Réalisation d'une plate-forme informatique dédiée aumétier du génie électrique autour des logiciels FLUX.Application à la réalisation de logiciels métiers.

Yves Souchard

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THESE

Pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Génie Electrique

Préparée au **Laboratoire d'Electrotechnique de Grenoble**
UMR 5529
Dans le cadre de l'Ecole Doctorale **E**lectronique, **E**lectrotechnique, **A**utomatique, **T**élécommunication, **S**ignal

Présentée et soutenue publiquement
par

Yves SOUCHARD

Le 18 Novembre 2005

Titre :

Réalisation d'une plate-forme informatique dédiée au métier du génie électrique autour des logiciels FLUX. Application à la réalisation de logiciels métiers.

Directeur de thèse : Yves Maréchal

JURY

M. PHILIPPE MASSE	Président
M. PATRICK DULAR	Rapporteur
M. YVES LE TRAON	Rapporteur
M. YVES MARECHAL	Directeur de thèse
M. GUY JEROME	Examinateur
M. FREDERIC NOEL	Examinateur

Remerciements

Ce travail de thèse a été réalisé au Laboratoire d'Electrotechnique de Grenoble d'octobre 2002 à octobre 2005, en convention CIFRE avec la société Cedrat.

Je tiens tout d'abord à remercier Philippe Masse, président du jury, ainsi que Patrick Dular, Yves Le Traon et Frédéric Noël. Je leur exprime ma gratitude pour avoir accepté de faire partie du jury, et pour l'intérêt qu'ils ont porté à mes travaux de recherche. Je remercie également mon directeur de thèse, Yves Maréchal, sans qui ce travail n'aurait pu être réalisé. Je tiens tout particulièrement à le remercier pour la pertinence de ses réflexions scientifiques, son encadrement remarquable ainsi que pour l'expérience qu'il a bien su me transmettre. Je remercie aussi mon co-encadrant, Patrick Eustache, pour sa bonne humeur et pour ses conseils avisés.

Au sein de la société Cedrat, je remercie tout d'abord Xavier Brunotte pour son implication dans le projet de partenariat avec le Laboratoire d'Electrotechnique de Grenoble ainsi que pour son soutient tout au long de cette thèse. J'adresse également ma plus profonde reconnaissance à Guy Jérome. Son dynamisme et sa bonne humeur, ainsi que ses compétences techniques ont permis d'assurer la qualité de son encadrement. Je tiens aussi à remercier tous les employés de la société Cedrat, pour leur accueil, leur écoute, leur aide ainsi que leurs conseils éclairés.

Durant les trois années passées au Laboratoire d'Electrotechniques de Grenoble, j'ai eu la chance d'être entouré par des personnes formidables. Je tiens à remercier toutes ces femmes et tous ces hommes qui font la vie du laboratoire. En particulier, j'adresse mes remerciements à tous les doctorants de toutes les équipes confondues qui égayent le quotidien du laboratoire. Je tiens également à remercier l'équipe du service informatique, pour leur inébranlable moral et pour le fonctionnement exemplaire des installations informatiques.

Je remercie également mes amis, d'ici et d'ailleurs. Plus particulièrement, je remercie sincèrement tous les fous volants de dégaines en dégaines ou de cols en cols. Je vous dis à très bientôt dans nos montagnes.

Je tiens à remercier vivement mes parents et ma famille pour leur éducation et leur soutient qu'ils m'apportent depuis toujours. Mes derniers remerciements sont pour ma femme et ma fille qui égayent ma vie un peu plus chaque les jours.

Table des matières

Introduction générale 9

1 Problématique des logiciels métiers 13

1.1 Introduction 13

1.2 Problématique 13
1.2.1 Logiciels métiers et objets métiers 14
1.2.2 Intérêts des logiciels métiers 15

1.3 Cahier des charges 15
1.3.1 Objectifs et livrables 16
1.3.2 Contraintes 16
1.3.3 Plan de développement 17

1.4 Etat de l'art 17
1.4.1 Approche pour la conception de logiciel métier 17
1.4.1.1 Méthode de conception par assemblage de composants 18
1.4.1.2 Méthode de conception par modélisation du logiciel 18
1.4.1.3 Analyse critique des méthodes de conception 18
1.4.2 Méta modélisation des logiciels métiers 19
1.4.3 UML, MOF, XMI et autre MDA 20
1.4.4 Adaptive Object Model 21
1.4.5 Analyse critique de la méta modélisation 21

1.5 Choix 22

1.6 Conclusion 24

2 Conception de logiciels adaptables 25

2.1 Introduction 25

2.2 Modèle statique d'un logiciel 26
2.2.1 Les modèles de données 26
2.2.2 Les paquetages 27
2.2.3 Les structures de données 28
2.2.3.1 Les types intrinsèques 29
2.2.3.2 Les énumérations 30
2.2.3.3 Les interfaces 30
2.2.3.4 Les entités 31
2.2.3.5 Les agrégats 32
2.2.3.6 Les champs 33
2.2.4 Les commandes 35
2.2.4.1 Les procédures 36
2.2.4.2 Les méthodes 37

2.2.4.3 Les méthodes intrinsèques 38
2.2.4.4 Les arguments 39
2.2.5 Les applications 40
2.2.6 Les cas d'utilisation 40

2.3 Modélisation de la présentation 41
2.3.1 Représentation graphique des données et des commandes 42
2.3.1.1 Les informations communes 42
2.3.1.2 Les catégories 43
2.3.1.3 Les entités 43
2.3.1.4 Les champs 44
2.3.1.5 Les commandes 45
2.3.1.6 Les arguments 46
2.3.2 L'interface homme machine globale 46
2.3.2.1 Les applications 47
2.3.2.2 Les cas d'utilisation 48
2.3.2.3 Les contextes 48
2.3.2.4 Les profils utilisateurs 50
2.3.2.5 Les menus 50
2.3.2.6 Les barres d'icônes 52
2.3.2.7 Les arbres de représentation 52
2.3.2.8 Les fenêtres graphiques 53
2.3.2.9 Les entités graphiques 54
2.3.3 Contextualisation 55

2.4 Modèle dynamique d'un logiciel 58

2.5 Génération du code 60

2.6 Exécution du modèle 61
2.6.1 Structure de la machine virtuelle 62
2.6.1.1 Le module Frontal 63
2.6.1.2 Le module Kernel 64
2.6.1.3 Le module Interface Utilisateur 65
2.6.1.4 Le module Chargeur de code applicatif 66
2.6.1.5 Le module Code applicatif 66
2.6.2 Démarche d'exécution des logiciels métiers 67
2.6.3 Héritage et surcharge 68

2.7 Conclusion 69

3 Conception d'un descripteur de logiciel 71

3.1 Introduction 71

3.2 Le logiciel FluxBuilder 72
3.2.1 Boîte de dialogue 75
3.2.2 Outils de synthèse 78
3.2.2.1 Management des informations de type interface utilisateur 80
3.2.2.2 Gestion et évaluation des masques 81

3.3 FluxFactory 84

3.4 Auto conception de FluxBuilder 85

3.5 Conclusion 86

4 Réalisations de logiciels métiers 89

4.1 Introduction 89

4.2 TransfoBuilder 90
4.2.1 Création du modèle de données 92
4.2.2 Le dialogue utilisateur 94
4.2.3 Algorithmes spécifiques 95
4.2.3.1 Création automatique de géométrie 95
4.2.3.2 Affectation automatique des paramètres 96
4.2.4 Evolutions et perspectives 97

4.3 Inductance Calculation [InCa] 98

4.4 FluxMotor 99
4.4.1 Démarche de conception d'objet métier 101
4.4.1.1 Patron d'objet métier 101
4.4.1.2 Template Editor 102
4.4.2 Intégration dans le logiciel généraliste 103
4.4.3 Algorithmes spécifiques 103
4.4.4 Evolutions et perspectives 104

4.5 Les logiciels Flux 105
4.5.1 Modèles des logiciels Flux 106
4.5.2 Evolutions et perspectives 106

4.6 Conclusion 107

Conclusion générale 109

Perspectives 111

Liste des publications 113

Bibliographie 115

Glossaire 119

Introduction générale

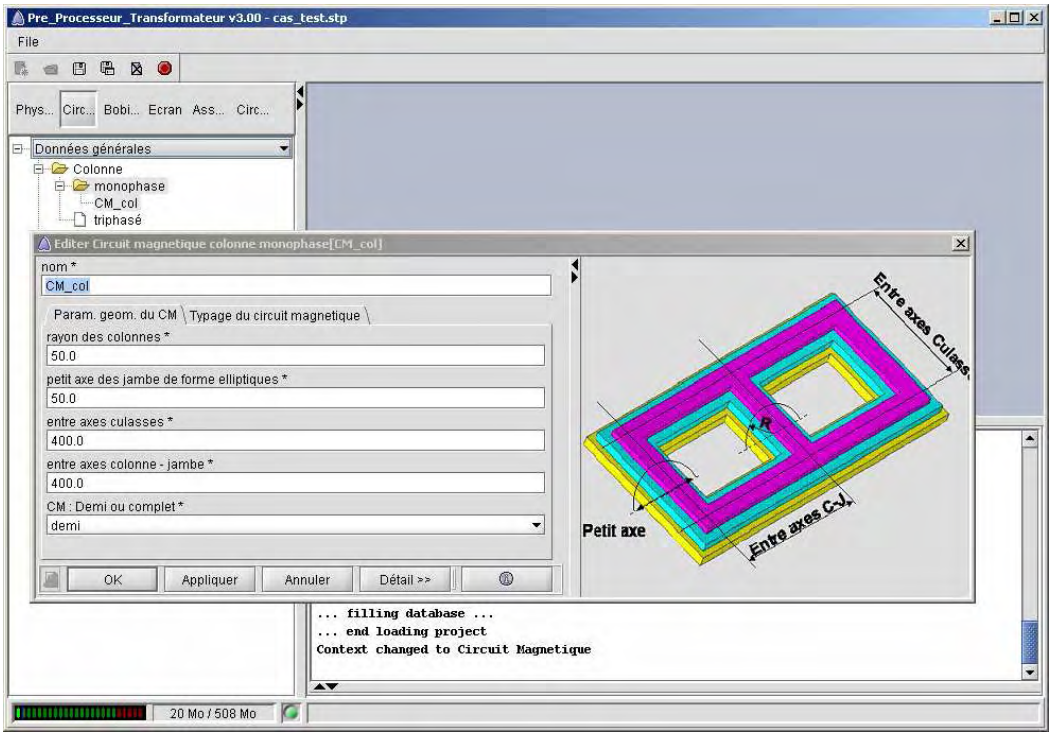
Les logiciels Flux sont des logiciels industriels de modélisation des phénomènes électromagnétiques en deux et trois dimensions par la méthode des éléments finis. Leurs domaines d'application sont la magnétostatique, la magnétodynamique en régime harmonique ou transitoire, l'électrostatique et la thermique. Ils sont destinés à l'industrie du génie électrique pour la modélisation et la conception de dispositifs variés tels que les moteurs électriques et les génératrices, les contacteurs et les actionneurs, les transformateurs, le chauffage par induction ou le contrôle non destructif. Leur large éventail d'utilisation permet d'étudier aussi bien une tête de lecture que la signature magnétique d'un porte-avions. Ils sont utilisés de par le monde (Europe, Etats-Unis et Asie du sud-est) par un millier de sociétés et d'universités.

Les logiciels Flux sont co-développés par la société Cedrat et par le Laboratoire d'Electrotechnique de Grenoble et commercialisés par le groupe Cedrat. L'ensemble Cedrat - Laboratoire d'Electrotechnique de Grenoble, dont le partenariat est exemplaire des relations entre industrie et université, est un acteur mondial majeur de la modélisation en électrotechnique aussi bien d'un point de vue industriel que recherche.

Aujourd'hui, les logiciels Flux sont utilisés essentiellement dans des centres de recherche par des experts. Une tendance lourde est l'utilisation d'outils de simulation et de conception dans les bureaux d'études à condition que ces outils soient d'un abord plus simple et répondent à un métier précis. C'est dans le but de pouvoir décliner les logiciels généralistes Flux en une multitude de logiciels métiers que cette thèse a été entreprise. L'objectif des travaux exposés dans ce manuscrit est le développement d'une plate-forme informatique dédiée au métier du génie électrique autour des logiciels Flux.

Depuis l'année 2000, la société Cedrat a entrepris une refonte complète de ses logiciels de simulation. Dans ce but, de nombreuses études et travaux de recherche ont été menés au sein du Laboratoire d'Electrotechnique de Grenoble [MARECHAL-01]. En 2003 la décision a été prise d'utiliser ces travaux afin d'améliorer le processus de développement de tous les logiciels de simulation de la société Cedrat. Mon travail de thèse, débuté en 2002, s'inscrit dans ce cadre, ma mission étant de fournir des outils et des méthodes pour faciliter cette refonte. Autour de mes travaux se sont cristallisés une grande partie des développements aussi bien pour Cedrat que pour le Laboratoire d'Electrotechnique de Grenoble.

Les logiciels métiers ont pour but de s'adapter au mieux aux métiers ciblés. Pour cela, ces logiciels doivent être dotés de commandes et d'un vocabulaire adapté à ce métier. La figure suivante montre un exemple de logiciel métier dédié à la conception de transformateur. Il comprend un ensemble de commandes permettant une description facile et conviviale.



Exemple de logiciel métier

Afin de présenter ces trois années de travail et leurs résultats, nous avons adopté le plan suivant.

Dans le premier chapitre d'introduction, nous proposons une analyse de la notion de logiciel métier et nous exposons les différentes techniques envisageables pour les concevoir et les réaliser. De cette recherche bibliographique, nous retenons une approche basée sur une modélisation des applications métiers et nous présentons plus précisément les techniques de modélisation et la notion de méta modélisation.

Afin de comprendre comment fonctionne la conception de logiciels métiers, ce travail de thèse s'appuie sur une seconde partie consacrée tout d'abord à la modélisation de ces applications. Ces modèles permettent de décrire le logiciel souhaité tant d'un point de vue statique que dynamique. La partie statique modélise les différentes données et commandes de l'application mais aussi son interface homme machine. La partie dynamique modélise les enchaînements d'actions dans le domaine temporel.

La suite de ce chapitre met l'accent sur la réalisation d'une "machine virtuelle" permettant l'interprétation de ces modèles. Nous expliquons dans ce chapitre comment cette machine virtuelle est structurée et comment elle fonctionne.

A l'issue de ces deux parties, décrivant la création et le fonctionnement de logiciels métiers, nous nous sommes investis dans le développement d'un descripteur de logiciel permettant une utilisation plus facile qu'un simple traitement de texte. Ainsi, ce logiciel appelé FluxBuilder permet de faciliter grandement la saisie des objets du modèle de données d'un logiciel grâce à une interface conviviale et intuitive. La description de FluxBuilder fait l'objet du troisième chapitre.

Tous les outils nécessaires à notre démarche de conception étant développés, nous sommes à même de réaliser des logiciels spécifiques pour des métiers tels que le dimensionnement de transformateurs, le calcul d'impédance dans des conducteurs ou la conception de moteurs. Le dernier chapitre de ce manuscrit expose les résultats obtenus par l'utilisation de notre démarche. Ainsi, divers logiciels scientifiques ont été réalisés dans des domaines variés tel que la conception de transformateur ou l'analyse et la détermination des inductances parasites.

Enfin, le dernier chapitre est consacré aux conclusions générales de ce travail de recherche et aux perspectives qu'il ouvre.

1 Problématique des logiciels métiers

1.1 Introduction

Les logiciels de calcul utilisés dans les départements de recherche et développement sont souvent des logiciels généralistes ouverts sur un grand nombre de domaines d'application. Ce type de logiciel est nécessaire pour mener des études diversifiées ou innovantes. Le logiciel Flux, co-développé dans le partenariat entre le Laboratoire d'Electrotechnique de Grenoble et la société Cedrat, fait partie de cette famille de logiciels. Cependant, son utilisation pour des travaux répétitifs ou s'appliquant toujours à un même domaine peut s'avérer moins adaptée, entraînant des pertes de temps et donc de compétitivité. A contrario, les logiciels métiers spécialisés par domaine d'application ou par métier permettent d'aller plus rapidement à l'essentiel en répondant de manière simple et directe aux besoins ciblés des utilisateurs. Le vocabulaire utilisé permet une meilleure compréhension et donc une meilleure utilisation de ce type d'application. Les commandes proposées par ces logiciels métiers correspondent aux fonctions attendues par les spécialistes du métier.

Le but de cette thèse est de mettre en place une plate-forme dédiée à la création de logiciels métiers dérivés du logiciel généraliste Flux. Cette plate-forme devra permettre de créer, de modifier et de maintenir ces logiciels dérivés à un coût de développement très faible, en s'appuyant sur les capacités de calcul de Flux.

Ce chapitre va nous permettre de comprendre les notions nécessaires à la réalisation de cette thèse, tels que les logiciels et objets métiers, mais aussi les méthodes de conception de logiciels métiers, ainsi que les différents concepts de modélisation de logiciel disponibles à l'heure actuelle. Après avoir présenté et analysé ces aspects bibliographiques, nous exposerons nos choix ainsi que notre plan de travail.

1.2 Problématique

Tout d'abord, avant d'expliquer ce qu'est un logiciel métier, il nous faut définir précisément la notion de métier. Un métier, du point de vue du génie logiciel, est avant tout un ensemble de concepts et d'actions, fournissant un vocabulaire et une grammaire spécifiques. Les concepts correspondent aux différentes notions manipulées par le métier. Par exemple, pour le métier de concepteur de micro inductance, le principal concept est celui de

micro inductance définie par le rayon de la première spire, le nombre de spires, l'espacement entre spires ou le type de section (circulaire ou rectangulaire).

Les actions d'un métier, quant à elles, sont les commandes spécifiques que les spécialistes du métier doivent pouvoir réaliser. Pour notre exemple, ces commandes sont les algorithmes spécifiques de construction de la géométrie d'une micro inductance, son maillage en fonction de ses paramètres géométriques, l'affectation de ses paramètres physiques …

L'ensemble concepts et actions doit répondre à des règles, issues du métier ou de la physique, qui constitue la grammaire du domaine.

Les logiciels métiers proposent des outils de calculs adaptés à un métier. Les concepts, algorithmes et méthodes spécifiques sont encapsulés dans une interface homme machine adaptée. Chaque logiciel métier comporte ses propres menus, barres d'outils, arbres de représentation.

1.2.1 Logiciels métiers et objets métiers

Nous sommes aussi amenés à distinguer les logiciels métiers et les objets métiers. Un objet métier est un concept isolé, disposant de comportements spécifiques mais étant intégré dans un logiciel généraliste. Un logiciel métier manipule généralement un ensemble de concepts métiers. Par ailleurs, sa présentation peut être totalement différente de celle d'un logiciel généraliste même s'il en dérive.

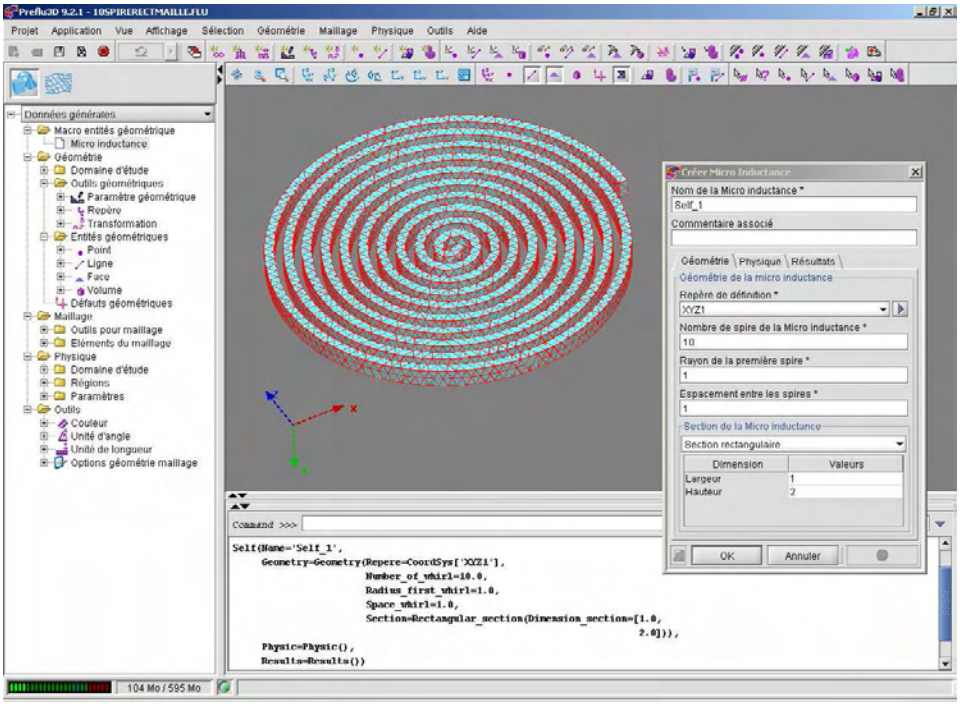


Figure 1.1 : Exemple d'objet métier : la micro inductance

L'exemple présenté ci-avant permet d'illustrer la définition d'objet métier. Dans le logiciel Flux, nous avons ajouté l'objet métier "micro-inductance". Le but est de simplifier les phases de création d'une micro-inductance en vue de sa résolution ultérieure par Flux. Dans cette optique, l'objet métier comprend des algorithmes de construction géométrique et d'affectation des paramètres physiques, le tout entièrement automatisé. De plus, une boîte de dialogue dédiée permet de saisir efficacement les paramètres des micro-inductances.

1.2.2 Intérêts des logiciels métiers

Le principal intérêt des logiciels métiers réside dans l'économie de temps qu'ils procurent. En effet, grâce à des algorithmes spécifiques au métier, les utilisateurs ont beaucoup moins de manipulations à effectuer pour obtenir les résultats souhaités. Ce gain de temps apparaît comme un attrait économique important, poussant les entreprises à investir dans le développement de tels logiciels.

Le deuxième intérêt est un gain en terme de communication et de convivialité. L'interface homme machine spécialisée, associée à un vocabulaire dédié, permet une meilleure prise en main et une utilisation plus sûre de ce logiciel dédié à leur métier. Dans un monde où les logiciels deviennent de plus en plus performants, cette meilleure compréhension de l'outil peut s'avérer un atout majeur dans le domaine commercial.

Néanmoins, la réalisation de logiciels métiers a un coût, qu'il ne faut pas négliger même lorsqu'on s'appuie sur un code généraliste comme boîte à outils de calcul. Dans cette situation, le principal écueil réside dans la maintenance : si la construction d'applications métiers peut se faire rapidement, la mise à jour et le suivi de nombreuses applications métiers peuvent s'avérer catastrophiques économiquement. Il est donc nécessaire de mener une réflexion sur la meilleure manière de concevoir ces logiciels métiers, en minimisant leurs coûts de développement et de maintenance. Dans ce but, seule une étude approfondie des différentes méthodes de conception de logiciel pourra nous permettre de répondre au mieux à notre cahier des charges.

1.3 Cahier des charges

Afin de répondre au mieux à la problématique de cette thèse, nous avons élaboré une sorte de "cahier des charges" permettant de hiérarchiser les objectifs à atteindre. Nous avons aussi listé les différentes contraintes que nous devions respecter dans notre démarche. Tout cela nous a permis de définir un plan d'action pour mener à bien nos recherches.

1.3.1 Objectifs et livrables

Le premier but de ce travail est de rendre possible la personnalisation d'une application. Cela consiste à pouvoir modifier certains aspects d'un logiciel pour mieux prendre en compte la manière de fonctionner de l'utilisateur, comme par exemple les informations affichées dans les boîtes de dialogue ou les composants de l'interface homme machine.

Dans un deuxième temps, l'objectif est de pouvoir créer, modifier et gérer facilement des objets métiers au sein du logiciel Flux. Afin de réduire leurs coûts de création et de maintenance, une démarche de conception adaptée à ce type d'objet doit être mise en place.

Ensuite, nous souhaitons permettre la conception d'applications métiers basées sur les logiciels généralistes Flux (algorithmes et interface homme machine) et associées à des objets métiers. La création de ces logiciels dérivés est réalisée en intégrant un ou plusieurs objets métiers au sein d'une application Flux afin d'obtenir un logiciel spécifique à un métier donné. Ce premier pas vers les logiciels métiers nous permettra de tester l'utilisation des objets métiers ainsi que de valider la démarche de conception de ceux-ci.

L'objectif final de ces travaux est la mise en place d'une démarche de conception pour la réalisation de logiciels métiers s'appuyant sur les algorithmes de calcul des logiciels Flux, tout en masquant totalement l'application généraliste. Cela passe par la réalisation d'une plate-forme de création et de maintenance permettant la description complète de logiciels métiers. Nous pourrons alors présenter aux utilisateurs finaux des applications totalement dédiées à leur métier, depuis les concepts jusqu'à la présentation en passant par les comportements. De ce fait, la prise en main de nos logiciels métiers sera beaucoup plus intuitive pour les spécialistes du métier ciblé, et son utilisation sera grandement facilitée.

1.3.2 Contraintes

Après avoir défini les différents objectifs de mes travaux de thèse, il faut établir toutes les contraintes à respecter lors de la réalisation de ces travaux.

Tout d'abord, les perspectives économiques actuelles nous oblige à réduire toujours plus les temps de conception de nos logiciels. C'est pourquoi, la première contrainte que nous devons respecter est la réduction à l'extrême des coûts de développement et de maintenance de nos futurs logiciels métiers.

La deuxième contrainte est directement liée à la complexité des métiers pour lesquels nous souhaitons réaliser des logiciels dédiés. Notre but étant de créer des applications correspondant très précisément aux métiers ciblés, nous devons pouvoir spécialiser nos

logiciels en intégrant tout le savoir de l'expert du métier. Cela implique l'utilisation d'un vocabulaire très spécifique au métier ainsi qu'une conception de nos logiciels pensée et orientée entièrement autour du métier choisi. Pour cela, il est obligatoire de collaborer avec les spécialistes des métiers sélectionnés. La démarche de conception des applications métiers doit intégrer ces experts non informaticiens et proposer des outils qu'ils peuvent prendre en main.

Enfin, il est indispensable de tenir compte des réalisations déjà existantes : tout développement devra se faire en accord avec les bases déjà instaurées pour les logiciels Flux. En effet, mes travaux de thèse seront intégrés dans le développement global de la société Cedrat. Cela implique l'obligation du respect d'un certain nombre de règles et de conventions établies depuis de nombreuses années. Tout ceci entraîne un apprentissage complet des méthodes de développement déjà mises en place dans le cadre de la réalisation des logiciels Flux au sein de la société Cedrat et du Laboratoire d'Electrotechnique de Grenoble.

1.3.3 Plan de développement

La réalisation de ce cahier des charges nous a conduit à un plan de développement défini pour mes trois années de thèse. Ainsi, la réalisation successive des différents buts à atteindre représente des jalons qui correspondent à différentes maquettes et logiciels métiers. Ces prototypes nous ont permis de tester et de valider le bien fondé et les performances de notre approche.

1.4 Etat de l'art

1.4.1 Approche pour la conception de logiciel métier

Dans cette partie, nous allons dresser un rapide état de l'art dans le domaine de la conception de logiciel métier. Partant d'un existant sous la forme d'un logiciel de calcul généraliste, il existe potentiellement deux grandes méthodes de création massive de logiciels métiers. Ces deux méthodes ont leurs spécificités et elles ne peuvent pas être mises en œuvre dans tous les contextes. Leur étude va permettre de mieux comprendre leur fonctionnement et donc de pouvoir choisir celle qui correspond le mieux à nos besoins.

1.4.1.1 Méthode de conception par assemblage de composants

La première méthode que nous avons étudiée est la conception par assemblage. Cette approche consiste à construire un logiciel en assemblant des composants préexistants ainsi que d'autres composants créés spécialement pour le nouveau logiciel.

Pour que cette approche soit efficace, il faut que la boîte à outils que constitue le logiciel généraliste puisse être découpée sous forme de composants, chacun d'entre eux possédant une interface de communication établie et publiée. L'assemblage peut éventuellement être réalisé graphiquement à l'aide d'outils spécialisés. Les Java Beans de Sun peuvent être considérés comme une approche très aboutie dans cette direction [JAVABEANS].

En général cependant, l'interface homme machine est à recoder pour chaque application métier car l'approche composant n'est pas très favorable à la réutilisation des comportements graphiques. En terme de coûts, la construction unitaire est relativement rapide, par contre la maintenance peut être relativement lourde si les fonctionnalités des composants de calcul évoluent. Quelque soit le niveau de la technologie employée, cette méthode par réutilisation et assemblage est utilisée depuis de nombreuses années et est grandement répandue dans l'industrie.

1.4.1.2 Méthode de conception par modélisation du logiciel

La seconde méthode, apparue dans les années 90 et sans doute moins courante que la précédente, consiste à formaliser l'ensemble du comportement de l'application métier par une modélisation extensive. Le modèle correspondant au logiciel souhaité peut alors être interprété par des programmes génériques, qui réalisent les comportements modélisés [MDA] [MOF] [OMG] [MARECHAL-01]. Pour être complet, le modèle du logiciel métier doit décrire les données, les commandes et toute l'interface homme machine.

