FORMALISATION ET EXPLOITATION DE L'EXPERTISE DE MODELISATION AU SEIN DES LOGICIELS DE SIMULATION: COUPLAGE DES APPROCHES SYSTEME EXPERT ET MODELISATION OBJET. APPLICATION A LA MODELISATION DE DISPOSITIFS ELECTROMAGNETIQUES.

Olivier Defour

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__|__|__|__|__|__|__|__|__|__|

THESE

pour obtenir le grade de

DOCTEUR DE L'INPGrenoble

Spécialité : Génie Electrique

préparée au Laboratoire d' Electrotechnique de Grenoble
dans le cadre de l'**Ecole Doctorale « Electronique, Electrotechnique, Automatique, Télécommunications, Signal »**

présentée et soutenue publiquement

par

Olivier DEFOUR

le 13 Décembre 2002

Titre :

FORMALISATION ET EXPLOITATION DE L'EXPERTISE DE MODELISATION AU SEIN DES LOGICIELS DE SIMULATION : COUPLAGE DES APPROCHES SYSTEME EXPERT ET MODELISATION OBJET. APPLICATION A LA MODELISATION DE DISPOSITIFS ELECTROMAGNETIQUES.

Directeurs de thèse :

Philippe MASSE
Yves MARECHAL

JURY

Pr. Paul JACQUET, Président
Dr. Thomas ZIMMERMANN, Rapporteur
Dr. Laurent NICOLAS, Rapporteur
Pr. Philippe MASSE, Directeur de thèse
Pr. Yves MARECHAL, Co-encadrant

Introduction

Ce travail de recherche s'est déroulé au Laboratoire d'Electrotechnique de Grenoble (LEG-ENSIEG-INPGrenoble), au sein de l'équipe « Modélisation et CAO en électromagnétisme », en collaboration avec le Laboratoire des Matériaux et du Génie Physique (LMGP-ENSPG-INPGrenoble). Cette collaboration a été soutenue par la fédération de laboratoires ELESA.

Avec plus de cent trente membres, dont près de la moitié d'étudiants en doctorat et post-doctorat, le LEG est le premier acteur de la recherche universitaire en France et en Europe dans le domaine du Génie Electrique.

La recherche au sein de l'équipe « Modélisation et CAO en électromagnétisme » se développe suivant deux axes :

- la modélisation des phénomènes électromagnétiques dans les milieux continus,
- la Conception Assistée par Ordinateur pour l'électromagnétisme.

Le premier axe constitue l'activité de fond de l'équipe. Elle est concrétisée par le développement de modèles de matériaux magnétiques et de méthodes générales pour la modélisation des milieux continus en électromagnétisme.

Les activités de recherche au sein de l'équipe sont soutenues par des contrats industriels constants. Ces derniers portent sur le développement de nouveaux modèles toujours plus performants, aussi bien dans la représentation de la physique, que sur la précision et le coût de la simulation.

Pour pérenniser ce savoir-faire, l'équipe développe ses propres environnements de simulation et outils CAO, où sont intégrés les nouveaux modèles physiques développés. Ces logiciels sont utilisés par les chercheurs du laboratoire et par le milieu industriel. On peut citer comme exemple les logiciels Flux2D® et Flux3D®, codéveloppés avec la société CEDRAT, et leaders européens pour la simulation des phénomènes électromagnétiques.

Le réalisation d'outils CAO demande une organisation rigoureuse des développements informatiques, mais également une intense activité conceptuelle amont, qui se fait en collaboration avec le LMGP. Le travail que nous présentons dans cet ouvrage fait partie de cette réflexion amont.

Sous l'influence de la miniaturisation, la simulation du comportement des dispositifs implique la mise en œuvre de modèles multi-physiques de plus en plus fins. Or, on constate que plus un modèle est proche d'un phénomène physique complexe, et plus sa mise en œuvre au sein du logiciel et son utilisation sont sujet à des restrictions.

Le savoir-faire et la compétence des concepteurs de modèles physiques s'incarne donc non seulement dans un modèle numérique implanté informatiquement, mais également dans la compréhension et la maîtrise des limitations de chaque modèle.

Ces deux parties complémentaires sont de natures différentes : le modèle numérique est d'essence **algorithmique**, alors que ses limitations appartiennent au domaine de l'**expertise**. Cette dualité de la connaissance se retrouve jusqu'à sa mise en œuvre, dont le résultat est d'une part un logiciel, et d'autre part généralement le manuel d'utilisation associé.

L'objet de notre recherche est d'introduire l'expertise au sein du logiciel de simulation numérique et de l'exploiter. Notre but n'est pas de concevoir un manuel d'utilisation en ligne, mais bien d'**intégrer l'expertise au processus global de modélisation et de simulation des dispositifs**.

L'expertise contenue dans un manuel d'utilisation a une structure inadaptée à un traitement informatique. L'exploitation de cette connaissance va donc nécessairement passer par une phase de **formalisation**. Sous cette forme structurée, elle sera alors exploitable informatiquement.

Une fois formalisée, l'expertise peut être traduite en structures de données informatiques. Cependant, l'information contenue dans ces données n'est pas de nature algorithmique, comme les modèles numériques. Pour l'exploiter, nous devrons mettre en œuvre une **technologie adaptée** au sein du logiciel.

Les structures de données et la technologie d'exploitation dédiée sont les éléments fondamentaux de notre travail. Ils n'en constituent pas le but. Une fois ces éléments disponibles, il faudra les mettre en œuvre au travers de **nouvelles fonctionnalités**. Ces dernières devront s'intégrer de manière naturelle au processus de simulation, sans l'alourdir, et en offrant de réels avantages.

Dans le prochain chapitre, nous nous pencherons sur les difficultés concrètes rencontrées par les utilisateurs de logiciels de simulations numériques, dues à l'absence de l'expertise des concepteurs de l'application. A cette occasion, nous cernerons deux exploitations intéressantes de l'expertise, pour l'**assistance dynamique à la modélisation des dispositifs**, et pour la **gestion des discontinuités de modèles**. Le chapitre suivant sera destiné à l'analyse des outils disponibles pour la formalisation et l'exploitation de l'expertise. Les chapitres trois et quatre sont dédiés respectivement à l'implantation des deux types d'exploitation que nous avons distinguées. Enfin, nous conclurons ce travail, et présenterons quelques prospectives à court et moyen termes.

Problématique

Sommaire

1 Domaines de validité et logiciels de simulation

Le premier paragraphe de ce chapitre illustre, au travers de quelques exemples concrets de modélisation de dispositifs, les difficultés qui sont rencontrées par les utilisateurs de logiciels de simulations numériques. Ces difficultés sont issues de certaines carences au niveau des logiciels. Leur analyse nous permettra de définir le cahier des charges d'outils destinés à pallier ces manques.

1.1 Mise en situation

Les principaux modèles qui régissent les phénomènes de la physique des milieux continus ont été pour l'essentiel établis au dix-neuvième siècle par Maxwell, Navier, Stokes, Kelvin,... Le but de ces modèles est de déterminer la valeur et la variation de **grandeurs physiques** caractéristiques (champs électrique, magnétique, température, pression, ...) en fonction des **propriétés** du milieu où ces grandeurs sont définies.

Ils se présentent sous la forme de systèmes d'équations, impliquant généralement des dérivées partielles de fonctions continues par morceau qui modélisent les grandeurs. L'obtention d'une solution exacte de ces systèmes n'est pas toujours possible sous forme analytique dans le cadre de l'analyse fonctionnelle.

Pour compléter les techniques analytiques, on fait appel à l'analyse numérique. Le but est de chercher une solution approchée du problème. Elle est obtenue en transformant le système d'équations aux dérivées partielles en une suite de systèmes linéaires à résoudre.

La puissance des ordinateurs actuels nous permet de construire et de résoudre de tels systèmes, afin de simuler le fonctionnement d'un dispositif réel avec une prise en compte très fine de la physique impliquée.

1.1.1 Concepts de base d'un logiciel de simulation

les environnements informatiques qui permettent la modélisation et la simulation de dispositifs complexes sont désignés sous l'appellation de **logiciels de simulations numériques**. Ils ont en commun de nombreuses caractéristiques.

a- Définitions usuelles

On appelle **formulation** la représentation mathématique du modèle physique. On associe à une formulation son **modèle numérique** qui est l'ensemble des expressions algébriques mises en œuvre par l'algorithme de construction des systèmes linéaires. Il a pour paramètres les propriétés des milieux modélisés ainsi que la géométrie du domaine où est recherchée la solution.

On appelle modèle de **matériau** l'ensemble des propriétés caractérisant le milieu. Chaque propriété est définie par un modèle mathématique permettant son évaluation.

La **région** est l'entité qui associe un modèle numérique et un modèle de matériau sur un domaine géométrique particulier. En cela, une région modélise un phénomène physique spécifique, circonscrit à une **géométrie** particulière. Le concept de région est fondé sur la trilogie phénomène/matériau/géométrie caractéristique de la physique des milieux continus. Le dispositif est décrit sous la forme d'un ensemble de régions. Cette description, qu'elle soit explicite ou implicite, réalise un partionnement de la physique modélisée.

Cette première étape de modélisation est appelée étape de pré-processing. La mise en œuvre de ces modèles permet la construction des systèmes linéaires globaux dont les solutions forment une

approximation des grandeurs physique. La construction de ces systèmes et leur résolution constitue l'étape de **simulation** proprement dite.

b- Simulations des régimes permanents et transitoires

On distingue deux catégories de simulations. Les simulations des **régimes permanents** englobent les problèmes statiques et harmoniques. Dans ce cas, le solveur calcule les valeurs des paramètres inconnus en fonction du modèle du dispositif conçu par l'utilisateur. Ce dernier a donc la maîtrise totale des paramètres d'entrées du solveur.

La simulation des **régimes transitoires** est plus complexe. Elle se déroule comme une séquence d'appels au solveur. À l'issue de chaque appel, les solutions calculées par le solveur sont stockées. Elles sont par la suite réutilisées comme paramètres d'entrées du solveur lors de l'appel suivant.

Ainsi, au cours d'une simulation de régime transitoire, l'utilisateur n'a pas la maîtrise des entrées du solveur, à l'exception du premier appel. En effet, dans ce cas particulier, il spécifie lui-même les conditions initiales.

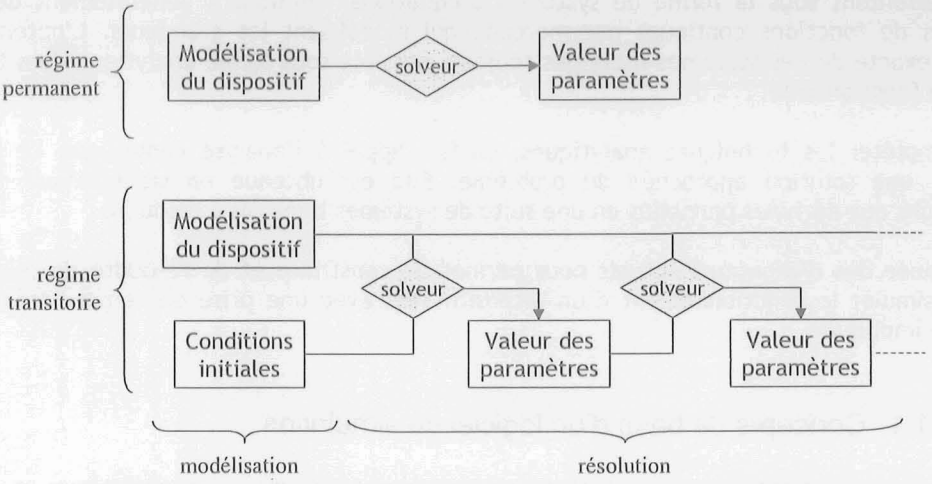


Figure 1 : Simulations des régimes permanents et transitoires

1.1.2 Validité d'un modèle

Tous les modèles numériques mis en œuvre par les logiciels de simulation analysés ici sont issus de la transformation d'un modèle physico-mathématique originel : un système d'équations aux dérivées partielles complété par le modèle des matériaux utilisés et celui de la géométrie du dispositif.

Ce premier modèle est valide dans le cadre d'un ensemble d'hypothèses physiques, formulables sous la forme de contraintes mathématiques. Le traitement numérique lui-même impose également le respect de contraintes d'ordre numériques. Cet ensemble de contraintes définit le **domaine de validité** du modèle numérique utilisé.

Un exemple typique d'hypothèse physique est de supposer une propriété constante ou linéaire dans un milieu. Un exemple de nature numérique est la contrainte de Courant-Frédérics-Lévy (CFL) associée au schéma numérique explicite de Lax-Wendroff [4.1] pour la résolution de l'équation d'advection :

$$\frac{\partial U}{\partial t} + a.\frac{\partial U}{\partial x} = 0$$

où 'a' est un coefficient réel. Ce schéma numérique explicite d'ordre deux s'écrit :

$$U_{n+1,j} = U_{n,j} - (a\Delta t/\Delta x)(U_{n,j+1} - U_{n,j-1}) + (a\Delta t/\Delta x)^2 (U_{n,j+1} - 2\,U_{n,j} + U_{n,j-1})/2$$

où $U_{n,j}$ dénote la valeur de U au temps t_n et au point x_j, et Δx et Δt sont respectivement les pas d'espace et de temps utilisé pour la résolution. Ce schéma est L^2-stable sous la condition CFL :

$$\frac{\Delta t}{\Delta x}|a| \leq 1$$

Les domaines de validité proposent une information contextuelle riche. Au même titre que les modèles numériques, ils font partie du savoir-faire et des compétences des concepteurs des logiciels de simulations numériques. Cependant, contrairement aux équations elles-mêmes, les domaines de validité ne sont généralement pas introduits directement dans l'application.

Cette séparation des compétences entre d'une part le modèle numérique, et d'autre part son domaine de validité, entraîne des situations absurdes.

Prenons un exemple concret, extrait des travaux de simulation effectués par des élèves ingénieurs, dans le cadre des projets de Calcul Scientifique. L'objectif est d'étudier l'écoulement d'un fluide avec transfert thermique à l'aide d'un logiciel commercial.

Le problème est ici constitué d'une unique région dont la géométrie est un tube en 'T'. Le milieu considéré est modélisé par un fluide caractérisé par sa viscosité cinématique, sa conductivité et sa densité, supposées ici isotropes et constantes. Le modèle numérique utilisé calcule les champs de vitesse et de température du fluide.

A l'aide du logiciel, les étudiants calculent les solutions en faisant varier quelques caractéristiques du problème. Ils constatent que, pour une viscosité faible et une vitesse d'entrée élevée, la solution calculée par le logiciel est aberrante (Figure 2 et Figure 3), alors que le logiciel ne présente aucun avertissement ou message d'erreur.

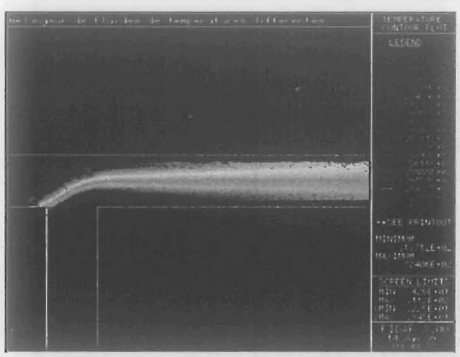


Figure 2 : Résultat correct

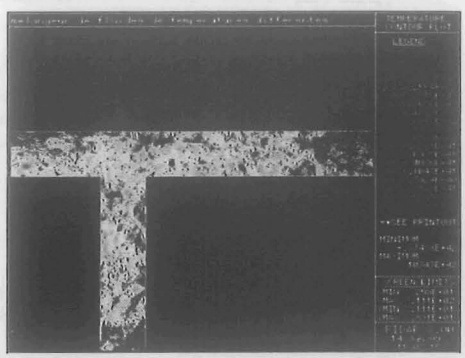


Figure 3 : Résultat aberrant

Seul le « bon sens » de l'utilisateur lui permet alors de rejeter cette solution. En effet, le problème modélisé est alors en régime turbulent, caractérisé par un nombre de Reynolds supérieur à 2400. Dans ce cas, il en conclut que le modèle numérique utilisé n'est pas valide. Cette contrainte est bien connue en mécanique des fluides, et plus généralement dans tous les phénomènes de transport.

Le logiciel admet donc que l'utilisateur puisse mettre en œuvre une méthode numérique particulière hors de son domaine de validité. Cet exemple illustre bien comment la fiabilité de la plupart des logiciels de simulations numériques peut très facilement être mise en défaut.

La question qui est soulevée ici, et à laquelle nous nous efforcerons d'apporter une réponse la plus générique possible, est de savoir comment prendre en compte les domaines de validité des modèles utilisés pour simuler un dispositif, et de garantir la fiabilité des résultats.

1.2 Modélisation des régimes permanents

Analysons les problèmes auxquels est confronté l'utilisateur lors de la modélisation de son dispositif. L'exemple choisi est celui du tutoriel de magnétostatique [3.2.4] du logiciel Flux3D™, commercialisé par la société Cedrat. Flux3D est un logiciel dédié à « l'analyse des dispositifs magnétiques, électriques et thermiques par la méthode des éléments finis ». Il est développé conjointement par la société CEDRAT et le Laboratoire d'Electrotechnique de Grenoble.

L'objectif de ce tutoriel est de parvenir à simuler le fonctionnement d'un contacteur électromécanique. Le contacteur étudié se compose d'une partie fixe (culasse) et d'une palette mobile en matériau ferromagnétique. Un aimant permanent maintient le contacteur en position fermée. L'alimentation en courant des bobines permet le basculement de la palette d'une position à l'autre.

La modélisation du contacteur comprend deux étapes [3.2.3] :

- la description géométrique du dispositif
- la description de la physique mise en œuvre avec :
  - § la modélisation des matériaux (air, aimant, acier)
  - § la création de régions comprenant :
    - le choix d'un matériau
    - le choix d'une formulation
    - le choix du support géométrique
  - § et enfin la création des inducteurs.

La première étape de description géométrique du dispositif amène déjà l'utilisateur à faire certains choix concernant le modèle qu'il va concevoir, notamment concernant la dimension (2D ou 3D) ou la présence de symétries ou de périodicité.

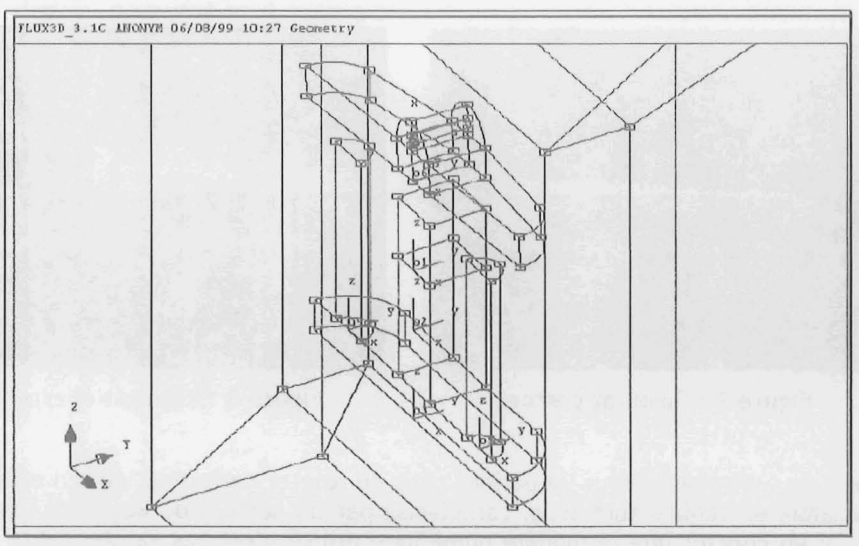


Figure 4 : Modélisation géométrique d'un contacteur sous Flux3D

Ayant décrit la géométrie du contacteur, l'utilisateur va maintenant spécifier le type de phénomène physique qu'il souhaite simuler. Pour cela, il va activer le « modèle » (de physique) adéquat dans Flux3D. Un modèle regroupe un ensemble de formulations.

Le logiciel Flux3D propose un menu comportant actuellement vingt modèles, allant de la modélisation de phénomènes diélectriques (2D, 3D ou axisymétrique) aux phénomènes magnétiques transitoires (2D, 3D ou axisymétrique). De nouveaux modèles, et par conséquent de nouvelles formulations, sont constamment développés, validés et intégrés dans le logiciel, afin d'étendre les domaines physiques descriptibles par Flux3D.

Par l'activation d'un modèle, l'utilisateur fait implicitement le choix de se placer dans le cadre de certaines hypothèses sous lesquelles les formulations disponibles dans ce modèle ont été développées. Ces hypothèses ne sont pas décrites explicitement à l'utilisateur, car elles sont supposées faire partie de sa culture générale.

Par exemple, les modèles magnétostatiques sont formulés sous l'hypothèse qu'aucun courant induit ne peut se développer dans les conducteurs. Les formulations associées ne sont donc pas conçues pour les prendre en compte, contrairement aux formulations des modèles magnéto-transitoires.

Pour chaque type de phénomène physique (magnétostatique par exemple), il existe généralement trois modèles : 2D, 3D ou axisymétrique. Or, lors de la description géométrique de son dispositif, l'utilisateur a déjà spécifié dans quelle catégorie se plaçait son modèle. Si le logiciel avait tenu compte dynamiquement de l'information déjà contenue dans sa base de données, il aurait pu proposer à l'utilisateur un menu mieux adapté à son problème spécifique, soit quatre modèles axisymétriques, sept modèles 2D ou neufs modèles 3D, au lieu de la totalité des modèles.

	Propriétés magnétiques			Propriétés électriques et diélectriques			
Modèles	Perméabilité relative μ	Perméabilité relative complexe μ	Induction rémanente (aimants)	Permitivité relative ε	Angle de perte $tg\delta$	Résistivité source normale ρ	Résistivité source tangentielle ρ
Constante	● ■	●	●	● ■	●	●	● ■
Droite						● fonction de la température	● ■ fonction de la température
Exponentielle						● fonction de la température	● ■ fonction de la température
Courbe de saturation splines	● ■ courbe B(H)			● ■ courbe D(E)			
Courbe de saturation analytique	● ■ pente origine + aimantation saturation			● ■ pente origine + polarisation saturation			
Parabole + droite	● ■			● ■			
Constante * exponentielle	● μ fonction de la température						
Saturation analytique * exponentielle	● B(H) fonction de la température						
Valeur nodale			●			●	●
Sous programme utilisateur	● ■ sauf fonction de la température	●	●	● ■	●	●	● ■

● Propriété isotrope ■ Propriété anisotrope

Tableau 1 : Exemples de modèles de propriétés

La modification dynamique du menu permettrait dans ce cas précis d'éviter par exemple que l'utilisateur ne sélectionne un modèle incompatible avec sa géométrie. Dans l'exemple traité par le tutoriel, le modèle choisit est le modèle magnétostatique tridimensionnel.

L'utilisateur peut maintenant s'attaquer à la phase de modélisation des matériaux du contacteur. En magnétostatique, les seules propriétés nécessaires sont la perméabilité magnétique μ et la conductivité électrique σ.

Pour modéliser chacune de ces propriétés, Flux3D dispose d'une large palette de modèles paramétrables (Tableau 1).

Chaque propriété peut être modélisée par au moins deux modèles différents, et jusqu'à sept. Pour chaque modèle, l'utilisateur saisit une liste de paramètres. Par exemple, pour modéliser la perméabilité magnétique μ de l'acier supposée isotrope, le tutoriel préconise l'emploi du modèle non-linéaire « B(H)_a_scal ». L'utilisateur paramètre son modèle en spécifiant l'aimantation à saturation J_S, la pente à l'origine 'a' et un coefficient d'ajustement du coude 'C_0':

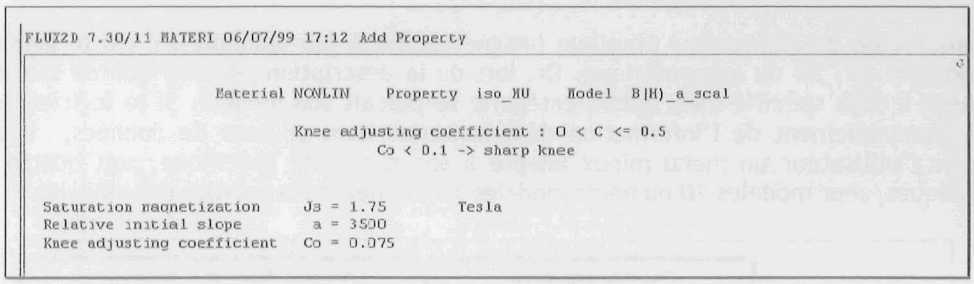


Figure 5 : Paramètres du modèle de perméabilité pour un matériau non-linéaire

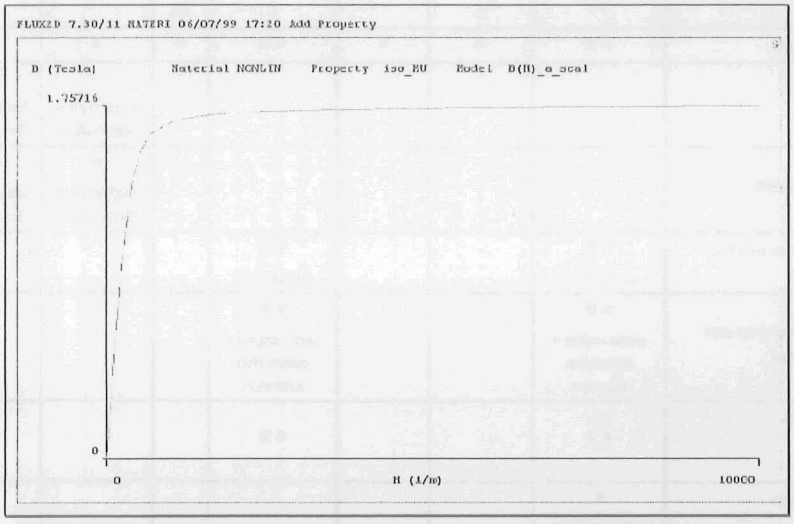


Figure 6 : Visualisation du modèle de perméabilité paramétré

On remarque sur la Figure 5 de quelle manière les concepteurs attirent l'attention de l'utilisateur sur la plage de valeur possible pour le paramètre d'ajustement du coude : $0 < C_0 <= 0.5$. La mise en œuvre d'une vérification algorithmique immédiate de cette contrainte lors de la saisie d'une valeur suffit à s'assurer que cette dernière est bien valide.

La contrainte sur le paramètre C_0 est très simple, et ne fait intervenir qu'une unique variable qui est le paramètre lui-même. C'est pourquoi la vérification algorithmique de la valeur saisie est simple à mettre en œuvre.

C'est toujours le cas lorsque la contrainte ne fait intervenir que des variables intrinsèques au modèle (ici : J_S, a et C_0). Cependant, la contrainte peut être de nature plus complexe, et faire intervenir des variables dont la valeur n'est pas définie au moment du choix.

Par exemple, physiquement, on sait que la perméabilité magnétique de certains matériaux chute brusquement lors du passage du point de Curie. On passe alors d'un modèle non linéaire dépendant de la température à un modèle constant .

Le modèle non-linéaire est par conséquent défini pour une valeur de la température inférieure au point de Curie du matériau. Cette valeur n'est pas connue lors de la conception du modèle, mais au moment de son utilisation. L'évaluation de cette contrainte est donc reportée ultérieurement à la conception, et devrait être effectuée préalablement à toute utilisation. En effet, utiliser un modèle dont la garantie de validité n'est pas assurée, c'est risquer de faire échouer la simulation.

Après la création de ses matériaux, l'utilisateur va s'attaquer à la création des régions. Comme nous l'avons précisé au premier paragraphe de ce chapitre, la région est l'entité qui regroupe une formulation, un matériau et un support géométrique. En magnétostatique tridimensionnel, les formulations disponibles sont au nombre de cinq :

Modèle	Formulation
MAG_STAT_3D	MS3SCA
	MS3RED
	MS3SCRT0W
	MS3VEC
	MS3AWT0W

Tableau 2 : Le modèle magnétostatique tridimensionnel

Le logiciel Flux3D comprend actuellement vingt modèles regroupant plus d'une centaine de formulations différentes. Pour un modèle choisi, l'utilisateur peut avoir le choix jusqu'à dix formulations différentes. La description des formulations est disponible dans le manuel de référence ou dans les notes de principe. Ces documents décrivent également les limitations et les contraintes d'utilisation du logiciel. Concernant les formulations, les contraintes sont de natures très diverses :

- couplages possibles entre formulations,
- prise en compte des sources de courant ou de champ, des symétries, des conditions aux limites,
- caractéristiques de la géométrie ou du matériau, ...

L'ensemble de ces contraintes définissent le **domaine de validité** des formulations. Pour une formulation donnée, la description de son domaine de validité est difficile. En effet, les nombreuses indications sont réparties sur l'ensemble de la documentation, soit 917 pages réunies en quatre volumes pour le manuel de référence du logiciel Flux3D.

Or, cette connaissance est indispensable pour pouvoir modéliser convenablement un dispositif. Ainsi, la formulation en potentiel scalaire total (MS3SCA) ne prend pas en compte les bobines, contrairement à la formulation en potentiel scalaire réduit (MS3RED). Cette contrainte est implicitement utilisée dans le tutoriel de magnétostatique lors de l'affectation de la formulation en potentiel scalaire réduit à la région AIR pour prendre en compte les bobines présentes (Figure 7).

Supposons que l'utilisateur ait créé une région contenant des bobines. Lors de l'affectation d'une formulation à cette région, l'utilisateur doit choisir entre les formulations proposées (Tableau 2). Le choix effectué par l'utilisateur devrait être **validé** par rapport au domaine de validité de la formulation choisie, en liaison avec les éléments de choix antérieurement spécifiés par l'utilisateur.

La simple invalidation du choix d'un utilisateur signifie qu'au moins un des éléments du modèle ne vérifie plus son domaine de validité. La mise en œuvre des domaines de validité permettrait à cette occasion de montrer à l'utilisateur l'élément invalide ainsi que son domaine de validité. Cela faciliterait le **diagnostic** de l'erreur par l'utilisateur, ceci afin d'y remédier plus efficacement.

De manière générale, les domaines de validité ne sont pas exploités au sein du logiciel, ou de façon trop simpliste. En effet, au moment d'affecter une formulation à la région, l'utilisateur en a choisi

une parmi celles présentées par le logiciel. Or, présenter à l'utilisateur une liste exhaustive des choix possibles, c'est lui permettre de commettre une erreur. Le logiciel devrait mettre en œuvre les domaines de validité des formulations afin de ne proposer que des choix valides en fonction de l'état du modèle.

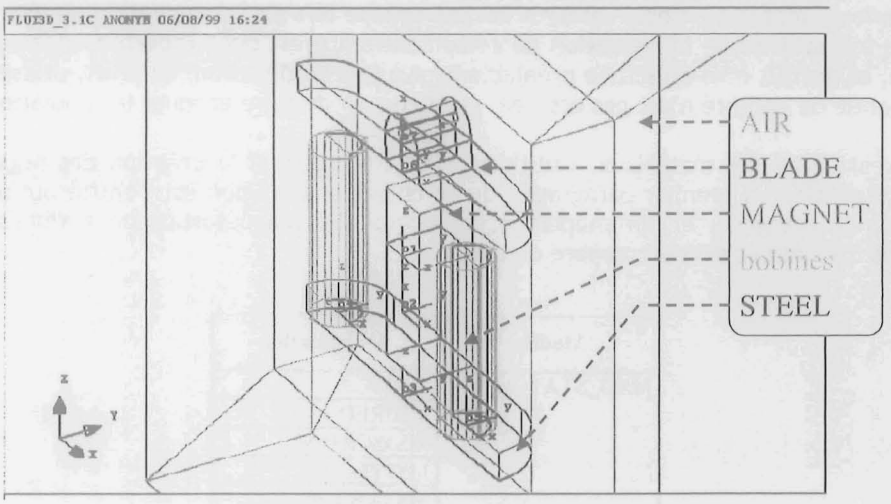


Figure 7 : Le contacteur modélisé

Là encore, le test des domaines de validité des formulations permettrait de sélectionner le sous-ensemble des formulations valides, compte tenu des choix faits antérieurement par l'utilisateur. Le menu se modifierait alors dynamiquement pour ne proposer que les choix validés.

La mise en œuvre du concept de domaine de validité dans un logiciel de simulation numérique lors de la modélisation d'un dispositif peut donc permettre d'apporter une aide à l'utilisateur avec une efficacité croissante. Cet aide pourrait prendre la forme de :

- la validation d'un choix lors de la conception d'un modèle
- l'explication des causes d'une invalidité
- la proposition dynamique des choix valides.

1.3 Modélisation des régimes transitoires

Plaçons-nous maintenant dans le cadre des simulations des régimes transitoires, en prenant pour exemple le chauffage par induction d'une billette d'acier [3.1.14]. La billette est disposée au centre d'un inducteur. Le passage d'un courant alternatif au sein de l'inducteur entraîne l'apparition d'un champ magnétique variable. Sous l'influence des variations du champ, des courants induits se développent dans la billette, entraînant son échauffement par effet Joule.

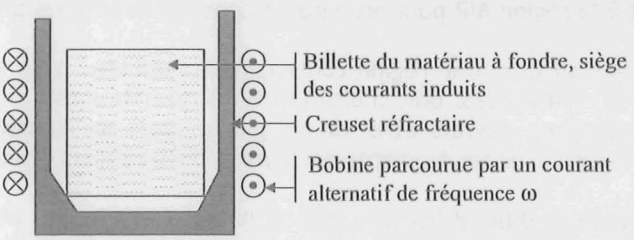
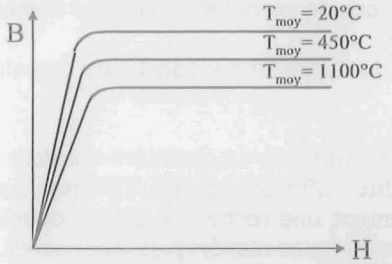


Figure 8 : Le four à induction

La région à laquelle nous allons nous intéresser est celle de la billette d'acier. Elle est le siège de phénomènes électromagnétiques (courants induits) aussi bien que thermiques (pertes Joule).

a- Modification du modèle de matériau

La région modélisant la billette est associée à un modèle de matériau magnétique où la perméabilité relative μ_R est non-linéaire. Elle est modélisée par une courbe dépendant de la température moyenne au sein du matériau (Figure 9).



$\mu(T_{moy}) = B(H,T_{moy})$ pour $T_{moy} < 1100°C$

Figure 9 : Modèle non-linéaire $\mu(T_{moy})$

Ce modèle de matériau est valide pour une température moyenne T_{moy} inférieure à la température de fusion T_F du matériau, soit 1100°C. Au-delà, la phase du matériau est modifiée :il devient liquide. Or ce type de milieux est caractérisé non seulement par sa perméabilité relative μ_R, mais également par de nouvelles propriétés physiques telles que sa viscosité ρ. La prise en compte du passage de la température de fusion va donc nécessiter la modification du modèle de matériau utilisé.

b- Modification du modèle numérique

La répartition des courants induits dans le dispositif dépend du champ de température. Notamment, en deçà du point de Curie T_C, les courants induits [3.1.13] se développent dans l'épaisseur de peau δ du matériau, dont la formule est donnée par :

$$\delta = \sqrt{\frac{2}{\omega\mu\sigma}}$$

où ω est la fréquence du courant dans l'inducteur, μ est la perméabilité relative et σ la conductivité électrique du matériau. Les courants induits se développent alors à proximité de la surface de la billette, et on peut utiliser pour les calculer une formulation en impédances de surfaces.

Au delà du point de Curie, les courants induits se développent dans l'épaisseur de la billette. Dans ces conditions , ils ne sont plus modélisables par des impédances de surfaces. Il faut alors utiliser un autre modèle numérique, volumique, tel que la formulation Tϕ par exemple.

c- Modification de la géométrie

Dans le creuset, la température de la billette va dépasser sa température de fusion T_F. A partir de ce moment, la modélisation de la billette avec une seule région ne sera plus possible. Il faudra prendre en compte les phases solide et liquide du matériau.

La détection de cet événement est à la charge de l'utilisateur, qui doit alors interrompre la simulation, et mettre en œuvre un nouveau modèle. Celui-ci sera composé d'au moins deux régions associées respectivement à chacune des deux phases. La modification de la répartition de la température au sein de la billette va entraîner la modification géométrique des deux régions.

d- Discontinuité des modèles

Comme on le voit dans l'exemple du creuset à induction, les états rencontrés par un dispositif en régime transitoire au cours d'une simulation ne sont pas modélisables par un seul et unique modèle. Ceci est lié au fait qu'un modèle du dispositif est composé de régions associant des méthodes numériques, des modèles de matériaux et des supports géométriques, chacun ayant son domaine de validité propre. La conjonction de ces derniers forme alors le domaine de validité global du modèle.

A un instant donné, la simulation du dispositif peut sortir du domaine de validité du modèle mise en œuvre.

Face à ce problème, il existe deux solutions. La première consiste à établir un modèle de dispositif ayant un large domaine de validité, afin de pouvoir l'utiliser pour l'ensemble de la simulation. Cette solution nécessite généralement une recherche et des développements longs et coûteux, car ces modèles sont fondés sur des techniques numériques innovantes.

Au sein de notre équipe, ils sont généralement l'aboutissement de travaux de thèse. Leur coût peut donc être évalué au minimum à trois ingénieurs-an. De plus, l'utilisation d'un modèle ayant un domaine de validité large se paie au prix de temps de calcul souvent plus long.

Le sens de l'histoire va majoritairement dans le sens opposé à cette démarche, c'est à dire vers le fractionnement des domaines de validité. Les modèles sont développés pour des cas d'utilisation très spécifiques. Ils sont par la suite regroupés au sein d'automates conçus « sur mesure » pour traiter des problèmes types parfaitement identifiés, et ceci de manière performante.

Malheureusement, tous les problèmes types possibles ne sont pas traités au sein d'un logiciel. Or, dans certains cas, on sait construire un modèle du dispositif pour chaque état particulier potentiellement rencontré au cours de la simulation. Dans ce cas, le tâche de l'utilisateur est de détecter lorsqu'un modèle est devenu invalide, d'arrêter alors la simulation. Il pourra relancer cette dernière en ayant préalablement conçu un nouveau modèle valide pour le dispositif. L'utilisateur est donc amené à **gérer la discontinuité des modèles.**

Si l'on reprend l'exemple du chauffage par induction, au début de la simulation l'utilisateur va modéliser la billette par une unique région associée à une formulation en impédances de surfaces afin de calculer les courants induits dans l'épaisseur de peau du matériau.

Dès que le matériau atteint la température de Curie, l'utilisateur doit modifier la formulation associée à la billette. La prise en compte de courants induits dans l'ensemble du volume de la billette va nécessiter l'utilisation de la formulation Tϕ par exemple.

Formulations et Matériaux	Région n°1 : • Impédances de surfaces • μ_R non linéaire	Région n°1 : • Tϕ • μ_R non linéaire	Région n°1 « solide » : • Tϕ • μ_R non linéaire Région n°2 « liquide » : • Tϕ + convection • μ_R =1, ρ
Domaine de validité	$T_{moy} < T_C$	$T_{moy} > T_C$ $T_{max} < T_F$	$T_{min} < T_F < T_{max}$

Tableau 3 : Trois modélisations d'un même dispositif

Enfin, dès que le matériau atteint son point de fusion, il faut complètement revoir le modèle du dispositif. Par rapport au modèle précédent, il apparaît une nouvelle région, associée à la phase liquide du matériau. Celui-ci sera alors modélisé par sa perméabilité relative μ_R, sa viscosité ρ... La formulation associée à cette région sera un couplage magnétohydrodynamique afin de prendre en compte le phénomène de convection. L'apparition de cette deuxième région va impliquer la modification de leur support géométrique au cours de la simulation.

Si on suppose que les trois modèles que nous venons de décrire sont disponibles, ainsi que leurs domaines de validité respectifs, alors la mise en œuvre de ces derniers permet de déterminer à chaque instant de la simulation quel modèle valide doit être utilisé.

1.4 Conclusions

Au travers des quelques exemples que nous venons de présenter, nous avons illustré de quelle manière la mise en œuvre du concept de domaine de validité peut contribuer à résoudre un certains nombre de problèmes couramment rencontrés par les utilisateurs de logiciels de simulations numériques.

Dans le cadre des simulations des régimes permanents, la méconnaissance des domaines de validité autorise l'utilisateur à concevoir des modèles erronés. La solution calculée par le solveur est alors nécessairement fausse. Concernant les simulations des régimes transitoires, l'erreur ne survient pas uniquement au moment de la modélisation du dispositif, mais également au cours de la simulation. Le modèle utilisé ne représente plus alors l'état dans lequel se trouve le dispositif. A partir de ce constat, nous voulons développer les outils qui facilitent la mise en œuvre des domaines de validité dans les logiciels de simulation afin de pallier à ces carences.

1.4.1 Cahier des charges

La mise en œuvre des domaines de validité va se traduire par quatre fonctionnalités : la définition et l'évaluation de la validité, l'explication, le diagnostic et le pilotage. Nous allons maintenant détailler chacune d'entre elles.

a- Description d'un domaine de validité et son évaluation

Les domaines de validité, qui sont un ensemble de contraintes, forment l'aspect contractuel d'une application. Les outils que nous souhaitons développer doivent permettre de décrire et d'évaluer ces domaines de validité afin d'assurer que le modèle conçu par l'utilisateur est conforme aux spécifications contractuelles.

Le modèle créé par l'utilisateur est stocké dans une base de données. Valider le modèle, c'est vérifier que la base de données satisfait l'ensemble des contraintes de l'application. Dans le cas des régimes permanents, respecter les domaines de validité permet de garantir à l'utilisateur le bon déroulement de la résolution d'une part et la validité des résultats obtenus d'autre part.

