Programmation orientée objet appliquée à la méthode des éléments finis : dérivations symboliques, programmation automatique

Dominique Eyheramendy — Thomas Zimmermann

Laboratoire de mécanique des structures et milieux continus
Ecole polytechnique fédérale de Lausanne
DGC-A2
Ecublens
1015 Lausanne, Suisse

RÉSUMÉ. *Les technologies informatiques nouvelles dans le domaine du calcul numérique permettent aujourd'hui d'aborder les problèmes de mécanique de manière différente. Ainsi le développement de la programmation orientée objet a permis de restructurer les codes de calcul numérique basés sur la méthode des éléments finis [ZIM 92] avec pour effet d'en faciliter le développement et la maintenance. Dans le cadre de cet article, on propose une formulation orientée objet appliquée à la dérivation symbolique et à la programmation automatique des formes matricielles élémentaires pour la méthode éléments finis.*

ABSTRACT. *New technologies in computer science applied to numerical computations allow new approaches of mechanical problems. The development of object-oriented programming leads to better structured codes for the finite element method and facilitates development and maintainability. The purpose of this paper is to extend the object-oriented approach to the symbolic derivation and automatic of the elemental matrix forms of the finite element method.*

MOTS-CLÉS : *méthode éléments finis, programmation orientée objet, programmation automatique, calcul symbolique.*
KEY WORDS : *finite element method, object-oriented programming, automatic programming, symbolic computation.*

Revue européenne des éléments finis. Volume 4 - n° 3/1995, pages 327 à 360

1. Introduction

En mécanique, la démarche éléments finis (E.F.) par approche variationnelle conduisant de la forme forte d'un problème aux conditions de bord à l'obtention d'un code de calcul numérique peut s'avérer longue et fastidieuse. Celle-ci débute par le choix d'une formulation variationnelle, l'obtention d'une forme faible convenable, le choix de la discrétisation du milieu et des interpolations des différents champs considérés, l'obtention des formes matricielles et enfin la programmation de celles-ci dans un code existant, le cas échéant. Ainsi de nombreux développements destinés à automatiser, en partie, certaines opérations ont été réalisés. Le calcul symbolique représente un apport considérable dans cette démarche. L'utilisation de logiciels de calcul symbolique permet de générer des formes matricielles et de les implémenter dans des langages existants. Une première approche de ce type a consisté à utiliser un logiciel de calcul symbolique afin de résoudre un problème particulier en fonction de paramètres ; cette démarche est illustrée sur des exemples simples d'élasticité dans [IOA 93]. Une autre démarche consiste à calculer symboliquement les matrices de masse ou de rigidité, ou leurs coefficients et à générer le code E.F. correspondant ; [HOA 80] [NOO 79] [CEC 77] [BAR 89] [KOR 79] et [WAN 86] en sont des exemples (la plupart de ces logiciels de calcul sont d'ailleurs capables de générer le code correspondant à une expression symbolique). Ces systèmes nécessitent cependant une étude théorique préalable et semblent relativement lourds à utiliser, bien qu'ils reposent sur un concept de programmation automatique.

Le but de cette étude est donc de donner à l'ingénieur mécanicien un environnement capable, d'une part, d'effectuer les différentes dérivations symboliques du problème de façon quasi automatique, en générant en particulier les formes matricielles symboliquement, et d'autre part, de générer automatiquement le code correspondant. Les données que l'on souhaite introduire au départ sont les équations différentielles du problème initial. Le système doit alors laisser tous les degrés de liberté nécessaires à l'utilisateur pour résoudre le problème, et ce dans un environnement interactif relativement convivial. Ceci est illustré par l'exemple de l'élastodynamique linéaire sur la figure 1.

Cet article présente d'abord brièvement le travail réalisé précédemment dans l'application de la programmation orientée objet à la méthode éléments finis (implémentation directe des formes matricielles), travail qui a débouché sur la réalisation du logiciel FEM-Object (voir Figure 1, codage direct dans le code E.F.). Une approche similaire a été appliquée à l'étude théorique qui précède l'implémentation du code numérique, approche qui débouche sur le code FEM-Theory (voir Figure 1, les outils de manipulations symboliques permettent d'obtenir une forme du problème que l'on introduit automatiquement dans le code E.F. FEM-Object). Ainsi, le nouvel environnement créé, qui combine FEM-Object et FEM-Theory, permet d'avoir une démarche continue depuis les équations différentielles du problème posé, jusqu'à la génération du logiciel E.F. qui est automatique. Les différentes barrières entre formes symboliques et numériques sont effacées. L'environnement de programmation qui a permis cette intégration est

SMALLTALK. Au stade actuel du développement, le logiciel FEM-Theory permet d'appréhender toute forme variationnelle qui fait intervenir des formes linéaires et des formes bilinéaires dont les termes peuvent inclure des dérivées premières. L'extension de la même approche à d'autres formes variationnelles et à d'autre termes dérivés ne présente cependant aucun problème de fond.

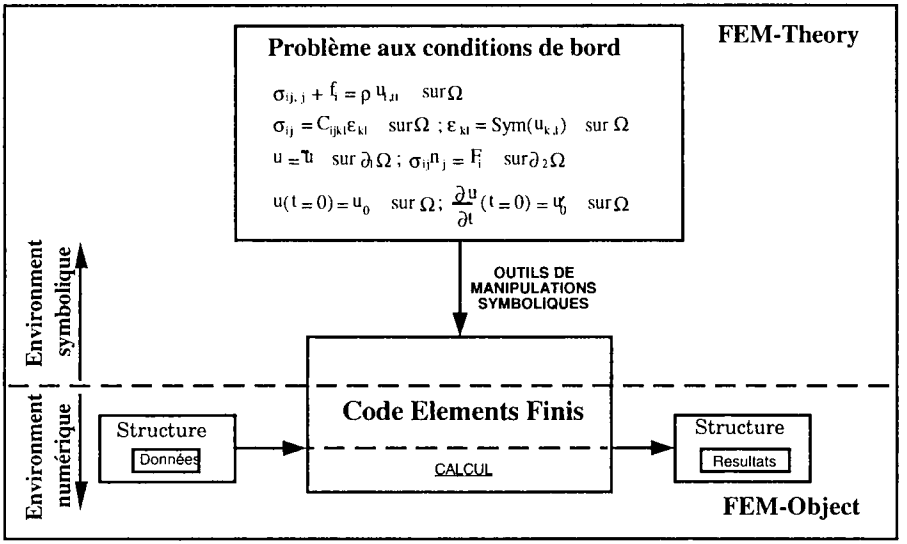

Figure 1. *Démarche de résolution d'un problème par la méthode éléments finis*

2. Un langage à objets : SMALLTALK

La programmation par objets dans les langages SMALLTALK et C++ appliquée à la M.E.F. est largement décrite dans [ZIM 92], [DUB 92] et [DUB 93]. Il est cependant intéressant de revenir sur les raisons qui ont amené au choix de SMALLTALK pour cette étude, et non de C++ par exemple, le logiciel FEM-Object existant dans les deux versions [DUB 92] et [DUB 93].

Tout d'abord il ressort nettement que SMALLTALK est un outil de prototypage des plus agréable et rapide ; C++ est par contre plus performant du point de vue calcul numérique. Cette convivialité vient du fait que plus qu'un langage de programmation, SMALLTALK est un environnement complet : environnement graphique, outils de débogage,... Ne voulant développer qu'un prototype, il était naturel de profiter de l'environnement SMALLTALK, étant donné que pour le programme projeté (FEM-Theory) la vitesse d'exécution n'est pas un facteur fondamental. Cet environnement de programmation possède de plus, un ensemble important de classes prédéfinies, source importante d'outils pour le développement.

Davantage encore, dans l'environnement SMALLTALK les modules FEM-Theory et FEM-Object sont regroupés dans un même environnement, ce qui permet d'avoir une démarche complètement continue des équations différentielles d'un problème, aux tests numériques du modèle, et ce sans compilation et lancement de modules indépendants. Il est possible, pour cette raison, d'avoir une démarche similaire à celle d'un développement sur papier pour la théorie, avec la possibilité d'effectuer des tests relativement importants, bien que limités par le manque de performance de SMALLTALK.

3. Implémentation de FEM-Object dans l'environnement SMALLTALK

3.1. Problème de référence et hiérarchie

Le problème initial sur lequel est basé la hiérarchisation, est celui de l'élastodynamique linéaire débouchant sur le problème semi-discret $Md'' + Kd = f(t)$ où M, la matrice de masse, K la matrice de rigidité et $f(t)$ le vecteur force généralisée sont formés à partir des contributions élémentaires. Le schéma d'intégration dans le temps choisi ici est un schéma implicite de différences finies, l'algorithme de Newmark. De ce problème de référence sont tirés les objets nécessaires à la résolution. Ainsi, dans la hiérarchie (Figure 2), on peut distinguer des objets découlant de l'analyse E.F.(classe **Element**,...), d'autres de l'analyse du schéma d'intégration dans le temps (classe **TimeStep**, ...) et enfin d'autres représentant des outils (classe **Tool**, ...), tel que le schéma d'intégration numérique par exemple.

3.2. Description des principales classes

La hiérarchie de FEM-Object est complètement décrite dans [DUB 92]. Cependant, une brève description des principales classes liées à l'analyse E.F. est nécessaire à la bonne compréhension de l'environnement complet, et met de plus en évidence la démarche systématique de description des codes objets.

*La classe **Element** et ses sous-classes*

L'élément est l'objet dont l'existence apparaît la plus évidente, et qui contient toutes les informations venant des équations différentielles du problème. Les sous-classes d'**Element** hériteront des attributs et méthodes communs à toutes les classes et implémenteront les comportements particuliers.

Les tâches réalisées par l'élément se décomposent en trois parties :

- identifications et manipulations des attributs suivants : noeuds, nombre de noeuds, matériau, charges réparties, points de gauss.
- calcul et assemblage des matrices de masse et de rigidité, et vecteur force généralisée

- mise à jour à chaque pas pour les calculs dynamiques ou non linéaires.

Les sous-classes d'**Element** (**PlainStrain**, **Truss2D**,...) prendront en charge les tâches suivantes :

- méthode d'instanciation (il n'y a pas d'instances de la classe **Element**, mais de ses sous-classes)
- calculs des matrices d'interpolations (nécessaires au calcul des matrices élémentaires) et calcul de la loi de comportement
- calculs liés au schéma d'intégration choisi, ici gaussien : calculs des points de gauss, de la matrice jacobienne (changement de repère local/global).