Pour pouvoir décrire les logiciels métiers par des modèles, il est nécessaire de s'appuyer sur un langage de modélisation. Le formalisme le plus courant est le "Unified Modeling Language" [UML] [OMG] [MULLER-00], qui est une norme internationale. Il est également nécessaire d'employer une machine virtuelle, programme permettant d'exécuter le logiciel prenant comme paramètres les modèles de l'application [MARECHAL-01].

1.4.1.3 Analyse critique des méthodes de conception

Les deux méthodes présentées ci-dessus ont chacune leurs avantages et leurs inconvénients. En fonction des choix de développement choisis, l'une ou l'autre peut s'avérer mieux adaptée.

Pour la création d'une multitude d'applications métiers, le choix de la conception par modélisation s'impose. En effet, la création ou la modification du comportement ainsi que la maintenance des applications sont faciles à réaliser grâce à la technique par modélisation car il suffit de modifier le modèle alors que par l'approche par assemblage, il est nécessaire de modifier directement le programme lui-même.

Par contre, il est vrai que la machine virtuelle dynamique interprétant les modèles d'application est plus complexe à écrire que les divers composants d'un logiciel fabriqué par assemblage. De plus, l'interprétation du modèle par la machine virtuelle peut rendre ce type d'application relativement lente du fait du plus grand nombre d'opérations à réaliser dans ce cas. Enfin, l'interface homme machine étant générique et automatique, elle a un comportement répétitif qui n'est peut être pas toujours aussi bien adapté qu'un dialogue cousu main.

Néanmoins, cette machine virtuelle n'est écrite qu'une seule fois et réutilisée de nombreuses fois. Par ailleurs, son évolution profite d'un seul coup à l'ensemble des applications métiers réalisées avec cette technologie.

Il en ressort que la politique de développement des logiciels métiers que nous avons à mettre en place nécessite d'opter pour la solution de conception par modélisation, le but final des travaux étant de pouvoir créer un grand nombre de logiciels métiers avec des coûts en développement et en maintenance très faibles.

1.4.2 Méta modélisation des logiciels métiers

Après avoir analysé deux approches différentes de développement de logiciel métier, nous avons pu faire un premier choix en optant pour la méthode de conception par modélisation de logiciel. Ce choix nous amène maintenant à étudier l'offre de langage de modélisation disponible dans la littérature.

Avant toute autre chose, il faut introduire la notion de méta modèle. Un méta modèle exécutable est un modèle de données dont les instances décrivent un modèle de données : c'est donc un vocabulaire et une grammaire destinés à créer et à manipuler des modèles. Cette montée en abstraction peut bien sûr se répéter, donnant lieu à un méta méta modèle, et ainsi de suite, répétition a priori sans limite, même si le méta méta modèle est généralement considéré comme suffisamment abstrait et simple pour être le dernier niveau d'abstraction nécessaire. Ces méta modèles sont naturellement particulièrement importants dans une approche de conception par modélisation, la richesse sémantique du modèle de l'application

étant directement liée aux capacités offertes par le méta modèle. Voici une illustration des quatre niveaux de modélisation généralement préconisés actuellement dans la norme Unified Modeling Language [UML].

- ✓ Le niveau "information" ou "données" enregistre les entrées de l'utilisateur final.
- ✓ Le niveau "modèle" est destiné au concepteur du logiciel de simulation. Il définit le langage de l'utilisateur final.
- ✓ Le méta modèle sert à produire un modèle. Le méta modèle est un langage abstrait permettant de décrire différents types de langages.
- ✓ Enfin le méta méta modèle permet de décrire le méta modèle.

Meta meta modèle	*Modèle codé* *(MetaModel , MetaClass, MetaField sont codés à ce niveau)*
Meta modèle	*MetaModel('meta model',* *[MetaClass('Entity',[MetaAttribute('id',String), MetaField('fields',List<Field>)]),* *MetaClass('Field',….)])*
Modèle	*Entity('Region', [Field('name',String),* *Field('formulation',Formulation),* *Field('material',Material)])*
Information	*Region('air',[ScalarFormuation, Air]),* *Region('core',[ScalarFormulation, Iron]),*

Nous allons donc dans un premier temps étudier plus en détails la norme Unified Modeling Language [UML] qui fait office de référence mondiale dans le domaine du génie logiciel. Puis, nous nous intéresserons à une autre approche, l'Adaptive Object Model [AOM], sans doute complémentaire et qui, en tout cas, adopte une option différente mais particulièrement intéressante.

1.4.3 UML, MOF, XMI et autre MDA

L'Unified Modeling Language [UML], normalisé par l'Object Management Group [OMG] en 1997, est le langage le plus utilisé dans le monde de la modélisation. Il permet de modéliser la structure, le comportement, l'architecture et le déploiement d'une application.

Le méta modèle d'Unified Modeling Language [UML] a été proposé et adopté quelques temps plus tard sous le nom de Meta Object Facility [MOF]. Ce concept a pour but d'unifier la représentation des méta modèles, en proposant une norme et ainsi de permettre la construction de base de données indépendantes d'un fournisseur particulier capable de stocker

des modèles et leurs données. Le méta modèle proposé par l'Object Management Group [OMG] est très complet et détaillé. Il peut être enregistré sous forme eXtensible Markup Language [XML] suivant la norme XML Metadata Interchange [XMI].

Depuis, l'Object Management Group [OMG] a proposé l'approche Model Driven Architecture [MDA] qui tente de procurer des outils pour séparer les fonctionnalités du système à réaliser de la plate-forme sur laquelle il est censé tourner.

Bien entendu, Meta Object Facility [MOF] est une norme qui apporte certaines réponses à notre besoin de modélisation des applications métiers. Mais d'autres approches existent.

1.4.4 Adaptive Object Model

L'Adaptive Object Model [AOM] se spécialise dans les applications où les modèles de données évoluent beaucoup. Dans ce cas, l'idée de l'Adaptive Object Model [AOM] est de programmer les méta classes et non les classes elles mêmes, celles-ci n'étant que des instances de ces méta classes et donc à ce titre, très facilement modifiables, y compris pendant l'exécution de l'application [ACHARYA-04] [KHALED-04] [POOLE-00] [RIEHLE-00] [YODER-00] [YODER-01].

Le point fort réside donc dans la rapidité de développement et la réutilisation de tout le code basé sur les méta classes. Par contre, dans cette approche, le comportement des classes du modèle n'est pas ou peu présent.

1.4.5 Analyse critique de la méta modélisation

Les deux approches sont relativement différentes : Meta Object Facility [MOF] dispose d'un méta modèle très complet et volumineux, alors que Adaptive Object Model [AOM] s'appuie sur un méta modèle très compact. Adaptive Object Model [AOM] est destiné à des modèles très changeants, alors que la modification d'un modèle décrit par Meta Object Facility [MOF] impose plus de contraintes. Cependant, Meta Object Facility [MOF] dispose de fonctionnalités de modélisation de comportements qui sont absentes d'Adaptive Object Model [AOM].

Reste quelques points forts dans ces deux approches. L'idée particulièrement séduisante d'Adaptive Object Model [AOM] est de n'écrire du code qu'au niveau méta, rendant ainsi toute cette programmation complètement indépendante de l'application ciblée et de son modèle. Par contre, les comportements de l'application à concevoir ne peuvent être décrit selon ce formalisme et une difficulté demeure de ce point de vue pour des applications de

calcul comme les nôtres. Meta Object Facility [MOF] associé à Unified Modeling Language [UML] apporte un standard pour gérer des modèles et leurs données.

Constatons néanmoins qu'aucune de ces deux approches ne proposent de solution complètement satisfaisante : le grain de Meta Object Facility [MOF] est très fin, ce qui est logique pour réaliser un langage standard, mais pas nécessaire pour notre application. A contrario, Adaptive Object Model [AOM] n'apporte pas les éléments pour gérer la complexité des comportements que nous souhaitons. Enfin ni l'une ni l'autre solution ne décrit la présentation de l'application (l'équivalent du modèle de style eXtensible Stylesheet Language [XSL] de la norme eXtensible Markup Language [XML]).

Notre choix se porte vers une nouvelle méta modélisation avec un méta modèle d'une complexité correspondante à la complexité souhaitée pour nos logiciels métiers. Notre méta modélisation tentera de combiner des bons aspects des deux approches présentées : conserver la dynamicité issue par la programmation au niveau méta comme avec Adaptive Object Model [AOM], produire une méta modélisation s'inspirant de Unified Modeling Language [UML] comme avec Meta Object Facility [MOF]. Nous devrons compléter cela par la description de la présentation de l'application métier.

1.5 Choix

Produire au moindre coût des applications métiers dérivées de notre application généraliste Flux, tel est notre but. Nous avons ébauché une direction de réflexion : pousser le plus loin possible la modélisation de l'application souhaitée. Nous devons désormais développer une proposition de solution.

Unified Modeling Language [UML] est reconnu comme le langage de modélisation universel pour les applications orientées objets. De fait, il couvre de nombreux aspects des besoins de modélisation : structure, comportements, et architecture. Si le modèle Unified Modeling Language [UML] est si proche de l'application finale, pourquoi ne pas rendre le langage Unified Modeling Language [UML] "exécutable" ?

Faisons le parallèle avec le langage Java. Contrairement aux autres langages de programmation, le code compilé est portable sur n'importe quelle plate-forme. Pour atteindre cet objectif, les concepteurs de Java ont proposé une machine virtuelle, interpréteur et compilateur vers le processeur réel de la machine.

Nous pourrions rendre le langage Unified Modeling Language [UML] exécutable en fournissant une machine virtuelle Unified Modeling Language [UML], capable d'interpréter

un modèle en présentant l'application modélisée. Cependant, pour combler l'absence de modèle de présentation dans Unified Modeling Language [UML], nous allons enrichir ce langage avec les concepts manquants. Nous proposons donc le langage Application Modeling Language [AML], langage exécutable par une machine virtuelle, modélisant intégralement la structure, le comportement et la présentation d'une application.

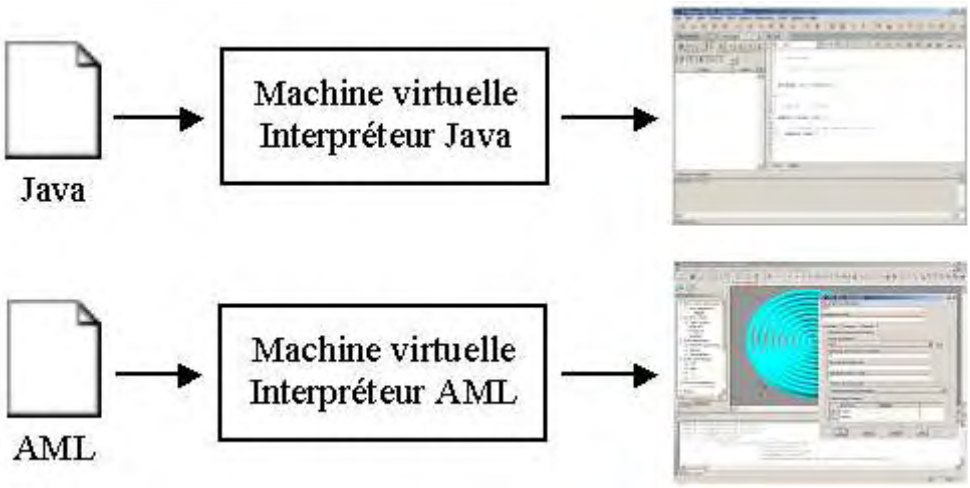


Figure 1.2 : Machines virtuelles Java et AML

Si nous atteignons nos objectifs, quelles en sont les conséquences ?

Tout d'abord le travail du concepteur se trouve radicalement simplifié. Retoucher une application revient à retoucher son modèle, concevoir une nouvelle application consiste à la modéliser. Tous les objectifs que nous nous étions fixés : personnalisation, objets métiers, et logiciels métiers sont accessibles rapidement et à un coût de développement uniquement attaché au coût de développement de la machine virtuelle.

La machine virtuelle est développée sur les méta classes et nous retrouvons bien ici ce qui fait la puissance de l'approche Adaptive Object Model [AOM]. Par contre, pour pouvoir introduire les comportements fins et en particulier les algorithmes de calculs, la modélisation ne suffit plus ou plus précisément est inefficace. Nous devons donc aussi pouvoir programmer au niveau modèle, comme cela se pratique d'habitude, pour créer tous les comportements calculatoires.

Reste à définir le langage Application Modeling Language [AML]. Pour cela, il suffit de définir son méta modèle. Il devra permettre de modéliser des aspects structure de données, comportements locaux et globaux et présentation de l'application. Le méta méta modèle Meta Object Facility [MOF] est à la fois trop détaillé et pas assez étendu pour cette fonction. Nous

devons donc aussi proposer un méta modèle adapté au langage Application Modeling Language [AML].

L'utilisation d'une telle démarche d'exécution pour nos applications comporte un certain nombre d'avantages. En effet, les fonctionnalités ne sont programmées qu'une seule fois dans la machine virtuelle et elles peuvent être ensuite utilisées par tous les logiciels adaptables. De plus, cette réutilisabilité est encore accrue par l'utilisation des mécanismes d'héritage et de surcharge issus de la Programmation Orientée Objet [POO], qui seront présents dans la machine virtuelle Application Modeling Language [AML].

1.6 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté l'intérêt économique et industriel de proposer des logiciels métiers complémentaires des logiciels généralistes comme Flux. Le logiciel métier adopte le vocabulaire et la démarche d'un métier identifié. Pour le concepteur, c'est un environnement plus naturel, qui doit lui permettre de mener ses projets avec une plus grande efficacité.

Pour structurer nos travaux, nous avons proposé une démarche, qui passe progressivement de la personnalisation de l'application généraliste, aux applications métiers en passant par les objets métiers insérés au sein d'applications généralistes. Puis, nous avons confronté deux approches de conception d'applications métiers, soit par assemblages de composants soit par adaptation du logiciel généraliste selon un modèle.

Après avoir opté pour une conception de nos logiciels métiers par modélisation, nous avons étudié deux approches de méta modélisation complémentaires. Meta Object Facility [MOF] issu de la norme Unified Modeling Language [UML] apporte une standardisation des modèles. L'approche Adaptive Object Model [AOM] procure une grande réactivité des modèles qui sont considérés comme des instances de méta classes.

Nous proposons finalement de développer un nouveau langage de modélisation, le langage Application Modeling Language [AML], basé sur un méta modèle qui combine les deux points forts précédents et qui procure des outils de modélisation de la présentation de l'application. Un tel langage doit permettre de développer rapidement des applications métiers et de maîtriser les coûts de maintenance.

2 Conception de logiciels adaptables

2.1 Introduction

Le chapitre précédent nous a permis d'étudier différentes méthodes de conception de logiciels existantes dans le domaine du génie logiciel. Cette recherche, orientée par des contraintes spécifiques de développement ainsi que des exigences économiques de réduction de coûts, a conduit à affiner le cahier des charges pour nos travaux. Notre analyse nous incite à considérer les logiciels métiers comme des dérivations du logiciel généraliste. Un logiciel métier est conçu comme un ensemble de modèles de données, de comportements et de présentation et d'un projecteur, appelé machine virtuelle. Ainsi, notre choix de conception par modélisation du logiciel a entraîné l'étude des différents méta modèles récemment mis en œuvre par les concepteurs de par le monde. Pour pouvoir effectuer une modélisation fine de nos applications nous avons donc décidé de créer notre propre langage de modélisation, l'Application Modeling Language, basé sur un méta modèle correspondant au mieux à la complexité des logiciels que nous souhaitons construire.

Par choix, le langage AML est considéré comme "exécutable", moyennant un interpréteur adapté. Cependant, si l'algorithmique de l'application métier est complexe, son codage dans un langage de programmation classique devient nécessaire. Pour assurer une bonne intégration de ce code métier, nous proposons de construire un générateur de code capable de traduire en langage objet le modèle structurel et comportemental (voir figure 2.1).

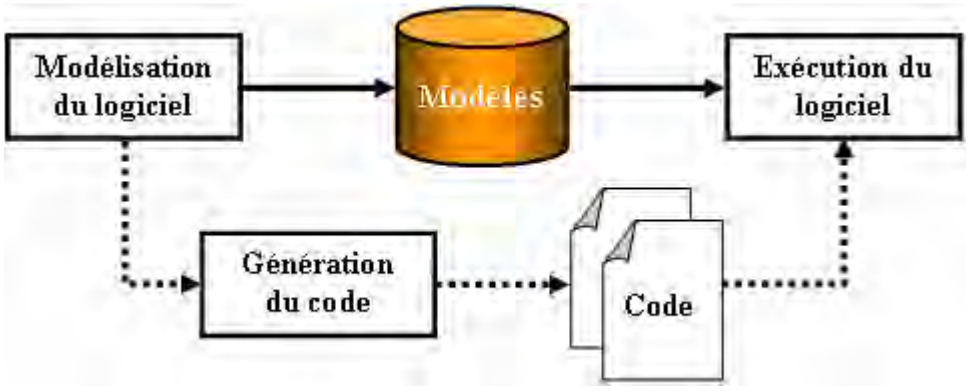


Figure 2.1 : Processus d'exécution

Ce chapitre est tout d'abord consacré à la description des modèles. La modélisation, écrite en langage AML, est composée de 3 parties : la modélisation statique, définissant la structure du logiciel, la modélisation dynamique caractérisant son comportement et la modélisation de

sa présentation graphique. L'analyse de chacun de ces modèles sera présentée, permettant de mieux comprendre les principes de notre modélisation de logiciel.

Une fois les différents modèles de l'application décrits et créés, ils peuvent ensuite être convertis en langage de programmation pour insérer du code métier et interprété par la machine virtuelle AML. Nous décrirons les fonctionnalités du générateur proposé ainsi que le fonctionnement et l'architecture de la machine virtuelle.

La dernière partie de ce chapitre est consacrée aux notions de contextualisation et de masquage. Ces concepts inspirés des techniques de la Programmation Orientée Objet [POO] permettent une meilleure structuration de l'application et donc des gains de développement non négligeables.

2.2 Modèle statique d'un logiciel

La modélisation statique d'un logiciel métier consiste à créer un modèle de données correspondant à la structure de ce logiciel. En fonction du niveau de modélisation souhaitée, ce modèle devra pouvoir décrire tout ou partie des composants constituant l'application. Parmi ces composants figurent, avant tout, les objets manipulés par les utilisateurs ainsi que les commandes proposées dans le logiciel, mais aussi l'application elle-même. Quant à l'interface homme machine, nous lui consacrerons une partie à part entière, même si elle est très fortement liée au modèle statique.

Les concepteurs de logiciel de par le monde s'appuient de plus en plus sur les derniers développements du domaine du génie logiciel. Issu de ce domaine, le formalisme Unified Modeling Language [UML] permet de modéliser d'une manière très performante les caractéristiques statiques d'une application, en particulier à travers les diagrammes de classe. Dans notre langage AML, nous réutiliserons donc une bonne part des notions constitutives d'Unified Modeling Language [UML], mais le méta modèle associé sera plus compact et plus dédié, car destiné à des acteurs du milieu du Génie Electrique. Nous allons maintenant présenter un à un les concepts proposés en Application Modeling Language [AML] pour décrire le modèle statique d'une application.

2.2.1 Les modèles de données

- **Définition**

Le modèle de données est l'ensemble des définitions des structures de données. Une application repose nécessairement sur un modèle de données mais peut aussi en utiliser plusieurs en particulier dans le cas d'applications modulaires qui traitent différents domaines

d'application. Un modèle de données peut être divisé en sous-modèles. Une telle structuration conduit à une composition en arbre. Les têtes des arbres sont des modèles principaux alors que les autres nœuds sont des sous-modèles. Les sous-modèles appartiennent au modèle père et ne peuvent pas être partagés avec d'autres modèles de données.

La structuration d'un modèle de données en sous-modèles de données est une structuration logique. Elle permet de séparer des domaines d'activités différents. A ce titre, un modèle de données est associé à une application. Ainsi, différentes applications décrites peuvent partager ou non un modèle de données. Les structures de données, éléments constituant le modèle de données, appartiennent à un et un seul modèle de données.

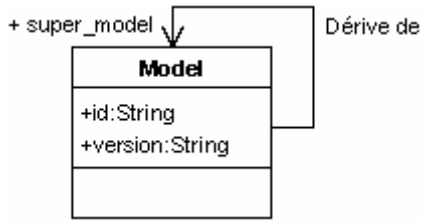


Figure 2.2 : Méta modèle des modèles de données

- **Structure**

Le modèle de données est défini par un nom dont la saisie est obligatoire ainsi qu'une version. Par convention d'utilisation, le nom est en minuscule, le séparateur entre mots étant le caractère "_". Un sous-modèle possède aussi une référence vers son modèle père.

```
Model(id='self',
      version='1.00',
      derivesFrom=Model['general'])
```

Figure 2.3 : Exemple d'un modèle de données

2.2.2 Les paquetages

- **Définition**

Les paquetages donnent la répartition physique des structures de données. Les paquetages forment un arbre auquel doit correspondre un arbre de répertoires. La tête de cet arbre est un paquetage principal alors que les autres nœuds sont des sous-paquetages. L'arbre des paquetages et celui du modèle de données ne sont pas nécessairement identiques car ils correspondent à deux vues d'un même ensemble de structures de données : vue logique et vue physique.

Les structures de données appartiennent au plus à un paquetage, mais peuvent aussi ne pas être localisées dans un paquetage particulier.

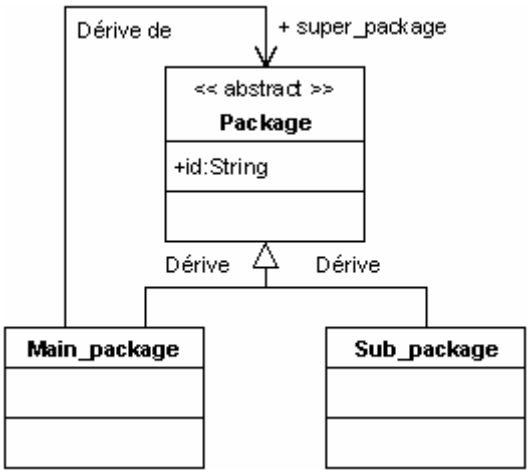


Figure 2.4 : Méta modèle des paquetages

- **Structure**

Le paquetage est défini par un nom dont la saisie est obligatoire. Par convention d'utilisation, ce nom est en minuscule, le séparateur entre mots étant le caractère "_". Un sous-paquetage possède aussi une référence vers son paquetage père.

```
SubPackage(id='self',
           superPackage=MainPackage['general'])
```

Figure 2.5 : Exemple d'un paquetage

2.2.3 Les structures de données

Les structures de données définissent tous les objets manipulés par une application. Pour notre exemple, le logiciel FluxSpire, logiciel de conception de micro inductance, ces objets sont le nombre de spires, le rayon de la première spire, l'espacement entre spires, la section … (voir figure 2.6).

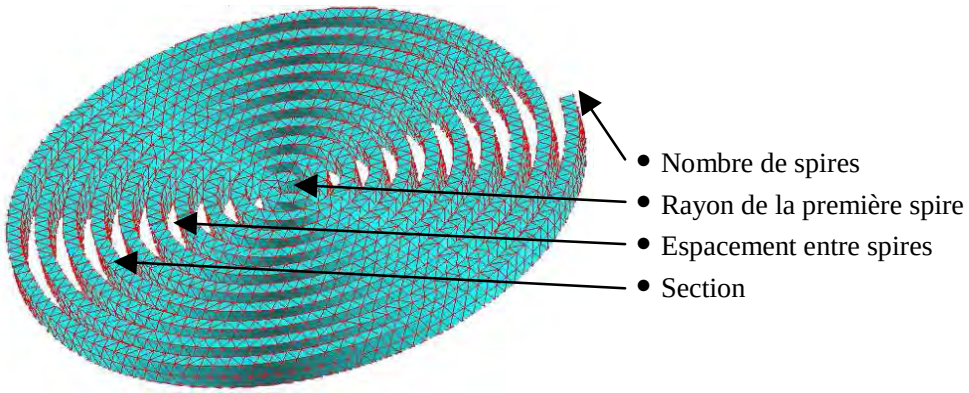


Figure 2.6 : Micro inductance

Ces structures de données doivent donc permettre de modéliser tous les types de données manipulables dans une application. Les principales données sont des objets que nous appelons entités, avec leurs champs permettant de les associer entre elles. D'autres données comme les types intrinsèques et les énumérations sont également utilisés pour décrire les notions d'un logiciel adaptable (voir figure 2.7).

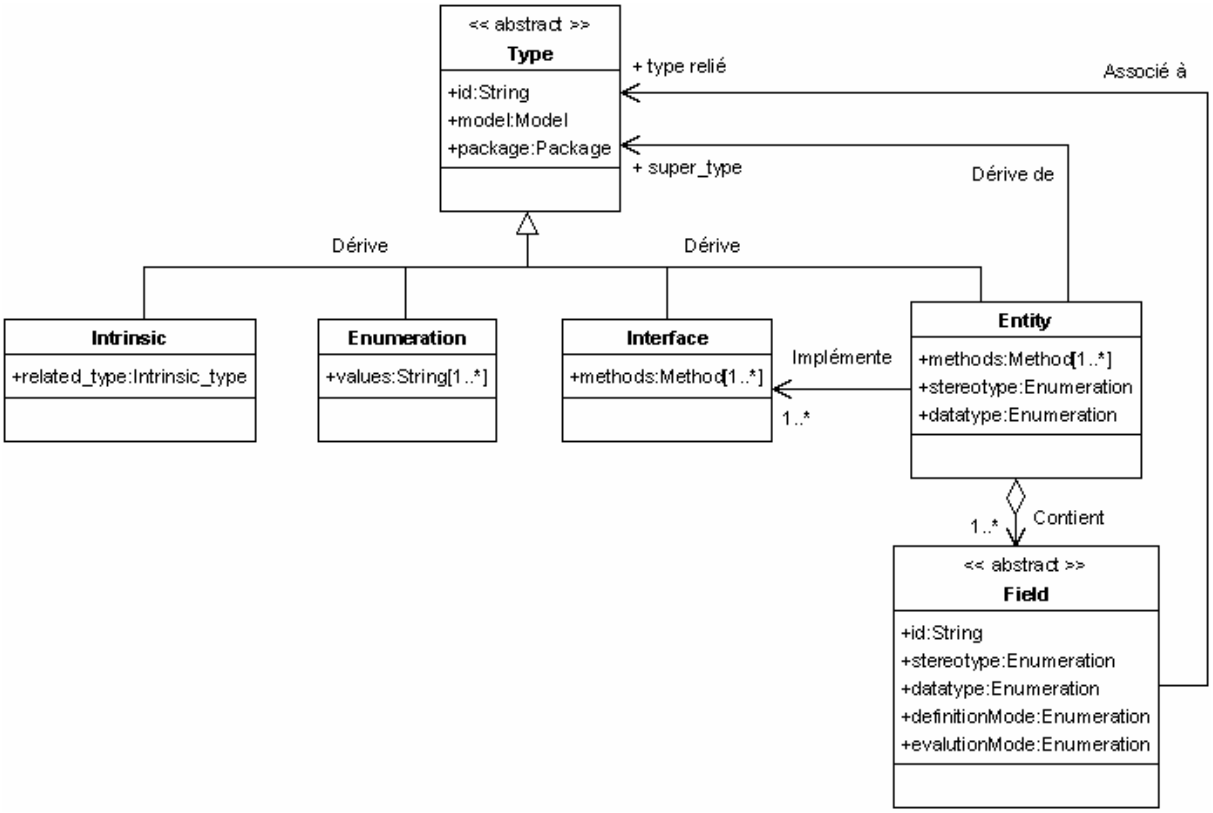


Figure 2.7 : Méta modèle des structures de données

2.2.3.1 Les types intrinsèques

- **Définition**

Les types intrinsèques définissent les types intrinsèquement connus et manipulés par une application. Parmi les types intrinsèques, il y a naturellement les types de base classiques : entiers, réels, chaînes de caractères. Mais les types intrinsèques peuvent aussi servir à définir des types de base dédiés : URL, image, formule, …

- **Structure**

Un type intrinsèque est défini par :

✓ Un nom (obligatoire).

✓ Un modèle de données (obligatoire).

✓ Un paquetage (optionnel).

✓ Un type sous-jacent choisi parmi : integer, real, numeric, string, numeric_or_string, boolean ou void (obligatoire).

```
Intrinsic(id='integer',
          modelOwner=Model['general'],
          relatedType='INTEGER',
          packageOwner=SubPackage['self'])
```

Figure 2.8 : Exemple d'un type intrinsèque

2.2.3.2 Les énumérations

- **Définition**

Les énumérations permettent de définir des champs dont les valeurs sont discrètes et dont l'ensemble des valeurs possibles est connu et fixé. Par exemple, une liste de couleurs peut être définie comme une énumération.

- **Structure**

Une énumération est définie par :

✓ Un nom (obligatoire).