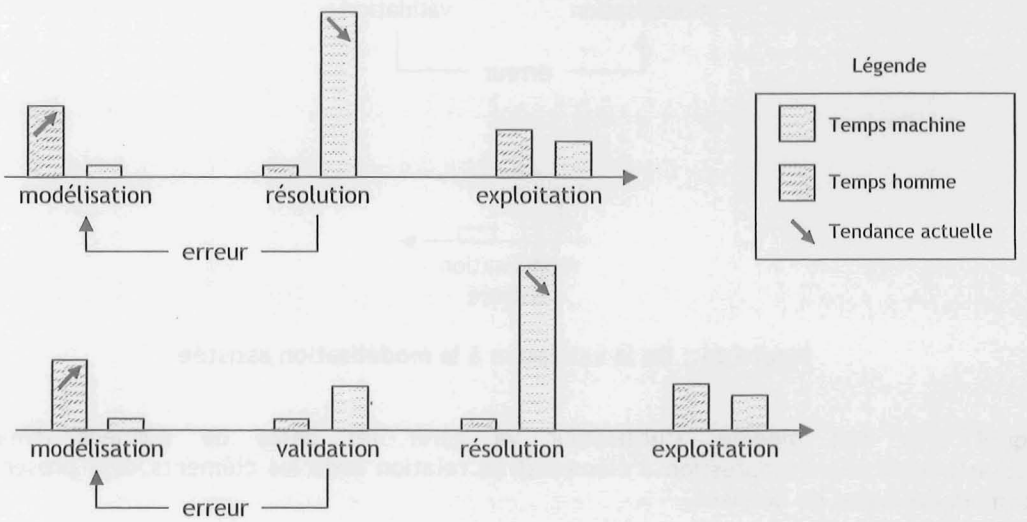


Figure 10 : Introduction d'une étape de validation au cours de la simulation

L'intérêt de valider le modèle avant la résolution est évident si le coût de cette opération est inférieur au coût d'une résolution (Figure 10). En effet, la réussite ou l'échec d'une résolution peut être considéré comme un test de validité du modèle. Or, le coût en temps d'une résolution peut être important, allant généralement de quelques minutes à la journée.

b- Diagnostic

Valider un modèle n'est pas suffisant. Comme nous l'avons fait remarquer, un modèle est généralement complexe. Son invalidation, sans autre explication, oblige l'utilisateur à retravailler ce modèle « en aveugle ».

L'analyse du modèle par l'utilisateur va consister à déterminer quel(s) élément(s) ne vérifie(nt) pas leur domaine de validité. Or, un test algorithmique dans le cas d'une programmation usuelle a nécessairement été implanté pour valider le modèle. Il est par conséquent regrettable de ne pas en faire profiter au maximum l'utilisateur pour le soulager de ce travail fastidieux.

Il est à noter que ce travail est d'autant plus fastidieux pour l'utilisateur que les domaines de validité se retrouvent généralement éparpillés dans la documentation, et souvent de manière incomplète ou implicite. Cette démarche d'analyse d'un modèle devient une réelle démarche d'expertise.

Comme un expert, nos outils doivent pouvoir diagnostiquer l'erreur commise par l'utilisateur, en exhibant l'élément mis en cause et la contrainte qu'il viole. Cette précision du diagnostic permet à l'utilisateur de gagner un temps précieux lors de la correction son modèle.

c- Proposition

Comme nous l'avons vu au travers des exemples du paragraphe précédent, il est possible d'aller au delà d'un automate pour la validation d'un modèle et le diagnostic d'erreur.

La tâche de modélisation consiste souvent pour l'utilisateur à renseigner une ou plusieurs bases de données contenant les informations propres à son modèle. Certaines bases de données sont statiques (les formulations disponibles par exemple), d'autres sont dynamiques (les matériaux, la géométrie, les régions, les sources).

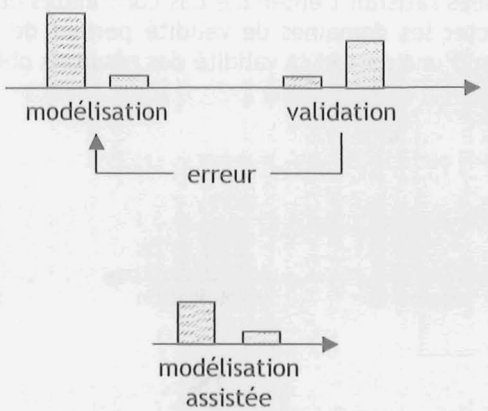


Figure 11 : De la validation à la modélisation assistée

Lorsqu'il crée son modèle, l'utilisateur va gérer les bases de données dynamiques (création/modification/suppression d'éléments) en relation avec les éléments déjà présents dans l'ensemble des bases de données.

Or le domaine de validité d'une formulation peut-être incompatible avec certaines données, et réciproquement. Pouvoir formuler une proposition à partir des domaines de validité va nécessiter de les **inverser**, c'est à dire trouver tous les éléments qui appartiennent aux domaines de validité, en fonction des choix déjà effectués, et limiter dynamiquement les choix futurs aux seuls éléments valides.

Assurer une proposition dynamique de choix valides au cours de la modélisation du dispositif nécessite d'intégrer l'expertise au processus même de la modélisation (Figure 11). On peut alors parler de modélisation assistée.

d- Pilotage

Dans le cas des régimes permanents, nous avons défini a priori les fonctionnalités que les outils que nous souhaitons développer devait implanter afin d'assurer la bonne tenue de la résolution, et de garantir la validité des résultats. Elles sont aux nombres de trois :

- la validation
- l'explication
- la proposition

Concernant les simulations en régime transitoire, nous savons seulement que les domaines de validité permettent de :

- déterminer lorsqu'un modèle est devenu obsolète
- choisir un modèle valide pour représenter l'état actuel du dispositif.

Le pilotage d'une simulation numérique va donc nécessairement utiliser une description des modèles des états potentiellement rencontrés par le dispositif au cours de la simulation ainsi que leurs domaines de validité respectifs.

1.4.2 Critères

Les outils que nous allons concevoir doivent présenter les fonctionnalités citées précédemment. Elles ne sont pas suffisantes pour former un cahier des charges complet. Il faut également spécifier le cadre d'utilisation de cet outil.

Nous avons dit que notre objectif est d'intégrer l'exploitation des domaines de validité au processus courant de la simulation. Or, parler d'intégration en matière de logiciel, c'est nécessairement faire appel aux techniques du **génie logiciel objet**. C'est en effet, à l'heure actuelle, la technologie la plus répandue pour la gestion du cycle de vie global d'un logiciel, depuis ses spécifications jusqu'à son implantation informatique.

Nous devons nous efforcer, dans la mesure du possible, de concevoir un outil sous la forme d'un **composant logiciel réutilisable**. Les développeurs d'applications pourront alors intégrer le composant dans leurs applications, sans remettre en cause les structures propres de ces dernières, afin de bénéficier des fonctionnalités déjà présentées.

2 Solutions techniques

Notre objectif est donc de concevoir un composant logiciel qui, se fondant sur l'expertise des concepteurs de l'application, sera en charge de guider l'utilisateur lors de la phase de modélisation de son dispositif, puis d'assurer la gestion dynamique des modèles de la simulation. L'objet du présent paragraphe est de justifier les choix technologiques que nous avons effectués afin d'implanter ce composant logiciel.

2.1 Systèmes experts et Prolog

2.1.1 Les systèmes experts

Notre but est de mettre en œuvre le concept de domaines de validité au sein d'une simulation numérique. Ceux-ci expriment une partie jusqu'ici non exploitée de la connaissance des concepteurs de l'application.

Nous avons vu que cette connaissance n'est pas de nature algorithmique. En fait, elle décrit les domaines de validité des modèles implantés. C'est une connaissance par nature experte et faiblement structurée.

Notre objectif est de formaliser cette connaissance, puis de l'exploiter au cours du processus de simulation. Or, il existe une technologie informatique appropriée pour la manipulation de ce type de connaissances : le **système expert** [2.1]. En effet, un système expert se compose d'une base de connaissance faiblement structurée et d'un moteur d'inférences [2.1.2.1]. Ce dernier manipule la base de connaissance afin de réaliser une expertise.

Fondées sur le concept de domaine de validité, nous avons développé quatre fonctionnalités qui sont la validité, le diagnostic, la proposition et la gestion des discontinuités de modèles. Les trois premières sont caractéristiques intrinsèques d'un système expert.

On distingue généralement trois catégories de systèmes experts, d'ordre zéro, un ou deux. Les systèmes experts d'ordre zéro sont utilisés pour asserter des propositions de l'utilisateur. En d'autre terme, ils prouvent ou infirment une assertion. Cela correspond à la fonctionnalité de validation.

Les systèmes experts d'ordre un déterminent l'ensemble des solutions (faits) qui vérifient une assertion comprenant des variables (requête). Ce niveau d'expertise convient à la fonctionnalité de proposition.

Enfin les systèmes experts d'ordre deux modifient dynamiquement leur raisonnement au cours de l'évaluation d'une requête. Cet aspect des systèmes experts ne sera pas abordé dans ce travail.

Un point commun à tout système expert digne de ce nom est la conservation de la trace des inférences (raisonnement) réalisées, afin d'expliquer comment il est arrivé à la conclusion qu'il propose. Un système expert est donc avant tout un outil de diagnostic.

La base de connaissance d'un système est partagée entre les faits et les règles. On retrouve cette distinction entre les éléments constituant un modèle particulier de dispositif et leurs domaines de validité respectifs, dont la conjonction forme le domaine de validité global du modèle. Cette similitude des **dualités** faits/règles et modèle/domaines de validité est une bonne indication de l'adaptabilité d'une technologie de type système expert pour traiter notre problématique. La base de faits sera constituée des éléments du modèle, et les règles seront les domaines de validité applicables à ces éléments.

Enfin, un domaine de validité s'exprime généralement sous la forme d'une conjonction de contraintes. Or, les systèmes experts sont dédiés à la manipulation de contraintes, mises sous la forme de prédicats (logique du premier ordre).

Notons enfin que divers travaux fondés sur la technologie des systèmes experts ont été réalisés au sein de la communauté du génie électrique [2.1.1]. Dans ce cadre, la technologie des systèmes experts a été principalement utilisée pour concevoir des outils de conception de dispositifs électromagnétiques. Notre objetif est de fondre dans un unique environnement informatique ce type d'application de conception experte avec les outils de simulation.

2.1.2 Prolog

Le langage **Prolog** [2.2] est un langage iso-normalisé [2.2.1][2.2.2] pour l'écriture de prédicats. Son usage est très largement répandu dans la communauté des systèmes experts pour leur implantation. C'est donc naturellement vers lui que nous nous tournerons. De nombreux interpréteurs existent et sont disponibles, en version communautaire (GnuProlog) ou commerciale (PrologIII™ et PrologIV™).

Enfin, notre domaine d'activité est la simulation numérique. Ceci implique que nécessairement certaines contraintes des domaines de validité vont être de nature numérique. On peut reprendre les exemples déjà cités : la contrainte CFL ($|a|\Delta t/\Delta x < 1$) ou la contrainte associée au nombre de Reynolds ($V_{moy}*L/\nu < 2400$).

Dans le cas de la contrainte CFL, la modélisation d'un problème nécessite la donnée du coefficient 'a' ainsi que des pas d'espace Δx et de temps Δt. Vérifier cette contrainte lorsque les trois coefficients sont déterminés est aisé.

Globalement, l'ensemble des contraintes associées à un modèle peuvent être vérifiées algorithmiquement dès l'instant que ce dernier est complet. Or notre objectif est plus ambitieux : nous ne voulons pas nous contenter d'invalider un modèle complet, mais proposer dynamiquement des choix valides à l'utilisateur au cours de l'étape de modélisation.

Du point de vue d'une contrainte discrète, il s'agit alors d'élaguer de la liste des choix possibles ceux qui sont invalides. Pour une contrainte de nature numérique, c'est plus compliqué. Il faut proposer un intervalle de valeurs possibles pour chacun des paramètres de la contrainte.

La Figure 12 ci-dessous présente l'évolution de la contrainte CFL et des intervalles de valeurs pour le coefficient 'a' et les pas de temps Δt et d'espace Δx en fonction des choix successifs faits par l'utilisateur :

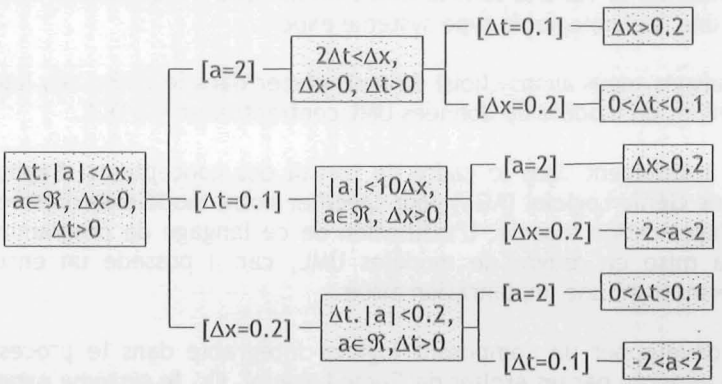


Figure 12 : Evolution de la contrainte CFL en fonction des choix

On cherche donc à **inverser les contraintes numériques**. Cela nécessite l'utilisation de la programmation logique par contraintes (PLC).

Pour l'ensemble de ces raisons, nous avons choisi de mettre en œuvre les domaines de validité au travers d'un système expert Prolog implantant également la programmation logique par contraintes.

Au début de ce travail, peu de logiciels offraient toutes ses caractéristiques dans un unique environnement. C'est pour cela que nous avons choisi le logiciel **PrologIV™** de la société PrologIA®, de Marseille-Luminy. Il est codé en langage C.

2.2 *Unified Modeling Language – Object Constraint Language*

Le système expert est donc la technologie que nous avons choisie pour mettre en œuvre les domaines de validité. Il est le cœur de(s) outil(s) que nous allons par la suite développer. Cependant, il ne faut pas perdre de vue le contexte d'utilisation que nous nous sommes fixé.

Nous souhaitons développer, autant que faire ce peut, des outils sous la forme de composants logiciels réutilisables. Ainsi, ils pourront être intégrés à des applications lors de leur conception. Ces applications seront par conséquent issues du génie logiciel objet.

Concernant les logiciels de simulations numériques orientés objet, le modèle d'un dispositif particulier est composé d'un ensemble d'instances de classes. Ces dernières appartiennent au modèle de données de l'application.

Le modèle de données est développé par les concepteurs de l'application. En génie logiciel objet, le langage le plus utilisé pour décrire un modèle de données objet est **Unified Modeling Language** (UML) [1.2]. C'est d'ailleurs un langage normalisé par l'Object Management Group (OMG), consortium international pour le développement et la normalisation des technologies objets.

C'est au niveau du modèle de données que les contraintes propres à l'application sont exprimées par les concepteurs. Elles composent l'aspect contractuel de l'application. En d'autres termes, c'est au niveau du modèle de données que sont exprimés les domaines de validités.

Jusqu'à présent, les modèles de données objet UML était faiblement contractualisés : aucun formalisme ne permettait de rendre compte des contraintes associées au modèle de données. Or, nous avons vu que l'utilisation des domaines de validité nécessitait leur formalisation.

Ce n'est que tout récemment, en août 2001, que l'OMG a adopté un nouveau langage standard : **Object Constraint Language** (OCL) [1.3]. OCL est un langage formalisé permettant de spécifier les contraintes qui s'exercent sur un modèle de données UML particulier.

L'écriture des domaines de validité sous la forme normalisée d'expressions OCL doit permettre leur manipulation par une technologie de type système expert.

Pour toutes ces raisons, nous avons choisi de nous placer dans le cadre des logiciels de simulations numériques possédant un modèle de données UML contractualisé via OCL.

Pour nous placer totalement dans le cadre de travail des concepteurs d'applications, nous avons utilisé un Atelier de Génie Logiciel (AGL) pour spécifier notre modèle de données UML, et générer le code **Java™** de l'application associé. L'utilisation de ce langage de programmation objet est très répandue pour la mise en œuvre de modèles UML, car il possède un ensemble de structures syntaxiques en permettant une implantation aisée.

Nous souhaitons développer un composant logiciel intégrable dans le processus de modélisation d'une application générée par un Atelier de Génie Logiciel. Or, le système expert que nous utilisons est codé en langage C. Par conséquent, pour assurer l'intégration du système dans l'application, nous serons obligé d'**encapsuler** le premier, et d'assurer le couplage entre ces langages C et Java. Pour cela, nous utiliserons la technologie Java Native Interface (**JNI**).

3 Conclusions

Comme nous l'avons vu, la technologie des systèmes experts n'est pas nouvelle au sein de la communauté du génie électrique [2.1.1]. Cependant, elle est restée principalement dédiée au développement d'applications mettant en œuvre une expertise dans le cadre de la conception de dispositifs électrotechniques. Ces applications sont conçues indépendamment de tout environnement de simulation. Ceci s'explique en partie par le fait que les technologies d'implantation de ces derniers ne sont pas les mêmes que celles utilisées pour les systèmes experts.

Or, comme nous l'avons montré sur quelques exemples, il existe des carences au sein des logiciels de simulations au niveau de la modélisation des dispositifs, ainsi que des difficultés intrinsèquement

liées aux régimes transitoires. Ces problèmes sont généralement palliés par une intervention humaine. Notre objectif est d'introduire et d'exploiter l'expertise en tant qu'élément à part entière au sein des environnements de simulation.

Pour cela, les outils méthodologiques d'une part, tels qu'UML et OCL, et technologiques d'autre part, tels que les systèmes experts et Prolog, sont matures. Dans le chapitre suivant, nous allons préciser leur nature, ainsi que leurs limitations. Nous montrerons alors que le rapprochement de ces outils dans un unique environnement est la source de synergies intéressantes et originales.

Représentation et Exploitation des Contraintes

Sommaire

L'objectif du présent chapitre est didactique. Les deux premiers paragraphes sont ainsi consacrés à l'introduction aux technologies des systèmes experts et à la modélisation objet, plus précisément les langages UML et OCL. La vocation du dernier paragraphe est de faire le point sur les limitation respectives de ces deux technologies, et d'analyser la synergie méthodologique apportée par un couplage.

1 Systèmes experts et Prolog

Les systèmes experts [2.1] sont nés en 1965, dans une équipe de l'Université de Stanford qui travaillait sur un problème auquel on ne connaît aucune solution algorithmique. Pour résoudre ce problème, l'unique solution était de faire constamment appel à des experts. L'équipe eut alors l'idée d'implanter informatiquement cette expertise [2.1.2.1].

1.1 Principe de fonctionnement des sytèmes experts

Le cadre algorithmique est tout à fait satisfaisant lorsque l'on sait identifier le problème-type auquel se rattache un problème donné, et qu'il existe une théorie centrale prévoyant le comportement de cette catégorie de problèmes.

L'usage de l'algorithmique est donc parfaitement justifié dans les domaines tels que les mathématiques, la physique, la mécanique... Dans ce cas, il suffit d'appliquer la méthode de résolution éprouvée pour le problème-type. Cependant, il arrive que la nature du problème ne soit pas aisément identifiable. Il faut alors faire appel à une réelle expertise pour le poser, et parvenir à le résoudre.

Le concept de système expert est fondé sur le postulat qu'une expertise se traduit par un ensemble de règles et de faits explicites qui reflètent par leur enchaînement le raisonnement des experts eux-mêmes.

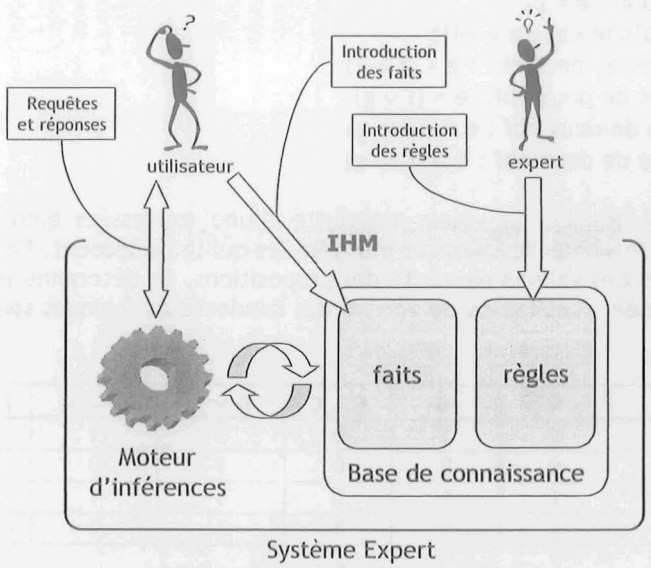


Figure 1 : Architecture d'un système expert

La connaissance est donc décrite sous la forme de faits et de règles, stockés dans la base de connaissance (Figure 1). Celle-ci est renseignée par les experts du domaine pour les règles et par les utilisateurs pour les faits, via une interface (IHM) qui formalise leur connaissance avant de la stocker.

Le moteur d'inférences enchaîne les règles déductives afin de parvenir à répondre aux requêtes de l'utilisateur.

1.2 Prolog

Le langage Prolog [2.2] a été conçu par Alain Colmerauer [2.2.5] au début des années soixante-dix. Prolog est un acronyme pour PROgrammation LOGique. La consécration internationale de ce langage date de 1982, date à laquelle il fût choisi comme support pour les ordinateurs japonais de cinquième génération, dans le cadre de la recherche en intelligence artificielle. Avant de décrire la syntaxe du langage Prolog lui-même, nous allons introduire les concepts de calcul des prédicats dont il est issu.

1.2.1 Des propositions aux prédicats

A priori, on peut considérer la programmation logique comme une tentative d'utiliser la logique comme langage de programmation. Tout commence donc avec le calcul propositionnel.

a- Calcul propositionnel

Une proposition, notée p_i, est par exemple une phrase simple dont on sait déterminer la valeur de vérité dans un contexte donné : en vous penchant à la fenêtre, vous pouvez déterminer si la proposition « il pleut » est vraie ou fausse. A toute proposition est associée sa valeur de vérité : 1 (vrai) ou 0 (faux). La donnée de la valeur de vérité pour l'ensemble des propositions est une **valuation.**

On construit une expression bien formée (ebf) à partir de l'ensemble des propositions et des connecteurs logiques. Une expression e est une expression bien formée si et seulement si e est :

- une proposition : $e = p_1$
- la négation d'une ebf : $e = \neg(f)$
- la conjonction de deux ebf : $e = (f \wedge g)$
- la disjonction de deux ebf : $e = (f \vee g)$
- l'implication de deux ebf : $e = (f \Rightarrow g)$
- l'équivalence de deux ebf : $e = (f \Leftrightarrow g)$

Pour une valuation V donnée, la valeur de vérité d'une expression bien formée complexe est fonction de la valeur de vérité des énoncés plus simples qui la composent. En envisageant toutes les combinaisons possibles des valeurs de vérité des propositions, on détermine les valeurs de vérité de l'expression bien formée. Les tables de vérités des connecteurs logiques sont récapitulées dans le Tableau 1 ci-dssous :

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
0	0	1	0	0	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
1	1	0	1	1	1	1

Tableau 1 : Tables de vérités

On sépare les expressions bien formées en trois catégories :

- les **tautologies**, qui sont les expressions vraies pour toute valuation (ex : $p \vee \neg p$)
- les **contradictions**, qui sont les expressions fausses pour toute valuation (ex : $p \wedge \neg p$)
- les expressions **contingentes**, dont la valeur de vérité dépend de la valuation choisie (ex : $p \wedge q$).

La méthode des tables de vérité permet de déterminer de manière automatique et mécanique le type de toute expression bien formée, en calculant sa valeur de vérité pour toutes les valuations possibles des propositions qui la forment. Comme le prouve respectivement Tableau 2 et Tableau 3, l'énoncé « si p implique q, et p, alors q » est une tautologie et l'énoncé : « si (non p) implique q, et p, alors (non q) » est contingente.

p	q	$p \Rightarrow q$	$(p \Rightarrow q) \wedge p$	$((p \Rightarrow q) \wedge p) \Rightarrow q$
1	1	1	1	1
1	0	0	0	1
0	1	1	0	1
0	0	1	0	1

Tableau 2 : Calcul d'une tautologie

p	q	$\neg p$	$\neg q$	$\neg p \Rightarrow q$	$(\neg p \Rightarrow q) \wedge p$	$((\neg p \Rightarrow q) \wedge p) \Rightarrow \neg q$
1	1	0	0	1	1	0
1	0	0	1	1	1	1
0	1	1	0	1	0	1
0	0	1	1	0	0	1

Tableau 3 : Calcul d'une expression contingente

Le calcul propositionnel permet d'établir un **système déductif** complet, basé sur la notion de **clause de Hörn**. Une clause de Hörn est une expression bien formée ayant la forme suivante :

$$(CH) : p_1 \wedge p_2 \wedge \ \ldots \ \wedge p_k \Rightarrow q$$

où les p_i sont des propositions appelées hypothèses ou prémisses, et q est la proposition appelée conclusion. La sémantique d'une clause de Hörn est très simple :

Si les prémisses p_1, p_2, ..., p_k sont vraies, alors la conclusion q l'est aussi.

Le système déductif fondé sur les clauses de Hörn se compose :

- de faits qui sont des propositions supposées vraies (clauses de Hörn sans conclusion : $p_1 \wedge p_2 \wedge \ \ldots \ \wedge p_k$)
- de règles déductives (clauses de Hörn avec prémisses et conclusion),
- et d'une requête c'est à dire une proposition à démontrer (clause de Hörn sans prémisse : $\Rightarrow q$).

Grâce à l'action des règles sur les faits, on construit d'autres faits jusqu'à l'obtention de la proposition à démontrer. Ce mode de démonstration est appelé « modus ponens » (Figure 2).

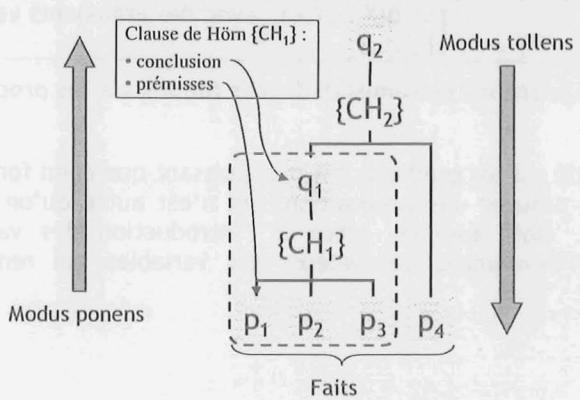


Figure 2 : Modus ponens, modus tollens

L'autre mode de démonstration est le « modus tollens ». Il est fondé sur la réversibilité des clauses de Hörn. En effet, si la conclusion q d'une règle est supposée vraie, alors les prémisses doivent l'être également. En partant de la proposition à démontrer (appelé but), et en chaînant les règles de leur conclusion vers leurs prémisses, on doit trouver un chemin vers les faits.

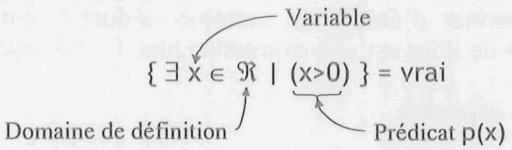
On retrouve souvent dans la littérature dédiée à Prolog les termes de chaînage avant et de chaînage arrière. Ils correspondent respectivement au modus ponens et au modus tollens. Dans le premier cas, la stratégie de démonstration est guidée par les faits, alors que dans le second cas, elle est guidée par le but.

b- Calcul des prédicats

Le calcul propositionnel est le point de départ des systèmes déductifs. Cependant ce cadre possède une lacune importante : l'absence des quantificateurs $\exists$ et $\forall$. Ces deux structures logiques induisent les notions :

- d'ensemble d'objets,
- de **variable** parcourant l'ensemble des objets (**ensemble de définition** de la variable)
- de propositions portant sur ces variables (**fonctions propositionnelles** ou **prédicats**).

Par exemple la phrase suivante : « il existe x appartenant à l'ensemble des nombres réels $\Re$ tel que x soit positif », est traduite par l'expression :



On appelle **unification** le processus par lequel une variable prend une valeur particulière dans son ensemble de définition. La valeur de vérité d'un prédicat dépend donc de l'unification de ses variables avec des valeurs particulières appartenant à leur domaine de définition. Dans l'exemple ci-dessus, quelques valeurs de vérité particulières du prédicat P sont : P(0) = faux, P(-1,525) = faux, P(43) = vrai,...

Dans le cadre des prédicats, le système déductif fondé sur les clauses de Hörn conserve le tryptique faits/règles/requêtes. Cependant, les notions de variables et de prédicats entraînent de légères modifications, liées à la nature unifiée ou non des variables (cf. Tableau 4).

	Propositions	Prédicats
Faits	$p_1 \wedge p_2 \wedge \ ... \ \wedge p_k$	$p_1(a) \wedge p_1(b) \wedge \ ... \ \wedge p_k(a,c)$, avec {a;b;...} des valeurs.
Règles	$p_1 \wedge p_2 \wedge \ ... \ \wedge p_k \Rightarrow q$	$p_1(X) \wedge ... \wedge p_k(Y,Z) \Rightarrow q(X,...,Y,Z)$, avec {X;...;Y;Z} variables **non unifiées.**
Requêtes	$\Rightarrow q$	$\Rightarrow q(X,...,Y,c)$, avec des arguments variables {X;...;Y} ou connus {c}.

Tableau 4 : Différences entre les systèmes déductifs fondés sur les propositions ou les prédicats

Le système déductif fondé sur les prédicats est plus puissant que celui fondé sur les propositions. Il permet évidemment de prouver une proposition, qui n'est autre qu'un prédicat dont toutes les variables sont unifiées. Mais au-delà, grâce à l'introduction des variables et du processus d'unification, il trouve l'ensemble des valeurs des variables qui rendent vrai le prédicat à démontrer (cf. Figure 3).

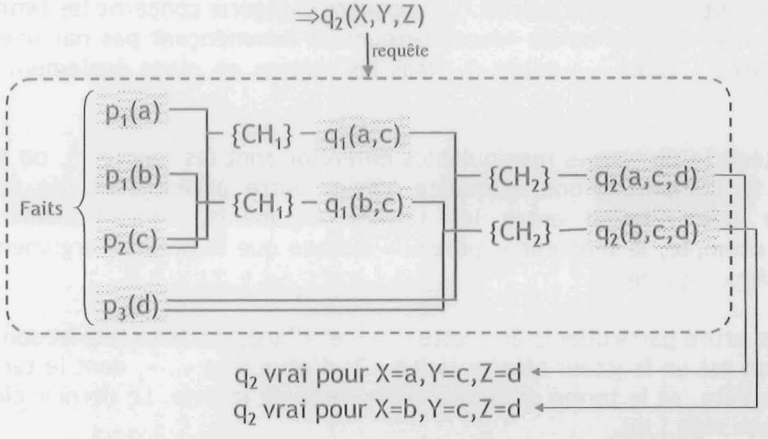


Figure 3 : Système déductif fondé sur les prédicats

1.2.2 Le langage Prolog

Le langage Prolog est un langage normalisé (norme ISO/IEC 13211-1) [2.2.1][2.2.2] permettant d'écrire des systèmes déductifs fondés sur les prédicats. Avant d'aborder l'écriture de programme Prolog et leur résolution, nous détaillerons succinctement la syntaxe du langage.

a- La syntaxe

En Prolog [2.2.3], l'écriture d'une clause de Hörn commence toujours le symbole « ?- », et se termine par un point final. La place de la conclusion et des prémisses sont inversées par rapport à l'écriture de la même clause en logique :

logique : $p_1(X) \wedge p_2(Y) \wedge \ \ldots \wedge p_k(Z) \Rightarrow q(X,Y,\ldots,Z)$

Prolog : ?- q(X,Y,Z) :- $p_1(X)$, $p_2(Y)$, ... , $p_k(Z)$.

étiquette variables — prédicat

tête — corps

Figure 4 : Structure d'une clause Prolog

Du fait de cette inversion, le symbole « :- » est équivalent à une implication à gauche (⇐). La conjonction (∧) de deux prédicats est dénotée par une virgule « , », et la disjonction (∨) par un point-virgule « ; ».

La partie gauche de la clause est appelée **tête**, et la partie droite **corps**. La tête est un **foncteur**, encore appelé **terme composé**. Un foncteur se compose d'une **étiquette**, qui est identificateur (chaîne de caractères), et d'une liste d'arguments. Ceux-ci seront par la suite utilisés dans le corps de la clause (cf. Figure 4). Le nombre d'arguments de la clause est son **arité**.

On désigne généralement une clause par son étiquette suivie de son arité, séparés par le symbole « / ». Dans l'exemple ci-dessus, nous avons la clause « q/3 ».

Dans le langage Prolog, toute chaîne de caractères commençant par une majuscule (ex : « X », « Var »,...) ou par le caractère « _ » (ex : « _1052 ») dénote une **variable**. Une variable est **unifiable** avec l'ensemble des **termes** disponibles.

Les termes se divisent en deux catégories. La première catégorie concerne les termes atomiques, ou **atomes**. Ce sont les nombres ou les identificateurs ne commençant pas par une majuscule ou le caractère « _ » (ex : « 12.63 », « table »). Dans les atomes on place également la liste vide ([]), appelée nil.

La deuxième catégorie de termes manipulables en Prolog sont les foncteurs, ou **arbres**. Rappelons qu'un foncteur se compose d'une **étiquette** suivie, entre parenthèses, de ses arguments. Un foncteur réalise une relation entre les termes arguments, sémantiquement connotée par l'étiquette. Par exemple, le foncteur « père/2 » indique que le premier argument du foncteur est le père du deuxième argument.

Une clause est un arbre particulier d'étiquette « :- » et d'arguments sa tête et son corps : « :-(Tête, Corps) ». Une liste est un foncteur binaire (arité = 2) d'étiquette « . », dont le terme de gauche est un élément de la liste, et le terme de droite est le reste de la liste. Le dernier élément d'une liste est toujours la liste vide : nil.

b- Le processus d'unification

Avant d'aller plus loin, regardons une base de faits réduite contenant trois points, et présentée dans le fichier Prolog suivant :

```
00
01 %% points/3 : identificateur du point, ses coordonnées X et Y
02    point( p1, 0, 0).
03    point( p2, 0, 1).
04    point( p3, 1, 1).
05
```

La ligne commençant par le symbole « %% » est un commentaire. L'interrogation de la base de faits s'effectue au moyen de requêtes. Dans l'exemple commenté[1] suivant, nous demandons à Prolog quels sont les points disponibles dans sa base de faits :

```
< trouver tous les points >
?- point( Id, X, Y).
   Id = p1, X = 0, Y = 0 ;    <- 1ère solution : le point p1
   Id = p2, X = 0, Y = 1 ;    <- 2ème solution : le point p2
   Id = p3, X = 1, Y = 1.     <- 3ème solution : le point p3
```

Cette question est traduite par la requête : « ?- point(Id, X, Y). ». Pour évaluer cette requête, l'interpréteur va parcourir l'ensemble de la base de faits. Dès qu'il trouvera un fait correspondant à la requête, il tentera d'unifier les variables de cette dernière (Id,X,Y) avec les valeurs spécifiées dans le fait (p1,0,0). Le balayage de la base de faits étant exhaustif, ceci implique que toutes les solutions possibles seront trouvées.

La recherche de deux points ayant la même coordonnée X est traduite par la conjonction de deux prédicats, ayant une variable commune X. L'évaluation de cette requête sur notre base de faits donne le résultat suivant :

```
< Trouver les points ayant la même abscisse X >
?- point( Id1, X, _), point( Id2, X, _).  <- le symbole « _ » dénote une variable muette
   Id1 = p1, X = 0, Id2 = p1 ;
   Id1 = p1, X = 0, Id2 = p2 ;
   Id1 = p2, X = 0, Id2 = p1 ;
   Id1 = p2, X = 0, Id2 = p2 ;
   Id1 = p3, X = 1, Id2 = p3.
```

Cet exemple illustre le fonctionnement de l'interpréteur (Tableau 5) :

[1] Les commentaires sont en italiques. Le sens de la requête est préalablement indiqué entre <...>.

- l'évaluation du premier prédicat composant la requête entraîne un premier balayage de la base de faits. Les variables du prédicat (Id1, X) sont unifiées successivement avec chacun des tuples solutions trouvés.
- Pour chaque tuple solution du premier prédicat, l'interpréteur entreprend l'évaluation du deuxième prédicat, en tenant compte des valeurs des variables précédemment unifiées. Le balayage reprend sur l'ensemble de la base de faits afin de déterminer les tuples solutions de ce deuxième prédicat.
- Ce processus se déroule jusqu'à ce que tous les prédicats composant la requête aient été évalués.

Faits	Prédicats évalués		Résultat
	point(Id1, X, _)	point(Id2, X, _)	
point(p1,0,0)	Id1=p1,X=0		
point(p1,0,0)		Id2=p1,X=0	true
point(p2,0,1)		Id2=p2,X=0	true
point(p3,1,1)		Id2=p3,X=1	false
point(p2,0,1)	Id1=p2,X=0		
point(p1,0,0)		Id2=p1,X=0	true
point(p2,0,1)		Id2=p2,X=0	true
point(p3,1,1)		Id2=p3,X=1	false
point(p3,1,1)	Id1=p3,X=1		
point(p1,0,0)		Id2=p1,X=0	false
point(p2,0,1)		Id2=p2,X=0	false
point(p3,1,1)		Id2=p3,X=1	true

Tableau 5 : Balayage de la base de faits

Evidemment, le caractère exhaustif dû au balayage de la base de faits implique que l'évaluation d'une requête comprenant de nombreuses règles, et sur un base de faits de taille conséquente, peut être très coûteuse (Annexe II).

c- Les opérateurs Prolog prédéfinis

Le langage normalisé Prolog comprend un ensemble d'opérateurs prédéfinis, ayant une syntaxe et une sémantique précise, et portant sur les structures de données introduites précédemment. Nous allons rapidement présenter les plus représentatifs.

Un ensemble d'opérateurs a été défini afin de tester le type des données :

Opérateur	Sémantique
« atom/1 »	Teste si l'argument est un identificateur
« atomic/1 »	Teste si l'argument est un identificateur ou un nombre
« compound/1 »	Teste si l'argument est un terme composé
« float/1 »	Teste si l'argument est un flottant
« integer/1 »	Teste si l'argument est un entier
« nonvar/1 »	Teste si l'argument est connu
« number/1 »	Teste si l'argument est un nombre
« rational/1 »	Teste si l'argument est un rationnel
« var/1 »	Teste si l'argument est une variable

Des opérateurs numériques, tels que des fonctions évaluables et des comparaisons arithmétiques, ont été également prédéfinies. Ils sont associés au prédicat « is/2 » qui ordonne leur évaluation.

Opérateur	Sémantique
« is/2 »	Evalue une formule arithmétique
« +/2 », « -/2 », « */2 », « //2 », « **/2 », « sin/1 », « cos/1 », ...	Fonctions évaluables

« </2 », « <=/2 », « =>/2 », « >/2 », « = :=/2 », « =\=/2 »,	Comparaisons arithmétiques

Au sujet de l'évaluation, il est important de noter qu'à l'exception du premier argument du prédicat « is/2 », aucun autre argument des prédicats d'évaluation ne peut être variable. En effet, dans ce cas, l'évaluation du prédicat est impossible, comme le montre l'exemple suivant :

```
?- X is 1 + 2.   <- évaluation de +(1,3) possible
   X = 3.
?- X is Y + 1.   <- évaluation de +(Y,1) impossible car Y est une variable
   false.
```

D'autre part, il est important de distinguer l'opérateur d'unification « =/2 » d'avec l'opérateur d'évaluation « is/2 », comme le montre l'exemple ci-dessous :

```
?- X = 1 + 2.          <- X est unifiée avec le foncteur +(1,2)
   X = 1+2.
?- X = 1 + 2, Z is X.  <- Z est unifiée avec la valeur évaluée de X
   X = 1+2,
   Z = 3.
```

Dans le premier cas, la variable X a été unifiée avec la fonction évaluable « +(1,2) », sans opérer l'évaluation, qui est du ressort de l'opérateur « is/2 ».

