Calcul des structures et architectures de logiciels

Olivier Débordes

Laboratoire de Mécanique et d'Acoustique
Ecole Supérieure de Mécanique de Marseille
Technopôle de Château-Gombert,
F-13451 Marseille cedex 20
debordes@imtumn.imt-mrs.fr

RÉSUMÉ. *En trente ans, l'évolution des logiciels de calcul de structures a accompagné les progrès de l'informatique. On examine donc comment les récents développements, en informatique, des concepts objet peuvent influer sur les architectures de ces logiciels de modélisation et simulation numériques, pour les éditeurs de logiciels mais surtout pour les équipes et laboratoires de recherche en mécanique. On s'attarde en particulier sur les notions de structuration des données, de méthodes associées aux objets et de relations entre objets, ainsi que sur la notion de langage de modélisation qui peut enrichir significativement ces logiciels.*

ABSTRACT. *For the past thirty years, the evolution of structural analysis softwares has been closely related to the evolution of computer science. Hence, we study how the recent developments of object concepts may influence the architecture of softwares for numerical modeling and simulation, for software editors as well as for research laboratories or groups in the field of finite element methods. We point out the concepts of data structures, of methods associated to objects and of relations between objects, as well as the concept of modeling language as a way to enhance significantly such softwares.*

MOTS-CLÉS : *architectures de logiciels, calcul des structures, éléments finis.*

KEY WORDS: *software architectures, structural analysis, finite elements.*

1. Introduction

La question de l'« architecture des logiciels du futur » est en fait multiforme, même dans le strict cadre des logiciels de calcul des structures. La relative jeunesse (une bonne trentaine d'années) de la notion de logiciel, le fait que la discipline scientifique considérée soit en prise directe avec le monde industriel, et une grande diversité dans les profils des équipes de recherche ou de développement concernées par la question, font qu'on ne peut apporter de réponse simple et encore moins suggérer une vision unique.

C'est donc avec pragmatisme qu'on essaiera (paragraphe 2) de retracer l'évolution des programmes (comme on disait à l'époque) de calcul des structures, de la préhistoire (1970) au moyen âge (1970 ... 1980) jusqu'aux temps modernes (1980), et ce pour comprendre comment on en est arrivé à la notion de logiciel. Puis, on présentera au paragraphe 3 ce qu'il est aujourd'hui convenu d'appeler les « concepts pour architectures de logiciels de modélisation » et qu'on peut résumer par la notion d'objets, et qui a succédé à la programmation structurée, à la programmation procédurale...

Le paragraphe 4 est au cœur du sujet, et, comme annoncé, on ne propose pas une seule réponse, mais on discute d'abord brièvement du cas des logiciels pour utilisation industrielle avant d'aborder de manière plus détaillée le cas d'architectures de logiciels dans le contexte d'équipes ou de laboratoires qui font de la modélisation et de la simulation numériques en mécanique un axe de recherche significatif. Ce contexte impose, à notre avis, de se limiter à des architectures accessibles à de jeunes doctorants en mécanique, dont la culture informatique est limitée et qui ne peuvent consacrer à la partie logicielle la totalité de leur travail de thèse. Cela nous conduit à proposer une architecture qui vise d'abord la simplicité.

On abordera ensuite (paragraphe 5) la question des langages, de programmation d'abord (nous nous contenterons d'en comparer très brièvement quelques-uns), de modélisation ensuite. C'est sans doute sur ce point-là que notre communauté scientifique peut contribuer le plus à enrichir la notion d'architecture de logiciel, en proposant des outils de manipulation des concepts de modélisation plutôt que des objets simplement informatiques.

Dans la conclusion, on reviendra sur les points qui nous paraissent les plus importants, avant de poser une question relevant aussi du génie logiciel, mais aussi de considérations de nature non scientifique.

Sur un sujet aussi vaste, une bibliographie exhaustive ne peut se limiter à une vingtaine de références. Aussi, nous ne prétendons donner que quelques pistes au lecteur qui souhaiterait entreprendre une étude beaucoup plus complète.

2. Evolution des programmes de calcul de structures

2.1. *La préhistoire*

Si on peut considérer que les bases théoriques de la méthode des éléments finis ont été établies au cours des années 1940 (il est classique de citer [COU 43]), il a fallu attendre une certaine maturité de l'informatique matérielle et logicielle (Fortran II) pour voir apparaître au cours des années 1950 les premiers calculs significatifs de structures (il est tout aussi classique de citer [TUR 56]). Ce n'est toutefois qu'au cours des années 1960 qu'ont commencé à être assez largement diffusés, au moins en Europe, des programmes de calcul des structures au sens où nous l'entendons aujourd'hui, c'est-à-dire des programmes d'un intérêt assez général et d'une certaine fiabilité pour résoudre des problèmes concrets dans un secteur scientifique ou technique bien identifié. Le plus populaire, car fourni sous forme de sources, a été le programme SAP (Berkeley). En Europe, on peut citer les programmes CASTEM (CEA), CASTOR (CETIM), SESAM (d'origine norvégienne)... C'est volontairement qu'est utilisé ici le terme de programme plutôt que celui de logiciel car, d'une part c'était la terminologie de l'époque, et d'autre part, la notion de logiciel était encore balbultiante (même chez les informaticiens). Il est quand même vraisemblable que des rudiments (au moins) d'architecture logicielle ont guidé le développement d'un programme comme SESAM qui faisait largement appel à la sous-structuration [[PRZ 83] pour résoudre, avec les moyens de l'époque, des problèmes comportant plus de 10 000 ddl comme on en rencontrait déjà en architecture navale.

2.2. *Le Moyen Age*

C'est sans doute au cours des années 1970, au moins au sein de la recherche publique française, que la question d'architecture logicielle a commencé à se poser avec une certaine acuité pour, d'une part dominer la complexité des programmes de calcul des structures, et d'autre part, permettre le codéveloppement, par plusieurs équipes de recherche, de programmes ou d'unités de programmes présentant une forte compatibilité entre eux.