✓ Un modèle de données (obligatoire).

✓ Un paquetage (optionnel).

✓ Une liste de valeurs séparées par des ";". Ces valeurs sont traitées comme des chaînes. Le caractère ";" est toujours interprété comme un séparateur.

```
Enumeration(id='colors',
            modelOwner=Model['general'],
            val='red;green;blue',
            packageOwner=SubPackage['self'])
```

Figure 2.9 : Exemple d'une énumération

2.2.3.3 Les interfaces

- **Définition**

Les interfaces définissent des comportements associables à des entités. Ils ne comportent que la déclaration des méthodes prenant part à la réalisation du comportement. Les entités peuvent satisfaire plusieurs comportements. Ceux-ci sont définis par les interfaces et forment un réseau : une interface peut étendre une ou plusieurs interfaces qui sont alors des super interfaces.

Remarque : Cette notion d'interface est similaire à la notion d'interface en Java ou de classe purement abstraite en C++.

- **Structure**

Une interface est définie par :

✓ Un nom (obligatoire).
✓ Un modèle de données (obligatoire).
✓ Un paquetage (optionnel).
✓ Un ensemble d'interfaces qui est étendu par cette interface.
✓ Un ensemble de méthodes.

```
Interface(id='interface',
          ModelOwner=Model['general'],
          PackageOwner=SubPackage['self'])
```

Figure 2.10 : Exemple d'une interface

2.2.3.4 Les entités

- **Définition**

Les entités définissent les notions manipulées par une application. Une entité est constituée de données (champs) et de services (méthodes) qui lui appartiennent et qui ne peuvent pas être partagés avec d'autres entités. Elle implante une liste d'interfaces et dérive éventuellement d'une entité ce qui permet de définir un héritage simple.

Elle possède aussi un stéréotype qui est choisi parmi :

✓ **Abstrait :** Si l'entité est abstraite au sens de la Programmation Orientée Objet [POO] et si elle décrit tous les comportements des sous entités qui en dérivent, ce type peut servir à manipuler toutes les instances des types dérivés de manière transparente (notion de polymorphisme). *Exemple : "section" (type abstrait) et "section circulaire", "section rectangulaire" en types concrets.*
✓ **Concret :** Si l'entité est concrète.
✓ **Nœud :** Si l'entité est abstraite au sens de la Programmation Orientée Objet [POO] mais ne connaît pas l'ensemble des comportements des sous entités qui en dérivent. Contrairement au type abstrait, ce type n'est pas générique pour tous les types dérivés.
✓ **Encapsuleur :** Si l'entité encapsule une entité concrète pour réutiliser et adapter tout ou partie de ses comportements.

Une entité possède enfin un type pour le stockage sous forme persistante :

- ✓ **Persistant :** L'entité fait partie du modèle persistant.
- ✓ **Transitoire :** L'entité ne fait pas partie du modèle persistant, elle ne doit pas être sauvegardée et sera régénérée lors de la lecture de la base de données.

Remarque : Les champs possèdent aussi une information de persistance qui surcharge celle définie sur l'entité.

- **Structure**

Une entité est définie par :

- ✓ Un nom (obligatoire).
- ✓ Un modèle de données (obligatoire).
- ✓ Un paquetage (optionnel).
- ✓ L'entité étendue par cette entité (optionnelle).
- ✓ Le stéréotype de l'entité (abstract, concrete, node, wrapper).
- ✓ Un ensemble d'interfaces implantées par cette entité (optionnel).
- ✓ Un ensemble de champs (optionnel).
- ✓ Un ensemble de méthodes (optionnel).
- ✓ Un type de stockage (information de persistance).

```
Entity(id='Self',
       modelOwner=Model['general'],
       stereotype='CONCRETE',
       packageOwner=SubPackage['self'],
       datatype='PERSISTENT')
```

Figure 2.11 : Exemple d'une entité

2.2.3.5 Les agrégats

- **Définition**

Les agrégats permettent de définir des paquets d'instances d'une entité. Le type de ces paquets est précisé par la notion de collection qui définit les cardinalités minimum et maximum et la catégorie du paquet : ensemble (unicité des membres, pas d'indexation, pas d'ordre), liste (pas d'unicité des membres, pas d'indexation, ordre préservé), table (pas d'unicité des membres, indexation et ordre préservé), sac (pas d'unicité des membres, pas d'indexation, pas d'ordre).

- **Structure**

Un agrégat est défini par :

✓ Un nom (obligatoire).

✓ Un modèle de données (obligatoire).

✓ Un paquetage (optionnel).

✓ L'entité dont elle forme un paquet d'instances (obligatoire).

```
Aggregate(id='Aggregate',
          modelOwner=Model['general'],
          type=Entity['Self'],
          collectionType=SetCollection(
           lowerBound=IntegerBound(boundValue=1),
           upperBound=IntegerBound(boundValue=5)))
```

Figure 2.12 : Exemple d'un agrégat

2.2.3.6 Les champs

- **Définition**

Les champs correspondent aux données des entités. Ils appartiennent à ces entités et ne peuvent donc pas être partagés par plusieurs entités. Les champs servent à représenter les relations entre objets. Ils pointent vers des structures de données qui définissent les types des objets associés. Les champs peuvent être simples ou de type collection. Dans ce dernier cas, ils relient une instance d'entité à un paquet d'instances d'entité, paquet défini par la collection associée.

Un champ dispose d'un mode de définition qui est choisi parmi :

✓ **Obligatoire :** La saisie de ce champ est obligatoire pour construire l'instance d'entité.

✓ **Finale :** La saisie de ce champ est obligatoire et la valeur saisie ne peut plus être modifiée par la suite. C'est en particulier le cas des identificateurs d'objets.

✓ **Optionnelle :** La saisie de ce champ n'est pas obligatoire. Si aucune donnée n'est introduite, une valeur par défaut sera fournie.

✓ **Dérivé :** La valeur du champ est obtenue par calcul à partir de données obligatoires et sa valeur est utile à la compréhension de l'instance considérée.

✓ **Interne :** La valeur du champ est obtenue par calcul à partir de données obligatoires et sa valeur n'est pas utile à la compréhension de l'instance considérée.

La relation définie par le champ se classe dans une des quatre catégories suivantes, conformément à la norme Unified Modeling Language [UML] :

- ✓ **Association :** Il s'agit de la relation par défaut lorsqu'un autre type de relation n'est pas applicable. Elle associe une instance à une ou plusieurs instances de la même entité ou d'une autre entité.
- ✓ **Agrégation :** Une agrégation définie une relation de type "partie d'un objet complexe". Elle peut se traduire par : "est constitué de". Il n'existe cependant pas d'appropriation stricte de l'instance reliée, puisqu'elle peut faire partie de plusieurs agrégations.
- ✓ **Composition :** Une composition est une agrégation "forte". Les deux objets reliés par une composition ont une même durée de vie. L'objet relié ne peut pas être partagé dans le cadre d'autres relations. Il est privé à l'objet qui en est constitué.
- ✓ **Identification :** Un identificateur est une composition qui permet d'indexer une entité. Dans une identification, l'entité utilisée doit donc définir une relation d'ordre. Cette notion n'existe pas directement dans la norme Unified Modeling Language [UML] sous cette forme.

Les champs possèdent un attribut "mode d'évaluation" permettant le guidage des arbres d'évaluation.

- ✓ **Non :** La modification de l'instance utilisée par ce champ ne nécessite pas la réévaluation de l'instance possédant ce champ.
- ✓ **Rétro propagation :** La modification de l'instance utilisée par ce champ nécessite la réévaluation de l'instance possédant ce champ.

Les champs possèdent enfin un type pour le stockage sous forme persistante dans une base de données :

- ✓ **Persistant :** Ce champ fait partie du modèle persistant.
- ✓ **Transitoire :** Ce champ ne fait pas partie du modèle persistant, il ne doit pas être sauvegardé et sera régénéré lors de la lecture d'une base de données.

Remarque 1 : Les entités possèdent aussi une information de persistance qui est surchargée par celle définie sur les champs.

Remarque 2 : Lorsqu'un champ utilise une entité de type intrinsèque ou énuméré, une partie de l'information concernant la catégorie de la relation (association, agrégation ou composition) perd son sens. En effet, une relation vers un type intrinsèque ou énuméré se comporte implicitement comme une composition. Par ailleurs, toujours dans le cas d'un champ utilisant une entité de type intrinsèque ou énuméré, une valeur par défaut et un domaine de validité peuvent être définis.

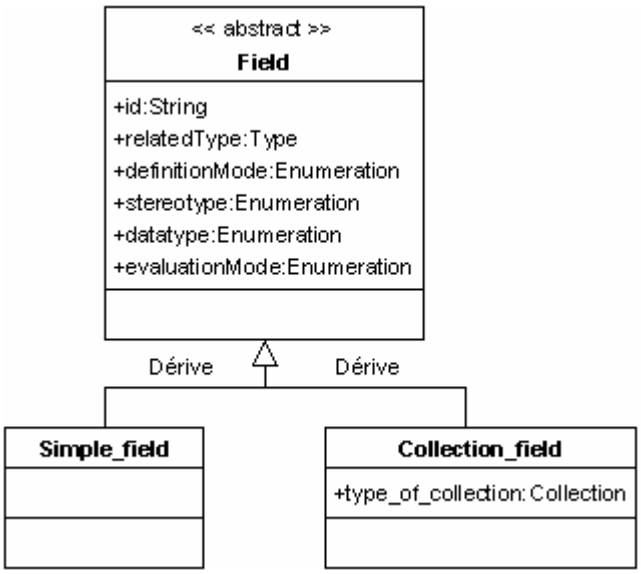


Figure 2.13 : Méta modèle des champs

- **Structure**

Un champ se crée à partir de l'entité qui le possède. Il est défini par :

- ✓ Un nom (obligatoire) composé du nom de l'entité suivi d'un "." puis suivi du nom du champ.
- ✓ Un type "champ simple" ou "champ collection". Dans le cas d'une collection, le type de la collection et les cardinalités de la collection sont précisés.
- ✓ Le type relié par la relation.
- ✓ Le mode de définition (forced, final, optional, derived, internal).
- ✓ Le stéréotype de la relation (association, composition, agrégation, identification).
- ✓ Des informations additionnelles, à savoir : les valeurs par défaut éventuelles, le domaine de validité éventuel, le type de persistance, le mode d'évaluation.

```
SimpleField(id='Self.name',
            relatedType=Intrinsic['string'],
            definitionMode='FORCED',
            stereotype='IDENTIFICATION',
            datatype='PERSISTENT',
            evaluationMode='NONE')
```

Figure 2.14 : Exemple d'un champ

2.2.4 Les commandes

Les commandes définissent toutes les actions proposées par un logiciel. Dans notre exemple, le logiciel FluxSpire, ces actions sont la construction automatique de la géométrie

de la micro inductance, l'affectation automatique des paramètres physiques aux régions de l'inductance ou tout simplement la destruction d'une micro inductance.

Les commandes sont dérivées en plusieurs types, les méthodes, les procédures ainsi que les méthodes intrinsèques. La grande différence entre les méthodes et les procédures est que les méthodes sont rattachées à une entité, alors que les procédures sont indépendantes et correspondent aux actions classiques d'un logiciel. Toutes ces différentes commandes peuvent nécessiter un ou plusieurs arguments (voir figure 2.15).

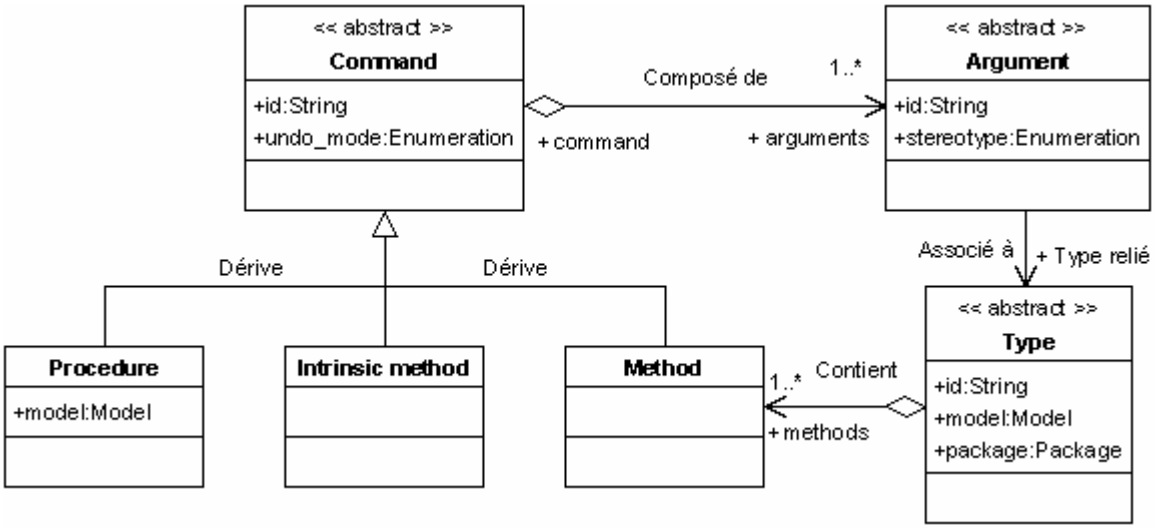


Figure 2.15 : Méta modèle des commandes

2.2.4.1 Les procédures

- **Définition**

Les procédures définissent les différents comportements d'une application. Une procédure est constituée de données (arguments) permettant de fournir les valeurs d'entrées et de sortie de la fonction à réaliser. Cela permet de décrire précisément l'interface de la fonction qui correspond à la procédure représentée.

- **Structure**

Une procédure est définie par :

✓ Un nom (obligatoire).
✓ Un modèle de données (obligatoire).
✓ Un mode de configuration des actions "défaire" (obligatoire).
✓ Un ensemble d'arguments en entrée (optionnel).
✓ Un ensemble d'arguments en sortie (optionnel).
✓ Des informations additionnelles, à savoir : l'emplacement du code à exécuter.

```
Procedure(id='Open_project',
          undo_mode='RAZ_NOTIFY_ALL',
          modelOwner=Model['general'])
```

Figure 2.16 : Exemple d'une procédure

Les procédures possèdent un mode de configuration des actions "défaire" (undo). Ce mode est composé de deux parties distinctes. La première partie correspond à l'action à réaliser sur la pile des actions "défaire" à la fin de l'exécution de la procédure :

- ✓ **Raz :** Ce mode permet de vider la pile des actions "défaire" après l'exécution de la procédure.
- ✓ **Ignore :** Ce mode permet d'ignorer l'exécution de la procédure et donc de ne pas modifier la pile des actions "défaire".
- ✓ **Active :** Ce mode permet d'incrémenter la pile des actions "défaire" avec l'action permettant de défaire la procédure exécutée.

La deuxième partie de ce mode permet de définir le mode de notification et donc de rafraîchissement désiré après l'exécution de la procédure :

- ✓ **Notify_all :** Ce mode entraîne un rafraîchissement de toutes les instances du type concerné par cette procédure. Ainsi, après l'exécution de la procédure "maillage" par exemple, tous les éléments de maillage créés sont rafraîchis.
- ✓ **Notify_none :** Ce mode n'entraîne aucun rafraîchissement. Il est utilisé par les procédures ne modifiant pas les instances.
- ✓ **Notify_incremental :** Ce mode entraîne le rafraîchissement des instances modifiées lors de l'exécution de la procédure. Ainsi, après l'exécution de la procédure "créer point" seul le point créé est rafraîchi.

2.2.4.2 Les méthodes

- **Définition**

Les méthodes correspondent aux services définis par les entités. Elles appartiennent à ces entités et ne peuvent donc pas être utilisées par d'autres entités. Elles permettent de représenter les comportements possibles des entités. De mêmes que les procédures elles sont constituées d'arguments en entrée et en sortie ainsi que d'un mode de configuration des actions "défaire".

- **Structure**

Une méthode est définie par :

✓ Un nom (obligatoire).

✓ Un mode de configuration des actions "défaire" (obligatoire).

✓ Un ensemble d'arguments en entrée (optionnel).

✓ Un ensemble d'arguments en sortie (optionnel).

✓ Des informations additionnelles, à savoir : l'emplacement du code à exécuter.

```
Method(id='Self_affect_region',
       undo_mode='ACTIVE_NOTIFY_INCREMENTAL')
```

Figure 2.17 : Exemple d'une méthode

2.2.4.3 Les méthodes intrinsèques

- **Définition**

Les méthodes intrinsèques correspondent à des actions réflexes applicables à tous les types d'objet. Elles permettent de modéliser les actions intrinsèques telles que la création, la suppression, la modification d'une donnée dans le logiciel. Ces actions sont donc automatiques et gérées pour tout nouveau type d'entité saisi.

De même que les méthodes ou les procédures elles sont constituées d'arguments en entrée et en sortie ainsi que d'un mode de configuration des actions "défaire".

- **Structure**

Une méthode intrinsèque est définie par :

✓ Un nom (obligatoire).

✓ Un mode de configuration des actions "défaire" (obligatoire).

✓ Un ensemble d'arguments en entrée (optionnel).

✓ Un ensemble d'arguments en sortie (optionnel).

✓ Des informations additionnelles à savoir : l'emplacement du code à exécuter.

```
Intrinsic_method(id='Edit_instance',
                 undo_mode='ACTIVE_NOTIFY_ALL')
```

Figure 2.18 : Exemple d'une méthode intrinsèque

2.2.4.4 Les arguments

- **Définition**

Les arguments correspondent aux paramètres des commandes. Ils appartiennent à ces commandes et ne peuvent donc pas être partagés par plusieurs commandes. Les arguments servent à représenter les relations entre données et commandes. Ils correspondent aux valeurs d'entrées et de sorties des commandes décrites.

De même que les champs, les arguments peuvent être simples ou de type collection. Dans ce dernier cas, ils permettent de définir un paquet d'instances d'objet, paquet défini par la collection associée.

Comme les champs, un argument dispose aussi d'un mode de définition. Ces différents modes permettent de définir si l'argument est optionnel ou obligatoire.

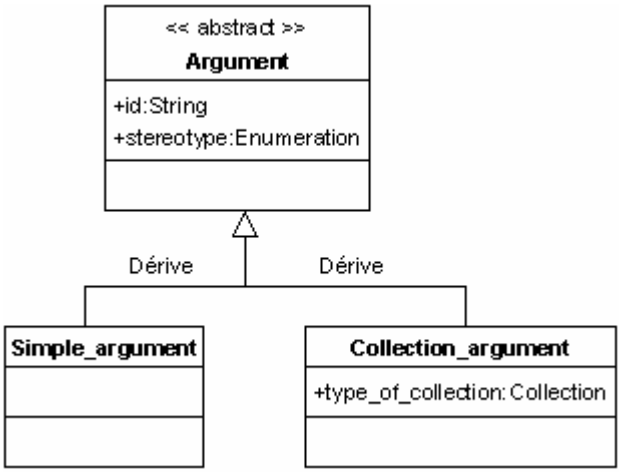


Figure 2.19 : méta modèle des arguments

- **Structure**

Un argument est défini par :

✓ Un nom (obligatoire) composé du nom de la commande suivi d'un "." puis suivi du nom de l'argument.

✓ Un type "argument simple" ou "argument collection". Dans le cas d'une collection, le type de la collection et les cardinalités de la collection sont précisés.

✓ Le type relié par la relation.

✓ Le mode de définition (forced, optional).

✓ Des informations additionnelles, à savoir : les valeurs par défaut éventuelles.

```
SimpleArgument(id='Open_project.file_name',
               relatedType=Intrinsic['string'],
               definitionMode='FORCED')
```

Figure 2.20 : Exemple d'un argument

2.2.5 Les applications

- **Définition**

Les applications représentent les entités exécutables. Elles sont composées d'une liste de modèles de données ainsi que de plusieurs cas d'utilisation.

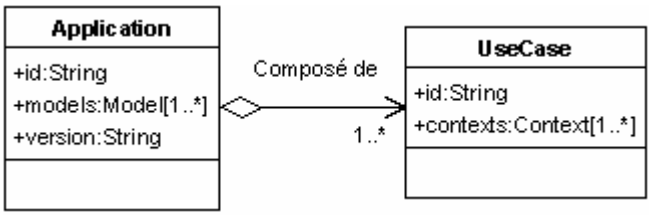


Figure 2.21 : Méta modèle des applications

- **Structure**

Une application est définie par :

✓ Un nom (obligatoire).

✓ Un ou plusieurs modèles de données (obligatoire).

✓ Un nom de version (optionnel).

✓ Une liste de cas d'utilisation (obligatoire). Les cas d'utilisation sont la propriété de l'application : elles ne peuvent donc pas être partagées par plusieurs applications mais peuvent être référencées hors de l'application.

```
Application(id='FluxSpire',
            models=[Model['general'],Model['self']],
            version='FluxSpire3D',
            usesCases=[useCase['FluxSpire3D']])
```

Figure 2.22 : Exemple d'une application

2.2.6 Les cas d'utilisation

- **Définition**

Les cas d'utilisation caractérisent les différents aspects de l'application. Dans l'approche Unified Modeling Language [UML], le diagramme de cas d'utilisation recense le

comportement souhaité par le client pour son application. C'est une forme de représentation graphique d'un cahier des charges.

Pour illustrer l'intérêt des cas d'utilisation, considérons quelques exemples dans le logiciel Flux. Ce dernier est composé de différents cas d'utilisation correspondant aux grandes familles d'études : magnéto statique, magnéto harmonique, magnétique transitoire, conduction électrique, électro statique, électro harmonique, électrolyse, thermique permanent, thermique transitoire … (voir figure 2.23).

Figure 2.23 : Cas d'utilisation du logiciel Flux

- **Structure**

Un cas d'utilisation est défini par :

✓ Un nom (obligatoire).

✓ Une liste de contextes (obligatoire). Les contextes servent à la présentation et au comportement global de l'application. Ils permettent de structurer les cas d'utilisations en domaines, par exemple géométrie, maillage … Les contextes sont exposés plus longuement dans le paragraphe 2.3.2.3.

```
UseCase(id='FluxSpire3D',
        contexts=[Context['FluxSpireGeo3D'],
                  Context['FluxSpireMesh3D']])
```

Figure 2.24 : Exemple d'un cas d'utilisation

2.3 Modélisation de la présentation

Tous les éléments de modèle proposés jusqu'à présent concernent la structuration des notions utilisées dans les logiciels adaptables. Ces informations doivent être complétées par les modèles de comportement et de présentation.

Dans cette partie, nous considérerons les aspects de présentation. Pour rendre notre modèle "exécutable", nous devons le compléter par des informations sur la représentation

graphique des données et des commandes, mais aussi sur l'interface homme machine générale du logiciel. Nous allons examiner ces deux aspects l'un après l'autre.

2.3.1 Représentation graphique des données et des commandes

Concernant le dialogue avec l'utilisateur proprement dit, il peut prendre trois formes : graphique sous forme de boîtes de dialogues, arborescente avec des menus associés à chaque nœud des arbres de représentation ou textuelle sous la forme d'un langage de commande. Le langage AML doit nous permettre d'incorporer dans notre modèle d'application des informations destinées à décrire l'interface utilisateur de tous nos objets. Grâce à des moteurs algorithmiques appropriés, ces informations vont produire dynamiquement l'interface utilisateur. Les informations supplémentaires destinées à la gestion dynamique de l'interface utilisateur viennent enrichir des structures déjà exposées lors de la définition des modèles de données et de commandes ou sont de nouvelles structures de données.

2.3.1.1 Les informations communes

D'une manière générale, une des caractéristiques communes des informations de l'interface utilisateur est la gestion du dialogue multilingue. L'ensemble des noms et commentaires va devoir être donné dans toutes les langues cibles de l'application. Par ailleurs, la présence d'un dialogue sous forme de langage de commande va aussi imposer de fournir le nom, monolingue cette fois et généralement en anglais, de l'entité vue sous sa forme langage de commande. Bien sûr, il peut y avoir identité entre : le nom de l'entité, le nom représentant la partie graphique et le nom représentant le langage de commande. Mais généralement les noms se distinguent pour les raisons suivantes :

- ✓ Le nom de l'entité est destiné à devenir le nom d'une classe. La convention préconisée est d'utiliser un identificateur éventuellement composé de plusieurs mots, sans espace de séparation entre mots et chaque mot commençant par une majuscule.
- ✓ Le nom représentant la partie graphique de l'objet est généralement plus explicite. Il n'y a plus de limitation sur la présence de séparateurs, au contraire. Ce nom sera celui présenté aux utilisateurs finaux.
- ✓ Le nom représentant le langage de commande de l'objet peut s'approcher du nom graphique ou de celui de l'entité. Néanmoins, il doit être compact, sans séparateur et adopter les mêmes conventions que pour le nom de l'entité. Une version abrégée du nom représentant le langage de commande peut aussi être fournie.

A ces informations basiques, il faudra naturellement ajouter de l'aide. Elle se présente sous deux formes :

- ✓ Aide courte sous la forme d'un message multilingue.
- ✓ Aide longue sous la forme d'un renvoi vers une page d'aide.

2.3.1.2 Les catégories

- **Définition**

Les catégories permettent de répartir les champs dans une boîte de dialogue en proposant des onglets ou dans un arbre en introduisant des nœuds nommés sous le nœud représentant l'instance. Elles ne sont par contre pas utilisées dans le dialogue par langage de commande.

- **Structure**

La structure des catégories est volontairement simple : Elle se réduit à un label multilingue.

```
Category(id='General',
         defaultLabel='Général',
         additionalLabels=['General'])
```

Figure 2.25 : Exemple d'une catégorie

2.3.1.3 Les entités

Aux informations destinées à la description d'une entité, il faut ajouter toutes les informations communes de l'interface utilisateur. Pour ces informations complémentaires, quelques précisions doivent être apportées :

- ✓ Le label est l'identificateur utilisé dès que l'utilisateur doit manipuler l'entité (boîtes de dialogues, arbres, ...). Sa présence surcharge le nom de l'entité dans le dialogue avec l'utilisateur.
- ✓ Le commentaire sert dans le cadre de la saisie lors de la présence d'entités dérivées. Le commentaire de l'entité de base procure le texte de la question pour guider le choix du sous type concret.
- ✓ La bulle d'aide permet de présenter à l'utilisateur une définition explicite de l'entité.
- ✓ La catégorie permet de répartir les champs dans la boîte de dialogue. La catégorie de l'entité procure une catégorie par défaut pour l'ensemble des champs de l'entité. Cette catégorie peut être surchargée en définissant une catégorie spécifique pour certains champs de l'entité.

A ces informations communes à toutes les commandes ou entités manipulables sous les formes graphique et langage de commande sont ajoutées des informations spécifiques aux entités :

- ✓ Un ensemble de droits pour l'entité en fonction du contexte et d'une liste de profils utilisateurs (voir 2.3.3 Contextualisation).
- ✓ Un mode de représentation de l'entité : une entité peut être autonome ou non. Ce mode permet de définir si l'entité est manipulable par l'utilisateur dans le logiciel ou si elle ne doit pas être visible.

Figure 2.26 : Méta modèle des informations de type interface utilisateur des entités

2.3.1.4 Les champs

Aux informations destinées à la description d'un champ, il faut ajouter toutes les informations communes de l'interface utilisateur :

- ✓ Le label est l'identificateur utilisé dès que l'utilisateur doit manipuler l'entité (boîtes de dialogues, arbres, ...). Sa présence surcharge le nom du champ dans le dialogue avec l'utilisateur.
- ✓ Le commentaire procure le texte de la question. Il permet de construire le texte précédant le composant de dialogue pour la saisie du champ.
- ✓ La bulle d'aide permet d'expliquer à l'utilisateur les informations à saisir pour ce champ. Elle peut permettre, par exemple, de présenter les limites de validité affectées à ce champ.
- ✓ La catégorie permet de répartir les champs dans la boîte de dialogue. La catégorie de l'entité procure une catégorie par défaut pour l'ensemble des champs de l'entité. La catégorie spécifique du champ permet de surcharger celle définie pour l'entité.

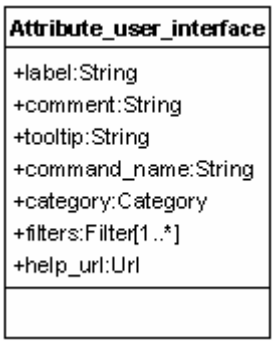


Figure 2.27 : Méta modèle des informations de type interface utilisateur des champs

A ces informations, communes à toutes les commandes ou entités, est ajouté un ensemble de droits pour le champ en fonction de l'application, du contexte et d'une liste de profils utilisateur (voir 2.3.3 Contextualisation).

2.3.1.5 Les commandes

De même que pour les entités, pour parfaire la modélisation des commandes, il faut ajouter au modèle toutes les informations communes de l'interface utilisateur. Ces informations complémentaires méritent quelques précisions :

- ✓ Le label est l'identificateur utilisé dès que l'utilisateur doit manipuler la commande (boîtes de dialogues, menus, ...). Sa présence surcharge le nom de la commande dans le dialogue avec l'utilisateur.
- ✓ Le commentaire procure le texte de la question. Il permet de construire le texte précédant le composant de dialogue pour l'exécution de la commande.
- ✓ La bulle d'aide permet d'expliquer à l'utilisateur les informations concernant la commande.