Une grande partie du comportement lié à l'assemblage est délégué au noeud et au degré de liberté.

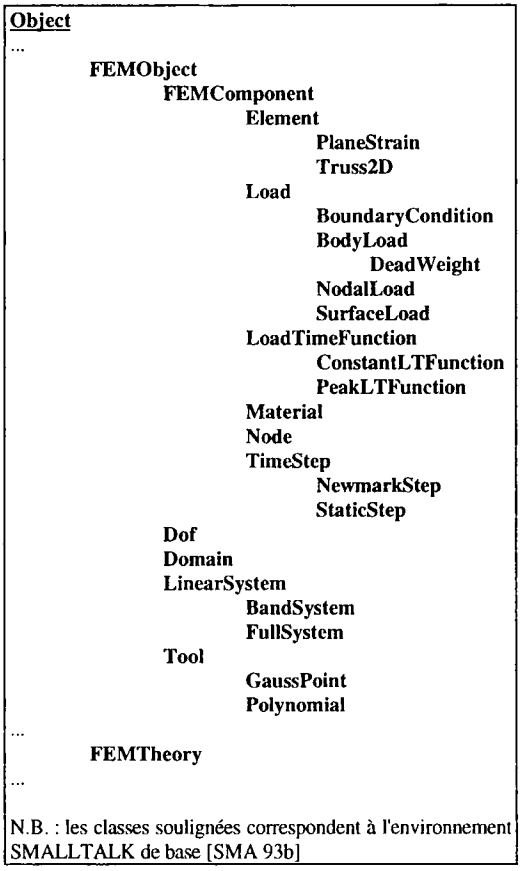

Figure 2. *Hiérarchie de FEM-Object [DUB 92]*

*La classe **Node***

Le noeud définit sa position par l'intermédiaire de l'attribut dictionnaire des coordonnées. De la même façon, il manipule ses degrés de liberté, son tableau d'assemblage dans le système linéaire global, et l'ensemble de ses charges nodales. On peut noter que l'élément s'adressera à lui pour l'assemblage dans le système global d'équations.

*La classe **Dof***

Cette classe a pour but de séparer les opérations associées à un seul degré de liberté. Ainsi, il manipulera les inconnues, la numérotation associée dans le système linéaire, et la manipulation des conditions de bord cinématiques.

*La classe **Domain***

Cette classe représente le problème numérique à résoudre. Son comportement se décompose en deux parties :

- se définir et manipuler ses composantes : ensemble d'éléments, de noeuds, des pas de temps, des pas de charge, ...
- résoudre le problème, c'est à dire former le système linéaire à chaque pas de temps et lui demander de se résoudre.

*La classe **TimeStep** et ses sous-classes*

Cette classe est au temps ce que la classe **Element** est à l'espace. Ce sont donc ses sous-classes qui implémentent la technique de résolution de l'équation différentielle ordinaire et donc la manipulation des coefficients algorithmiques.

Cette description mettant en évidence les principaux objets et leurs comportements associés permet d'identifier aisément les parties de code qu'il est nécessaire de créer pour introduire une nouvelle théorie dans FEM-Object. Si l'on désire implémenter un nouveau schéma d'intégration dans le temps, on créera une sous-classe de **TimeStep** ; pour un nouveau schéma d'intégration numérique, on créera une classe du même type que **GaussPoint** et on modifiera en conséquence les classes où l'intégration intervient ; pour une nouvelle théorie, on implémentera un nouvel élément (sous-classe de **Element**). Ces trois dernières remarques mettent en évidence les changements apportés à la structure et à l'organisation des codes E.F. par la programmation orientée objet, ce qui les rend plus facilement extensibles et réutilisables.

4. Implémentation de FEM-Theory dans l'environnement SMALLTALK

La suite de cet article s'attache à montrer comment, partant d'une théorie, on peut arriver à l'implémenter dans l'environnement décrit ci-dessus de façon quasi-automatique.

4.1. Problème de référence et hiérarchie

4.1.1 - Problème de référence

Le problème théorique choisi pour dégager la démarche et les objets nécessaires à la hiérarchisation est celui de l'élastodynamique linéaire. La formulation complète de la forme forte du problème se trouve sur la figure 3. Les notations usuelles sont utilisées (voir [HUG 87]). La démarche choisie pour appliquer la méthode E.F. est l'approche par formulation variationnelle. Elle est illustrée par la figure 3.

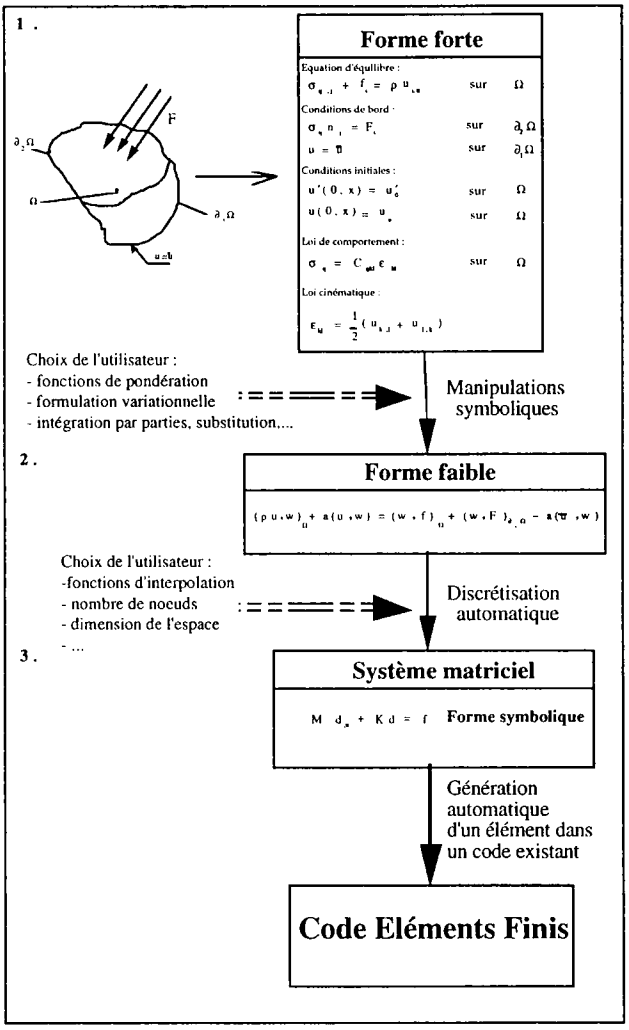

Figure 3. *Démarche générale de résolution par approche variationnelle.*

Afin de limiter l'étude, on ne prendra pas en compte le schéma d'intégration dans le temps.

Une étude détaillée de la démarche permet de mettre en évidence trois étapes illustrées en figure 3. La première est le passage du problème sous forme différentielle à une forme faible équivalente, forme nécessaire à l'application de la méthode E.F. Ceci nécessite un certain nombre d'opérations et un utilisateur compétent : choix de la formulation variationnelle, intégration par parties de certains termes, remplacement de certains termes, ... La deuxième partie correspondant à l'application de la M.E.F. peut être entièrement automatisée, à condition qu'un certain nombre d'indications soit données par l'utilisateur (dimension de l'espace, nombre de noeuds, nom des fonctions d'interpolation pour un champ considéré, ...). L'automatisation de ce processus est illustrée par un exemple.

Choisissons le terme suivant :

$$a(u,w)=\int_{\Omega} C_{ijkl}u_{(i,j)}w_{(k,l)}\,dv$$

On note $u_{(k,l)}$ la partie symétrique de l'opérateur gradient appliqué au champ vectoriel u et C est un tenseur du 4$^{\text{ième}}$ ordre.

Après discrétisation, l'opérateur $a(.,.)$devient $a(u^h,w^h)=d^t K d^*$ où u^h et w^h sont les approximations des champs u et w et où les matrices sont définies de la façon suivante :

$$K=\mathop{\mathrm{A}}_{e=1}^{n_{el}}(k^e)$$
$$d=\mathop{\mathrm{A}}_{e=1}^{n_{el}}(d^e)$$
$$d^*=\mathop{\mathrm{A}}_{e=1}^{n_{el}}(d^{*e})$$

où A exprime l'opérateur d'assemblage des contributions élémentaires.

Les matrices élémentaires s'expriment alors de la façon suivante :

$$
\begin{vmatrix}
k^e = [k^e_{PQ}] \\
k^e_{AB} = \int_{\Omega_e} B'_A D B^*_B \, dv \\
k^e_{PQ} = e'_i k^e_{AB} e_j \\
\text{avec :} \\
1 \le P, Q \le n_{ee} = (n_{ed}.n_{en}) \\
P = n_{ed}(A-1) + i \\
Q = n_{ed}(B-1) + j
\end{vmatrix}
$$

où :

n_{ee} est le nombre d'équations de l'élément

n_{ed} est le nombre de degrés de liberté par noeud

n_{en} est le nombre de noeuds de l'élément

n_{sd} est la dimension de l'espace

e_k représente le k-ième vecteur de base

et $D = \begin{bmatrix} A & F & E \\ F & B & D \\ E & D & C \end{bmatrix}$ est la loi constitutive.

On peut noter que dans la formation de k^e la matrice B_A (voir [HUG 87]) découle de la discrétisation de l'opérateur gradient appliqué à u et B_B^* de la discrétisation de l'opérateur gradient appliqué à w. Ainsi on peut reconstituer automatiquement les matrices élémentaires à partir des champs et de l'opérateur qui leur est appliqué. Il ne reste alors qu'à calculer les structures B_i pour le champ considéré. C'est cette démarche qui a été adoptée pour l'approximation éléments finis dans FEM-Theory.

Notons enfin que pour cette seconde étape le résultat de la discrétisation de la forme faible considérée est : $d''^t M d^* + d'^t K d^* = f'^t d^*$ où d^* découle de w, le champ virtuel. Afin d'obtenir la forme désirée pour l'implémentation, il faut invoquer le fait que cette équation est vraie quel que soit d^* , on peut alors écrire : $d''^t M + d'^t K = f'^t$. Cette opération constitue un nouvel outil pour la forme discrétisée. La forme ainsi obtenue est exploitable pour générer le logiciel, c'est la troisième et dernière étape de la démarche.