Citons d'abord le logiciel MODULEF [BER 85] qui a proposé une approche reposant sur des données structurées, et sur une bibliothèque modulaire d'éléments finis. Ce logiciel a connu un succès certain dans le monde de la recherche publique à cause de sa quasi-gratuité, et à cause de son ouverture particulièrement favorable aux échanges de modules, de structures de données, donc favorable au codéveloppement. Ensuite, vers la fin des années 1970 et sous l'impulsion du CEA et de la CISI, est apparue la notion de base de données structurées pour logiciels scientifiques, et en particulier en calcul de structures. Cette notion repose d'abord sur ESOPE [VER 89] qui désigne à la fois l'ensemble d'un petit nombre d'extensions au Fortran IV puis au Fortran 77 permettant de créer et manipuler des

structures de données, et un précompilateur transformant ces extensions en instructions Fortran 77. Ensuite, GEMAT est l'ensemble de modules logiciels permettant de gérer, en mémoire ou sur disques, des bases de données ainsi structurées.

Toujours vers la fin des années 1970, citons le livre de Dhatt et Touzot [DHA 81] dont les parties dédiées aux aspects de programmation constituent, d'une certaine manière, l'aboutissement de l'approche classique « programme pour calcul de structures » avec, entre autres, un petit moteur de gestion dynamique de la mémoire. Cet ouvrage a servi de base, non seulement au cours des années 1980 mais aussi au cours des années 1990, pour le développement de nombreux logiciels internes à une équipe de recherche. Ce succès résulte sans doute du bon compromis réalisé par le livre entre l'exposé des principes de la méthode des éléments finis, celui des méthodes numériques associées, et celui des techniques de programmation correspondantes. Cet équilibre permet à un petit groupe de personnes, voire même une seule, de bien dominer les aspects d'un programme de calcul de structures (dans le même style, citons [BAT 90] et [BAT 96], plus récents).

2.3. *Les Temps Modernes*

Le début des années 1980 a été l'objet, au moins dans la communauté francophone, d'une réflexion sur l'organisation interne (pour les développeurs) et externe (pour les utilisateurs) des programmes de calcul des structures : nous parlerons donc de logiciels et d'architecture de logiciels. Cette réflexion a essentiellement porté sur trois points :

– la structuration des données manipulées par le logiciel et l'organisation de ces structures de données en mémoire (organisation interne) sous forme d'une *base de données*,

– la décomposition des actions en commandes élémentaires indépendantes, regroupables par l'utilisateur en commandes plus élaborées pour lui permettre de piloter à sa guise le logiciel de manière à résoudre des problèmes de plus en plus complexes ; l'outil pour effectuer ces regroupements est appelé *langage de commande*,

– l'implémentation de ces commandes élémentaires indépendantes dans le logiciel sous forme d'une *bibliothèque de commandes.*

Le premier point, déjà abordé au cours de la décennie précédente (MODULEF, ESOPE-GEMAT), visait d'abord à dominer des données de plus en plus nombreuses et complexes (à l'horizon 2000 : millions de nœuds et d'éléments, dizaines de matériaux très variés, centaines de sous-structures, chargements complexes, couplages de disciplines...). Il visait aussi à bien gérer la mémoire (toujours insuffisante), à assurer la rapidité d'accès aux données (calculs toujours trop longs !), et à garantir la pérennité des données sur des dizaines d'années (hantise des utilisateurs en charge de grands projets !).

Le deuxième point était novateur et partait du constat suivant : les développements de logiciels et leur maintenance étaient de plus en plus lourds et coûteux, donc relevant de spécialistes (les développeurs) de plus en plus éloignés des utilisateurs. Il était alors tentant de séparer les développements d'intérêt général (effectués par les développeurs) de ceux à façon pour résoudre des problèmes particuliers qui resteraient à la charge des utilisateurs. On est arrivé ainsi à distinguer les commandes élémentaires indépendantes, d'un intérêt assez général, de la notion de langage de commande permettant à l'utilisateur de faire ce qu'il veut, et en particulier des développements pointus, avec ces commandes.

Le troisième point se rapproche de la notion de bibliothèque modulaire d'éléments finis (MODULEF) mais élargie en fonction des commandes élémentaires retenues.

On arrive ainsi à une architecture telle que celle schématisée sur la figure 1 ; elle a été effectivement mise en œuvre dans les logiciels CASTEM 2000 [CAS 92] et SIC [AUN 90] et [BRE 92], tous deux réalisés au cours de la deuxième moitié des années 1980. Dans le premier, la base de données ne porte pas de nom particulier (on la désigne usuellement par la couche ESOPE-GEMAT [VER 89]), les commandes s'appellent des opérateurs, et le langage de commande s'appelle GIBIANE. Dans le second, la base de données s'appelle banalement le « gestionnaire d'objets » et les deux autres parties ont le même nom que sur la figure 1.

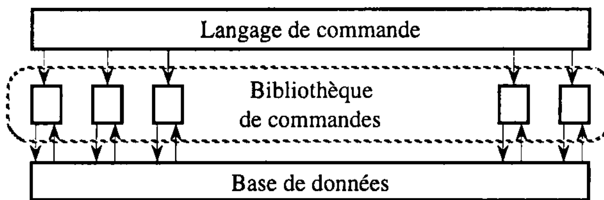

Figure 1. *Exemple d'architecture des années 1980*

3. Concepts pour architectures de logiciels de modélisation

Dans le domaine du calcul de structures, le début des années 1990 a vu l'arrivée massive du concept de programmation objet, accompagné d'un cortège de langages de programmation objet : C++ (le plus répandu), Smalltalk (le plus pur ?), Ada 95 (le plus complet ?), Eiffel (?), Java (le plus récent, donc le plus à la mode !)...

Parallèlement, la technologie objet a fait récemment un gros effort de standardisation pour aboutir à la notion de « UML : Unified Modeling Language »

[MUL 97] qui est un langage de notations (qu'on peut éventuellement prendre aussi pour un langage de spécifications), et qui, avec certains outils de génie logiciel, peut conduire à la génération automatique de parties de code en C++, Java... Dans ce paragraphe, nous nous contenterons de rappeler brièvement les trois principaux concepts de cette approche : données structurées en objets, méthodes associées aux objets, relations entre objets. Nous utiliserons une partie des notations UML pour schématiser ces concepts, et renvoyons le lecteur à [BRE 99] pour une présentation plus complète de ces concepts.

3.1. *Données structurées en objets*

Nous avons vu (MODULEF, CASTEM 2000, SIC) que la notion de structuration des données est assez ancienne. Il suffit d'insister ici sur le caractère systématique de cette structuration : *toute donnée* (qui doit perdurer « quelque temps » dans un logiciel) *doit être structurée dans un objet relevant d'une certaine classe*. On voit ainsi apparaître deux notions de structuration : *structuration des données en objets, structuration des objets en classes* ; c'est la notion de classe qui définit une structure de données, et deux objets relevant de la même classe ne diffèrent que par les valeurs des données placées dans ces structures.