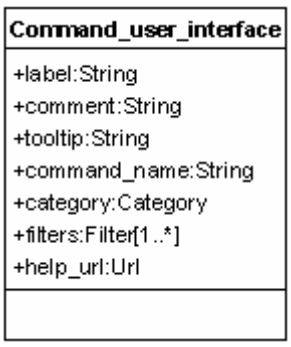


Figure 2.28 : Méta modèle des informations de type interface utilisateur des commandes

A ces informations, communes à toutes les commandes ou entités, est ajouté un ensemble de droits pour la commande en fonction de l'application, du contexte et d'une liste de profils utilisateur (voir 2.3.3 Contextualisation).

2.3.1.6 Les arguments

De même que pour les entités, pour parfaire la modélisation des commandes, il faut ajouter au modèle toutes les informations communes de l'interface utilisateur. Ces informations complémentaires méritent quelques précisions :

- ✓ Le label est l'identificateur utilisé dès que l'utilisateur doit manipuler l'argument (boîtes de dialogue, ...). Sa présence surcharge le nom de l'argument dans le dialogue avec l'utilisateur.
- ✓ Le commentaire procure le texte de la question. Il permet de construire le texte précédant le composant de dialogue pour la saisie de l'argument.
- ✓ La bulle d'aide permet d'expliquer à l'utilisateur les informations à saisir pour cet argument. Elle peut permettre, par exemple, de présenter les limites de validité affectées à cet argument.

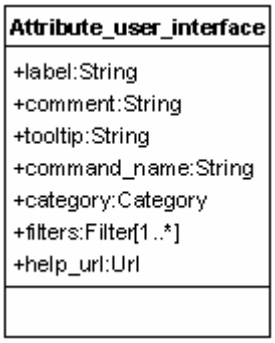


Figure 2.29 : Méta modèle des informations de type interface utilisateur des arguments

A ces informations communes à toutes les commandes ou entités sont ajoutées un ensemble de droits pour l'argument en fonction, de l'application, du contexte et d'une liste de profils utilisateurs (voir 2.3.3 Contextualisation).

2.3.2 L'interface homme machine globale

Pour les applications issues de la modélisation, l'interface homme machine générale est relativement standardisée. Elle est composée de différents menus, arbres de représentations, barres d'outils, de vues graphiques, d'une fenêtre d'écho des commandes successives et d'une zone d'entrée de commandes sous leur forme textuelle (voir figure 2.30).

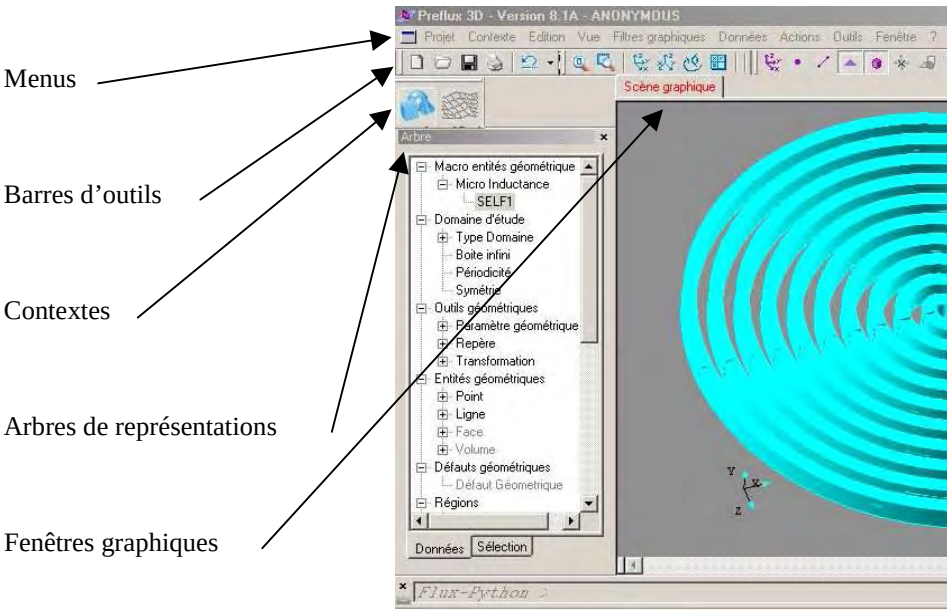


Figure 2.30 : Standard d'interface homme machine d'un logiciel modélisé.

Si toutes ces notions sont généralement présentes dans les logiciels actuels, la notion de contexte demande peut être quelques explications supplémentaires. Un contexte représente une vue restreinte des structures et comportements du logiciel. Les contextes peuvent être utilisés pour décrire des étapes successives dans l'utilisation du logiciel (par exemple, géométrie, maillage, physique, résolution et post traitement dans un logiciel de simulation par éléments finis) ou plus généralement des modules distincts de l'application complète. Chaque contexte possède ses propres menus, arbres d'entités, barres d'outils et fenêtres graphiques.

Dans les applications modélisées, nous avons aussi souhaité prendre en compte la personnalisation des applications en fonction de profils d'utilisateurs pré établis (expert, novice, utilisateur occasionnel par exemple). Les commandes disponibles, les familles d'objets représentées peuvent changer en fonction du profil sélectionné.

Nous allons incorporer dans le langage AML les moyens de décrire l'ensemble de ces informations. Le principe de cette description s'appuie sur la mise en place de liens entre les composants de l'interface homme machine (menus, arbres …) et le modèle de données (entités, commandes) du logiciel modélisé.

2.3.2.1 Les applications

Aux informations destinées à la description structurelle d'une application, il faut ajouter :

- ✓ Les labels, commentaire et texte d'aide courte de l'application dans les différentes langues (voir informations communes).

- ✓ Une liste de profils utilisateurs disponibles dans l'application. Les profils utilisateurs sont la propriété de l'application : ils ne peuvent donc pas être partagés par plusieurs applications mais peuvent être référencés hors de l'application.

2.3.2.2 Les cas d'utilisation

Pour gérer la complexité de nos applications, chaque cas d'utilisation est subdivisé en contextes dans lesquels seules certaines commandes ou objets sont accessibles. Ainsi, le modèle de données d'un logiciel peut représenter l'application avec ses différents cas d'utilisation, eux-mêmes composés de contextes. Ces derniers ont chacun différents menus, barres d'outils, arbres et vues graphiques (voir figure 2.31).

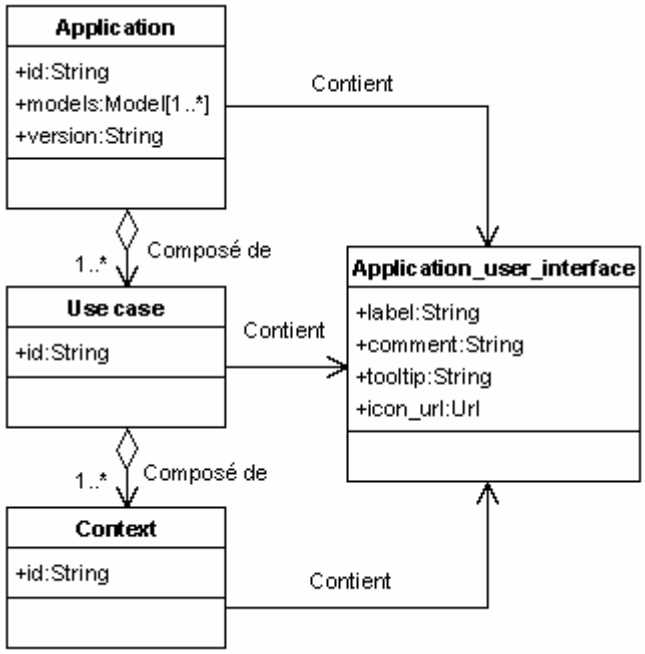


Figure 2.31 : Méta modèle des applications

2.3.2.3 Les contextes

- **Définition**

Les contextes structurent l'utilisation des applications modulaires. Ils permettent de classer les commandes et les entités par domaine d'utilisation, par exemple : maillage, géométrie, résolution, … Associés aux filtres sur les données et les commandes, ils permettent de masquer les entités ou certains champs des entités et les commandes suivant le contexte en cours d'utilisation et le profil de l'utilisateur (voir 2.3.3 Contextualisation). Les contextes sont vus des utilisateurs. Ils comportent donc un label multilingue.

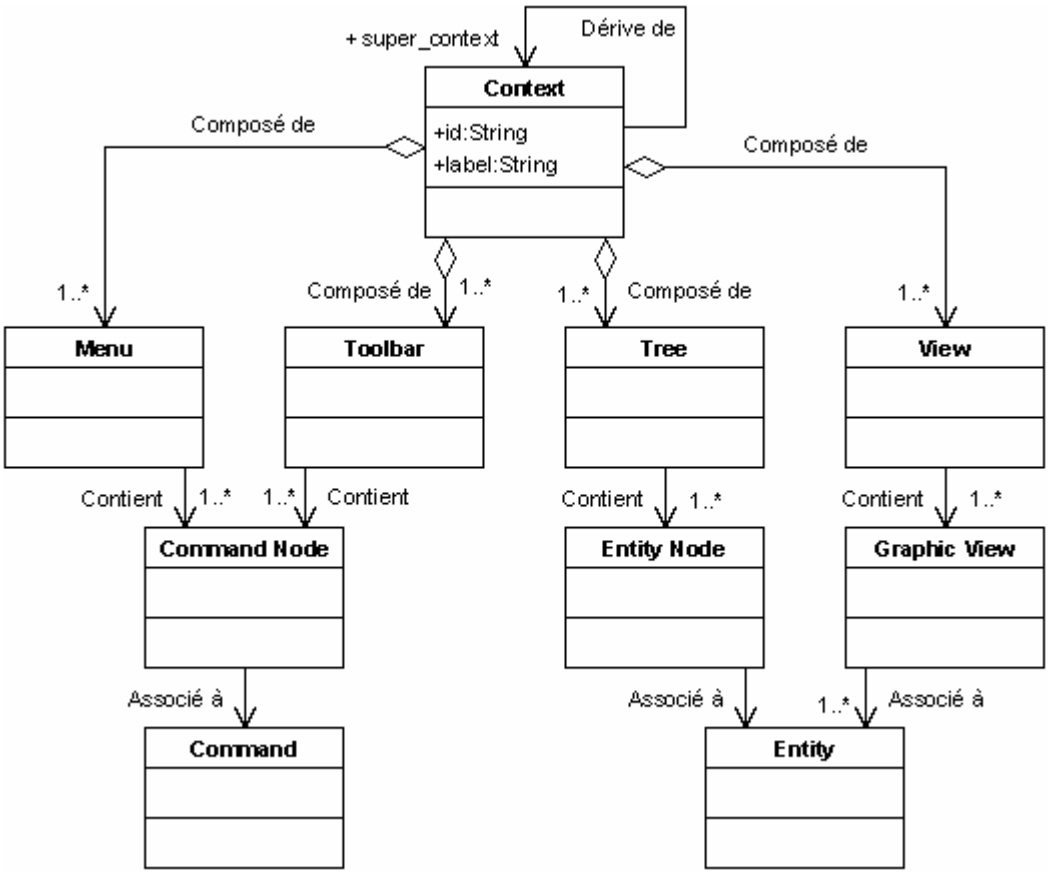


Figure 2.32 : Méta modèle de l'interface homme machine

- **Structure**

Les contextes sont définis par :

✓ Un nom (obligatoire).

✓ Un label multilingue destiné aux utilisateurs (obligatoire).

✓ Le contexte étendu par ce contexte (optionnel).

✓ Une liste d'arbres permettant la représentation des différentes données de l'application.

✓ Une liste de menus représentant les commandes disponibles pour ce contexte.

✓ Une liste de barres d'icônes.

✓ Une liste de vues graphiques représentant les différentes vues disponibles pour ce contexte.

De plus, chaque contexte de l'application modélisée contient la liste de chacun de ces composants constituant l'interface avec les utilisateurs (voir figure 2.32). Les composants de l'interface homme machine peuvent soit être différents ou soit être réutilisés dans plusieurs contextes de l'application.

2.3.2.4 Les profils utilisateurs

- **Définition**

Les profils utilisateur permettent de structurer les différentes catégories d'utilisateur susceptibles d'utiliser l'application. L'utilisateur étant identifié lors de l'ouverture d'une session, l'interface utilisateur devra se configurer pour ne laisser apparaître que les entités et commandes souhaitées et définies comme accessibles par ce profil. La notion de profil utilisateur permet de gérer proprement les niveaux de compétences des utilisateurs. Mais elle permet aussi de tendre vers des applications métiers en faisant disparaître la plupart des commandes et en ne laissant par exemple accessibles que les paramètres, les commandes de résolution et d'exploitation. Les profils utilisateur sont vus des utilisateurs, ils comportent donc un attribut label multilingue.

- **Structure**

Un profil utilisateur est défini par :

✓ Un nom (obligatoire).

✓ Un label multilingue destiné aux utilisateurs (obligatoire).

```
UserProfile(id='all',
            defaultLabel='Tous les utilisateurs',
            additionalLabels=['All users'])
```

Figure 2.33 : Exemple d'un profil utilisateur

2.3.2.5 Les menus

- **Définition**

La modélisation des menus permet de décrire totalement l'aspect souhaité pour sa barre de menu. Un menu est constitué de nœuds pouvant être de différents types :

✓ **Groupe :** Un groupe est composé de nœuds. Ce groupe de nœuds permet donc de décrire les sous-menus. Un groupe contient des informations de type interface utilisateur permettant de décrire le label, le commentaire, la bulle d'aide et l'icône du groupe.

✓ **Commande :** Le nœud de type "Command" permet de pointer sur une des commandes du modèle des commandes. C'est ce type de nœud qui réalise le lien entre les menus et le modèle de données. La représentation d'un nœud de type commande utilise les informations de type interface utilisateur de la commande associée.

✓ **Séparateur :** Le séparateur permet simplement d'insérer des séparateurs dans un menu.

De plus, chaque menu et chaque nœud possèdent des informations de validité permettant de décrire ces conditions d'affichage. En effet, un menu ou un nœud peut être visible ou invisible, ainsi que valide ou grisé. Pour gérer ces informations, des conditions ont été intégrées au modèle de données. Ces conditions permettent de décrire tous les types de situation nécessaires à la gestion de la validité de nos menus.

De même que pour toutes les données de nos logiciels, pour parfaire la modélisation des menus, il faut ajouter au modèle toutes les informations communes de l'interface utilisateur :

- ✓ Un label représentant l'identificateur utilisé dès que l'utilisateur doit manipuler le menu. Sa présence surcharge le nom du menu dans le dialogue avec l'utilisateur.
- ✓ Une bulle d'aide permet d'expliquer plus en détail le contenu du menu.

- **Structure**

Un menu est défini par :

- ✓ Un nom (obligatoire).
- ✓ Une liste de nœud (optionnelle).
- ✓ Des conditions d'affichage (optionnelles).
- ✓ Des informations de type interface utilisateur (optionnelles).

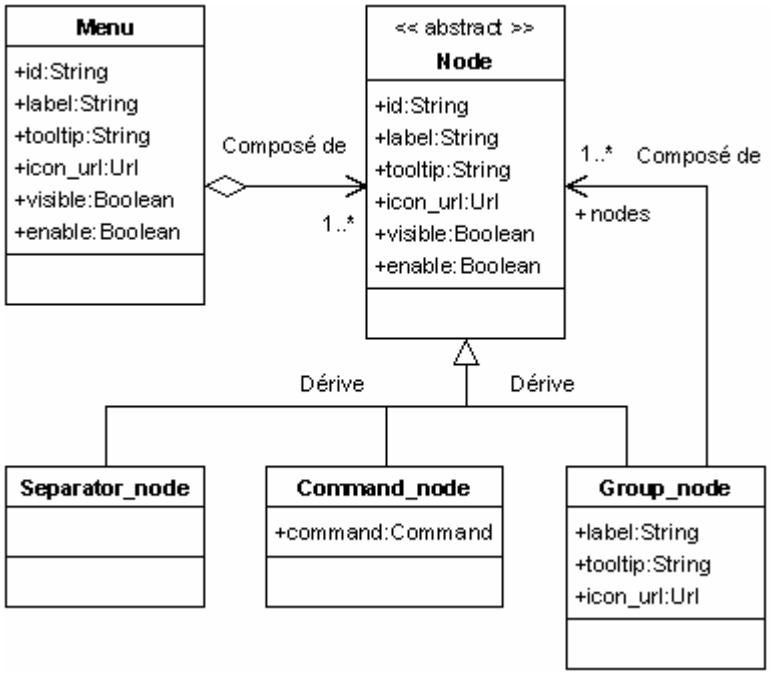


Figure 2.34 : Méta modèle des menus

2.3.2.6 Les barres d'icônes

- **Définition**

Les barres d'icônes sont définies de la même manière que les menus. Elles sont donc composées de nœuds pointant sur des commandes. La représentation est réalisée sous forme de barres au lieu de menu et l'affichage sous forme d'icônes et de bulles d'aide seulement.

- **Structure**

Une interface est définie par :

✓ Un nom (obligatoire).

✓ Une liste de nœud (optionnelle).

✓ Des conditions d'affichage (optionnelles).

✓ Des informations de type interface utilisateur (optionnelles).

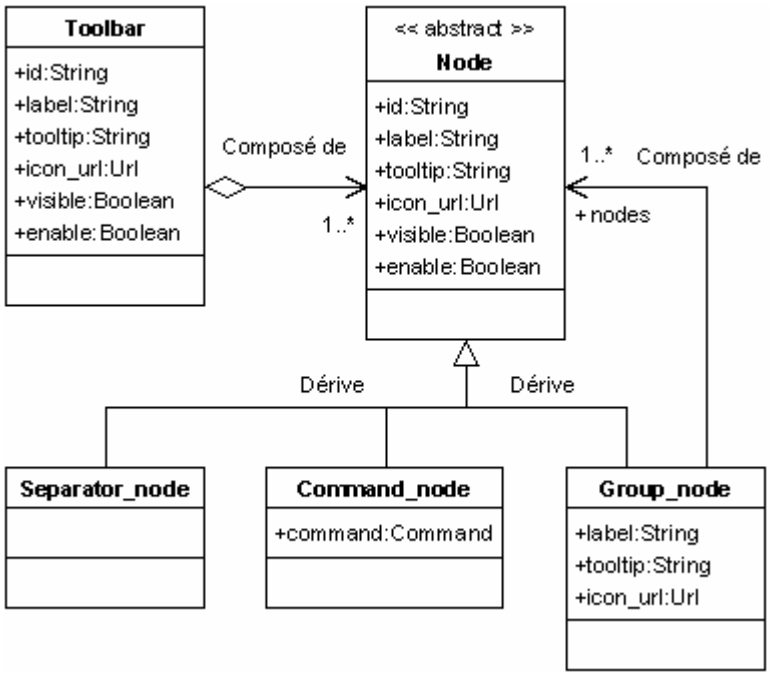


Figure 2.35 : Méta modèle des barres d'icônes

2.3.2.7 Les arbres de représentation

- **Définition**

Les arbres de représentation servent, comme leur nom l'indique, à représenter de manière structurée les différentes données d'une application. Comme les menus, les arbres de représentation sont constitués de nœuds de différents types :

✓ **Groupe :** Un groupe est composé de nœuds. Ce groupe de nœuds permet donc de décrire les sous-branches de l'arbre. Un groupe contient des informations de type interface

utilisateur permettant de décrire le label, le commentaire, la bulle d'aide et l'icône du groupe.

✓ **Entité :** Le nœud de type entité permet de pointer sur une des entités du modèle de données. C'est ce type de nœud qui réalise le lien entre les arbres de représentation et le modèle de données. La représentation d'un nœud de type entité utilise les informations de type interface utilisateur de l'entité associée.

- **Structure**

Un arbre de représentation est défini par :

✓ Un nom (obligatoire).

✓ Une liste de nœud (optionnelle).

✓ Des informations de type interface utilisateur (optionnelles).

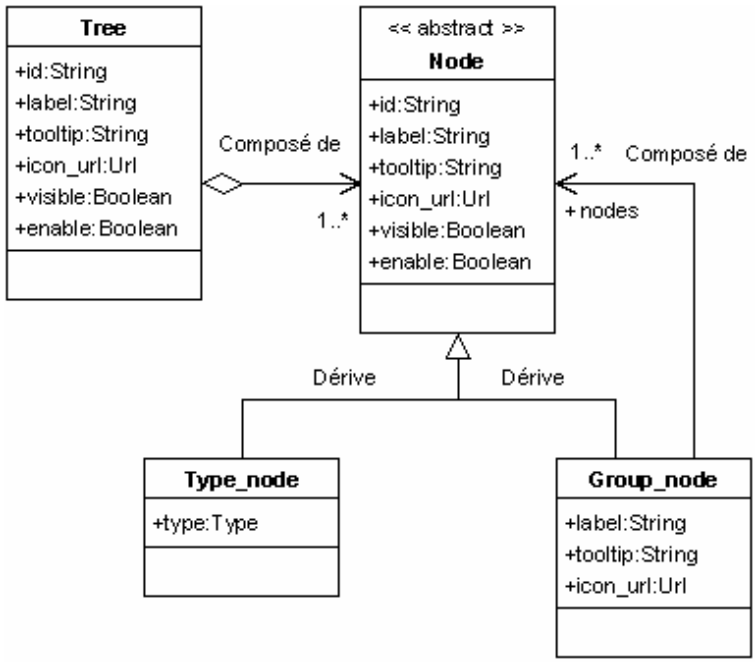


Figure 2.36 : Méta modèle des arbres de représentation

2.3.2.8 Les fenêtres graphiques

- **Définition**

Les fenêtres graphiques sont les éléments d'une application les plus représentatives pour les utilisateurs. Elles permettent de représenter les données de l'application de manière purement graphique et offre ainsi une vision très pertinente pour les utilisateurs.

Une fenêtre graphique est constituée d'une dimension (1D, 2D ou 3D) ainsi que d'un groupe d'entités graphiques. Ces entités sont elles mêmes composées de représentations graphiques et pointent sur une entité du modèle de données (voir figure 2.37).

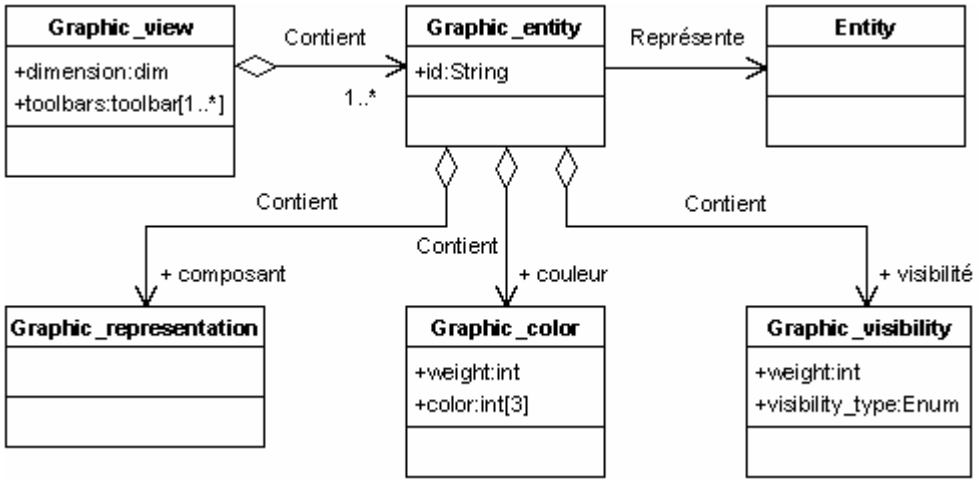


Figure 2.37 : Méta modèle des fenêtres graphiques

Ainsi, une fenêtre graphique permet, par exemple, d'afficher la représentation graphique (un petit rond, un triangle ou un cône) d'un point ou de n'importe quelle autre entité du modèle de données.

- **Structure**

Une fenêtre graphique est définie par :

✓ Un nom (obligatoire).

✓ Un groupe d'entité graphique (optionnel).

✓ Une dimension associée à la fenêtre graphique pouvant être deux ou trois dimensions.

✓ Une liste de barres d'icônes (optionnelle).

2.3.2.9 Les entités graphiques

- **Définition**

Les entités graphiques sont les éléments qui composent une fenêtre graphique. Elles sont constituées d'une représentation graphique définissant la forme physique que devra prendre l'entité pour son affichage. Une entité graphique possède aussi un lien sur une entité du modèle de données. Ce lien permet de réaliser la correspondance entre les objets du problème et l'interface homme machine. Cela permet, par exemple, d'afficher tous les points d'un problème dans la fenêtre graphique.

Ensuite, une entité graphique contient des poids de couleur et de visibilité. Ces poids ont pour fonction de gérer les priorités d'affichage entre les différentes entités graphiques d'une même fenêtre. Ainsi, si deux entités doivent être affichées au même endroit, alors, seule celle qui aura le poids de visibilité le plus élevé sera affichée avec la couleur prioritaire. Ce type de

stratégie d'affichage doit souvent être mis en place. Considérons par exemple des volumes géométriques et des régions physiques. Si l'utilisateur souhaite voir les deux familles d'objets sous forme graphique, le poids défini la priorité de l'une par rapport à l'autre. Dans le cas proposé, on choisira sûrement de privilégier la couleur de la région au dépend de celle du volume .

- **Structure**

Une entité graphique est définie par :

✓ Un nom (obligatoire).
✓ Une référence sur une entité du modèle de données (obligatoire).
✓ Un poids de couleur permettant de gérer les priorités d'affichage des couleurs.
✓ Un poids de visibilité permettant de gérer les priorités d'affichage des entités graphiques.

2.3.3 Contextualisation

Les informations de l'interface utilisateur ont principalement pour vocation de permettre la génération automatique de boîtes de dialogue en associant à la description de structure des données et des commandes des commentaires multilingues. Cependant, l'intégralité des entités et de leurs champs n'est pas destinée à être présentée à l'utilisateur. Certaines informations sont purement internes ou obtenues par dérivation d'autres informations dans le but de faciliter l'algorithmique par exemple. Il faut donc pouvoir masquer ces entités ou ces champs à l'utilisateur. Plus généralement, ce masquage peut éventuellement dépendre du contexte, du cas d'utilisation, et du profil de l'utilisateur, voire tout simplement de l'application. En effet plusieurs applications peuvent réutiliser des modèles de données sans pour autant se présenter de la même manière. Donc, le modèle de présentation de l'application doit être enrichi d'un mécanisme de masquage adapté.

Nous avons déjà vu que chaque entité ou commande est enrichie par les informations de représentation de l'interface utilisateur. Ces informations contiennent un groupe de masques permettant de décrire le comportement de l'objet en fonction du contexte et de l'utilisateur. Les masques sont énumérés. Ainsi, une entité ou un champ, peut être créable, modifiable, supprimable ou non dans un contexte donné. D'autre part, les commandes et leurs arguments peuvent être exécutables, visibles ou non visibles (voir figure 2.38).

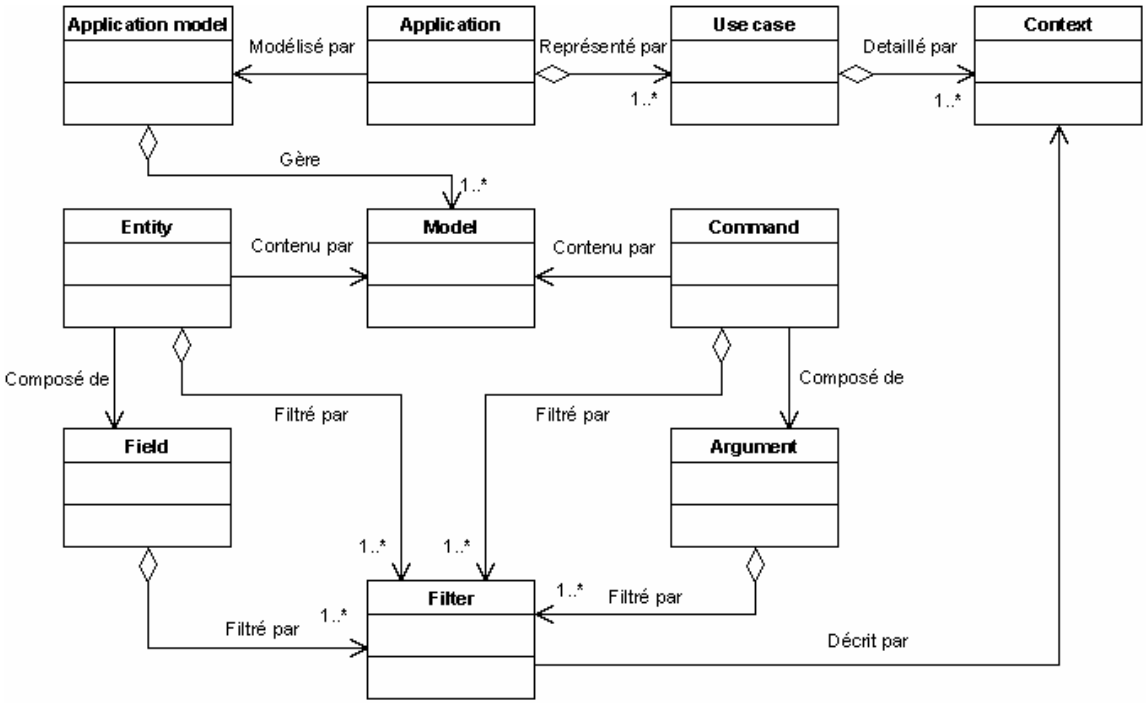


Figure 2.38 : Méta modèle représentant le concept de contextualisation

Ce concept joue donc un rôle important dans la création de la boîte correspondante à l'action souhaitée, il permet d'obtenir des boîtes différentes en fonction du contexte actif et de l'utilisateur. Par exemple, dans le logiciel métier FluxSpire, la modification du nombre de spires d'une micro inductance dans le contexte "géométrie" est permise alors que la même information n'est visible qu'en lecture seule dans le contexte "maillage" (voir figure 2.39).