Le langage normalisé Iso-Prolog comprend des opérateurs de contrôles, tel que le « ,/2 » pour la conjonction de deux contraintes, que nous avons déjà vu précédemment. Les opérateurs les plus utilisés sont :

Opérateur	Sémantique
« ,/2 »	Conjonction de contraintes
« ;/2 »	Disjonction de deux contraintes
« ->/2 »	Si-alors
« -> ;/3 »	Si-alors-sinon
« !/0 »	cut

Voici quelques exemples qui illustrent l'utilisation de ces opérateurs sur la base de faits contenant trois points introduite précédemment :

```
< Trouver tous les points d'identificateur « p1 » ou d'abscisse X = 0 >
?- point( Id, X,Y), ( Id = p1 ; X = 0 ).
   Id ~p1, X = 0, Y = 0 ;  <- 1ère solution : le point p1 à cause de son nom
   Id ~p1, X = 0, Y = 0 ;  <- 2ère solution : le point p1 à cause de son abscisse
   Id ~p2, X = 0, Y = 1.   <- 3ère solution : le point p2 à cause de son abscisse

< pour tous les points, si son ordonnée est nulle, écrire [...]
  sinon calculer Z = abscisse/ordonnée >
?- point( Id, X, Y), ( Y =:= 0 -> write('division par zéro : ' ; Z is X / Y ).
   division par zéro : Id ~p1, X = 0, Y = 0, Z ~ tree ;  <- la variable Z n'est pas unifié.
   Id ~p2, X = 0, Y = 1, Z = 0 ;
   Division par zéro : Id ~p3, X = 0, Y = 0, Z ~ tree.

< Trouver le premier point d'abcisse X = 0 >
?- point( Id, X, Y), X = 0, !.
   Id ~ p1, X = 0, Y = 0.
```

Prolog dispose d'opérateurs permettant d'analyser, voir de modifier dynamiquement sa base de connaissance :

Opérateur	Sémantique
« functor/3 »	(Dé)compose un terme en son étiquette et son arité
« =../2 »	(Dé)compose un terme en son étiquette et sa liste de fils
« arg/3 »	Sélectionne un argument d'un terme
« asserta/1 »	Asserte une clause en tête du paquet de règles
« assertz/1 »	Asserte une clause en fin du paquet de règles
« retract/1 »	Retire une clause du paquet de règles

Voici quelques requêtes utilisant ces prédicats :

```
< étiquette et arité du foncteur « point/3 » >
?- functor( point(_,_,_), Etiquette, Arité).
   Etiquette = point, Arité = 3.

< Décompose le foncteur point(p2,0,1) >
?- =..( point(p2,0,1), Res).
   Res = [ point, p2, 0, 1].

< Compose le foncteur binaire Res d'étiquette '.' et d'arguments a et [b,c] >
?- =..( Res, [ '.', a, [b, c]]).
   Res = [a,b,c].

< Asserte le point p4 de coordonnées (1,1) en queue du paquet de règles >
?- assertz( point( p4, 1, 1) ).
   true.

< Trouve tous les points tel que X = Y >
?- point( Id, X, Y), X = Y.
   Id ~ p1, X = 0, Y = 0 ;  <- 1ère solution : le point p1
   Id ~ p4, X = 1, Y = 1. <- 2ème solution : le point p4, le point que l'on vient d'asserter

< Retire le point p4 du paquet de règles >
?- retract( ( point( p4, 1, 1) :- true ) ).
   true.

< Trouve tous les points tel que X = Y >
?- point( Id, X, Y), X = Y.
   Id ~ p1, X = 0, Y = 0.   <- il n'y a plus qu'une seule solution : le point p1
```

Enfin, pour terminer avec les prédicats prédéfinis en Prolog, nous citerons le prédicat « findall/3 » qui récupère dans une liste l'ensemble des tuples solutions dans l'ordre rencontrées, lors de l'évaluation du deuxième argument :

```
< Trouve toutes les valeurs Id tel que Id soit l'identificateur d'un point d'abscisse nulle >
?- findall( Id, point( Id,0,_), L).
   L = [ p1, p2].  <- Liste des valeurs prises par Id, solutions de point(Id,0,_)
```

Nous avons disposé en **Annexe II** un exemple de programme Prolog plus détaillé.

1.2.3 Programmation Logique par Contraintes

La programmation logique par contraintes est une généralisation du calcul des prédicats. Les seules structures de données manipulées en programmation logique sont des arbres, ce qui limite le domaine du discours.

En particulier, les solveurs utilisés en programmation logique n'autorisent la manipulation d'expressions arithmétiques que du strict point de vue de fonctions évaluables. Comme nous l'avons

vu plus haut, l'évaluation de l'expression « X is Y+1 » échoue si la valeur de Y est indéterminée (symbole «⊥»), car le solveur ne peut pas calculer la valeur de X (cf. Tableau 6).

L'adjonction d'un nouveau solveur arithmétique à l'interpréteur Prolog a permis d'élargir le domaine du discours de la programmation logique. Les expressions arithmétiques peuvent alors être manipulées comme de véritables contraintes numériques inversibles. L'expression « X = Y+1 » est alors fausse si et seulement si les valeurs de X et de Y sont incompatibles (cf. Tableau 6).

X	Y	X is Y+1	X = Y+1
⊥	⊥	faux	X, Y ∈ ℜ
a	⊥	faux	Y = a-1
⊥	b	X = b+1	X = b+1
a	b	vrai si a = b+1, faux sinon	vrai si a = b+1, faux sinon

Tableau 6 : Evaluation d'une fonction et d'une contrainte

Le choix du solveur logique ou arithmétique pour l'évaluation de l'expression est déterminé par l'usage du prédicat « is/2 » ou « =/2 » lors de l'interprétation de la contrainte. Dans le premier cas, le prédicat « is/2 » est un prédicat prédéfini en Prolog qui évalue une expression en tant que fonction évaluable. Cette dernière est traitée par le solveur logique. Le symbole « =/2 » ne fait pas partie de la norme Prolog. Elle type l'expression comme une contrainte numérique, traitée par un solveur arithmétique spécifique au logiciel PrologIV [2.2.6].

1.3 Bilan

La technologie des systèmes experts a fait ses preuves dans beaucoup de domaines où l'expertise est prépondérante par rapport à l'algorithmique, tel que la médecine, les sciences de la taxonomie, ... On peut notamment cité MYCIN, un système expert médical célèbre, et dont le fonctionnement a été abondamment commenté.

Malgré ce succès, les systèmes experts n'ont jamais véritablement réussi à séduire les développeurs de logiciels de simulations numériques. D'ailleurs, dans l'enseignement classique de l'informatique, ces deux techniques de programmation sont nettement séparées, et destinées à deux usages et deux publics différents.

2 Modélisation Objet

La modélisation orientée objet [1] est un outil pour la conception, le développement et la maintenance de systèmes complexes, tels que par exemple les applications logicielles. Cette méthodologie a rencontré un succès sans cesse croissant. En 1989 était fondé l'Object Modeling Group (OMG), un consortium constitué des principaux acteurs du monde de l'informatique et de l'industrie dans ce domaine

Le but de l'OMG est d'assurer un cadre commun pour le développement de logiciels dans un environnement hétérogène. Les spécifications de l'OMG se retrouvent aujourd'hui dans de très nombreuses applications.

Nous allons introduire successivement les principes fondamentaux de la modélisation objet, puis la principaux éléments syntaxiques des normes Unified Modeling Language (UML) et Object Constraint Language (OCL).

2.1 Les concepts de la modélisation orientée objet

L'approche orientée objet organise une application ou, plus généralement, un problème en une collection d'entités physiques en interactions : les objets [1.1.1]. Chaque objet est une entité physique unique, qui possède une identité qui lui est propre. C'est le principe de l'**identité**. Suivant ce principe, deux objets, par ailleurs totalement semblables, sont néanmoins distincts par leur identité.

Certains objets possèdent une structure et un comportement communs. On appelle **classe** la description de la structure et du comportement communs d'un ensemble d'objets. C'est le principe de la classification, ou taxinomie. Chaque objet est alors une **instance** de cette classe.

Les **attributs** d'une classe définissent sa structure, et les **méthodes** son comportement. Chaque instance de classe possède ses propres valeurs pour chaque attribut (les données), mais partage les noms d'attributs et les méthodes avec les autres instances de la classe. Chaque objet contient une référence implicite à sa classe : « il sait ce qu'il est ».

Dans l'exemple donné par la Figure 5, la classe « Matériau » définit le modèle de données associé aux matériaux dans cette application. Ils sont caractérisés par leur nom, leur perméabilité et leur conductivité. Le processus d'instanciation permet de créer des (instances de) matériaux, chacun avec ses caractéristiques propres.

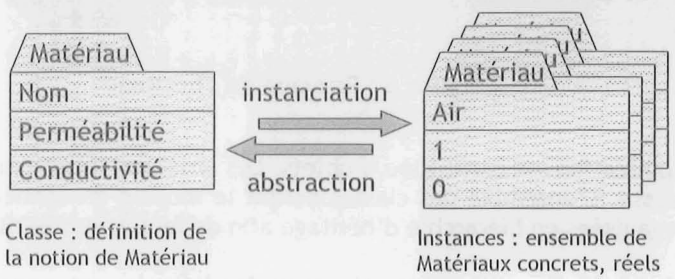


Figure 5 : Classe et objet

Un objet et sa classe n'appartiennent pas au même niveau d'existence. L'objet existe réellement au sein de l'application, alors que la classe appartient à un niveau supérieur, qualifié de méta par rapport au niveau de l'objet. La classe contient une part de la connaissance que nous avons des objets. Elle leur donne un sens, une interprétation.

La classe est le modèle des objets, et l'ensemble des classes constitue le modèle de données objet de l'application.

Une opération est identifiée par son nom, l'ensemble des types passés en paramètres et le type de retour. Cette structure, appelée **signature**, identifie de manière unique une opération. Dans l'approche orientée objet, une opération (tâche) peut être réalisée différemment selon la classe qui l'implante. L'implantation d'une opération dans une classe est appelée méthode. Une même opération peut être implanter différemment par des classes différentes. C'est le concept du **polymorphisme**.

Si deux classes possèdent des attributs et des méthodes en commun, on évite la redondance de code en créant une troisième classe factorisant ces éléments partagés, et en établissant une relation d'**héritage** entre cette dernière classe et les deux premières.

La classe factorisant les méthodes et attributs communs est appelée super-classe. Les autres classes, dites sous-classes, héritent de la super-classe, et la spécialisent en rajoutant leurs propres attributs et méthodes.

De manière générale, lors de la phase d'analyse objet du problème, un lien d'héritage existe entre deux classes A et B si « A est une sorte de B ». Par exemple, une bobine est caractérisée par l'intensité I et la fréquence ω du courant qui la parcourt, ainsi que par sa forme. Quelque soit sa forme particulière, la loi de Biot et Savart pemet de calculer le champ généré par elle en tout point

de l'espace. La Figure 6 illustre le fait que les bobines circulaire ou rectangulaire sont par définition des bobines.

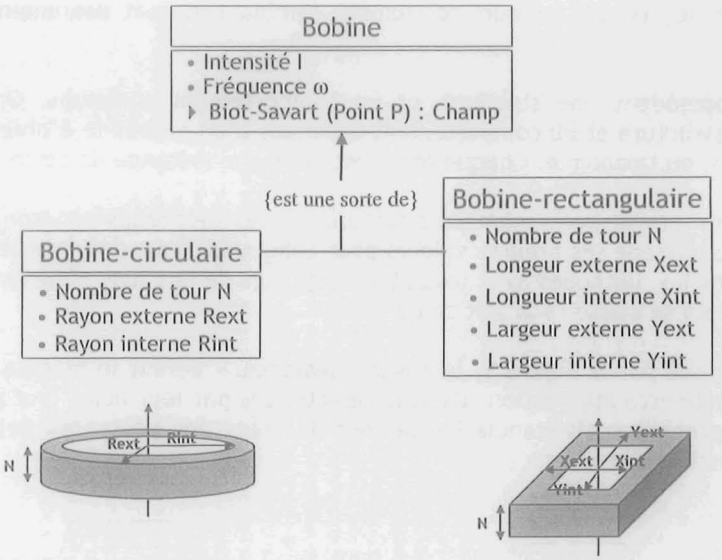


Figure 6 : Principe de l'héritage

En résumé, une application est constituée d'objets. Les objets ayant des propriétés communes sont décrits par une classe. L'ensemble des classes définit le modèle de donnée objet de l'application. Les classes sont organisées en hiérarchie d'héritage afin de factoriser l'écriture du code.

Au sein de l'application, les objets entretiennent des liens les uns avec les autres. Il est donc naturel de représenter ces liens au niveau des classes, comme élément à part entière du modèle objet de l'application. Ce sont les **associations**.

Une association est un lien qui existe entre deux classes. Ce lien peut être nommé. Chaque classe participante peut se voir désignée par un **rôle** au sein de l'association.

En reprenant l'exemple précédent, on peut définir la classe « Courant » caractérisée par une intensité I et une fréquence ω. La classe « Bobine » entretient alors un lien avec la classe « Courant » :

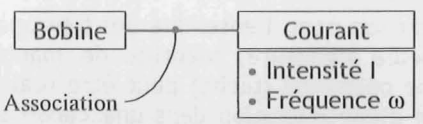


Figure 7 : Association entre Bobine et Courant

La notion d'association place au même niveau les deux partenaires de celle-ci. Or il peut être nécessaire d'utiliser une relation de type composant/composé : c'est l'**agrégation**. Contrairement à l'association où les objets ont un rôle de même niveau, l'agrégation subordonne un des objets (le composant) à un autre (le composé).

Le dernier principe important de la modélisation orientée objet est celui d'**encapsulation**, réalisé grâce aux **interfaces**. Une classe est encapsulée si sa structure de données interne est cachée aux autres classes. Seule les méthodes de la classe doivent y avoir accès.

En effet l'implantation des opérations d'une classe (les méthodes) est fondée sur les structures de données internes de la classe elle-même. On veut pouvoir réutiliser les opérations de la classe, mais sans ce soucier des détails de leur implantation. Pour cela on utilise le concept d'interface.

L'interface est l'entité de plus haut niveau et de plus haute généricité présente dans un modèle objet. Elle propose un ensemble de services (opérations) réalisables, sans spécifier leur implantation (attributs et méthodes), contrairement aux classes.

Une classe qui implémente une interface s'engage contractuellement à réaliser les services qui sont spécifiés par l'interface. L'implémentation d'une interface diffère de l'héritage au sens où la classe n'hérite d'aucune méthode implantée ou attribut.

L'intérêt des interfaces réside dans la généricité quelles apportent aux applications développées. L'application ne manipulant qu'une interface, un développeur peut sans souci remplacer au sein de l'application une classe qui implémente l'interface par une autre classe plus performante, sous réserve qu'elle implémente la même interface.

2.2 *Unfied Modeling Language*

Fin 1997, l'OMG adoptait un langage standard et normalisé pour la modélisation orientée objet nommé Unified Modeling Language (UML) [1.2]. L'objet de ce paragraphe est d'introduire les éléments syntaxiques principaux du langage, ainsi que leur sémantique, afin de comprendre et de concevoir un modèle de données objet en UML.

2.2.1 Présentation du langage

UML est un langage de modélisation objet graphique. En d'autres termes, il permet de décrire schématiquement n'importe quel système objet. C'est un outil extrêmement expressif et visuel. De plus, chaque symbole de la notation UML possède une sémantique bien définie et normalisée. Ainsi un modèle peut être interprété sans risque d'ambiguïté.

Un modèle UML est un ensemble de diagrammes, chaque diagramme étant composé d'éléments du langage en relation les uns avec les autres. En théorie, un diagramme peut contenir n'importe quelle combinaison d'éléments et de relations. Cependant, en pratique, seul un petit nombre de combinaisons fréquentes se répètent. Ce sont les neufs diagrammes traditionnels d'UML :

- le diagramme de classes
- le diagramme d'objets
- le diagramme de déploiement
- le diagramme de composants
- le diagramme de cas d'utilisation
- le diagramme de d'états-transitions
- le diagramme de d'activités
- le diagramme de séquences
- le diagramme de collaborations

Les cinq premiers diagrammes représentent des vues structurelles du système, alors que les quatre derniers représentent des vues comportementales. Dans le cadre de notre travail, nous ne nous intéresserons qu'au premier diagramme, le diagramme de classes.

Un diagramme de classes est composé d'un ensemble de classes, d'interfaces et de collaborations, ainsi que leurs relations. Il représente la conception statique d'un système.

2.2.2 Les éléments du langage

a- Les entités

Une **classe** représente un ensemble d'objets qui partagent les mêmes attributs, les mêmes opérations, les mêmes relations et la même sémantique. Une classe est symbolisée par un rectangle (cf. Figure 8). Chaque classe doit avoir un nom unique, différent de celui des autres classes.

Un **attribut** est une propriété nommée d'une classe. Il représente une propriété commune des objets modélisés par cette classe. Il décrit le type de valeur que peut prendre cette propriété. Une classe peut avoir un ou plusieurs attributs, ou ne pas en avoir du tout. Les attributs sont listés dans le compartiment situé juste en dessous du nom de la classe. L'ensemble des valeurs prises par les attributs d'un objet définit son état.

Une **méthode** est l'implémentation d'une **opération** qui peut être demandée à toutes les instances de la classe dans le but de déclencher un comportement. Une classe peut aussi bien comporter plusieurs méthodes que ne pas en avoir du tout. L'invocation d'une méthode sur un objet peut avoir pour effet d'en modifier l'état. Les méthodes sont listées dans le compartiment juste en dessous de celui des attributs de la classe.

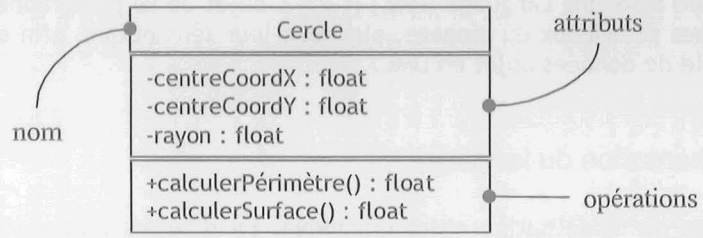


Figure 8 : classe

Une **interface** est un ensemble d'opérations utilisées pour décrire un service. Contrairement à la classe, elle ne décrit aucune structure (pas d'attribut) ni aucune implémentation pour les opérations (pas de méthode).

Dans sa forme non développée, une interface se représente par un simple cercle. Sous sa forme développée, elle est représentée par un rectangle. Le premier compartiment contient le nom de l'interface, avec la mention « interface », afin de bien la distinguer d'une classe. Le compartiment au-dessous contient la liste des opérations (cf. Figure 9).

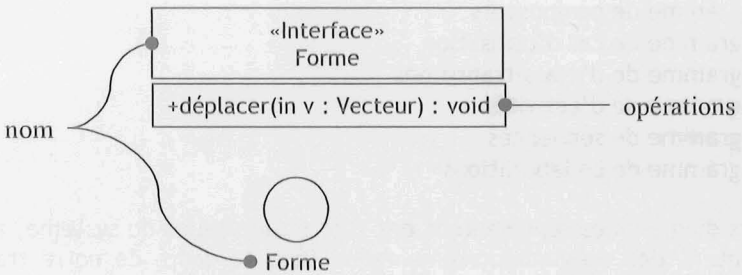


Figure 9 : Interface développée et non développée

Enfin, UML permet de définir des énumérations, c'est à dire un ensemble de valeurs regroupées comme un type de données particulier (cf. Figure 10 : Enumeration).

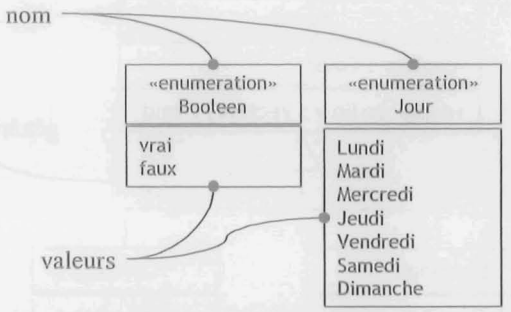


Figure 10 : Enumeration

b- Les relations

Une relation est une connexion sémantique entre deux éléments d'un diagramme. En modélisation orientée objet, les relations les plus importantes sont les dépendances, les associations, les généralisations et les réalisations.

Une **dépendance** est une relation d'utilisation qui établit que la modification des spécifications d'un élément peut en affecter un autre qui l'utilise. La réciproque n'est pas nécessairement vraie. On se sert des dépendances pour montrer qu'un élément en utilise un autre.

Par exemple, une classe peut en utiliser une autre dans la signature d'une opération. Si la classe utilisée change, l'opération de l'autre classe peut en être affectée parce que la première présente maintenant une interface ou un comportement différent.

Une dépendance est symbolisée par une flèche en pointillées, dirigée vers l'élément à l'égard duquel existe la dépendance (cf. Figure 11). Dans cet exemple, la méthode d'affichage d'un dessin dépend de la fenêtre où il s'affiche. Toute modification de l'implantation de la classe « Fenetre » aura donc une conséquence directe sur la classe « Dessin ».

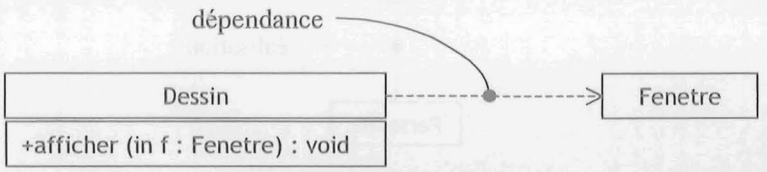


Figure 11 : Dépendance

La **généralisation** est une relation d'héritage entre un élément général (la super-classe) et un élément dérivé plus spécifique (la sous-classe). La sous-classe hérite des propriétés (attributs et méthodes) de sa super-classe. Lorsque la sous-classe possède une méthode ayant la même signature que sa super-classe, le principe de polymorphisme impose l'utilisation de la plus spécifique, c'est à dire celle de la sous-classe.

La généralisation implique que des instances de la sous-classe puissent être utilisées partout où des instances de la super-classe peuvent être utilisées. La réciproque est fausse.

Dans une hiérarchie de classes, la classe n'ayant pas de super-classe est qualifiée de classe de base. Les classes qui ne sont pas spécifiées sont dites classes feuilles.

Comme une classe, une interface peut participer à une relation de généralisation d'une autre interface.

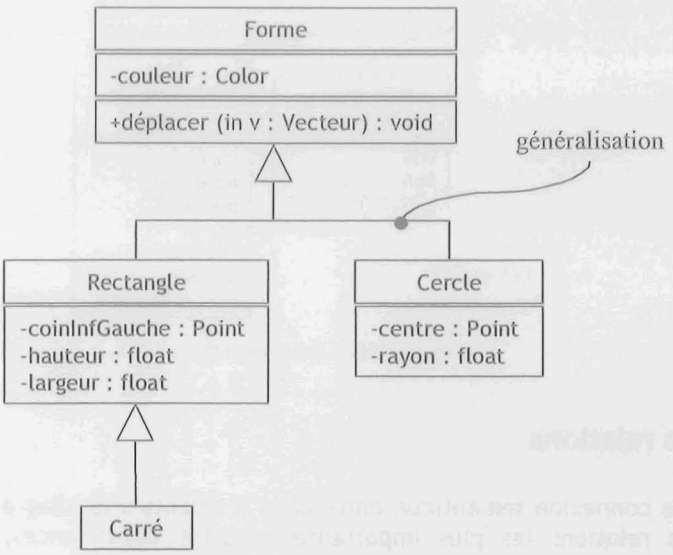


Figure 12 : Généralisation

La généralisation est symbolisée par une flèche pleine dont la pointe est creuse, et dirigée de la sous-classe vers la super-classe (cf. Figure 12).

La **réalisation** est une relation entre une interface et une classe. L'interface décrit un contrat de service (les opérations) dont la réalisation est garantie par la classe (attributs et méthodes).

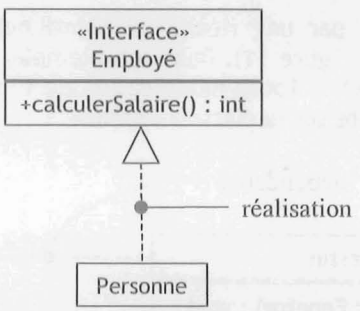


Figure 13 : Réalisation

En UML, une réalisation se représente par une flèche en pointillés dont la pointe est creuse, et dirigée de la sous-classe vers la super-classe (cf. Figure 13).

Une **association** est une relation structurelle entre deux classes. Elle est représentée par une ligne pleine qui relie les deux classes. Il n'est pas interdit que les deux extrémités d'une association forment une boucle en se rattachant à la même classe. Cela signifie qu'une instance de cette classe peut entretenir un lien avec une autre instance de cette classe.

Une association peut avoir un nom, qui sert à décrire la nature de la relation. Pour éviter toute ambiguïté, on note généralement la direction du nom par un triangle qui pointe dans le sens désiré.

Quand une classe participe à une association, elle y tient un **rôle** spécifique. Le rôle est la fonction que la classe assume dans l'association. Il peut être indiqué sur l'association, près de la classe en question.

Dans de nombreuses situations de modélisation, il est important de définir combien d'objets peuvent être reliés par l'intermédiaire d'une instance d'association. Ce nombre définit la **multiplicité** d'un rôle d'une association. La multiplicité d'un rôle implique le nombre d'instances

qui se trouvent à cette extrémité de l'association lorsqu'elle est réalisée. On peut représenter une multiplicité exacte (1, 3) ou un intervalle (3..5, 1..*).

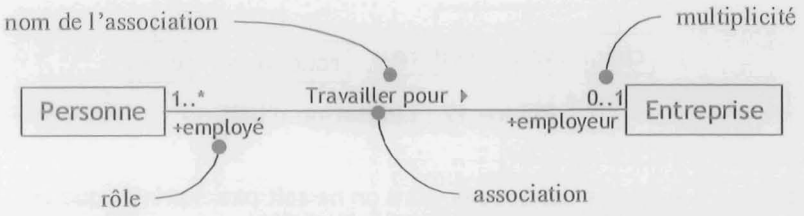


Figure 14 : Association

Dans le cas représenté par la Figure 14, au moins une personne, qui est un employé, travaille pour au plus une entreprise, qui est son employeur.

Une association entre deux classes représente une relation structurelle entre deux classes de même niveau conceptuel. Or, il est parfois souhaitable de représenter la relation « tout/partie » dans laquelle une classe représente un élément plus grand (le « tout », ou composite) composé d'éléments plus petits (les « parties », ou composants). Ce genre particulier d'association est une **agrégation**.

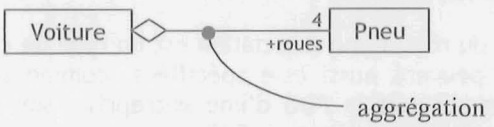


Figure 15 : Agrégation

Elle est représentée par l'ajout d'un losange du coté du composite (cf. Figure 15).

Il existe une variation de l'agrégation - la **composition** - qui ajoute des règles sémantiques importantes par rapport à l'agrégation simple. Les composants peuvent être créés après le composite, mais ils vivent et meurent avec lui. Ils peuvent également être retirés de manière explicite avant la mort du composite. En d'autres termes : la destruction du composite entraîne la destruction des composants.

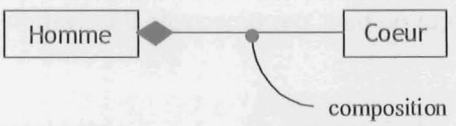


Figure 16 : Composition

La composition se symbolise comme une agrégation, à la différence près que le losange de l'extrémité est plein (cf. Figure 16).

Le langage UML comprend d'autres éléments, et d'autres relations. Nous laissons le soin au lecteur intéressé de se reporter soit l'excellent livre « Le guide de l'utilisateur UML » de Booch, Rumbaugh et Jacobson, traduit et paru aux éditions Eyrolles, soit directement aux spécifications du langage UML disponibles sur le site Internet de l'OMG : http://www.omg.org.

Bien que le langage UML soit le langage normalisé et standardisé pour la modélisation objet, et malgré sa richesse et son expressivité, UML a ses limites. Prenons l'exemple suivant :

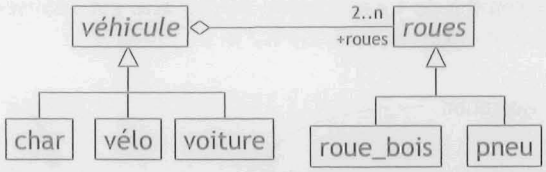


Figure 17 : Limitation d'UML

Tout type de véhicule a au moins deux roues. Mais on ne sait pas exprimer que :

- un char et un vélo ont exactement deux roues, alors qu'une voiture en a quatre
- un char utilise tout type de roues, mais les vélos et les voitures n'utilisent que des pneus, et jamais de roues en bois.

UML ne possède pas les éléments syntaxiques appropriés pour décrire ce genre de contraintes sur un modèle de données. Cette information peut néanmoins être décrite en utilisant le formalisme OCL.

2.3 *Object Constraint Language*

Un diagramme de classes ne représente pas seulement la structure d'un système objet, à savoir ses éléments et leurs relations. Certaines notations du diagramme permettent également de définir des contraintes de conception de manière informelle.

Par exemple, la multiplicité du rôle d'une association est un type de contrainte. Des contraintes de conception plus complexes peuvent aussi être spécifiées, comme sur la Figure 18, en langage naturel. Ainsi, une personne est soit le PDG d'une entreprise, soit un employé. L'ensemble des généraux forme un sous-ensemble des militaires. Enfin, un conscrit est nécessairement majeur.

L'utilisation du langage naturel pour exprimer des contraintes de conception entraîne très souvent l'apparition d'ambiguïtés. Un outil informatique ne peut donc les analyser et les manipuler. Il est donc nécessaire de définir un langage standard pour exprimer les contraintes. C'est le but de l'Object Constraint Language (OCL) [1.3].

OCL est un langage formel et lisible permettant d'exprimer les contraintes qui existent au sein d'un modèle UML. Ces contraintes font partie intégrante du modèle UML en question.

Une expression OCL est une contrainte sans effet de bord. L'évaluation d'une expression OCL retourne simplement une valeur sans modifier l'état du système sur lequel elle est évaluée. C'est donc un langage de requête, qui ne peut pas provoquer l'activation d'une quelconque opération du système.

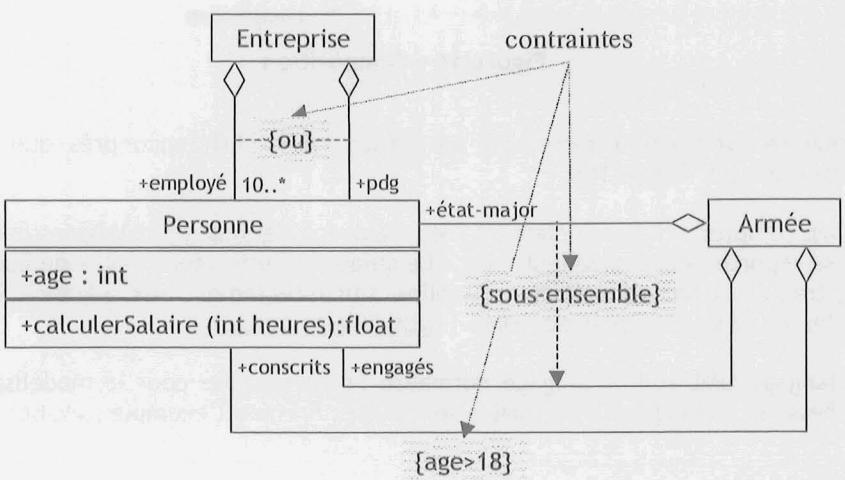


Figure 18 : Diagramme UML contractualisé

Le langage OCL distingue trois types de contraintes : les invariants, les pré et les post-conditions.

2.3.1 Les invariants

Une expression OCL de type invariant exprime une contrainte associée à une classe d'un modèle UML. Cette classe constitue le contexte de l'expression. La contrainte peut être évaluée sur toute instance de cette classe. Elle est représentée dans le corps de la contrainte par le mot-clé « self ». La forme générale d'un invariant de classe est :

context *Une_Classe* inv *Nom_Règle* [2] :
[corps de la contrainte évaluée sur self, une instance de *UneClasse*]

Une expression OCL de type invariant de classe exprime une contrainte que doit vérifier toute instance de cette classe, à tout moment. Le résultat de l'évaluation de cette expression sur une instance particulière retourne donc une valeur booléenne.

Voici les invariants de classe associées aux trois contraintes spécifiées en langage naturel sur la Figure 18, suivie de leur traduction littérale en italique pour leur compréhension :

```
context Armée inv :
  self.conscrits -> forAll ( cons | cons.age > 18 )
```

Soit « self » l'armée. Tous les conscrits de l'armée (self.conscrits) doivent être agé de plus de 18 ans.

```
context Armée inv :
  self.état-major -> forAll ( off | self.engagés -> exists ( eng | eng == off ) )
```

Soit « self » l'armée. Tout membre de l'état-major (désigné par « off ») fait partie des engagés (« eng »).

```
context Entreprise inv :
  self.employés -> forAll ( emp | emp <> self.pdg )
```

Soit « self » une entreprise. Les employés de l'entreprise (self.employés) ne sont pas son pdg (self.pdg).

La multiplicité du rôle d'une association est également une contrainte, que l'on peut traduire par un invariant de classe :

```
context Entreprise inv :
  self.employés -> size > 10
```

Soit « self » une entreprise. Le nombre d'employés de l'entreprise (self.employés-> size) doit être supérieur à 10.

Sur les exemples que nous venons de citer, on constate l'utilisation d'opérations sur des instances ou leurs attributs (<>, ==, ->forAll(...), ->size, ...). Ces opérations sont prédéfinies dans le langage OCL, et nous y reviendrons par la suite.

2.3.2 Les pré et post-conditions

Une contrainte de pré ou de post-conditions n'est plus associée à une classe, mais à une méthode de classe. La syntaxe d'une expression OCL de pré/post-condition est la suivante :

[2] Le nom de la règle est un élément facultatif

```
context UneClasse ::UneMéthode ( p1 : Type1, p2 : ...) : TypeRetour
  pre  : p1 > ...
  post : return = ...
```

Après le mot-clé « context » se trouve la signature de la méthode, spécifiant son nom complet (*UneClasse ::UneMéthode*), la liste ordonnée des types de ses arguments, et le type retourné par la méthode.

Le mot-clé « self » peut être utilisé dans le corps de la contrainte de la même manière que pour les invariants. Il représente l'instance de la classe « UneClasse » sur laquelle la méthode « UneMéthode » peut être invoquée. Les identificateurs p1, p2... et le mot-clé « return » représentent réciproquement les instances passées en argument de la méthode et le résultat de celle-ci.

Voici par exemple une pré et une post-condition portant sur la méthode « calculerSalaire » de la classe « Personne » de la Figure 18 :

```
context Personne::calculerSalaire (heures : int) : float
  pre  : heures > 0
  post : return > 0
```

Soit la méthode "calculerSalaire" de la classe "Personne". Le nombre d'heures (heures) passé en paramètre doit être strictement positif, ainsi que le résultat obtenu (return).

Une pré-condition est une contrainte qui doit être évaluée sur les instances participant à la méthode, avant l'invocation de celle-ci. À contrario, une post-condition est évaluée sur les instances après invocation de la méthode.

Nous ne développons pas davantage cet aspect d'OCL, dans la mesure où nous n'en n'avons pas eu l'utilité dans la suite de ce travail.

2.3.3 Types et opérations prédéfinis

OCL est un langage typé. Il dispose d'un ensemble de types et d'opérations prédéfinis.

Il y a d'abord les types de base : les booléens (Boolean), les entiers (Integer), les réels (Real) et les chaînes de caractères (String). OCL définit des opérations sur ces types, récapitulées dans le Tableau 7 :

Type	Opérations
Boolean	and, or, xor, not, implies, if-then-else
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()
String	ToUpper(), concat()

Tableau 7 : Opérations sur les types prédéfinis

Chaque contrainte OCL est écrite dans le contexte d'un modèle UML particulier. Toute classe de ce modèle peut donc être utilisée comme un type à part entière dans toutes les contraintes associées à ce modèle.

Bien sûr, les attributs et associations définis sur ces classes sont disponibles, pour permettre l'accès à leur valeur dans le cadre de leur évaluation dans le corps de la contrainte.

En ce qui concerne les méthodes et les opérations, seules celles qui sont certifiées sans effet de bord sont disponibles, conformément aux spécifications du langage. En effet, l'invocation d'une opération ne doit pas modifier l'état du système sur lequel la contrainte est évaluée. Une méthode ou une opération est garantie sans effet de bord si son attribut « isQuery » est valué à vrai dans le modèle UML.

L'ensemble des attributs, associations, méthodes et opérations sans effet de bord d'une classe sont appelées les propriétés de la classe. La valeur d'une propriété d'une instance de classe est accessible par un point, suivit du nom de la propriété :

```
context Personne inv :
  self.age > 0
```

L'évaluation d'une association de multiplicité inférieure ou égale à un est une instance. Si la multiplicité est supérieure à un, l'évaluation renvoie un ensemble d'instances. OCL possède des types spécifiques pour manipuler ce genre de résultat : ce sont les types « Collection », « Set », « Bag » et « Sequence ». OCL type une association de multiplicité supérieure à 1 comme un « Set ». Si l'association est ornée du mot-clé « ordered » (qui est une contrainte sur l'association !), elle est typée comme une « Sequence ». « Set », « Bag » et « Sequence » dérivent du type « Collection ».

Ces types disposent d'opérations prédéfinies accessibles par le symbole « -> ». Ces opérations et leur sémantique sont récapitulées dans le Tableau 8. Dans ce tableau, certaines opérations de sélection nécessitent un critère de filtrage (noté « Exp »), c'est à dire une expression évaluable portant sur les éléments de la Collection elle-même.

Opération	Sémantique
Collection->size() : Integer	Retourne le nombre d'instances de la collection
Collection->empty() : Boolean	Retourne vrai si la collection est vide, faux sinon
Collection->notEmpty() : Boolean	Retourne vrai si la collection est non vide, faux sinon
Collection->select(Exp) : Collection	Retourne le sous-ensemble des instances de la collection vérifiant l'expression Exp
Collection->reject(Exp) : Collection	Opération réciproque de l'opération précédente.
Collection->forAll(Exp) : Boolean	Retourne vrai si l'ensemble des instances de la collection vérifie l'expression Exp.
Collection->exists(Exp) : Boolean	Retourne vrai si il existe au moins une instance de la collection vérifiant l'expression Exp.

Tableau 8 : Opérations prédéfinies sur les Collections

Soit « c » une collection contenant quatre Personnes : c = {p1, p2, p3, p4}, d'ages respectifs : p1.age = 43, p2.age = 41, p3.age = 19, p4.age = 7. Voici quelques exemples d'utilisation de ces opérations sur la Collection « c », accompagnés d'une explication en italique :

```
c -> size = 4
```
L'ensemble « c » contient quatre éléments.

```
c -> empty = false
c -> notEmpty = true
```
L'ensemble « c » est non vide.

```
c -> select( p | p.age > 18 ) = {p1, p2}
```
L'ensemble des éléments de « c » (notés « p ») dont l'age est supérieur à 18 est {p1,p2}

```
c -> reject( p | p.age > 18 ) = {p3, p4}
```

L'ensemble des éléments de « c » dont l'age n'est pas supérieur à 18 est {p1,p2}

c -> forAll (p . age > 0) = true
Tous les éléments de « c » ont un age positif.

c-> exists (p.age = 19) = true
Il existe au moins un élément de « c » dont l'age est 19.

Enfin OCL incorpore un ensemble d'opérations portant sur le typage des objets eux-mêmes (cf. Tableau 9) :

Opération	Sémantique
Object->oclIsTypeOf(Type) : Boolean	Retourne vrai si le type de l'objet est « Type »
Object->oclIsKindOf(Type) : Boolean	Retourne vrai si l'objet est de type « Type »
Object->oclAsType(Type) : Object	Retourne l'objet casté dans le type « Type »

Tableau 9 : Opérations de typage

Par exemple soit p une instance de la classe « Personne » (cf. Figure 18), alors :

p -> oclIsTypeOf(Personne) = true.
« p » est une instance de la classe « Personne ».

2.4 Bilan

UML est aujourd'hui un succès méthodologique reconnu, autant qu'un succès industriel. Il est le principal outil de spécification d'un système objet. Des logiciels dédiés à UML permettent de générer directement une partie du code informatique associé à un modèle de données.

Comme nous l'avons déjà évoqué, malgré son expressivité, UML n'est pas un langage permettant de spécifier les contraintes d'une application. C'est pour combler cette carence qu'a été conçu OCL. On retrouve dans la séparation UML/OCL la dualité que nous avons déjà diagnostiquée dans les connaissances algorithmiques ou expertes des concepteurs de logiciels de simulations numériques. OCL sera le langage dans lequel nous décrirons nos domaines de validité. Les expressions structurées insi obtenue seront manipulées par un système expert.

3 Limitations et Synergies

L'objet des deux premiers paragraphes était la description détaillée et didactique des deux technologies que nous allons faire cohabiter dans notre outil. Il est temps maintenant d'analyser les limitations de l'une et l'autre technologie, afin de faire ressortir les synergies potentielles qui existent entre elles.

3.1 Limitations

3.1.1 Limitations ses systèmes experts

Comme toute solution technologique, les systèmes experts ont été conçus par une communauté d'utilisateurs, ici celle de l'intelligence artificielle, pour répondre à leurs besoins spécifiques. Ces besoins ont défini le domaine d'utilisation des systèmes experts, ainsi que leurs limitations intrinsèques.

a- Le coût

Le premier inconvénient majeur d'un système expert est le coût de calcul d'une inférence. Calculer une inférence, c'est trouver toutes les solutions possibles d'une requête par rapport à une base de faits donnée. Une requête est la conjonction d'un ensemble de contraintes.

Soit N le nombre de contraintes composant une requête spécifique. Si on suppose que chaque contrainte possède exactement P solutions dans la base de faits, alors l'évaluation d'une requête va nécessiter l'évaluation des N contraintes dans la base de faits. La requête aura donc N * P solutions.

Or, la puissance d'un système expert tient précisément à deux facteurs :

- une expertise pointue reflétée par des règles complexes (N grand)
- une expertise exhaustive sur une grand collection de faits (P grand).

Par conséquent un système expert puissant et exhaustif peut être extrêmement coûteux. A ce sujet, le lecteur peut se reporter à l'**Annexe II** où l'étude du coût d'un système expert que nous avons conçu est détaillé.

b- L'absence de réutilisabilité

Le deuxième inconvénient d'un système expert est sa réutilisabilité quasi-nulle. L'industrie du logiciel, afin de diminuer ses coûts de développement, privilégie les solutions logicielles par intégration de composants. On peut citer les méthodologies de Frameworks et de Design Pattern, ainsi que les technologies de type Java™ Beans.

Chaque composant est développé et testé conformément à des spécifications précises pour réaliser un ensemble de services. Ils sont réutilisés à chaque fois qu'une nouvelle solution logicielle composite nécessite ce type de service. Ceci évite de refaire ce qui à déjà été fait et éprouvé.

Or, un système expert est, par essence même, dédié. En effet, il est conçu pour raisonner dans un domaine de connaissance particulier. L'ensemble des concepts d'un domaine, et les règles qui établissent une relation sémantique entre ces concepts, est appelée ontologie du système expert. Elle est définie par les experts du domaine.

Par exemple, l'ontologie de l'électromagnétisme inclut les concepts de champs électriques et magnétiques, de matériaux ferromagnétiques, de perméabilité relative, de sources de champs ou de courants... ainsi que les équations de Maxwell ou la loi de Biot et Savart qui sont des règles reliant ces concepts.

L'ontologie d'un système expert nécessite un formalisme pour être exprimée. Le point crucial de notre analyse est que le formalisme utilisé pour exprimer une ontologie particulière est souvent spécifique à l'ontologie elle-même.