Les objets dégagés de cette étude sont énumérés et brièvement présentés ci-dessous.

L'objet le plus évident qui apparaît pour construire l'abstraction des équations différentielles du problème est l'expression. Celle-ci est organisée sous forme de somme de produits de termes. La classe **Expression**, abstraction de l'expression, a donc pour principal attribut une liste modélisant le comportement d'une somme, instance de la classe **SumList**. Cette liste contiendra elle même des instance de la classe **ProdList**, classe qui modélise le comportement de produit de termes ou d'expressions. On peut noter que le produit a pour attribut un signe (+ ou -). Enfin, l'entité de base est le terme. Il est décrit sur la figure 4. Ces attributs sont son nom et ses différents indices. Afin d'illustrer cette description, la figure 5 montre un exemple d'abstraction d'expression ; produits, sommes et termes y sont représentés. L'expression $\left(\sigma_{ij,j} + f_i\right)w_i$ est le produit de l'expression $\left(\sigma_{ij,j} + f_i\right)$ et du terme w_i, le tout avec un signe positif. L'expression $\left(\sigma_{ij,j} + f_i\right)$ est la somme de deux produits contenant chacun d'eux un seul terme, respectivement $\sigma_{ij,j}$ et f_i.

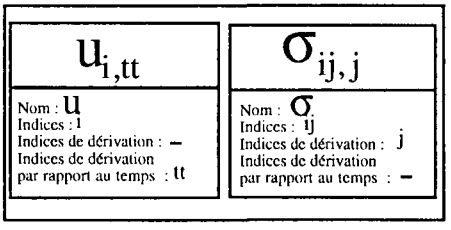

Figure 4. *Deux abstractions de termes*

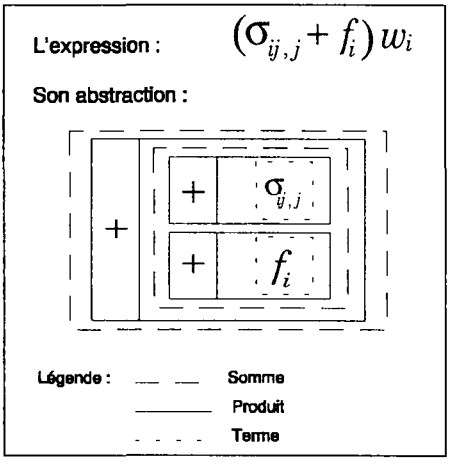

Figure 5. *Abstraction d'une expression*

A partir de ces quatre objets, on peut déjà modéliser partiellement le système différentiel de départ (Figure 3, point 1). Afin de modéliser une formulation variationnelle, d'autres objets sont nécessaires.

L'intégrale, classe **Integral**, est le premier d'entre eux. On peut décrire l'intégrale comme suit (Figure 6) :

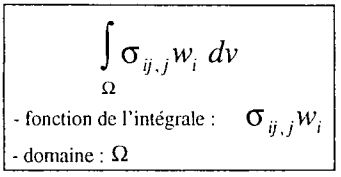

Figure 6. *Abstraction d'une intégrale*

Les attributs importants de la classe sont donc sa fonction, instance de la classe **Expression**, et le nom de son domaine d'intégration, une chaîne de caractères.
L'écriture d'une formulation variationnelle nécessite d'avoir à sa disposition une fonctionnelle (classe **Functional**), par exemple $J = \int_{\Omega} (\sigma_{ij,j} + f_i - \rho u_{i,tt}) w_i \, dv$.
Une fonctionnelle est une expression dont les entités de base sont des intégrales ; expression et fonctionnelle ont donc des comportements voisins.

Une formulation variationnelle est une équation dont les membres de droite et de gauche sont des fonctionnelles. Le nom de la classe est donc **IntEquation**.

Le système matriciel de la figure 3, finalement, est un ensemble d'équations dont les membres sont des expressions discrétisées (classe **DiscretizedExpression**), c'est à dire que les entités élémentaires de ces expressions particulières sont ici des matrices de discrétisation (classe **DiscretizationMatrix**), expressions symboliques des matrices élémentaires, et dont le comportement est proche de celui du terme.
Ainsi décrits, les objets peuvent être hiérarchisés.

4.1.2 - Hiérarchie

La hiérarchie proposée, suffisante pour résoudre le problème considéré, est montrée dans la figure 7. Les différentes classes énumérées dans le paragraphe précédent sont intégrées dans l'environnement SMALLTALK ([SMA 93a et b]).

On remarque tout d'abord que FEM-Object (code E.F.) et FEM-Theory coexistent dans la même hiérarchie. Programmer en SMALLTALK ne revient qu'à enrichir l'environnement de base. Ceci se fait de trois manières différentes :

- spécialisation de classes existantes par ajout d'une sous-classe
- ajout de nouvelles classes
- ajout de comportements particuliers à des classes existantes (ajout de méthodes)

Ainsi les classes simulant les sommes et produits ne sont que des spécialisations d'une classe existante représentant une collection ordonnée d'objets quelconques (classe **OrderedCollection**). Les classes **SumList** et **ProdList** sont ainsi regroupées sous les deux classes abstraites **FEMTheoryOrderedCollection** et **ExpressionLists**. La première regroupe toutes les classes particulières que l'on utilisera pour cet environnement, la deuxième les deux types de listes utiles à l'abstraction d'expressions et de fonctionnelles.

La majeure partie des classes sont regroupées sous la classe abstraite **FEMTheory**. On remarque que les classes **Functional** et **DiscretizedExpression** ne sont que des spécialisations de la classe **Expression**, leur structure est la même. La même remarque peut être faite pour les classes **IntEquation** et **Term** pour leurs sous-classes respectives.
Enfin, notons que quelques méthodes ajoutées à la classe **String** (abstraction d'une chaîne de caractères [SMA 93a et b]) permettent d'instancier les expressions à partir de chaînes de caractères.
Cet environnement permet de dériver symboliquement les formes successives du problème, et d'effectuer les calculs symboliques nécessaires afin d'obtenir les formes souhaitées.

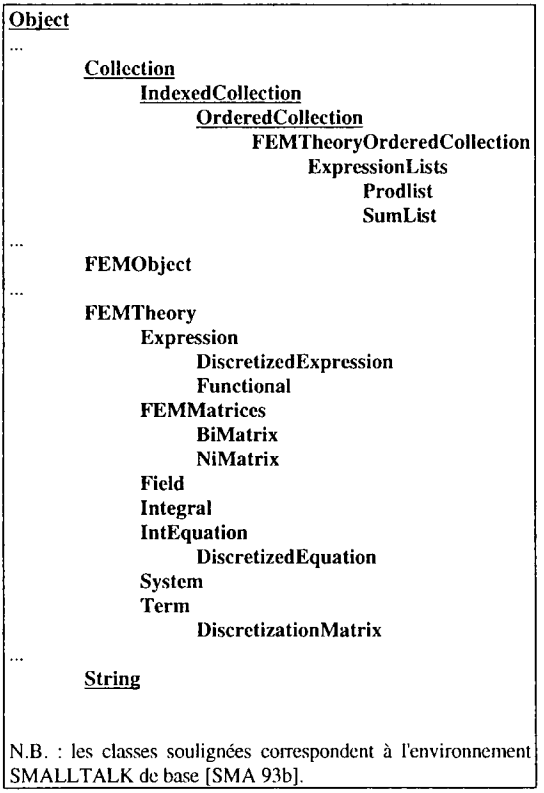

Figure 7. *Hiérarchie de FEMTheory*

4.2. Description des classes

*4.2.1. Classe **FEMTheory** et ses sous-classes*

*La classe **FEMTheory***

Cette classe regroupe les principales classes ajoutées à la hiérarchie. Elle n'est jamais instanciée. Son seul attribut est **hierarchicParent**. Toutes les sous-classes hériteront de celui-ci. Cet attribut permet de faciliter la circulation d'information entre objets, de la façon suivante :

L'objet formulation variationnelle (instance de la classe **IntEquation**) a une structure de "poupées gigognes". Son membre de gauche est une fonctionnelle, c'est à dire une "liste somme" (instance de **SumList**) de "listes produits" (instance de **ProdList**), contenant elles-mêmes des intégrales. La fonction à intégrer est une expression, une fois de plus liste de listes de termes. Si l'on considère que chacun de ces termes a besoin d'une même information, le plus simple est alors de la stocker au niveau de l'équation. La difficulté consiste ensuite pour le terme à remonter à celle-ci. Or, si par exemple l'objet "intégrale" (instance de **Integral**) connaît sa fonction à intégrer ou si les "listes sommes" connaissent leurs éléments (listes produits), la réciproque n'est pas vraie et donc les informations ne peuvent circuler que de haut en bas (encapsulation des données). Pour palier à cet inconvénient, on donne à chaque objet un attribut **hierarchicParent** qui n'est que l'objet qui est au dessus de lui dans cette structure ; une hiérarchie de structure est créée. Ainsi si un objet désire une information qu'il n'a pas, il la demande à l'objet qui le contient, son attribut **hierarchicParent**. Celui-ci fait de même s'il n'a pas l'information, le message remonte ainsi jusqu'à ce qu'un objet ait le moyen de retrouver l'information demandée. Le comportement est illustré ci-dessous.

La classe FEMTheory n'implémente que le comportement générique lié à son attribut **hierarchicParent**. Prenons par exemple la méthode qui permet de retrouver la dimension de l'espace. Elle est implémentée ici de la façon suivante :

```
giveSpaceDimension

    ^ hierarchicParent giveSpaceDimension
```

C'est une instance de la classe **IntEquation** qui est en mesure de fournir la dimension de l'espace. Elle dispose en cela de la méthode suivante :

```
giveSpaceDimension

spaceDimension isNil
        ifTrue:[ spaceDimension := self getSpaceDimension].
^spaceDimension
```

Grâce au polymorphisme, le message «self giveSpaceDimension» renverra à la méthode correcte, celle qui correspond à l'objet lui-même (*self*).
Toutes les autres méthodes de cette classe sont basées sur ce même principe.

