen de la variable d'environnement `CC`, comme suit :

```
> ./configure --prefix=<chemin> CC=<compilateur>
```

De même, il est possible d'utiliser d'autres options de compilation et d'édition de liens que celles fournies par défaut, en renseignant les variables d'environnement `CPPFLAGS` (préprocesseur C), `CFLAGS` (compilation), et `LDFLAGS` (édition de liens) lors de l'appel à `configure`.

On peut aussi, au lieu de remplacer les options par défaut, les compléter, avec les variables d'environnement `CPP_ADD`, `CC_ADD`, `LD_ADD`, et `LIBS_ADD`.

8.1.3 Variables d'environnement dépendantes du système d'exploitation

Il existe des variables qui dépendent du système et qui sont positionnées automatiquement dans le fichier `ecs_config.h` généré par le script `configure` et inclus dans les sources.

On donne ci-dessous, le rôle joué par chacune de ces variables, lorsqu'elles sont définies :

Variable	Rôle
_POSIX_SOURCE	(Variable déclarée par la norme POSIX) Utilisation de fonctions POSIX
_XOPEN_SOURCE	(Variable déclarée par la norme X-OPEN) Utilisation de fonctions X-OPEN
_XOPEN_SOURCE_EXTENDED	(Variable déclarée par la norme X-OPEN2) Utilisation de fonctions X-OPEN2
HAVE_CGNS	permet l'utilisation de la librairie CGNS
HAVE_MED	permet l'utilisation de la librairie *MED*
HAVE_METIS	permet l'utilisation de la librairie *Metis*

8.2 Mémoire

Une fonctionnalité du Préprocesseur donne la possibilité de retranscrire l'état de la mémoire à chaque allocation ou libération d'un bloc mémoire. Ceci est utile surtout si l'on ne dispose pas d'un outil comme *Purify* ou *Valgrind*.

Cette fonctionnalité est prise en compte dans le Préprocesseur chaque fois que la variable d'environnement CS_PREPROCESS_MEM_LOG a été positionnée à l'exécution.

Cette fonctionnalité réalise automatiquement :
- pour chaque allocation :
  - l'impression sur fichier de la taille mémoire allouée ;
  - l'impression sur fichier du cumul de la mémoire allouée ;
- pour chaque libération :
  - l'impression sur fichier de la taille mémoire libérée ;
  - la vérification que le bloc libéré a bien été alloué ;

8.3 Impression de la structure Maillage

Le Préprocesseur offre la possibilité de retranscrire sur fichier le contenu d'une structure ecs_maillage_t ainsi que de toutes ses structures composantes.

La retranscription est faite par défaut dans le fichier

```
ecs_impression.ascii
```

chaque fois que la fonction

```
ecs_maillage__imprime()
```

est appelée (soit avant et après les principales étapes du traitement), à condition d'ajouter l'option documentée uniquement ici -dump [n] à la ligne de commande du Préprocesseur, où *n* indique que l'on imprime les *n* premiers et derniers éléments de chaque tableau (0 par défaut).

Cette fonction prend pour argument :
- la structure ecs_maillage_t à imprimer ;
- le nom du fichier où est faite l'impression ;
- un titre dont le rôle est de séparer différentes impressions de la structure ecs_maillage_t pour différents appels.

```
void ecs_maillage__imprime
(
 const ecs_maillage_t *this_maillage,
 const char           *nom_fichier_dump,
 const char           *titre
)
```

9 Perspectives de développement

Par rapport aux versions 1.0 à 1.2 de l'Enveloppe, qui assurait le post traitement des champs calculés par le Noyau, et servait d'intermédiare entre celui-ci et le code SYRTHES en cas de couplage, le rôle de l'Enveloppe s'est considérablement réduit avec le version 1.3, les aspects post traitement et couplage étant maintenant directement gérés par le Noyau, en s'appuyant sur la librairie FVM (d'où le renommage de l'Enveloppe en Préprocesseur).

On devrait pouvoir encore migrer certaines des fonctionnalité du Préprocesseur vers les Noyau, ce qui permettra de simplifier progressivement les structures et modèles du Préprocesseur, dont le remplacement complet par FVM nécessitera plusieurs années.

9.1 Améliorations du codage du Préprocesseur

La traduction du code source en Anglais et la mise en cohérence de la mise en page avec le Noyau et la librairie FVM sont souhaitables ; ce travail est lourd, mais peut être effectué « au fil de l'eau ».

9.2 Evolutions du code

On établit ici la liste des nouvelles évolutions à apporter, dans l'ordre de leur priorité à ce jour :

1. modifier la représentation des cellules de type « polyèdre » en connectivité nodale : plutôt que de repérer les frontières entre faces dans la connectivité cellules sommets en considérant que le premier sommet d'une face apparaît une deuxième fois pour indiquer une fin de face, on pourrait utiliser une connectivité secondaire (un deuxième champ principal de type définition) indiquant pour chaque polyèdre le nombre de sommets associés à chacune de ses faces. Ce tableau indexé n'existerait qu'en présence de polyèdres en connectivité nodale.

   On pourrait alors construire un maillage nodal de type FVM en pointant sur les diverses zones de la définition principale principale de la connectivité, sans devoir dupliquer l'essentiel de la définition $faces \rightarrow sommets$, ce qui est important du point de vue consommation mémoire en cas de maillage polyédrique pur ; les connectivités $cellules \rightarrow faces$ devraient être renseignées pour les sections polygonales, ainsi que l'index de la connectivité $faces \rightarrow sommets$, mais ne représentent pas l'essentiel du coût mémoire (par exemple, pour un polyèdre avec 12 faces de 5 sommets chacune, on a besoin de $(1 + 12) + 12 = 25$ entiers pour ces tableaux, contre $12 \times 5 = 60$ entiers pour la définition $faces \rightarrow sommets$).
2. Simplification de la structure globale. Il serait possible d'intégrer directement les fonctionnalités de type *vecteur indexé entier* et *vecteur indexé réel*, dans les structures `ecs_champ_t`, mais il faudrait dans ce cas réfléchir au problème du typage du champ. Un « cast » initial pour accéder au tableau des valeurs n'est pas très pratique, il semble préférable soit de nommer différemment un tableau de valeurs entières d'un tableau de valeurs réelles, en groupant éventuellement de même les deux pointeurs correspondant dans une union anonyme afin de n'en permettre qu'un à la fois.
3. Renommer `ecs_champ_t` en `ecs_table_t` (et Champ en Table), pour une meilleure cohérence avec le vocabulaire usuel.

10 Bibliographie

[1] Y. FOURNIER
Code_Saturne version 1.1 : guide pratique et théorique du module Enveloppe,
Rapport EDF HI-83/03/007/A, 2003

[2] D. POIZAT, Y. FOURNIER
Enveloppe du Code_Saturne : description de la version 1.0,
Rapport EDF HI-83/01/013/A, 2001

[3] BOUCKER M., ARCHAMBEAU F., MÉCHITOUA N.,
Quelques éléments concernant la structure informatique du Solveur Commun - Version 1.0_init0,
Compte-rendu express EDF I81-00-8, 2000.

[4] Y. FOURNIER
Définition du module Enveloppe pour le Solveur Commun,
Rapport EDF HE-41/99/017/A, 1999

[5] MÉCHITOUA N., ARCHAMBEAU F.,
Prototype de solveur volumes finis co-localisé sur maillage non-structuré pour les équations de Navier-Stokes 3D incompressibles et dilatables avec turbulence et scalaire passif,
Rapport EDF HE-41/98/010/B, 1998.