Application de la POO pour la conception d'un logiciel de simulation par éléments finis en mise en forme des matériaux

Jean-Claude Gelin — Pascal Paquier — Luc Walterthum

Laboratoire de Mécanique Appliquée Raymond Chaléat
UMR. CNRS. 6604
Université de Franche-Comté
24, rue de l'Epitaphe
F-25030 Besançon

RÉSUMÉ. *La programmation orientée objet (POO) appliquée à la conception d'un logiciel de simulation en mise en forme des matériaux est ici présentée. Dans un premier temps, un bref historique et un rappel des concepts fondamentaux de la POO sont énoncés. L'adéquation de la POO avec la méthode des éléments finis est ensuite démontrée dans le cadre d'un logiciel de simulation de procédés de mise en forme des matériaux programmé en langage C++ dont l'architecture de base est décrite. Plusieurs exemples de simulations numériques sont donnés dans le cadre de problèmes mono ou multi-domaines et mono ou multi-analyses pour illustrer l'évolutivité qu'offre la POO.*

ABSTRACT. *The object oriented programming (OOP) applied to the definition of the structure of a finite element software for modelling material forming processes is presented. In a first part, a brief report on the evolution of programming techniques is given followed by the principles of OOP. Then the adequation of OOP is demonstrated for a simulation software used for material forming processes written in C++. The basic structure of the software is given and several examples are presented giving the possibilities of OOP for multi-analyses and multi-domains problems.*

MOTS-CLÉS : *méthode des éléments finis, programmation orientée objet, langage C++, procédés de mise en forme des matériaux.*

KEY WORDS: *finite elements, object oriented programming, C++ language, material forming processes.*

1. Introduction

Née avec l'apparition des ordinateurs et avec l'essor des sciences pour l'ingénieur et plus particulièrement de la mécanique des structures, la méthode des éléments finis (MEF) n'a cessé de se développer depuis 1960 et est désormais utilisée dans de nombreux logiciels de simulation numérique [ZIE 91].

Sa mise en œuvre informatique privilégia le FORTRAN, langage le plus utilisé dans la communauté scientifique de l'époque qui dut essentiellement sa popularité grâce à ses fonctionnalités réduites qui permettaient un développement rapide et efficace.

Cependant ce langage modulaire très efficace ne possédait aucune structure de données de haut niveau. Vint ensuite le langage C qui permettait de mieux structurer les développements. Mais le FORTRAN, tout comme le langage C, se révélèrent inefficaces face à l'évolution de la complexité des problèmes physiques à traiter car ils ne répondaient plus à deux critères fondamentaux pour tout développement de codes de calculs, à savoir la facilité de maintenance et l'assurance de l'extensibilité.

Des extensions fructueuses, toujours basées sur le FORTRAN, ont été réalisées pour pallier les problèmes inhérents associés au langage et des modules ont été écrits pour simuler quelques-unes des fonctionnalités des langages orientés objets (allocation dynamique de la mémoire, notion d'objets, ...). On peut notamment citer le code modulaire SIC [BRE 92] ainsi que le code CASTEM 2000 [VER 88].

La programmation orientée objet (POO) est née avec le concept du génie logiciel visant à l'uniformité des logiciels au niveau de l'architecture et à la maîtrise du développement des logiciels complexes et de grande taille [BOO 93]. Grâce à son grand niveau d'abstraction, elle présente désormais un intérêt indéniable pour le développement de codes de calcul utilisant la méthode des éléments finis.

L'introduction des concepts de la POO dans les codes de calcul par éléments finis a commencé il y a une dizaine d'années. Les premiers articles en la matière décrivent les entités de base des codes de calcul par éléments finis (nœud, élément, ddl) ainsi que les objets mathématiques associés [MIL 88] [FOR 90] [FEN 90] [MIL 91] [SCH 92].

La structure des logiciels de simulation par éléments finis a été abordée dans les séries d'articles de Zimmermann et al. [ZIM 92] [DUB 92, 93] qui décrivent la méthodologie à suivre pour la réalisation de logiciels orientés objets.

Un certain nombre d'équipes ont ensuite appliqué les concepts de la POO pour la réalisation de logiciels de simulation de large envergure, dans le domaine de la mécanique des solides et des matériaux [AAZ 93] [BES 97] ou de la simulation en mise en forme des matériaux [GEL 95, WAL 96].

De nouvelles techniques on été introduites récemment dans le domaine de la programmation automatique dans le cadre de logiciels de simulation par éléments finis [EYH 95, 96a, 96b].

Le présent article donne un bref aperçu de la POO tout en en rappelant les finalités et les concepts de base. L'utilisation des avantages des nouvelles fonctionnalités de l'approche orientée objet est ensuite présentée dans le cadre d'un logiciel de simulation numérique de procédés de mise en forme des matériaux qui a

été développé en langage C++ au sein du Laboratoire de Mécanique Appliquée de Besançon.

L'architecture générale de ce logiciel est ainsi décrite à travers les différentes étapes classiques de la résolution de problèmes de base, à savoir les problèmes mono-domaine et mono-analyse.

L'extension aux problèmes complexes comme les problèmes multi-domaines et multi-analyses tels que le moulage par injection d'un fluide incompressible ou le problème de couplage thermo-mécanique est abordée et illustrée.

2. La programmation orientée objet (POO)

2.1. *Introduction*

Née du langage SIMULA (1965), l'approche objet dédiée au génie logiciel offre une toute nouvelle méthode pour la conception et la programmation qui répond à plusieurs critères fondamentaux :

- Fiabilité
- Extensibilité
- Efficacité
- Intelligibilité

La POO s'est concrétisée à travers divers langages de programmation orientés objets (LOO) comme Simula (1965), Smalltalk (années 70), Ada (fin des années 70), Object Pascal (début des années 80), C++ (début des années 80) ou encore le très récent langage Java (1995).

Il existe plusieurs méthodologies de conception orientée objet. Les principales sont HOOD [LAI 91], OMT [RUM 91] et BOOCH [BOO 93]. Les méthodes OMT et BOOCH sont adaptées au C++ et sont assez similaires, alors que la méthode HOOD est plutôt adaptée au langage ADA. Les développements relatés ici sont basés sur la méthode BOOCH.

2.2. *Concepts de base et avantages de la POO*

2.2.1. *Objet et classe*

La POO est basée sur la notion centrale d'*objet* qui associe dans une même entité informatique données et traitements définis au sein d'une *classe*. Chaque objet est une *instance de classe* et est défini par ses caractéristiques ou *attributs* et par des actions appelées encore *méthodes* ou messages. Un objet est donc une entité dont l'état varie en fonction des messages qu'il reçoit. L'ensemble des opérations/actions/messages appelées méthodes que l'on peut effectuer sur un objet, définit son interface.

2.2.2. *Principe d'encapsulation*

Les détails d'un objet ne sont accessibles que par ses opérations visibles (méthodes) et ses attributs sont cachés. Par conséquent, toute modification reste locale aux objets ce qui accroît la modularité et l'évolutivité d'un programme.

2.2.3. *Niveaux de visibilité externe des méthodes et des attributs : public et private*

Les méthodes publiques constituent l'interface de la classe et sont accessibles par tout utilisateur de la classe. Les méthodes privées complètent les méthodes publiques et ne sont accessibles qu'à l'implémenteur de la classe.

Il en est de même pour les attributs. Ceux déclarés *private* sont accessibles et modifiables uniquement au sein des méthodes et ceux déclarés *public* le sont de l'intérieur comme de l'extérieur. Dans le deuxième cas, le principe d'encapsulation est brisé puisque les attributs ne sont plus protégés.

2.2.4. *Héritage*

L'héritage constitue avec le principe d'encapsulation l'apport essentiel de la POO et consiste en un mécanisme de transmission des propriétés d'une classe vers une sous-classe. Les notions de classe mère et classe dérivée en découlent.

Une classe dérivée hérite des propriétés de la classe supérieure mais peut cependant posséder ses propres propriétés.

L'héritage peut être simple ou multiple selon qu'une classe hérite d'une seule classe ou de plusieurs classes.