*La classe **Expression** et ses sous-classes*

*La classe **Expression***

Le comportement de la classe comprend trois parties.
La première concerne toutes les manipulations symboliques. On notera les outils mathématiques d'addition, multiplication, ... , d'inversion, de dérivation en repère local et global,... Des méthodes de recherche de termes de types particuliers, recherche d'une divergence par exemple (méthode giveDivTerm qui renvoie le premier terme dont l'opérateur appliqué est la divergence). De plus, l'expression peut effectuer une distribution complète de tous ses termes, et effectuer un remplacement de ses termes par d'autres termes.
La seconde partie concerne la discrétisation d'une expression (par exemple du type $C_{ijkl}\varepsilon_{kl}(u)\varepsilon_{ij}(w)$).
La troisième touche à la création de code.
Il est important de noter que la quasi totalité du comportement de la classe est déléguée à l'attribut «somme», instance de la classe **SumList**.

*Les sous-classes de la classe **Expression***

*La classe **Functional***

La totalité du comportement lié à la manipulation (expansion) est héritée de la super-classe mis à part les tâches de génération de code et de discrétisation qui sont spécifiques. De plus, la fonctionnelle pourra demander à une de ses composantes (intégrale) de s'intégrer par parties.

*Classe **DiscretizedExpression***

Les remarques concernant la classe précédente restent valables.
Seuls deux outils de manipulation et un outil d'analyse sont ajoutés, à savoir une méthode qui invoque la minimisation par rapport à un champ virtuel, une méthode qui transpose ses termes, et enfin une méthode capable de rechercher la liste des instances de **DiscretizationMatrix** provenant de champs virtuels.

*Les Classes **FEMMatrices**, **BiMatrix** and **NiMatrix***

Ces classes implémentent les structures de base permettant la génération des matrices élémentaires. L'implémentation d'un nouvel opérateur mathématique dans la théorie entraîne la création d'une nouvelle sous-classe. La classe *BiMatrix* correspond à l'opérateur *B* issu de l'opérateur gradient (voir [HUG 87]).

*La classe **Integral***

Cette classe implémente sensiblement les mêmes outils qu'**Expression** en ce qui concerne les manipulations (addition, soustraction, développement,...). La

différence notable est que le développement d'une intégrale utilise la propriété de linéarité de l'intégrale.

*La classe **IntEquation** et ses sous-classes*

*La classe **IntEquation***

Les tâches principales de cette classe sont les manipulations des données liées aux inconnues du problème et aux différents termes. Cet objet englobe la totalité des autres objets composant le problème avant discrétisation ; après discrétisation, c'est une instance de **DiscretizedEquation** qui joue ce rôle.

*La classe **DiscretizedEquation***

Le comportement qui concerne les manipulations est hérité. Seules les tâches nouvelles liées à la nature des composantes élémentaires sont implémentées, c'est à dire «invocation de l'indépendance linéaire» des équations du système différentiel et transposition.

*La classe **System***

L'invocation de l'indépendance linéaire de la forme faible discrétisée par rapport à chacun des champs virtuels donne à priori un système d'équations (cas des formulations mixtes). Cet objet n'intervenant que dans la troisième étape de la démarche, son comportement n'est lié qu'à la génération de code et à la manipulation de données (dimension de l'espace, nombre de noeuds géométriques,...). On peut noter que la fonction de génération de la matrice jacobienne (changement de repère local/global dans la formulation E.F.) lui incombe. Ceci s'explique par le fait que cet objet est la représentation complète du problème, et qu'en particulier il contient les notions relatives à l'élément (éléments finis), celui-ci étant défini par rapport à un repère de référence local.

*La classe **Term** et ses sous-classes*

*La classe **Term***

La particularité de cet objet réside dans le fait qu'il est capable de s'analyser. Il peut rechercher sa nature, c'est à dire savoir s'il est un champ vectoriel ou scalaire, déterminer l'opérateur qui lui est appliqué, donner sa nature (virtuel ou solution). De plus, il possède tout le comportement permettant de se discrétiser, suivant sa nature.

*La classe **DiscretizationMatrix***

Son comportement est lié au fait que cette classe représente l'abstraction des matrices élémentaires. On peut ainsi la transposer, effectuer des substitutions dans sa matrice et l'objet sait s'analyser : matrice définie sur un domaine, sur son bord, inconnue, loi de comportement, ...

*4.2.2. Classe **FEMTheoryOrderedCollection** et ses sous-classes*

*La classe **ExpressionLists** et ses sous-classes*

*La classe **ExpressionLists***

Cette classe abstraite ne fait que spécialiser le comportement de la classe existant à la base, c'est à dire **OrderedCollection**. Cette classe a un seul attribut **hierarchicParent** (voir classe **FEMTheory**). Ainsi, aux méthodes de base sont ajoutées des méthodes plus appropriées de manipulation des objets contenus. Par exemple, la méthode d'ajout d'un objet à la liste, modifie l'attribut **hierarchicParent** de ce dernier, qui devient la liste elle-même.

*La classe **SumList***

Cette classe est impliquée dans la construction des objets expression (**Expression**), fonctionnelle (**Functional**) et expression discrétisée (**DiscretizedExpression**). Son comportement est très lié à celui de ces classes (l'attribut «somme» de ces objets est une instance de **SumList**). Son comportement se décompose en trois parties. La première concerne les manipulations de liste. On trouve ici des méthodes liées à l'expansion d'expressions, additions, mais également des méthodes liées à la dérivation d'expressions. Pour une somme, l'opération de dérivation n'est qu'une simple manipulation : la somme demande à ses produits de se dériver (la dérivée de la somme est égale à la somme des dérivées). La deuxième partie est l'analyse de ses composantes comme par exemple le recherche d'un terme sur lequel est appliqué l'opérateur divergence. Les tâches sont déléguées aux produits. La troisième est la discrétisation de celle-ci, également déléguée aux produits.

*La classe **ProdList***

Les tâches associées à cet objet peuvent être classées en trois groupes, de la même façon que pour la classe **SumList**. Les manipulations principales effectuées sur la liste produit sont l'ajout et l'enlèvement d'éléments d'une autre liste, et la dérivation qui est implémentée de façon naturelle $(fg)' = f'g + fg'$. L'analyse de ses composantes constitue la partie la plus importante de son comportement. Le produit peut retrouver toutes sortes de termes contenus, par exemple un champ virtuel ou un champ solution. Enfin, pour le processus de discrétisation, le produit demande leur discrétisation à chacun de ses termes, et réarrange leurs différentes contributions pour construire la matrice élémentaire finale.

*4.2.3. Méthodes ajoutées à une classe : classe **String***

La saisie des différentes équations se fait via le clavier et donc des chaînes de caractères. Ainsi la classe **String** aura deux méthodes supplémentaires, la première instanciant les entités élémentaires que sont les termes et la deuxième instanciant les expressions.

4.3. Généricité de l'environnement

Du point de vue des manipulations symboliques, l'environnement créé permet l'introduction de n'importe quel type d'équations différentielles dans les formulations choisies ; en effet, la convention de notation indicielle le permet. Le seul outil spécifique développé ici, est l'intégration par parties avec application du théorème de la divergence. On peut envisager aisément l'introduction de nouveaux outils à ce niveau, ou la modification d'outils existants, telle que la séparation des tâches d'intégration par parties et d'application du théorème de la divergence, et ce sans avoir à reprendre le code de FEM-Theory. Il suffit de venir ajouter les méthodes réalisant ces manipulations. L'objectif d'enrichissement de l'environnement avec de nouveaux outils de manipulation est atteint.

D'un point de vue E.F., les opérateurs qui peuvent être traités sont restreints à ceux correspondants aux structures *B* et *N* élémentaires (voir [HUG 87]). La prise en compte de nouveaux opérateurs nécessite l'enrichissement de la classe **FEMMatrices**. Le traitement d'un même opérateur, via des structures matricielles différentes, nécessite, en plus, une possibilité de choix laissée à l'utilisateur, entre plusieurs structures. L'objectif d'enrichissement de l'environnement avec des structures de traitement des opérateurs différentiels nouveaux est également atteint.

La description de la hiérarchie permettant d'obtenir le système matriciel sous forme symbolique étant terminée, il reste à établir le lien entre les deux "modules", FEM-Theory et FEM-Object de l'environnement. Ceci est fait par l'intermédiaire de la génération automatique d'un nouvel élément dans FEM-Object.

5. Génération automatique du code Eléments Finis en SMALLTALK : lien entre les environnements symbolique et numérique

La dernière étape dans le symbolisme est la génération du code. Les formules ainsi programmées seront évaluées numériquement. Le développement des paragraphes précédents montre qu'il suffit, à partir du système matriciel obtenu, de générer une sous-classe de la classe **Element** dont toutes les tâches sont dévolues au calcul des matrices élémentaires.

Toutes les informations nécessaires à la génération de la classe sont encapsulées dans les différents objets composant l'instance de **System**. Ainsi chacun d'eux va apporter les informations qu'il possède quand celles-ci lui seront demandées. Implémenter en SMALLTALK revient en fait à enrichir l'environnement soit en ajoutant des classes soit en ajoutant des méthodes. Les classes clés de la programmation automatique sont les classes **DiscretizationMatrix**, **Matrix**, **Integral** et **Expression**. En effet, ce sont elles qui vont créer le code proprement

dit, les autres classes ne faisant que contrôler le processus en créant par exemple les sélecteurs de méthode ('computeStiffnessMatrix', 'computeMassMatrix'...).

La classe ***System***
L'ordre de créer un nouvel élément est envoyé au système. Celui-ci crée alors une sous-classe à la classe **Element** de FEM-Object. Il donne à cette classe la méthode d'instanciation de l'élément. Le message permettant de créer les méthodes de calcul de la matrice jacobienne est également envoyé, car le système possède cette information (information globale concernant le changement de repère local/global). Le reste du comportement est délégué aux instances de **DiscretizedEquation**.

La classe ***DiscretizedEquation***
Les tâches sont déléguées à l'instance de **DiscretizedExpression** formant le membre de gauche (tous les termes étant préalablement ramenés dans ce membre).

La classe ***DiscretizedExpression***
Les tâches sont déléguées à l'attribut «somme» instance de **SumList**

La classe ***SumList***
Les tâches sont déléguées à chaque instance de **ProdList** contenue dans **SumList**.