En d'autres termes, on ne peut généralement pas utiliser un formalisme conçu dans le cadre de l'ontologie d'un domaine de connaissance particulier pour exprimer les concepts d'un autre domaine.

Or, les mécanismes de raisonnement d'un système expert sont fondés sur le formalisme de la base de connaissance. Un moteur d'inférences ne peut donc être mis en œuvre que dans un domaine de connaissance spécifique, décrit avec le formalisme idoine.

Le formalisme est donc l'élément fondamental autour duquel s'articule l'implantation d'un système expert. Ce rôle central est sans doute la raison pour laquelle le développement et la réalisation d'un système expert en tant que composant logiciel réutilisable n'ai jamais été effectuée à ce jour.

3.1.2 Limitations d'OCL

Le paradigme objet est aujourd'hui mature, aussi bien au niveau des langages de programmation (Java, C++,...) que des méthodologies de modélisation. Cette maturité se traduit par le développement de normes standards, et des outils qui leurs sont dédiées. De nombreux Ateliers de Génie Logiciel (AGL) fondés sur UML sont disponibles, tels que Rational-Rose™, Objecteering/UML™, UML-Studio™ ou encore Poséidon™.

Ces ateliers de Génie Logiciel possèdent pour la plupart les fonctionnalités suivantes :

- aide à l'analyse d'un système et conception du modèle UML associé,
- génération automatique de code,
- génération de la documentation,
- gestion des versions,
- rétroingénierie dans certains cas (opération réciproque de la génération de code à partir du modèle UML).

Du fait de l'arrivée tout à fait récente du langage OCL, il n'existe encore que très peu d'outils permettant d'utiliser ce nouveau standard pour spécifier l'aspect contractuel d'une application. D'autre part, même lorsque cette norme est implantée, aucune réelle fonctionnalité ne lui est associée, si ce n'est parfois une analyse syntaxique des expressions OCL.

Cela provient du fait que, en dehors d'un contenu informatif riche et intéressant pour les développeurs, la seule fonction d'une expression OCL est d'être évaluée soit sur une instance de classe pour les invariants, soit sur une opération pour les pré/post-conditions. Or, toute évaluation implique nécessairement l'existence préalable d'objets sur lesquels les susdites expressions puissent être évaluées.

Les expressions OCL sont donc créées en même temps que le modèle UML de l'application, et appartiennent à ce dernier. Lors de la génération de code (Figure 19), on constate la disparition de cette information contextuelle.

En effet, les expressions OCL sont formalisées dans un langage de modélisation de très haut niveau, proche du langage humain. A l'inverse, les langages de programmations informatiques classiques ne possèdent pas les structures syntaxiques propres à l'écriture de l'information contractuelle.

Il est à noter cependant deux exceptions notables Le langage Eiffel permet d'associer à une opération des pré/post-conditions, nommées assertions. D'autre part, la dernière évolution du langage Java (Java 1.4 du J2SE) implante la programmation d'assertions. Mais là encore, c'est une innovation des plus récentes.

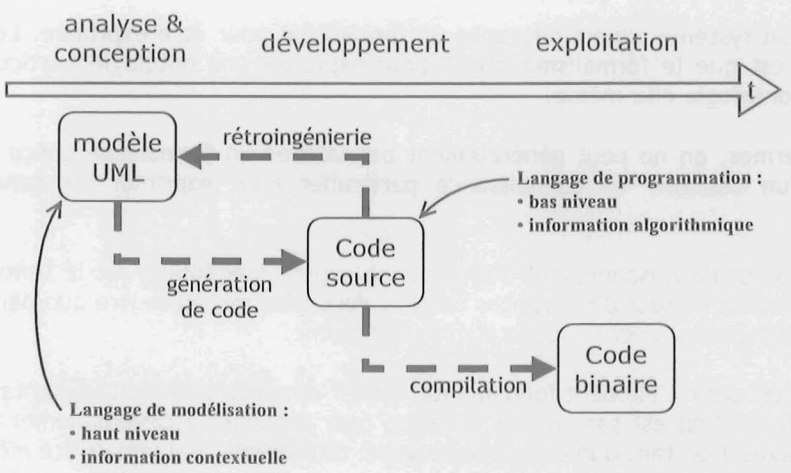


Figure 19 : Différents niveaux de langage dans un Atelier de Génie Logiciel

En conclusion, le langage OCL est un outil riche de promesses. Il est utilisé pendant la phase de spécifications pour contractualiser l'application. Il n'existe pas encore aujourd'hui d'outils

permettant une réelle exploitation de ces expressions dans le cadre de l'utilisation de l'application proprement dite.

3.2 Synergies

Notre travail rapproche deux technologies particulières, dont les limitations sont celles que nous venons de voir. Ce rapprochement que nous initions peut être le vecteur d'une synergie au niveau des méthodologies.

3.2.1 Apport des systèmes experts à la modélisation objet

Les ateliers de Génie Logiciel sont des outils destinés aux concepteurs d'applications. Ils permettent la description de diagrammes UML à partir desquels les principales structures du code informatique peuvent être générées automatiquement. Ils permettent ainsi aux concepteurs de se consacrer à leurs tâches les plus importantes : la modélisation et l'implantation des algorithmes de calcul.

Le langage OCL est un formalisme puissant dédié à la contractualisation des modèles de données UML. Les développeurs d'applications l'utilisent d'ores et déjà dans la phase de spécification d'une application.

Cependant, OCL est une norme récente. Par conséquent, les ateliers de Génie Logiciel qui permettent la description des expressions OCL possèdent peu de fonctionnalités associées, à l'exception de la vérification syntaxique. Au delà des ateliers, les logiciels générés ne mettent pas en œuvre l'évaluation des expressions OCL au cours de leur utilisation, sur les instances présentes dans leur bases de données.

Il existe deux origines à cet état de fait. D'une part les expressions OCL, associées au modèle UML, ne sont pas présentes en tant que tel dans le code informatique de l'application. En effet, les langages informatiques de programmation objet ne possèdent pas les structures syntaxiques permettant de décrire les expressions OCL. A ce titre, on retrouve ici le fossé séparant nettement les natures algorithmique et experte de la connaissance. L'absence des structures syntaxiques implique naturellement l'absence des mécanismes algorithmiques permettant l'évaluation des expressions OCL.

A notre connaissance, il n'existe aucun logiciel mettant en œuvre l'évaluation des expressions OCL appartenant à son modèle objet. Le système expert que nous avons implanté est donc l'un des tous premiers outils à réaliser cette opération dans le cadre concret d'une utilisation réelle d'une application.

En outre, l'utilisation d'un système expert permet de dépasser l'apport formel que propose le langage OCL, et sa seule utilisation originellement prévue : la validation. En effet, le système expert traite les expressions OCL comme des prédicats Prolog **inversibles**. Il permet alors d'expliquer les erreurs commises par l'utilisateur, et de lui proposer dynamiquement des choix valides.

3.2.2 Apport de la modélisation objet aux systèmes experts

Nous avons vu que la technologie des systèmes experts a deux limitations fortes : sa faible réutilisabilité et son coût d'utilisation.

Nous avons vu précédemment que la réutilisabilité limitée des systèmes experts est une conséquence directe de la manipulation de formalismes spécifiques à chaque ontologie. Or, l'ontologie d'un logiciel de simulations numériques particulier se réifie au travers de ses structures de données, ses algorithmes de calcul et ses domaines de validité. L'existence et l'utilisation de

formalismes standards pour la description des structures de données (UML) et les domaines de validité (OCL) garantissent la réutilisabilité d'un moteur d'inférences fondé sur ces formalismes.

Les langages standards de la modélisation objet permettent donc de concevoir et de développer des composants logiciels experts. Ils sont intégrables dans toute application possédant un diagramme UML associé à un ensemble d'expressions OCL.

La deuxième limitation des systèmes experts est leur coût d'utilisation. Nous savons que ce coût combinatoire est directement lié aux nombres de règles à évaluer d'une part et surtout au nombre de faits disponibles d'autre part.

Or, dans ce travail, nous nous intéressons particulièrement à la mise en œuvre des domaines de validité des modèles de dispositifs dans les logiciels de simulations numériques. Or, même pour un cas très complexe, la modélisation d'un dispositif contient un ensemble relativement restreint d'instances. Dans le cas du contacteur du tutoriel de Flux3D par exemple, on compte seulement quatre instances de régions volumiques, chacune associée à une formulation, un matériau et un volume. Par conséquent, l'utilisation d'un système expert dont la base de faits se restreint aux instances modélisant le dispositif aura pratiquement un coût tout à fait raisonnable.

D'autre part, l'outil que nous allons concevoir est destiné à s'intégrer au processus de modélisation, dont l'unique intervenant est l'utilisateur. Dans ces conditions, il n'est pas rédhibitoire d'avoir des temps de réponse pour la validation, l'explication ou la proposition qui soit de l'ordre de la seconde.

3.3 Bilan

L'analyse critique des deux technologies que nous allons utiliser met en évidence leurs carences respectives. L'implantation de systèmes experts fondés sur les formalismes objets standards tels que UML et OCL permet d'établir une réelle synergie pour combler en partie ces carences.

Le système expert que nous allons développer dans le prochain chapitre pourra être conçu sous la forme d'un composant logiciel **réutilisable**. Son intégration au processus de modélisation, étape comprenant relativement peu de faits, et dont l'acteur privilégié est l'utilisateur, implique des temps de réponse raisonnables. Il est à noter que ce composant logiciel est un des premiers outils pratiques utilisant les expressions OCL. Il dépasse le cadre initial prévu par OCL grâce à un développement original qui s'appuie sur l'inversion des domaines de validité : il devient un moteur de propositions valides.

L'avant-dernier chapitre sera consacré à la probl ématique de la gestion des discontinuités de modèles dans le cadre de la simulation des régimes transitoires. Contrairement au concept de domaine de validité, lié au modèle de données, l'expertise dans ce cas est très proche de l'algorithmique de résolution. Or la formalisation dans le langage UML des aspects dynamiques est d'usage nettement moins répandu dans la communauté des programmeurs que la modélisation des structures de données objet. En conséquence, dans ce chapitre, notre approche tend à plus de pragmatisme, au détriment de la conceptualisation. Enfin, nous aborderons les conclusions générales relatives au travail effectué, et les prospectives qui s'en dégagent.

Assistance dynamique à la Modélisation

Sommaire

L'objectif de ce chapitre est de concevoir un outil permettant l'exploitation des domaines de validité afin de guider dynamiquement l'utilisateur dans la modélisation de dispositifs. Nous allons tout d'abord établir les spécifications d'un logiciel de simulation traitant les principales formulations électromagnétiques dans le cadre quasi-stationnaire. Nous exprimerons sur le modèle de données UML développé les expressions OCL traduisant les contraintes d'utilisation des formulations. Nous illustrerons par quelques exemples de dispositifs électrotechniques concrets de quels manières nous entendons exploiter les expressions OCL introduites. Enfin nous présenterons l'implantation de notre outil, et sa validation sur les exemples précédents.

1 Exemple d'application en électromagnétisme

1.1 Les formulations électromagnétiques et leurs domaines de validité

Nous allons introduire dans ce paragraphe les principales formulations en potentiels, issues des équations de Maxwell quasi-stationnaires. Pour chaque formulation, nous détaillerons ses contraintes d'utilisation.

1.1.1 Les équations de Maxwell quasi-stationnaires

Les équations de Maxwell en régime quasi-stationnaire sont utilisées principalement pour résoudre le problème type illustré sur la Figure 1. Soit $\Omega_0[\mu_0]$ un volume d'air. A l'intérieur de ce volume se trouve des matériaux magnétiques ($\Omega_1[\mu_1]$) éventuellement conducteurs ($\Omega_2[\mu_2;\sigma_2]$), des sources ayant une densité de courant imposée j_S et des aimants ($\Omega_3[B_R]$). Enfin, aux limites du domaine,

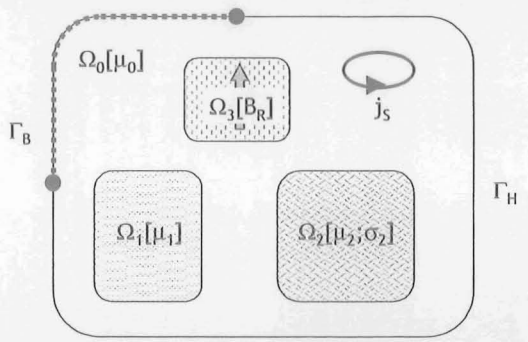


Figure 1 : Le problème type

Les propriétés des champs magnétiques et électriques sont décrites par les équations de Maxwell simplifiées pour les basses fréquences, en négligeant les courants de déplacement. Elles s'écrivent sous la forme de :

$$\text{Loi d'Ampère :} \quad \text{rot}\,\vec{H} = \vec{j}_S + \vec{J} \quad \text{dans } \Omega \qquad (1)$$

$$\text{Loi de Faraday :} \quad \text{rot}\,\vec{E} = -\frac{\partial \vec{B}}{\partial t} \quad \text{dans } \Omega \qquad (2)$$

où B est l'induction magnétique, H le champ magnétique, E le champ électrique, j_S la densité de courant imposée et J la densité de courant induit. Nous nous placerons dorénavant dans le cadre magnétodynamique, où les courants sont supposés sinusoidaux et les matériaux linéaires. Alors la loi de Faraday (2) s'exprime sous la forme :

$$\text{rot}\,\vec{E} = -j\omega\vec{B} \quad \text{dans } \Omega \qquad (3)$$

où ω est la pulsation. Des équations (1) et (3), on déduit immédiatement que l'induction B et la densité de courant induit J sont à flux conservatif:

$$\mathrm{div}(\vec{B}) = 0 \qquad \text{dans } \Omega \qquad (4)$$

$$\mathrm{div}(J) = 0 \qquad \text{dans } \Omega \qquad (5)$$

A ces deux équations on ajoute les lois de comportement des matériaux :

$$\vec{B} = \mu \vec{H} \qquad \text{dans } \Omega \qquad (6)$$

$$\vec{J} = \sigma \vec{E} \qquad \text{dans } \Omega \qquad (7)$$

La perméabilité magnétique μ est généralement une propriété anisotrope et dépendant du champ H pour les matériaux non-linéaires. De même pour la conductivité électrique σ par rapport au champs électrique E. D'autre part, les aimants sont caractérisés par leur induction rémanente B_R. Pour ce type de matériaux, l'équation (6) est remplacée par :

$$\vec{B} = \mu\vec{H} + \vec{B}_R \qquad \text{dans } \Omega \qquad (8)$$

Les équations (1) à (8) régissent les comportements magnétiques et électriques de nos dispositifs. Il faut maintenant définir les condition aux limites du domaine :

$$\vec{B}.\vec{n} = 0 \qquad \text{sur } \Gamma_B \qquad (9)$$

$$\vec{H} \wedge \vec{n} = \vec{0} \qquad \text{sur } \Gamma_H \qquad (10)$$

A l'interface $\Gamma_{1\text{-}2}$ entre deux milieux Ω_1 et Ω_2 aux propriétés physiques différentes, et où on supposera l'absence de courant de surface, on a :

$$[\vec{H}_1 - \vec{H}_2] \wedge \vec{n} = \vec{0} \qquad \text{sur } \Gamma_{1\text{-}2} \qquad (11)$$

$$[\vec{B}_1 - \vec{B}_2].\vec{n} = 0 \qquad \text{sur } \Gamma_{1\text{-}2} \qquad (12)$$

$$[\vec{E}_1 - \vec{E}_2] \wedge \vec{n} = \vec{0} \qquad \text{sur } \Gamma_{1\text{-}2} \qquad (13)$$

$$[\vec{J}_1 - \vec{J}_2].\vec{n} = 0 \qquad \text{sur } \Gamma_{1\text{-}2} \qquad (14)$$

Certaines composantes des variables d'états utilisées dans ces équations présentent une forte discontinuité aux interfaces entre deux milieux. Or, l'utilisation de la méthode des éléments finis nodaux traditionnelle implique la continuité des grandeurs modélisées [3.1.2]. Afin de pouvoir utiliser cette méthode numérique malgré tout, nous allons transformer ces équations en champs en formulations en potentiels. Les nouvelles variables d'états introduites alors ont la propriété d'être continues. Notons au passage que les éléments finis dits "d'arêtes" permetent de modéliser des variables discontinues [3.1.4].

1.1.2 Formulations en potentiel vecteur magnétique

a- Formulation AV

L'induction magnétique B est à flux conservatif (4). On en déduit donc qu'elle dérive d'un champ de vecteurs dit " potentiel vecteur magnétique", noté A et défini à un gradient près par la formule :

$$\vec{B} = \mathrm{rot}\,\vec{A} \qquad \text{dans } \Omega \qquad (15)$$

En substituant B par sa valeur dans la loi de Faraday en régime harmonique (3) , on déduit que le champ électrique E est la somme du potentiel vecteur magnétique A et du gradient d'une fonction scalaire V nommée "potentiel scalaire électrique", définie à une constante près par la formule :

$$\vec{E} = -j\omega \vec{A} - \text{grad}\, V \qquad \text{dans } \Omega \qquad (16)$$

A partir de la loi d'Ampère (1), par substitutions successives des variables d'état H, J, B et E grâce aux relations (6), (7), (15) et (16), on obtient l'équation :

$$\text{rot}\left(\frac{1}{\mu}\text{rot}\,\vec{A}\right) + j\omega\sigma\,\vec{A} + \sigma\,\text{grad}\,V = \vec{0} \qquad \text{dans } \Omega \qquad (17)$$

Nous avons vu que les potentiels A et V n'étaient pas définis de manière unique. Pour obtenir l'unicité de la solution, on doit fixer la valeur des potentiels vecteur magnétique A et scalaire électrique V en au moins un point du domaine Ω [Coulomb-81]. Pour cela, on introduit la jauge de Coulomb (div A = 0) dans l'équation (17) sous la forme d'un terme de pénalité. Le système à résoudre devient :

$$(S_{\vec{A}V}) \begin{cases} \text{rot}\left(\frac{1}{\mu}\text{rot}\,\vec{A}\right) + j\omega\sigma\,\vec{A} - \text{grad}\left(\frac{1}{\mu}\text{div}\,\vec{A}\right) + \sigma\,\text{grad}\,V = \vec{0} \\ \text{div}\left(\sigma\left(j\omega\vec{A} + \text{grad}\,V\right)\right) = 0 \end{cases} \qquad (18)$$

L'unicité du potentiel scalaire électrique V est obtenu en fixant sa valeur en un point. En substituant B par (15) et en utilisant le produit mixte, la condition limite (9) sur un mur électrique se traduit aisément par :

$$\vec{A} \wedge \vec{n} = \vec{0} \quad \text{sur } \Gamma_B \qquad (19)$$

De plus un rapide calcul démontre que la continuité de A et de V implique celle des composantes normale de l'induction B (12) et tangentielle du champ électrique E (13) à l'interface entre deux milieux aux caractéristiques physiques différentes.

Enfin, les conditions limites (9) et (10) ainsi que les conditions de passage (11) et (14) devront donc être assurées sous forme intégrale dans la forme faible de (S_{av}).

Cette formulation est la plus générale. Cependant son emploi est coûteux, car elle utilise simultanément deux inconnues, vectorielle et scalaire. Traitée par la méthode des éléments finis nodaux, cette formulation est donc caractérisée par quatre degrés de liberté par nœuds. Enfin, l'utilisation de la formulation AV suppose la présence de courants induits ($\sigma>0$).

b- Formulation A*

Un nouveau potentiel peut être défini à partir du potentiel vecteur magnétique A et du potentiel scalaire électrique V : le potentiel vecteur modifié A* qui, en magnétodynamique, est proportionnel au champ électrique E :

$$\vec{E} = \frac{\partial \vec{A}}{\partial t} + \text{grad}\,V = \frac{\partial \vec{A}^*}{\partial t} = -j\omega\vec{A}^* \qquad (20)$$

En partant de la loi d'Ampère (1), et en substituant H, J, B et E, on obtient la formulation A* :

$$(S_{\vec{A}^*}) \left\{ \text{rot}\left(\frac{1}{\mu}\text{rot}\,\vec{A}^*\right) + j\sigma\omega\,\vec{A}^* = \vec{0} \right. \qquad (21)$$

Cette formulation hérite du même comportement vis-à-vis des conditions limites et des conditions de passage que la formulation AV. On remarque au passage qu'elle ne nécessite pas l'emploi d'une jauge, puisque la variable d'état est directement liée au champ électrique E.

L'avantage de cette formulation par rapport à la formulation AV est évidemment son moindre coût, puisqu'elle n'utilise qu'une seule inconnue vectorielle. Cependant son domaine d'application est plus restreint, car il impose un milieu conducteur continu ([σ]=0 en tout point du domaine).

Comme la formulation AV, la formulation en potentiel vecteur modifié A* est utilisée dans les régions nécessitant la prise en compte de courants induits (σ>0).

c- Formulation A

Dans le cas où aucun courant de Foucault ne se développe dans le matériau, le milieu peut alors être considéré de notre point de vue comme un isolant électrique (σ=0). Dans ces conditions, les seules densités de courant définies dans ces régions sont connues et imposées.

L'induction magnétique B, toujours à flux conservatif (4), dérive du potentiel vecteur magnétique A à un gradient près. La loi d'Ampère et l'équation (6) donne rapidement la formulation A avec condition de jauge de Coulomb :

$$\mathrm{rot}\left(\frac{1}{\mu}\mathrm{rot}\,\vec{A}\right) - \mathrm{grad}\left(\frac{1}{\mu}\,\mathrm{div}\,\vec{A}\right) = \vec{j}_S \quad \text{dans } \Omega \qquad (22)$$

La formulation A est la plus générale dans les milieux isolants, mais elle est coûteuse par rapport à d'autres formulations que nous allons introduire maintenant. De plus, elle possède un mauvais comportement dans les arêtes et les coins des régions fortement perméables. Dans ce cas, l'induction B est repoussée à l'intérieur du dispositif [3.1.5].

1.1.3 Formulations en potentiel scalaire magnétique

Les formulations utilisant le potentiel scalaire magnétique sont duales des formulations en potentiel vecteur magnétique. Cependant, leur domaine d'utilisation est restreint à cause du problème bien connu dit « de connexité ». Ce dernier sera abordé après avoir introduit les trois formulations en potentiel scalaire magnétique .

a- Formulation TΦ

De manière tout à fait duale à la formulation AV, on part du fait que la densité de courant J soit à flux conservatif (5) pour établir qu'elle dérive d'un champ de vecteurs nommé "potentiel vecteur électrique" défini à un gradient près, et noté T :

$$\vec{J} = \mathrm{rot}\,\vec{T} \quad \text{dans } \Omega \qquad (23)$$

Puis, grâce à la loi d'Ampère, on introduit le potentiel scalaire magnétique Φ défini à une constante près tel que :

$$\vec{H} = \vec{T} - \mathrm{grad}\,\Phi \quad \text{dans } \Omega \qquad (24)$$

En suivant le même raisonnement fait pour la formulation AV, on obtient le système à résoudre en TΦ, avec la condition de jauge de Coulomb (div T=0) :

$$(S_{\vec{T}\Phi}) \begin{cases} \mathrm{rot}(\frac{1}{\sigma}\mathrm{rot}\,\vec{T}) - \mathrm{grad}(\frac{1}{\sigma}\mathrm{div}\,\vec{T}) + j\omega\mu\,\vec{T} - j\omega\mu\,\mathrm{grad}\,\Phi = \vec{0} \\ \mathrm{div}(j\omega\mu\,(\vec{T} - \mathrm{grad}\,\Phi)) = 0 \end{cases}$$

et la condition limite (10) qui s'exprime de la manière suivante sur le potentiel T :

$$\vec{T} \wedge \vec{n} = \vec{0} \quad \text{sur } \Gamma_H \tag{25}$$

Dans ce système, la continuité des potentiels T et Φ assurent la continuité des composantes normale de la densité de courant J (14) et tangentielle du champ magnétique H (11). La condition limite (9) ainsi que les conditions de passage (12) et (13) seront assurées sous forme intégrale dans la forme faible de ($S_{T\Phi}$).

Cette formulation est utilisée dans les régions où se développent des courants induits, et a le même coût que la formulation AV, puisqu'elle utilise également une inconnue vectorielle et une inconnue scalaire.

b- Formulation Φ

Lorsque qu'il n'existe aucune source de courant dans une région isolante, le rotationnel du champ magnétique H y est partout identiquement nul, ce qui signifie qu'il dérive d'un potentiel scalaire magnétique dit "total" Φ défini à une constante près :

$$\vec{H} = -\mathrm{grad}\,\Phi \quad \text{dans } \Omega \tag{26}$$

On résout alors l'équation (5), ce qui donne la formulation Φ :

$$\mathrm{div}\,(\mu\,\mathrm{grad}\,\Phi) = 0 \quad \text{dans } \Omega \tag{27}$$

L'unicité de la solution est obtenue grâce aux conditions limites, ou par couplage avec une région qui fixe la valeur du potentiel scalaire magnétique.

La formulation Φ, utilisable uniquement dans les régions isolantes, a un très faible coût par rapport à la formulation A puisqu'elle n'utilise qu'une inconnue scalaire. Cependant, elle ne prend pas en compte les sources de courants.

c- Problème de connexité des formulations en Φ

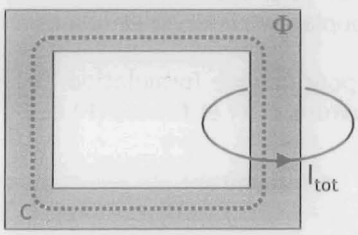
Les formulations en potentiel scalaire magnétique total sont notablement moins coûteuse que les formulations en potentiel vecteur magnétique, surtout dans le cas des régions isolantes. En effet, dans le premier cas, l'inconnue est scalaire, et dans le second elle est vectorielle.

Cette différence est d'autant plus critique que la modélisation des dispositifs électromagnétiques nécessite généralement l'utilisation d'une grande boite d'air englobant. Un souci d'économie favoriserait donc naturellement le choix des formulations en potentiel scalaire magnétique au détriment des formulations en potentiels vecteur magnétique.

Cependant, il subsiste un problème intrinsèquement lié à l'utilisation du potentiel scalaire magnétique total, aussi bien dans les régions conductrices qu'isolantes. Le théorème d'Ampère affirme que la circulation du champ magnétique H le long d'un contour fermé C doit être égal au courant total I_{tot} qui traverse ce contour :

$$\text{Théorème d'Ampère :} \quad \int_C \vec{H}.d\vec{l} = I_{tot} \tag{28}$$

Les formulations utilisant comme variable d'état le potentiel scalaire total ne peuvent pas prendre en compte les circuits magnétiques non simplement connexes, car alors le théorème d'Ampère n'est plus respecté (Figure 2). Pour résoudre ce problème, on introduit le potentiel scalaire réduit.



$$\int_C H.dl = \int_C -\operatorname{grad}\Phi.dl = 0 \neq I_{tot}$$

Figure 2 : Violation du théorème d'Ampère

d- Formulation Φ_r

On suppose cette fois qu'il existe une densité de courant j_s connue et imposée dans une région de type air, c'est à dire isolante non perméable. On va alors décomposer le champ magnétique H en deux parties : $H = H_J + H_R$ où H_J est le champ magnétique dû aux densités de sources de courant présentes j_S , et calculé par la loi de Biot et Savart :

$$\text{Loi de Biot et Savart :} \quad \vec{H}_J(P) = \frac{1}{4\pi} \iiint_{Q\in\Omega} \vec{j}_S(Q) \wedge \frac{\overrightarrow{PQ}}{\|\overrightarrow{PQ}\|^3} d\Omega \quad \forall P \in \Omega \qquad (28)$$

Le champ H_R est le champ réduit, c'est à dire le champ de réaction dû à la présence de matériaux ferromagnétiques et aux régions conductrices. Le rotationnel de H_J valant j_S, il est aisé de conclure que H_R dérive d'un potentiel scalaire magnétique dit "réduit" Φ_R [3.1.8]. D'où finalement la formulation Φ_R :

$$\operatorname{div}(\mu \operatorname{grad}\Phi_R) = \operatorname{div}(\mu \vec{H}_J) \quad \text{dans } \Omega \qquad (29)$$

La formulation Φ_R est plus générale que la formulation précédente, puisqu'elle rend possible la prise en compte des sources de courants présentes dans la région isolante. Elle est également très économique, mais nécessite cependant le calcul du champ H_J produit par les sources.

Dans le cas où la perméabilité du milieu est élevée cependant, cette formulation est inutilisable. En effet, dans ce cas, le champ réduit H_R et le champ calculé H_J sont du même ordre de grandeur, mais de sens opposé. Le champ total H est alors noyé dans le bruit numérique.

1.1.4 Récapitulatif des domaines de validité et couplages

Un modèle de dispositif ne met pas en œuvre une seule formulation, mais généralement plusieurs en couplage. On peut classer ces derniers en fonction de la nature des variables d'état respectives des formulations à coupler.

Si les deux formulations partagent les mêmes variables d'états, le couplage est qualifié de naturel, car il ne nécessite pas d'équation supplémentaire au niveau de l'interface pour être pris en compte.

Dans le cas où les deux formulations ne partagent pas de variables d'état, il peut cependant exister une relation simple entre elles, notamment une conservation de flux. Ce type de couplage est qualifié d'intégral.

Enfin, dans tous les autres cas, un traitement spécifique du couplage doit être entrepris à l'interface entre les deux formulations. On peut citer notamment des techniques d'introduction de degrés de libertés supplémentaires pour assurer des sauts de potentiel. Dans le cadre de notre exemple, nous supposerons ces couplages irréalisables, car non implantés.

Le tableau ci-dessous récapitule pour chaque formulation l'ensemble des contraintes (C) qui y est attaché, ainsi que les couplages naturels (N) et faibles (F) entre formulations :

	Contraintes							Couplages					
	simplement connexe	pas de source	Courants induits ($\sigma>0$)	Pas de courant induit ($\sigma=0$)	$\sigma_1=\sigma_2$	Perméabilité faible ($\mu<1000$)	Φ	Φ_r	TΦ	A	Av	A*
Φ	C	C		C			N	N	N			F
Φ_r				C		C	N	N	N			F
TΦ	C		C				N	N	N			
A				C		C				N	N	
Av			C							N	N	
A*			C		C		F	F				N

Tableau 1 : Différentes contraintes associées aux formulations

Nous précisons qu'il existe d'autres contraintes sur les formulations, en particulier liées à la présence de coins rentrant ou sortant dans les dispositifs utilisant les formulations TΦ ou A en éléments finis nodaux. Pour des questions de simplicité de la démonstration, ces contraintes ne sont pas exprimées dans ce tableau.

1.2 *Modèle de données UML et ses contraintes*

Le développement des équations de Maxwell quasi-stationnaires dans le cadre de la théorie des éléments finis nodaux sans saut de potentiel met à notre disposition un ensemble de formulations. Ces dernières sont mises en œuvre dans un outil de simulation dont nous allons maintenant préciser un modèle de données objet simplifié grâce au formalisme UML. Ce modèle spécifiera à la fois les aspects structurel et contractuel de l'application.

1.2.1 Modèle de données simplifié

A titre d'exemple, nous allons mettre en place un modèle de données UML simplifié d'un logiciel de simulation élément fini en électromagnétisme, pour la mise en œuvre des formulations que nous avons introduites précédemment.

Nous insistons sur le fait que le modèle de données UML proposé ne représente qu'une partie du modèle de données total de l'application, si nous avions pour objectif son implantation opérationnelle. Le but que nous recherchons à travers ce diagramme de classes est de disposer des principaux concepts utiles à la modélisation d'un dispositif électromagnétique.

Pour concevoir le modèle objet de l'application, nous avons utilisé un atelier de génie logiciel. Ce dernier nous a permis de concevoir rapidement notre modèle de données simplifié en UML. A partir de ce dernier, l'atelier génère et compile les classes java correspondantes.

Dans notre modèle de données, l'entité principale est le **Problème**. Un problème est composé de **Domaine**. Le domaine est la structure qui localise un phénomène physique particulier (attribut « phénomène »), modélisé par une **Formulation**, sur le support géométrique (attribut « support ») où il se produit.

Notre modèle de données distingue deux types de domaines : les **régions** et les **interfaces**[1]. Cette distinction se fonde sur le type de phénomène physique associé à chaque type de domaine. La région est toujours associée à une formulation de type **régionale**, alors que l'interface est quant à elle associée à une formulation de type **interfaciale**, correspondante à des conditions de passages ou des couplages faibles entre deux régions, ou encore à des conditions limites de type **Dirichlet** ou **Neumann**.

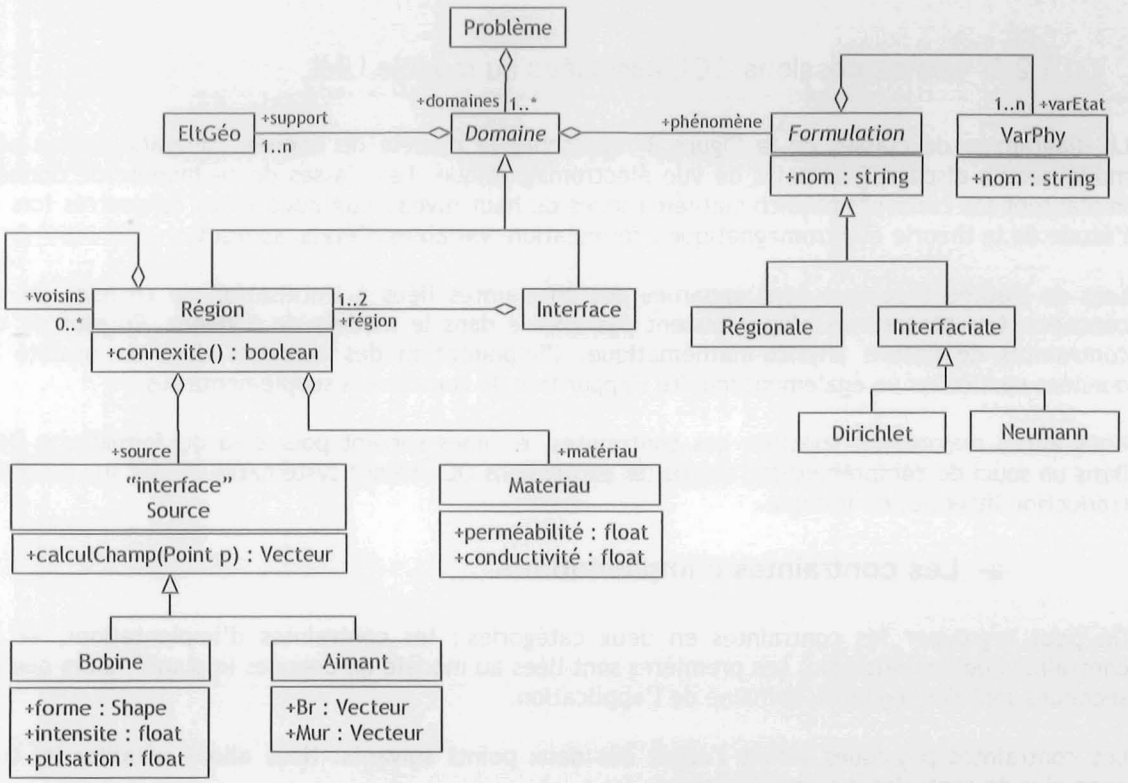


Figure 3 : Modèle de données UML simplifié d'un logiciel élément fini de simulation électromagnétique

A chaque Formulation est associé son nom et ses variables d'état (attribut « varEtat » de type VarPhy). Le diagramme de la Figure 4 ci-dessous représente par exemple une instance de la classe Formulation à laquelle est associée deux instances de la classe VarPhy, qui sont ses variables d'état.

L'ensembles des régions adjacentes à une région particulière sont spécifiées dans son attribut « voisins ». Les régions qui participent à une interface sont accessibles via l'attribut « région » de celle-ci. Ces deux attributs définissent la topologie des domaines du problème.

Chaque région doit être associée à un **matériau**, caractérisé par sa **perméabilité** et sa **conductivité**. Ces propriétés sont supposées constantes et isotropes. De plus, une région peut

[1] Rem : Ne pas confondre la classe « Interface » avec le mot-clé « interface ».

également être associée à une **source** de champ. Les seules sources considérées sont soit les **bobines** parcoures par un courant sinusoïdal, soit les **aimants** permanents. 'opération « connexité », appliquée à une région, renvoie un booléen indiquant si le support géométrique de cette région est simplement connexe (vrai) ou non (faux). Cette opération est sans effet de bord.

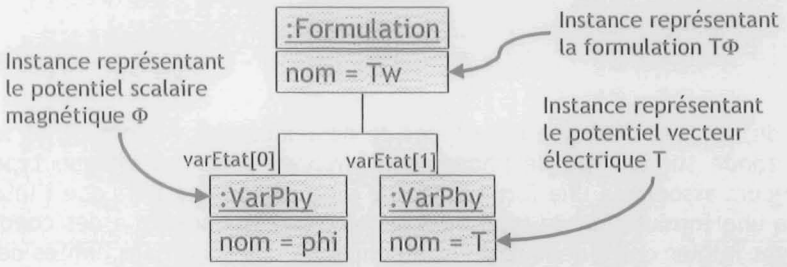


Figure 4 : Exemple d'instance de Formulation

1.2.2 Les expressions OCL associées au modèle UML

Le diagramme de classes de la Figure 3 représente le modèle de données simplifié utilisé pour modéliser un dispositif du point de vue électromagnétique. Les classes de ce modèle de données implantent les concepts physico-mathématiques de haut niveau que nous avons rencontrés lors de l'étude de la théorie électromagnétique : formulation, variables d'états, sources,...

Lors de l'étude théorique sont apparues des contraintes liées à l'utilisation de certains de ces concepts. Ces contraintes n'apparaissent pas encore dans le modèle de données. En plus de ces contraintes de nature physico-mathématique, l'implantation des concepts dans un modèle de données particulier va également induire l'apparition de contraintes supplémentaires.

Nous allons maintenant spécifier ces contraintes, en nous servant pour cela du formalisme OCL. Dans un souci de compréhension, toutes les expressions OCL seront systématiquement suivies d'une traduction littérale, en italique.

a- Les contraintes d'implantations

On peut regrouper les contraintes en deux catégories : les contraintes d'implantations, et les contraintes de modélisation. Les premières sont liées au modèle de données implanté, alors que les secondes sont des règles du domaine de l'application.

Les contraintes physiques seront l'objet des deux points suivants. Nous allons détailler ici trois exemples de contraintes d'implantations.

Un exemple de contrainte d'implantation est la **multiplicité** du rôle d'une association. On peut citer par rapport à notre application le fait que le support géométrique d'un domaine soit composé d'au moins un élément :

```
context Domaine inv support_defini :
  self.support -> notEmpty
```

Soit « self » un domaine. Son support géométrique (attribut « support ») doit être défini (non vide).

Ou encore, que toute formulation est associée à au moins une variable d'état :

```
context Formulation inv var_etat_defini :
  self.varEtat -> notEmpty
```

Soit « self » une formulation. L'ensemble de ses variables d'état (attribut « varEtat ») doit être non vide.

Un dernier exemple de contrainte d'implantation est le fait qu'une région ne puisse par être sa propre voisine. Avant l'intégration d'OCL au standard UML, l'unique possibilité offerte pour spécifier une telle contrainte était de l'écrire en langage libre, entre crochet, dans le diagramme UML. Avec OCL, nous pouvons écrire :

```
context Region inv est_sa_propre_voisine :
  self.voisins -> forAll( elt | elt <> self )
```

Soient « self » une région. Pour toute région notée « elt » appartenant au voisinage de « self » (attribut « voisisns »), « elt » doit être différent de « self ».

Une dernière contrainte d'implantation des formulations dans notre modèle de données est le distingo établi entre région et interface. Au premier type de domaine sont associées uniquement des formulations dites régionales, et au second uniquement des formulations dites interfaciales.

```
context Region inv formulation_regionale :
  self.phénomène.oclIsTypeOf(Regionale)
```

Soient « self » une région. Le phénomène physique associé (attribut « phénomène ») doit être modélisé par une formulation de type régionale.

```
context Interface inv formulation_interfaciale :
  self.phénomène.oclIsTypeOf(Interfaciale)
```

Soient « self » une interface. Le phénomène physique associé doit être modélisé par une formulation de type interfaciale.

b- Les contraintes de modélisation

i- Les contraintes associées au matériau

Tout d'abord, le modèle de matériau utilisé ici est tout à fait simple, puisqu'il s'agit d'un matériau caractérisé par ses propriétés de perméabilité relative et de conductivité électrique. Ces dernières étant supposées constantes et isotropes dans tout le matériau, elles sont alors modélisées par une valeur de type 'float'.

Il existe une contrainte physique associée à chacune des propriétés précédentes, concernant leur valeur. La perméabilité relative est toujours supérieure ou égale à l'unité, et la conductivité est positive. Ces contraintes sont exprimées par les invariants de classe suivants :

```
context Materiau inv regle_No_1 :
  self.permeabilite >= 1
```

Soit « self » un matériau. Sa perméabilité doit être de valeur supérieure ou égale à un.

```
context Materiau inv regle_No_2 :
  self.conductivite >= 0
```

Soit « self » un matériau. Sa conductivité doit être de valeur supérieure ou égale à zéro.

A titre de remarque, soulignons comment une hypothèse forte (matériau constant isotrope) est traduite par un choix au niveau du modèle de données (attributs « perméablité » et « conductivité » de la classe Matériau de type 'float').