2.2.5. *Polymorphisme*

Il est possible de redéfinir une opération héritée pour pouvoir lui donner une implémentation différente (code source différent) pour tenir compte des propriétés locales non héritées. L'opération redéfinie a la même signature que celle de la classe supérieure excepté le fait qu'elle s'applique aux instances de la classe dérivée. Une opération redéfinie peut donc prendre différentes formes ou implémentations en fonction du type d'objet auquel elle s'applique, d'où la notion de polymorphisme.

2.2.6. *Méthode virtuelle et liaison dynamique (dynamic binding)*

Le concept de méthode virtuelle est étroitement lié à celui de l'héritage. Une méthode déclarée virtuelle dans une classe de base ne doit pas forcément être redéfinie dans toutes les classes dérivées. Si elle ne l'est pas pour une classe dérivée, la fonction appelée pour les instances de cette classe sera celle de la classe de base par défaut. Ce principe permet de mettre en place le mécanisme de liaison dynamique à savoir que la méthode (déclarée virtuelle dans la classe de base) appelée correspond à la classe de l'objet. Le choix de la fonction est effectuée au moment de l'exécution et non plus au moment de la compilation.

2.2.7. *Classe abstraite*

Une classe abstraite est une classe de base dont une méthode au moins est déclarée virtuelle pure. De ce fait, il n'est plus possible d'instancier des objets de cette classe et cette fonction doit être redéfinie dans toutes les classes dérivées. Le rôle de ce type de classe est de donner naissance à d'autres classes par héritage et de regrouper les méthodes communes de traitement pour toutes ses classes dérivées.

3. La POO appliquée à la méthode des éléments finis

3.1. *Introduction*

Quelques avantages de la POO pour la méthode des éléments finis sont ici illustrés à travers ses concepts de base et les différents critères de qualité requis pour un grand code de simulation.

3.2. *Intelligibilité*

L'intelligibilité d'un logiciel passe avant tout par une bonne lisibilité de celui-ci. La méthode des éléments finis nécessite beaucoup de calculs vectoriels, matriciels voire tensoriels. La transcription de ces calculs est souvent fastidieuse. Afin de pallier ce problème, le langage C++ permet de surdéfinir la plupart des opérateurs arithmétiques, d'affectation pour les adapter à n'importe quel objet. Ainsi la notation suivante :

$$A = B * C$$

peut s'appliquer de façon totalement transparente aux types de base classiques (int, float, double) mais surtout à des vecteurs, matrices ou tenseurs à partir du moment où les opérateurs = et * sont surdéfinis dans toutes les classes d'appartenance des objets. De ce fait la lisibilité d'un code de calcul est grandement améliorée car l'implémentation des calculs est de plus en plus proche de la formulation théorique.

De plus, le polymorphisme et le typage dynamique permettent l'écriture de méthodes communes qui s'appliquent à tous les objets des classes dérivées d'une classe mère. Tout en gardant une formulation générique, le traitement interne peut être spécifique à chaque objet. Ainsi, il est possible d'avoir ce type d'écriture générale pour le calcul des matrices de rigidité élémentaire :

```
for (int i = 0; i < nombre_elements; i++)
{
  ELEMENT* element = p_element[i];
  element->Calculer_Rigidite();
}
```

quelle que soit la formulation éléments finis choisie.

3.3. *Fiabilité*

La fiabilité d'un code dépend de ses possibilités de gestion et de détection d'erreurs. La surdéfinition d'opérateurs tout en assurant la lisibilité d'un programme, réduit les risques d'erreurs. De plus, en autorisant la surdéfinition des opérateurs () et [] elle permet le contrôle total de l'indexation utilisée dans un programme et supprime les risques de dépassement de bornes.

Autre atout en faveur de la fiabilité, le principe d'encapsulation qui permet de protéger totalement les attributs d'un objet en les déclarant *private*. De ce fait, ces attributs sont uniquement accessibles par les méthodes. Toute modification d'un attribut d'un objet étant restreinte au niveau des méthodes, les risques d'erreurs sont moindres d'autant plus que chaque méthode accède directement aux attributs d'un objet sans qu'il soit nécessaire de les passer comme arguments.

3.4. *Modularité*

De par le principe d'encapsulation, les classes sont indépendantes entre elles et peuvent jouer le rôle de modules. Chaque classe peut être testée séparément ce qui facilite les actions localisées et le travail d'équipe.

3.5. *Extensibilité*

L'extensibilité est probablement la propriété fondamentale de tout code de calculs puisque sa pérennité en découle directement. Un code évoluant difficilement est voué à une mort certaine ou à une utilisation restreinte.

L'héritage et le polymorphisme sont les clés de l'extensibilité d'un code de calcul puisqu'ils permettent de greffer de nouvelles fonctionnalités par dérivation de classes de base constituant le cœur de l'architecture.

Un code de calculs utilisant la méthode des éléments finis est susceptible d'évoluer à différents niveaux comme la formulation des éléments finis, la définition des paramètres physiques ou matériels ou encore le choix de l'algorithme de résolution. En prenant le cas de la formulation éléments finis, il est naturel de définir une classe de base FORMULATION qui contient tous les traitements communs comme le calcul de la rigidité ou le calcul de la masse et d'en faire dériver tous les différents éléments finis dans lesquels sont redéfinies toutes les méthodes de base comme le montre la figure 1.

Chaque nouvel élément vient s'ajouter par dérivation de la classe de base FORMULATION et son implémentation spécifique s'effectue au niveau de ses fonctions membres locales qui surdéfinissent celles de la classe de base.

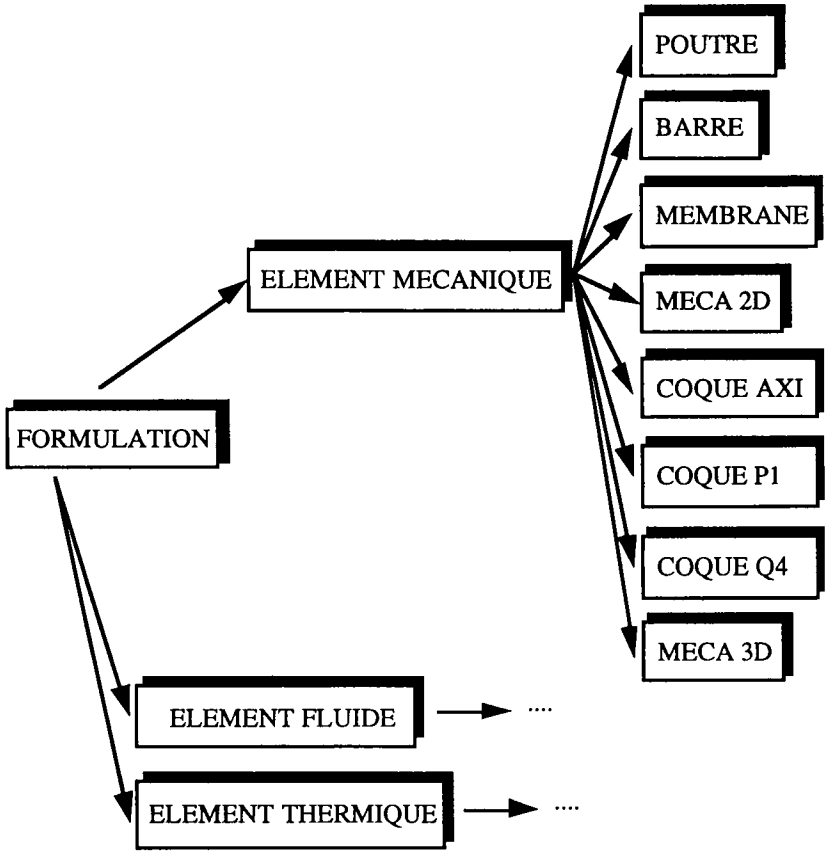


Figure 1. *Classe FORMULATION*

3.6. *Disponibilité d'outils pour le stockage d'objets*

Il existe déjà des bibliothèques orientées objet écrites en langage C++ dans lesquelles des classes conteneurs sont disponibles et offrent de nombreuses possibilités de stockage d'objets. Notre choix initial s'est porté sur la librairie LEDA [NÄH 95] qui est l'une des plus efficaces et des évolutions sont prévues vers STL [MUS 96]. Ces bibliothèques comportent les classes conteneurs suivantes:

- array<T> tableau monodimensionnel
- array2<T> tableau bidimensionnel
- list<T> liste doublement chaînée
- set<T> collection d'objets de même type
- dictionary<T> dictionnaire à accès rapide mais par valeur des objets stockés
- d_array<T> dictionnaire à accès direct à la référence d'un objet

D'autres classes pour stocker des grandeurs classiques sont utilisées :

- VECTEUR
- MATRICE
- String pour les chaînes de caractères

3.7. *Richesse des relations entre les différentes classes*

Deux types de relations au niveau des différentes classes sont utilisées :

- les relations de dérivation,
- les relations de « contenu » à « conteneur ».