La classe ***ProdList***
La liste va déterminer la nature de la matrice élémentaire en vue de création de la méthode permettant de l'évaluer numériquement dans FEM-Object. Ainsi, A dans le produit $Ad_{,ll}$ sera reconnue matrice de masse et il lui sera demandé de créer une méthode de sélecteur de méthode 'computeMassMatrix', alors que B dans le produit Bd sera reconnue matrice de rigidité (sélecteur de méthode 'computeStiffnessMatrix'). Si le produit ne contient qu'un seul terme, c'est que la matrice correspond à une contribution de force généralisée (de volume ou de surface) ; les tâches de création de code lui sont donc déléguées pour la création d'une méthode de calcul de charges.

La classe ***DiscretizationMatrix***
Son comportement a deux aspects.

- dans la méthode *createMethod: aSelector inElement: anElement* les tâches sont décentralisées vers l'attribut **elementaryMatrix**, une matrice instance de **Matrix**.

- dans la méthode *createLoadMethodsInElement: anElement* l'objet détermine s'il est défini sur le domaine ou sur son bord et envoie un message dont le sélecteur de méthode est celui présenté ci-dessus.

Le reste du comportement est décentralisé vers l'instance de **Matrix** de l'attribut «matrice élémentaire».

La suite de la description est consacrée aux classes qui vont créer le code de calcul proprement dit.

La classe ***Matrix***

L'objet matrice a deux types de méthodes à créer :

- le premier est la méthode qui renverra la structure matricielle et qui calculera chacun des coefficients de la matrice.
- le deuxième est l'ensemble des méthodes permettant de calculer chacun des coefficients ; ceci est laissé à la charge de chacun des coefficients de la forme symbolique (non anticipation sur leur nature)

Décrivons le processus à partir d'un exemple simple. Prenons la matrice 2:2 suivante :

$$\text{stiffnessMatrix} = \begin{bmatrix} obj1 & obj2 \\ obj3 & obj4 \end{bmatrix}$$

où *obj1*, *obj2*, *obj3* et *obj4* sont des objets pouvant être des instances de **Integral**, **Matrix** ou **Expression** (non-anticipation sur le contenu de la matrice).

Le code généré par l'objet matrice pour le calcul de la matrice stiffnessMatrix sera (le sélecteur de méthode choisi est 'computeStiffnessMatrix'):

```
                                                    Commentaires
computeStiffnessMatrix
  | answer |

answer := Matrix new: (2@2).                        Création d'une matrice 2:2

  answer at:(1@1) put: (self computeStiffnessMatrix11).   Met la valeur du résultat du calcul
                                                          'self computeMatrixX11' en position 1:1
  answer at:(1@2) put: (self computeStiffnessMatrix12).
  answer at:(2@1) put: (self computeStiffnessMatrix21).
  answer at:(2@2) put: (self computeStiffnessMatrix22).
^answer                                             Renvoie la réponse
```

La matrice demande ensuite à ses composantes *obj1*, *obj2*, *obj3* et *obj4* de créer les méthodes leur permettant de se calculer, en leur donnant le sélecteur de méthode correct. Pour l'objet *obj1*, ce sélecteur est : 'computeStiffnessMatrix11', '11'

indique la composante concernée et est rajouté au sélecteur de la matrice elle-même. Il en est ainsi pour chacune des composantes.

*La Classe **Integral***

La liste des tâches liées à l'intégrale pour la création de code est la suivante :

- création d'une méthode qui renvoie le tableau des points de gauss (le choix du type d'intégration se fait ici, on pourrait par exemple réaliser une intégration symbolique exacte)
- création de la méthode qui renvoie le calcul numérique de l'intégrale
- envoi du message permettant à la fonction à intégrer de créer les méthodes pour se calculer (cas de l'intégration numérique)

Si l'on considère l'exemple suivant :

$$obj1 = \int_{D^e} objx \, dv$$

Le code créé pour calculer numériquement l'intégrale par un schéma numérique de type Gauss est (le sélecteur de méthode 'computeStiffnessMatrix11' pour calculer *obj1* provient de l'exemple précèdent) :

```
                                                        Commentaires
computeStiffnessMatrix11
| answer gaussPointsArray gp coordinates weight function determinant |

answer := 0.                                            Initialisation de la valeur à 0
gaussPointsArray := self giveGaussPointsArray.          Instanciation des points de Gauss

                                                        Schéma d'intégration

gaussPointArray do:[:gp|                                Boucle sur les points de Gauss
                                                        gp est le point de Gauss courant
coordinates := gp giveCoordinates.                      Récupération des coordonnées de gp

function := self computeStiffnessMatrixFunction11At: coordinates.   Calcul de la fonction à intégrer de
                                                        obj1 qui est f (f a créé sa méthode de
                                                        calcul)

determinant := (self giveJacobianMatrixAt: coordinates) determinate.  Calcul du déterminant de la
                                                        jacobienne au point courant

weight := gp giveWeight.                                Récupération du poids de gp

answer := answer + (function * determinate * weight).
                ].

^answer                                                 Renvoie la réponse
```

L'intégrale demande ensuite à sa fonction, ici *f*, de créer sa méthode pour se calculer en un point quelconque passé en argument. Le sélecteur de méthode correct lui est donné. Pour l'objet *f*, ce sélecteur est : 'computeStiffnessMatrix11FunctionAt:', 'FunctionAt:' indique que *f* est la fonction d'une intégrale.
Remarque : *f* peut être une instance des classes **Integral**, **Matrix** ou **Expression**.

*La classe **Expression***
Le principe de création de méthode dans l'élément considéré est le suivant. Il s'agit de créer la chaîne de caractères correspondant au code source en déclarant les variables, ici locales au niveau de la méthode, et en les instanciant (module de Young par exemple), la forme symbolique de l'expression étant obtenue par le message : *self printString* ; c'est une chaîne de caractère. La chaîne ainsi obtenue peut être compilée.
Dans la continuité de l'exemple précédent, on suppose *f* instance de **Expression**, et de la forme :

$$f = ((0.5x+0.5)x1+(0.5x-0.5)x2)ES((0.5x+0.5)x1+(0.5x-0.5)x2)$$

La méthode correspondante est alors :

```
                                              Commentaires
computeStiffnessMatrix11FunctionAt: point

| answer material x x1 e s |                  Déclaration des variables

material := self giveMaterial.                L'élément donne son matériau
e := material give: 'youngModulus'.           Le matériau donne ses caractéristiques
s := material give: 'area'.

x := point giveCoordinate: 1.                 Instanciation de la coordonnée courante

x1 := (self giveNode: 1) giveCoordinate: 1.   Instanciation de la coordonnée courante
x2 := (self giveNode: 2) giveCoordinate: 2.   pour les noeuds

answer := ((0.5*x+0.5)*x1+(0.5x-0.5)*x2)*e*s
   *((0.5*x+0.5)*x1+(0.5*x-0.5)*x2).          Calcul de l'expression (ses variables sont instanciées)

^ answer                                      Renvoie la réponse
```

Il est à noter que l'utilisateur doit fournir la liste des constantes qu'il a utilisées ainsi que la correspondance avec celles que l'on trouvera dans le fichier de données du problème numérique. Par convention, les variables x, y, z correspondent aux

coordonnées du point courant, et les variables x1, x2, y1, ..., yN, aux coordonnées des noeuds de l'élément.

Cette part de comportement dépend bien entendu du langage d'implémentation, mais seulement d'un point de vue syntaxique. La localisation des tâches est tout à fait générale. Le même schéma peut s'appliquer à n'importe quel couple langage informatique / code E.F.

6. Applications

Dans cette partie, sont décrits deux exemples d'écoles de dérivation, depuis la formulation du problème variationnel, jusqu'au test numérique des éléments créés. On n'ira pas au delà de cet aspect descriptif dans le cadre de cet article. Ces exemples ont été choisis de façon à avoir une formulation très classique (équation de la chaleur) et une formulation mixte (problème de stokes). Les aspects théoriques des deux formulations développées dans cette partie sont décrits en détails dans [HUG 87].

6.1. Equation de la chaleur

6.1.1. Forme forte du problème

La forme forte du problème aux conditions de bord et conditions initiales peut être donnée comme suit [HUG 87] :

Trouver T : $\overline{\Omega} \times [0, \mathrm{T}]$ telle que :

$$\rho C T_{,t} + q_{i,i} = f \qquad \text{sur } \Omega \times]0, \mathrm{T}[$$

$$T = g \qquad \text{sur } \Gamma_g \times]0, \mathrm{T}[$$

$$-q_i n_i = h \qquad \text{sur } \Gamma_h \times]0, \mathrm{T}[$$

$$q_i = -K_{ij} T_{,j} \qquad \text{sur } \Omega \times]0, \mathrm{T}[$$

$$T(x,0) = T_0(x) \qquad x \in \Omega$$

où :
Ω est le domaine
Γ_h est le bord du domaine sur lequel sont appliquées les conditions naturelles
Γ_g est le bord du domaine sur lequel sont appliquées les conditions essentielles
T est le champ de température solution
f, g, et h dépendent des coordonnés spatiales et temporelles
n est la normale extérieure au domaine

6.1.2. Formulation variationnelle

La formulation variationnelle du problème proposée est la suivante :

Soit $\mathcal{V}$ l'ensemble des champs de température virtuels :

$$\mathcal{V} = \left\{ T(x,t)\ x \in \Omega \text{ et } t \in [0,\mathrm{T}] \,\middle|\, T(x,t) = 0 \text{ pour } x \in \Gamma_g \text{ et } T \text{ régulière} \right\}$$

Soit $\mathcal{S}$ l'ensemble des champs de température solutions :

$$\mathcal{S} = \left\{ T(x,t)\ x \in \Omega \text{ et } t \in [0,\mathrm{T}] \,\middle|\, T(x,t) = g(x,t) \text{ pour } x \in \Gamma_g \text{ et } T \text{ régulière} \right\}$$

Trouver $T \in \mathcal{S}$ telle que $\forall\, w \in \mathcal{V}$ on ait :

$$\int_\Omega \rho C T_{,t} w dv + \int_\Omega q_{i,i} w dv = \int_\Omega f w dv$$

avec $$\int_\Omega \rho C T(0) w dv = \int_\Omega \rho C T_0 w dv$$

6.1.3 Dérivation dans FEMTheory

Notation

Après dérivation complète du problème la fenêtre de FEMTheory (environnement fenêtré Windows) se présente comme le montre la figure 8. Il est tout d'abord nécessaire de décrire les notations employées (Tableau 1). Le principe de représentation est que tout objet est capable de donner sa représentation sous forme de chaîne de caractères. Ainsi une expression recompose sa représentation en demandant à chacune de ses composantes de lui fournir sa contribution.