Une première difficulté sérieuse apparaît : *comment structurer les données pour définir les classes* ? A titre d'exemple, examinons donc, figure 2, comment les logiciels CASTEM 2000 et SIC définissent la classe « élément », qu'on peut penser essentielle dans une méthode d'éléments finis. On note tout d'abord que CASTEM 2000 a choisi de regrouper les données liées aux éléments en collections, appelées « maillage », chacun d'eux pouvant être soit un maillage élémentaire (collection d'éléments de même géométrie : triangle, quadrilatère...), soit une collection de maillages élémentaires.

Ensuite, on note certaines similitudes de données entre « élément » et « maillage élémentaire » (n° de famille de l'élément <–> n° du type d'élément géométrique, liste des ID des nœuds <–> liste des n° des nœuds de chaque élément), mais aussi des choix différents. Dans SIC, les objets de la classe « élément » savent de quel matériau ils sont constitués, quelles sont les sollicitations qui leur sont imposées, disposent d'une zone de stockage (pour leurs variables internes...)... Dans CASTEM 2000, ce sont les matériaux, les sollicitations... qui savent quels maillages ils concernent.

On constate donc que *la structuration des données ne peut être pensée indépendamment des relations entre objets* (voir paragraphe 3.3). Par ailleurs, il est souhaitable que cette réflexion tienne compte de normes ou de standards, comme STEP [OWE 93], liés aux logiciels (pré ou post-processeurs, CAO...) avec lesquels un logiciel d'éléments finis est susceptible d'échanger des données. Un autre point est important : *la structuration des données doit éviter toute redondance*.

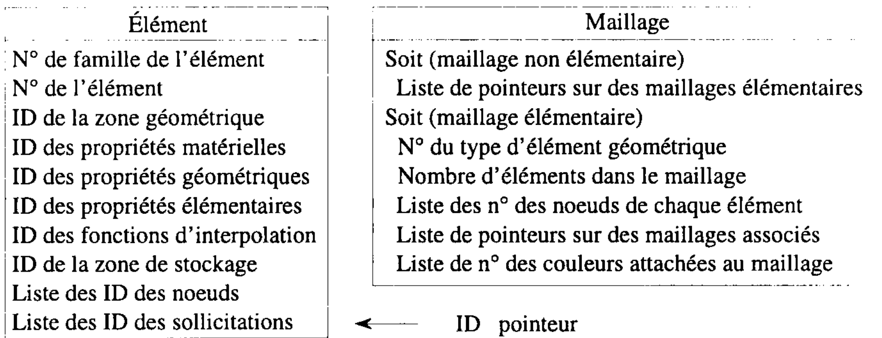

Figure 2. *Classe « élément » dans SIC et classe « maillage » dans CASTEM 2000*

D'un point de vue pratique, la structuration des données est plus un travail de spécification (plus exactement de déclaration) que de programmation, donc automatisable avec des outils simples de génie logiciel dès lors qu'on dispose d'un standard (UML par exemple) pour exprimer ces spécifications ; même sans de tels outils, ce travail de spécification est grandement facilité par la notion de fichiers « include ». A titre indicatif, CASTEM 2000 comporte environ 30 classes d'objets, et SIC environ 90 (dont 27 pour des aspects géométriques non directement liés à la méthode des éléments finis).

3.2. *Méthodes associées aux objets*

Si la programmation objet n'a pas, au moins pour ce que nous avons vu jusqu'à maintenant, eu d'apport significatif pour la structuration des données, le concept de méthodes associées aux objets (plus exactement associées aux classes) est lui très novateur par rapport à la notion de bibliothèque de commandes ou d'opérateurs non explicitement liés aux structures de données sur lesquelles ils opèrent. De récents développements, [ZIM 99] par exemple, renforcent ce caractère novateur.

L'intérêt essentiel de ce concept est de rendre la notion de classe, *i.e.* (données structurées) + (méthodes associées), *autonome, donc facilement exportable.* En ce sens, ce concept est particulièrement important pour le codéveloppement de logiciels entre équipes de recherche, ou, plus généralement, pour la réutilisation de classes d'objets disponibles sur le réseau de l'internet. La figure 3 donne un exemple de méthodes associables à la classe « élément ». Cet exemple fait ressortir deux types de méthodes : un type qui permet à tout objet d'échanger des données avec le monde extérieur (méthodes « fournir_donnée_... » et « recevoir_donnée... »), et un second type de méthodes spécifiques de la classe « élément ».

Elément	
Structuration des données, voir figure 2	
Fournir_donnée_1	Calculer_mat_raideur
Fournir_donnée_2	Calculer_mat_amortissement
......	Calculer_mat_masse
Recevoir_donnée_	Calculer vect forces externes
Recevoir_donnée_	Calculer_vect_forces_internes
......	Calculer_contraintes
	......

Figure 3. *Exemple de méthodes associées à la classe « élément »*

Au contraire de la structuration des données qui implique un travail de programmation relativement facile, la programmation des méthodes d'une classe peut être très importante : par exemple, dans un logiciel de calcul par éléments finis, les méthodes qu'on associerait naturellement à la classe « élément » constituent de l'ordre de la moitié des méthodes (voire plus si on exclut les solveurs). Il est donc tentant d'essayer de *factoriser*, autant que faire se peut, des méthodes.

Pour les méthodes « fournir_... » et « recevoir_... », on pourra les factoriser en créant une « classe_générale » et en lui associant ces méthodes, sous réserve de disposer de mécanismes permettant à cette « classe_générale » de connaître les structures de données de toutes les autres classes, et permettant aux autres classes de fournir ou recevoir les données ainsi gérées par cette « classe_générale ». Le premier mécanisme est facile à concevoir, le second est plus délicat.

Pour les méthodes « calculer_... », il est évident qu'elles peuvent dépendre du type géométrique de l'élément fini, du type de modélisation mécanique (barre, poutre, plaque, coque, volume...), d'hypothèses telles que petites perturbations, grandes perturbations, du type de comportement mécanique du matériau... de la discipline physique concernée. Bref, une grande diversité existe à ce niveau et, dans les principaux logiciels de calcul par éléments finis commercialisés, les bibliothèques d'éléments finis comportent facilement une centaine d'éléments. On est donc amené à considérer que la classe « élément » doit correspondre en réalité à de nombreuses classes qui peuvent avoir des structures de données et des méthodes communes, mais qui auront aussi des structures de données et des méthodes (ou des réalisations de méthodes) spécifiques. On peut illustrer cela par la figure 4 pour comprendre qu'il est facile d'aboutir, lorsque l'on recherche un certain degré de généralité, à plusieurs centaines (voire quelques milliers) de classes d'objets (il suffit de faire un simple calcul de combinatoire).