Le principe de dérivation est utilisé pour l'ajout de nouvelles fonctionnalités à partir d'une classe de base et pour le maintien d'une formulation générique à l'aide du polymorphisme et du typage dynamique.

L'autre type de relation se concrétise par la déclaration de listes, tableaux, dictionnaires d'objets, voire de pointeurs d'objet comme attributs d'une autre classe. La classe FORMULATION contient, par exemple, divers objets nécessaires à la formulation éléments finis :

```
class FORMULATION
{
   ...
   MAILLE*                 p_geometrie;
   POINT_INTEGRATION*      p_point_integration;
   MATERIAU*               p_materiau;
   MATRICE                 p_coordonnees;
   MATRICE                 p_rigidite;
   MATRICE                 p_masse;
   VECTEUR                 p_efforts_externes;
   VECTEUR                 p_efforts_internes;
   VECTEUR                 p_solution;
   ...
}
```

4. Architecture orientée objet d'un logiciel de simulation en mise en forme des matériaux

4.1. *Introduction*

La conception d'une architecture orientée objet a été réalisée pour un logiciel de simulation de procédés de mise en forme des matériaux [GEL 95, WAL 96]. Ce

code de simulation est programmé en langage C++ pour des raisons de performances en temps de calcul et des raisons de portabilité, il répond aux exigences suivantes :

- possibilité de traiter des procédés de mise en forme de natures très diverses,
- gestion des problèmes complexes multi-domaines et multi-analyses,
- facilité d'extensibilité.

Les simulations réalisées dans le logiciel impliquent la définition de plusieurs domaines et la gestion de leurs interactions. Les différents domaines ne peuvent cependant pas être gérés de façon identique. Les outillages, par exemple, ne nécessitent pas toujours d'être discrétisés pour les simulations et leur gestion se limite à celle de leur déplacement ou à l'évolution d'un paramètre telle la température.

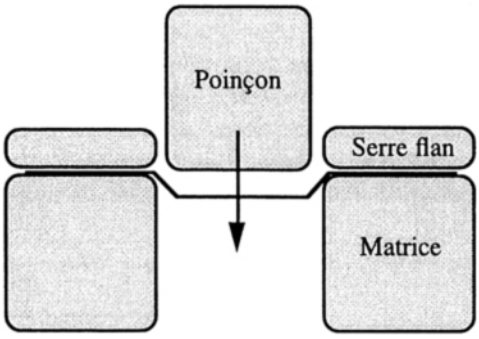


Figure 2. *Illustration schématique d'un cas d'emboutissage*

Ainsi, dans le cas de la simulation de l'emboutissage des tôles minces, les outils (poinçon, matrice, serre flan) sont simplement considérés comme des contacteurs rigides pouvant être animés de mouvements de solides (figure 2). Dans ce cas le problème est mono-analyse avec un domaine déformable et plusieurs contacteurs rigides.

Dans le cas de la simulation de la coupe des métaux, l'outil est généralement discrétisé pour calculer entre autres la distribution de température à l'intérieur de celui-ci, due à l'échauffement associé dans la zone de formation du copeau (figure 3). Il s'agit dans ce cas d'une simulation multi-domaines (matériau usiné, outil) qui peut pour certains des domaines être multi-analyses : dans le domaine correspondant au matériau usiné, on souhaite calculer à la fois les déplacements, vitesses et accélérations, ainsi que la distribution de température.

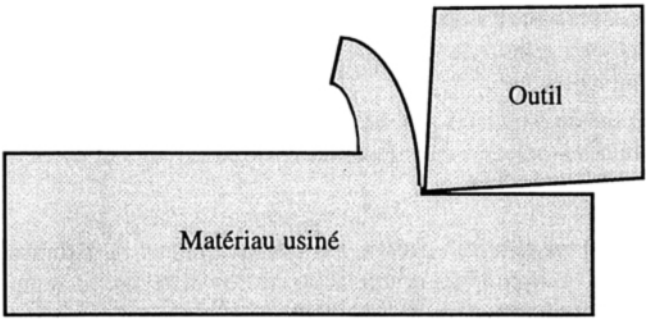


Figure 3. *Illustration schématique d'un cas de coupe orthogonale des métaux*

4.2. *Description globale de l'architecture*

Pour assurer une gestion cohérente des outils et des domaines discrétisés, une classe générique DOMAINE a été créée regroupant les attributs et traitements communs aux différents domaines. Deux autres classes en dérivent : la classe DOMAINE_DISCRETISE (traitant les domaines maillés) et la classe CONTACTEUR traitant des domaines non maillés mais simplement représentés par leurs contours.

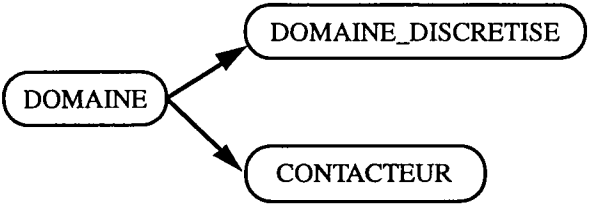


Figure 4. *Création d'une classe de base DOMAINE*

D'autre part, pour la gestion simultanée de plusieurs domaines, une classe MODELE a été ajoutée. Cette dernière contient la liste des domaines et la liste des contacteurs. La première liste permet d'effectuer des traitements communs pour les instances de DOMAINE_DISCRETISE et CONTACTEUR, la seconde permet d'effectuer des traitements spécifiques aux contacteurs. Elle comprend aussi la méthode *Resoudre* qui décrit l'algorithme général de résolution sur les domaines discrétisés ou non (figure 5).

```
class MODELE
{
   Resoudre
   list<DOMAINE*> p_liste_domaines;
   list<CONTACTEUR*> p_liste_contacteurs;
}
```

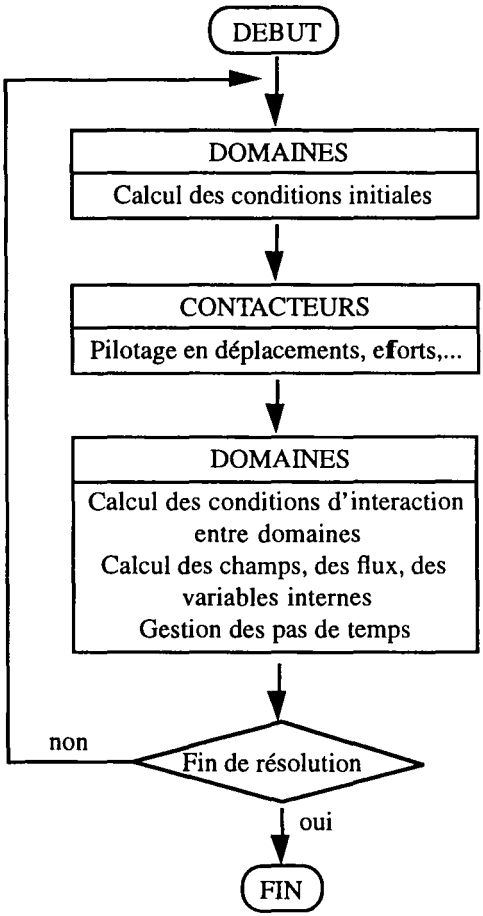


Figure 5. *Algorithme général de résolution*

4.3. Les éléments clés de l'architecture de base

Les points clés de l'architecture de base du code de calcul sont illustrés autour des étapes classiques de la résolution de problèmes élémentaires telles que :

- définition de la géométrie,
- maillage de la géométrie,
- définition des interactions avec l'extérieur,
- discrétisation par éléments finis,
- définition du matériau,
- choix de l'algorithme de résolution.

L'architecture du logiciel s'articule autour de plusieurs classes de base associées à chacune des phases précitées.