La convention adoptée ici est qu'un terme est représenté par une lettre capitale, son nom, les indices étant des lettres minuscules et les indices de dérivation étant précédés d'une virgule. Cela permet de conserver des notations proches de celles utilisées couramment.

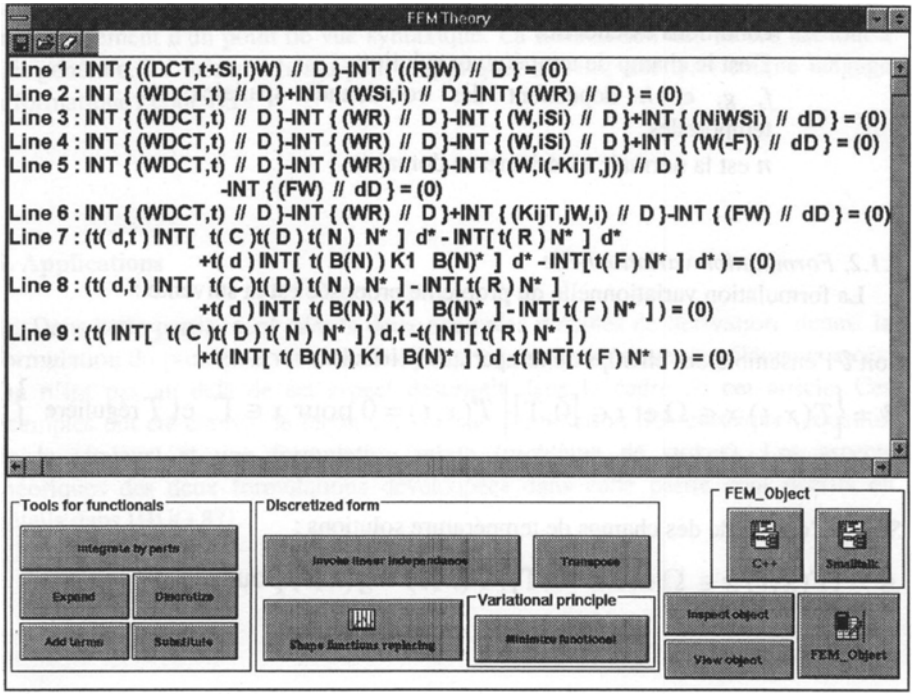

Figure 8. *Dérivation du problème de thermique dans FEMTheory*

	Notations mathématiques usuelles	Notation dans FEMTheory
Terme	$\frac{\partial^2 U_i}{\partial t^2}$	Ui,tt
Expression	$\rho C T_{,t} + q_{i,i}$	(RCT,t+Qi,i)
Intégrale	$\int_D q_{i,i} w \, dv$	INT{(Qi,iW) // D}
Fonctionnelle	$\int_D q_{i,i} w \, dv + \int_D f w \, dv$	INT{(Qi,iW) // D}+INT{(FW) // D}
Equation	$\int_D q_{i,i} w \, dv + \int_D f w \, dv = 0$	INT{(Qi,iW) // D}+INT{(FW) // D}=(0)

Tableau 1. *Notations employées dans FEMTheory*

Description

Décrivons le problème à partir de l'écran de FEMTheory (Figure 8). Les outils disponibles sont regroupés dans le bas de la fenêtre. Chacun d'eux envoie un message soit à l'objet courant, c'est à dire à celui représenté sur la dernière ligne écrite à l'écran, soit à l'objet sélectionné à l'écran. Celui-ci réagit en fonction de la méthode activée. Ces outils sont présentés et décrits dans [EYH 94], nous ne nous intéresserons ici qu'aux dérivations apparaissant sur la fenêtre principale.

- Ligne 1: La saisie des différentes composantes permettant d'instancier l'objet représenté sur cette ligne, c'est à dire une instance de **IntEquation**, se fait par l'intermédiaire de fenêtres de saisie. Les expressions simples et équations sont instanciées à partir de chaînes de caractères, par exemple la chaîne «DCT,t+Si,i-R=0» pour former le principe variationnel. Ainsi on appellera «D» la densité volumique du milieu (à ne pas confondre avec le domaine ayant le même nom) et «C» la capacité thermique du milieu.

- Ligne 2: Afin de pouvoir traiter convenablement chacun des membres du principe variationnel dans le but d'obtenir une forme faible, il est nécessaire de développer la ligne 1. Le bouton 'Expand' permet de réaliser cette opération. Les fonctions à intégrer de chaque intégrale sont développées (instances d'Expression) et la propriété de linéarité de l'intégrale est appliquée. Les deux instances d'intégrales sont remplacées par trois nouvelles instances dans le membre de gauche de l'équation (lhs).

- Ligne 3: L'instance représentée par «INT{(WSi,i) // D}» est sélectionnée à l'écran sur la ligne 2 pour être intégrée par parties (en fait pour intégration par partie et application du théorème de la divergence), en activant le bouton 'Integrate by parts' ; le message correspondant est envoyé à l'instance sélectionnée. Elle est remplacée dans l'équation par deux instances de **Integral** dont les fonctions à intégrer sont instanciées à partir du sien. L'instanciation des nouveaux termes se fait à partir des noms et indices des termes initiaux. On peut noter ici le terme «Ni» qui représente par convention la normale extérieure à la surface au domaine et le caractère «d» placé devant le nom du domaine qui représente le bord de celui ci.

- Ligne 4: L'instance «INT{(NiWSi)//dD} est sélectionnée à l'écran afin d'utiliser la condition de bord naturelle. Par l'intermédiaire d'une fenêtre, l'utilisateur donne d'une part l'expression qu'il veut remplacer et d'autre part celle par laquelle il veut la remplacer. La fonction à intégrer, instance d'**Expression**, ôte de ses composantes les instances de termes concernées, ici «Ni» et «Si», et les remplace par la nouvelle instance d'**Expression** contenant une seule instance de **Term** : «(F)». L'outil permettant cette opération est 'Substitute'.

- Ligne 5: La même opération qu'en ligne 4 est réalisée pour utiliser la loi constitutive.

- Ligne 6: La même opération qu'en ligne 2 est réalisée, c'est à dire que les deux membres de l'équation sont développés.

- Ligne 7: L'objet de la ligne 6 représente une forme faible du problème. Dans le passage des lignes 6 à 7 sont condensées les opérations suivantes : passage de la notation tensorielle à la notation vectorielle, approximation du problème par la méthode de Galerkin, discrétisation du domaine en éléments et interpolation des différents champs sur l'élément. Ce schéma est pour l'instant le seul implémenté, et le bouton 'Discretize' permet d'envoyer le message correspondant à ces opérations à l'objet équation. Le processus de discrétisation a été décrit précédemment. L'utilisateur aura à fournir un certain nombre de renseignements tels que la dimension de l'espace, le choix de l'interpolation ($u = \sum_{1}^{4} N_i d_i$ par exemple, le choix des fonctions d'interpolation pouvant être réalisé plus tard), choix des coefficients des matrices constitutives,... Le résultat est une instance de **DiscretizedEquation** dont la représentation est sur cette ligne. On peut noter que la représentation d'une matrice élémentaire, donnée par son attribut **name**, indique la façon dont elle a été obtenue ; par exemple «INT[t(B(N)) K1 B(N)*]» représente une matrice de type rigidité, elle est obtenue par le produit $B^t K B$ intégré sur le domaine élémentaire (notation définie dans [HUG87]). Dans ce cas, l'opérateur B est appliqué aux fonctions d'interpolations N_i .

- Ligne 8: L'équation de la ligne 7 étant vérifiée pour tout d^* , son coefficient est identiquement nul. Cette opération est réalisée sur l'équation discrétisée avec l'outil 'Invoke linear independance' et elle est réalisée en utilisant l'outil de dérivation, dérivation par rapport aux termes découlant de la discrétisation de champs virtuels. Le résultat de cette opération est un ensemble d'équations instances de **System** (voir la formulation mixte du deuxième exemple), chaque instance d'équation étant ajoutée à celui-ci.

- Ligne 9 :

-a: La première chose que l'on puisse faire est de transposer chaque membre de l'équation. Le bouton 'Transpose' permet de réaliser cette opération. Chacun des termes de l'équation discrétisée, qui sont des instances de **DiscretizationTerm**, est transposé, c'est à dire que sa matrice élémentaire est transposée et qu'il est affecté d'un indice de transposition indiquant dans la représentation que le terme a été transposé.

-b: Cet objet étant la représentation du système à partir duquel on pourra générer le code E.F. il faut remplacer les différentes fonctions d'interpolation par leurs valeurs dans les matrices élémentaires. Une fenêtre permet de sélectionner différents types de fonctions et de les attribuer aux fonctions, ce sont des fonctions bilinéaires sur un élément à 4 cotés. Cette opération consiste à créer un dictionnaire de fonctions dans lequel sont mises les fonctions et leurs dérivées successives (qui sont évaluées symboliquement), et à remplacer les différentes fonctions par leurs valeurs dans l'attribut **elementalMatrix** des instances de termes discrétisés.

-c: La dernière étape est la génération automatique de code. La seule option possible pour l'instant est la génération d'un élément dans FEM-Object

version SMALLTALK. Pour cela il faut cliquer le bouton 'SMALLTALK' dans la fenêtre FEMObject. Le message 'createNewElementInFEMObject' est envoyé à l'objet System. La procédure suit ensuite la description donnée dans le chapitre précédent.

Le but de la démarche n'est pas aujourd'hui de générer du code pour effectuer des calculs importants (en d'autres termes, le code généré n'est pas optimisé). Il permet cependant de réaliser des calculs suffisamment grands pour tester l'approche utilisée. Ceci est montré à l'aide du test numérique suivant.

6.1.4. Test de l'élément

Formulation du problème numérique

Le problème test que l'on se propose de résoudre dans cet exemple est présenté en figure 9.

Le maillage du domaine est présenté figure 10 , il contient 27 éléments.