Nous pourrions atteindre une plus grande liberté de modélisation en enrichissant notre modèle de données de matériaux. Cet enrichissement s'accompagnerait parallèlement de l'apparition de nouvelles contraintes d'utilisation.

ii- Les contraintes associées aux formulations régionales

Intéressons nous maintenant aux domaines d'utilisation des formulations, qui sont récapitulés dans le **Tableau 1** de la page 9. Chaque formulation que nous avons développée dans le paragraphe précédent est associée à une instance particulière de la classe Régionale. Le tableau ci-dessous établit la correspondance entre chaque formulation et le nom de l'instance correspondante :

Formulation	Nom de l'instance
AV	AV
A	pot_vect
A*	pot_vect_mod
TΦ	Tw
Φ	pot_sca_tot
Φ_R	pot_sca_red

Tableau 2 : Formulations régionales et instances correspondantes

La première contrainte exprimée dans la première colonne du tableau (colonne « simplement connexe ») est celle liée au problème de connexité. Elle s'exprime en OCL par l'invariant de classe suivant :

```
context Region inv regle_de_connexite :
  self.phénomène.varEtat -> exists( v | v.nom = "phi") implies implies self.connexite()
```

Soit «self» une région. Si la formulation associée à la région « self » utilise le potentiel scalaire total ("phi"), alors la region «self» doit être simplement connexe.

Autrement dit, toute région associée à une formulation régionale utilisant le potentiel scalaire magnétique total ϕ doit être simplement connexe.

La deuxième règle exprime le fait que la formulation régionale en potentiel scalaire magnétique total ϕ ne prend pas en compte les sources de courant, c'est à dire dans notre cas les bobines. Ce qui est traduit par l'invariant de classe :

```
context Region inv regle_bobine :
  self.phénomène.nom = "pot_sca_tot" implies
  self.sources -> forAll( s | not s.oclIsTypeOf( Bobine) )
```

Soit «self» une région. Si la région «self» est associée à la formulation ϕ (cf. Tableau 2), alors aucune source de la région ne doit être de type «Bobine».

Les troisième et quatrième contraintes déterminent le type de formulation régionale adéquate en fonction de la nature du matériau, et plus précisément si des courants induits sont supposés s'y développer. Dans ce cas, le matériau sera supposé conducteur (conductivité positive strictement). Sinon il sera supposé isolant (conductivité null). Pour plus de clarté dans l'écriture de la contrainte OCL associée à cette règle physique, nous avons choisi de l'écrire sous la forme de quatre invariants de classe :

```
context Region inv absence _de_courant_induit_1 :
  ( self.materiau.conductivite = 0 implies
      ( self.phénomène.nom = "pot_sca_tot" or
        self.phénomène.nom = "pot_sca_red" or
        self.phénomène.nom = "pot_vect" ) )
```

L'absence de courant induit dans le matériau (conductivité = 0) associé à la région « self » implique que cette région doit être associée aux formulations régionales Φ, Φ_R ou A.

```
context Region inv absence_de_courant_induit_2 :
 ( ( self.phénomène.nom = "pot_sca_tot" or
     self.phénomène.nom = "pot_sca_red" or
     self.phénomène.nom = "pot_vect" ) implies
   self.materiau.conductivite = 0 )
```

Si la région est associée à l'une des formulations Φ, Φ_R ou A, alors il ne se développe pas de courant induit dans son matériau (conductivité nulle).

```
context Region inv présence_de_courant_induit_1 :
( self.materiau.conductivite > 0 implies
     ( self.phénomène.nom = "Tw" or
       self.phénomène.nom = "AV" or
       self.phénomène.nom = "pot_vect_mod" ) )
```

La prise en compte des courants induits dans le matériau (conductivité > 0) associé à la région »self » implique que celle dernière soit associée aux formulations régionales Tw, AV ou A.*

```
context Region inv présence_de_courant_induit_2 :
 ( ( self.phénomène.nom = "Tw" or
     self.phénomène.nom = "AV" or
     self.phénomène.nom = "pot_vect_mod" ) implies
   self.materiau.conductivite > 0 )
```

Si la région est associée à l'une des formulations Tw, AV ou A, alors son matériau est supposé conducteur (conductivité > 0) pour prendre en compte les courants induits.*

L'avant-dernière colonne des contraintes est relative exclusivement à la formulation en potentiel vecteur modifié A*. Elle précise que son utilisation est restreinte à un ensemble de régions conductrices adjacentes ayant la même valeur de conductivité, ceci afin d'exclure des sauts de conductivité :

```
context Region inv regle_saut_conductivite :
 self.phénomène.nom = "pot_vect_mod" implies
 ( self.voisins -> select( r | r.phénomène.nom = "pot_vect_mod" )
                -> forAll( r | self.materiau.conductivite = r.materiau.conductivite) )
```

Soit «self» une région. Si la région «self» est associée à la formulation A (cf. Tableau 2) alors pour toute région (notée «r») voisine de la région «self» et associée également à la formulation A*, la valeur de la conductivité du matériau de la région « self » doit être égale à celle du matériau de la région « r ».*

En d'autre termes, deux régions conductrices adjacentes peuvent être associées à la formulation en potentiel vecteur modifié A* si et seulement si la valeur de la conductivité des deux régions est la même.

Enfin, la dernière contrainte associée aux formulations en potentiel scalaire magnétique réduit Φ_R et au potentiel vecteur magnétique A porte sur leur limite d'utilisation dans les régions fortement perméables :

```
context Region inv règle_permeabilité_élevée :
 ( self.phénomène.nom = "pot_sca_red" or
   self.phénomène.nom = "A" ) implies self.materiau.permeabilite < 1000
```

Soit «self» une région. Si la région «self» est associée aux formulations Φ_R (cf. Tableau 2) ou A, alors sa perméabilité doit être inférieure à 1000.

La deuxième partie du **Tableau 1** récapitule les couplages possibles entre deux formulations appartenant à deux régions adjacentes. Les seuls couplages implantés sont d'une part les couplages naturels entre deux formulations ayant une variable d'état commune, et d'autre part le couplage faible entre les formulations en potentiel vecteur modifié A* et les formulations en potentiel scalaire, total Φ ou réduit Φ_R :

```
context Region inv couplage_pot_sca :
  self.phénomène.nom = "pot_sca_tot" implies
  self.voisins -> forAll( v | v.phénomène.nom = "pot_sca_tot" or
                              v.phénomène.nom = "pot_sca_red" or
                              v.phénomène.nom = "Tw" or
                              v.phénomène.nom = "pot_vect_mod" )
```

Soit « self » une région. Si la région « self » est associée à la formulation en potentiel scalaire total Φ, alors ses régions voisines sont associées à l'unes des formulations suivantes : Tw, en potentiels scalaires, total Φ et réduit Φ_R, et en potentiel vecteur modifié A.*

```
context Region inv couplage_pot_red :
  self.phénomène.nom = "pot_sca_red" implies
  self.voisins -> forAll( v | v.phénomène.nom = "pot_sca_tot" or
                              v.phénomène.nom = "pot_sca_red" or
                              v.phénomène.nom = "Tw" or
                              v.phénomène.nom = "pot_vect_mod" )
```

Soit « self » une région. Si la région « self » est associée à la formulation en potentiel scalaire réduit Φ_R, alors ses régions voisines sont associées à l'unes des formulations suivantes : Tw, en potentiels scalaires, total Φ et réduit Φ_R, et en potentiel vecteur modifié A.*

```
context Region inv couplage_Tw :
  self.phénomène.nom = "Tw" implies
  self.voisins -> forAll( v | v.phénomène.nom = "pot_sca_tot" or
                              v.phénomène.nom = "pot_sca_red" or
                              v.phénomène.nom = "Tw" )
```

Soit « self » une région. Si la région « self » est associée à la formulation Tw, alors ses régions voisines sont associées à l'unes des formulations suivantes : Tw et en potentiels scalaires, total Φ et réduit Φ_R.

```
context Region inv couplage_pot_vect :
  self.phénomène.nom = "pot_sca_vect" implies
  self.voisins -> forAll( v | v.phénomène.nom = "pot_vect" or
                              v.phénomène.nom = "AV " )
```

Soit « self » une région.Si la région « self » est associée à la formulation en potentiel vecteur A, alors ses régions voisines sont associées à l'unes des formulations suivantes : AV ou en potentiel vecteur A.

```
context Region inv couplage_AV :
  self.phénomène.nom = "AV" implies
  self.voisins -> forAll( v | v.phénomène.nom = "AV" or
                              v.phénomène.nom = "pot_vect" )
```

Soit « self » une région. Si la région « self » est associée à la formulation AV alors ses régions voisines sont associées à l'unes des formulations suivantes : AV ou en potentiel vecteur A.

```
context Region inv couplage_pot_vect_mod :
 self.phénomène.nom = "pot_vect_mod " implies
 self.voisins -> forAll( v | v.phénomène.nom = "pot_vect_mod " or
                             v.phénomène.nom = "pot_sca_tot" or
                             v.phénomène.nom = "pot_sca_red" )
```

Soit « self » une région. Si la région « self » est associée à la formulation en potentiel vecteur modifié A, alors ses régions voisines sont associées à l'unes des formulations suivantes : en potentiels scalaires, total Φ et réduit Φ_R, et en potentiel vecteur modifié A*.*

iii- Une contrainte de problème bien posé

Notre modèle de données nous permet de spécifier les conditions aux limites du domaine de notre problème, qui peuvent être soit de Dirichlet, soit de Neumann. Nous avons vu lors du développement de la formulation AV que l'unicité de la solution nécessite de fixer la valeur du potentiel scalaire électrique V en un point. De manière générale, un problème de modélisation de phénomènes électromagnétiques, pour être bien posé, nécessite au moins une condition de Dirichlet. L'expression OCL associée à cette contrainte est :

```
context Probleme inv cond_limite_Dirichlet :
 self.domaines -> exists ( d | d.phénomène -> oclIsTypeOf(Dirichlet) )
```

Soit « self » un problème. Il existe au moins un domaine associé à une condititon limite de Dirichlet (formulation interfaciale de type Dirichlet) dans l'ensemble des domaines définissant le problème « self ».

Bien sûr, c'est une condition nécessaire mais non suffisante. Elle n'assure pas l'existence de la solution. Dans un modèle de données plus détaillé, d'autres contraintes de ce genre pourraient assurer la compatibilité des conditions aux limites, etc...

2 Exemples d'utilisation

Nous disposons à présent des classes permettant la modélisation de dispositifs électromagnétiques, ainsi que de leurs contraintes d'utilisation exprimées sous la forme d'invariants OCL. L'objet du présent paragraphe est d'illustrer sur des exemples concrets l'aide que l'exploitation des domaines de validité est susceptible d'apporter à l'utilisateur lors de la modélisation de son dispositif.

2.1 Validation, proposition et explication

Le premier exemple de dispositif que nous prendrons est celui du transformateur (Figure 5). Il se compose de deux bobinages, enroulés autour d'un circuit ferromagnétique. Nous nous placerons dans un premier temps dans le cas où des courants induits sont susceptibles de se développer.

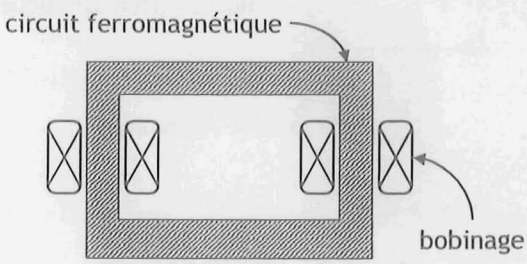


Figure 5 : le transformateur

Nous avons modélisé ce problème par deux régions correspondant respectivement au circuit ferromagnétique et à la boite d'air englobant. Supposons que l'utilisateur choisisse d'associer à ces deux régions respectivement les formulations Tw et Φ.

Nous attendons d'une expertise fondée sur les contraintes du modèle de données qu'elle invalide ce choix. En effet, non seulement la formulation en potentiel scalaire total Φ ne prend pas en compte les sources de courant, mais en plus on se trouve confronté typiquement au problème de connexité en choisissant le potentiel scalaire Φ.

Le système expert doit avertir l'utilisateur des erreurs commises, et les expliciter. Dans notre cas, il s'agit simplement de désigner à l'utilisateur la région de la boite d'air, accompagnée par exemple simplement du nom des deux contraintes OCL qu'elle viole, à savoir : regle_de_connexite et regle_bobine.

En se reportant aux invariants eux-mêmes (ou à une traduction littérale dans laquelle nous aurions remplacé le mot-clef « self » par une description de l'instance elle-même), l'utilisateur appréhende immédiatement son erreur :

```
context Region inv regle_de_connexite :
  self.phénomène.varEtat -> exists( v | v.nom = "phi" ) implies self.connexite()
```

Soit «self» une région. Si la formulation associée à la région « self » utilise le potentiel scalaire total ("phi"), alors la region «self» doit être simplement connexe.

```
context Region inv regle_bobine :
  self.phénomène.nom = "pot_sca_tot" implies
  self.sources -> forAll( s | not s.oclIsTypeOf( Bobine) )
```

Soit la région « air ». L'association de la formulation ϕ (cf. Tableau 2) à cette région implique que toutes les sources de la région doivent être de type «Bobine».

L'expertise nous permet de souligner l'endroit et la nature de l'erreur commise. Elle ne la corrige pas. Cependant, l'utilisateur peut effacer son choix malheureux (suppression de l'association « modèle » liant une formulation à une région) et laisser le soin à l'expertise de déterminer pour chaque région le modèle possible.

Dans notre cas, il n'existe que deux modélisations possibles : soit par un couplage Av-A, soit par un couplage A^*-Φ_R, chaque formulation étant associée respectivement au circuit magnétique et à l'air.

2.2 Contrainte numérique

Etudions maintenant un autre dispositif électromagnétique : l'électroaimant (Figure 6). Il se compose d'un noyau ferromagnétique entouré d'un bobinage. Lorsque qu'un courant d'intensité I parcours ce dernier, le noyau ferromagnétique s'aimante.

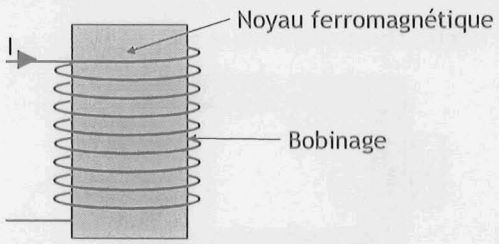


Figure 6 : L'électroaimant

Ce dispositif peut être modélisé par deux régions : une région d'air contenant l'inducteur, et la région correspondant au noyau ferromagnétique. Si le bobinage est parcouru par un courant I permanent, l'étude du dispositif se fait alors dans le cadre magnétostatique. Dans ce cas, par souci de coût, l'utilisateur va choisir de préférence les formulations en potentiel scalaire au lieu de la formulation en potentiel vecteur. Pour prendre en compte l'inducteur, il va alors affecter la formulation en potentiel scalaire réduit Φ_R aux deux régions. Or, le domaine de validité de cette formulation dépend en partie de la nature du matériau de la région :

```
context Region inv règle_permeabilité_élevée :
 ( self.phénomène.nom = "pot_sca_red" or
   self.phénomène.nom = "A" ) implies self.materiau.permeabilite < 1000
```

Soit «self» une région. Si la région «self» est associée aux formulations Φ_R (cf. Tableau 2) ou A, alors sa perméabilité doit être inférieure à 1000.

Il existe également deux contraintes numériques associées aux propriétés des matériaux. Elles spécifient que perméabilité relative doit être de valeur supérieure ou égale à l'unité, et que la conductivité électrique doit être positive ou nulle. Au sein d'une région, matériau et formulation exerce une influence réciproque. Dans notre cas, le choix de la fomulation en potentiel scalaire réduit imposera un matériau faiblement perméable.

Cet exemple illustre de quelle manière les contraintes numériques doivent être prise en compte dynamiquement en fonction des choix faits par l'utilisateur.

2.3 Conditions aux limites

Le dernier exemple de dispositif que nous considérerons est celui du contacteur électromécanique introduit au premier chapitre (Figure 7). Il se compose d'une palette mobile ayant deux positions d'équilibres privilégiées non-consommatrices d'énergie grâce à l'aimant central. Le passage du courant dans les bobines enroulées autour de la culasse permet le basculement de la palette d'une position d'équilibre à une autre.

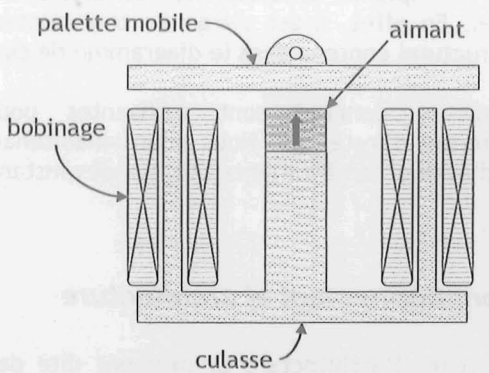


Figure 7 : le contacteur

Le dispositif sera composé de quatre régions modélisant respectivement la boite d'air englobant, la culasse, la palette mobile, l'aimant. La première région sera associée à la formulation en potentiel scalaire réduit pour prendre en compte les bobines, alors que les trois autres seront associées à la formulation en potentiel scalaire total.

Lorsque l'utilisateur a fini de décrire son problème sous la forme de ces quatre régions, l'évaluation de la validité du problème global doit lui indiquer qu'il manque un condition limite de Dirichlet :

```
context Probleme inv cond_limite_Dirichlet :
 self.domaines -> exists ( d | d.phénomène -> oclIsTypeOf(Dirichlet) )
```

Soit « self » un problème. Il existe au moins un domaine associé à une condition limite de Dirichlet (formulation interfaciale de type Dirichlet) dans l'ensemble des domaines définissant le problème « self ».

Il faudra par conséquent qu'il rajoute une condition limite de Dirichlet afin de fixer le potentiel. Le problème pourra alors être résolu.

2.4 Synthèse

La conception et le développement de logiciels de simulation numérique orientés objet nécessite l'analyse, la description et l'implantation des **concepts** du domaine d'application physique modélisé. Les diagrammes de **classes UML** permettent de réifier ces concepts en tant que classes. Les **invariants OCL** décrivent les contraintes d'utilisation de ces concepts : leur **domaine de validité**.

A partir de ces spécifications, la tâche d'implantation informatique se concentre principalement sur les méthodes algorithmes qui manipulent les structures de données. En effet, les domaines de validité sont très peu exploités, car les langages de programmation objet actuels ne prévoient ni de structures syntaxiques pour les exprimer, ni de mécanismes algorithmiques adéquats.

Dans la première partie de ce travail, nous avons montré l'utilisation du langage OCL pour exprimer les domaines de validité. Dans la seconde partie, nous allons développer un composant logiciel réutilisable permettant l'exploitation de cette information.

3 Implantation de notre système expert

Nous disposons maintenant d'un modèle de données objet UML avec ses contraintes OCL associées. A partir de ce modèle de données, un atelier de génie logiciel génère l'application correspondante sous formes d'un ensemble de classes Java. Ce langage de programmation objet a été retenu dans le cadre de notre outil, car il implante la réflexivité des objets. Nous précisons que le logiciel obtenu n'est pas fonctionnel. En effet, il est vierge de tout contenu algorithmique, les classes n'implantant que l'aspect structurel contenu dans le diagramme de classes.

Cependant, ces structures de données sont suffisantes pour modéliser un dispositif électromagnétique sous la forme d'instances. Nous allons maintenant développer notre système expert, qui sera en charge d'évaluer les invariants OCL sur des instances appartenant à un modèle de dispositif.

3.1 Principe de fonctionnement et architecture

Avant de rentrer plus avant dans l'architecture proprement dite de notre système expert, nous allons mettre en place son principe général de fonctionnement.

3.1.1 Principe de fonctionnement

Ce travail est à replacer dans le contexte général du génie logiciel objet pour la conception et le développement d'applications. C'est d'ailleurs l'objet des paragraphes précédents, dans le cadre d'un exemple de logiciel de simulation en électromagnétisme.

Le génie logiciel objet s'appuie sur la modélisation objet des applications. Nous supposerons donc que nous avons à notre disposition le modèle objet complet d'une application. Ce modèle sera composé d'une part d'un diagramme UML, et d'autre part des contraintes OCL associées.

Nous souhaitons guider l'utilisateur dans la phase de modélisation de son dispositif, non seulement pour gagner du temps lors de cette étape, mais également afin de garantir la bonne tenue de la simulation, et donc une validité des résultats.

Notre système expert s'intercale donc entre l'utilisateur et l'application elle-même. Il doit disposer non seulement d'un accès aux classes de l'application (1-Figure 8) afin de permettre la modélisation du dispositif, mais également du stockage (2-Figure 8) des instances participant au modèle en gestation. Ces instances constituent notre base de faits.

D'autre part, tout système expert est évidemment constitué d'un moteur d'inférences (3-Figure 8). Sa base de connaissances est constituée du modèle objet complet de l'application (4-Figure 8), à savoir son diagramme UML et les expressions OCL associées. Le moteur d'inférences doit évidemment avoir accès à la base de faits (5-Figure 8).

Remarquons que notre système expert est totalement indépendant de l'application objet dont l'utilisateur souhaite se servir. L'intérêt de cette indépendance est de pouvoir fournir un composant logiciel d'aide totalement réutilisable, sans coût de développement ultérieur.

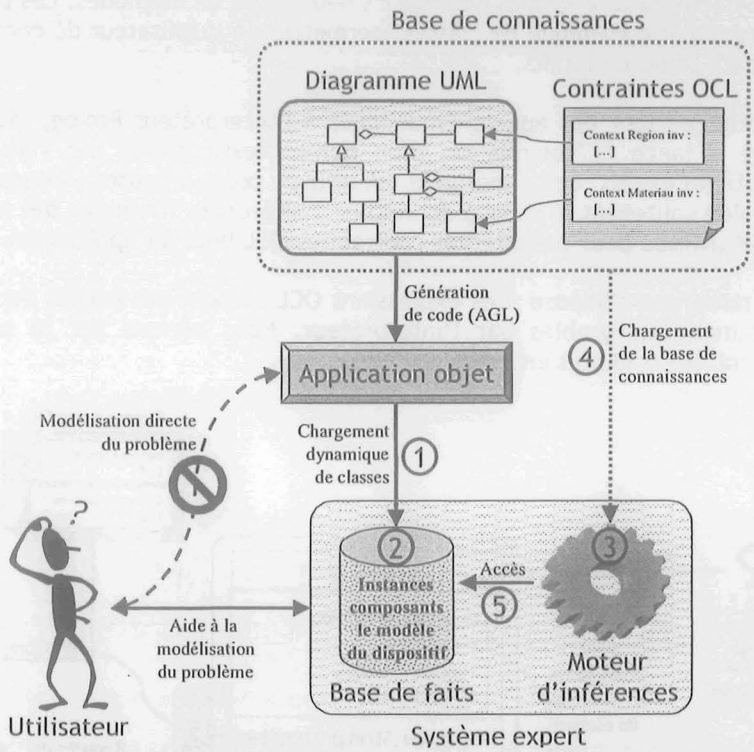


Figure 8 : Principe de fonctionnement du système expert

L'autre avantage de l'indépendance est évidemment que l'utilisateur retrouve à chaque fois le même système d'aide intégré. Il lui devient familier, et ceci même si l'application considérée change.

3.1.2 Architecture du système expert

L'architecture générale du système expert que nous souhaitons créer dépend de deux choses : d'une part du cahier des charges définissant les fonctionnalités à implanter, et d'autre part des outils informatiques à notre disposition.

Notre système expert doit pouvoir charger dynamiquement des classes d'une application. Il doit en outre avoir accès au modèle objet de l'application. Le langage Java permet non seulement le chargement dynamique de classes (1-Figure 9) dans la machine virtuelle, mais en plus, grâce au package « java.lang.reflect », il met en œuvre la réflexivité (2-Figure 9) et par conséquent l'accès à l'information structurelle dont nous avons besoin.

Le système expert proprement dit est donc composé d'une classe « SystemeExpert » associée à la classe « BaseFaits » dont l'attribut « modèle » de type « Vector » contient les instances participant à la modélisation du dispositif. C'est donc notre base de faits.

La classe « SystèmeExpert » encapsule l'interpréteur Prolog, codé en C. L'encapsulation implique un dialogue entre ces deux entités. L'interpréteur que nous avons utilisé comporte un prédicat spécial (new_java_pred/5) permettant l'appel à une méthode Java.

Le dialogue dans l'autre sens n'est pas encore implanté. Il a été nécessaire de développer des méthodes natives par la technologie des Java Native Interface (JNI) pour réaliser l'appel dynamique depuis Java à l'interpréteur Prolog.

Dans la classe « SystemeExpert », nous distinguons deux types de méthodes. Les premières, mettant en œuvre le chargement dynamique de classes, permettent à l'utilisateur de construire son modèle et de le stocker dans la base de faits.

Les secondes méthodes sont des appels dynamiques à l'interpréteur Prolog, via JNI. La méthode « compiler(String) » lance la compilation d'un fichier texte prolog par l'interpréteur. Dès le lancement du système expert, cette méthode est utilisée pour initialiser l'interpréteur et analyser le programme Prolog contenant le codage du moteur d'inférences (chaînage des règles). Elle est par la suite également utilisée pour compiler les contraintes OCL liées à l'application.

A ce stade, une remarque s'impose : les expressions OCL ne sont pas écrites en Prolog, et ne sont donc pas directement analysables par l'interpréteur. Nous verrons par la suite comment les expressions OCL ont été traduites en prédicats Prolog.

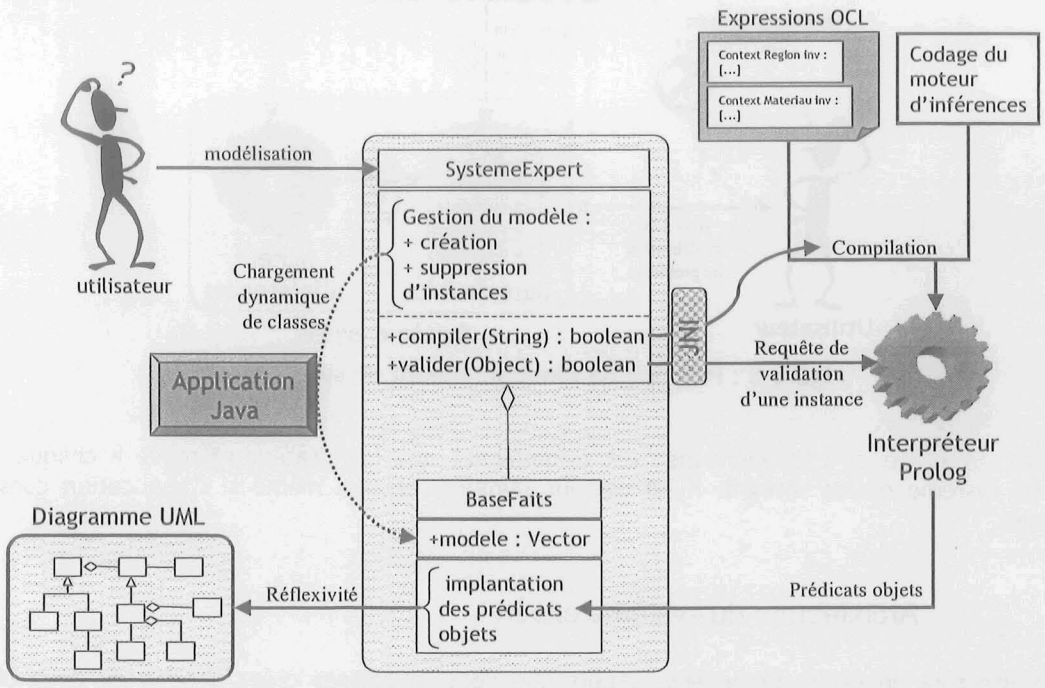


Figure 9 : Architecture du système expert

Le second appel dynamique depuis Java vers Prolog est la méthode « valider ». Elle envoie la requête de validation sur un objet passé en paramètre. L'interpréteur Prolog va donc être naturellement amené à manipuler des instances. Il a donc besoin de deux choses :

- une représentation des instances en Prolog
- des méthodes pour manipuler ces instances.

Ces deux points sont respectivement l'objet des deux paragraphes suivants.

3.2 La représentation des instances

La base de faits est constituée d'un ensemble d'instances de classes de l'application sur laquelle l'utilisateur travaille. Le moteur d'inférences doit pouvoir manipuler ces faits, et par conséquent disposer d'une représentation Prolog de ces instances Java.

A l'exclusion des types primitifs, la structure de donnée principale manipulée par l'interpréteur Prolog est l'arbre. Il est donc naturel de transcrire une instance sous cette forme. Nous avons choisi d'implanter la représentation arborescente suivante :

```
?- instance( Id, Classe, Attributs).
```

Le premier argument de l'arbre « instance/3 » est un identificateur de l'instance ainsi représentée. En Java, l'identificateur d'un objet est un entier appelé hashcode. C'est ce nombre que nous avons repris pour identificateur, précédé du symbole « # ». Le deuxième argument est la classe de l'instance, suivie de la liste des attributs de l'instance.

Voici deux exemples de la représentation d'une instance des classes Matériau et Formulation définies dans le diagramme UML de la Figure 3 :

```
?- instance( #536210, matériau, [ 1, 0 ]).
?- instance( #123515 , formulation, [ "Tw", instance( #235651, varPhy, [ phi ]),
                                      instance( #66301, varPhy, [ "T" ]) ] ).
```

La Figure 10 ci-dessous met en parallèle les représentations Prolog et UML de l'instance de Formulation "Tw" de notre application.

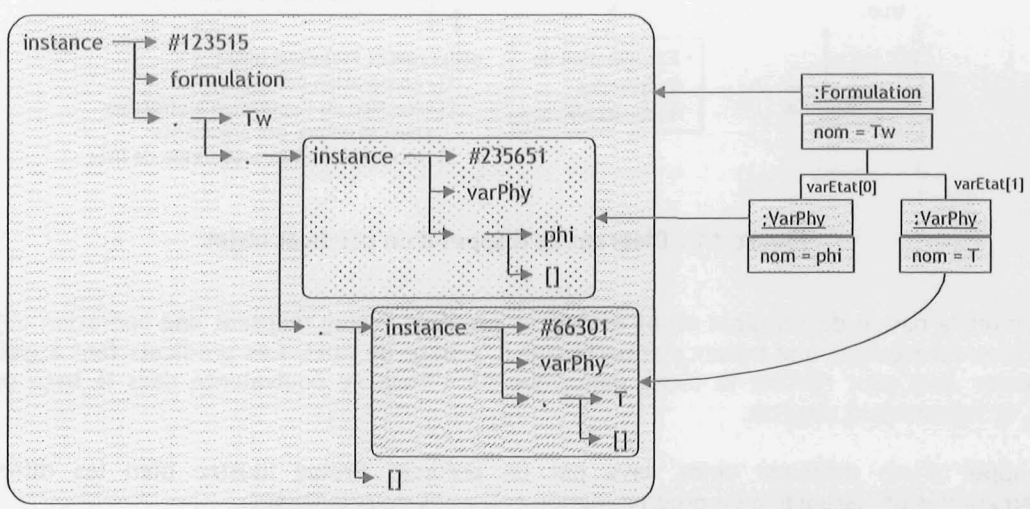


Figure 10 : Exemple d'arbre représentant une instance

On remarque tout de suite que dans cette représentation les attributs, qui sont soit des instances, soit des types primitifs, font partie intégrante de l'instance en tant que telle. Ceci permet, lors de

l'unification de deux arbres représentant la même instance, d'unifier également l'ensemble des sous-arbres représentant leurs attributs.

Un problème récurrent dans les langages orientés objet est le cas de figure où deux instances se référencent mutuellement. Le groupe ainsi formé devient persistant, et encombre la mémoire alors même que l'utilisateur n'a plus accès à cette ressource.

A l'inverse, la programmation logique supporte bien ce type de données. Cependant, nous verrons dans le paragraphe dédié aux chaînages que la présence de références mutuelles induit un traitement spécifique lors de l'évaluation des contraintes sur les instances.

3.3 *Les prédicats objet*

3.3.1 Le principe

Notre base de faits contient donc des instances issues de la base données de l'application. Ces dernières doivent être manipulables de la même manière que les objets qu'ils représentent. A cet effet, notre système expert doit implanter les opérations d'accès aux attributs d'une instance, ou encore les opérations OCL prédéfinies.

Les arbres « instance/3 » ne sont qu'une structure intermédiaire permettant l'accès à la base de faits contenant physiquement les instances. Sur ces dernières, il est aisé de coder en Java les opérations basiques d'accès à la valeur d'un attribut, de test de type, etc...

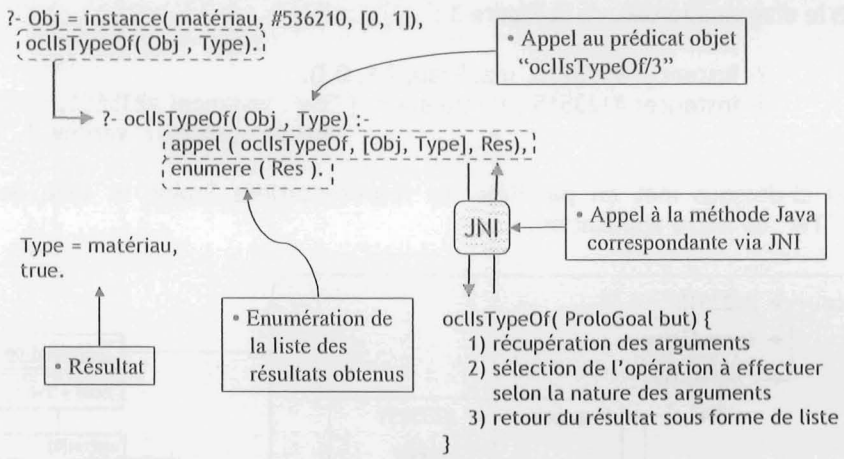


Figure 11 : Diagramme d'appel d'un prédicat objet

On définit la notion de **prédicat objet** qui est un prédicat Prolog réalisant une opération objet sur un arbre représentant une instance appartenant à la base de faits. Ces prédicats font appel à des méthodes Java pour réaliser la tâche demandée sur l'instance équivalente dans la base de faits Java, et retourner le résultat.

Cet appel d'une méthode objet Java par un prédicat Prolog illustre bien les différences fondamentales de paradigme de programmation entre ces deux langages.

En effet, une méthode Java est d'essence algorithmique : elle code une opération à réaliser sur un jeux de paramètres d'entrée identifiés et connus à l'appel, et retourne un résultat dont la nature est déterminée a priori.

A l'inverse, en programmation logique, on ne sait pas a priori, lors du codage de la règle, les paramètres de l'appel qui seront connus. Un prédicat d'arité n réalise donc 2^n requêtes possibles.

Par exemple prenons le prédicat « oclIsTypeOf/2 ». Il définit la relation objet qui existe entre une instance et son type. Il existe 2^2 soit 4 requêtes possibles :

requête :	Interprétation :
?- oclIsTypeOf (instance(...), region). true.	L'instance spécifiée est-elle de type région ?
?- oclIsTypeOf (X, region). X = instance(...), true.	Existe-t-il des instances de type région ?
?- oclIsTypeOf (instance(...), Y). Y = region, true.	Quel est le type de l'instance spécifiée ?
?- oclIsTypeOf (X, Y). X = instance(...), Y = region, true.	Détermine tous les couples instance-type

Tableau 3 : Interprétations d'une requête

Ce non-déterminisme se reflète lors du codage d'un prédicat Prolog établissant une relation objet entre ses arguments. La méthode Java doit traiter tous les cas possibles de requêtes. Si la méthode Java ne peut pas traiter la requête, alors elle se contente de retourner la valeur booléenne faux comme résultat de l'évaluation du prédicat.

Une autre différence importante entre une méthode Java et un prédicat Prolog concerne la solution fournie. Une méthode Java renvoie une et une seule réponse par appel. Un prédicat va énumérer un ensemble de solutions qui vérifient la requête.

C'est pour cela que les méthodes Java qui sont appelées par les prédicats objet calculent en une fois l'ensemble des tuples solutions sur la base de faits. Cette liste de tuples solutions est retournée au prédicat appelant, qui se charge par la suite d'énumérer les solutions contenues dans la liste.

Ainsi, malgré l'appel à une méthode Java, les prédicats objet ont un comportement similaire à celui de tout prédicat Prolog. Ils répondent à plusieurs types de requêtes, et énumèrent les tuples solutions.

3.3.2 Les opération OCL prédéfinies

Afin de manipuler les instances, nous avons donc construit des prédicats Prolog implantant la manipulation des instances. Ces prédicats font appel à des méthodes Java, sur le principe exposé ci-dessus. Nous allons maintenant détailler les prédicats objet qui ont été implantés.

a- attribut/3

Le premier prédicat que nous avons câblé est évidemment celui permettant l'accès dynamique à la valeur de l'attribut d'une instance. Il s'agit du prédicat « attribut/3 » :

```
?- attribut ( Instance, NomAttribut, Valeur ).
```

Le premier argument est un arbre représentant une instance. Le deuxième argument est le nom de l'attribut et enfin le troisième sa valeur pour l'attribut concerné.

Ce prédicat est d'arité égale à trois. On devrait par conséquent coder huit méthodes Java pour chacune des requêtes possibles. Cependant, ce prédicat, comme les autres prédicats codant une opération objet, n'a pas pour finalité d'être mis à disposition de l'utilisateur, mais d'être exploité par le moteur d'inférence.

Ce dernier implante des fonctionnalités bien précises, qui ne nécessitent que certaines requêtes issues des opérations objet. Par conséquent, seules les requêtes utilisées par le moteur d'inférence ont été implantées par des méthodes Java.

Dans le cas du prédicat « attribut/3 », seules deux requêtes ont été codées en Java :

- la vérification que la valeur spécifiée est bien celle de l'attribut pour l'instance passée en premier argument,
- la recherche des valeurs possibles de l'attribut pour l'instance spécifiée.

Dans les deux cas, l'évaluation du prédicat va nécessiter l'accès à la valeur l'attribut concerné de l'instance spécifiée. Il existe alors deux cas de figure possibles, suivant que la valeur de l'attribut est définie ou non dans la base de donnée.

Dans le premier cas, le prédicat « attribut/3 » unifie son troisième argument avec un arbre représentant la valeur de cet attribut. Sinon, l'instance est partiellement définie, et le prédicat énumère un tuple de valeurs possibles pour cet attribut.

Lorsque l'attribut non-défini est de type primitif, Le moteur d'inférence lui associe une variable du bon type qu'il peut par la suite contraindre. Dans l'exemple de la Figure 12 ci-dessus, l'attribut perméabilité du matériau #521003 est non-défini. Le compilateur crée une variable qu'il va contraindre à être de type entier.

Voici, dans le cas de la Figure 12 ci-dessus, le résultat de l'appel du prédicat « attribut/3 » :

```
?- attribut( instance(#875023,region,_), materiau, Val).
   A ex Val = instance(#521003, materiau, [ 0, A] ) ;
   Val = instance(#521003, materiau, [ 10000, 1] ).

?- attribut( instance(#521003, materiau,_), permeabilite, Mu ).
   Mu ~ real.
```

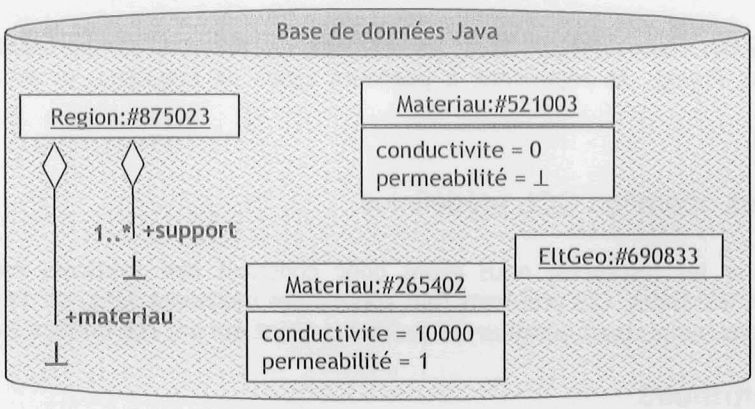


Figure 12 : Cas d'utilisation pour le prédicat « attribut/3 »

Dans la première évaluation, le prédicat « attribut/3 » a bien unifié l'attribut demandé avec les deux instances présentes dans la base de faits. Pour la première solution, Prolog crée une variable muette A correspondant à l'attribut « perméabilité » non défini pour cette instance de matériau. Dans la deuxième évaluation, la variable Mu a bien été contrainte a être de type réel.

Le cas de figure où l'attribut à construire est un tableau est plus problématique. En effet, il peut y avoir dans ce cas une infinité de solutions possibles à construire. Par exemple, l'attribut « support » de l'instance Région:#875023 dans le cas de la Figure 12 a pour solutions possibles tous les tableaux contenant une, deux, trois, ..., N fois l'instance EltGeo:#690833.

Seul deux types de tableaux sont énumérables : les tableaux dont la dimension maximum est définie, et les tableaux où les éléments ne peuvent pas être présents plus de N fois. Pour cette raison, lors de l'évaluation d'un attribut de type tableau, le système expert va faire appel à l'utilisateur pour spécifier la taille maximale du tableau :

```
?- attribut ( instance ( #123050, region, _), support, Valeur).
   préciser la taille maximum de tableau autorisé : 2.
   Valeur = [] ;
   Valeur = [ instance ( #690833, eltgeo, []) ] ;
   Valeur = [ instance ( #690833, eltgeo, []), instance ( #690833, eltgeo, []) ],
   true.
```

Le prédicat objet « attribut/3 » construit donc toutes les valeurs possibles de l'attribut d'une instance donné. Par conséquent c'est sur ce prédicat spécifique que repose la fonctionnalité de proposition de notre système expert.

b- oclIsTypeOf/2

Le deuxième prédicat d'opération objet que nous avons codé est « oclIsTypeOf/2 » :

```
?- oclIsTypeOf( Instance, Type).
```

Il établit une relation de typage entre l'instance passée en premier argument, et le type passé en second argument. L'étiquette de ce prédicat, « oclIsTypeOf », correspond à l'opération OCL prédéfinie qu'il code.

Les requêtes disponibles par ce prédicat sont :

- la vérification du type d'une instance,
- la recherche du type d'une instance,
- la recherche des instances d'un type défini.