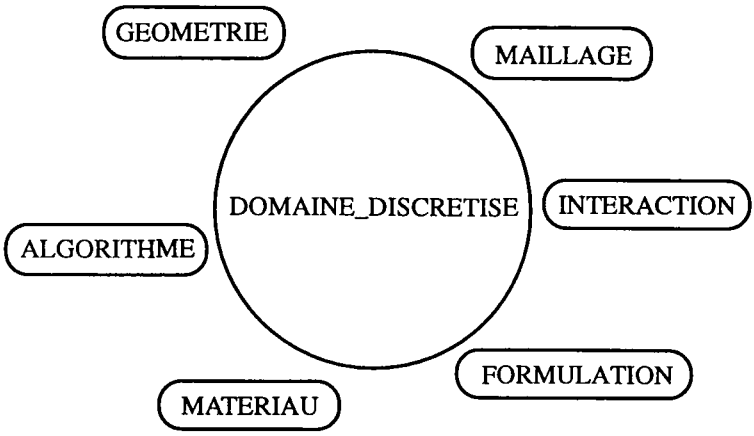


Figure 6. *Architecture de base du logiciel*

Le pilotage de l'ensemble est effectué par la classe DOMAINE_DISCRETISE représentant la modélisation du problème traité. Les attributs et méthodes de la classe DOMAINE_DISCRETISE sont donnés dans le tableau 1.

La classe DOMAINE_DISCRETISE contient tous les attributs correspondant aux différentes étapes de la résolution ainsi que les méthodes pour les réaliser.

Classe **DOMAINE_DISCRETISE**	
Attributs	**Methodes**
Géométrie	Definir_Materiau
Matériau	Definir_type_element
Formulation éléments finis	Ajouter_Interaction
Liste des interactions	Definir_Algorithme
Algorithme de résolution	Calcul de la rigidité
Matrice de rigidité	Calcul de la masse
Efforts externes/internes	Calcul des efforts externes/internes
Réactions	Calcul des flux
Solution	

Tableau 1. *Contenu de la classe DOMAINE_DISCRETISE*

4.3.1. *Définition de la géométrie d'un domaine*

La géométrie d'un domaine est associée à la classe GEOMETRIE définie par un assemblage d'entités géométriques représentées par les classes POINT, SEGMENT, SURFACE et VOLUME. Chaque entité géométrique est constituée d'autres entités plus simples et dérive d'une classe abstraite ZONE (figure 7) comportant les méthodes communes de traitement.

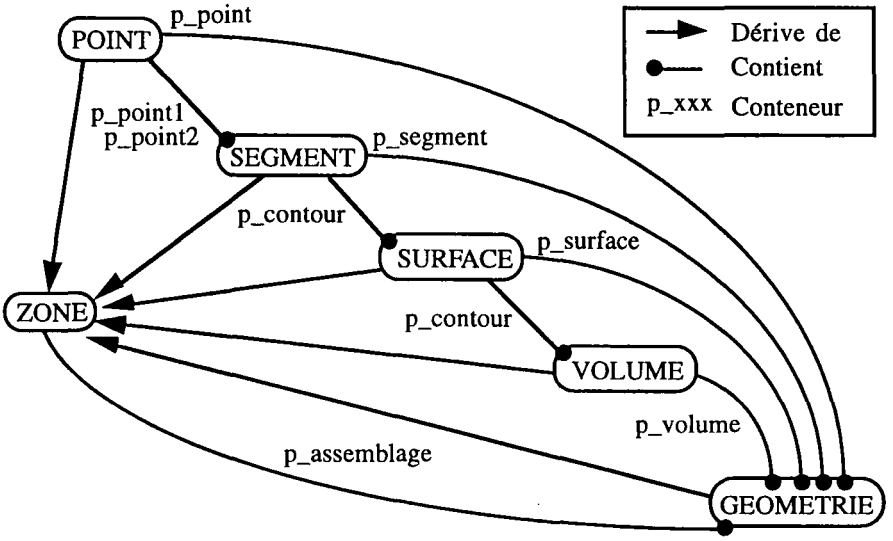


Figure 7. *Organisation de la géométrie*

4.3.2. *Maillage du domaine*

Chaque entité géométrique peut être maillée selon deux techniques : maillage réglé ou maillage libre [BAI 98], implémentées sous forme de fonctions virtuelles pures au sein de la classe abstraite ZONE et redéfinies dans les classes dérivées afin de permettre un traitement spécifique selon la géométrie à mailler (volume, surface, segment ou point). Les différents maillages sont stockés dans une instance de la classe MAILLAGE contenue également dans ZONE.

```
class ZONE
{
   ...
   MAILLAGE p_maillage;
   ...
   virtual void Mailler_Libre() = 0;
   virtual void Mailler_Regle() = 0;
   ...
};
```

Le maillage utilise alors un schéma récursif représenté sur la figure 8.

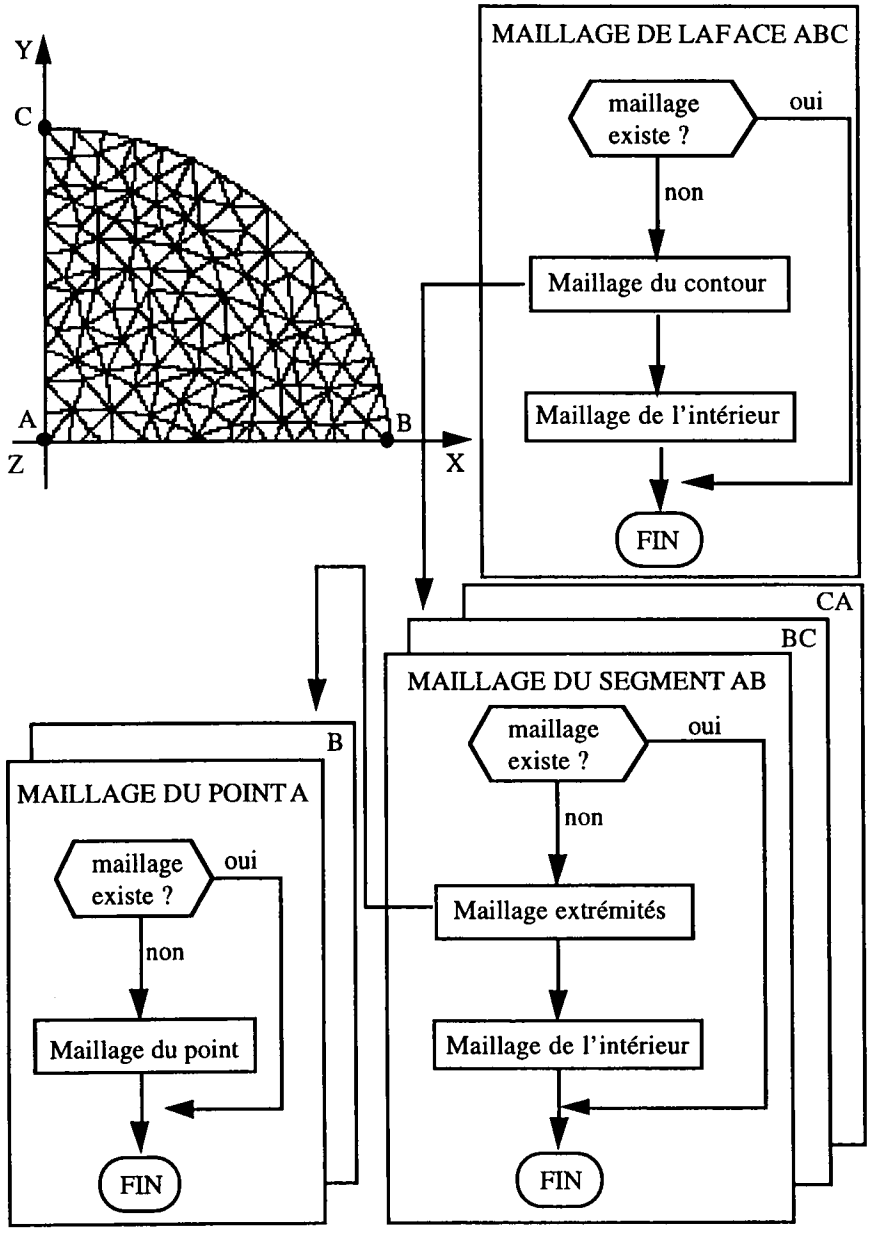


Figure 8. *Principe de fonctionnement du maillage*

Les entités géométriques constituant la liste *p_assemblage* de la classe GEOMETRIE sont balayées une à une et les techniques de maillage se propagent jusqu'aux entités les plus simples (points). De ce fait, les nœuds ainsi que les éléments créés peuvent appartenir à plusieurs maillages relatifs à différentes entités géométriques, ce qui comporte deux avantages :

– chaque entité ayant son propre maillage, les applications de conditions aux limites et de chargement y sont facilitées,

– la frontière commune à deux surfaces ou à deux volumes ne sera maillée qu'une seule fois.