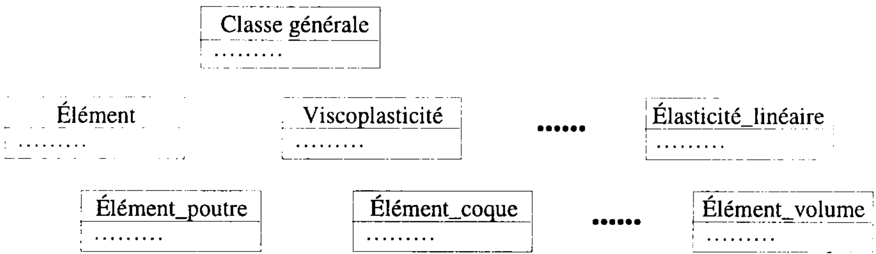

Figure 4. *Génération « spontanée » de classes d'objets*

Une deuxième difficulté apparaît donc : *comment limiter drastiquement la prolifération quasi spontanée* (comme certains mécanismes de langages objet la facilitent) *de classes* ? Une autre question est tout aussi évidente : *à quoi peuvent servir toutes ces classes si on n'établit pas de relations entre elles* ? Cette deuxième question fait l'objet du paragraphe 3.3. Pour revenir à la première question, la notion de bibliothèques (scientifiques pour ce qui nous concerne ici) de méthodes, associées à des classes sans structure de données (appelées « classes utilitaires »), apporte une première réponse. Une deuxième réponse, plus satisfaisante semble-t-il (mais sans doute plus difficile à dégager), pourrait consister à rechercher une forte généricité dans la définition des méthodes, afin d'éviter de devoir créer une nouvelle classe dès que l'on souhaite apporter une variante à une méthode.

A titre indicatif, CASTEM 2000 comporte 442 opérateurs, et SIC comporte 190 commandes. Il est clair que si chaque opérateur ou commande de ces logiciels pouvait opérer sur chaque classe d'objets, on arriverait à un nombre pharamineux de classes au sens de UML. Si, plus raisonnablement, on estime à environ 3 le nombre moyen de classes concernées par chacun de ces opérateurs, alors un simple calcul conduirait à environ 1 000 classes. Il semble qu'effectivement les logiciels « objet » d'éléments finis comporte de l'ordre du millier de telles classes (quelques centaines pour les logiciels les plus simples).

3.3. *Relations entre objets*

Si les classes, au sens UML, sont faites pour être autonomes, *la notion même de logiciel repose sur le concept de classes coopérant, par l'intermédiaire de relations, afin de remplir des fonctions bien précises*. La puissance et l'efficacité d'un logiciel sont donc tributaires de ces relations ; nous les classerons en deux types : les *relations structurelles*, résultant de la manière dont les classes ont été conçues, et les *relations ponctuelles*, résultant de la manière dont ces classes sont utilisées lors d'une exécution du logiciel.

3.3.1. *Relations structurelles*

Elles sont au nombre de deux : l'héritage et le polymorphisme. La relation d'*héritage* est particulièrement utile dans deux cas :

– faciliter la construction de classes dérivées en permettant à celles-ci d'hériter automatiquement de la structuration des données et des méthodes de la classe « parent » en les enrichissant ou en les modifiant, ce qu'on appelle la *spécialisation*,

– faciliter la factorisation de méthodes et le regroupement de structures de données entre plusieurs classes se ressemblant étrangement, ce qu'on appelle la *généralisation*.

Par exemple, une partie de la figure 4 peut bénéficier grandement de cette relation d'héritage pour devenir la figure 5, dans laquelle on a spécialisé la classe « élément » en sous-classes « élément_poutre », « élément_coque », « élément_volume »...

En constatant que les classe « élasticité_linéaire » et « viscoplasticité » ont la même fonction (calculer un tenseur des contraintes à partir d'une histoire du tenseur des déformations), il est alors tentant de généraliser ces classes en introduisant une classe « comportement », comme indiqué sur la figure 6. Dans la foulée, il est plus que tentant de faire un double héritage afin que toute classe « élément_... » puisse bénéficier, pour sa méthode « calculer_contraintes », des méthodes présentes dans la classe « Comportement » et ses classes dérivées. Cela est parfaitement possible en faisant hériter chaque classe « élément_... » aussi de la classe « Comportement », comme illustré sur la figure 6 ; on y constate la génération « spontanée » de sous-classes « élément_poutre_viscoplastique »... Pour éviter de telles proliférations, l'*héritage multiple* est à utiliser avec la plus grande précaution (certains langages de programmation objet ne l'autorisent pas) ; Besson et Foerch [BES 98] proposent une technique d'*association* pour y remédier.

Pour résoudre le problème « comment faire bénéficier chaque classe « élément_... » des méthodes présentes dans la classe « comportement » et ses classes dérivées ? », une autre solution est possible. Elle consiste, dans la programmation de la méthode « calculer_contraintes » des classes « élément_... » à appeler (voir paragraphe 3.3.2) une méthode, appelons-la « contraintes », de la classe « comportement ». Le *polymorphisme* permet alors, en fonction du contexte (nom ou classe ou nombre ou... des arguments d'appel) de décider quelle réalisation (viscoplastique, élastique_linéaire...) de la méthode « contraintes » est concernée. De cette manière, qui peut être dynamique (c'est-à-dire que la décision peut n'être prise qu'à l'exécution), la classe « élément_coque » peut prendre l'apparence et les fonctionnalités d'un classe dérivée « élément_coque_viscoélastique » ou celle de « élément_coque-élastique_linéaire ». Si cette méthode est élégante, elle reste néanmoins assez délicate à utiliser et ses performances sont sujettes à caution.