Voici un exemple d'utilisation, toujours basé sur le diagramme de classes de la Figure 3, la base de données contenant deux instances de matériau :

```
?- oclIsTypeOf ( instance ( #130552, _, _), materiau).
   true.

?- oclIsTypeOf ( X, materiau).
   X = instance ( #130552, materiau, [ 0, 1]) ;
   X = instance ( #956330, materiau, [ 10000, 1]) ,
   true.

?- oclIsTypeOf ( instance ( #130552, _, _), Type).
   Type = materiau,
   true.
```

c- query/3

Le prédicat objet « query/3 » code l'invocation d'une méthode sans effet de bord :

```
?- query ( Instance, NomOpération, Arguments ).
```

Le premier argument de ce prédicat est l'instance sur laquelle va être invoquée l'opération spécifiée en deuxième argument. Le dernier argument est la liste des paramètres de l'opération.

Pour ce prédicat, seule la requête d'invocation, avec tous les arguments connus, a été codée. Pour reprendre un exemple d'utilisation toujours dans le cadre de notre logiciel de simulation électromagnétique (Figure 3) :

```
?- query ( instance( #122001, region, _), connexite, [] ).
   true.
```

Ce prédicat met l'accent sur l'appel dynamique de méthodes Java depuis l'interpréteur Prolog. Il est à noter que le logiciel PrologIV dispose d'un prédicat spécifique permettant de réaliser un foncteur associé à une méthode Java. C'est le prédicat « new_java_pred/5 », dont les arguments sont, dans l'ordre :

- l'étiquette du foncteur appelant dynamiquement la méthode Java,
- l'arité du foncteur,
- le nom de la méthode Java appelée,
- la classe où est définie la méthode,
- le chemin du répertoire où se situe la classe.

Voici l'exemple d'utilisation extrait du tutoriel de PrologIV :

```
>> new_java_pred('inv_liste', 2, 'inverse', 'exemples', 'file:/C:/users/classes/').
>> inv_liste( [1,2,3], L).   <- appel la méthode inverse sur la liste [1,2,3]
   L = [3,2,1].
```

Nous n'avons pas utilisé ce prédicat, en raison de la présence du chemin d'accès à la classe. En effet, l'interpréteur de PrologIV est encapsulé dans un environnement Java que nous avons développé, et où les classes ont été préalablement chargées. Il est donc inutile, voir dangereux, de spécifier un chemin, alors que la réflexivité des classes Java nous permet d'accéder à la méthode souhaitée.

d- forAll/3

Le prédicat « forAll/3 » :

```
?- forAll ( Liste, Elt, Regle ).
```

est un prédicat objet particulier dans le sens où il ne fait appel à aucune méthode Java pour implanter l'opération OCL prédéfinie « forAll ». Il parcourt et unifie les éléments de la liste passée en premier argument avec le deuxième argument, et applique la règle passée en dernier argument à l'élément courant. Ce prédicat échoue si :

- la liste d'éléments ou la règle est non-définie,
- il existe un élément de la liste qui ne vérifie pas la règle spécifiée.

Pour illustrer l'utilisation de ce prédicat, voici un exemple :

```
?- Liste = [ instance (#120301, materiau, _), instance( #230100, materiau, _) ],
   forAll ( Liste, Mat, ( ( attribut( Mat, conductivite, Sigma), Sigma >= 0 ) ) ).
   true.
```

où les deux instances de matériaux ont une conductivité respective de 0 et de 10.000. Le deuxième argument « Mat » du prédicat « forAll/3 » est destiné a spécifier la variable dans le corps de la règle passé en dernier argument. Cette variable sera par la suite unifiée avec les éléments de la liste. Le deuxième argument de ce prédicat est donc toujours une variable nommée.

Voici le programme Prolog qui code le prédicat « forAll/3 » :

```
00
01 %% ---- forAll/3 ----
02 %% implantation de l'instruction OCL prédéfinie forAll
03 %% L : liste d'instances (toujours connue)
04 %% Elt : variable nommée, utilisée dans Corps
05 %% Corps : corps de la règle qui s'applique à Elt
06
07   forAll( L, Elt, Corps ) :-
```

```
08              %% assertion de la clause Corps dans la base de connaissance
09              dynamic( ^/(regleTmp,2) ),
10              numberForAllCorps( N ),
11              assertz( ( regleTmp( N+1, Elt) :- Corps ) ),
12              %% application de la règle aux instances de la liste
13              forAll( L, N+1).
14
15  %% ---- forAll/2 ----
16  %% application de la règle sur les instances de la liste
17  %% arg1 : la liste des instances
18  %% arg2 : le numéro de la règle à appliquer
19
20    forAll( [], _).
21
22    forAll( [ Elt|Q], Num) :-
23              %% création du foncteur Act de la règle à appliquer
24              %% sur l'instance courante
25              =..( Act, [regleTmp, Num, Elt]),
26              %% appel de la règle regleTmp( Num, Elt)
27              Act,
28              %% passage à l'instance suivante
29              forAll( Q, Num).
30
31  %% ---- numberForAllCorps/1 ----
32  %% retourne le nombre de corps de regle déjà assertés
33  %% N : le nombre de corps
34
35    numberForAllCorps( N ) :-
36              ( findall( I, clause(regleTmp(Num,_),_), L) ->
37                size(N,L); N = 0 ).
38
```

Tableau 4 : Le prédicat « forAll/2 »

On notera à la ligne 11 du programme l'assertion dynamique d'une nouvelle règle « regleTmp/2 » dans la base de connaissance.

e- exist/3, select/4

Les prédicats « exist/3 » et « select/4 » sont des prédicats objet de manipulation de liste d'instances prédéfinis dans l'OCL :

```
?- exist ( Liste, Elt, Regle ).
?- select ( Liste, Elt, Regle, ListeRes ).
```

Le prédicat « exist/3 » renvoie vrai si il existe un élément de la liste passée en premier argument qui vérifie la règle passée en dernier argument.

Le prédicat « select/4 » retourne en dernier argument une liste contenant les éléments de la liste passée en premier argument et qui vérifie la règle passée en troisième argument.

Ces deux prédicats ont le même fonctionnement que le prédicat « forAll/3 » présenté précédemment. Le corps de la règle à appliquer est passé en troisième argument, et fait référence à la variable nommée placée en deuxième argument. La règle est dynamiquement assertée dans la base de connaissances lors de l'appel à ces prédicats.

3.4 Des expressions OCL aux règles Prolog

Nous avons utilisé un AGL pour concevoir l'architecture UML de notre application et générer son code Java. Comme nous l'avons vu dans le premier paragraphe de ce chapitre, les expressions OCL attachées au modèle UML de l'application disparaissent lors de la génération de code.

Ces dernières doivent être transmises à notre système expert, puisqu'elles constituent la base de règles de notre système expert. Pour ce faire, il a été nécessaire de traduire les invariants OCL en prédicats Prolog.

3.4.1 Le prédicat Invariant/3

Le prédicat « invariant/3 » est la structure de règle équivalente aux expressions OCL de type invariant de classe :

```
?- invariant ( NomRegle, Classe, Instance).
```

Le premier argument est le nom de la règle, le deuxième la classe à laquelle est associée cet invariant et enfin en dernier argument l'instance sur laquelle s'applique la contrainte de classe.

Le corps de la règle d'un invariant traduit une contrainte qui s'exerce sur l'instance spécifiée. C'est la traduction Prolog d'un invariant de classe OCL. L'évaluation de la contrainte sur l'instance implique l'utilisation des prédicats objets définis précédemment.

Voici un exemple d'invariant de classe issu de la Figure 3, contraignant la perméabilité relative d'un matériau à être de valeur supérieure ou égale à l'unité :

```
context Materiau inv regle_No_1 :
 self.permeabilite >= 1
```

et sa traduction en prédicat Prolog :

```
?- invariant ( regle_No_1, materiau, Self) :-
     attribut ( Self, permeablilite, Mur),
     Mur >= 1.
```

On notera l'utilisation dans le corps de la règle des prédicats objets qui permettent la manipulation des instances. Dans l'exemple ci-dessus, la règle utilise le prédicat « attribut/3 » afin d'unifier la variable Mur avec la valeur de la perméabilité du matériau représenté l'instance Self.

3.4.2 Quelques exemples de règles Invariants/3

Nous allons maintenant traduire certains invariants de classe OCL introduits au paragraphe 1.2.2 en règles Prolog. Le problème de connexité est décrit par l'invariant de classe OCL suivant :

```
context Region inv regle_de_connexite :
 self.phénomène.varEtat -> exists( v | v.nom = "phi" or v.nom = "phir" )
 implies self.connexite()
```

dont la traduction en règle Prolog est :

```
?- invariant ( regle_de_connexite, region, Self) :-
    attribut ( Self, phénomène, Mod),
       attribut ( Mod, varEtat, ListeVar),
       ( existe ( ListeVar, E, ( ( attribut ( E, nom, phi) ; attribut ( E,nom, phir) ) )
         -> query ( Self, connexite, []) ;
```

```
true).
```

On retrouve bien sûr dans le corps de la règle ci-dessus l'utilisation des prédicats objets (en **gras**) définis dans au paragraphe précédent. On remarque sur cet exemple que certains mots-clefs ou certaines structures de la syntaxe OCL possèdent une traduction immédiate en Prolog. Par exemple le mot-clef «or» se traduit naturellement en Prolog par l'utilisation d'un «;».

L'invariant OCL suivant est la seconde contrainte du **Tableau 1** concernant la non prise en compte des sources de courants par la formulation en potentiel scalaire total :

```
context Region inv regle_bobine :
  self.phénomène.nom = "pot_sca_tot" implies
  self.sources -> forAll( s | not s.oclIsTypeOf( Bobine) )
```

et la règle Prolog associée :

```
?- invariant ( regle_bobine, region, Self) :-
        attribut ( Self, phénomène, Mod),
        attribut ( Self, sources, Src),
        ( attribut ( Mod, nom, pot_sca_tot ) ->
            forAll ( Src, Elt, ( non( oclIsTypeOf( Elt, bobine) ) ) ) ;
            true ).
```

Pour terminer avec la traduction des invariants OCL, nous donnons ci-dessous dans l'ordre les règles correspondantes au contraintes exprimées dans le **Tableau 1** :

```
?- invariant ( regle_materiau_isolant, region, Self ) :-
        attribut ( Self, materiau, Mat),
        attribut ( Self, phénomène, Mod),
        attribut ( Mod, nom, Nom),
        ( attribut ( Mat, conductivite, 0) ->
            ( Nom = pot_sca_tot ; Nom = pot, sca_red ; Nom = pot_vect) ;
            true ).

?- invariant ( regle_materiau_isolant, region, Self ) :-
        attribut ( Self, materiau, Mat),
        attribut ( Self, phénomène, Mod),
        attribut ( Mod, nom, Nom),
        ( ( Nom = pot_sca_tot ; Nom = pot, sca_red ; Nom = pot_vect) ->
            attribut ( Mat, conductivite, 0) ;
            true ).

?- invariant ( regle_materiau_isolant, region, Self ) :-
        attribut ( Self, materiau, Mat),
        attribut ( Self, phénomène, Mod),
        attribut ( Mod, nom, Nom),
        attribut ( Mat, conductivite, Sig)
        ( ( Sig > 0 ) ->
            ( Nom = "Tw" ; Nom = "Av" ; Nom = pot_vect_mod) ;
            true ).

?- invariant ( regle_materiau_isolant, region, Self ) :-
        attribut ( Self, materiau, Mat),
        attribut ( Self, phénomène, Mod),
        attribut ( Mod, nom, Nom),
        attribut ( Mat, conductivite, Sig)
        ( ( Nom = "Tw" ; Nom = "Av" ; Nom = pot_vect_mod) ->
            ( Sig > 0) ;
            true ).
```

```
?- invariant ( regle_saut_conductivite, region, Self ) :-
        attribut ( Self, phénomène, Mod),
        ( attribut ( Mod, nom, pot_vect_mod) ->
          ( attribut ( Self, Materiau, Mat),
            attribut ( Mat, conductivite, Sigma),
            attribut (Self, voisins, ListeReg),
            forAll ( ListeReg, Reg,
               (  attribut ( Reg, phénomène, Mod_),
                  ( attribut ( Mod_, nom, pot_vect_mod) ->
                    ( attribut ( Reg, materiau, Mat_),
                      attribut ( Mat_, conductivite, Sigma_),
                      Sigma = Sigma_ ) ;
                    true ) ) ) ;
          true ).
```

On remarque avec cette dernière règle notamment le manque d'élégance d'écriture d'un invariant en Prolog, comparé au même invariant exprimé en OCL. Cet aspect n'est que secondaire. Il découle de notre volonté de rester le plus proche possible de la syntaxe OCL dans la traduction Prolog des invariants.

3.5 *Le chaînage*

A ce stade de notre travail, la base de connaissance du système expert est prête. La base de faits est constituée de la base de données Java contenant les instances, rendue dynamiquement accessibles via les prédicats objet tel que « attribut/3 ». La traduction en prédicats Prolog des invariants OCL constitue notre base de règles. Il reste à coder le moteur d'inférences proprement dit, c'est à dire le chaînage des règles.

3.5.1 Récupération et application des invariants

Le but du moteur est d'évaluer les invariants de classe sur une instance spécifiée. Cette requête correspond au prédicat « contraindre/1 » dans notre moteur d'inférence, dont voici le code Prolog associé :

```
00
01 %% ---- contraindre/1 ----
02 %%  recherche la liste des contraintes et les applique
03 %% à l'instance Self passée en argument
04
05     contraindre ( Self ) :-
06         Self = instance( Id, Class, Att),
07         findall( (Nom, Class), clause( invariant( Nom, Class,  _), _), ListeRegles),
08         appliquerContraintes ( Self, ListeRegles ).
09
10 %% ---- appliquerContraintes/2 ----
11 %%  applique la liste des contraintes
12 %%  arg1 : l'instance
13 %%  arg2 : la liste des contraintes
14
15     appliquerContraintes ( _,[]).
16
17     appliquerContraintes ( Self, [ (Nom, Class)| Reste] ) :-
18         ( oclIsTypeOf ( Self, Class) -> ( invariant ( Nom, Class, Self) -> true ;
19                                                 avertir( Nom, Self ) ) ;
20                                         true ),
21         appliquerContraintes ( Self, Reste).
22
```

```
23 %% ---- avertir/2 ----
24 %% prévient l'utilisateur lors de l'échec de l'évaluation d'un invariant
25 %% arg1 : l'invariant violé
26 %% arg2 : instance évaluée
27
28 avertir( Nom, Self ) :-
29         Self - instance( Id, _, _),
30          write(' >> évaluation de  '), write(Nom), write(' sur '),
31          write(Id), write(' : échec').
32
```

Tableau 5 : Le prédicat "contraindre/3"

L'unique argument du prédicat est la représentation Prolog de l'instance Java que l'on souhaite valider. La tâche de validation peut être décomposer en deux sous-tâches :

- la récupération des invariants associés à l'instance en cours d'évaluation,
- l'application les invariants récupérés.

Le premier point est trivial, si l'on n'oublie pas de récupérer l'ensemble des invariants applicables à l'instance, c'est à dire non seulement les invariants de sa classe, mais également ceux de ses super-classes. Pour notre part, nous avons choisi de récupérer l'ensemble des invariants (l.07 du Tableau 5), puis de parcourir la liste obtenue et de n'évaluer que les invariants appartenant à la hiérarchie de classe de l'instance (l.18 du Tableau 5).

L'évaluation d'un invariant va nécessairement impliqué l'accès aux attributs de l'instance évaluée via le prédicat objet « attribut/3 ». Or, nous avons vu dans le paragraphe dédié à ce prédicat que non seulement ce dernier récupère dynamiquement dans la base de données Java la valeur d'un attribut défini, mais qu'en plus il énumère automatiquement les valeurs possibles dans le cas d'un attribut non défini.

Le système expert avertit l'utilisateur lors de l'échec de l'évaluation de chaque invariant (l.19 du Tableau 5). Finalement, l'évaluation du prédicat « contraindre/3 » implique donc les fonctionnalités de validation (« invariant/3 »), d'explication (« avertir/2 ») et de proposition (« attribut/3 »).

3.5.2 Propagation de l'évaluation le long du graphe objet

On défini le graphe objet issu d'une instance A comme étant le graphe dont :

- les nœuds sont des instances
- les arcs sont des relations d'associations entre deux instances.

En d'autres termes, le graphe objet issu de l'instance A représente l'ensemble des instances accessibles depuis cette première, et leurs chemins d'accès. La Figure 13 représente notamment le graphe objet issu de l'instance Region:#856003.

Le graphe objet d'une instance est un graphe orienté. La présence d'attributs réciproques implique que le graphe n'est pas un arbre, mais peut au contraire présenter des cycles (Figure 13).

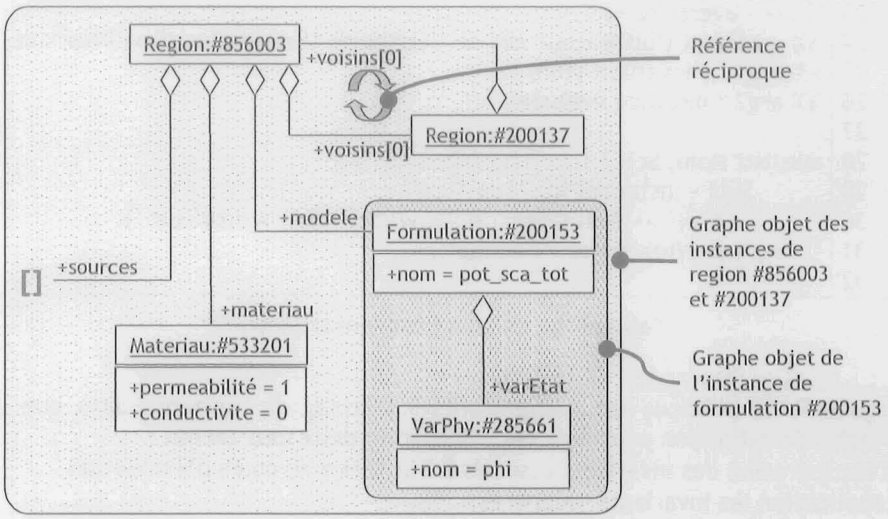


Figure 13 : Exemple de graphes objet avec référence réciproque

Dans le paragraphe précédent, le prédicat « contraindre/3 » récupère et évalue les invariants de classe sur l'instance passée en paramètre. L'évaluation de invariants entraîne potentiellement la proposition de valeurs possibles pour les attributs non défini (prédicat « attribut/3 »).

Or, au niveau sémantique, il existe divers types d'associations. Notamment, l'association de composition établie une relation de type composite/composant entre deux objets. Dans ce cas, l'objet composite est valide si et seulement si :

- il vérifie ses invariants de classe,
- ses composants sont valides.

La nuance sémantique établie dans UML entre association et composition n'est pas présente dans le langage de programmation Java que nous utilisons. Par conséquent, nous supposerons que tout attribut d'une classe Java est potentiellement une composition.

Par conséquent, nous avons implanté la validation d'une instance donnée par un algorithme de parcours du graphe objet (Tableau 6) dans lequel chaque nœud est validé. La conservation de la liste des instances déjà validées permet de parcourir le graphe objet en évitant les cycles.

Le Tableau 6 ci-dessous présente le code Prolog commenté des prédicat « valider/1 » et « parcoursGraphe/2 » :

```
00
01 %% ---- valider/1 -----
02 %% requête de validation de l'instance Self passée en paramètre
03
04     valider ( Self ) :- parcoursGraphe ( [ Self ], [] )
05
06 %% ---- parcoursGraphe/2 -----
07 %% requête de validation de l'instance passée en paramètre
08
09     parcoursGraphe ( [], _ ).
10
11     parcoursGraphe ( [ Self | Reste], Vus ) :-
12         %% trouve et applique les contraintes OCL sur l'instance Self
13         contraindre ( Self ),
14         %% récupère les instances attributs de Self qui n'ont pas déjà
15         %% été contraintes (∉ Vus)
16         recupererAttributs ( Self, Vus, AttNonVus),
```

```
17      %% empile le reste des instances à voir et les
18      %% attributs non-vus dans la liste AVoir
19      empiler ( Reste, AttNonVus, AVoir),
20      %% récurrence
21      parcoursGraphe ( AVoir, [ Self | Vus ]).
22
```

Tableau 6 : Le prédicat "valider/1"

4 Retour sur les exemples d'utilisation

Nous disposons maintenant d'un outil qui nous permet d'évaluer des expressions OCL traduites en Prolog, sur des instances. Nous allons maintenant valider cet outil sur les exemples que nous avons présentés au deuxième paragraphe de ce chapitre.

4.1 Contexte d'utilisation

Préalablement à l'utilisation du système expert proprement dit, il nous a semblé important de détailler l'environnement d'utilisation Java que nous avons implanté.

Cet environnement permet à l'utilisateur de :

- charger dynamiquement des classes Java,
- gérer une bibliothèque d'instances de ces classes (création, suppression, relations),
- lancer la compilation de fichiers Prolog par le système expert,
- évaluer une instance de la bibliothèque.

La première tâche à effectuer par l'utilisateur est de charger les classes du logiciel qu'il souhaite utilisé, et lancer la compilation des fichiers contenant les invariants OCL de ce logiciel sous forme de prédicats Prolog. Ensuite seulement intervient le processus de modélisation. L'utilisateur va modéliser son dispositif en instanciant des classes du logiciel. Ces instances sont stockées dans sa bibliothèque.

A tout moment, l'utilisateur peut demander l'évaluation de la validité d'une instance de cette dernière. Si aucun invariant n'a été préalablement compilé, l'évaluation sera toujours un succès. Sinon, l'environnement récupère les tuples solutions et les affiche sans autre traitement.

4.2 Les exemples d'utilisation

Le modèle UML de notre application étant dorénavant complet, aussi bien structurellement que contractuellement, nous allons pouvoir valider l'environnement que nous avons développé dans le cadre de la modélisation des dispositifs que nous avons présentés au second paragraphe. Par conséquent nous ne reviendrons pas sur la description des dispositifs.

4.2.1 La validation, l'explication et la proposition

Dans un premier, nous allons modéliser totalement notre dispositif, en créant deux régions R_{air} et R_{cir}, respectivement pour la boite d'air et le circuit ferromagnétique. Leurs attributs sont récapitulés dans le tableau suivant :

Région	Formulation	Matériau	Source	Support
R_{air}	Φ	Vacuum (μ=1,σ=0)	1 inducteur	1 volume
R_{cir}	Tw	Steel (μ=10000,σ=2000)	aucune	1 volume

Les deux volumes associés à chacune des régions est non simplement connexe. Demandons au système expert de valider chacune des deux régions. Le résultat est alors [2] :

```
?- valider( instance ( #air, region, Arg) ).
   >> évaluation de regle_de_connexite sur #air : échec
   >> évaluation de regle_bobine sur #air : échec
   Arg = [ ... , instance ( #phi, regionale, [...]), ... ],
   true.

?- valider( instance ( #circuit, region, Arg) ).
   >> évaluation de regle_de_connexite sur #circuit : échec
   Arg = [ ... , instance ( #Tw, regionale, [...]), ... ] ],
   true.
```

Le système expert nous indique donc que chacune des deux régions que nous avons crées est invalide. En effet, l'une comme l'autre sont associées à une formulation utilisant le potentiel scalaire magnétique total Φ malgré leur support non simplement connexe. Par conséquent, elles violent le théorème d'Ampère, ce qui est indiqué par l'invariant nommé « regle_de_connexite ». De plus, pour la région d'air, l'utilisation de la formulation en scalaire magnétique total est inadaptée à la prise en compte de l'inducteur : c'est l'invariant « regle_bobine ».

Cet exemple illustre bien le fonctionnement de notre système expert pour la validation d'une instance, et l'explication de l'origine de l'invalidité.

Pour prendre en compte les indications délivrées par le système expert, nous allons supprimer les formulations des régions. Nous allons voir quelles solutions sont proposées pour la région d'air :

```
?- valider( instance ( #air, region, Arg) ).
   Arg = [ ... , instance ( #pot_vect, regionale, [...]), ..., [ instance( #circuit,
   region,[ ..., instance ( #AV,  regionale, [...]), ... ]) ], ... ] ;
   Arg = [ ... , instance( #phi_red, regionale, [...]), ..., [ instance( #circuit,
   region,[ ..., instance( #pot_vect_mod, regionale, [...]), ... ]) ], ... ],
   true.
```

Dans chacun des cas, le système expert a déterminé deux solutions. Analysons la première solution qui nous ai fournie (l.02-03) :

```
Arg = [ ... , instance ( #pot_vect, regionale, [...]), ...,
              instance( #circuit, region,[ ..., instance ( #AV,  regionale, [...]), ... ]), ... ]
```

La première instance présentée dans la liste des arguments est la formulation, proposée ici dynamiquement par Prolog. Il s'agit de la formulation en potentiel vecteur A. Le deuxième argument présenté est l'attribut « voisins » d'une région, qui dans le cas de la région R_{air} est composé d'une liste ne contenant que l'instance de la région R_{cir}. La propagation de l'évaluation sur l'ensemble du graphe objet va entraîner l'évaluation de cette région pour laquelle Prolog va également déterminer dynamiquement une formulation : ici, la formulation AV.

Le système expert a donc bien déterminé les deux solutions possibles à notre problème, soir par un couplage AV-A, soit par un couplage A^*-Φ_R.

4.2.2 Les contraintes numériques

Passons maintenant à l'exemple de l'électroaimaint. Le dispositif est alors modélisé par l'utilisateur sous la forme d'une région d'air R_{air} et d'une région ferromagnétique R_{noy}. Afin de prendre en

[2] Par souci de lisibilité : 1) les « ... » sont substitués à des informations non pertinentes par rapport à la requête posée et 2) les identificateurs d'instances (#123005) sont parfois remplacés par des noms plus significatifs (#air).

compte l'inducteur, la formulation associée aux deux régions sera la formulation en potentiel scalaire réduit Φ_R. Dans le cadre de cet exemple, nous allons supposer que la perméabilité relative du matériau associé au noyau ferromagnétique n'a pas été déterminée par l'utilisateur.

Région	Formulation	Matériau	Source	Support
R_{air}	Φ_R	Vacuum (μ=1,σ=0)	1 inducteur	1 volume
R_{noy}	Φ_R	Steel (μ= ?,σ=0)	aucune	1 volume

Demandons alors au système de valider la région du noyau en l'état :

```
?- valider( instance ( #noyau, region, Arg) ).
   A ex Arg = [ ... , instance( #steel, materiau, [ A, 0 ] ), ...],
   A ~ cc(1,1000).
   true.
```

Le système expert a déterminer que la perméabilité relative du matériau devait être comprise entre un et mille. Si on impose une perméabilité relative nulle, on obtient l'erreur suivante :

```
?- valider( instance ( #circuit, region, Arg) ).
   >> évaluation de regle_No_1 sur #circuit : échec
   true.
```

A l'inverse, si on impose une perméabilité relative élevée (10000 par exemple) :

```
?- valider( instance ( #circuit, region, Arg) ).
   >> évaluation de regle_permeabilité_élevée sur #circuit : échec
   true.
```

La conjonction des deux contraintes numériques permet de définir un intervalle de validité pour la valeur de la perméabilité relative du matériau.

4.2.3 Les conditions aux limites

Enfin, comme dernier exemple, nous prendrons le contacteur électromécanique (Figure 14) du tutoriel de Flux3D présenté dans le premier chapitre. Nous le modéliserons de la même manière, à savoir par quatre régions associées respectivement à la culasse (R_{cul}), la palette mobile (R_{pal}), l'aimant (R_{aim}) et l'air (R_{air}).

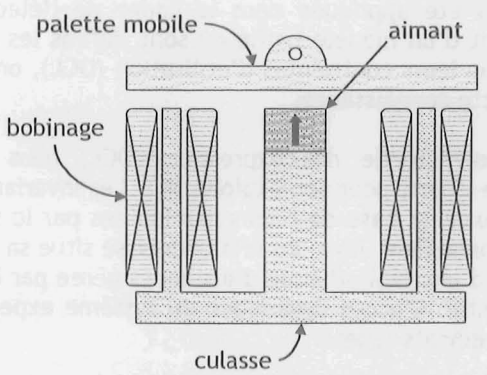


Figure 14 : le contacteur

Région	Formulation	Matériau	Source	Support
R_{air}	Φ_R	Vacuum (μ=1,σ=0)	1 inducteur	1 volume
R_{pal}	Φ	Steel (μ= 10000,σ=2000)	aucune	1 volume

R_{cul}	Φ	Steel (μ= 10000,σ=2000)	aucune	1 volume
R_{aim}	Φ	Aimant (μ= 10000,σ=0)	aucune	1 volume

Dans le cas du contacteur, les volumes sont simplement connexes. Nous allons maintenant demander au système expert d'évaluer la validité du problème global, composé de ces quatre régions :

```
?- valider( instance ( #contacteur, probleme, Arg) ).
   >> évaluation de cond_limite_Dirichlet sur #contacteur : échec
   true.
```

Tel que nous l'avons modélisé, le problème est mal posé. Il manque une condition limite de Dirichlet pour fixer le potentiel.

La base de faits associée au modèle de ce dispositif contient dix-sept instances. La base de règles comporte vingt-deux invariants OCL. On constate donc sur cet exemple assez complet que le choix d'un système expert est tout à fait adapté à notre type d'application. En effet, l'évaluation globale du problème du contacteur par le système expert a été effectuée quasi-instantanément pour l'utilisateur.

5 Conclusion

Notre travail est fondé sur le constat que les compétences et le savoir-faire des concepteurs d'applications informatiques, et plus particulièrement de logiciels de simulation numérique, présentent une dualité entre nature algorithmique et expertise. Cette différence d'essence est la raison pour laquelle les logiciels exploitent principalement la connaissance de nature algorithmique, et très peu l'expertise. Notre objectif est de réintroduire cette dernière au sein même du logiciel, en vue de son exploitation.

Le génie logiciel objet est une méthodologie éprouvée et largement répandue pour la conception et la spécification des systèmes à objets. Le langage standardisé UML permet de décrire l'aspect structurel de tels systèmes sous la forme de diagrammes de classes. Le langage OCL est utilisé pour spécifier l'aspect contractuel, c'est à dire déclarer les contraintes associées à chaque classe et opération du modèle UML.

Dans le cadre de logiciels de simulation orientés objet, le modèle UML réifie les concepts du domaine d'application physique. Dans ce cas, il est possible d'utiliser le langage OCL afin d'exprimer les **domaines de validité** de ces concepts physico-numériques. Cette **démarche méthodologique originale** a été appliquée dans le cadre de l'électromagnétisme avec succès. Puisqu'on dispose dorénavant d'un modèle unifié où sont définis les concepts physico-numériques mis en œuvre (UML) ainsi que leurs contraintes d'utilisation (OCL), on s'attache dans un deuxième temps à l'exploitation de cette connaissance.

Etant donné la nature contractuelle des expressions OCL, nous avons choisi d'utiliser une technologie de type système expert pour son exploitation. Les invariants OCL ont alors été traduits en prédicats Prolog, constituant la base de règles manipulées par le système expert. Ce dernier a été encapsulé dans un environnement Java, au sein duquel se situe sa base de faits. En effet, celle-ci se présente sous la forme d'une bibliothèque d'instances gérée par l'utilisateur et constituant les éléments du dispositif modélisé. L'accès dynamique du système expert à cette bibliothèque a été implanté sous la forme de prédicats objet.

La traduction des invariants OCL en prédicats Prolog et l'encapsulation d'un système expert dans un environnement Java a permis l'évaluation des premiers sur des instances de classes. Ceci constitue déjà une **solution technique concrète et performante** pour l'implantation informatique de l'évaluation des expressions OCL. En outre, cette solution possède deux avantages, issus de la **synergie** entre système expert et formalisme objet :

- le système expert permet l'**inversion des expressions OCL**,
- les formalismes objet permettent de concevoir un **moteur d'inférences réutilisable.**

Gestion de la discontinuité de modèles

Sommaire

A ce stade de notre travail, nous disposons d'un système expert dont la base de connaissance est constituée du modèle de données objet structurel et contractuel d'une application, et la base de faits d'un ensemble d'instances de ce modèle de données.

L'accès à l'aspect structurel de la base de connaissance pour une application donnée a été implanté en utilisant le processus de réflexivité propre au langage Java, et non pas en utilisant directement un diagramme de classes UML.

Les expressions OCL contractualisant le modèle de données objet ont été traduites en fichier de règles Prolog. Cette traduction des expressions OCL en règles Prolog est complétement automatisable, mais non implantée.

La base de faits représente la modélisation effective d'un dispositif à simuler. Le moteur d'inférence codé permet :

- de valider la base de faits (validité des instances et de leurs relations par rapport aux contraintes OCL associées)
- de diagnostiquer une erreur de modélisation (exhibition des instances et leurs invariants infirmés)
- de proposer des choix valides (instances ou intervales de valeur) pour réaliser un attribut ou une association d'une instance particulière.

L'objectif de ce système expert fondé sur le modèle de données objet contractualisé d'une application est d'assister l'utilisateur lors de la modélisation de son dispositif. L'utilisation de formalismes standards pour la représentation du modèle de données objet contractualisé (UML et OCL) a deux avantages :

- premièrement, le système expert ainsi réalisé devient un composant logiciel autonome réutilisable dans toutes les applications respectant ces standards de la modélisation objet,
- deuxièmement, l'ontologie de l'application a été explicitée par ses concepteurs dans le modèle de données objet contractualisé, évitant ainsi une laborieuse extraction de connaissances a posteriori.

L'assistance dynamique à la modélisation d'un dispositif est suffisante pour assurer la validité des résultats de la simulation dans un cadre statique. Nous allons maintenant étudier le cas des régimes transitoires.

1 Introduction

L'objet de ce chapitre est la simulation numérique de dispositifs en régime transitoire. Le premier paragraphe ci-dessous illustre notre problématique avec deux exemples concrets. Le paragraphe suivant est dédié à l'analyse de ces exemples, et à l'introduction de l'axe de recherche que nous développerons dans la suite de ce travail. Cet axe est fondé sur une stratégie générale de résolution des problèmes en régime transitoire décrite dans le dernier paragraphe.

1.1 Deux exemples de limitations des modèles

Afin d'illustrer la problématique de la validité des simulations en régime transitoire, nous avons choisi de développer deux exemples : celui de la commutation d'un contacteur électromécanique, et un problème de conduction thermique pure.

1.1.1 La commutation d'un contacteur

La Figure 1 ci-dessous représente un contacteur électromécanique. Il se compose d'une palette mobile et d'une culasse fixe, en acier. A l'intérieur de la culasse se trouve une bobine. Un ressort maintient le contacteur en position ouverte. Le principe physique utilisé par ce contacteur est identique à celui du contacteur du tutoriel présenté au premier chapitre.

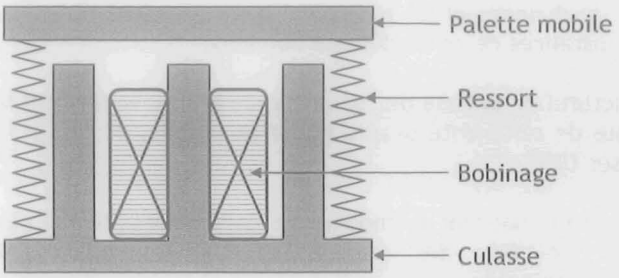


Figure 1 : Contacteur électromécanique

Pour modéliser ce contacteur, on dispose des formulations en potentiel scalaire, total ϕ ou réduit ϕ_r, en potentiel vecteur, normal A ou modifié A*, ainsi que les formulations AV et Tϕ. On supposera en outre que nous disposons de deux modèles de matériau et d'un modèle de source de courant :

- un matériau « vacuum » ayant un modèle de perméabilité linéaire isotrope (lin_iso) modélisant l'air,
- un matériau « steel » ayant un modèle de perméabilité non-linéaire (nonlin) et de conductivité isotrope constant (lin_iso) modélisant l'acier, et dans lequel sont susceptibles de se développer des courants induits,
- une source de courant « bobine » parcourue par un courant d'intensité I et de fréquence ω.

En position ouverte, le dispositif peut être modélisé par quatre régions volumiques (Figure 2). A chaque région est associée une formulation, un matériau et une éventuelle source de courant :

- R_{air} [ϕ_r ; vacuum; bobine] modélisant l'air avec l'inducteur,
- R_{ent} [ϕ ; vacuum] modélisant les entrefers larges,
- R_{cul} [Tϕ ; steel] modélisant la culasse,
- R_{pal} [Tϕ ; steel] modélisant la palette mobile.

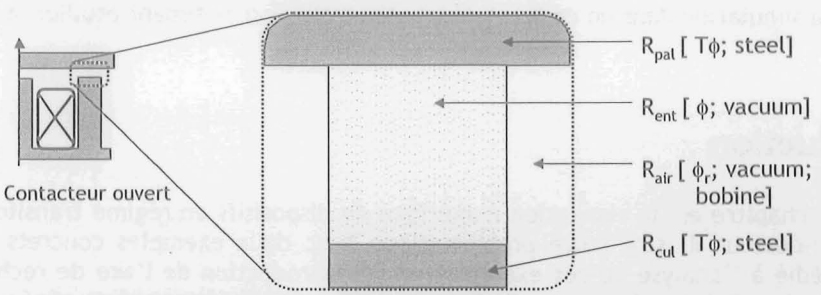


Figure 2 : Modélisation valide d'un contacteur à entrefer large

Cette modélisation utilise uniquement des formulations fondées sur des éléments finis volumiques. Elles nécessitent un maillage suffisamment fin des régions pour obtenir une solution acceptable, notamment dans les régions conductrices.

Lors du passage de la position ouverte à la position fermée, la dimension caractéristique des entrefers diminue de façon sensible, passant de quelques millimètres à celui du micromètre. Or, le maillage volumique des régions minces est problématique.

Dans le cas des entrefers minces maillés en trois dimensions, le système linéaire obtenu est volumineux et mal conditionné. La solution obtenue se dégrade avec la dimension de l'entrefer. Ce modèle est par conséquent inutilisable.

Or, dans le cas d'un entrefer mince, le champ magnétique H est presque normal au plan principal de ce dernier, ce qu'on peut assimiler à un saut du potentiel scalaire magnétique ϕ dont il dérive.

Pour prendre en compte ce phénomène, on utilise une formulation « ϕ_saut » dédiée à des éléments surfaciques avec sauts de potentiels.

A l'aide de cette formulation, le nouvel état du contacteur peut alors être modélisé par (Figure 3) :

- R_{air} [ϕ_r ; vacuum; bobine] modélisant l'air,
- S_{ent} [ϕ_saut ; vacuum] modélisant les entrefers minces,
- R_{cul} [Tϕ ; steel] modélisant la culasse,
- R_{pal} [Tϕ ; steel] modélisant la palette mobile.

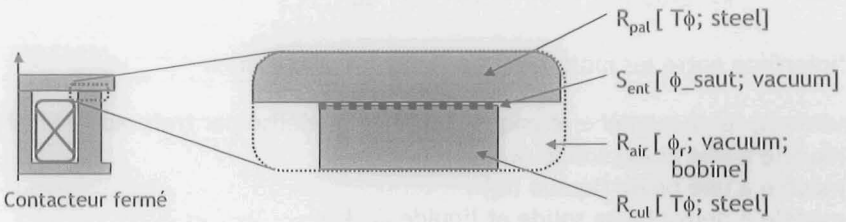


Figure 3 : Modélisation valide d'un contacteur avec entrefer mince

Lors de la simulation complète de l'ouverture ou de la fermeture du contacteur, le dispositif va rencontrer deux états distincts, selon l'épaisseur de l'entrefer. Ces deux états sont modélisés de deux manières différentes.

1.1.2 Un exemple thermique en régime transitoire

Prenons un exemple issu de la thermodynamique : l'élévation en température d'une billette d'acier. Du point de vu physico-mathématique, il s'agit d'un problème de diffusion classique modélisé par l'ensemble d'équations :

$$\frac{\partial T}{\partial t} - \text{div}(k.\text{grad}\,T) = q \text{ dans } \Omega_x \times [t_0;t_n] \quad (1)$$

$$T(x,t_0) = T_0(x,y) \text{ dans } \Omega_x \quad (2)$$

$$T = f \text{ sur } \Gamma_{dir} \times [t_0;t_n] \quad (3)$$

$$\text{grad}\,T.\vec{n} = g \text{ sur } \Gamma_{neu} \times [t_0;t_n] \quad (4)$$

où :

- **T**(x,t) est le champ de température en tout point (x,t) du domaine spatio-temporel d'étude $\Omega_x \times [t_0;t_n]$
- **k** est la conductivité thermique du matériau supposée constante isotrope, et **q** la source de chaleur
- $T_0(x)$ est la condition initiale
- f(x,t) est la valeur connue du champ de température sur $\Gamma_{dir} \times [0 ; t_{max}]$
- g(x,t) est la valeur connue de la dérivée normale de la température sur $\Gamma_{neu} \times [0 ; t_{max}]$
- **n** la normale extérieure au bord de Ω_x, constitué de Γ_{dir} et de Γ_{neu}

Les équations (1) à (4) modélisent un problème à une phase. Or, dans le cas d'une élévation en température, la nature du matériau dépend du champ de température. Notamment lors du passage du point de fusion, le matériau passe d'un état solide à un état liquide.

Le changement de phase induit l'apparition d'une nouvelle région, caractérisée par un nouveau modèle de matériau (liquide) ainsi qu'éventuellement de nouveaux phénomènes physiques tels que les mouvements de convection par exemple. De plus, des phénomènes à l'interface entre les deux régions peuvent se produire.

Si nous nous plaçons sous les hypothèses suivantes :

- les différents phases du matériau sont modélisées par des conductivités thermiques supposées constantes et isotropes, respectivement notées k_1 et k_2
- a l'état liquide, les mouvements de convection sont supposés négligeables par rapport à la conduction thermique pure
- il n'y a pas de phénomène de chaleur latente lors du passage du point de fusion.