Les coefficients sont pris tels que (voir notation dans la dérivation dans FEMTheory):

D=1 U.S.I.(densité), C=10 U.S.I. (capacité calorifique) et K=10 U.S.I.(conductivité thermique)

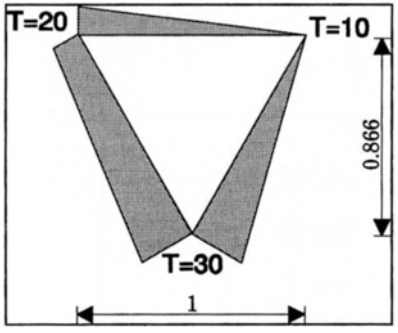

Figure 9. *Description du problème*

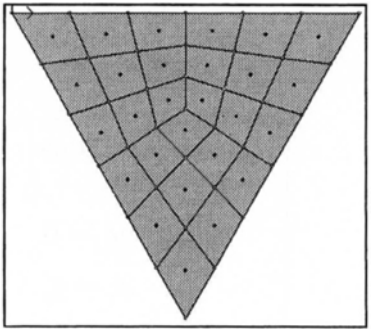

Figure 10. *Maillage du domaine*

Résultats

L'intégration dans le temps se fait par la méthode de Newmark (voir [HUG 87] pour la théorie et [DUB 92] pour l'implémentation dans FEMObject) avec α=0.5. Le pas de temps est pris égal à 0.1. Les résultats sont présentés en figure 11.

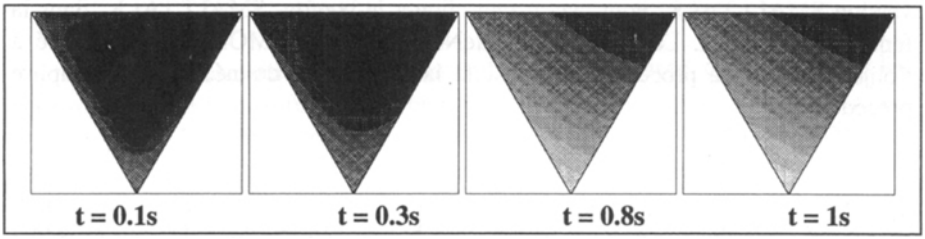

Figure 11. *Résultats numériques*

On observe que le champs de température converge vers la solution du régime établi qui est un gradient uniforme de températures. Ce calcul permet ainsi de valider la formulation choisie.

6.2. Problème de Stokes

Le but de ce paragraphe est de montrer que l'approche proposée est applicable à des formulations mixtes. La présentation est la même que pour le paragraphe précédent. Seules seront commentées les spécificités liées à cette formulation.

6.2.1. Forme forte du problème

La forme forte du problème aux conditions de bord peut être donnée ainsi [HUG 87] :

Trouver u: $\overline{\Omega}$ telle que :

$$\sigma_{ij,j} + f_i = 0 \qquad \text{sur } \Omega$$

$$u_{i,i} = 0 \qquad \text{sur } \Omega$$

$$u_i = g_i \qquad \text{sur } \Gamma_{g_i}$$

$$\sigma_{ij} n_j = h_i \qquad \text{sur } \Gamma_{h_i}$$

$$\sigma_{ij} = -p\delta_{ij} + T_{ij} \qquad \text{sur } \Omega$$

$$T_{ij} = K_{ijkl}\varepsilon_{ij}(u) \qquad \text{sur } \Omega$$

où :
Ω est le domaine
Γ_{g_i} est le bord du domaine sur lequel sont appliquées les conditions essentielles
Γ_{h_i} est le bord du domaine sur lequel sont appliquées les conditions naturelles
u est le champ de vitesse
p est le champ de pression
f, g, et h dépendent des coordonnées spatiales

6.2.2. *Formulation variationnelle*

La formulation variationnelle du problème proposée est la suivante :

Soit $\mathcal{V}$ l'ensemble des champs de vitesse solution :

$$\mathcal{V} = \left\{ u(x)\ x \in \Omega \ \middle|\ u(x) = 0 \text{ pour } x \in \Gamma_g \text{ et } u \text{ régulière} \right\}$$

Soit $\mathcal{S}$ l'ensemble des champs de vitesse virtuels :

$$\mathcal{S} = \left\{ u(x)\ x \in \Omega \ \middle|\ u(x) = g(x) \text{ pour } x \in \Gamma_g \text{ et } u \text{ régulière} \right\}$$

Soit $\mathcal{P}$ l'ensemble des champs de pression solution :

$$\mathcal{P} = \left\{ p(x)\ x \in \Omega \ \middle|\ p \text{ régulière} \right\}$$

Il suffit ici de définir un seul ensemble de champ de pression p car le problème ne fait pas intervenir de condition de bord en pression. On prendra le même ensemble pour les champs de pression virtuel ou solution.

$$\left| \begin{array}{l} \text{Trouver } u \in \mathcal{S} \text{ et } p \in \mathcal{P} \text{ tel que :} \\ \int_\Omega w_i(\sigma_{ij,j} + f_i)dv + \int_\Omega q w_{i,i} dv = 0\ ,\ \forall\ w \in \mathcal{V} \text{ et } \forall\ q \in \mathcal{P} \end{array} \right.$$

6.2.3. *Dérivation dans FEMTheory*

La figure 12 montre la dérivation du problème.

La description est la même que celle faite dans la partie précédente. On peut simplement noter certains points intéressants liés à la formulation mixte. Ainsi à la ligne 11 on peut noter que le système possède deux équations, cela venant du fait que la formulation est construite à partir de deux champs virtuels (pression et vitesse). En bref :

Ligne 1: Représentation du problème.

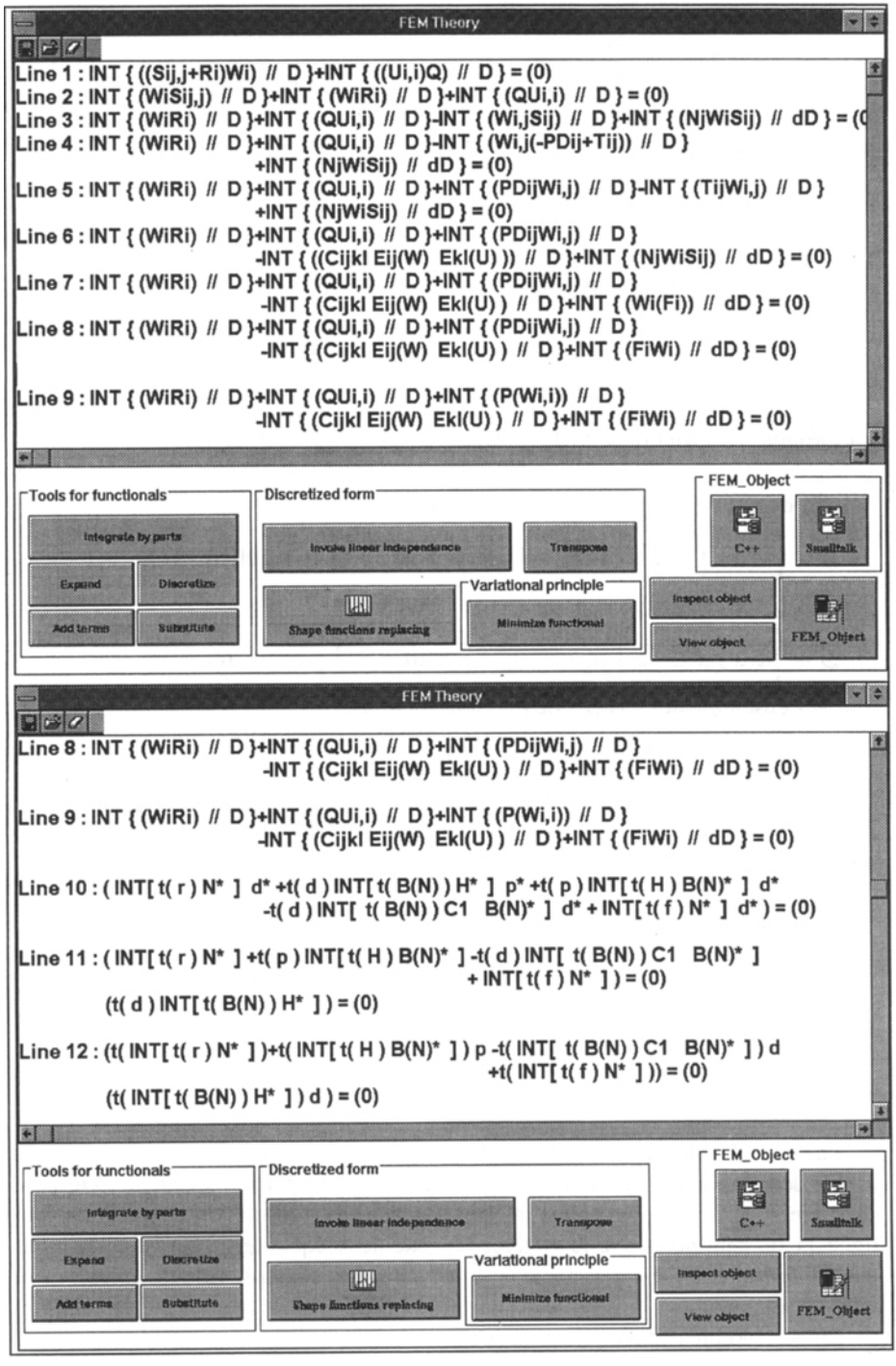

Figure 12. *Dérivation du problème de Stokes dans FEMTheory.*

Ligne 2: Développement de chaque membre de l'équation.

Ligne 3: Intégrations par parties et application du théorème de la divergence à «INT{(WiSij,j)//D}.

Ligne 4: Séparation de la partie sphérique et déviatorique du tenseur «Sij». «Dij» est la représentation de la fonction Dirac.

Ligne 5: Développement de chaque membre de l'équation.

Ligne 6: Substitution de «Sij» en utilisant la loi constitutive.

Ligne 7: Substitution de «SijNj» en utilisant la condition de bord naturelle.

Ligne 8: Développement de chaque membre de l'équation.

Ligne 9: Substitution de «Wi,jDij» par «Wi,i» (contraction d'indices).

Ligne 10: Approximation et discrétisation du problème.

Ligne 11: Invocation du principe d'indépendance linéaire.