Le maillage est constitué de listes de nœuds et de mailles.

```
class MAILLAGE
{
    ...
    list<NOEUD*> p_liste_nœuds; //nœuds du maillage
    list<MAILLE*> p_liste_mailles;//maille du maillage
    ...
    void Ajouter_Nouveau_Nœud(); //création de nœuds
    void Ajouter_Nouvelle_Maille(); //création de mailles
    ...
};
```

Chaque maille possède une liste de pointeurs sur les nœuds du maillage et est associée à un élément de référence caractérisant l'élément géométrique choisi pour le maillage.

```
class MAILLE
{
    ...
    list<NOEUD*> p_liste_noeuds;//connectivités de la
maille
    ELEMENT_REFERENCE* p_reference;
    ...
}
```

4.4. *Interactions des domaines avec l'extérieur*

Les interactions entre un domaine et l'extérieur sont de deux types :

– les conditions aux limites,

– les chargements.

Ces interactions sont appliquées uniquement sur les entités géométriques (ZONE) définissant le domaine. L'interaction est alors définie par :

– la zone sur laquelle elle s'applique,

– la direction d'application (suivant un ddl, une normale...),

– son évolution dans le temps,

– son évolution dans l'espace,

– le type de répartition dans le cas d'un chargement (répartition nodale, linéique, surfacique ou volumique).

Les classes FCTN_TEMPS et FCTN_ESPACE ont été créées pour représenter des fonctions du temps et de l'espace. Ces fonctions sont constantes, linéaires, quadratiques ou trigonométriques.

Toutes les interactions sont des instances de la classe INTERACTION

```
class INTERACTION
{
   ...
   ZONE* p_zone_application;
   TYPE_DIRECTION p_type_direction;
   int p_direction;
   TYPE_REPARTITION p_type_repartition;
   FCTN_TEMPS* p_fonction_temps;
   FCTN_ESPACE* p_fonction_espace;
   ...
}
```

et sont stockées sous forme de liste au sein du domaine discrétisé qui les gère à l'aide de deux méthodes, l'une pour ajouter des conditions sur des entités géométriques (ZONE) et l'autre pour les introduire au niveau de la matrice de rigidité et des efforts externes par pénalisation.

```
class DOMAINE_DISCRETISE
{
   list<INTERACTION*> p_condition_limite;
   list<INTERACTION*> p_chargement;
   ...
   void Ajouter_Interaction_Sur(...);
   void Introduire_Conditions_Limites_Dans(...);
}
```

5. Discrétisation par éléments finis

La discrétisation par éléments finis met en œuvre quatre phases classiques :

– interpolation,

– intégration,

– choix de la formulation du problème physique,

– choix des paramètres matériels.

L'objectif est de maintenir ces quatre notions indépendantes et de conserver une formulation générique quel que soit le problème traité.

5.1. *Interpolation et intégration*

Les fonctions d'interpolation ainsi que leur dérivées sont définies à partir d'éléments de référence choisis lors du maillage. Pour que le calcul de ces fonctions soit traité de façon générique quel que soit le type d'élément choisi, une classe abstraite ELEMENT_REFERENCE a été créée avec deux fonctions virtuelles pures *Calculer_N* et *Calculer_DN* permettant le calcul des interpolations et de leurs dérivées. De cette classe dérivent tous les éléments de référence classiques.

```
class ELEMENT_REFERENCE
{
   virtual void Calculer_N(...)=0;
   virtual void Calculer_DN(...)=0;
   virtual void Definir_Integration(...);
   int p_nombre_noeuds;
   int p_degre_interpolation;
   list<POINT_INTEGRATION_REF*> p_pt_integration;
};
```

Le type d'intégration est défini au sein de l'élément de référence à l'aide de la méthode *Definir_Integration* qui définit le nombre de points d'intégration de référence et leurs caractéristiques telles que les coordonnées de référence, le poids et les valeurs des fonctions d'interpolations calculées en ces points. Ces données sont stockées au sein d'une classe POINT_INTEGRATION_REF pour bien détacher la notion de point d'intégration. Le lien entre l'élément de référence et ses points d'intégration est symbolisé par une liste au sein de la classe ELEMENT_REFERENCE.

```
classe POINT_INTEGRATION_REF
{
   double p_poids;
   VECTEUR p_coordonnees;
   VECTEUR p_N;
   VECTEUR p_DN;
}
```

Les notions d'éléments de référence et de points d'intégration de référence permettent de ne calculer les caractéristiques de l'interpolation et de l'intégration qu'une seule fois pour un même élément de référence ce qui représente un gain au niveau du stockage et un avantage en faveur de la lisibilité du logiciel de simulation.

Le lien entre la représentation de référence et celle réelle au niveau de l'intégration est concrétisé par la classe POINT_INTEGRATION. Chaque point d'intégration dans la configuration réelle pointe sur un point d'intégration de référence.

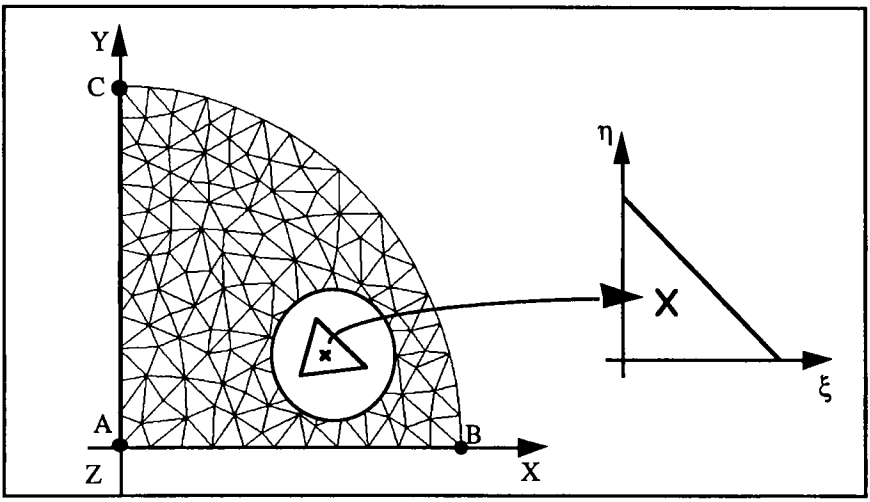


Figure 9. *Lien entre la configuration réelle et la configuration de référence*

5.2. *Formulation élément fini*

La classe abstraite FORMULATION regroupe toutes les fonctions classiques qui caractérisent le traitement global pour différentes formulations éléments finis comme les grandeurs suivantes :

- grandeurs aux points d'intégration,
- volumes élémentaires,
- gradients,
- déformations,
- rigidité, masse,
- efforts internes.

```
class FORMULATION
{
   VECTEUR p_I2;
   MATRICE p_I4;
   VECTEUR p_coefficients;
   ...
   Calculer_Parametres_Geometriques (virtuelle pure)
   Calculer_Gradient (virtuelle pure)
   Calculer_Rigidite (virtuelle)
   Calculer_Masse (virtuelle)
   Calculer_Efforts_Internes (virtuelle)
   Calculer_Deformations (virtuelle)
};
```

Toutes les fonctions virtuelles offrent un calcul générique commun aux différentes formulations dérivées de la classe de base tandis que les fonctions virtuelles pures sont nécessairement redéfinies dans les classes dérivées et donc spécifiques à la formulation choisie. Un exemple de traitement générique indépendant des formulations est donné pour le calcul de la rigidité élémentaire :

```
MATRICE& FORMULATION::Calculer_Rigidite()
{
  int npg = p_nombre_points_integration;
  for (POINT_INTEGRATION* pg = p_point_integration; npg--; pg++)
  {
    MATRICE& D = p_materiau->Donner_Operateur_Tangent(pg);
    MATRICE& B = pg->Donner_Gradient();
    double dv = pg->Donner_Volume_Elementaire();
    p_rigidite += B.Transposer() * D * B * dv;
  }
  return p_rigidite;
};
```