Alors, le problème à deux phases est modélisé par les équations 1 à 4 auxquelles s'ajoute la condition de passage à l'interface Γ_{1-2} entre les deux milieux :

$$k_1 . \text{grad}\, T . \vec{n}_1 = k_2 . \text{grad}\, T . \vec{n}_2 \quad \text{sur} \quad \Gamma_{1-2} \qquad (5)$$

où Γ_{1-2} est l'interface entre les milieux de conductivité différentes.

Dans ces conditions, le phénomène physique peut être modélisé par trois modèles différents :

- le modèle à une phase solide (k_1)
- le modèle à une phase liquide (k_2)
- le modèle à deux phases solide et liquide (k_1-k_2)

1.2 Gestion de la discontinuité des modèles

Les deux exemples que nous venons de donner ci-dessus illustre la manière dont un dispositif en régime transitoire est susceptible d'être le siège de l'apparition ou de la disparition de phénomènes physiques. La prise en compte de ces transitions de modèles physiques est une des sources de difficultés rencontrées dans la simulation de ces dispositifs.

1.2.1 Position du problème

La mise en œuvre d'une simulation en régime transitoire nécessite tout d'abord de choisir une algorithmique d'évolution. Pour faciliter notre analyse, nous avons décidé de nous placer dans le cas le plus simple d'un schéma en pas à pas sur le temps explicite.

Le modèle d'un problème transitoire mis en œuvre dans ce type d'algorithme évolutif est composé d'une partie invariante, et une partie évolutive (Figure 4). La partie invariante du modèle, notée M_α, contient les informations physique et topologique du problème transitoire. Nous l'appellerons **modèle de topologie physique**.

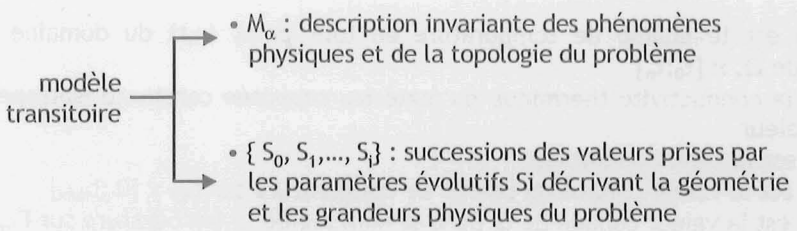


Figure 4 : Structure d'un modèle de problème transitoire

La partie évolutive, notée {S_0, S_1,..., S_n}, décrit la successions des valeurs prises au cours de la simulation par ses paramètres évolutifs. Ceci comprennent les caractéristiques géométriques et des grandeurs physiques du dispositif modélisé. La partie évolutive est dynamiquement construite par le solveur en fonction de son état antérieur et de la partie invariante du modèle (Figure 5).

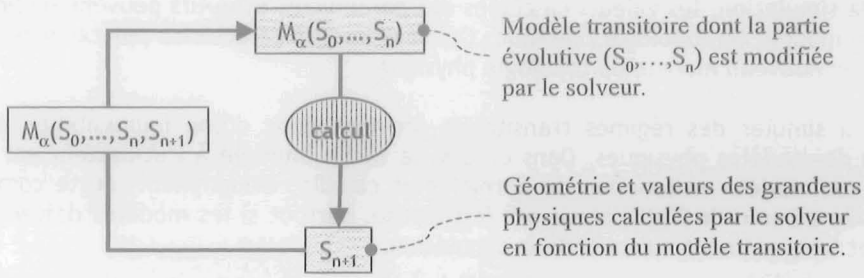


Figure 5 : Algorithme évolutif explicite pour les régimes transitoires

Dans l'exemple du contacteur, nous avons deux modèles de topologie physique différents. Chacun des modèles permet de calculer la valeur des paramètres évolutifs de la simulation.

Le premier modèle, à entrefer large, est uniquement fondé sur des éléments finis volumiques. Sa partie invariante est composée des quatre régions $R_{air}[\ \phi_R$; vacuum; bobine], $R_{ent}[\ \phi$; vacuum], $R_{cul}[\ T\phi$; steel] et $R_{pal}[T\phi$; steel]. Sa partie évolutive comprend la géométrie dont la position de la palette, la valeur des différents potentiels et du courant dans la bobine, ainsi que les forces électromagnétique et mécanique exercées sur la palette.

Le second modèle, à entrefer mince, nécessite l'emploi d'éléments finis volumiques et surfaciques. Sa partie invariante est composée des quatre régions $R_{air}[\ \phi_R$; vacuum; bobine], $S_{ent}[\ \phi\_saut$; vacuum], $R_{cul}[\ T\phi$; steel] et $R_{pal}[T\phi$; steel]. Sa partie évolutive comprend la géométrie dont l'épaisseur de l'entrefer mince, la valeur des différents potentiels et du courant dans la bobine, ainsi que les forces électromagnétique et mécanique exercées sur la palette.

Puisqu'il est impossible de changer de modèle de topologie physique au cours de la simulation, l'utilisateur va choisir et mettre en œuvre celui qui sera valide en fonction de l'état initial de son dispositif. Supposons que le contacteur soit initialement en position ouverte. Le modèle de topologie physique utilisé pour le modéliser sera donc le modèle à entrefer large.

Le passage d'un courant dans la bobine peut entraîner la fermeture du contacteur, c'est à dire une évolution de la géométrie caractérisée par une diminution de la taille des entrefers. En deçà d'une taille critique définie par le domaine de validité du modèle de topologie physique, ce dernier ne sera plus valide. Il faudrait alors mettre en œuvre le deuxième modèle, celui à entrefer mince.

Cependant, dans le cas où le courant dans la bobine n'est pas suffisamment fort, la position de la palette peut être modifiée, sans toutefois qu'il y ait fermeture complète du contacteur. Le modèle initial reste par conséquent valide pour la simulation complète.

Dans le problème thermique en régime transitoire, il y a trois modèles de topologie physique différents : le modèle à une phase solide, le modèle à deux phases solide et liquide et le modèle à une phase liquide. Au cours de l'échauffement en température, le dispositif peut être modélisé successivement par chacun des trois modèles.

Dans un logiciel traditionnel, à chaque pas de la simulation, l'obtention d'un résultat erroné indique que le modèle transitoire est devenu invalide. En d'autre termes, la valeur des paramètres évolutifs S_i de la simulation sort du domaine de validité du modèle de topologie physique M_α utilisé.

1.2.2 Discontinuité des modèles

Pour un logiciel de simulation donné, un même problème transitoire peut être modélisé de différentes manières. L'utilisateur choisi de mettre en œuvre un modèle particulier, qui soit valide en fonction de ce qu'il connaît du dispositif, c'est à dire son état initial. La création de ce modèle initial valide est d'ailleurs l'objet de la première partie de notre travail.

Au cours de la simulation, les valeurs calculées des paramètres évolutifs peuvent sortir du domaine de validité du modèle de topologie physique. Or, le dispositif caractérisé par ces valeurs peut être modélisé par un nouveau modèle de topologie physique.

La difficulté à simuler des régimes transitoires provient donc d'une impossibilité de gérer une commutation de modèles physiques. Dans ce cas, la tâche imposée à l'utilisateur est de repenser globalement son modèle en fonction du dernier état calculé. Evidemment, cette commutation de modèles laissée au soin de l'utilisateur est fastidieuse, surtout si les modèles doivent s'enchaîner rapidement et fréquemment au cours de la simulation.

Les logiciels traditionnels de simulation numérique ne sont généralement pas adaptés à la gestion d'une discontinuité de modèles, à l'exception notable du logiciel FluxExpert. Ce dernier met en œuvre des opérateurs de passage d'un modèle physique à un autre. Ces opérateurs sont associés à des filtres algorithmiques statiques.

On connaît la difficulté d'ajouter au sein d'un logiciel classique utilisant la programmation impérative un nouveau cas parmi ceux déjà traités. Ce n'est pas le cas de la programmation déclarative utilisée dans les systèmes experts, puisqu'il suffit simplement de rajouter un prédicat dans la base de règles.

Par conséquent, notre proposition est d'utiliser un système expert qui, compte tenu de l'historique de la simulation, soit susceptible de déterminer le modèle physique à adopter à tout moment, ainsi que l'opérateur de passage d'un modèle physique à un autre.

Cette étape de commutation, comprenant l'appel au système expert et la commutation du modèle physique, vient s'intégrer dans l'algorithme évolutif explicite de la Figure 5 de la façon suivante :

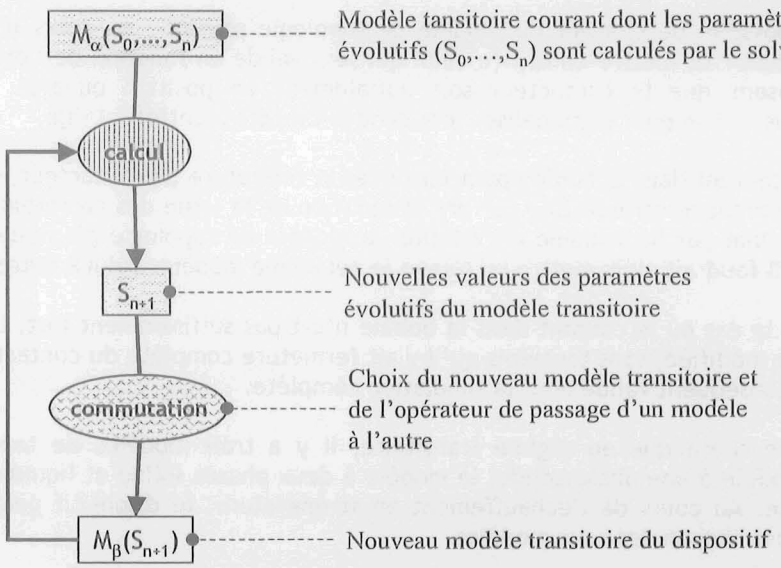


Figure 6 : Stratégie générale de résolution en transitoire

2 Commutation de modèle

Mettre en œuvre la commutation de modèles nécessite deux outils : posséder une description des modèles physiques disponibles, et savoir comment les enchaîner.

2.1 Le modèle physique potentiel

Toute la physique des milieux continus est fondé sur le triptyque géométrie/matériau/phénomène. On retrouve cette forme aussi bien dans les équations aux dérivées partielles (1-5) que dans l'implantation des logiciels, de manière implicite ou explicite comme les régions dans le logiciel Flux3D.

Le modèle physico-mathématique est composé d'un ensemble d'équations ayant chacune un domaine de définition (support géométrique). Les relations topologiques existant entre ces derniers sont représentées sous la forme d'un schéma, sur lequel sont également reportées les équations. L'ensemble constitué du système physico-mathématique et des relations topologiques entre leurs domaines de définitions est le **modèle physico-topologique** du problème (Figure 7).

Le modèle physico-topologique est le modèle abstrait le plus complet qu'on puisse avoir d'un problème traité, puisqu'il résume :

- les phénomènes (équations)
- le lieu topologique où se produit chaque phénomène (domaines de définitions des équations)
- les matériaux (propriétés des domaines de définitions)
- les relations topologiques entre les différents domaines de définitions.

$$\begin{cases} \dfrac{\partial T}{\partial t} - \operatorname{div}(k.\operatorname{grad} T) = q \quad \text{sur } (\Omega_1 \cup \Omega_2)\times[t_0; t_n] \\ T(x, y, t_0) = T_0(x, y) \qquad \text{sur } \Omega_{x,y} \\ T = f \qquad \text{sur } \Gamma_{dir} \\ \operatorname{grad} T.\vec{n} = g \qquad \text{sur } \Gamma_{neu} \\ k_1 \operatorname{grad} T.\vec{n}_1 = k_2 \operatorname{grad} T.\vec{n}_2 \quad \text{sur } \Gamma_{1\text{-}2} \end{cases}$$

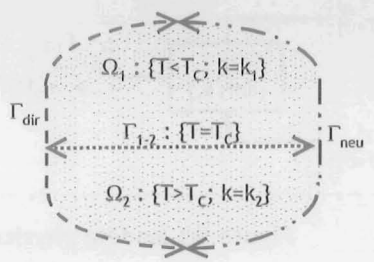


Figure 7 : Le modèle physico-topologique

Le modèle de la Figure 7 ci-dessus est le modèle physico-topologique d'un problème thermique en régime transitoire. La principale difficulté de ce modèle provient de l'apparition ou de la disparition d'éléments de la topologie (Ω_1, Ω_2, $\Gamma_{1\text{-}2}$), liée à la modification de leurs supports géométriques en fonction de la répartition de température (Figure 8).

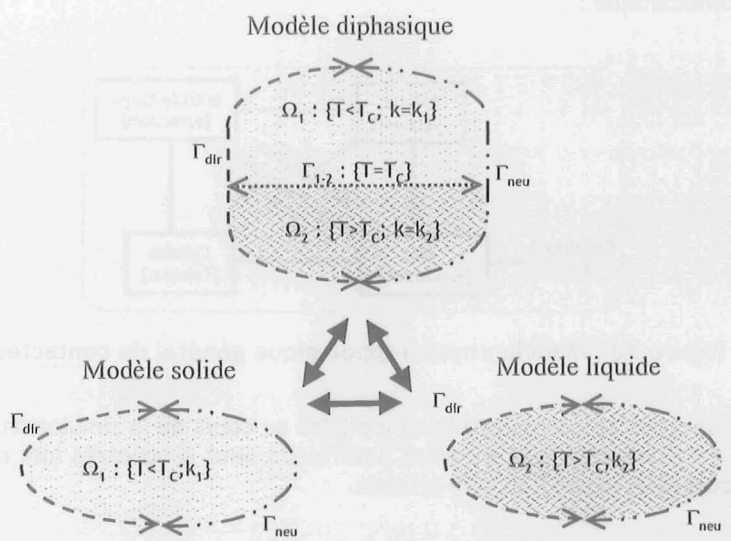


Figure 8 : Commutation de trois modèles physico-topologiques

On constate sur cet exemple que les éléments topologiques des modèles à une phase solide (Ω_1, Γ_{dir}, Γ_{neu}) et liquide (Ω_2, Γ_{dir}, Γ_{neu}) sont contenus dans le modèle à deux phases. Ce dernier est le modèle général de notre problème transitoire, au sens où il contient la description potentielle des différentes phases rencontrées par le dispositif au cours de la simulation.

Nous appellerons par la suite **modèle physique potentiel** le modèle général d'un phénomène physique transitoire. Ce modèle est constitué des éléments de topologie physique potentiellement présents au cours de la simulation.

Comme sur la Figure 8, la tradition représente souvent un modèle de topologie physique sous la forme de patatoïdes. Pour offrir une meilleure lisibilité d'un modèle physique potentiel, ses différents éléments physico-topologiques ainsi que leurs relations ont été formalisées.

La Figure 9 ci-dessous représente le modèle physique potentiel formalisé du problème thermique bidimensionnel en régime transitoire :

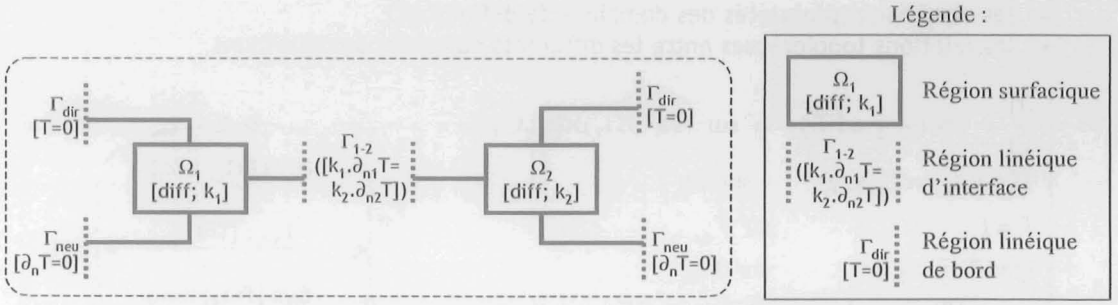


Figure 9 : Modèle physico-topologique potentiel du problème thermique

Chaque rectangle représente un élément topologique surfacique, associé à une formulation ('diff' = diffusion) et à un matériau (k_1, k_2). Les éléments topologiques de nature linéique sont représentés en trait pointillé, avec un trait pour un élément linéique de bord, deux traits pour une interface. Le type de phénomène est également précisé pour chacun : contraintes de Dirichlet (T=0) et de Neumann ($\partial_n T=0$) ou interface ($k_1\partial_{n1}T= k_2\partial_{n2}T$).

La Figure 10 représente le modèle physique potentiel associé au problème de la commutation du contacteur électromécanique :

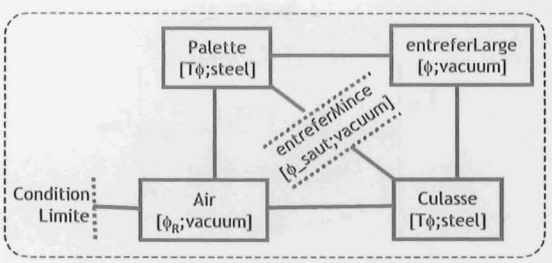


Figure 10 : Modèle physico-topologique général du contacteur

Notre objectif est de gérer la commutation de modèles au cours de la simulation. La première étape était de disposer de l'ensemble des modèles potentiellement rencontrés lors de la simulation. Le modèle physique potentiel répond à cette attente.

2.2 *Du modèle physique potentiel au modèle physique instancié*

Le modèle physique potentiel est l'élément fondamental autour duquel va s'articuler notre gestion des discontinuités de modèles pour la simulation en régime transitoire. Cependant, ce modèle est

totalement abstrait. Il ne peut pas être utilisé directement par un solveur pour résoudre un problème concret.

Au temps t_n de la simulation, l'obtention d'un modèle concret et valide du dispositif à partir du modèle physique potentiel nécessite de :

- détecter des éléments physico-topologiques présents, qui vont formés un modèle physico-topologique valide du dispositif (Figure 11-[2]),
- leur affecter un support géométrique (Figure 11-[3]).

Le **modèle physique instancié** ainsi obtenu est valide à un instant t_n de la simulation. Il va alors être utilisé par le solveur pour calculer la solution au pas de temps suivant.

Dans l'exemple thermique représenté sur la Figure 11, on suppose que :

- le modèle instancié courant est la section carrée d'une billette d'acier solide soumise à la diffusion d'une source de chaleur, avec une condition de Dirichlet au bord,
- le champ de température courant calculé à partir de ce modèle se situe de part et d'autre de la température de fusion du matériau.

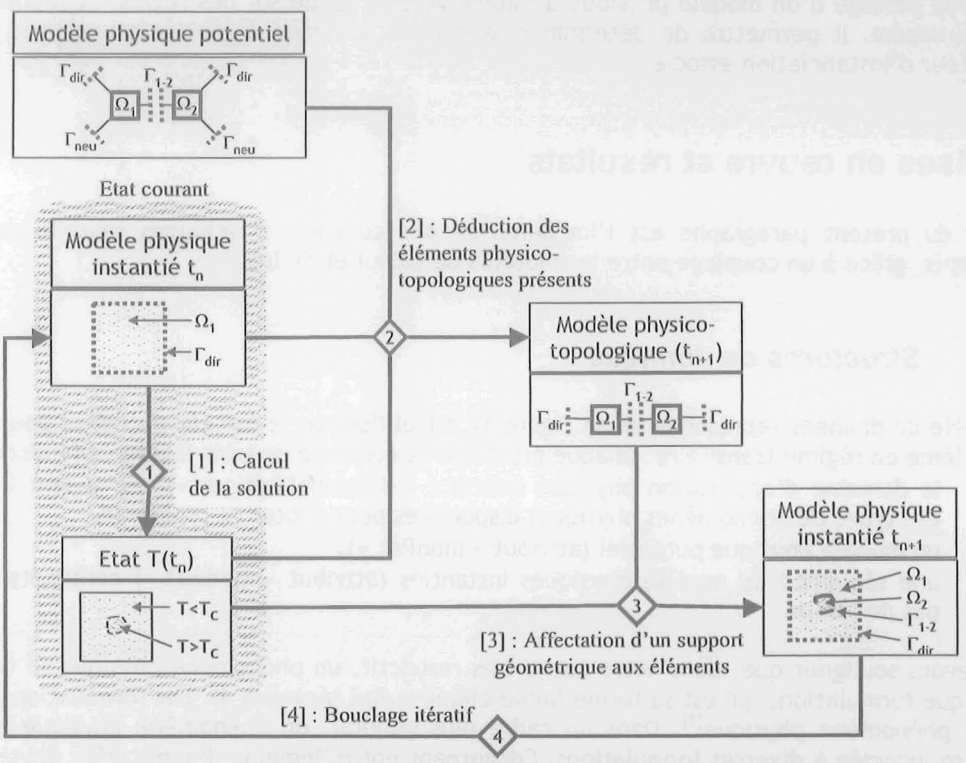


Figure 11 : Processus d'instantiation d'un modèle physique potentiel

Or, le problème thermique en régime transitoire est soumis à quatre règles simples :

- si la température minimale est en deçà de la température de fusion, alors il y a présence de matériau en phase solide.
- si la température maximale est au-delà de la température de fusion, alors il y a présence de matériau en phase liquide.
- si le matériau existe sous ses deux phases, alors il y a présence d'une interface.
- si une contrainte est spécifiée, alors elle est conservée.

L'application de ces règles à l'état courant nous indique que désormais, le modèle physico-topologique à appliquer est le modèle à deux phases avec contrainte de Dirichlet. Il suffit alors de déterminer et d'affecter les différents supports géométriques à chaque élément topologique pour obtenir le nouveau modèle physique instancié valide.

Chaque itération temporelle se compose donc de trois tâches :

- le calcul de l'état courant en fonction du modèle physique instancié valide courant
- en fonction des deux éléments sus-cités, déduction des éléments physico-topologiques actifs au sein du modèle physique potentiel à cet instant,
- l'affectation d'un support géométrique à ces éléments pour obtenir le nouveau modèle physique instancié

Le premier point est nécessairement effectué par le moteur de calcul, alors que le deuxième point, mettant en œuvre des règles pour effectuer une déduction, est réalisé par un système expert. La simulation transitoire est donc conjointement gérée par un moteur de calcul et un moteur logique.

2.3 Synthèse

L'une des difficultés rencontrées par les simulations de régimes transitoires est l'apparition ou la disparition de phénomènes physiques. Simuler des régimes transitoires va nécessiter de disposer de la description de ces différents phénomènes physiques : c'est le modèle physique potentiel. La gestion du passage d'un modèle physique à l'autre va être fondé sur des règles, appliquées par un système expert. Il permettra de déterminer le modèle physique à mettre en œuvre, ainsi que l'opérateur d'instanciation associé.

3 Mises en œuvre et résultats

L'objet du présent paragraphe est l'implantation des concepts développés dans le paragraphe précédents, grâce à un couplage entre les moteurs de calcul et de logique.

3.1 Structures de données

Le modèle de données représenté sur la Figure 12 est utilisé par le moteur de calcul pour résoudre un problème en régime transitoire. Chaque problème se compose de trois éléments fondamentaux :

- le domaine d'application physique (attribut « domainePhysique ») regroupant l'ensemble des types de phénomènes physiques disponibles pour traiter ce problème,
- un modèle physique potentiel (attribut « modPot »),
- une séquence de modèle physiques instantiés (attribut « modInst ») construits à chaque pas de temps.

Nous devons souligner que, dans notre cadre très restrictif, un phénomène physique se traduit par une unique formulation, qui est sa forme faible élément fini (**Annexe 4**). Les formulations jouent le rôle de phénomène physique[1]. Dans un cadre plus général, un phénomène physique particulier peut-être associée à diverses formulations. Concernant notre domaine d'application physique, nous n'avons implanté que quatre type de phénomène physique : un phénomène de diffusion surfacique, deux contraintes de Dirichlet et de Neumann et un phénomène d'interface.

Le modèle physique potentiel du problème est composé de domaine potentiel qui sont soit des régions surfaciques potentielles, soit des régions linéiques potentielles. A chaque domaine potentiel est associé un type de phénomène physique (formulation). Les attributs « voisins » et « bord_de » définissent bien les relation topologiques entre les différents domaines.

A chaque itération temporelle, un nouveau modèle physique instancié est construit et ajouté à l'attribut « modInst », permettant le calcul de la solution courante. Chaque modèle instancié possède une géométrie en représentation de frontières (attribut « géométrie ») composée d'éléments géométriques (classe abstraite « EltGéo »).

[1] Au sens de l'association UML (cf. Figure 12)

Tous les domaines instanciés formant le modèle physique instancié sont associés à un ensemble d'éléments géométriques qui sont le support du phénomène auquel ce domaine instancié est associé, via un domaine potentiel.

Il existe deux types de domaines potentiels : les régions surfacique et linéique. Le premier type est exclusivement associé à une région surfacique potentiel, lui-même exclusivement associé à une formulation surfacique. Le deuxième type est associé à une région linéique potentiel, lui-même associé à une formulation de type linéique. Chaque région surfacique possède un matériau où est définie la valeur de sa conductivité k.

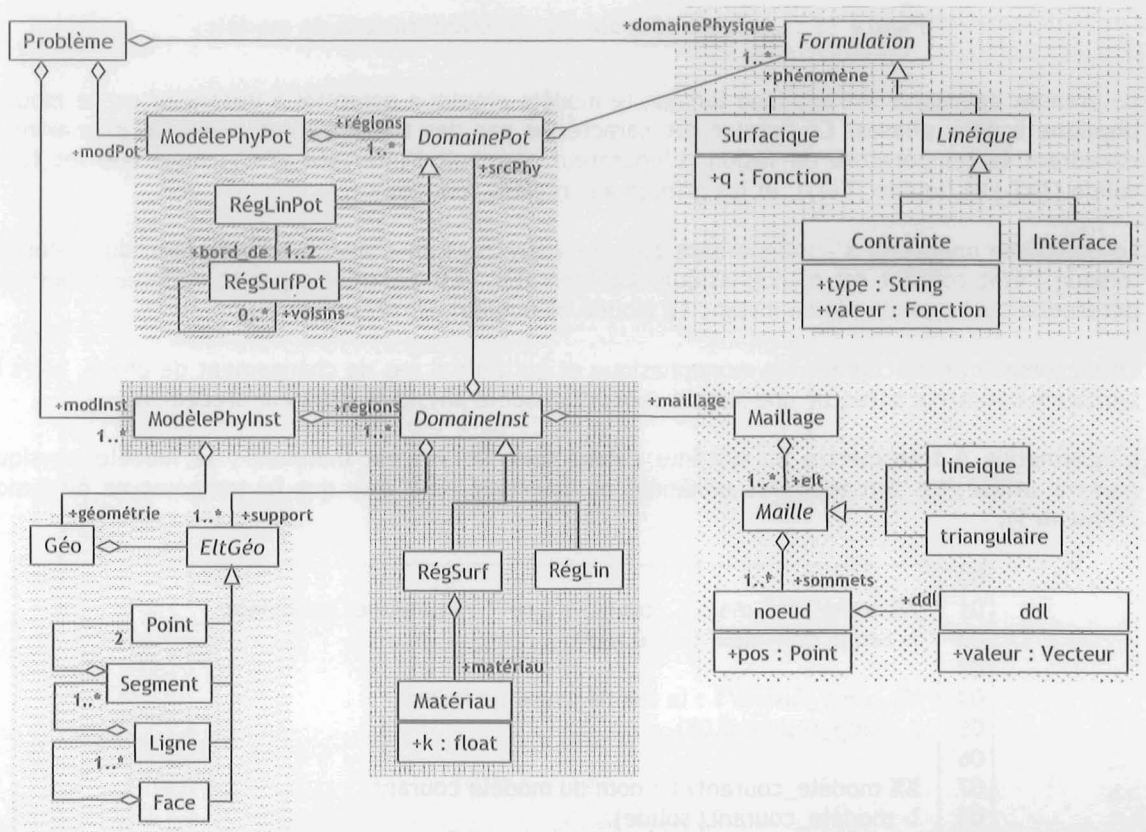


Figure 12 : Modèle de données

3.2 *Instantiation d'un modèle physique*

Le système expert que nous avons implanté est destiné à gérer la discontinuité de modèles. Son objectif est donc de permettre la construction dynamique de modèle physiques instanciés valides pour chaque itération temporelle.

A partir de l'état courant du dispositif, on sait mettre en œuvre des opérateurs algorithmiques capables de construire un modèle physique instancié particulier. La Figure 13 ci-dessous récapitule les trois opérateurs de commutations que nous avons implantés.

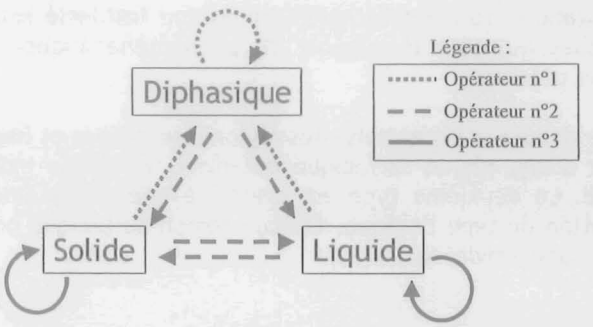


Figure 13 : Les trois opérateurs de commutations de modèles

Le premier opérateur est appliqué lorsque le modèle physique potentiel à instancier est le modèle physique à deux phases. Ce dernier est caractérisé par des températures minimale et maximale encadrant la température de fusion. L'opérateur construit l'interface ($T=T_F$) ainsi que les faces solide ($T<T_F$) et liquide ($T>T_F$), et les affecte aux régions concernées.

Le deuxième opérateur s'applique dans tous les autres cas de changement de phase du matériau, lorsque l'état courant est monophasique. Cet opérateur est paramétré par le système expert qui détermine quel type de région instanciée (solide ou liquide) est présente.

Enfin, lorsque l'état courant est monophasique et qu'il n'y a pas de changement de phase, alors le troisième opérateur effectue une simple copie du modèle physique instancié précédent.

L'information à transmettre au système expert dans ce cas est minimale : le modèle physique courant utilisé, les températures minimale et maximale ainsi que que la température de fusion (Tableau 1).

```
00
01  %% temp_min_max/2 : température[2] minimale et maximale
02  ?- temp_min_max( 0.0, 0.048).
03
04  %% temp_fusion/1 : la température de fusion
05  ?- temp_fusion( 0.05).
06
07  %% modèle_courant/1 : nom du modèle courant
08  ?- modèle_courant( solide).
09
```

Tableau 1 : Exemple de faits

La base de règles est elle aussi limitée. Elle est composée de deux prédicats (Tableau 2) :

- « état_courant/1 » qui, compte tenu de la base de faits, détermine le modèle physique à mettre en œuvre,
- « opérateur/1 » qui détermine ensuite l'opérateur de passage à appliquer pour instancier le nouveau modèle physique.

```
00
01  %% état_courant/1 : état courant du dispositif
02  ?- état_courant( solide_liquide ) :- temp_min_max( Min, Max),
03        temp_fusion( Fusion),
04        Min < Fusion,
05        Max > Fusion.
06
```

[2] Les « températures » indiquées sont des grandeurs réduites, issues des équations normalisées. Elles ne représentent donc pas les températures réelles au sein du matériau.

```
07  ?- état_courant( solide) :- temp_min_max( Min, Max),
08        temp_fusion( Fusion),
09        Max < Fusion.
10
11  ?- état_courant( liquide) :- temp_min_max( Min, Max),
12        temp_fusion( Fusion),
13        Min > Fusion.
14
15  %% opérateur/1 : nom de l'opérateur à appliquer
16  ?- opérateur( opérateurNo1 ) :-
17        état_courant( solide_liquide ).
18
19  ?- opérateur( opérateurNo2 ) :-
20        modèle_courant( solide_liquide ),
21        état_courant( EC),
22        inlist( EC, [solide, liquide]).
23
24  ?- opérateur( opérateurNo2 ) :-
25        ( modèle_courant( solide),
26          état_courant( liquide) ) ;
27        ( modèle_courant( liquide),
28          état_courant( solide) ).
29
30  ?- opérateur(opérateurNo3 ) :-
31        modèle_courant( Id),
32        état_courant( Id),
33        inlist( Id, [ solide, liquide]).
34
```

Tableau 2 : Le fichier de règles

3.3 *Résultats*

Nous avons résolu le problème thermique en régime transitoire suivant, avec des grandeurs normalisées :

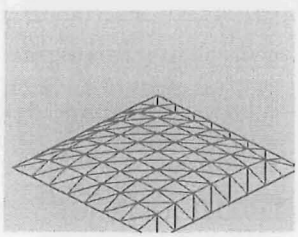
$$\frac{\partial T}{\partial t} - \text{div}\,(k\,\text{grad}\,T) = q \quad \text{sur } \Omega_{x,y} \times \Omega_t \qquad (1)$$

$$T_0(x, y) = 0 \qquad (2)$$

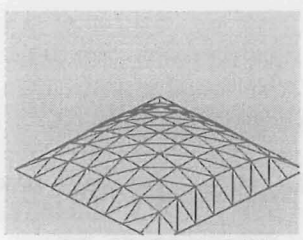
$$T = 0 \quad \text{sur } \partial\Omega_{x,y} \times \Omega_t \qquad (3)$$

$$k_1\,(\text{grad}\,T).\vec{n}_1 = k_2\,(\text{grad}\,T).\vec{n}_2 \quad \text{sur } \Gamma_{1-2} \qquad (4)$$

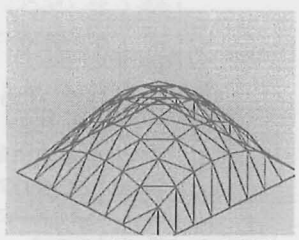
avec $\Omega_{x,y} \times \Omega_t = [\,0;1]^2 \times [\,0;\,0.5]$, maillé initialement avec huit mailles par direction, soit 81 nœuds et un pas de temps de 0.1. La conductivité k vaut 1 dans le solide, 2 dans le liquide, la température de fusion T_F = 0.05 et la source de chaleur est isotrope constante q = 1. Les différents champs de température pour calculés sont représentés ci-dessous :



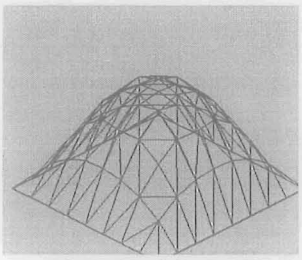
t=0.1 : T∈ [0; 0.03]



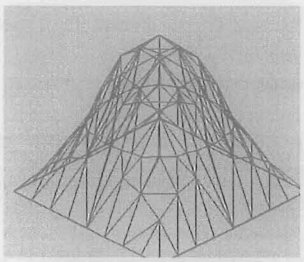
t=0.2 : T∈ [0; 0.048]



t=0.3 : T∈ [0; 0.068]



t=0.4 : T∈ [0; 0.89]



t=0.5 : T∈ [0; 0.12]

Tableau 3 : Résultats de simulation

On remarque sur la solution graphique le passage d'un modèle physique à une phase solide, à un modèle physique à deux phases. Cette constatation est confirmée par l'affichage du déroulement des étapes de la simulation :

```
00
01  Prolog IV v1.3.1 (161), Octobre 2000 (C)PrologIA 1995..2000
02  [ Tue Nov 14 15:11:48 MET 2000 ]
03
04  ?- compile ("/mnt/zip/PROLOG/GEOM/iso.p4"). true.
05  ?- iso. true.
06  ?- compile ("/mnt/zip/PROLOG/GEOM/physique.p4"). true.
07  ?- dynamic (temp_min_max/2). true.
08  ?- dynamic (temp_fusion/1). true.
09  ?- dynamic ( modèle_courant/2). true.
10  ?- assertz ( temp_fusion( 0.05) ). true.
11
12  >> Iter n°1 t=0.1 <<
13  >> maillage
14  >> integration/assemblage
15  domaine.integration('Fe Sol', 'diffusion dynamique')
16  domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
17  >> stockage
18  >> resolution
19  >> fin
20  #ddl(phy) = 81
21  min/max = 0.0 0.030106426611127002
22  >> commutation
23  ?- assertz( modèle_courant(solide) ). true.
24  ?- assertz( temp_min_max(0.0, 0.030106426611127002) ). true.
25  ?- état_courant( X ). X = solide, true.
26  ?- opérateur( X ). X = opérateurNo3, true.
27
28  >> Iter n°2 t=0.2 <<
29  >> maillage
30  >> integration/assemblage
31  domaine.integration('Fe Sol', 'diffusion dynamique')
32  domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
33  >> stockage
34  >> resolution
35  >> fin
36  #ddl(phy) = 81
37  min/max = 0.0 0.048545784775036335
38  >> commutation
39  ?- assertz( modèle_courant(solide) ). true.
40  ?- assertz( temp_min_max(0.0, 0.048545784775036335) ). true.
41  ?- état_courant ( X ). X = solide, true.
42  ?- opérateur ( X ). X = opérateurNo3, true.
```

```
43
44  >> Iter n°3 t=0.3 <<
45  >> maillage
46  >> integration/assemblage
47  domaine.integration('Fe Sol', 'diffusion dynamique')
48  domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
49  >> stockage
50  >> resolution
51  >> fin
52  #ddl(phy) = 81
53  min/max = 0.0 0.06837939232186857
54  >> commutation
55  ?- assertz( modèle_courant(solide) ). true.
56  ?- assertz( temp_min_max(0.0, 0.06837939232186857) ). true.
57  ?- état_courant ( X ). X = solide_liquide, true.
58  ?- opérateur ( X ). X = opérateurNo1, true.
59
60  >> Iter n°4 t=0.4 <<
61  >> maillage
62  >> integration/assemblage
63  domaine.integration('Fe Sol', 'diffusion dynamique')
64  domaine.integration('Fe Liq', 'diffusion dynamique')
65  domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
66  domaine.integration('interface', 'interface')
67  >> stockage
68  >> resolution
69  >> fin
70  #ddl(phy) = 95
71  min/max = 0.0 0.08925225765887307
72  >> commutation
73  ?- assertz( modèle_courant(solide_liquide) ). true.
74  ?- assertz( temp_min_max(0.0, 0.08925225765887307) ). true.
75  ?- état_courant ( X ). X = solide_liquide, true.
76  ?- opérateur ( X ). X = opérateurNo1, true.
77
78  >> Iter n°5 t=0.5 <<
79  >> maillage
80  >> integration/assemblage
81  domaine.integration('Fe Sol', 'diffusion dynamique')
82  domaine.integration('Fe Liq', 'diffusion dynamique')
83  domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
84  domaine.integration('interface', 'interface')
85  >> stockage
86  >> resolution
87  >> fin
88  #ddl(phy) = 102
89  min/max = 0.0 0.12027645882206645
90
```

Tableau 4 : Déroulement d'une simulation transitoire

La simulation se déroule comme suit :

- Lancement (l.01-02) et initialisation (l.04-10) de l'interpréteur Prolog. Ce dernier traite dynamiquement les requêtes envoyées par le programme principal Java.
- Boucle temporelle :
  - § Maillage de la géométrie,
  - § Construction du système linéaire (intégration/assemblage),
  - § Résolution,
  - § Constitution de la base de faits du système expert, et requête sur le nouveau modèle physique valide et l'opérateur à appliquer,

§ Application de l'opérateur pour passer à l'itération suivante.

Lors du déroulement de la simulation (Tableau 4), des requêtes sont dynamiquement envoyées et traitées par l'interpréteur Prolog. Une fois traitées, elles sont retournés avec leur résultat au programme principal Java, qui se charge de les afficher sur une ligne commençant par le symbole « ?- ». D'autres indications concernant l'état d'avancement de la simulation sont également affichées.

La première itération est fondé sur le modèle physique instancié solide. A l'issue de la résolution, la base de faits envoyé au système expert est la suivante :

modèle physique utilisé : solide (l.23)
températures minimale et maximale : 0 et 0.03 (l.24)

En fonction de cette base de faits, le système expert détermine le prochain modèle physique valide (l.25) et l'opérateur à appliquer afin de commuter les modèles. A la seconde itération, la température maximale est toujours inférieure à la température de fusion : on reste dans le cas précédent.

A la troisième itération cependant, la température maximale (l.56 : 0.068) est supérieure à la température de fusion (l.10 : 0.05). Le modèle physique utilisé (l. 55 : solide) n'est alors plus valide. Le système expert détermine le nouveau modèle physique (l.57 : solide_liquide) et l'opérateur de commutation adéquat (l.58 : opérateurNo1).

Enfin, à l'issue de la quatrième étape de résolution (l.68), la base de faits dynamiquement déclarée au système expert est composée du modèle physique utilisé solide_liquide (l.73) et des températures 0 et 0.089 (l.74). En fonction de ces nouveaux faits, le système expert détermine que le nouveau modèle physique à instancier sera le modèle à deux phases solide_liquide (l.75) grâce à l'utilisation de l'« opérateurNo1 ».

4 Conclusion

Ce chapitre est dédié à la part de l'expertise des concepteurs de logiciels de simulations numériques portant sur la modélisation des dispositifs en régimes transitoires. La difficulté de simuler de tels dispositifs provient en grande partie de la nécessité de gérer les discontinuités de modèles physiques potentiellement présents au cours de la simulation.

La solution usuelle pour traiter les simulations transitoires est de développer un automate spécifique au problème modélisé, chargé de la gestion de la commutation d'un ensemble identifié de modèles. Ceux-ci sont définis statiquement par les concepteurs de l'automate. Cette approche est performante, mais pose des difficultés de maintenance : la prise en compte d'un nouveau modèle physique nécessite de développer un nouvel automate. Afin de dynamiser cette approche, nous avons introduit dans le modèle de données du logiciel de simulation le concept de modèle physique potentiel. Il permet à l'utilisateur de décrire les phénomènes physiques potentiellement présents.