Ligne 12: Résultat de la transposition. Remplacement des fonctions d'interpolation (fonctions bilinéaires pour la vitesse, constantes pour la pression). Génération du code E.F.

6.2.4. Test de l'élément dans le code FEMObject

Formulation du problème numérique.

Le problème numérique choisi pour illustrer cette partie est celui de l'écoulement le long d'une cavité. Il est décrit en figure 13. Le maillage contient 121 éléments (Figure 14).

Le coefficient de viscosité dynamique est pris égal à 1.

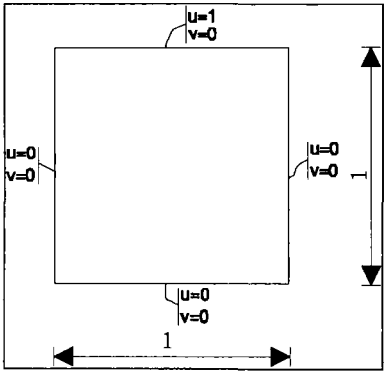

Figure 13. *Description du problème*

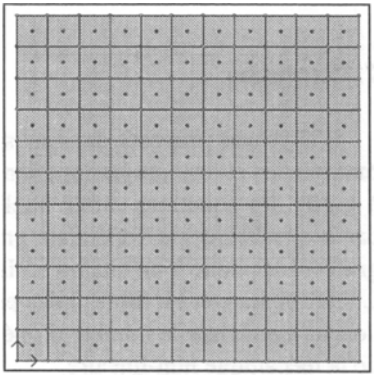

Figure 14. *Maillage du domaine.*

Résultats numériques

Les résultats numériques sont donnés figure 15 pour les données ci dessus ; ainsi les lignes de courant et le champ de pression sont donnés. Ces résultats, ainsi que tous les autres exemples réalisés permettent de montrer que cette formulation n'est pas adéquate, en particulier pour la pression (par exemple un maillage 10*10 éléments met en évidence le phénomène de 'checkerboard' dû au choix de l'interpolation). Ainsi, des formulations stabilisées seront plus appropriées pour ce genre de calculs. On a ainsi pu très rapidement vérifier que cette formulation ne conduit pas à des résultats satisfaisants.

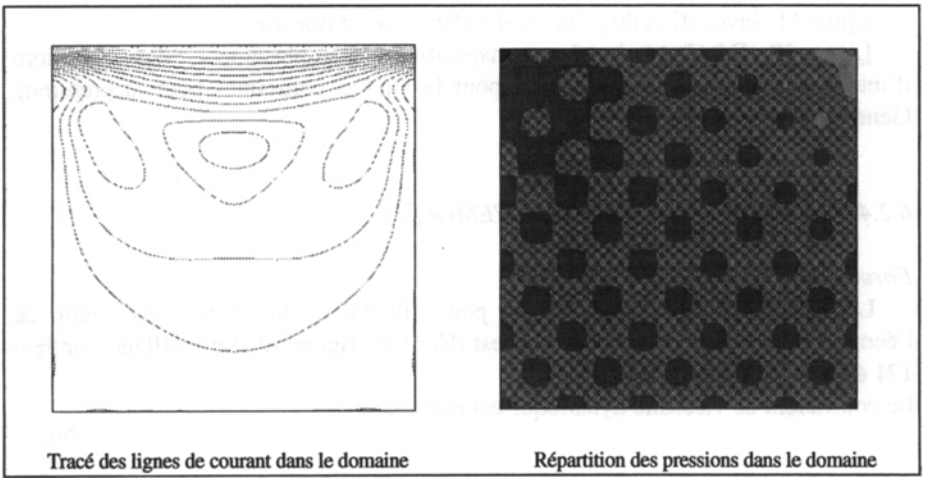

Figure 15. *Présentation des résultats numériques pour le problème de Stokes.*

7. Conclusion

L'approche proposée dans cet article permet d'effectuer la dérivation des formes matricielles et leur programmation automatique dans un code existant, directement à partir des équations différentielles. Les opérateurs reconnus à ce jour, sont la partie symétrique de l'opérateur gradient appliqué à un champ tensoriel, l'opérateur gradient appliqué à un champ scalaire, et ce quel que soit la dimension de l'espace. L'environnement est limité à des problèmes linéaires. La suite logique du travail est donc d'augmenter le nombre d'opérateurs, et d'étendre leur application au domaine de la mécanique non-linéaire.

L'approche développée ici est tout à fait générale et peut être envisagée avec n'importe quel type de langage. On s'est restreint ici à implémenter le code E.F. FEM-Object en version SMALLTALK par souci de rapidité de mise en application des concepts. La convivialité de l'environnement SMALLTALK et l'aspect naturel

de sa conception en font un puissant outil de prototypage. FEM-Object version C++ sera utilisé lorsque le souci d'efficacité dans le calcul numérique prédomine.

La dérivation s'effectue dans un environnement graphique permettant de réaliser les différentes opérations naturellement (fenêtres, boutons, sélection à l'écran, ...). Ce nouvel environnement, FEM-Theory, permet d'avoir une démarche continue depuis le problème théorique posé, jusqu'à la génération automatique du code numérique. Cette approche a été testée sur des problèmes simples aux conditions initiales et conditions de bords. La dérivation complète d'un problème prend environ une heure, tests compris. L'intérêt d'un tel outil pour évaluer le potentiel d'une nouvelle formulation variationnelle est mis en évidence en particulier par une tentative de formulation simple du problème de Stokes.

Ce type d'approche s'insère dans le cadre plus large des systèmes d'aide à la conception dans le cadre de l'analyse mécanique. On se référera à [FRI 92] et [TWO 93]. L'approche objet apporte un modèle naturel de description du problème. Ainsi, on pourra s'affranchir de toute étude sur papier pour les développements théoriques, ici une nouvelle formulation pour la M.E.F., et réduire le temps de développement.

Une approche symbolique-objet a été choisie afin de s'éloigner de la programmation, et de se rapprocher du langage naturel du mécanicien : on manipule ici des expressions, intégrales, formulations variationnelles,... On s'affranchit ainsi des contraintes de programmations liées aux outils de calculs symboliques classiques. L'apport d'outils de haut niveau d'abstraction est d'une aide considérable en mécanique. Dans le domaine des codes numériques, l'approche orientée objet a apporté de grands changements dans l'organisation et la construction des codes. Une meilleure organisation du logiciel permet d'en améliorer la maintenance, la réutilisabilité, et l'extension.

Les caractéristiques de la programmation orientée objet conduisant à ce résultat sont l'encapsulation des données, l'organisation hiérarchique du code associée à la notion d'héritage, le polymorphisme et enfin l'envoi de messages. Il en résulte des codes de modularité accrue, permettant un prototypage rapide, et une décentralisation complète du contrôle au niveau du programme, qui ressemble alors davantage à un assemblage de modules plus ou moins indépendants, mais appartenant au même environnement.

8. Bibliographie

[BAR 89] N.S. Bardel, The application of symbolic computing to the hierarchical finite element method, *Internat. J. Numer. Methods Engrg.* , vol. 28 (1989) pp 1181-1204.

[CEC 77] M.M. Cecchi and C. Lami, Automatic generation of stiffness matrices for finite element analysis, *Internat. J. Numer. Methods Engrg.* , vol. 11 (1977) pp 396-400.

[DUB 92] Y. Dubois-Pèlerin, Th. Zimmermann and P. Bomme, Object-oriented finite element programming : II. A prototype program in Smalltalk, Comput. *Methods Appl. Mech. Engrg.*, vol. 98 (1992) pp. 361-397.

[DUB 93] Y. Dubois-Pèlerin and Th. Zimmermann, Object-oriented finite element programming : III. An efficient implementation in C++, *Comput. Methods Appl. Mech. Engrg.*, vol. 108 (1993) pp. 165-183.
[EYH 94] D. Eyheramendy and Th. Zimmermann, Object-oriented finite element programming : Beyond fast prototyping, *Proceedings of CST 94* , Athens Greece, vol. *Artificial intelligence and object oriented approaches for structural engineering*, Civil Comp Press, (1994) pp. 121-127.
[FRI 92] P. Fritzson and D. Fritzson, The need for high-level programming support in scientific computing applied to mechanical analysis, *Computers & Structures*, vol. 45 (1992) pp. 387-395.
[HOA 80] S. V. Hoa and S. Sankar, A computer program for automatic generation of stiffness and mass matrices in finite-element analyis, *Computers & Structures*, vol. 11 (1980) pp. 147-161.
[HUG 87] T. J. R. Hughes, *The Finite Element Method*, Prentice-Hall, 1987.
[IOA 93] N.I. Ioakimidis, Elementary applications of MATHEMETICA to the solution of elasticity problems by the finite element method, *Comput. Methods Appl. Mech. Engrg.*, vol. 102 (1993) pp. 29-40.
[KOR 79] A.R. Korncoff and S.J. Fenves, Symbolic generation of finite element stiffness matrix, *Computers & Structures*, vol. 10 (1979) pp. 95-118.
[NOO 79] A.K. Noor and C.M. Andersen, Computerized symbolic manipulation in structural mechanics-progress and potential, *Computers & Structures*, vol. 10 (1979) pp. 95-118.
[NOO 81] A.K. Noor and C.M. Andersen, Computerized symbolic manipulation in nonlinear finite element analysis, *Computers & Structures*, vol. 13 (1981) pp. 379-403.
[SMA 93a] Smalltalk for Win32, Reference guide, Digitalk Inc. (1993).
[SMA 93b] Smalltalk for Win32, Encyclopedia of classes, Digitalk Inc. (1993).
[TWO 93] W.W. Tworzydlo and J.T. Oden, Towards an automated environment in computational mechanics, *Comput. Methods Appl. Mech. Engrg.* , vol. 104 (1993) pp. 87-143.
[WAN 86] P.S. Wang, FINGER : A symbolic System For Automatic Generation of Numerical Programs in Finite Element Analysis, *J. Symbolic Computation*, vol. 2 (1986) pp 305-316.
[ZIM 92] Th. Zimmermann, Y. Dubois-Pèlerin and P. Bomme, Object-oriented finite element programming : I. Governing principles, Comput. *Methods Appl. Mech. Engrg.* , vol. 98 (1992) pp. 291-303.

Article reçu le 5 juillet 1994.
Version révisée le 25 décembre 1994.