5.3. *Choix du matériau*

Chaque domaine peut posséder un ou plusieurs matériaux stockés dans des dictionnaires au sein de la classe DOMAINE_DISCRETISE. Une information supplémentaire est cependant utile pour avoir l'association matériau-entité géométrique.

```
class DOMAINE_DISCRETISE
{
  dictionary<string,MATERIAU*> p_materiau;
  dictionary<MATERIAU*,ZONE*>p_repartition_materiaux;
};
```

La classe MATERIAU comporte les trois opérations essentielles :

- calcul de l'opérateur tangent,
- calcul des contraintes,
- calcul de l'évolution des paramètres internes.

```
class MATERIAU
{
  Calculer_Operateur_Tangent (virtuel)
  Calculer_Contraintes (virtuel)
  Calculer_Fin_Increment (virtuel)
  Calculer_Conditions_Initiales (virtuel)
  ...
  dictionary<string,double> p_parametre;
  FORMULATION* p_formulation;
}
```

L'implémentation du calcul de l'opérateur tangent correspondant au comportement élastique linéaire donne un exemple de l'indépendance avec la formulation au niveau de l'écriture du code :

```
double G = p_parametre.access("G");
double K = p_parametre.access("K");
VECTEUR& c = p_formulation->Donner_Coefficients();
VECTEUR& I2 = p_formulation->Donner_I2();
VECTEUR& I4 = p_formulation->Donner_I4();
operateur_tangent = (I4-I2*I2*(1./3.))*(2.G)+I2*I2*K;
```

L'implémentation est proche de la formulation théorique :

$$D = 2G\left(I_4 - \frac{1}{3}(I_2 \otimes I_2)\right) + K(I_2 \otimes I_2)$$

où K et G sont respectivement les modules de compressibilité et de cisaillement élastiques.

Les tenseurs identité d'ordre 2 et d'ordre 4 ainsi que les coefficients sont stockés au sein de la formulation qui définit leurs caractéristiques.

5.4. *Choix de l'algorithme de résolution*

Les différents algorithmes de résolution sont définis au sein de classes dérivées de la classe abstraite ALGORITHME selon le schéma de la figure 10.

La classe ALGORITHME regroupe les attributs et méthodes communs aux différents algorithmes. Elle contient notamment la liste des paramètres définissant l'algorithme ainsi que les méthodes générales redéfinies au sein de chaque classe d'algorithme et qui contiennent la formulation du type de résolution associée.

```
class ALGORITHME
{
    ...
    void Definir_Parametre(...);
    dictionary<string, string> p_parametre;
    ...
    virtual void Calculer_Conditions_Initiales(...) = 0;
    virtual void Calculer_Increment(...) = 0;

};
```

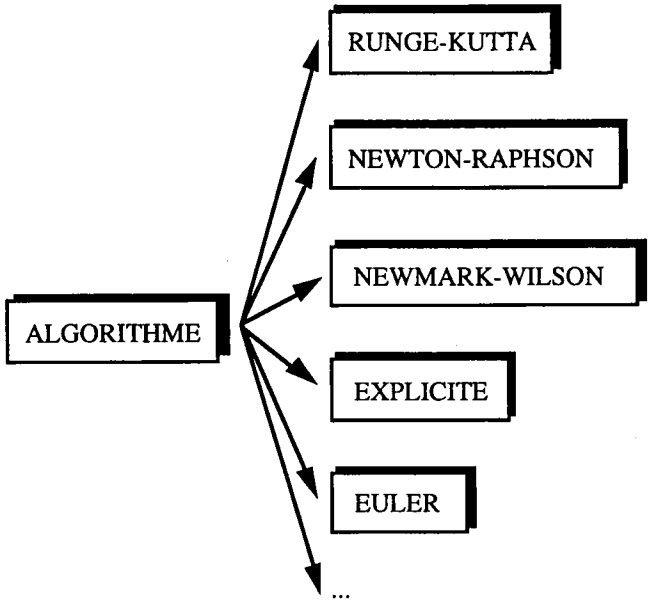


Figure 10. *Classe de base ALGORITHME*

6. Extensions et applications

L'architecture de base a été décrite pour les problèmes mono-domaine et mono-analyse. Mais c'est dans le cadre de problèmes plus complexes tels les problèmes multi-domaines et multi-analyses que l'extensibilité d'un code de calculs et l'apport de la POO peuvent être jugés.

6.1. *Problèmes multi-domaines avec contact et frottement*

Le traitement des problèmes multi-domaines avec contact et frottement nécessite la création d'une classe INTERACTION permettant de gérer les conditions aux limites liées entre les domaines.

La gestion du contact est effectuée par l'ajout de la méthode *Etablir_Zone_Contact_Avec* au niveau de la classe GEOMETRIE qui génère une instance de la classe ZONE_CONTACT qui est stockée au sein du domaine et qui contient toutes les informations relatives au contact entre deux domaines.

```
class ZONE_CONTACT
{
   list<NOEUD*> p_liste_noeuds_tangents;
   list<NOEUD*> p_liste_noeuds_penetres;
   d_array<NOEUD*, VECTEUR> p_vecteur_penetration;
   d_array<NOEUD*, double> p_amplitude_penetration;
   list<MAILLE*> p_liste_mailles_en_contact;
}
```

Cette classe contient toutes les informations nécessaires à la gestion du contact avec frottement qui est effectuée par la méthode *Introduire_Conditions_De_Contact* ajoutée à la classe DOMAINE_DISCRETISE. La détection du contact est réalisée au sein de la géométrie par la méthode *Etablir_Zone_Contact_Avec* qui permet de déceler le contact entre deux domaines. La figure 11 illustre l'application du contact avec frottement entre deux domaines déformables dans le cas de la simulation de la coupe orthogonale des métaux.

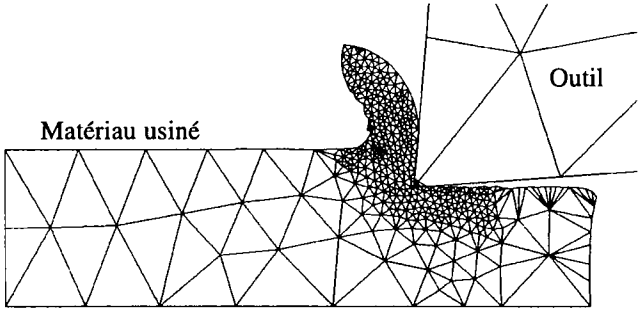


Figure 11. *Illustration d'un cas de coupe orthogonale des métaux*

6.2. *Remplissage d'un moule par un fluide visqueux incompressible mono ou polyphasique*

Ce type de simulation nécessite la prise en compte de deux nouvelles notions :

- le remplissage d'un domaine,
- le traitement de l'incompressibilité.

6.2.1. *Gestion du remplissage*

Dans ce cas précis, les calculs ne sont effectués que sur les mailles remplies par le fluide. Etant donné que chaque domaine discrétisé peut être remplissable, les modifications se feront par ajout d'attributs et de méthodes au sein de la classe DOMAINE_DISCRETISE :

– Ajout d'un drapeau pour indiquer s'il s'agit d'un problème avec remplissage ou non : *int p_remplissage.* En fonction de cet indicateur, tous les calculs globaux ne sont effectués que sur les mailles remplies.

– Une liste de mailles remplies *list<MAILLE*> p_liste_mailles_remplies.*

– Une liste des mailles formant le front du fluide *list<MAILLE*> p_mailles_front.*

– Une méthode de gestion de l'avancée du front du fluide appelée lors de la mise à jour de l'état du domaine en fin d'incrément *Mettre_A_Jour_Front,*

– Ajout d'un taux de remplissage comme attribut de chaque MAILLE.

6.2.2. *Traitement de l'incompressibilité : domaines à interpolation mixte*

La condition d'incompressibilité par des méthodes mixtes nécessite le calcul explicite de la pression en des nœuds particuliers. Pour cela, on superpose au maillage existant un autre maillage aux nœuds desquels la pression sera calculée et dont les mailles possèdent leurs propres points d'intégration. Il existe deux types de maillages superposés : continu ou discontinu.