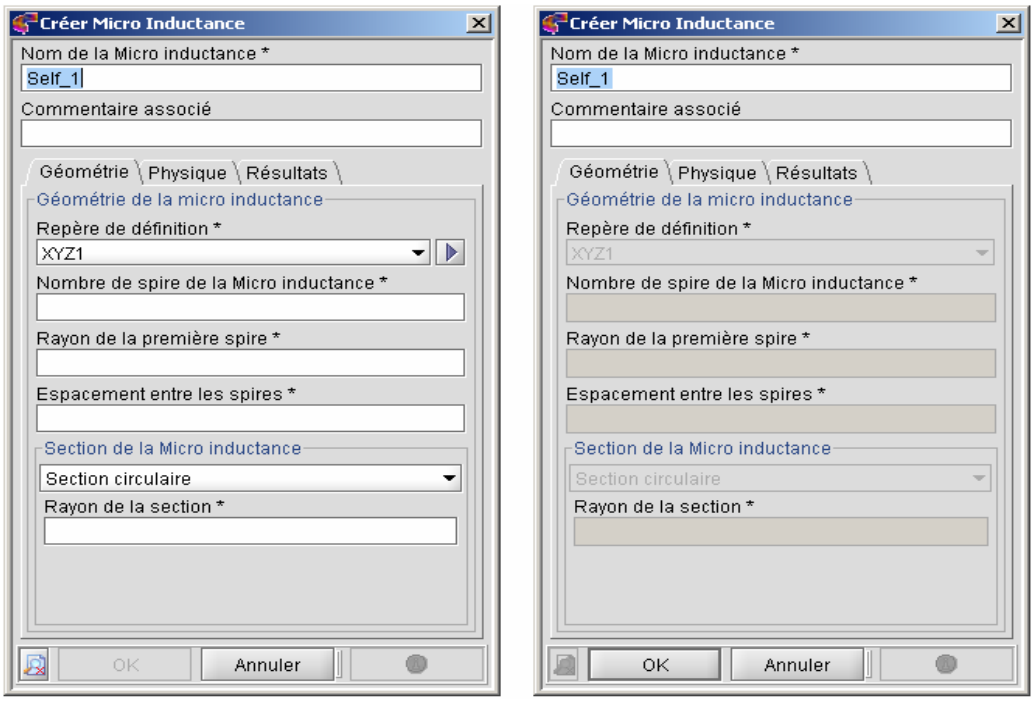


Figure 2.39 : Boîte de dialogue d'une micro inductance en contexte géométrie puis maillage

Pour pouvoir utiliser convenablement le concept de contextualisation, il faut connaître le mécanisme d'évaluation des masques. Les masques sont surchargés selon le mécanisme d'héritage issu de la Programmation Orientée Objet [POO]. Ce mécanisme, utilisé lors de l'exécution du logiciel métier, évalue pour chaque entité, commande et leurs champs ou arguments associés, le type de masquage actif à l'instant de leur utilisation (voir figures 2.40 et 2.41).

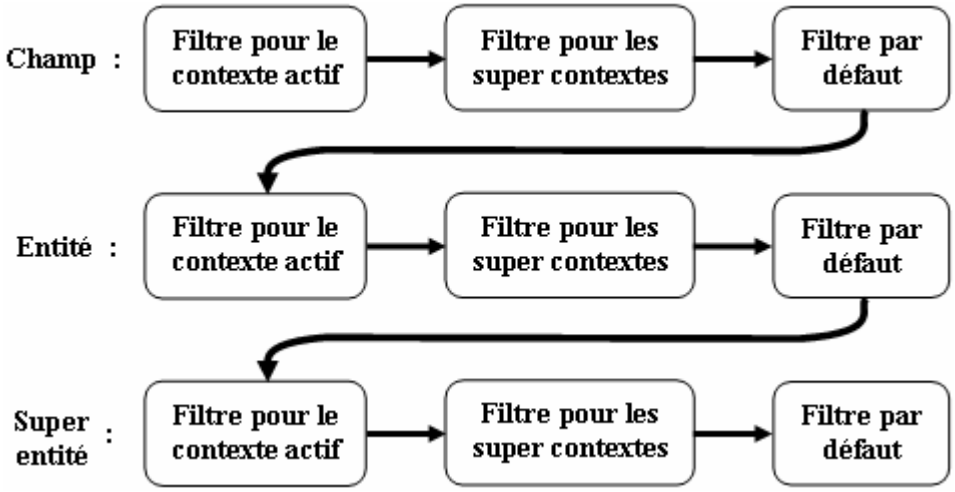


Figure 2.40 : Mécanisme d'évaluation des masques pour les champs et les entités

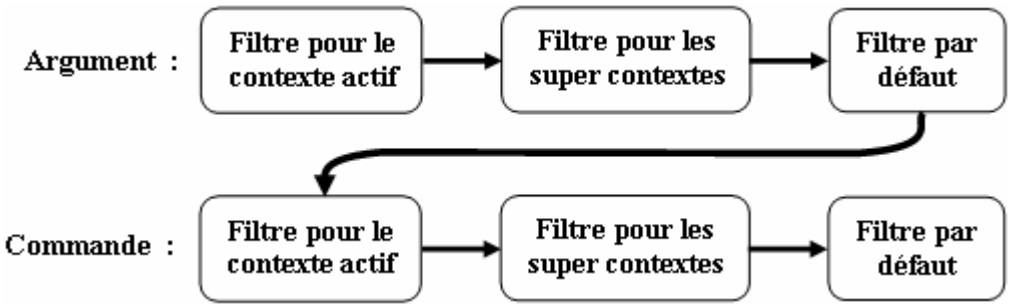


Figure 2.41 : Mécanisme d'évaluation des masques pour les arguments et les commandes

Le principe de fonctionnement présenté sur ces deux figures est le suivant. Lorsque la machine virtuelle doit afficher un champ, elle cherche dans celui-ci l'information de masquage pour le contexte actif, si elle n'en trouve pas, elle cherche alors pour les super contextes du contexte actif, puis dans le masquage par défaut du champ. Si aucun filtre n'est trouvé alors la machine continue sa recherche par l'entité portant le champ, avec le même mécanisme. Si toujours aucun filtre n'est trouvé se sont les super entités qui seront étudiées. Le premier filtre trouvé fait office de filtre actif pour la configuration actuelle du logiciel. Le fonctionnement pour les arguments suit le même raisonnement, après avoir cherché le filtre sur l'argument, la machine virtuelle étudie la commande portant cet argument.

La notion de masquage est aussi très utile dans la modélisation de l'interface homme machine. En effet, la configuration des arbres, menus et icônes par contextes et donc par cas d'utilisation permet là aussi d'utiliser les mêmes menus dans plusieurs contextes en obtenant des résultats différents. Par exemple, une commande d'un menu peut avoir le masque exécutable dans un certain contexte et non visible dans un autre.

En conclusion, ce concept est primordial pour notre démarche car il permet de pouvoir utiliser les mêmes entités, les mêmes champs, les mêmes menus en ayant des résultats graphiques différents suivant l'application ou un contexte d'utilisation. Nous pouvons donc mutualiser le modèle de données ainsi que le modèle graphique et ne pas décrire plusieurs fois les mêmes objets. Dans l'optique de création d'un grand nombre de logiciels métiers, il est essentiel de pouvoir utiliser ce concept de contextualisation.

2.4 Modèle dynamique d'un logiciel

Après avoir étudié les modèles statiques et de présentation, nous allons maintenant explorer la modélisation des aspects dynamiques d'un logiciel et le méta modèle qui en découle. Dans ce domaine, si on s'en réfère à la norme Unified Modeling Language [UML], il existe principalement trois types de formalisme : les diagrammes d'interactions (séquence et collaborations), les diagrammes d'état transition et les diagrammes d'activité. Nous n'avons malheureusement pas pu explorer l'ensemble de ces voies. Il semble que seuls les diagrammes d'états transitions et d'activités soient réellement pertinents pour notre type de modélisation, les diagrammes d'interaction étant d'une finesse sans doute excessive par rapport à nos objectifs. En définitive, nous n'avons porté notre intérêt que sur la modélisation des "workflow".

La modélisation des workflow dans un logiciel métier sert à décrire les enchaînements d'actions que le concepteur de l'application souhaite proposer à ses utilisateurs. Elle permet par exemple de décrire les macro actions spécifiques du métier ciblé, mais aussi les assistants destinés à aider les utilisateurs pendant l'exécution d'une commande complexe.

Cette modélisation doit pouvoir intégrer des opérateurs logiques permettant de structurer les enchaînements d'actions. Ces opérateurs liés à des actions sont par exemple les actions conditionnelles, les boucles d'actions, les branches parallèles … [MANOLESCU-01].

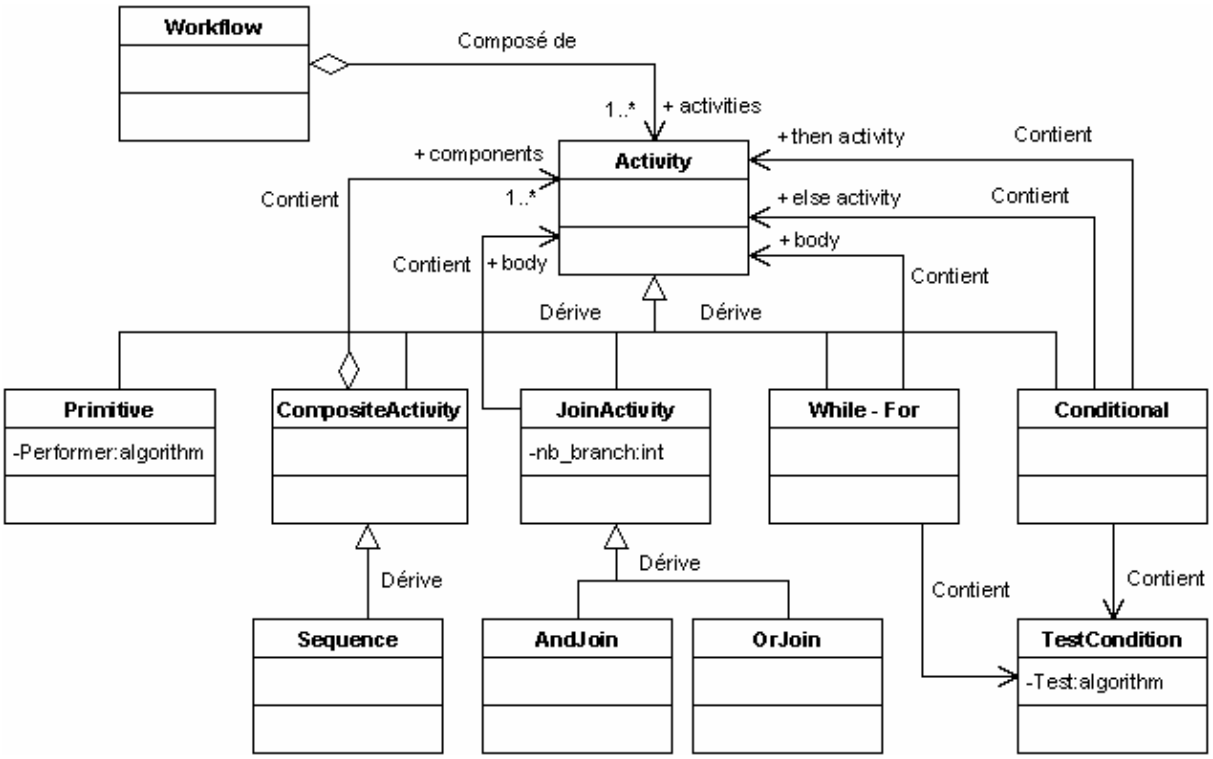


Figure 2.42 : Méta modèle de workflow – Modèle dynamique

Le méta modèle, correspondant à la modélisation dynamique d'un logiciel, est donc composé d'une méta entité "Workflow" contenant des "Activity". Ces activités peuvent être de différents sous-types, "Primitive", "While", "For", "Composite", "Conditional", "Join" et "Fork" (voir figure 2.42). Toutes ces différentes activités permettent de pouvoir créer tous les types d'enchaînements imaginables :

- ✓ **Primitive :** Cette activité permet de créer un lien entre le workflow (modèle dynamique) et une commande (modèle statique). Cette primitive est l'élément de base d'un workflow, elle correspond à l'exécution d'une commande.
- ✓ **While et For :** Ces activités permettent comme dans tout langage de programmation de modéliser des boucles. Elles sont donc composées d'une méta entité correspondant à un test et d'une activité correspondant au corps de la boucle.
- ✓ **Composite :** Cette activité permet simplement d'enchaîner plusieurs activités.
- ✓ **Conditional :** L'activité "Conditional" correspond à la séquence If, Then, Else de tout langage de programmation. Une méta entité test permet d'orienter le workflow vers l'activité "Then" ou "Else".
- ✓ **Join et Fork :** Ces deux dernières activités permettent de pouvoir décrire des branches parallèles. Le "Fork" permet de diviser le workflow en plusieurs branches et le "Join" permet de les regrouper.

De plus, un Workflow est un sous type de commande. Cela permet de réaliser un lien entre le workflow et l'interface homme machine et par conséquent d'intégrer des workflow dans des menus ou des barres d'icônes.

Cependant, le modèle dynamique ici présenté n'est encore que dans sa phase préliminaire. Cette première ébauche nous permet déjà de réaliser différents enchaînements. Néanmoins, des tests complexes nous permettant d'éprouver ainsi que d'améliorer notre démarche restent nécessaires. Seule une utilisation poussée de nos modèles permettra de les valider entièrement.

De plus, le modèle dynamique nécessite comme le modèle statique un programme permettant d'interpréter ce modèle et d'exécuter les workflow décrits. Une première version de cet interpréteur de modèle dynamique fonctionne déjà. Cependant, il n'a pas encore été validé par des exemples concrets de grande taille.

2.5 Génération du code

Notre démarche est basée sur une réalisation en trois grandes étapes : la modélisation, la génération du code et enfin l'interprétation du modèle complété par le code métier. Après avoir décrit en détail la modélisation de nos applications, nous allons maintenant expliquer comment et pourquoi ces modèles peuvent être convertis en code informatique.

Strictement parlant, le modèle que nous avons proposé peut devenir exécutable et la génération de code à partir du modèle vers un langage de programmation n'est pas utile. La machine virtuelle qui interprète le modèle et le transforme en application n'a en effet pas besoin de cette traduction. Cette souplesse est très pratique dans les premières itérations de la conception du logiciel métier, car elle permet une très grande réactivité et une réduction très sensible du temps de cycle essai / erreur / correction. Par contre, lorsque le modèle est stable et que l'on souhaite implanter concrètement les comportements métiers, le codage devient indispensable.

Deux solutions sont alors envisageables : le code correspondant aux concepts métiers n'est pas présent, il faut alors utiliser une approche de programmation qui manipule les objets sous une forme "relationnelle" (pas de typage, identification des instances par numéro, …). Seconde possibilité, le code des concepts modélisés est généré, il est alors possible de profiter de toute la puissance de l'approche objet pour programmer les comportements métiers. C'est dans ce but que nous avons développé un générateur automatique de code, s'appuyant sur la représentation AML des concepts métiers. Ce générateur permet de créer, pour chaque entité,

une traduction en langage Java contenant toutes les informations inscrites dans le modèle de données :

- ✓ **Le nom :** Correspondant au nom de l'entité de notre modèle.
- ✓ **Le paquetage :** Correspondant au paquetage définit dans le modèle.
- ✓ **Le stéréotype :** L'entité est-elle abstraite ou concrète ?
- ✓ **Les interfaces :** Correspondant aux interfaces qu'implante l'entité.
- ✓ **Les super types :** les super classes de la classe
- ✓ **Les champs :** La déclaration des champs de l'entité ainsi que de leurs méthodes de création, de modification, de test et suppression associées.

Chaque fichier Java ainsi généré est constitué d'une partie correspondant au modèle (constructeurs et assesseurs) mais aussi d'une autre partie contenant les relations et méthodes spécifiques ajoutées par les développeurs. A chaque reconstruction du code, la partie spécifique est récupérée et fusionnée dans le nouveau code source créé.

Au final, le code généré a deux fonctions : permettre l'introduction des comportements métiers dans une approche orientée objet d'une part ; apporter la persistance en mémoire des objets et de leurs relations. Notons sur ce dernier point que si le code n'est pas construit, la persistance mémoire est assurée par une base de données de type relationnel.

2.6 Exécution du modèle

Après avoir conçu les modèles des logiciels adaptables que nous souhaitons créer, il faut maintenant concevoir un programme, que nous appellerons machine virtuelle, capable d'interpréter les modèles statiques, dynamiques et de présentation d'une application afin de d'exécuter physiquement le logiciel. Notre machine virtuelle est basée sur une architecture réutilisable de composants logiciels, nommée FluxCore. Cette plate-forme d'exécution est totalement écrite au niveau méta modèle et est donc indépendante d'un modèle particulier (un modèle étant en fait une application métier) (voir figure 2.43).

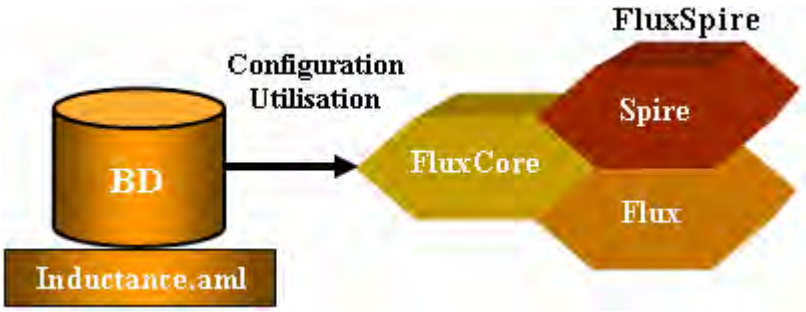


Figure 2.43 : Démarche de conception : Configuration et utilisation

A grands traits, la machine virtuelle lit le modèle du logiciel adaptable, initialise l'ensemble de ses composants avec ce modèle, construit un bureau et une présentation conforme aux souhaits exprimés dans le modèle, puis se met en attente d'évènements utilisateurs. Elle gère des actions réflexes de type création, modification et suppression d'instances et fournit une persistance fichier du projet construit par l'utilisateur final.

La machine virtuelle FluxCore est composée de différents modules reliés à un bus de données. Ces modules seront expliqués en détail afin de bien comprendre leurs rôles dans l'exécution de l'application. Ensuite, nous exposerons la démarche d'exécution de nos logiciels adaptables.

2.6.1 Structure de la machine virtuelle

La machine virtuelle dynamique, appelée FluxCore implémente toutes les fonctionnalités généralement disponibles dans la plupart des logiciels de simulation. Tous ces différents services sont structurés de la manière décrite dans la figure suivante.

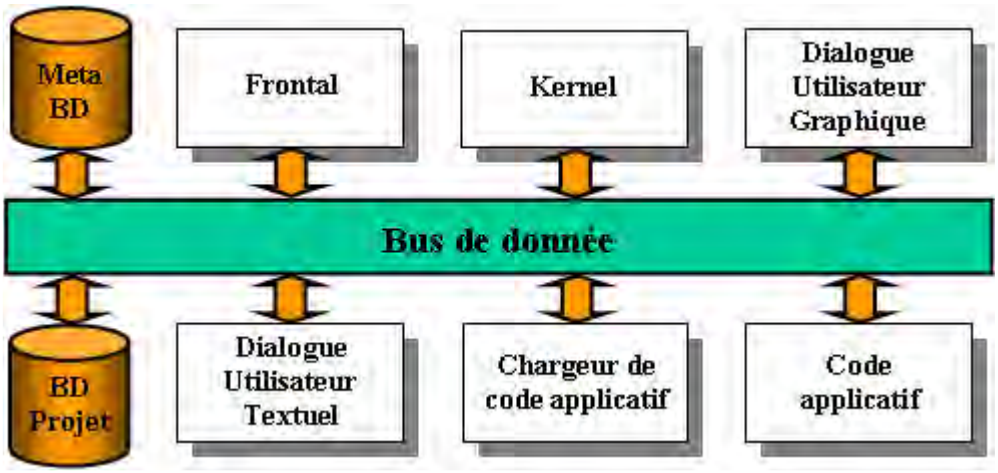


Figure 2.44 : Structure de la machine virtuelle dynamique

Cette structuration fait donc apparaître la notion de modules connectés à un bus de données. Elle met aussi en évidence les deux bases de données de l'application, la base de données "Méta" correspondant au modèle saisi par le concepteur d'application, et la base de données "Projet" correspondant aux données de l'utilisateur du logiciel.

Cette machine virtuelle est totalement configurée par le modèle qui décrit les données, les commandes, l'application ainsi que son interface homme machine. La machine virtuelle FluxCore n'est programmée qu'une seule et unique fois mais elle permet d'exécuter tous les logiciels modélisés. Tous les composants de la machine virtuelle ne s'appuient que sur des

méta classes. Seul le module contenant le code applicatif dépend du modèle lui-même. Ce module change suivant l'application qui est exécutée.

Cette structuration en modules permet aussi de pouvoir envisager un nouveau type d'utilisation de nos logiciels. Les transferts de données entre modules se réalisant selon un protocole compatible avec les appels à distance (Remote Method Invocation [RMI]), il est donc possible d'exécuter les applications modélisées en mode client serveur. L'utilisateur, "client", n'ayant sur son ordinateur que le module graphique par exemple pendant que le "serveur" gère les autres modules [LACOMBE-03].

Après cette brève explication de la machine virtuelle FluxCore, nous allons maintenant décrire plus en détail les différents modules les uns après les autres.

2.6.1.1 Le module Frontal

Le module que nous appelons Frontal est le module qui gère toute la partie graphique du logiciel exécuté. Le Frontal assure donc comme fonctions l'affichage de toutes les informations à l'écran, ainsi que le lien avec l'utilisateur en interceptant les demandes d'actions via les évènements clavier ou souris (voir figure 2.45).

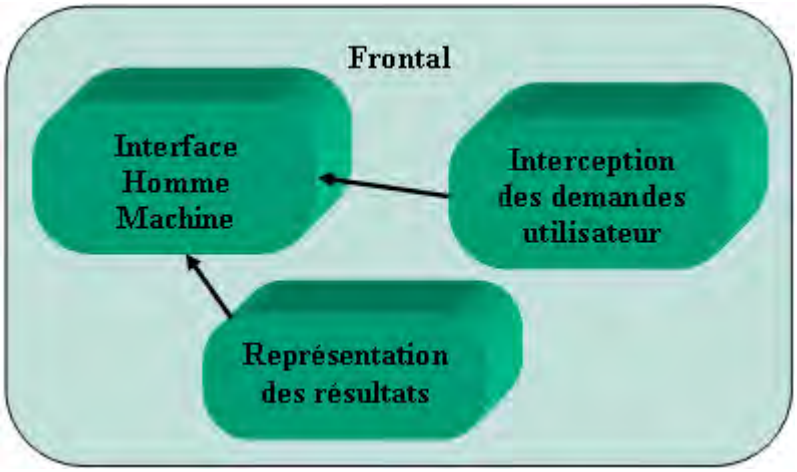


Figure 2.45 : Structure du module Frontal

Ainsi, le module Frontal permet de réaliser les fonctions suivantes :

- L'affichage et le rafraîchissement des différents éléments composant l'interface homme machine :
  - ✓ La fenêtre de l'application dont le titre reprend le nom de l'application et du problème en cours d'exécution.
  - ✓ Les contextes de l'application.
  - ✓ Les menus, barres d'icônes, arbres de représentations et fenêtres graphiques.

- ✓ Les consoles du langage de commande et d'affichage des informations.
- ✓ Différentes informations comme un vumètre de la mémoire utilisé par le graphique ou l'action en cours d'exécution.

- L'affichage des données nécessaires aux différents éléments du graphique. Ces éléments sont envoyés au module Frontal par les autres modules grâce au bus de données. Ainsi les boîtes de dialogue, envoyées par le module "Dialogue utilisateur graphiques", ainsi que les données graphiques, lues dans la base de données "Projet" grâce au module Kernel, sont affichées grâce au module Frontal.
- L'interception des demandes d'exécution d'action. Ces demandes sont interceptées par le module Frontal et sont ensuite envoyées au module Kernel afin d'être exécutées.

2.6.1.2 Le module Kernel

Ce module correspond au noyau même de la machine virtuelle. Le Kernel est donc le cœur du logiciel, il implémente un grand nombre de fonctionnalités (voir figure 2.46).

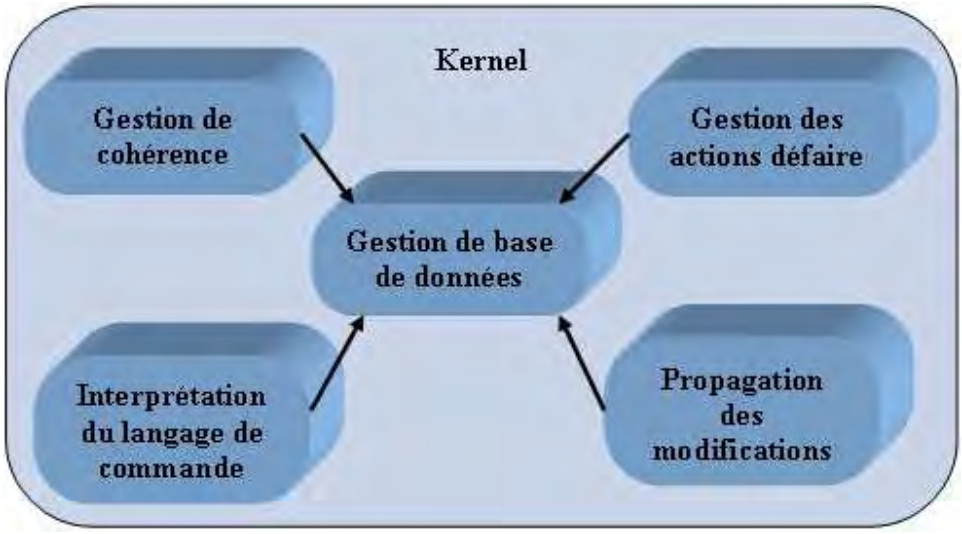


Figure 2.46 : Structure du module Kernel

- La lecture et la gestion des bases de données "Projet" et "Méta". Cette fonction permet de gérer la persistance des données de l'utilisateur en enregistrant celles-ci dans la base de données "Projet". Quant à la base de données méta, elle gère toute l'information saisie par le concepteur du logiciel. Cette information va servir par exemple à réaliser la gestion de cohérence des données du projet, à gérer le frontal, à construire le dialogue graphique ou textuel.
- La gestion de la cohérence des données. Suivant les informations issues de la base de données méta, des vérifications de cohérence sont effectuées lors de toutes les saisies

de l'utilisateur : tous les champs déclarés obligatoires sont-ils remplis ? Les cardinalités des associations sont-elles respectées ? Les identificateurs déclarés uniques sont-ils réellement uniques ? Les compositions d'objets sont-elles gérées correctement ? La destruction d'un objet est-elle autorisée ? Toute cette cohérence que l'on trouve classiquement dans les gestionnaires de base de données est ici présente.

- La gestion de la propagation des modifications. Ainsi, la modification d'un objet peut nécessiter la réévaluation des objets dont le type est relié à l'objet modifié. Cette réévaluation est elle aussi décrite dans le modèle de données du logiciel (voir 2.2.3.6 Rétro propagation).
- La gestion des actions "défaire". Comme dans tout logiciel, il est possible de défaire les dernières actions réalisées. Cette gestion permet de rétablir l'état de l'application avant l'action que l'on vient d'annuler.
- L'interprétation du langage de commande. Grâce à un interpréteur Python intégré, le module Kernel peut interpréter et exécuter tous les fichiers de langage de commande. L'interpréteur sert aussi pour le dialogue graphique. En effet, toute boîte de dialogue est convertie en langage de commande pour être interprétée et exécutée.
- La gestion et l'appel aux algorithmes métiers. Ainsi, les algorithmes de création de la géométrie d'une micro inductance, par exemple, sont appelés par le module Kernel, les données sont stockées dans la base de données "Projet" puis envoyées au module Frontal pour être affichées.