Un modèle physique potentiel conçu par un utilisateur n'est pas exploitable en tant que tel par le solveur. Il doit être instancié. Dans le cadre de notre travail, une simulation est décomposée en une succession de modèles physiques instanciés. La description d'un nouveau modèle physique est possible grâce à l'application d'opérateurs de commutation. Ce type d'approche, utilisant des opérateurs de commutation, a déjà été mise en œuvre avec succès au sein de logiciels tel que FluxExpert par exemple. Dans ce dernier cependant, le choix des opérateurs est effectué par l'évaluation de filtres statiques algorithmiques. Pour notre part, nous préconisons l'utilisation d'un système expert, qui est une technologie beaucoup plus souple pour la description et l'évaluation des filtres.

Conclusions et Prospectives

L'ensemble du travail présenté dans cet ouvrage est fondé sur le constat qu'une partie non négligeable des compétences et du savoir-faire des concepteurs de logiciels de simulations numériques n'est pas exploitée : leur expertise. La nature de celle-ci, essentiellement non algorithmique, ne facilite ni sa description ni son exploitation au sein de logiciels écrits principalement dans des langages de programmation usuelle ou objet. Il existe cependant une forme de programmation parfaitement adaptée à la manipulation de connaissances faiblement structurées : la programmation déclarative, sur laquelle est fondée la technologie des systèmes experts.

Notre travail de réflexion a d'abord porté sur l'identification de l'expertise, afin de déterminer la meilleure manière de l'introduire au sein de l'application. De ce travail préliminaire à toute exploitation, nous avons discerné deux champs d'exploitation possibles : d'une part les domaines de validité des concepts physico-numériques mis en œuvre au sein du logiciel lors de la modélisation des dispositifs, et d'autre part la gestion des discontinuités de modèles dans le cadre des régimes transitoires.

1 Assistance dynamique à la modélisation

1.1 Conclusions

Lors de la conception d'un système objet, la première étape consiste à spécifier les concepts mis en œuvre, les opérations et les contraintes du système. UML et OCL sont les deux langages normalisés standards pour spécifier un système objet. Or, dans le cadre de logiciel de simulation numérique, les concepts manipulés appartiennent au domaine de la physique modélisée. Par conséquent, ils possèdent des domaines de validités. Ceux-ci sont très peu exploités au sein de l'application elle-même car les langages informatiques, à quelques rares exceptions près tel que Eiffel, ne possèdent pas les structures syntaxiques adéquats à leur description, et encore moins à leur évaluation.

Nous avons utilisé le langage OCL afin de formaliser les domaines de validité des concepts physiques décrits dans le modèle de données UML d'une application électromagnétique. On obtient ainsi un modèle unifié où se trouve déclaré sous un formalisme normalisé standard l'ensemble des connaissances algorithmiques et expertes des concepteurs du logiciel.

Ce modèle unifié va être le support du développement de deux moteurs distincts, l'un algorithmique (le solveur), l'autre logique. Pour notre part, nous nous sommes concentré sur l'implantation informatique de ce dernier. La nature contractuelle des expressions OCL nous a amené à choisir la technologie des systèmes experts pour l'implantation du moteur logique. Ceci a nécessité notamment la traduction des invariants OCL en prédicats Prolog. Par ce biais, non seulement le système expert met en œuvre l'évaluation des expressions OCL sur des instances de classes, mais en plus il permet leur inversion.

L'utilisation du langage OCL pour décrire les **domaines de validité** des concepts du domaine physique modélisé est originale. En effet, la majorité des travaux en cours actuellement concernant OCL ont pour cadre le développement de logiciels critiques, et ont pour objet la validation du code conçu. L'exploitation des expressions OCL dans ce cas est réservée aux concepteurs d'applications. Dans notre cas, elles sont un vecteur de connaissances depuis ces premiers à destination des utilisateurs.

D'autre part, notre proposition de traduire les expressions OCL en prédicats Prolog, afin de les manipuler sur des instances de classes Java, par le biais d'un système expert encapsulé, est une solution élégante et originale pour l'implantation de l'évaluation de ces expressions. Les prédicats obtenus étant **inversibles**, cette implantation va au-delà des spécifications strictement préconisées par le langage OCL.

Le dernier point important à souligner dans cette première partie est l'apport de l'utilisation de formalismes standards et normalisés tels que UML et OCL pour la technologie des systèmes experts. En effet, les moteurs d'inférences généralement développés sont peu réutilisables, car fondés sur

des formalismes spécifiques à chaque domaine d'application, voir à chaque expert du domaine. Le développement d'un système expert sous la forme d'un **composant logiciel réutilisable** est possible sous la condition sine qua non que l'ontologie du domaine de l'expertise soit décrite à l'aide de formalismes standards. C'est l'apport méthodologique fondamental d'UML et d'OCL à la technologie des systèmes experts.

1.2 Prospectives

Les prospectives de cette première partie de notre recherche sont nombreuses. Premièrement à court terme, il est indispensable de finaliser l'implantation de l'ensemble des éléments syntaxiques du langage OCL en terme de prédicats objet Prolog. Notre attention se portera notamment sur les **pré et post-conditions** associées aux opérations, et qui ne font pas fait l'objet d'une implantation en l'état actuel d'avancement.

A ce stade de réalisation, il faudra automatiser la tâche de **traduction** des expressions OCL en prédicats objet Prolog. En effet, comme nous l'avons vu, l'écriture des expressions OCL sous la forme de prédicats est peu élégante, et rapidement lourde. Ces inconvénients sont dus à notre volonté qu'à chaque élément de la syntaxe OCL corresponde un prédicat objet Prolog particulier. L'isomorphisme ainsi établi entre les deux syntaxes autorise un développement rapide d'un automate de traduction des expressions OCL en prédicats Prolog. Enfin, cet isomorphisme est la condition sine qua non garantissant la réutilisabilité du composant logiciel ainsi que la pérennité des développements futurs.

A ce stade, plusieurs options sont envisageables à long terme. La première consiste à offrir un environnement convivial de modélisation de dispositifs fondé sur le modèle unifié d'un logiciel de simulations numériques. Les concepteurs de ces applications peuvent alors se consacrer à leur cœur de métier, à savoir la simulation proprement dite des phénomènes physiques. On raccourcit ainsi le délai de production du logiciel. De plus, l'assistance dynamique à la modélisation est un outil permettant de réduire le temps de conception d'un modèle de dispositif. Son intégration dans un logiciel de simulation est donc un atout pour les bureaux d'études. C'est également un vecteur de connaissance du domaine physique modélisé, et par conséquent un outil pédagogique.

Enfin, il existe actuellement au sein de la communauté du génie logiciel des travaux portant sur le développement de plateformes d'intégration des solveurs existants. C'est par exemple le cas du RNTL Salomé auquel participe notre laboratoire. Ces plateformes multi-physiques ne peuvent pas exister sans l'usage de la modélisation et de la méta-modélisation. Ces dernières permettent la description et la manipulation des concepts physiques modélisés. Par conséquent, elles sont le terrain de prédilection pour l'introduction et l'exploitation des domaines de validités.

2 Gestion des discontinuités de modèles

2.1 Conclusions

La simulation correcte d'un dispositif en régime transitoire est rendue difficile par l'apparition ou la disparition de phénomènes physiques. Le modèle conçu à l'origine par l'utilisateur, même s'il était valide à l'instant initial de la simulation, peut ne pas prendre en compte des phénomènes physiques transitoires. Nous avons par conséquent développé les concepts de modèles physiques potentiel et instancié. Le premier contient l'ensemble des phénomènes physiques potentiellement présents au cours de la simulation. Le second type de modèle, issu du premier, correspond au modèle valide du dispositif à un instant donné de la simulation. Il est mis en œuvre par un opérateur d'instanciation.

La gestion de la discontinuité des modèles instanciés est traitée avec succès par un système expert. Cette technologie est d'une grande souplesse pour la description des règles de commutations des modèles et du choix d'un opérateur.

2.2 Prospectives

Dans la maquette que nous avons développée, la notion de modèles potentiel et instancié ne s'applique qu'au phénomène physique. Il y a d'ailleurs une assimilation simplificatrice et abusive que nous avons signalée, concernant les notions de phénomène physique et de traitement numérique associé. Or, un même phénomène physique possède plusieurs traitements numériques distincts. La conception d'un logiciel multi-physiques et multi-méthodes passe donc par la distinction entre ces deux concepts au niveau du modèle de données.

Enfin, le modèle potentiel que nous avons développé est essentiellement de nature statique. Il faudrait élargir le cadre de cette étude à l'algorithmique temporelle avec, par exemple, l'introduction du concept de macro-algorithme.

3 Conclusion finale

Il est temps de conclure la réflexion que nous avons engagée dans le cadre de ce travail de thèse. Nous pensons que l'avenir des futurs logiciels de simulations numériques, qu'ils soient mono ou multi physiques telles que les plateformes d'intégrations, passe obligatoirement par l'introduction et l'exploitation de l'expertise non algorithmique de leurs concepteurs. Les outils, qu'ils soient de nature technologique tels que les systèmes experts et la programmation objet, ou méthodologique tels que les formalismes UML, OCL et au-delà la méta-modélisation, sont matures.

Du point de vue de la conception amont, les applications posséderont un modèle unifié, spécifiant l'ensemble des connaissances des concepteurs, et notamment l'ontologie complète, conceptuelle et contractuelle, du domaine physique modélisé. La mise en œuvre informatique sera réalisée au sein d'un environnement bipolaire où cohabiteront un moteur de calcul et un moteur de logique.

Nous espérons avoir démontré que la convergence de l'algorithmique et de l'expertise au niveau de la conception, du calcul et de la logique au niveau de l'implantation informatique, est non seulement réalisable, mais fortement souhaitable, car vecteur de synergies profondes débloquant un certain nombre de verrous technologiques actuels.

Bibliographie

Sommaire

1 Génie logiciel objet

1.1 Concepts de la modélisation objet

[1.1.1] - Booch G. : « Analyse et conception orientées objets ». Ed. Addidon-Wesley France, 1997

[1.1.2] - Meyer B. : « *Object-oriented software construction* ». CAR HOARE series editor, Prentice Hall, 1988

[1.1.3] - Rumbaugh J. : « *OMT : modélisation et conception orientées objet* ». Paris : Masson, 1996

1.2 UML

[1.2.1] - Object Management Group : « UML specifications ». http://www.omg.org

[1.2.2] - Lai M. : « *UML : la notation unifiée de modélisation objet : applications en Java* ». Paris : InterEditions, 1997

[1.2.3] - Muller P.A. : « Modélisation objet avec UML », Paris : Eyrolles, 2ème édition, 2000

[1.2.4] - Booch G., Rumbaugh J., Jacobson I. : »Le guide de l'utilisateur UML ». Paris : Eyrolles, 2000.

1.3 OCL

[1.3.1] - Object Management Group : « OCL specifications ». http://www.omg.org

2 Systèmes experts et Prolog

2.1 Systèmes experts

2.1.1 Dans la communauté du génie électrique

[2.1.1.1] - Darnault R. : « *Système expert d'aide à la conception : application au positionnement d'appareils électriques* ». Th : Génie électrique : INPGrenoble : 1995

[2.1.1.2] - Escande E. : « *Modélisation "objet" du processus de conception dans le domaine du génie électrique : application au cas de la machine asynchrone, le système OPUS* ». Th : Génie électrique : INPGrenoble : 1992

[2.1.1.3] - François F., Escande E., Bigeon J. : « *A design methodology through expert system in the domain of the electrical engineering* ». Proceedings de COMPUMAG93

[2.1.1.4] - Gerbaut L. : « *Aide à la conception des ensembles machines-convertisseurs-commande. Apport d'une démarche système expert* ». Th : Génie electrique : INPGrenoble : 1993

[2.1.1.5] - Gentilhomme A. : « *C.O.C.A.S.E : un système expert d'aide à la conception*

d'appareillages électriques ». Th : Génie électrique : INPGrenoble : 1988

[2.1.1.6] - Lowther D. : « *The automated design of electromagnetic devices. Fact or fiction ?* ». ICS Newsletter, pp. 6-9, 1995

2.1.2 Et ailleurs

[2.1.2.1] - Farreny H. : « *Les systèmes experts : principes et exemples* ». Cepadues editions, 1989

[2.1.2.2] - Liebowitz J. : « *Expert systems for business et management* ». Yourdon Press, Prentice Hall, 1990

[2.1.2.3] - Gardin J.C. et al. : « *Systèmes experts et sciences humaines : le cas de l'archéologie* ». Eyrolles, 1987

[2.1.2.4] - Gallouin J.F. : « *Transfert de connaissances : systèmes experts, techniques et méthodes* ». Editions Eyrolles, 1988

[2.1.2.5] - Massé P. : « *Intégration de l'activité de conception dans l'enseignement assisté par ordinateur : le système C.A.S.C.A.D.E* ». Th : INPGrenoble : 1975

2.2 Langage Prolog

[2.2.1] - International Organization for Standardization : « ISO/IEC 13211-1 Programming Language Prolog part 1 -- General Core », 1995

[2.2.2] - International Organization for Standardization : « ISO/IEC 13211-1 Programming Language Prolog part 2 -- Modules », 2000

[2.2.3] - Collard P. : « *Programmation déclarative et impérative en Prolog* ». Paris, Masson, 1992

[2.2.4] - Elbaz J. : « Programmer en Prolog ». Paris : Ellipses, 1991

[2.2.5] - Colmerauer A., Giannesini F. : « *Prolog* ». Paris : InterEditions, 1985

[2.2.6] - Société PrologIA : « *PrologIV : Manuel d'utilisation* ». Parc technologique de Luminy - Case 919, 13288 Marseille cedex 09 - FRANCE, 1998

[2.2.7] - Llyod M. : « Foundations of logic programming ». Springer-Verlag, 1985

[2.2.8] - Shapiro M., Sterling Y. : « The art of PROLOG ». MIT Press, 1987

3 Génie électrique

3.1 Méthodes numériques pour l'électromagnétisme

[3.1.1] - Dhatt G., Touzot G. : « *Une présentation des éléments finis* ». Collection Université de Compiègne. Maloin, 1984

[3.1.2] - Coulomb J.L. : « *Analyse tridimensionnelle des champs électriques et magnétiques par la méthode des éléments finis* ». Th : Génie électrique : INPGrenoble :1981

[3.1.3] - Chari M.V.K, Salon S.J : « *Numerical methods in electromagnetism* ». Academic Press, 2000

[3.1.4] - Bossavit A. : « Computational electromagnetism : variationnal formulations, complementarity, edge elements ». Electromagnetism. Academic Press, San Diego, USA, 1998

[3.1.5] - Biro O., Preis K. : « On the use of the magnetic vector potential in the finite element analysis of the three dimensionnal eddy-currents ». IEEE Transactions on magnetism, vol. 25, no.4, pp.3145-3149, 1989

[3.1.6] - Albanese R., Rubinacci G. : « Magnetostatic field computations in terms of two component vector potential ». International Journal for Numerical Methods in Engineering, vol.29, pp.515-532, 1990

[3.1.7] - Meunier G., Salon S.J., Coulomb J .L., Krahenbuhl L. : « Hybrid finite element-boudary element solutions for three dimensionnal scalar potential problems ». IEEE Transactions on magnetism, vol.22, no.5, 1986

[3.1.8] - Biro O., Preis K., Vrisk G., Richter K.R., Ticar I. : « *Computation of 3D magnetostatic fields using a reduced scalar potential* ». IEEE Transaction on magnetism, vol.29, no.2, pp. 1329-1332, 1993

[3.1.9] - Herault C. : « Vers une simulation sans maillage des phénomènes électromagnétiques ». Th : Génie électrique : INPGrenoble : 2000

[3.1.10] - Bossavit A. : « *Le calcul des courants de Foucault en trois dimensions, en présence de corps à haute perméabilité magnétique* ». Revue de Physique Appliquée, vol.23, pp.1147-1205, 1988

[3.1.11] - Leconte V. : « *Simulation des convertisseurs électromécaniques* ». Th : Génie électrique : INPGrenoble: 2000.

[3.1.12] - Leconte V., Mazauric V., Meunier G., Maréchal Y. : « *Comparing FEM-BEM coupling for large air-gap deformations* ». Proceedings de CEFC2000, p.318

[3.1.13] - Stroll R.L. : « *The analysis of eddy currents* ». Monographs in electrical and electronic engineering, Clarendon press, 1974

[3.1.14] - Jouguet M. : « *Courant de Foucault et fours à induction* ». Gaulthier-Villars, 1944

3.2 Flux3D

[3.2.1] - Coulomb J.L., Sabonnadière J.C. : « *Eléments finis et CAO en électrotechnique* ». Hermes Publishing, 1985

[3.2.2] - Albertini J.B. : « *Contribution à la réalisation d'un logiciel de modélisation de phénomènes électromagnétiques en 3D par la méthode des éléments finis : FLUX3D* ». Th : Génie électrique : INPGrenoble : 1988

[3.2.3] - Société CEDRAT : « *Flux3D : manuel de référence* ». 2002

[3.2.4] - Société CEDRAT : « *Flux3D : tutoriel de magnétostatique* ». 2000

[3.2.5] - Société CEDRAT : « *Note de principes de Flux3D version 3.1* ». 1999

[3.2.6] - Maréchal Y. : « *Modélisation des phénomènes magnétostatiques avec terme de transport : application aux ralentisseurs électromagnétiques* ». Th : Génie électrique :INPGrenoble :1991

[3.2.7] - Golovanov C. : « *Développement de formulations éléments finis 3D en potentiel vecteur magnétique : application à la simulation de dispositifs électromagnétiques en mouvement* ». Th : Génie électrique : INPGrenoble :1997

[3.2.8] - Brunotte X. : « *Modélisation de l'infini et prise en compte de régions magnétiques minces. Application à la modélisation des aimantations des navires* ». Th : Génie électrique : INPGrenoble : 1991

[3.2.9] - Zgainski F.X. : « *Un pré-processeur pour l'électromagnétisme , l'électromécanisme et l'électroacoustique* ». Th : Génie électrique : INPGrenoble : 1996

[3.2.10] - Guérin C. : « *Détermination des pertes par courants de Foucault dans les cuves de transformateurs : modélisation de régions minces et prise en compte de la saturation des matériaux magnétiques en régime harmonique* ». Th : Génie électrique : INPGrenoble : 1994

[3.2.11] - Razek A.A., Coulombn J.L., Feliachi M., Sabonnadière J.C. : « Conception of an air-gap element for the dynamic analysis of the electromagnetic field in electric machines ». IEEE Transaction on magnetism, vol.18, pp.655-659, 1982

[3.2.12] - Coulomb J.L., Duterail Y., Meunier G. : « Flux3D : a finite element package for magnetic computation ». IEEE Transactions on magnetics, vol.21, no.6, pp.2499-2503, 1985

4 Divers

[4.1] - Godlewski E., Raviart P.A. : « *Hyperbolic systems of conservation laws* ». Collection SMAI. Paris : Ellipse,1991

[4.2] - Massé, P : « *Analyse méthodologique de la modélisation numérique des équations de la physique des milieux continus à l'aide de la méthode des éléments finis : flux-expert, un système d'aide à la construction de logiciels* ». Th : Sciences : INPGrenoble : 1983

Annexes

Sommaire

1 API Java Native Interface

Le langage Java est organisé en package, chaque package offrant un certain nombre de classes implantant une fonctionnalité particulière. Dans notre cas, nous avons besoin d'encapsuler le système expert, codé en C, dans l'environnement Java. Le package dédié à ce service s'appelle Java Native Interface.

Utiliser ce package revient à déclarer des méthodes Java comme étant natives. Une méthode native n'est pas implémentée dans la classe Java, mais uniquement déclarée, comme une interface. C'est pourquoi de telles méthodes sont qualifiées d'interfaces natives.

Le corps de la méthode est implanté dans une fonction écrite dans un fichier C. Le fichier d'en-tête contenant la déclaration de cette fonction est généré automatiquement à partir de l'exécutable javah (cf. **Figure 1**).

D'autre part, un autre fichier d'en-tête (fichier jni.h) contient la déclaration de structures de données C qui assurent le passage des paramètres lors de l'appel de l'interface native Java à la fonction C. Il contient également la déclaration de fonctions C permettant la manipulation de ces structures de données.

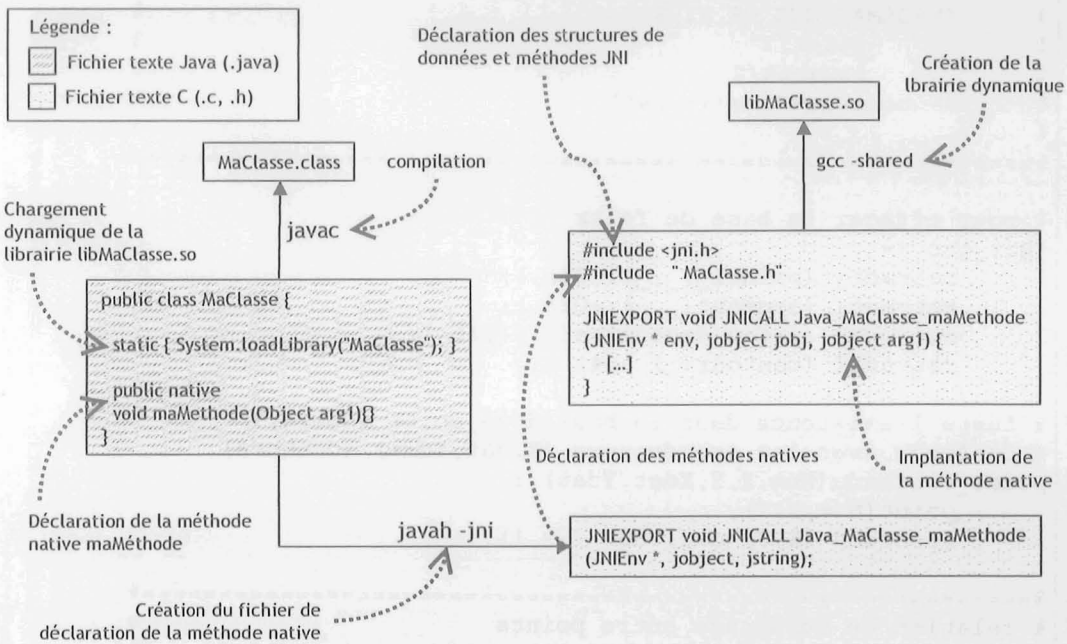


Figure 1 : utilisation de Java Native Interface

2 Programme Prolog de reconstruction de la géométrie

L'objet du programme Prolog présenté dans cette annexe est de reconstruire des éléments géométriques de haut niveau (lignes, contours, boucles) à partir des éléments les plus bas niveau, à savoir les points et les segments.

```
00
01 %============================================================%
02 % fichier topo.p4 : essai sur la topo2D                      %
03 % auteur : O. Defour                                         %
04 % date   : 15/02/2001                                       %
```

```
%=============================================================%
%                                                             %
% ATTENTION : la consultation de ce fichier necessite les     %
%     declarations prealables suivantes dans PIV              %
%                                                             %
%     1- Predicats Dynamiques :                               %
%        ?- dynamic(ligne_degeneree/1).                       %
%        ?- dynamic(contour/1).                               %
%        ?- dynamic(boucle/1).                                %
%                                                             %
%     2- Predicats Java :                                     %
%        ?- init_server("127.0.0.1").                         %
%        ?- new_java_pred("angle",7,"angle","Trigo",          %
%           "file:/home/defour/PROLOG/DIALOG/").              %
%                                                             %
%     3- Autres Predicats :                                   %
%        ?- consult("liste.p4").                              %
%        ?- consult("fig1.p4").                               %
%                                                             %
%=============================================================%

%=============================================================%
%     INFORMATIONS DE NIVEAU 0D :                             %
%           point/3                                           %
%           segment/2                                         %
%     ?- consult("fig[i].p4").                                %
%                                                             %
%=============================================================%

% pour effacer la base de faits
init_geo :-
      retract( (point(_,_,_):-Q1) );
      retract( (segment(_,_):-Q2) );
      retract( (ligne_degeneree(_):-Q3) );
      retract( (contour(_):-Q4) ).

% teste l'existence dans la base d'un point (Nom,X,Y)
% confondu avec les coordonnees (X,dat,Ydat) (donnees).
point_confondu(Nom,X,Y,Xdat,Ydat) :-
      point(Nom,X,Y),
      (confondu(X,Y,Xdat,Ydat) -> !).

%=============================================================%
% relation de voisinage entre points                          %
%=============================================================%

voisin(P1,P2) :-
      segment(P1,P2);
      segment(P2,P1).

nombre_voisin(P,N) :-
      point(P,_,_),
      setof(X,voisin(P,X),L),
      size(N,L).

%=============================================================%
% classification des points selon leur nombre de voisins :    %
%   #voisin = 1 -> point extremite                            %
%   #voisin = 2 -> point simple                               %
%   #voisin > 2 -> point de bifurcation                       %
%=============================================================%
```

```
65
66  point_bifurcation(P) :-
67        nombre_voisin(P,N),
68        ge(N,3).
69
70  point_extremite(P) :-
71        nombre_voisin(P,1).
72
73  point_simple(P) :-
74        nombre_voisin(P,2).
75
76  point_singulier(P) :-
77        point_simple(P);
78        point_bifurcation(P).
79
80  %=============================================================%
81  %     INFORMATIONS DE NIVEAU 1D :                             %
82  %           ligne_degeneree/1                                 %
83  %           contour/1                                         %
84  %           boucle/1                                          %
85  %=============================================================%
86
87  %==============================
88  % creation des lignes degenerees -> suppression de la relation de
89  % segment qui les lient aux autres points
90  %==============================
91
92  % regle de construction des lignes degenerees :
93  construct_ligne_degeneree(L0,Res) :-
94        conc(L0,_,[Pext]),
95        voisin(Pext,Pnext),
96        ( point_singulier(Pnext) -> conc(Res,L0,[Pnext]);
97                                    conc(L1,L0,[Pnext]),
98                                    construct_ligne_degeneree(L1,Res)).
99
100 % creation des lignes degenerees et suppression des segments
101 create_ligne_degeneree(E) :-
102       point_extremite(P),
103       construct_ligne_degeneree([P],Ligne),
104       conc(Ligne,_,[Psimple,Pbifurcation]),
105       ( segment(Psimple,Pbifurcation) ->
106             retract( (segment(Psimple, Pbifurcation):-Q) );
107             retract( (segment(Pbifurcation, Psimple):-Q) )),
108       assertz(ligne_degeneree(Ligne)).
109
110 %============================
111 % creation des contours
112 %============================
113
114 exist_contour :- contour(L),!.
115
116 % construction des contours a partir des points de bifurcation
117 create_contour(E) :-
118       point_bifurcation(P0),
119       voisin(P0,P1),
110       ( point_extremite(P1) ->
121             fail;
122             ( construct_contour([P1,P0],[P1|Reste]),
123               ( exist_contour -> findall(X,contour([P|X]),Liste),
124                                  ( notSemblableInList(Reste,Liste) ->
                                     assertz(contour([P1|Reste])) );
```

```
                                        assertz(contour([P1|Reste])))))).

% regle auxiliaire qui assure que les contours ne passe pas
% deux fois par un meme segment en sens oppose
regle_aux(L1,P,Next) :-
      conc(L0,_,[P]),
      inlist(Next,L0),
      inlist(P,L0),
      index(Next,L0,I),
      index(P,L0,J),
      I \== J+1,
      J \== I+1.

% regle de creation des contours a partir d'une liste germe L0
construct_contour(L0,Res) :-
      L0 = [End|_],
      conc(L0,_,[P1,P]),
      choice_next_point(P,P1,Next),
      regle_aux(L0,P,Next),
      ( Next==End -> conc(Res,L0,[Next]);
                     conc(L1,L0,[Next]),
                     construct_contour(L1,Res)).

% renvoie les voisins de P sauf Exception
voisin_sauf(P,Exception,Voisin) :-
      voisin(P,Voisin),
      Voisin \== Exception.

% P2 est le point suivant le segment [P1,P]
choice_next_point(P,P1,P2) :-
      ( point_extremite(P) -> fail;
                              ( point_simple(P) -> voisin(P,P2),
                                                   P2 \== P1;

findall(X,voisin_sauf(P,P1,X),Liste),

choice_in_list(P,P1,P2,Liste,Angle))).

% initialisation de cette regle
choice_in_list(P,P1,P2,[P2],Angle) :-
      angle(P,P1,P2,Angle).

% cette regle selectionne parmi les points de la Liste celui qui
% doit succeder a [P1,P] : [P1,P,P2]
choice_in_list(P,P1,P2,Liste,Angle) :-
      Liste = [Pretendant|Queue],
      size(N,Liste), ge(N,2),
      choice_in_list(P,P1,Challenger,Queue,Val1),
      angle(P,P1,Pretendant,Val2),
      choix(Val1,Val2,Angle),
      ( Angle == Val1 -> P2 = Challenger;
                         P2 = Pretendant).


% choix entre deux valeur : si(neg) alors max(neg) sinon min(pos)
choix(Val1,Val2,Res) :-
      (Val1 > 0 -> (Val1>Val2 -> Res = Val2 ; Res = Val1),!);
      (Val2 > 0 -> (Val1>Val2 -> Res = Val2 ; Res = Val1),!);
      (Val1>Val2 -> Res = Val1 ; Res = Val2).

%============================
```

```
% creation des boucles
%==========================

% enumere tous les points faisant parmi d'au moins un contour
enum_points_contour(X) :-
      contour([P|L]),
      size(N,L),
      enum(I,1,N),
      index(X,L,I).

exist_boucle :- boucle(_),!.

% boucle : contour compose uniquement de points simples
boucle(L0,Res) :-
      conc(L0,[End],_),
      conc(L0,_,[P2,P1]),
      voisin(P1,Next),
      Next \== P2,
      ( Next == End -> conc(Res,L0,[Next]);
                  conc(L1,L0,[Next]),
                  boucle(L1,Res)).

% creation des boucleS
create_boucle :-
      point_simple(P0),
      setof(X,enum_points_contour(X),Liste),
      outlist(P0,Liste),
      voisin(P0,P1),
      voisin(P0,P2),
      P1 \== P2,
      angle(P0,P1,P2,A1),
      ( A1 > 0 -> L0 = [P1,P0,P2,P1] ;
                L0 = [P2,P0,P1,P2]),
      ( voisin(P1,P2) -> Res = L0;
                     boucle(L0,Res)),
      conc(Res,[Debut],Reste),
      ( exist_boucle -> findall(X,boucle([P|X]),L),
                    ( notSemblableInList(Reste,L) ->
                       assertz(boucle(Res)));
                       assertz(boucle(Res))).

%=================================================================%
% calcul trigo sur des points faisant appel a du code Java       %
% necessite la declaration prealable des predicats Java          %
%=================================================================%

% fait appel a la methode Java : Trigo.angle()
angle(P1,P2,P3,A) :-
      point(P1,X1,Y1),
      point(P2,X2,Y2),
      point(P3,X3,Y3),
      angle(X1,Y1,X2,Y2,X3,Y3,A).


affiche_geo :-
      (ligne_degeneree(L),write(L),nl);
      (contour(L),write(L),nl);
      (boucle(L),write(L),nl).
```

Nous avons testé ce programme avec la figure suivante :

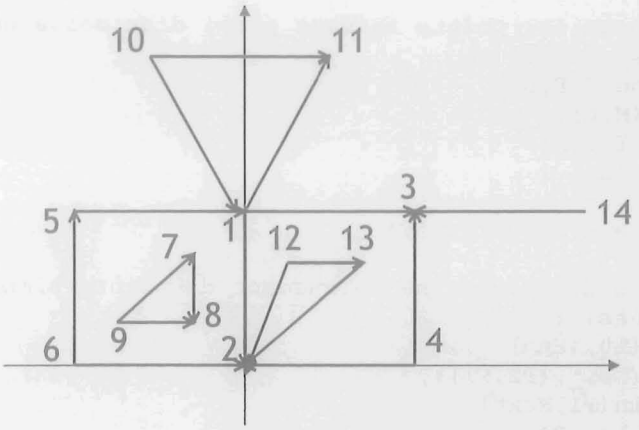


Figure 2 : Cas test

On obtient une ligne dégénérée ([3,14]) et cinq contours : [1,2,12,13,2,4,3,1], [1,2,6,5,1], [1,11,10,1], [2,13,12,2] et [7,9,8].

Nous avons alors souhaité réutiliser ce programme Prolog pour résoudre un problème concret. Dans le cas d'une problème de chauffe d'un matériau, il est nécessaire de recalculer à chaque pas de temps le front de solidification-fusion à partir du champ de température obtenu sur un maillage. Il s'agit en fait de recomposer la géométrie.

A partir du maillage valué, une méthode calcule pour chaque maille les segments de bord et d'isovaleur (Figure 3). Par la suite, le programme Prolog se charge de recomposer la géométrie pour l'itération suivante.

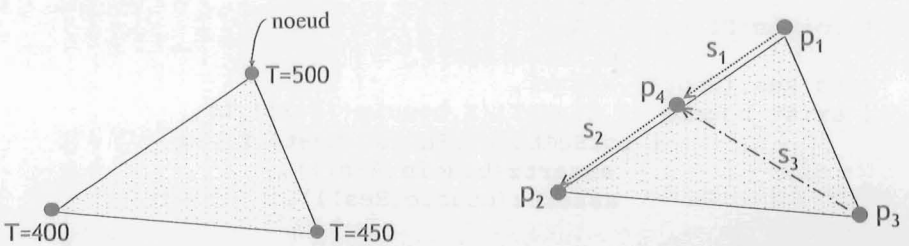


Figure 3 : Exemple de points et segments générés sur une maille

Nous avons testé notre programme sur une géométrie très simple, un carré de côté unité, avec deux maillages différents : un maillage 9*9 et un maillage 13*13. Dans tous les cas, la géométrie a bien été reconstruite.

Cependant les temps d'exécution[1] du programme Prolog varient énormément (Tableau 1). Pour le maillage 9*9, et dans le cas où il n'y a pas d'isovaleur, la base de faits est constituée des 36 segments formant le bord du carré. Le programme Prolog met alors 25 secondes a recalculer la géométrie.

Sur le maillage 13*13, et dans les mêmes conditions, la base de faits contient 52 segments. Le temps d'exécution passe à 88 secondes. Lorsqu'il y a présence d'une isovaleur, les segments la composant s'ajoutent aux segments définissant le bord de la géométrie. La base de faits passe alors à 81 occurrences de segments, et le temps à 870 secondes, soit presque un quart d'heure.

[1] Programme exécuté sur un processeur 500 Mhz avec 256Mo de mémoire vive.

Nombre de segments :	Coût (secondes) :
36	25
46	56
51	81
52	88
73	462
81	870

⇒

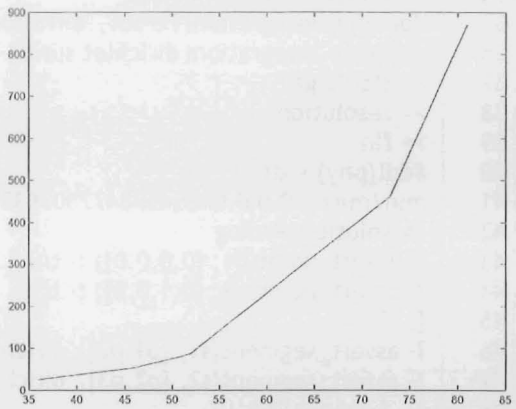


Tableau 1 : Coût d'exécution du programme Prolog

La programmation logique doit donc être réservée à des problématiques d'expertise. En effet, son coût combinatoire est difficilement compatible avec des problèmes de nature algorithmiques, caractérisés par des bases de faits rapidement conséquente.

3 Déroulement d'une simulation

```
00
01  Prolog IV v1.3.1 (161), Octobre 2000 (C)PrologIA 1995..2000
02  [ Tue Nov 14 15:11:48 MET 2000 ]
03
04  ?- compile("/mnt/zip/PROLOG/GEOM/iso.p4") :- true.
05  ?- iso :- true.
06  ?- compile("/mnt/zip/PROLOG/GEOM/topo2D.p4") :- true.
07  ?- dynamic(point/3) :- true.
08  ?- dynamic(segment/5) :- true.
09  ?- dynamic(ligne/3) :- true.
10  ?- dynamic(contour/3) :- true.
11  ?- dynamic(face/3) :- true.
12
13  >> Iter n°1 t=0.1 <<
14  >> maillage
15  >> integration/assemblage
16  domaine.integration('Fe Sol', 'diffusion dynamique')
17  domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
18  >> stockage
19  >> resolution
20  >> fin
21  #ddl(phy) = 81
22  min/max = 0.0 0.030106426611127002
23  >> solution2Prolog
24  ?- assert_point(p1, [0.0,0.0]) :- true.
25  ?- assert_point(p2, [0.1,0.0]) :- true.
26  [...]
27  ?- assert_segment(s1, [p1,p2], 'dirichlet', 'solide', 'null') :- true.
28  ?- assert_segment(s2, [p2,p3], 'dirichlet', 'solide', 'null') :- true.
29  [...]
30  >> finReconstruction
31
32  >> Iter n°2 t=0.2 <<
33  >> maillage
```

```
>> integration/assemblage
domaine.integration('Fe Sol', 'diffusion dynamique')
domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
>> stockage
>> resolution
>> fin
#ddl(phy) = 81
min/max = 0.0 0.048545784775036335
>> solution2Prolog
?- assert_point(p1, [0.0,0.0]) :- true.
?- assert_point(p2, [0.1,0.0]) :- true.
[...]
?- assert_segment(s1, [p1,p2], 'dirichlet', 'solide', 'null') :- true.
?- assert_segment(s2, [p2,p3], 'dirichlet', 'solide', 'null') :- true.
[...]
>> finReconstruction

>> Iter n°3 t=0.3 <<
>> maillage
>> integration/assemblage
domaine.integration('Fe Sol', 'diffusion dynamique')
domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
>> stockage
>> resolution
>> fin
#ddl(phy) = 81
min/max = 0.0 0.06837939232186857
>> solution2Prolog
?- assert_point(p1, [0.0,0.0]) :- true.
?- assert_point(p2, [0.1,0.0]) :- true.
[...]
?- assert_segment(s1, [p1,p2], 'dirichlet', 'solide', 'null') :- true.
?- assert_segment(s2, [p2,p3], 'dirichlet', 'solide', 'null') :- true.
[...]
>> finReconstruction

>> Iter n°4 t=0.4 <<
>> maillage
>> integration/assemblage
domaine.integration('Fe Sol', 'diffusion dynamique')
domaine.integration('Fe Liq', 'diffusion dynamique')
domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
domaine.integration('interface', 'interface')
>> stockage
>> resolution
>> fin
#ddl(phy) = 95
min/max = 0.0 0.08925225765887307
>> solution2Prolog
?- assert_point(p1, [0.0,0.0]) :- true.
?- assert_point(p2, [0.1,0.0]) :- true.
[...]
?- assert_segment(s1, [p1,p2], 'dirichlet', 'diffusion dynamique', 'null') :- true.
[...]
?- assert_segment(s38, [p26,p55], 'interface', 'solide', 'liquide') :- true.
[...]
>> finReconstruction

>> Iter n°5 t=0.5 <<
>> maillage
```

```
94  >> integration/assemblage
95  domaine.integration('Fe Sol', 'diffusion dynamique')
96  domaine.integration('Fe Liq', 'diffusion dynamique')
97  domaine.integration('dirichlet sur b1 = 0', 'dirichlet')
98  domaine.integration('interface', 'interface')
99  >> stockage
100 >> resolution
101 >> fin
102 #ddl(phy) = 102
103 min/max = 0.0 0.12027645882206645
104
```

4 Traitement numérique du problème thermique transitoire

Le problème thermique en régime transitoire s'écrit sous la forme mathématique suivante :

$$\frac{\partial T}{\partial t} - k\,\mathrm{div}(\mathrm{grad}\,T) = q \text{ sur } \Omega_{x,y} \times \Omega_t \quad (1)$$

$$T(x,y,0) = T_0(x,y) \text{ sur } \Omega_{x,y} \quad (2)$$

$$T = f \text{ sur } \Gamma_{dir} \times \Omega_t \quad (3)$$

$$\mathrm{grad}\,T \cdot \vec{n} = g \text{ sur } \Gamma_{neu} \times \Omega_t \quad (4)$$

$$[k\,\mathrm{grad}\,T \cdot \vec{n}] = 0 \text{ sur } \Gamma_{1\text{-}2} \times \Omega_t \quad (5)$$

Sur ce modèle, nous avons fait l'hypothèse que la conductivité k est constante et isotrope dans un matériau donné (k(x,y,t) = k). Pour discrétiser ce problème, nous avons choisi de faire des différences finis implicites en temps, et des éléments finis nodaux du premier ordre en espace. Le schéma différences finies temporel implicite s'écrit :

$$\frac{T^{n+1} - T^n}{\Delta t} - k\,\mathrm{div}(\mathrm{grad}\,T^n) = q^{n+1}$$

qui est équivalent à :

$$T^{n+1} - \Delta t.k.\,\mathrm{div}(\mathrm{grad}\,T^n) = \Delta t.q^{n+1} + T^n \quad (6)$$

où $T^{n+1}=T(x,y,t_{n+1})$ et Δt est le pas de temps.

La forme faible de l'équation (6) est obtenue en la multipliant par une fonction de forme $\varphi(x,y)$ et en intégrant par parties :

$$\iint_{\Omega_{x,y}} \left[T^{n+1} - \Delta t.k.\,\mathrm{div}(\mathrm{grad}\,T^n)\right]\varphi \; dxdy = \iint_{\Omega_{x,y}} \left[\Delta t.q^{n+1} + T^n\right]\varphi \, dxdy$$

$$\Leftrightarrow \iint_{\Omega_{x,y}} \left[T^{n+1}.\varphi\right] dxdy - \Delta t.k \int_{\partial\Omega_{x,y}} \left[(\mathrm{grad}\,T^{n+1}.\vec{n}).\varphi\right] dxdy + \Delta t.k \iint_{\Omega_{x,y}} \left[(\mathrm{grad}\,T^{n+1}).(\mathrm{grad}\,\varphi)\right] dxdy =$$

$$\iint_{\Omega_{x,y}} \left[\Delta t.q^{n+1} + T^n\right]\varphi \, dxdy$$