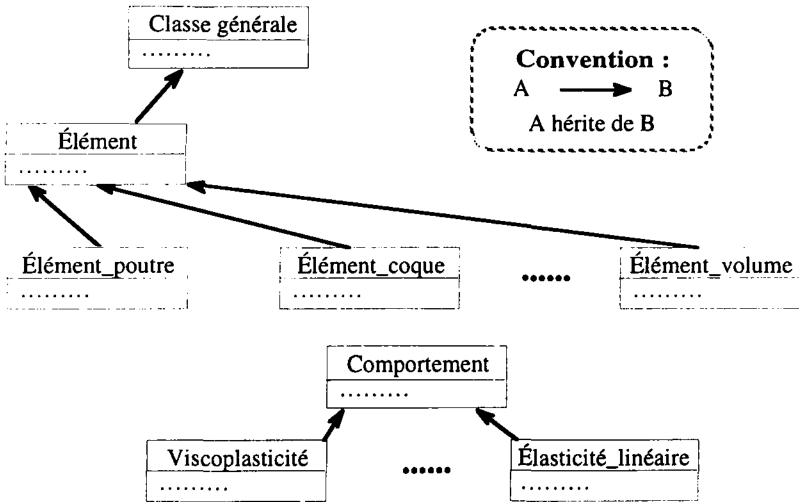

Figure 5. *Exemple d'héritages*

3.3.2. *Relations ponctuelles*

Les relations ponctuelles sont beaucoup plus classiques, et de la forme « un objet envoie un *message* (éventuellement accompagné de données) à un autre objet ou classe pour qu'il ou elle fasse une certaine action (et éventuellement lui retourne un résultat) ». Par exemple, en cours de résolution d'un problème élastique linéaire, des messages de la forme *Calculer_mat_raideur()* seront envoyés à tous les objets relevant d'une classe dérivée de la classe « élément ». C'est volontairement qu'est utilisé le mot *message* pour faire ressortir les potentialités de ces concepts pour le calcul distribué.

Malgré le classicisme de ces relations ponctuelles, le fait qu'elles puissent être très nombreuses et très variées nécessite un soin particulier, aussi bien dans la conception des méthodes associées aux classes (car ces méthodes n'ont de sens que s'il elles sont destinataires de messages) que dans la conception et l'ordonnancement des messages. Pour protéger les données d'une classe d'accès intempestifs, ou ses méthodes d'utilisations abusives, le standard UML propose les notions de données (ou méthodes) privées, protégées ou publiques. L'existence même de ces notions est clairement indicatrice du fait que les relations ponctuelles constituent le moteur du logiciel.

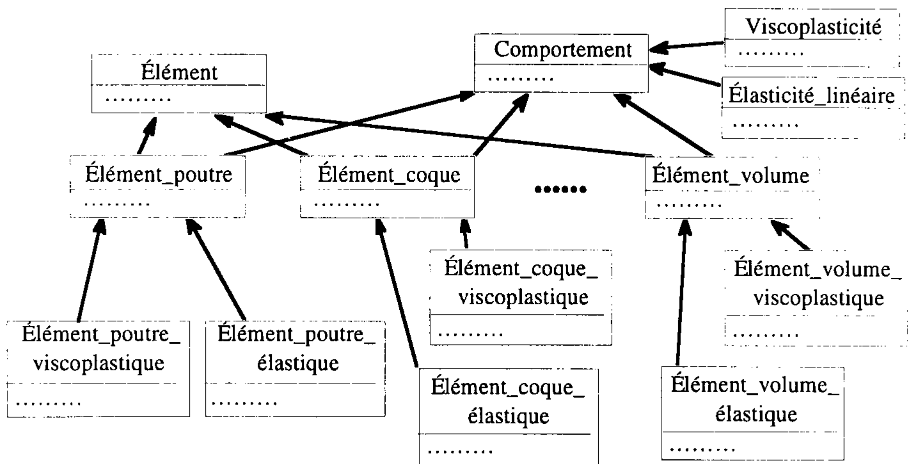

Figure 6. *Exemple de double héritage*

4. A propos d'architectures de logiciels

C'est volontairement que l'on parle d'architectures de logiciels au pluriel, même dans le strict cadre de logiciels de calcul par éléments finis, indépendamment des logiciels amont ou aval ou de ceux susceptibles d'y être couplés (comme pour l'optimisation de formes ou de matériaux). Un minimum de classification s'impose, tant les options, les intérêts, les finalités peuvent être nombreuses ; nous avons choisi de distinguer les architectures de logiciels pour utilisation industrielle de celles de logiciels pour la recherche.

4.1. *Architectures de logiciels pour utilisation industrielle*

Les utilisateurs sont donc en général des bureaux d'études (pour, par exemple, définir, dimensionner, optimiser des pièces mécaniques) et des bureaux des méthodes (définir, dimensionner, optimiser des procédés de fabrication). Sauf, dans quelques cas très particuliers [AST 97] et de plus en plus rares, ils utilisent des logiciels développés, maintenus et commercialisés par des sociétés spécialisées (les éditeurs de logiciels).

Les éditeurs de logiciels dont les produits sont éprouvés et bien implantés sur le marché semblent très réticents à l'idée de remettre en cause tout ou partie de l'architecture de leurs produits ; c'est en particulier le cas pour les logiciels généralistes dont le marché semble proche de la saturation, même si ils sont conscients que l'architecture de ces logiciels (certains ont plus de trente ans) ne peut

être éternelle. Comme les utilisateurs ne sont pas particulièrement sensibles aux notions d'architectures (tout ce qu'ils demandent est que le logiciel remplisse correctement ses fonctions), les éditeurs ont une position très conservatrice. A l'opposé, les nouveaux éditeurs de logiciels, ou les éditeurs qui développent un nouveau logiciel sans réutilisation significative de composants ou modules logiciels, semblent plus ouverts sur les questions d'architecture, sans pour autant envisager forcément des architectures totalement orientées objet.

Même lorsqu'un logiciel de calcul par éléments finis doit être couplé à des logiciels d'architecture objet (comme certains logiciels de CAO), la question est ouverte. Une raison essentielle est l'existence de bibliothèques scientifiques efficaces (le plus souvent en Fortran 77) ; une autre raison, aussi importante, est le maintien d'un bon niveau de performances, point sur lequel les architectures objet sont souvent critiquées.

4.2. *Architectures de logiciels pour la recherche*

On ne discutera pas ici de recherche (au sens de l'informatique) sur les architectures de logiciels de calcul par éléments finis, mais seulement d'architectures de logiciels pour la recherche en mécanique des solides, des structures, des sols... On ne discutera pas non plus du cas où une équipe de recherche utilise un logiciel pour concevoir ou simuler numériquement des essais, un procédé de fabrication... cas d'utilisation somme toute assez proche d'une utilisation industrielle.