2.6.1.3 Le module Interface Utilisateur

Ce module a pour rôle de réaliser le dialogue avec l'utilisateur. S'appuyant sur les informations ajoutées aux entités, champs, commandes et arguments, il produit des boîtes de dialogue structurées ou interprète la sémantique d'une chaîne de langage de commande (voir figure 2.47).

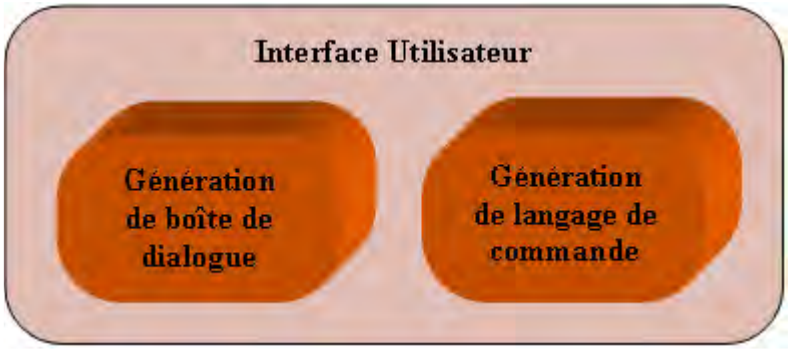


Figure 2.47 : Structure du module Interface Utilisateur

Les deux modes de fonctionnement complémentaires sont :

- **Graphical User Interface [GUI] :** Cette notion représente la solution graphique de l'interface utilisateur. Elle est basée sur un dialogue par boîte graphique construite grâce aux données du projet et aux méta données du modèle de l'application
- **Textual User Interface [TUI] :** Cette notion permet un dialogue textuel grâce notamment à une console permettant d'exécuter directement des instructions de langage de commande. Remarquons que toute boîte graphique est convertie en texte afin d'assurer un espionnage des commandes et un point d'interprétation unique sous la forme texte.

L'interface graphique est évidemment plus immédiate d'accès. Cependant, le mode langage de commande est particulièrement efficace pour l'automatisation d'enchaînements de commandes paramétrées.

2.6.1.4 Le module Chargeur de code applicatif

Ce module réalise la fonction d'interfaçage entre les modèles de données de nos applications et les autres modules de la machine virtuelle. Ce module permet de réaliser le lien entre la base de données projet et les classes construites par le générateur de code. Il s'appuie pour cela sur les capacités de chargement dynamique, de création d'instance à partir d'un nom de classe et d'invocation de méthodes par son nom fournies par Java (module reflect).

2.6.1.5 Le module Code applicatif

Le module Code Applicatif est utilisé pour stocker tous les algorithmes spécifiques du métier. Par exemple, pour notre logiciel de conception de micro inductance, ces algorithmes sont la construction de la géométrie en fonction des paramètres de l'inductance ou l'affectation automatique des paramètres physiques aux régions de l'inductance. Ce module est sollicité par le module Kernel lors de la demande d'exécution de ces commandes spécifiques.

Le module Code Applicatif peut aussi hériter et surcharger les modules précédemment exposés afin de modifier le comportement général programmé dans la machine virtuelle FluxCore. Cette démarche d'héritage et de surcharge est détaillée dans le paragraphe 2.6.3.

2.6.2 Démarche d'exécution des logiciels métiers

Le meilleur moyen pour comprendre comment fonctionnent les logiciels conçus par modélisation est encore de présenter leur processus d'exécution. Le fonctionnement de nos logiciels est donc le suivant :

Au lancement de l'application, le premier élément à être activé est la machine virtuelle. Cette dernière récupère, dans le modèle de données de l'application, les informations nécessaires à la configuration de l'ensemble des modules du logiciel. Ainsi, le module Frontal, gérant l'interfaçage avec l'utilisateur, peut commencer à afficher les éléments graphiques correspondant au modèle de l'interface homme machine décrit par le concepteur (menus, arbres, icônes, fenêtres graphiques). La configuration des autres modules correspond à leur initialisation dans l'attente de sollicitations.

La figure suivante permet de visualiser le déroulement de l'exécution d'une commande issue d'une requête utilisateur.

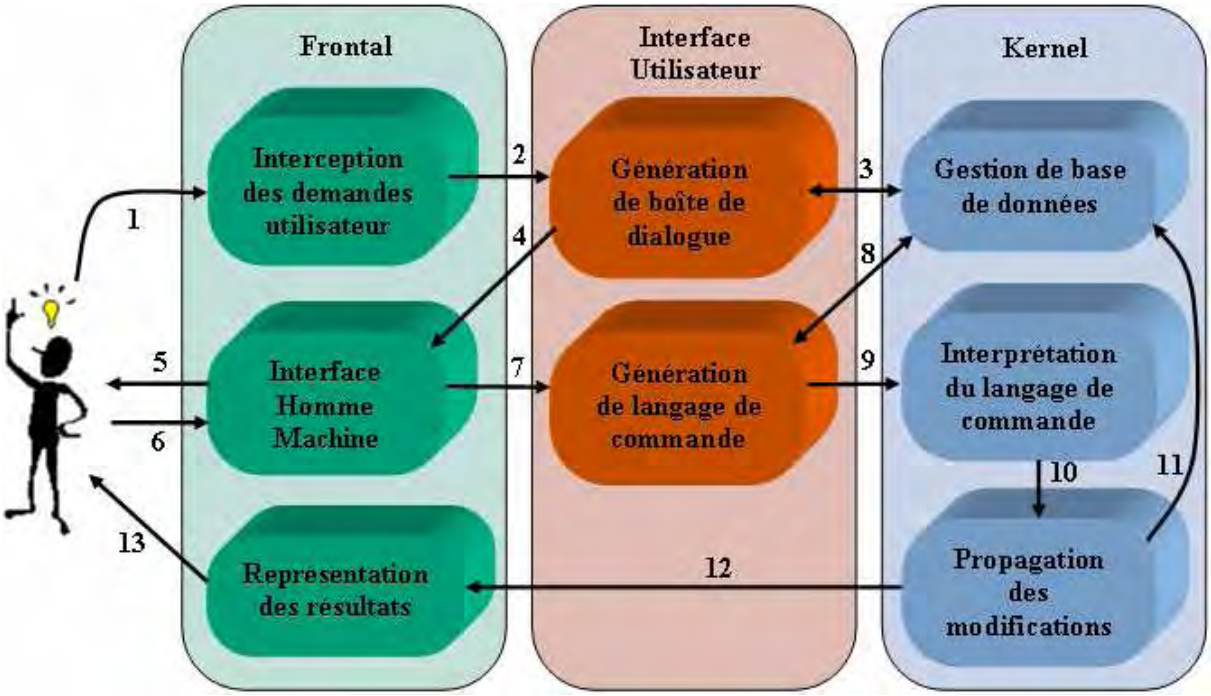


Figure 2.48 : Démarche d'exécution d'une commande

Lorsque l'utilisateur lance une requête, celle-ci est interceptée par le Frontal grâce à l'interface utilisateur. Cette demande de l'utilisateur est ensuite convertie en boîte de dialogue correspondante grâce au module Graphical User Interface [GUI]. Puis, après la saisie et la validation des données projet par l'utilisateur, cette boîte est interprétée par le module Textual User Interface [TUI]. Ce dernier permet de convertir les données d'une boîte de dialogue en

commandes en langage Python [Python]. Ces commandes peuvent ensuite être envoyées au module Kernel afin d'y être interprétées.

Les différents modules nécessaires à la réalisation de la requête utilisateur sont appelés pour exécuter les différentes tâches correspondantes à cette demande. Si les commandes sont des actions spécifiques à un métier, l'appel au module Code Applicatif du métier correspondant est obligatoire et géré par le module Kernel.

Ensuite, tous les objets créés, modifiés ou supprimés par le gestionnaire de propagation des modifications sont stockés dans la base de données projet et le résultat, s'il doit être affiché à l'écran, sera envoyé à l'interface homme machine du module Frontal. Toutes les données échangées pendant l'exécution de la commande sont bien entendues véhiculées grâce au bus de données reliant tous les modules.

2.6.3 Héritage et surcharge

Comme nous avons pu le voir dans le paragraphe précédent, notre machine virtuelle est structurée en modules. Dans l'exemple du logiciel FluxSpire, la machine virtuelle FluxCore est complétée par deux modules applicatifs : Le module Flux permettant d'obtenir les logiciels Flux, ainsi que le module Spire ajoutant à cet ensemble les spécificités du métier de concepteur de micro inductance. L'application ainsi obtenue correspond au logiciel métier FluxSpire (voir figure2.49).

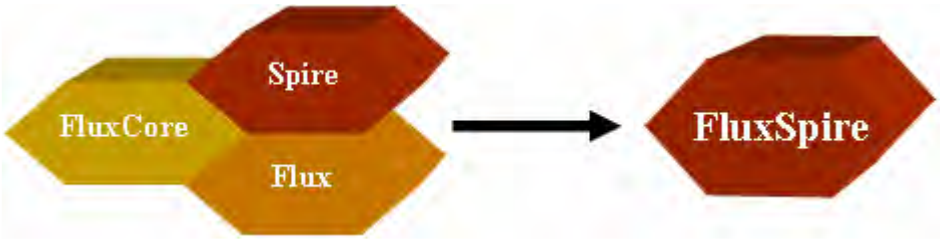


Figure 2.49 : Constitution du logiciel métier FluxSpire

Il arrive cependant que les modules ou les composants présents dans FluxCore ne satisfassent pas totalement les besoins de l'utilisateur final. Pour réaliser des assemblages sur mesure, l'architecture de notre machine virtuelle est construite de manière à ce que chacun de ces modules ou de ces composants soit surchargeable ou débranchable. La surcharge des modules est assurée par les mécanismes d'héritage et de surcharge issus de la programmation objet, complétés par des modèles de conception de type "Factory".

Chaque logiciel adaptable peut donc, s'il le souhaite, modifier le comportement de la machine virtuelle FluxCore. Cette méthode d'héritage et de surcharge peut être mise en œuvre

de manière parallèle ou bien les héritages peuvent s'imbriquer les uns dans les autres. Ainsi, un module Spire peut hériter d'un module de la surcharge Flux qui lui même hérite d'un module de la machine virtuelle FluxCore.

Rappelons simplement que cette surcharge n'est bien sûr pas obligatoire, mais permet de personnaliser un peu plus la présentation des applications.

2.7 Conclusion

Dans ce chapitre, nous avons exposé toutes les notions nécessaires à la compréhension de notre démarche de conception de logiciels adaptables. Notre modélisation d'une application est basée sur des modèles statique, dynamique et de présentation correspondant aux différents éléments ainsi qu'aux comportements de nos logiciels. Nous avons proposé un contenu pour ces différents modèles.

La modélisation statique comprend, tout d'abord, le modèle des entités et des commandes. Néanmoins, pour permettre un dialogue efficace avec les utilisateurs nous avons décidé de compléter ce modèle en lui ajoutant des informations graphiques de type interface utilisateur. Pour cela, des données décrivant l'interface des différentes entités, commandes ainsi que leurs champs et arguments associés ont été rajoutées. De plus, des actions réflexes, permettant de réaliser par exemple la création, la modification et la suppression de tout type de données sont aussi venues enrichir le modèle existant. Une application est constituée de données, mais elle est aussi composée d'une interface homme machine. Le modèle correspondant à cette interface nous permet donc de décrire les menus, les barres d'icônes, les arbres de représentation et les fenêtres graphiques de nos logiciels. Cependant, ce modèle statique ne serait pas complet sans la modélisation de l'application elle-même. Chaque logiciel est composé de divers cas d'utilisation eux-mêmes composés de différents contextes. Après avoir défini avec précision le modèle statique de nos logiciels, nous avons proposé des éléments de modélisation dynamique de nos applications. Ce modèle permet de modéliser les différents comportements de nos logiciels. Ainsi, toutes les macros actions constituant un des avantages des logiciels adaptables peuvent être modélisées grâce à ce modèle dynamique.

Nous avons présenté un générateur automatique de code pour introduire les algorithmes métiers. Ce dernier permet d'écrire des fichiers en langage de programmation objet en fonction des objets du modèle de données.

Ensuite, nous avons présenté la machine virtuelle FluxCore, capable de transformer un modèle en une application exécutable. Cette machine virtuelle est découpée en modules

permettant une structuration performante pour un gain en temps de développement et une compatibilité inter modules optimale. Le dernier point abordé dans ce chapitre est le concept d'héritage et de surcharge. Ce mécanisme, combiné avec la structuration en module de la machine virtuelle FluxCore, sert à minimiser les coûts de création ainsi que de maintenance de nos logiciels en permettant une réutilisation maximale des fonctionnalités intégrées.

Toute cette démarche de conception par modélisation permet d'obtenir rapidement une application totalement finalisée avec un nombre important de fonctionnalités intégrées dynamiquement. Ainsi, la persistance des données, le dialogue graphique et textuel, la gestion du langage de commande, la configuration générale du bureau sont des comportements génériques entièrement pilotés par le modèle de l'application. Les fonctionnalités intégrées au noyau telle que la gestion de cohérence, la gestion des actions défaire ou la propagation des modifications sont donc réutilisables par toutes nos applications.

Adopter une telle stratégie nous apporte un grand nombre d'avantages. La modification du modèle de données est beaucoup plus rapide qu'avec la démarche standard de Programmation Orientée Objet et elle peut être faite à coût quasi nul (excepté l'implémentation effective des algorithmes). De plus, beaucoup de services sont incorporés une fois pour toutes. Enfin, la maintenance du code est rigoureusement réduite tant que la machine virtuelle reste le noyau commun de toutes nos applications. Ainsi, n'importe quelle amélioration ou correction de bug dans la machine virtuelle est immédiatement disponible dans toutes les applications.

Cependant, certains travaux restent encore à la charge du concepteur. En effet, la saisie du modèle de données mais aussi des algorithmes correspondant aux comportements métiers de l'application demeurent inévitables. Autant le codage des algorithmes ne peut pas être simplifié, autant la construction du modèle de données peut, quant à elle, être grandement facilitée. En effet, ce modèle, sauvegardé dans un fichier texte, peut être construit à l'aide d'un logiciel spécialisé. Ce logiciel muni d'outils facilitant la saisie et offrant une vision d'ensemble du modèle de données permet d'obtenir beaucoup plus facilement des modèles d'application complexes. Sa présentation fait l'objet du chapitre suivant.

3 Conception d’un descripteur de logiciel

3.1 Introduction

Notre démarche de conception de logiciel par modélisation nécessite trois éléments clefs. Le premier est la définition précise d’un langage de modélisation. Ensuite, pour pouvoir exécuter un logiciel métier, il faut concevoir un programme informatique appelé machine virtuelle. Ce programme configure dynamiquement l’application grâce au modèle du logiciel réalisé par le concepteur. Ces deux premières étapes étant réalisées, les concepteurs de logiciels métiers doivent maintenant saisir les modèles définissant leurs applications. Cette dernière phase est relativement délicate si on ne dispose pas d’outils dédiés. En effet, la complexité toujours croissante des logiciels ainsi que le nombre important d’éléments que nous avons décidé d’intégrer dans nos modèles de logiciel rendent difficile la description d’une application avec un simple traitement de texte.

C’est pourquoi nous avons choisi de concevoir un logiciel spécifique permettant d’aider les concepteurs à modéliser leurs applications métiers. Ce logiciel, appelé FluxBuilder, permet de simplifier la description d’un modèle de logiciel en proposant une vision graphique du modèle que l’on crée (voir figure 3.1).

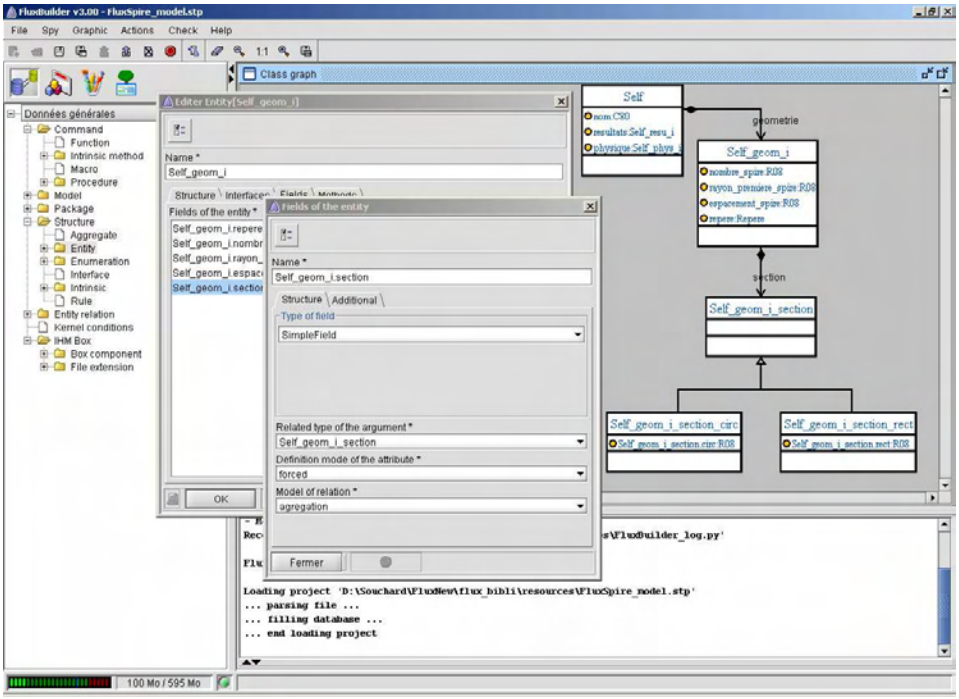


Figure 3.1 : Logiciel FluxBuilder : Description de modèle

Le descripteur d’application permet aussi de faciliter la communication entre les différentes personnes participant à la modélisation d’un logiciel. En effet, pour pouvoir bien comprendre les besoins des utilisateurs et les contraintes de développement des concepteurs, il faut posséder un système de communication clair et basé sur des normes et des représentations reconnues par tous les acteurs prenant part à la démarche de conception. Cette collaboration est rendue possible dans le logiciel FluxBuilder par la représentation de notre modèle mais aussi par un dialogue graphique clair facilitant l’échange entre les concepteurs collaborant pour la création de modèles.

Le logiciel FluxBuilder propose une vision structurée du modèle de données. La capture d’écran de la figure 3.1 nous permet de noter la présence d’un diagramme de classes au format Unified Modeling Language [UML] ainsi qu’un arbre montrant tous les types d'entité, de commandes et d’autres types déjà créés. De plus, au premier plan, une boîte de dialogue servant à la saisie ou à la modification d’une entité permet d’avoir une vision détaillée des objets que nous créons pour la conception de nos modèles de logiciel. Ces boîtes de dialogue aident à la création de tous les types d’objets intégrés dans le méta modèle de nos logiciels. Ces méta données sont stockées dans une base de données correspondant au modèle de l’application métier. Ce modèle peut être rechargé pour modification et est interprété par la machine virtuelle FluxCore.

Ce chapitre commence par la présentation du descripteur de modèle d’applications. Ces fonctions sont détaillées afin d’exposer tous les outils facilitant la saisie d’un modèle d’application par son concepteur. Ensuite, pour mieux comprendre l’utilité du logiciel FluxBuilder, nous exposerons le principe de notre plate-forme de conception appelé FluxFactory. Et pour finir, l’architecture des logiciels métiers nous permet d’introduire une dernière notion, assez surprenante, l’auto conception du logiciel FluxBuilder.

3.2 Le logiciel FluxBuilder

FluxBuilder permet la création et la modification du modèle complet d’une application métier. Cette démarche (voir figure 3.2) doit être rendue la plus simple possible car le nombre de logiciels métiers peut être important. De plus, la conception de ces modèles doit être réalisée par des experts du métier certes, mais pas forcément par des experts en modélisation et en programmation.

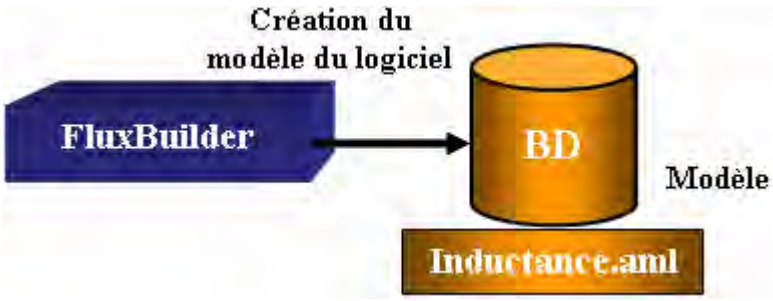


Figure 3.2 : Démarche de conception du modèle du logiciel FluxSpire

Le logiciel FluxBuilder réalise les fonctions suivantes :

- **Modéliser la structure de données :** La fonction principale de FluxBuilder reste la description du modèle de données d'une application. Pour cela, le logiciel FluxBuilder s'appuie sur une interface homme machine et des outils dédiés destinés à faciliter la saisie des objets composant l'application.
- **Introduire les informations de type interface utilisateur :** Comme nous l'avons vu dans le chapitre précédent, le modèle de données d'une application est enrichi d'informations de type interface utilisateur. Le logiciel FluxBuilder gère ces informations. La gestion de ces informations est regroupée avec la gestion des objets de l'application, ainsi leur saisie est aussi facile que celle des autres données du logiciel.
- **Introduire le modèle graphique d'un logiciel :** Une application est composée d'une interface homme machine. Afin de rendre le plus dynamique possible notre démarche de conception nous avons décidé de modéliser cette interface. Le logiciel FluxBuilder permet de modéliser tous les éléments de l'interface homme machine.
- **Faciliter la communication :** Le logiciel FluxBuilder permet aux différents concepteurs prenant part à la réalisation du modèle de l'application métier de pouvoir communiquer facilement et clairement entre eux. Ils ont besoin d'outils de communication communs à leur spécialité. Le développement du logiciel FluxBuilder a aussi été fait dans cet objectif en proposant une représentation du modèle dans un format reconnu de tous, l'Unified Modeling Language [UML].
- **Générer le code informatique à partir du modèle de données :** Le logiciel FluxBuilder intègre un algorithme de génération automatique de code en langage Java. Ainsi, le modèle de données d'une application peut être converti en fichiers informatiques, complétés et utilisés pour l'exécution du logiciel.

- **Gérer l'application :** Le Logiciel FluxBuilder permet enfin de modéliser l'application elle-même. Associé à tous les autres modèles, cet ensemble permet de décrire totalement le logiciel.

Pour réaliser toutes ces fonctions, le logiciel FluxBuilder s'appuie sur de nombreux outils intuitifs simplifiant la compréhension et donc la communication autour du modèle de données de l'application. Nous nous sommes appuyés sur une interface homme machine conviviale et sur des éléments graphiques simplifiant la saisie et la communication du modèle. Ces composants permettent d'obtenir une vision d'ensemble des modèles et de bien comprendre les différents liens qui existent entre les objets que nous modélisons. Le logiciel FluxBuilder propose quatre types d'outils dédiés à la présentation des données :

- **Outils Graphique :** La représentation sous forme de diagramme Unified Modeling Language [UML] propose un affichage graphique des données de l'application. Cette présentation par diagramme permet d'obtenir une vision structurée des données de l'application décrite.
- **Dialogue de type graphique :** Les données du modèle peuvent être présentées grâce à des boîtes de dialogue. Chaque boîte permet de présenter un objet du modèle de données. Ces boîtes facilitent la saisie et la modification des données du modèle.
- **Dialogue de type textuel :** Ce type de représentation est basé sur le langage de commande que nous avons mis en place. Il permet de visualiser les données du modèle sous forme purement textuelle grâce à une console en langage Python.
- **Outils de synthèse :** Afin de simplifier encore l'utilisation du descripteur FluxBuilder, nous avons développé une nouvelle génération d'outils basée sur un composant graphique performant. Ce nouveau composant, constitué d'un arbre de représentation intégré dans un tableau nous permet de synthétiser à l'extrême la représentation des données. En effet, nous pouvons désormais représenter à la fois les liens qui relient les différents objets d'une application grâce à la représentation en arbre, mais nous permettons dans un même composant la modification des données composant ces objets. De plus, contrairement aux boîtes de dialogue, ces outils de synthèse permettent de visualiser et de modifier plusieurs objets en même temps.

Nous allons maintenant présenter le fonctionnement des boîtes de dialogue et de quelques outils de synthèse simplifiant la tâche du concepteur.

3.2.1 Boîte de dialogue

La boîte de dialogue est sans aucun doute le premier outil pour simplifier la saisie du modèle d'une application. Elle est structurée en plusieurs parties correspondant aux différents éléments à saisir mais aussi en onglets permettant de regrouper les informations du même type dans chaque onglet (voir figures 3.3).

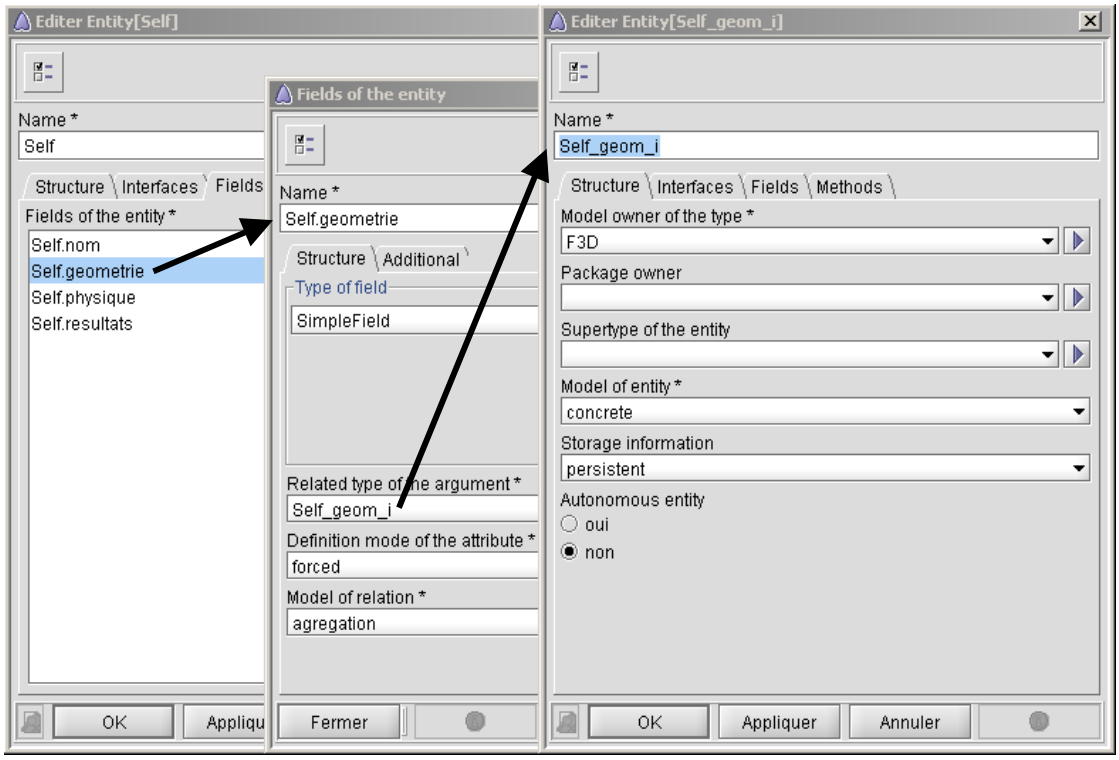


Figure 3.3 : Boîte de dialogue des entités "Self" et "Self_geom"

Cette figure montre les différents éléments à renseigner pour créer une entité. Pour l'application FluxSpire, nous souhaitons structurer l'entité "Self" et ses champs. La saisie des champs nécessite de renseigner l'ensemble des informations décrites dans le méta modèle de nos applications. Les champs contiennent un mode de définition pouvant être forcé, optionnel, ou dérivé, un modèle de relation pouvant être de type agrégation, association ou identification. Ces champs peuvent aussi être simples ou agrégés et doivent décrire le type de données auquel ils seront reliés. Dans notre exemple, le champ "Self.geometrie" est relié à l'entité "Self_geom" qui décrit la géométrie de la micro inductance. Ainsi, la saisie du modèle complet de cette entité "Self" permet d'obtenir, lorsque le logiciel FluxSpire sera exécuté, la boîte de création et de modification d'une micro inductance.

La structuration des entités grâce aux champs et aux autres éléments du méta modèle ne suffit cependant pas pour obtenir la boîte de dialogue correspondante. En effet, les

informations graphiques de type interface utilisateur sont obligatoires pour permettre l'affichage de la boîte de saisie. C'est la raison pour laquelle nous avons ajouté ces informations dans la description de nos objets. Leur saisie est réalisée dans le logiciel FluxBuilder grâce à l'icône situé en haut des boîtes de dialogue. Ce bouton permet d'ouvrir une autre boîte contenant ces informations supplémentaires (voir figure 3.4).