Les modifications interviennent à différents niveaux :

– Maillage

Chaque entité géométrique (ZONE) possède en plus de son maillage classique, un maillage mixte créé par la méthode *Creer_Maillage_Mixte.*

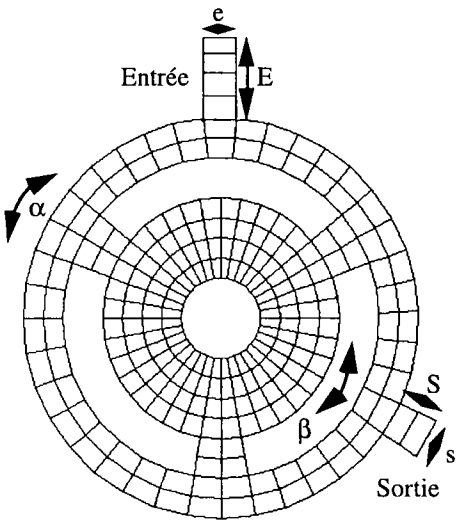
– Domaine

Un domaine supplémentaire DOMAINE_MIXTE est créé, dérivant de DOMAINE_DISCRETISE, dans lequel les grandeurs globales sont redimensionnées afin de prendre en compte les ddls en pression. Les méthodes de calcul sont également redéfinies mais font cependant appel aux méthodes de DOMAINE_DISCRETISE pour les calculs relatifs aux ddls en dehors de la pression.

– Formulation

Ajout d'une nouvelle méthode comme *Calcul_Rigidite_Mixte.*

En utilisant les concepts définis ci-dessus, le remplissage de moules par un fluide visqueux mono ou biphasique a été traité [DUT 98]. Un problème type est illustré sur la figure 12, tandis que la figure 13 présente les résultats obtenus en termes de taux de remplissage.



GEOMETRIE
- Rayons : 1, 3, 4 et 5 cm
- $\alpha = 20°$, $\beta = 45°$
- e = s = 0,86 cm
- E = 2 cm, S = 1 cm

MODELISATION
- fluide visqueux newtonien
  $\mu = 100$ Pa.s

CONDITIONS AUX LIMITES
- parois glissantes
- vitesse constante à l'entrée
  v = 1 cm/s

Figure 12. *Données initiales concernant un problème de remplissage de moule*

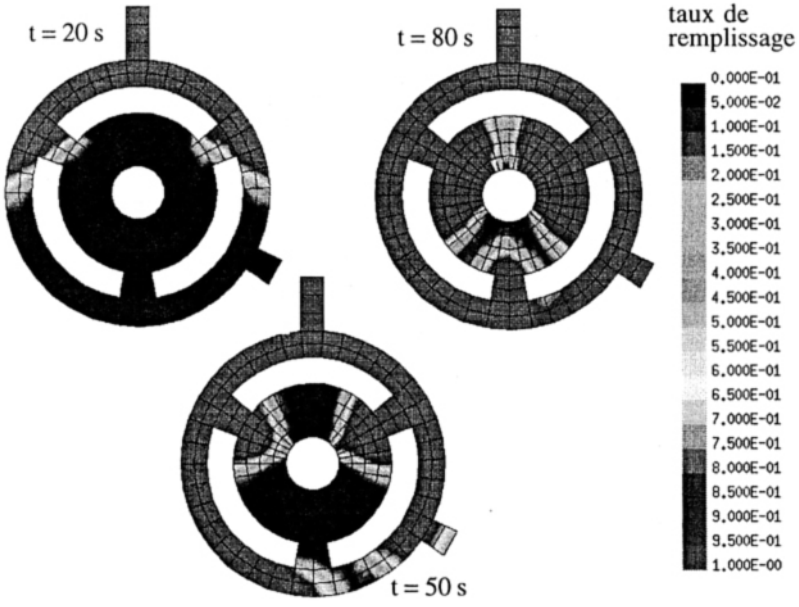


Figure 13. *Evolution du taux de remplissage au cours du temps*

6.3. *Problèmes thermomécaniques : domaines couplés*

Ce type de problèmes comprend plusieurs analyses simultanées sur un même domaine. Une solution est de créer une formulation thermomécanique complète en utilisant les domaines discrétisés décrits précédemment mais il est préférable d'utiliser les formulations thermiques et mécaniques déjà existantes. L'idée est de coupler deux domaines discrétisés, l'un pour l'analyse mécanique et l'autre pour l'analyse thermique. Le couplage est alors fondé sur un échange d'informations entre les deux domaines qui influent l'un sur l'autre. Les deux domaines sont gérés selon le schéma maître-esclave. Les deux domaines couplés sont vus de l'extérieur comme un seul domaine, en l'occurrence le domaine maître. L'autre domaine est uniquement visible par le domaine maître qui le contrôle. Chaque domaine possède son propre algorithme de résolution mais le domaine maître comporte en plus un algorithme de couplage qui pilote les algorithmes propres aux domaines maître et esclave.

Le schéma suivant (figure 14) résume le principe des domaines-couplés :

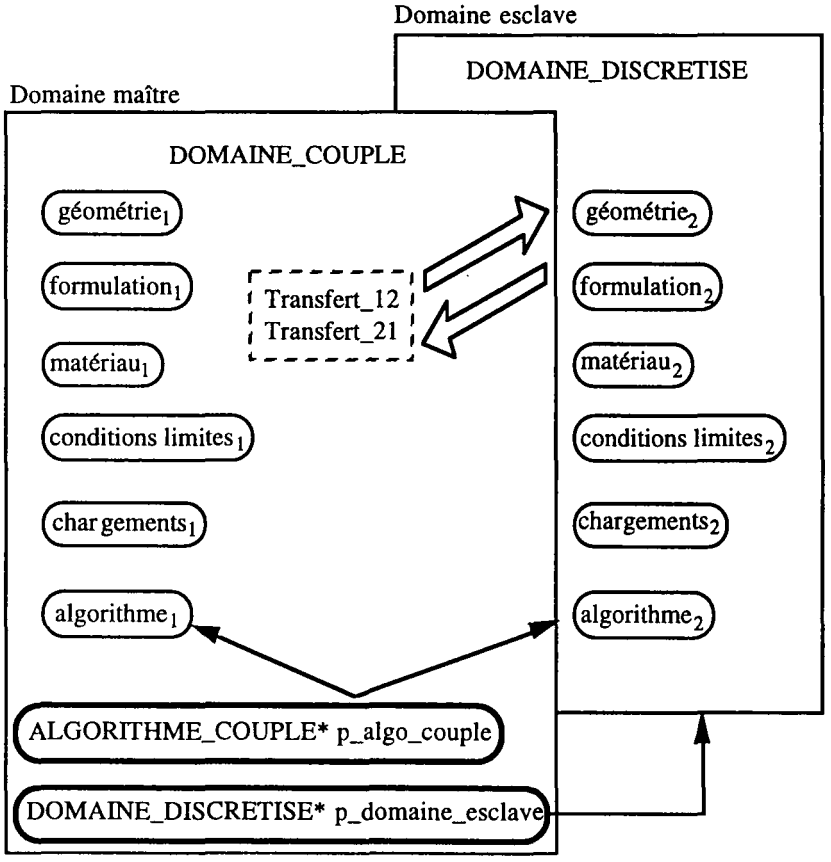


Figure 14. *Schéma utilisé pour les problèmes couplés*

L'implémentation implique la création d'une nouvelle classe DOMAINE_COUPLE qui dérive de la classe DOMAINE_DISCRETISE et qui comporte :

– un pointeur sur le domaine esclave *DOMAINE_DISCRETISE* p_domaine_esclave,*

– un algorithme de couplage : ALGORITHME_COUPLE p_algorithme_couple, la classe ALGORITHME_COUPLE étant créée par dérivation de la classe ALGORITHME,

– des méthodes de transfert entre domaine maître et esclave (transfert_12 et transfert_21,

– la redéfinition de certaines méthodes de DOMAINE_DISCRETISE pour qu'elles traitent les deux domaines.

```
void DOMAINE_COUPLE::Methode()
{
        DOMAINE_DISCRETISE::Methode();
        p_domaine_esclave->Methode();
}
```

L'algorithme couplé ne fait qu'appeler les algorithmes des différents domaines et les fonctions de transfert entre les deux.

```
void ALGORITHME_COUPLE::Calculer_Increment(...)
{
  p_domaine->Donner_Algorithme_1()-
>Calculer_Increment(...);
  p_domaine->Transfert_1_2();
  p_domaine->Donner_Algorithme_2()-
>Calculer_Increment(...);
  p_domaine->Transfert_2_1();
}
```

La figure 15 illustre les possibilités de l'architecture mise en place pour le traitement de problèmes thermomécaniques couplés dans le cas de l'extrusion d'un domaine axisymétrique à travers une filière tronconique.