Ce qui nous concerne ici est le cas d'une ou plusieurs équipes de recherche qui ont besoin d'une architecture suffisamment ouverte pour pouvoir y effectuer des codéveloppements, y pérenniser ceux de ces développements qui sont d'un intérêt significatif... Le plus souvent, faute d'ingénieurs de recherche ou d'ingénieurs d'études que les équipes pourraient affecter à ces tâches, ces développements sont effectués par des doctorants en mécanique qui n'ont pas reçu de formation particulière ; au mieux, ils dominent à peu près correctement un langage de programmation (souvent Fortran ou C, rarement un langage objet). En outre, leur sujet de thèse étant un sujet de mécanique, ces développements ne peuvent constituer la totalité de leur travail de doctorat et doivent être validés sur des cas réalistes. Enfin, par définition d'un travail de thèse, ces développements doivent permettre de modéliser et simuler des problèmes nouveaux de mécanique, le caractère novateur pouvant provenir des phénomènes modélisés, ou de la formulation des problèmes, ou du degré de complexité des problèmes considérés, ou de l'efficacité des méthodes ou des algorithmes mis en œuvre...

On est ainsi amené à définir un cahier des charges pour une architecture de logiciels pour la recherche :

– les *concepts de l'architecture* doivent pouvoir être très rapidement acquis par un étudiant de niveau bac+5,

– les *principes pour le développement* doivent pouvoir être tout aussi rapidement dominés,

– l'*existant doit être parfaitement visible* pour en faciliter la réutilisation,

– un certain nombre de *services génériques* doit être fourni,

– un *langage simple, mais d'un niveau suffisamment élevé*, doit être disponible pour rendre les développements faciles, rapides et fiables,

– ce *langage* doit être facilement *extensible* pour faciliter l'intégration des développements et en augmenter le niveau,

– un bon niveau de *performances* doit être assuré.

Des trois concepts (structuration des données, méthodes associées aux objets, relations entre objets) exposés au paragraphe 3, les deux premiers sont assez naturels et accessibles à un jeune doctorant. Par contre, selon l'utilisation qui est faite du troisième, l'architecture d'un logiciel peut rester simple ou devenir d'une complexité hors de portée de non-spécialistes.

Les principes pour le développement doivent donc viser à limiter le nombre de classes et de méthodes associées, à limiter l'utilisation de relations structurelles (héritage, polymorphisme) pour éviter la génération spontanée visible (héritage) ou masquée (polymorphisme) de classes, ce qui permettra de limiter les relations ponctuelles entre objets. Toutefois, la recherche nécessite un minimum d'espace vital et des limitations trop drastiques pourraient être un frein à l'imagination des doctorants. Par ailleurs, dans une approche strictement objet (c'est-à-dire une approche selon laquelle les méthodes sont encapsulées dans des classes), ces limitations constituent un frein aussi pour la factorisation des méthodes. Il y a donc là un point très sensible, qui doit retenir toute l'attention des personnes en charge de la définition d'une architecture.

Une manière pour lever en partie ces contradictions pourrait être d'éviter une architecture trop intégrée en concevant des blocs assez autonomes (ce qui est bien dans l'esprit d'une approche objet) ; par exemple, on pourrait avoir un bloc pour toutes les classes vraiment spécifiques des méthodes d'éléments finis, un bloc pour toutes les classes liées aux solveurs, un bloc pour toutes les classes pré ou post-processeurs... En interdisant alors les relations structurelles (héritage, polymorphisme) entre blocs et en limitant drastiquement les relations ponctuelles entre eux, on devrait pouvoir obtenir une architecture assez simple.

Cette simplicité d'architecture permettrait alors d'offrir une bonne lisibilité, sous réserve de disposer d'une documentation claire, complète et à jour. Souvent, les logiciels pour la recherche ont une documentation (quand ils en ont une) incomplète et rarement actualisée. Cette simplicité peut bénéficier grandement de la notion de services génériques ayant pour objectif de décharger les utilisateurs et les développeurs d'un certain nombre de tâches fastidieuses (sauvegardes, reprises,

ramasse-miettes... gestion dynamique de la mémoire si nécessaire). Un exemple de tels services génériques est constitué, dans les architectures des années 1980 (figure 1), par le moteur de la base de données ; dans SIC, ce moteur est capable de gérer tout type de structure de données.

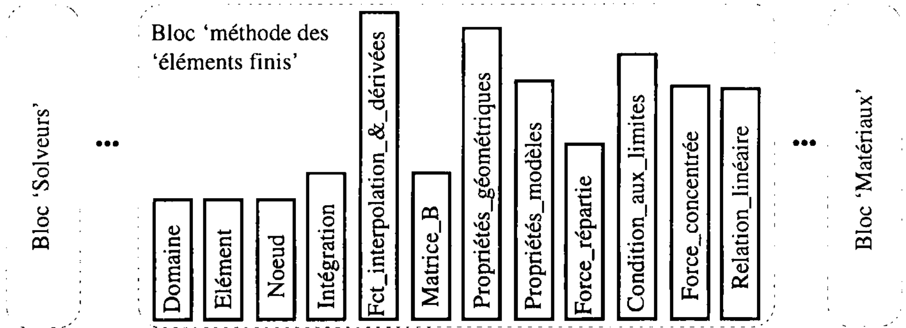

Figure 7. *Exemple d'architecture simple*

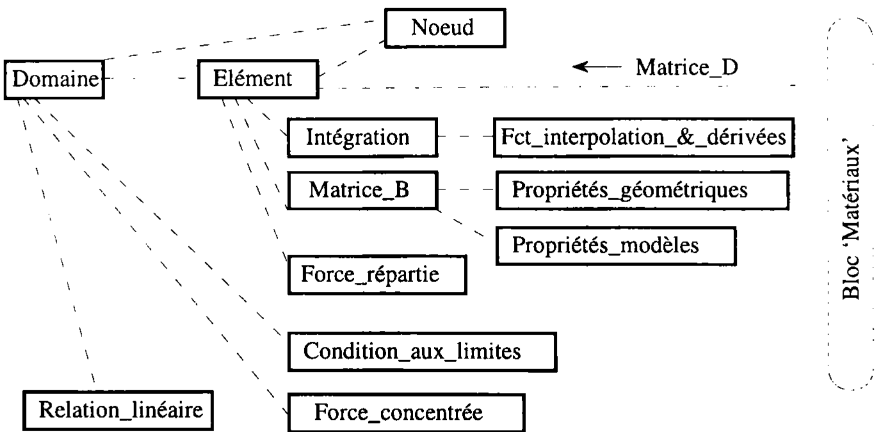

Figure 8. *Diagramme des relations ponctuelles pour un problème d'élasticité linéaire*

A ce stade, on est donc en mesurer d'esquisser une proposition d'architecture simple, figure 7, pour poser (par exemple) un problème d'élasticité linéaire avec des éléments isoparamétriques. Sur la figure 8, on explicite les relations ponctuelles (sans détailler qui émet les messages et quels sont les contenus des messages)

nécessaires au calcul de la matrice raideur et à celui du vecteur second membre. Sur ces figures, on constate qu'on n'a pas eu besoin de relations structurelles (héritage ou polymorphisme), ce qui fait que la figure 7 ressemble beaucoup à la figure 1 dans laquelle on oublierait la couche du langage de commande et dans laquelle on casserait la couche de la base de données pour en rassembler les morceaux avec la couche opérateurs.