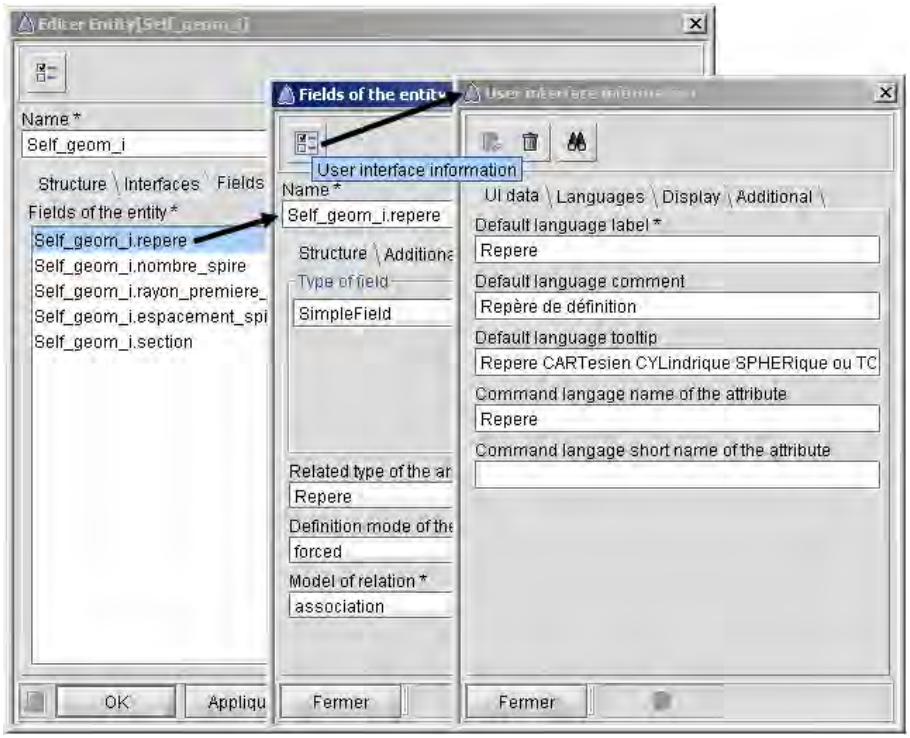


Figure 3.4 : Boîte des informations de type interface utilisateur

Toutes les informations utilisées par le générateur automatique de boîte de dialogue peuvent facilement être saisies par le concepteur. Ces informations sont présentées dans une boîte séparée afin de bien les dissocier des informations correspondant à la structure de données du logiciel. La figure suivante présente comment les informations de type interface utilisateur sont utilisées pour produire le dialogue graphique au niveau modèle (voir figure 3.5).

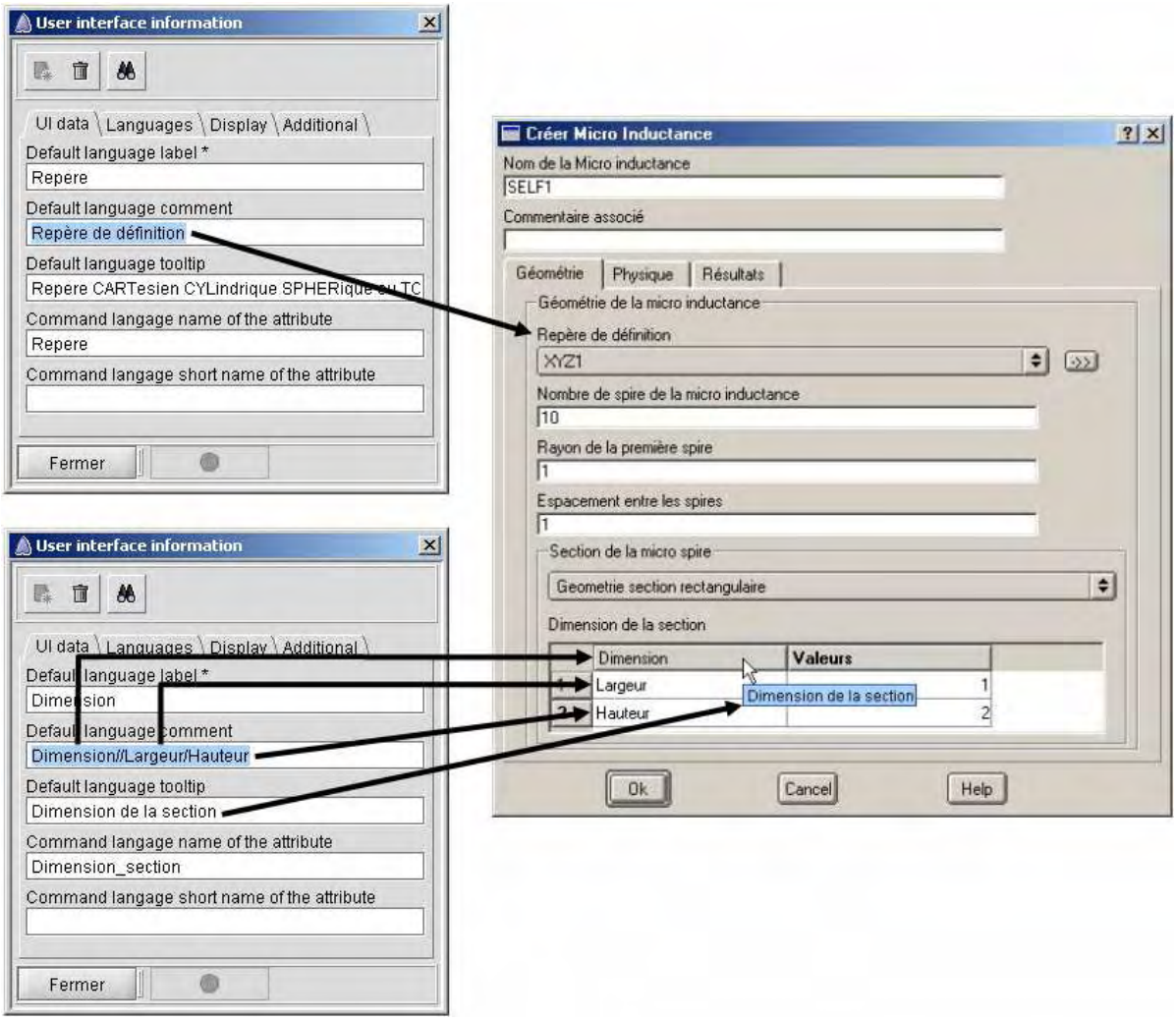


Figure 3.5 : Correspondance entre le modèle de données et la boîte de saisie correspondante

En conclusion, la boîte de dialogue est l'outil le plus utilisé par les concepteurs de modèles de logiciels métiers. Elle permet de saisir à la fois les entités, les commandes ainsi que leurs champs et arguments mais aussi tous les autres méta données définissant nos applications. Ces boîtes de dialogue permettent de décrire la structure de nos objets, l'interface homme machine du logiciel modélisé, les menus, barres d'outils et arbres de représentation, mais aussi l'application, ses cas d'utilisations, et ses contextes (voir figure 3.6).

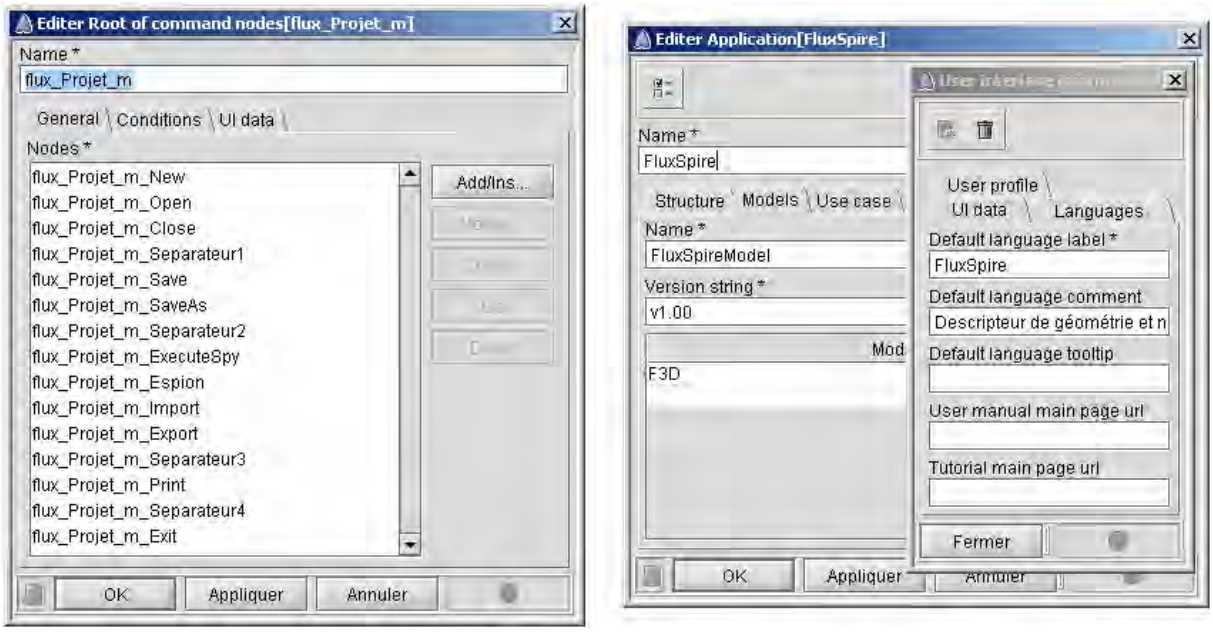


Figure 3.6 : Modèle du menu projet et de l’application FluxSpire

3.2.2 Outils de synthèse

Dans le but de rendre toujours plus simple l’utilisation du logiciel FluxBuilder, nous avons décidé de développer une série d’outil intégrant un nouveau type de composant graphique de synthèse. Cet élément est composé d’un tableau dont la première colonne est un arbre de représentation. L’utilisation jumelée d’un arbre et d’un tableau permet d’ajouter une dimension supplémentaire au tableau classique et d’obtenir alors la représentation d’un tableau en "trois dimensions".

Grâce à ce nouveau composant, il est possible de représenter beaucoup plus d’informations que dans un tableau classique avec, en plus, une structuration des objets plus évoluée. Les outils construits avec ce type de composants graphiques permettent de proposer aux concepteurs d’application métier une vision globale de leurs modèles. En effet, nous pouvons désormais à la fois représenter les liens qui relient les différents objets d’un modèle grâce à la représentation en arbre, et permettre dans ce même composant la modification des données composant ces objets (voir figure 3.7).

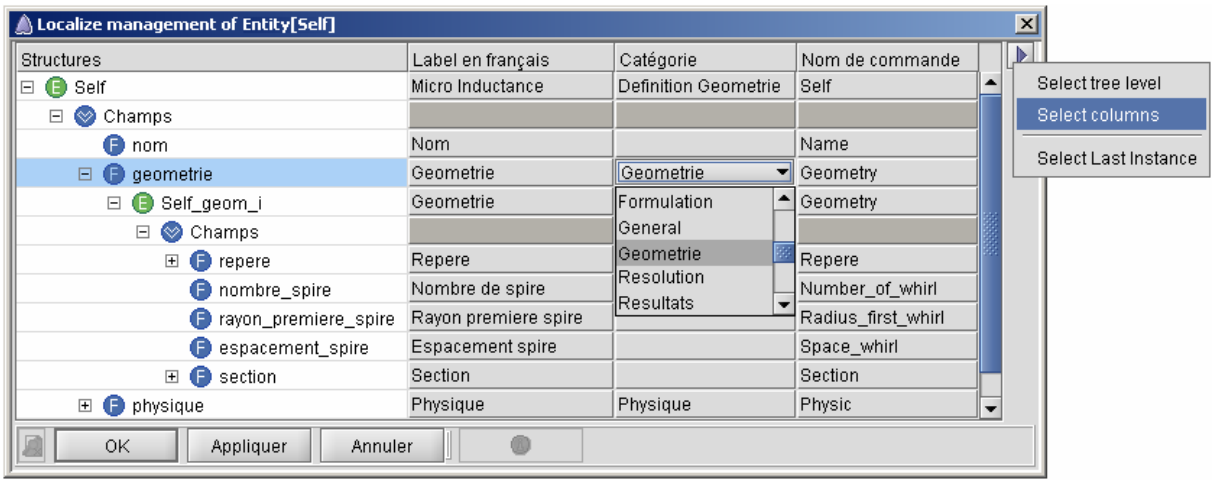


Figure 3.7 : Exemple d'outil basé sur le composant de synthèse

De plus, le composant de synthèse utilisé par ces nouveaux outils est totalement intégré dans notre démarche de conception. Ainsi, ce type de composant est décrit dans le logiciel FluxBuilder et il peut être utilisé pour créer des outils de synthèse dans tous les logiciels métiers.

Dans un premier temps nous allons nous intéresser à la partie représentation en arbre de ce composant. Cet arbre a pour but de présenter au concepteur une structure complète des objets qu'il souhaite éditer ou modifier. Pour cela, nous avons conçu un algorithme permettant de construire cet arbre en respectant la structure du méta modèle que nous avons défini dans le chapitre précédent (voir figure 3.8).

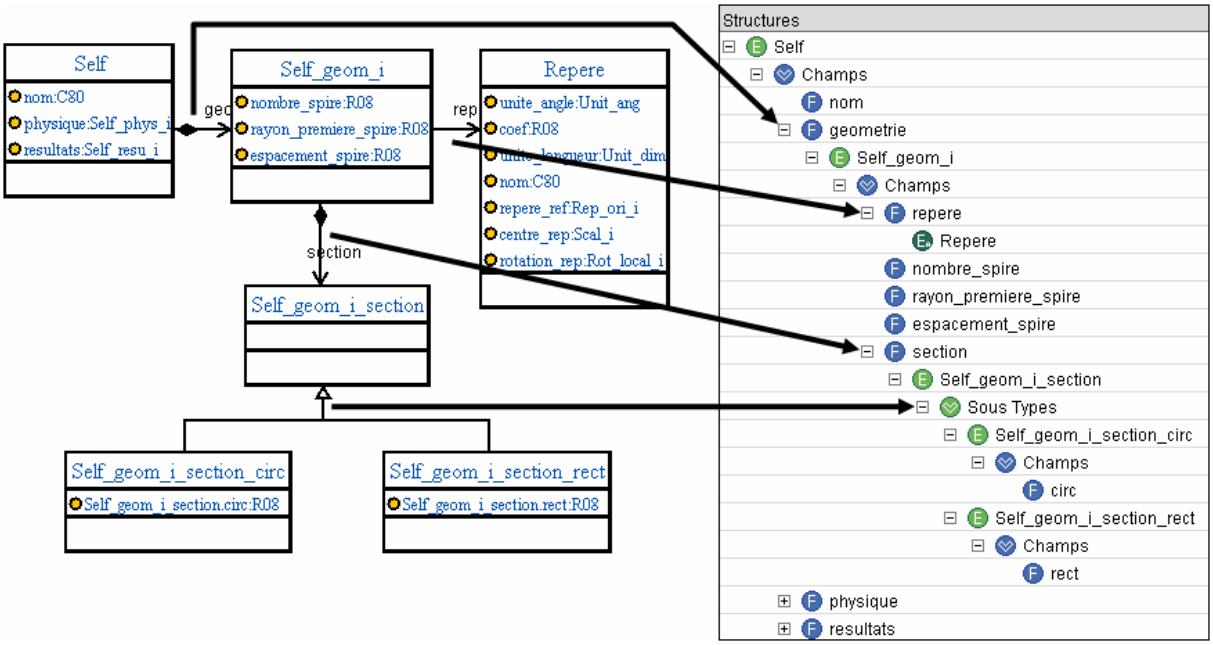


Figure 3.8 : Représentation UML et arborescente de l'entité Self

Ainsi, sur cet exemple, la représentation par arbre permet de visualiser tous les champs, méthodes et sous-types de l'entité Self et pour chacun des champs, la présentation est récursive et affiche les entités reliées aux champs et ainsi de suite. La représentation en arbre permet d'afficher seulement les informations souhaitées et de compacter les autres (comme les champs "Physique" et "Résultats" sur la figure 3.8). Cette représentation complète la représentation par boîte de dialogue qui, pour présenter l'arborescence de la figure 3.8, devrait afficher six boîtes.

Après avoir exposé les avantages de la représentation par arbre des données d'un modèle, nous allons maintenant présenter les outils que nous avons réalisés grâce à l'association de l'arbre et d'un tableau. Le premier est conçu pour modifier les informations User Interface de nos objets, les deux outils suivants sont eux développés dans le but de faciliter la gestion du masquage des objets créés par les concepteurs de logiciels métiers.

3.2.2.1 Management des informations de type interface utilisateur

L'outil de gestion de dialogue utilisateur permet de faciliter la saisie et la modification des informations de type interface utilisateur des entités, commandes ainsi que de leurs champs et arguments respectifs. En effet, cet outil permet d'afficher une boîte contenant un composant de synthèse présentant, sous la forme d'un arbre, la structure des objets à afficher et dans un tableau associé, l'ensemble des données de type interface utilisateur de ces mêmes objets, éventuellement dans toutes les langues modélisées (voir figure 3.9).

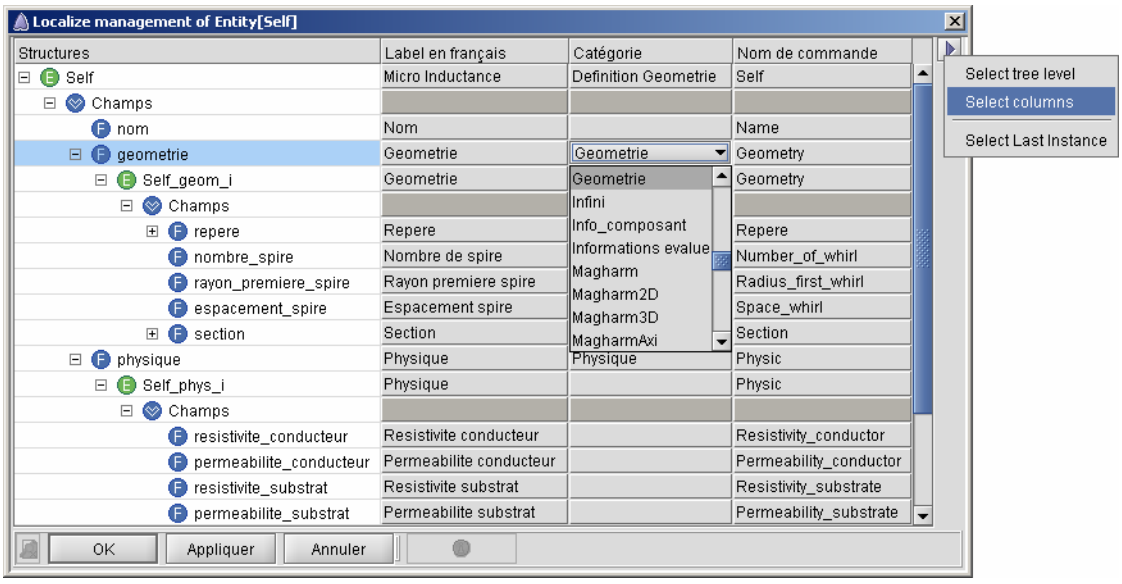


Figure 3.9 : Outil de gestion d'interface utilisateur

Sur cette figure, nous apercevons seulement trois colonnes de données mais, grâce au menu contextuel situé en haut à droite de la boîte, l'ensemble des douze colonnes représentant toutes les informations de type interface utilisateur décrites dans le modèle peuvent être affichées. Cet outil est très utile car il permet de visualiser l'ensemble des informations qui seront présentées dans la boîte de dialogue de création d'une micro inductance (voir figure 3.10).

On peut ainsi très rapidement trouver le label ou le commentaire du champ ou de l'entité reliée que l'on désire modifier et ainsi traduire ou vérifier la traduction de l'ensemble d'un logiciel. Pour finir, la commande de gestion d'interface utilisateur permet aussi de saisir et de modifier la valeur de l'information "Catégorie" (voir figure 3.9). Cela dans le but d'organiser les champs en plusieurs onglets pour rendre encore plus lisible la boîte de création d'objets résultante. Les figures 3.9 et 3.10 illustrent bien l'utilisation de ces onglets. Les objets correspondant au champ "Géométrie" de l'entité "Self" sont effectivement affichés dans l'onglet correspondant à la catégorie "Géométrie".

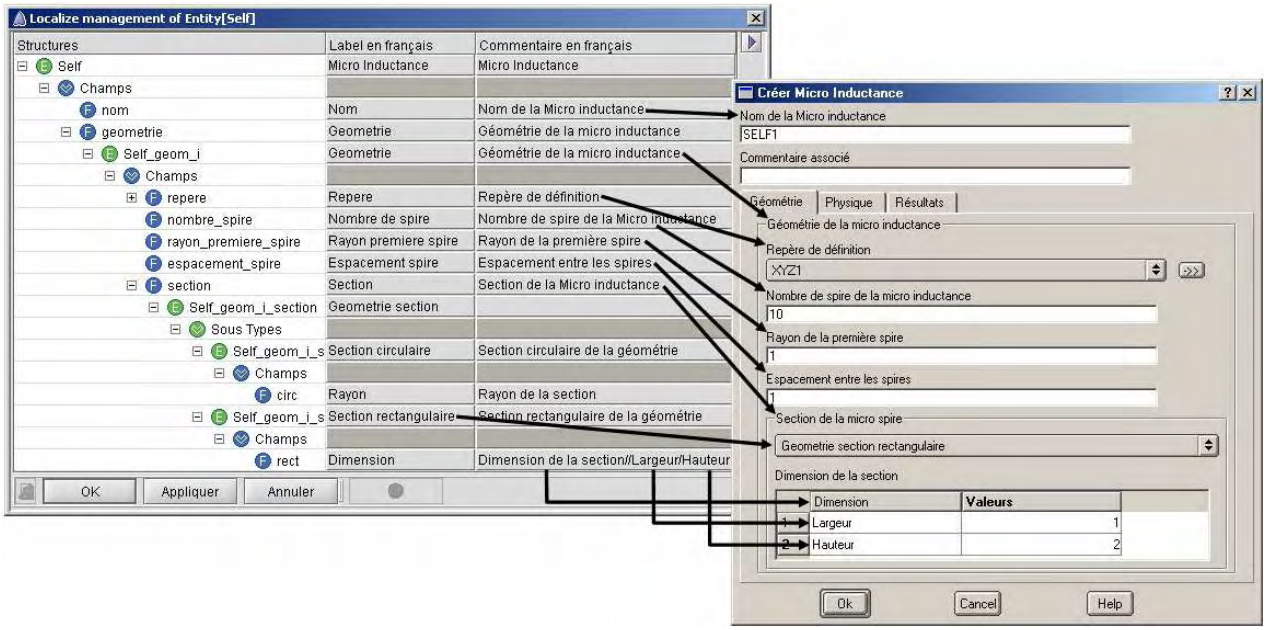


Figure 3.10 : Gestion des données d'interface utilisateur et la boîte de saisie correspondante

3.2.2.2 Gestion et évaluation des masques

Le nombre grandissant des cas d'utilisation et des contextes utilisés pour la modélisation de nos logiciels nous a amené à créer d'autres outils perfectionnés pour rendre le plus lisible possible le mécanisme de masquage. Notre but est de rendre compréhensible par tous les

concepteurs le fonctionnement des masques, et cela même pour des applications aussi complexes que les logiciels Flux.

Ces deux nouveaux outils de gestion et d'évaluation des masques permettent d'obtenir une utilisation optimale du mécanisme de contextualisation. En effet, la commande de gestion des masques permet de saisir et de modifier les masques d'une entité, d'une commande ainsi, que leurs champs et arguments respectifs.

Le composant de synthèse utilisé pour réaliser ces outils permet de visualiser à la fois la structure des objets et le masquage des données correspondant aux contextes de l'application modélisée. Le masquage par défaut prédominant sur tous les autres masques et aussi présenté par ce composant. De plus, pour ces deux outils, une dimension supplémentaire a été ajoutée au niveau des titres de colonnes. En effet, un contexte pouvant avoir des sous-contextes, l'affichage de ceux-ci comporte donc plusieurs niveaux correspondant à la structuration des contextes (voir figure 3.11).

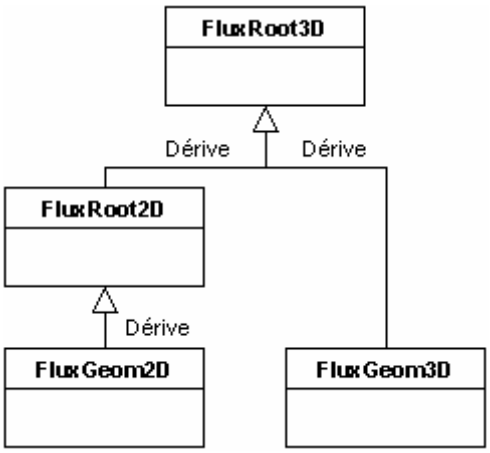


Figure 3.11 : Hiérarchie des contextes

Le but de ces outils est donc de pouvoir affecter un masque spécifique à un objet par rapport à un contexte donné, c'est-à-dire cacher les informations que l'on ne souhaite pas voir ou que l'on ne veut pas rendre modifiables par l'utilisateur. Toute la difficulté des masques repose dans la surcharge et l'héritage qui s'opère entre les contextes et leurs sous-contextes ainsi qu'entre les entités, les super-types et les champs. Le masque effectif pour chaque entité et chaque champ est donc évalué (voir figure 3.12).

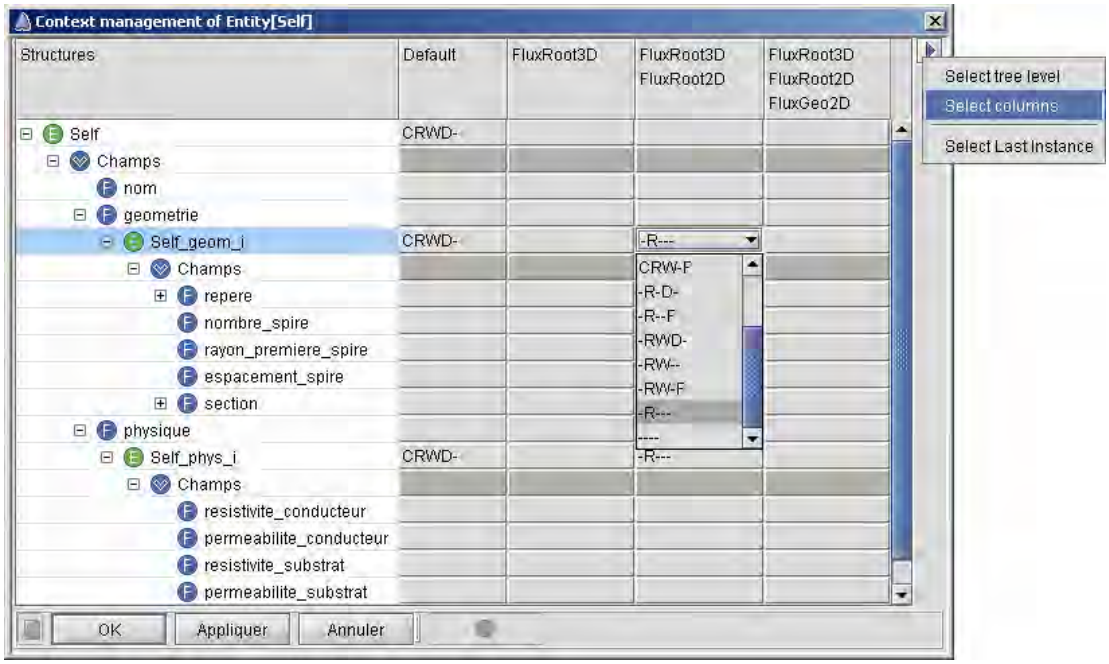


Figure 3.12 : Outil de management des masques

L'outil de gestion des masques permet de visualiser pour les contextes sélectionnés et donc visibles dans la boîte d'affichage, le masque effectif évalué lors de l'exécution du logiciel modélisé (voir figure 3.13).