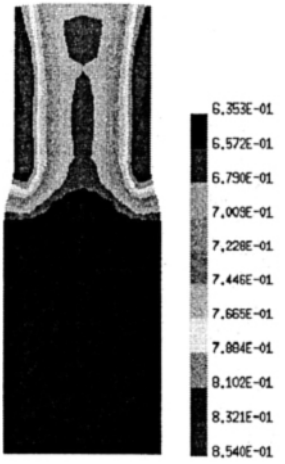


Figure 15a. *Champ de vitesse de déformation dans le cas de l'extrusion*

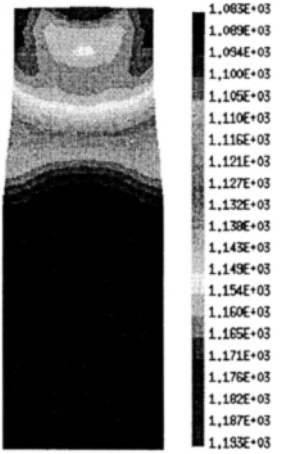


Figure 15b. *Champ de température dans le cas de l'extrusion*

7. Conclusion

Une architecture orientée objet a été implémentée pour un logiciel de simulation de procédés de mise en forme qui permet le traitement de problèmes multi-domaines et multi-analyses. Dans ce cadre, l'adéquation de la POO avec la méthode des éléments finis a été démontrée. En effet, grâce à ses deux apports essentiels, à savoir l'héritage et le polymorphisme, elle facilite l'évolutivité et le développement du logiciel de simulation. A partir d'une architecture générale et d'éléments de base, l'ajout de nouvelles fonctionnalités se fait naturellement par simple dérivation de classe et pour des modifications mineures, l'ajout d'attributs et de méthodes dans les classes existantes suffit. Ce qui a l'avantage de toujours respecter l'architecture de base sans avoir à la remettre en cause à chaque modification. L'architecture de base est cependant amenée à évoluer par ajout de nouvelles classes de base souvent abstraites qui jouent le rôle de coordinateurs. A chaque fois, l'évolution s'effectue de façon modulaire, ce qui améliore la maintenance et l'efficacité des développements. Mais, l'atout indéniable de la POO est qu'elle tend à la représentation fidèle des entités du monde physique réél par sa notion principale d'objet. Ainsi, les frontières entre le mode de fonctionnement de la pensée humaine et la conception informatique deviennent infimes.

8. Bibliographie

[AAZ 93] AAZIZOU K., BESSON J., CAILLETAUD G., HOURLIER F., « Une Approche C++ du Calcul par éléments finis », Colloque National en Calcul des Structures, Vol. 2, p. 709-722, 11-14 Mai 1993 Giens, France, Hermès.

[BAI 98] BAIDA M., Génération automatique des maillages par une méthode de type Delaunay. Application à l'adaptation de maillages en élasticité, PhD Thesis, University of Franche-Comté, Besançon, 1998.

[BES 97] BESSON J., FOERCH R., Large scale object-oriented finite element code design, *Comp. Meth. Appl. Mech. Engrg.*, Vol. 142, n° 1-2, p. 165-184, 1997.

[BOO 93] BOOCH G., *Object-oriented analysis and design with applications*, 2nd edition, Benjamin Cummings, Redwood City, 1993.

[BRE 92] BREITKOPF P., TOUZOT G., « Architecture des logiciels et langages de modélisation », *Revue Européenne des Eléments Finis*, Vol. 1, n° 3, p. 333-368, 1992.

[DUB 92] DUBOIS-PÈLERIN Y., ZIMMERMANN T., BOMME P., « Object-oriented finite element programming : II. A prototype program in Smalltalk », *Comp. Meth. Appl. Mech. Engrg.*, Vol. 98, p. 361-397, 1992.

[DUB 93] DUBOIS-PÈLERIN Y., ZIMMERMANN, « Object-oriented finite element programming III. An efficient implementation in C++ », *Comp. Meth. Appl. Mech. Engrg.*, Vol. 108, p. 165-183, 1993.

[DUT 98] DUTILLY M., Modélisation et simulation par éléments finis du moulage par injection métallique, PhD Thesis, Université de Franche-Comté, Besançon, 1998.

[EYH 95] EYHERAMENDY D., ZIMMERMANN T., « Programmation orientée objet appliquée à la méthode des éléments finis: dérivations symboliques, programmation automatique », *Revue Européenne des Eléments finis*, Vol. 4, n° 3, p. 327-360, 1995.

[EYH 96a] EYHERAMENDY D., ZIMMERMANN T., « Object-oriented finite elements: I. Principles of symbolic derivation and automatic programming », *Comput. Methods Appl. Mech. Engrg.*, 132:259-276, 1996.

[EYH 96b] EYHERAMENDY D., ZIMMERMANN T., « Object-oriented finite elements: II. A symbolic environment for automatic programming », *Comput. Meth. Appl. Mech. Engrg.*, 260:277-304, 1996.

[FEN 90] FENVES G.L, « Object-Oriented programming for engineering software development », *Engineering with computers*, 6, 1-15, 1990.

[FOR 90] FORDE B.W.R., FOSCHI R.O., STIEMER S.F., « Object-Oriented Finite Element Analysis », *Computers & Structures*, Vol. 34, n° 3, p. 355-374, 1990.

[GEL 95] GELIN J.C., WALTERTHUM L., Design of an object oriented software for the computer aided simulation of complex forming processes, *Proc. of the 5th Int. Conf. on Numerical Methods in Industrial Forming Processes*, Ed by S.F. Shen et al, A.A. Balkema, p. 729-735, 1995.

[LAI 91] LAI M., *Conception orientée objet. Pratique de la méthode HOOD.* Dunod, 1991.

[MIL 88] MILLER G.R., « A LISP-Based Object-Oriented Approach to Structural Analysis », *Eng. with Comput.*, Vol. 4, p. 197-203, 1988.

[MIL 91] MILLER G.R., « An Object Oriented Approach to Structural Analysis and Design », *Computers & Structures*, Vol. 40, p. 75-82, 1991.

[MUS 96] MUSSER D.R., SAINI A., *STL Tutorial and Reference Guide, C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.

[NÄH 95] NÄHER S., The LEDA user manual. Technical report, Max-Planck Institut für Informatik, Saarbrücken, Germany, 1995.

[RUM 91] RUMBAUGH J., Object oriented modeling and design, Prentice Hall, 1991.

[SCH 92] SCHOLTZ S.-P., « Elements of an Object-Oriented FEM++ program in C++ », *Computers & Structures,* Vol. 43, p. 517-529, 1992.

[VER 88] VERPEAUX P., CHARRAS T., MILLARD A., « CASTEM 2000: une approche moderne du calcul des structures », Conférence, Calcul des Structures et Intelligence Artificielle, Vol. 2, J.M. Fouet, P. Ladevèze, R. Oyahon eds, Pluralis, 1988.

[WAL 96] WALTERTHUM L., Programmation orientée objet et calculs par éléments finis, Application à la conception d'un logiciel de simulation en mise en forme des matériaux, PhD Thesis, Université de Franche-Comté, Besançon, 1996.

[ZIE 91] ZIENKIEWICZ O.C., TAYLOR R., *The Finite Element Method,* Fourth Edition, Vol. 1 and 2, Mc Graw-Hill, 1991.

[ZIM 92] ZIMMERMANN T., DUBOIS-PÈLERIN Y., BOMME P., « Object-oriented finite element programming: I. Governing principles », *Computer Methods in Applied Mechanics and Engineering,* 98, 291-303, 1992.