5. A propos de langages

Même s'il est essentiel de penser une architecture de logiciels de calcul par éléments finis indépendamment du ou des langages de programmation qui permettront de la concrétiser en un logiciel, il faut être conscient des possibilités qu'ils offrent, de leurs limitations apparentes ou cachées, du niveau de performances qu'ils permettent d'obtenir... et de leur durée de vie probable : un certain nombre de langages encensés au cours des années 1960 ou 1970 ont totalement disparu, d'autres plus récents n'ont pas été adoptés, malgré un certain nombre de qualités intrinsèques, par notre communauté scientifique à cause de lourdeurs ou de performances insuffisantes. Outre les langages de programmation, nous insisterons sur la notion de langage de modélisation qui nous paraît être un concept presque aussi important.

5.1. *Langages de programmation*

Les deux langages les plus utilisés dans les logiciels de calcul par éléments finis sont Fortran 77 (il doit bien subsister un peu de Fortran IV quelque part) et C. On reconnaît à Fortran 77 de meilleures performances numériques, mais aussi des limitations qui font que C a pris une part significative au cours des dix dernières années. Toutefois, aucun des deux ne peut être considéré comme un langage orienté objet, et, pour essayer de comparer quelques langages plus récents, le tableau 1 reproduit un tableau extrait de [MUL 97], auquel nous avons pris le risque de rajouter une colonne Fortran 90 en nous appuyant sur [CAR 96] et [DEC 97].

Dans cette colonne, la présence d'un * indique une propriété non intrinsèque au langage, mais qui peut lui être rajoutée moyennant une construction assez facile à réaliser. Pour chiffrer la comparaison entre les langages, notons 0 une propriété absente, 2 une propriété pleinement satisfaite, et 1 une propriété partiellement satisfaite (héritage simple) ou satisfaite seulement par un artifice de construction (*).

On arrive ainsi à 11 pour Ada 95, 8 pour C++, 12 pour Eiffel, 11 pour Java, 5 pour Smalltalk, et 8 pour Fortran 90. Même sans les *, le score de Fortran 90 n'est pas ridicule, bien qu'il ne soit pas considéré comme un langage de programmation objet.

	Ada 95	C++	Eiffel	Java	Smalltalk	Fortran 90
Paquetage	Oui	Non	Non	Oui	Non	Oui
Héritage	Simple	Multiple	Multiple	Simple	Simple	Multiple*
Généricité	Oui	Oui	Oui	Non	Non	Non
Typage	Fort	Fort	Fort	Fort	Non	Fort
Liaison dynamique	Oui	Oui	Oui	Oui	Oui	Oui*
Parallélisme	Oui	Non	Non	Oui	Non	Oui
Ramasse-miettes	Non	Non	Oui	Oui	Oui	Non
Assertions	Non	Non	Oui	Non	Non	Non
Persistance	Non	Non	Non	Non	Non	Non

Tableau 1. *Propriétés de quelques langages de programmation, d'après [MUL 97]*

5.2. *Langages de modélisation*

A côté des classiques logiciels de calcul par éléments finis, qui offrent une palette de plus en plus large de fonctionnalités activables par des mots-clés, des logiciels comme CASTEM 2000 ou SIC mettent à disposition des utilisateurs un langage de modélisation *interactif*. Il leur permet de formuler de manière plus fine les problèmes à résoudre, de définir leurs propres algorithmes, et d'étendre ce langage de modélisation en introduisant des procédures (CASTEM 2000) ou des macrocommandes (SIC).

Dans CASTEM 2000, les composants de base du langage (appelé Gibiane) sont des opérateurs utilisés comme dans

Objet = **OPÉRATEUR** *objet1 objet2 ... objetn*

où les *objetn* sont des structures de données, **OPÉRATEUR** est le nom d'un opérateur, et où ***Objet*** est une (liste de) structures de données. CASTEM 2000 comporte 442 opérateurs dont certains servent à définir des procédures ou des structures logiques (boucles, tests...).

Dans SIC, le langage de modélisation [BRE 92] est un langage de commande qui ressemble à un langage naturel (très frustre) bâti avec des verbes, compléments, attributs et qualifieurs. Typiquement, une phrase de ce langage est de la forme ;

CALCULER réactions/*nœuds*=[8, 1, 12] **vecteur/***nom* = reac

qui demande le calcul des réactions aux nœuds n° 8, 1 et 12 et de placer le résultat dans le vecteur de nom « reac » (**CALCULER** est un verbe, ***réactions*** est un complément, *vecteur* est un attribut, *nœuds* et *nom* sont des qualifieurs).

L'expérience de quelques centaines (au moins) d'utilisateurs (dont beaucoup de doctorants) de l'un ou l'autre de ces langages de modélisation, est clairement positive. Souvent, le travail de développement du doctorant se limite à affiner, avec le langage de modélisation, des algorithmes de résolution pas à pas de problèmes d'évolution pour mieux tenir compte des spécificités de leur sujet de thèse. Parfois,

il lui faut effectuer des développements plus au cœur du logiciel (intégration d'une loi de comportement...) ; le langage de modélisation permet alors de rendre le prototypage beaucoup plus rapide, mais une phase de codage classique reste indispensable pour obtenir des performances correctes. Dans quelques cas, le langage de modélisation est trop pauvre, et la structuration des données n'est pas assez fine, pour éviter le codage classique dès le début du travail. Une certaine pression apparaît alors pour coupler le langage de modélisation à un langage de programmation interprété, au risque d'accroître significativement la difficulté de l'apprentissage du logiciel. Enfin, on notera l'effort actuel du CEA pour inclure dans Gibiane les éléments lui permettant d'apparaître comme un langage objet.

6. Conclusion

Parler d'architectures de logiciels du futur est un exercice difficile, même dans le strict cadre du calcul par éléments finis ; d'ailleurs ce cadre n'est-il pas trop limitatif à une époque où on constate un gros effort d'intégration de la CAO, des mailleurs, des outils d'analyse de résultats, et bien sûr des logiciels de calcul, dans un même ensemble logiciel disposant d'une interface homme-machine la plus régulière possible ?

En ce qui concerne les logiciels pour utilisation industrielle, et sous réserve que la question soit encore d'actualité (existe-t-il encore une place pour de nouveaux logiciels généralistes de calcul par éléments finis ?), l'utilisation systématique des récents concepts « objet » (pour développer un nouveau logiciel) est sans doute souhaitable pour mieux maîtriser le développement du logiciel, mais nécessite sans doute aussi une équipe de développeurs dominant parfaitement ces concepts. Pour les logiciels existants, l'utilisation d'une partie de ces concepts pour encapsuler l'existant en un ou quelques blocs est une voie envisageable pour simplifier l'architecture interne de cet existant, favoriser les échanges avec d'autres logiciels et obtenir ainsi un degré élevé d'intégration.

En ce qui concerne les logiciels pour la recherche, c'est-à-dire pour les équipes ou laboratoires développant des logiciels ou des modules logiciels de manière significative, il y a un équilibre à trouver entre l'utilisation de concepts et d'outils informatiques d'un degré d'abstraction élevé, et la formation initiale (de mécanique) des personnes amenées à utiliser ces logiciels, y développer de nouvelles fonctionnalités... Ces personnes étant le plus souvent destinées à quitter les équipes ou laboratoires de recherche, l'architecture de ces logiciels doit être plus simple. C'est cette recherche de simplicité qui nous a incités à proposer une architecture n'usant que très modérément des notions d'héritage et de polymorphisme au profit de celle de message qui paraît plus souple. De plus, partir d'une architecture simple laisse davantage d'espace de liberté pour l'enrichir, avant d'arriver à une « usine à gaz » !

Il nous semble que, par son niveau d'abstraction spécifique du métier de modéliseur, la notion de langage de modélisation peut apporter, aux logiciels de calcul mais surtout à leurs utilisateurs, au moins autant, si ce n'est plus, que l'utilisation d'un vrai langage de programmation objet. C'est sans doute dans cette voie qu'on peut espérer enrichir significativement les concepts de modélisation et de simulation numériques en évitant le recours systématique à un langage de programmation pour le moindre développement. Enfin, même si on reconnaît aux approches objets de réelles aptitudes pour la persistance des objets (néanmoins le tableau 1 n'est pas très optimiste sur ce point) et le codéveloppement de logiciels, l'éparpillement des équipes et laboratoires de recherche sur le territoire ou entre de nombreuses structures (CNRS, universités...) constitue un réel frein aux échanges de modules logiciels ou objets, aux co-développements logiciels... Quelles améliorations pourrait-on apporter à cette situation ?

7. Bibliographie

[AST 97]] ASTER, Référence, version 3, EDF, Clamart, 1997.

[AUN 90] AUNAY S., Architecture de logiciels de modélisation et traitements distribués, Thèse de doctorat, Université de technologie de Compiègne, 1990.

[BAT 90] BATOZ J.L., DHATT G., *Modélisation des structures par éléments finis*, Editions Hermès, Paris, 1990.

[BAT 96] BATHE K.J.. *Finite element procedures.* Prentice Hall, 1996.

[BER 85] BERNADOU M., GEORGE P.L., HASSIM A., JOPLY P., LAUG P., PERRONNET A. *et al.*, *Modulef : une bibliothèque d'éléments finis*, Ed. INRIA, Rocquencourt, 1985.

[BES 98] BESSON J., FOERCH R., « Object oriented programming applied to finite element method – Part I : General concepts », *Rev. Eur. Elements Finis*, 5, 1998, Editions Hermès, Paris.

[BRE 92] BREITKOPF P., TOUZOT G., « Architecture des logiciels et langages de modélisation », *Rev. Eur. Elements Finis*, 3, 1992, p. 333-368, Editions Hermès, Paris.

[BRE 99] Breitkopf P.. « Démarche objet et modélisation numérique », *Actes du 4ème Colloque national de calcul des structures* », Giens, 18-21 mai 1999, Editions Teknea, Toulouse, p. 79-84.

[CAR 96] CARY J.R., SHASHARINA S.G., CUMMINGS J.C., REYNDERS J.V.W., HINKER P.J., Comparaison of C++ and Fortran 90 for object-oriented scientific programming, Report LA-UR-96-4064, 1996, Los Alamos National Laboratory.

[CAS 92] CASTEM 2000, Manuel de référence, Rapport CEA/DMT/LAMS, juillet 1992.

[COU 43] COURANT R., « Variational methods for the solution of problems of equilibrium and vibrations », *Bull. Am. Math. Soc.*, 49, 1943, p. 1-23.

[DEC 97] DECYK V.K., NORTON C.D., SZYMANSKI B.K., « Expressing object-oriented concepts in Fortran 90 », *ACM Fortran forum.* 16-1, 1997.

[DHA 81] DHATT G., TOUZOT G., *Une présentation de la méthode des éléments finis*, Editions Maloine S.A., Paris, 1981.

[MUL 97] P MULLER A., *Modélisation objet avec UML*. Eyrolles, 1997.

[OWE 93] OWEN J., *STEP : An introduction*, Information Geometers Ltd., UK, 1993.

[PRZ 83] PRZEMIENIECKI J.S., *Theory of structural matrix analysis*, Mc Graw-Hill, 1968.

[TUR 56] TURNER M.J., CLOUGH R.W., MARTIN H.C., TOPP L.J., « Stiffness and defection analysis of complex structures », *J. Aeraunotical Science*, 23, 1956, p. 805-823.

[VER 89] VERPEAUX P., Esope Gemat. Manuel d'utilisation, Rapport CEA/DRN/DMT /SEMT/LAMS, n° 89/435, 1989.

[ZIM 99] ZIMMERMANN T., BOMME P., COMMEND S., « Un concept d'objet intelligent pour applications scientifiques », *Actes du 4^e Colloque national de calcul des structures*, Giens, 18-21 mai 1999, Editions Teknea, Toulouse, p. 85-90.