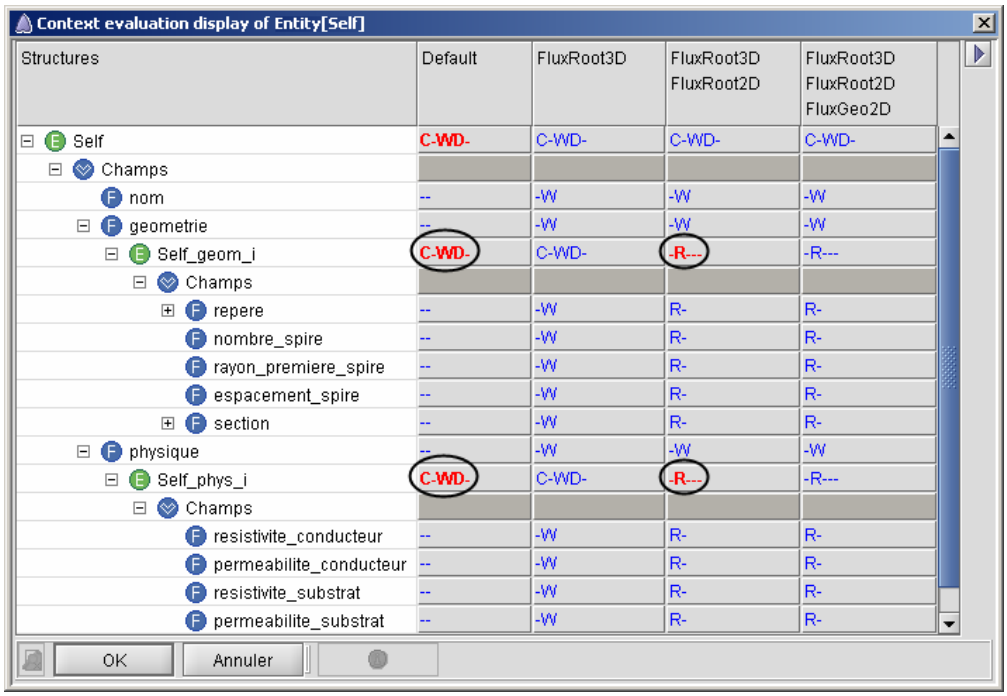


Figure 3.13 : Outil d'édition et d'évaluation des masques

Sur cette figure, nous pouvons apercevoir que cet outil fonctionne grâce à un code de couleur qui permet de distinguer les masques réellement saisis, en rouge et gras, des masques

déterminés par l'algorithme d'évaluation du masquage, en bleu et normal. Tous les masques affichés sont donc la représentation exacte du fonctionnement réel du logiciel. Ainsi, le logiciel FluxSpire permet par défaut de créer une micro inductance pour tous les contextes sauf pour ceux dérivant du contexte "FluxRoot2D". En effet, le modèle de la micro inductance est décrit en trois dimensions, il n'a pas de signification en deux dimensions (voir figure 3.13).

En conclusion, associés avec l'outil de gestion d'interface utilisateur, les outils de gestion des masques contextuels permettent de pouvoir modifier le rendu ainsi que le comportement de la boîte de dialogue résultant de la modélisation des concepteurs de logiciel. L'avantage principal de ces outils est la représentation basée sur l'association d'un arbre et d'un tableau qui permet une synthèse des données et donc une vision d'ensemble facilitant le travail de modélisation ainsi que l'échange d'informations entre concepteurs.

3.3 FluxFactory

La FluxFactory correspond à l'union de la machine virtuelle FluxCore et du logiciel de description de modèle FluxBuilder. Cette plate-forme comprend l'ensemble des composants nécessaires à la construction d'un logiciel métier, à savoir, le concepteur de modèle de données et le programme permettant d'interpréter et d'exécuter le modèle résultant.

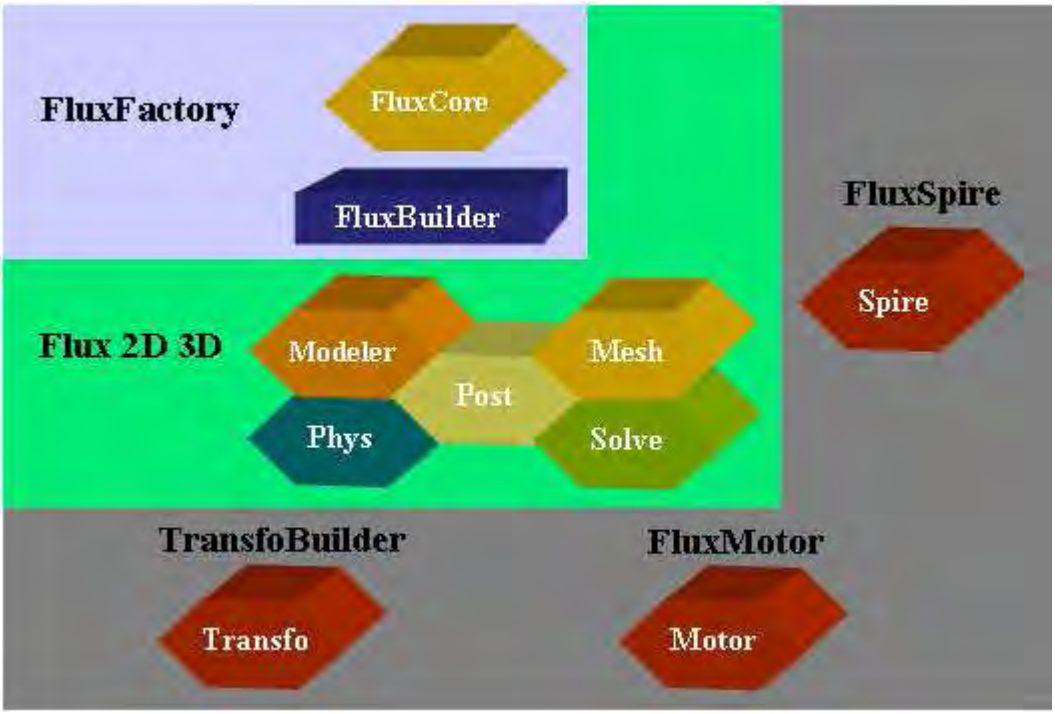


Figure 3.14 : Architecture des logiciels métiers

La FluxFactory est la plate-forme qui constitue la base de tous nos logiciels généralistes ou métiers autour du génie électrique. Elle peut, entre autre, être surchargée pour prendre en compte les spécificités des différents métiers modélisés (voir figure 3.14).

Pour réaliser nos logiciels métiers la FluxFactory est effectivement augmentée par les modules correspondant aux logiciels Flux. Ceux-ci étant surchargés par les modules correspondant aux logiciels métiers TransfoBuilder, FluxMotor ou FluxSpire.

Cette architecture clarifie notre stratégie globale de conception de logiciel métier. La plate-forme FluxFactory, réutilisée par l'ensemble des logiciels, permet de mutualiser au maximum les algorithmes communs à tout logiciel et donc de minimiser drastiquement la maintenance de toutes nos applications.

3.4 Auto conception de FluxBuilder

Le développement ultime de notre démarche de conception de logiciel s'illustre par la capacité d'auto conception de FluxBuilder. En effet, pour la réalisation de ce dernier, plusieurs choix s'offrent à nous : la conception indépendante du logiciel de conception de modèle, la conception par assemblage de blocs pré-programmés ou la conception par modélisation du logiciel FluxBuilder. Les avantages de la conception par modélisation nous ont déjà incité à construire nos logiciels par cette approche. Le logiciel FluxBuilder pouvant lui aussi être considéré comme une application spécifique au métier de conception de modèle de données, il nous a donc semblé tout à fait naturel de concevoir le logiciel FluxBuilder en le modélisant et en utilisant la machine virtuelle pour l'exécuter (voir figure 3.15).

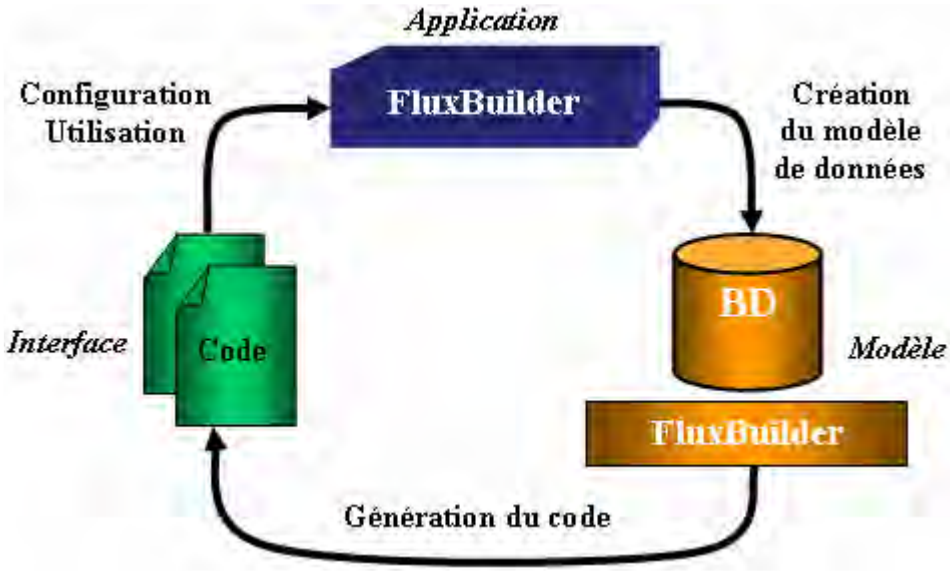


Figure 3.15 : Auto conception de FluxBuilder

Le logiciel FluxBuilder est donc, comme nos autres logiciels métiers, créé par modélisation et peut donc être modélisé en utilisant le logiciel d'aide à la création de modèles de données, à savoir le logiciel FluxBuilder lui-même. Notre démarche de conception est poussée à son paroxysme, le logiciel FluxBuilder facilitant la création de modèles de données permet de décrire son propre modèle et génère son propre code informatique.

3.5 Conclusion

La description du modèle de données d'un logiciel métier peut être fastidieuse. Les concepteurs doivent avoir une vision d'ensemble du logiciel qu'ils souhaitent modéliser. Sans outil approprié, la communication entre les différents acteurs participant à la modélisation est très difficile, cela complique la collaboration entre les différents experts et augmente les temps de développement et de maintenance des modèles.

C'est pourquoi, nous avons décidé de développer un descripteur autonome nommé FluxBuilder, permettant de remédier à tous ces problèmes. Ce logiciel est conçu pour faciliter la description des modèles de données de nos applications métiers. Il permet, grâce à une interface utilisateur conviviale et dédiée de présenter de manière claire la modélisation effectuée. Cette représentation de la structure des données permet aussi de pouvoir échanger ces informations entre tous les acteurs de la démarche de conception.

Le logiciel FluxBuilder est basé sur une représentation graphique de toutes les données décrivant le logiciel métier modélisé. Plusieurs outils comme des boîtes de saisie ou des diagrammes de classe, représentent d'une manière graphique et intuitive les données saisies par les concepteurs. D'autres outils perfectionnés, basés sur un nouveau type de composant graphique, permettent d'éclaircir les points délicats de la modélisation de logiciel et apportent une vision globale. La gestion des masques par exemple est relativement complexe pour un logiciel évolué comprenant un nombre important de contextes. Donc, pour rendre limpide pour n'importe quel concepteur ce mécanisme de masquage, deux outils spécifiques sont développés et intégrés dans le logiciel FluxBuilder. Tous ces composants permettent d'améliorer la compréhension du modèle de données de nos applications et donc de réduire de manière significative le temps de développement de nos modèles de données.

Le logiciel FluxBuilder permet aussi de simplifier la modification ou la correction de nos modèles en réduisant significativement le temps consacré à la maintenance de nos modèles de données. Enfin, poussant jusqu'au bout notre démarche de modélisation, le logiciel FluxBuilder est issu lui-même de sa modélisation et dans une certaine mesure s'auto construit.

Toute cette démarche, depuis la création du méta modèle de nos logiciels en passant par la réalisation de la machine virtuelle FluxCore et finissant par la conception du logiciel FluxBuilder, nous permet de décrire et de maintenir un grand nombre de logiciels.

4 Réalisations de logiciels métiers

4.1 Introduction

Après avoir expliqué les différentes étapes de notre démarche de conception de logiciel métier par modélisation, nous allons maintenant exposer les résultats que nous obtenons en utilisant cette méthode. Les exemples de logiciels, métiers ou généralistes, que nous présentons dans ce chapitre sont entièrement développés grâce à la méthode de conception que nous avons exposée dans les chapitres précédents: les modèles de nos applications sont construits grâce au logiciel FluxBuilder et exécutés grâce à la machine virtuelle FluxCore. Cette plate-forme FluxFactory permet de réaliser à la fois nos logiciels métiers mais aussi nos objets métiers intégrés dans le logiciel généraliste Flux.

Nous allons présenter dans ce chapitre des applications de notre démarche de conception de logiciels. Ces applications ont été réalisées dans le cadre de stage de master, de thèse au sein des équipes de développement de la société Cedrat.

Le premier exemple que nous souhaitons présenter est un logiciel métier appelé TransfoBuilder. Cette application dédiée est utilisée par le service de recherche et développement d'Electricité De France pour faciliter le travail de caractérisation de leurs transformateurs.

Ensuite, pour valoriser notre approche, la société Cedrat et le Laboratoire d'Electrotechnique de Grenoble ont réalisé une mise à jour du logiciel InCa déjà développé au Laboratoire d'Electrotechnique de Grenoble. Pour cela, nous avons redéfini ce logiciel de calcul d'impédance en modélisant non seulement sa partie données mais aussi son interface homme machine.

Un des axes principal de nos travaux est de rendre possible l'intégration d'objets métiers dans les logiciels généralistes Flux. Cela permet d'utiliser la puissance du logiciel généraliste tout en facilitant le travail de l'utilisateur en intégrant des objets et algorithmes spécifiques à son métier. Cette démarche a été validée en intégrant dans le logiciel généraliste Flux, des objets métiers moteurs. L'union de ces objets métiers et du logiciel généraliste a permis d'obtenir l'application métier FluxMotor. Cette application, dédiée au métier de motoriste, permet de faciliter le travail de conception de moteurs en réalisant automatiquement les phases de création géométrique de l'objet moteur et en affectant, aussi de manière

automatique, les paramètres physiques et électriques aux régions surfaciques constituant le moteur.

Enfin, les derniers exemples que nous avons décidé d'exposer sont les logiciels généralistes Flux eux-mêmes. La modélisation de ces applications, également développées grâce à notre démarche de conception, permet de valider de manière définitive l'efficacité de notre approche. En effet, les logiciels Flux sont très complexes autant d'un point de vue modèle de données que d'un point de vue interface homme machine. De plus, ils sont composés d'un très grand nombre de cas d'utilisation et de contextes. Ces logiciels généralistes nous ont permis, à la fois de faire évoluer notre démarche mais aussi de valider efficacement nos outils de développement intégrés dans le logiciel de description de modèle FluxBuilder. Un outil comme le gestionnaire de masques est indispensable pour des logiciels aussi complexes que les logiciels Flux.

4.2 TransfoBuilder

L'application TransfoBuilder est un logiciel métier de caractérisation et de conception de transformateurs. Il est développé en partenariat entre le Laboratoire d'Electrotechnique de Grenoble et la société Electricité De France. Le but de ce projet est de pouvoir caractériser un transformateur sous la forme d'un schéma électrique RLCG équivalent. Les ingénieurs du centre de recherche et développement d'Electricité De France utilisent ensuite ce modèle pour l'analyse du comportement des transformateurs soumis à des surtensions telles que les coups de foudre, l'enjeu étant l'étude du vieillissement des matériels électriques (voir figure 4.1).

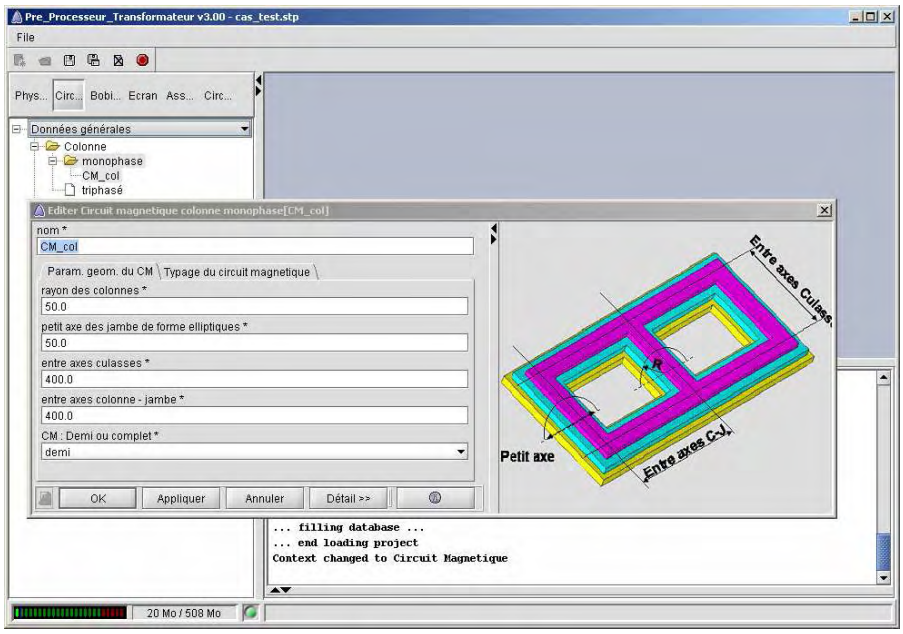


Figure 4.1 : Le logiciel TransfoBuilder

L'objectif du logiciel métier TransfoBuilder est de réduire le temps de modélisation des transformateurs réalisés auparavant directement dans le logiciel généraliste Flux3D. Grâce à cette application dédiée, on peut maintenant saisir rapidement les caractéristiques du transformateur que l'on souhaite modéliser. Puis, à l'aide des algorithmes spécifiques créés par les experts du métier, le logiciel TransfoBuilder crée des fichiers en langage de commande Python directement exécutables sous les logiciels généralistes Flux. Ces fichiers de commande construisent automatiquement, dans Flux, la géométrie et affectent aussi les paramètres physiques et électriques aux différents volumes constituant le transformateur (voir figure 4.2).

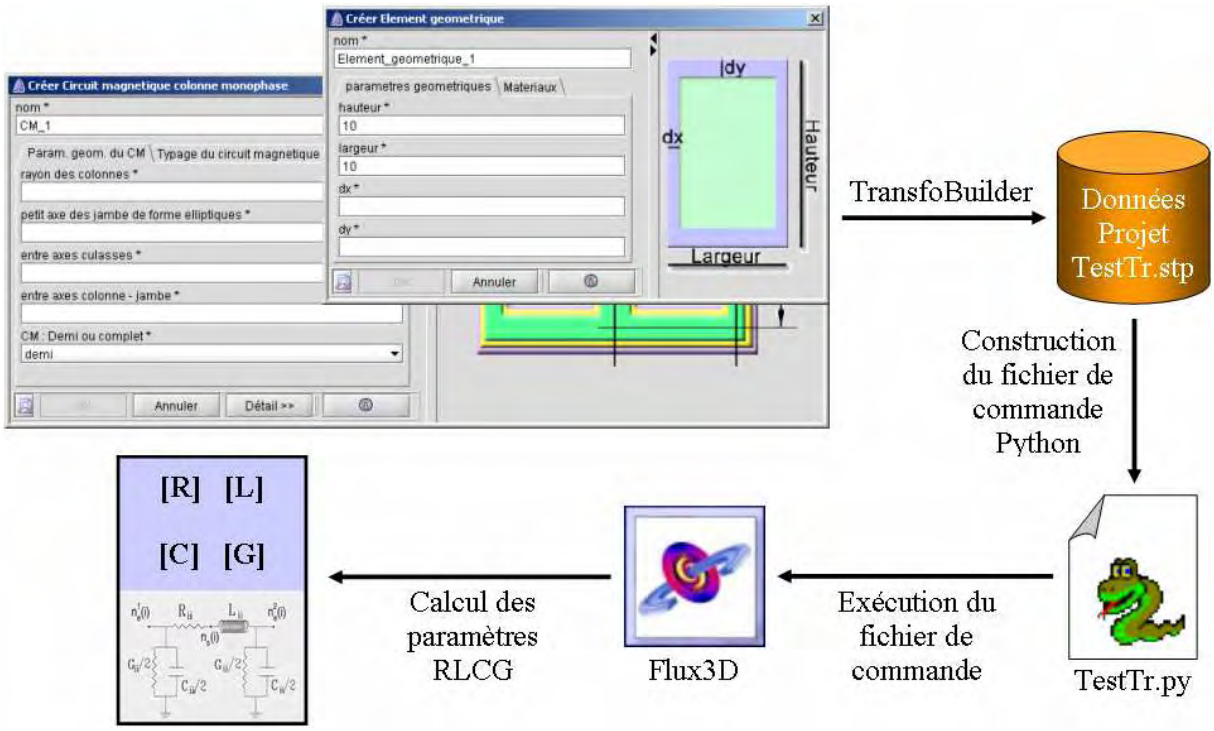


Figure 4.2 : Démarche d'utilisation du logiciel métier TransfoBuilder

La démarche de caractérisation des transformateurs mise en place par les ingénieurs d'Electricité De France se poursuit alors par le calcul des paramètres RLCG. Ces paramètres sont ensuite utilisés pour l'étude du comportement de ces dispositifs soumis à des surtensions.

Le logiciel TransfoBuilder est l'exemple le plus représentatif de notre démarche car il est totalement indépendant des logiciels généralistes Flux. Cette application métier est construite directement avec la machine virtuelle FluxCore et n'utilise donc aucun module spécifique des applications Flux. De plus, contrairement aux objets métiers intégrés dans une interface déjà

existante, ce logiciel possède sa propre interface homme machine, il peut donc être spécialisé au maximum pour le métier de concepteur de transformateur.

Enfin, il faut noter qu'il a été entièrement conçu et réalisé par des stagiaires de Master, ce qui prouve l'utilisabilité de notre démarche, y compris par des stagiaires de courte durée [JEANNIARD-03] [DELCROIX-04] [ALLARD-05].

4.2.1 Création du modèle de données

Le modèle de données du logiciel TransfoBuilder contient tous les objets nécessaires à la description de deux types de transformateurs, les transformateurs cuirassés et les transformateurs à colonnes (voir figure 4.3).

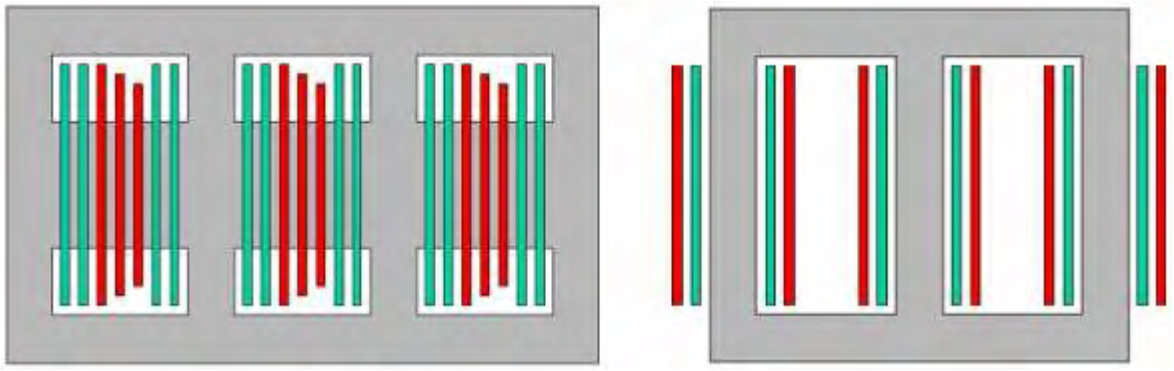


Figure 4.3 : Transformateur cuirassé et transformateur à colonnes

Pour organiser au mieux les données, des règles de construction du modèle ont été mises en place afin d'obtenir le résultat le plus lisible possible pour tout développeur participant ou reprenant le projet TransfoBuilder.

L'architecture des données doit correspondre de la façon la plus proche possible à la constitution physique réelle d'un transformateur. Dans le logiciel TransfoBuilder, le modèle est découpé en packages représentant les grandes étapes de la construction d'un transformateur : le bobinage, le circuit magnétique, les conducteurs, les écrans, le circuit électrique (voir figure 4.4).

Ainsi, la prise en main de ce logiciel métier est grandement facilitée, le langage utilisé étant exactement celui employé par le spécialiste du métier de concepteur de transformateurs.

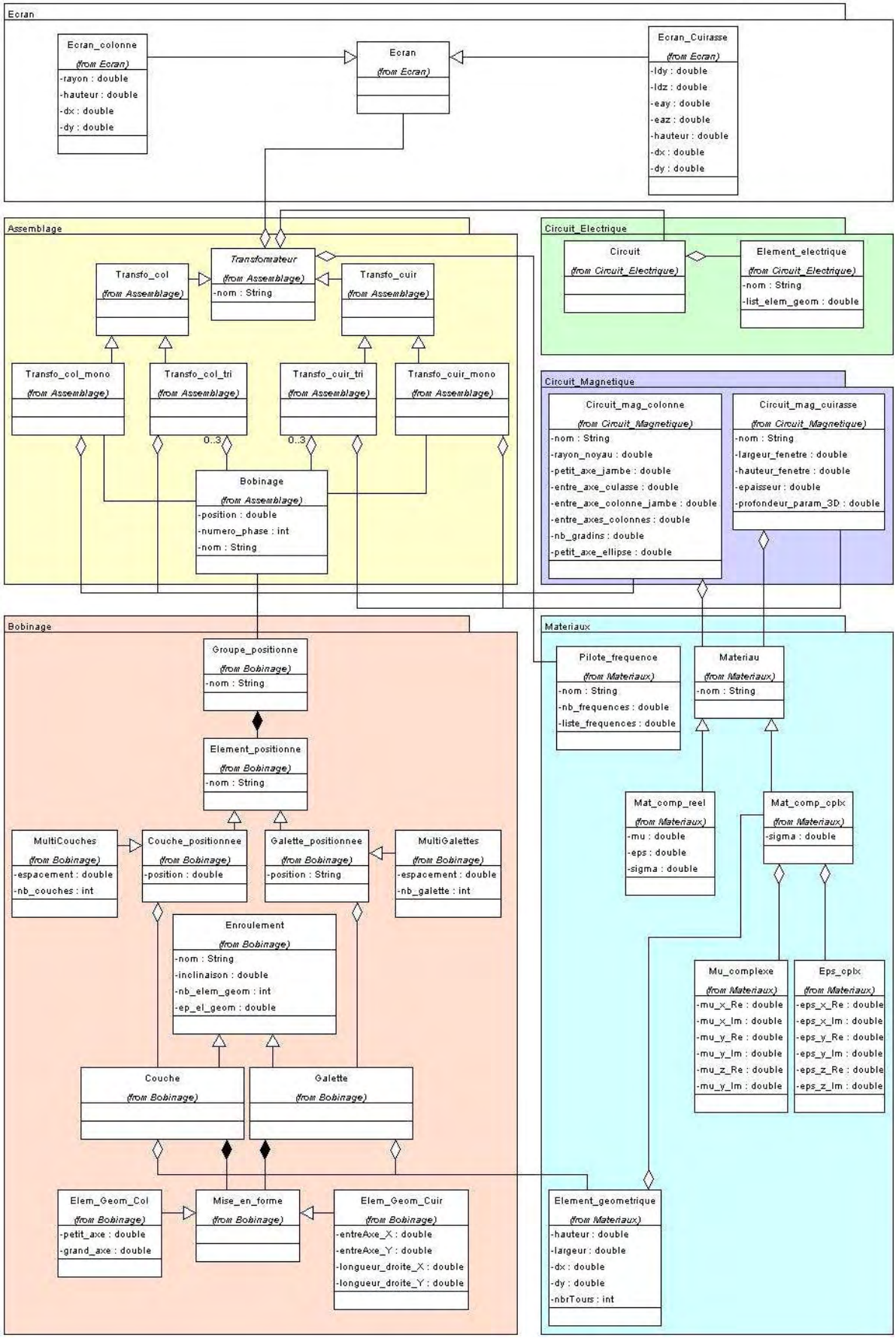


Figure 4.4 : Modèle de données du logiciel TransfoBuilder

4.2.2 Le dialogue utilisateur

Le logiciel TransfoBuilder est une application totalement indépendante et construite entièrement grâce à notre démarche de conception. Cela permet de pouvoir spécialiser l'intégralité de cette application dédiée au métier de concepteur de transformateur. Le logiciel TransfoBuilder est composé de plusieurs contextes correspondant aux différentes phases de conception d'un transformateur. Cette interface permet de représenter le plus fidèlement possible les différentes tâches réalisées lors de l'assemblage d'un transformateur. Ainsi, l'utilisateur, lors de sa prise en main du logiciel se retrouve confronté à cinq onglets structurant les grands éléments d'un transformateur :

- **Physique :** Permet la création des matériaux et des conducteurs. Correspond au choix des câbles composant le bobinage.
- **Circuit Magnétique :** Création des circuits magnétiques. Empilage des couches feuilletées pour former le circuit magnétique.
- **Bobinage :** Création des enroulements. Phase où l'opérateur crée ses enroulements en enroulant du conducteur sur un cône.
- **Ecran :** Création des écrans électrostatiques.
- **Circuit électrique :** Création du circuit représentant les connections entre les différents conducteurs.
- **Assemblage :** Phase où l'opérateur positionne les bobinages et les écrans par rapport au circuit magnétique.

Assez rapidement, lors de la conception du logiciel TransfoBuilder, le besoin de fournir à l'utilisateur une aide à la saisie des différents paramètres est devenu une nécessité. En effet, le grand nombre et la complexité des paramètres à saisir rendent nécessaires ces explications supplémentaires. Ainsi, nous avons décidé d'associer aux boîtes de saisie, des images permettant d'expliquer à quoi correspondent les différents paramètres affichés dans ces boîtes (voir figure 4.5).

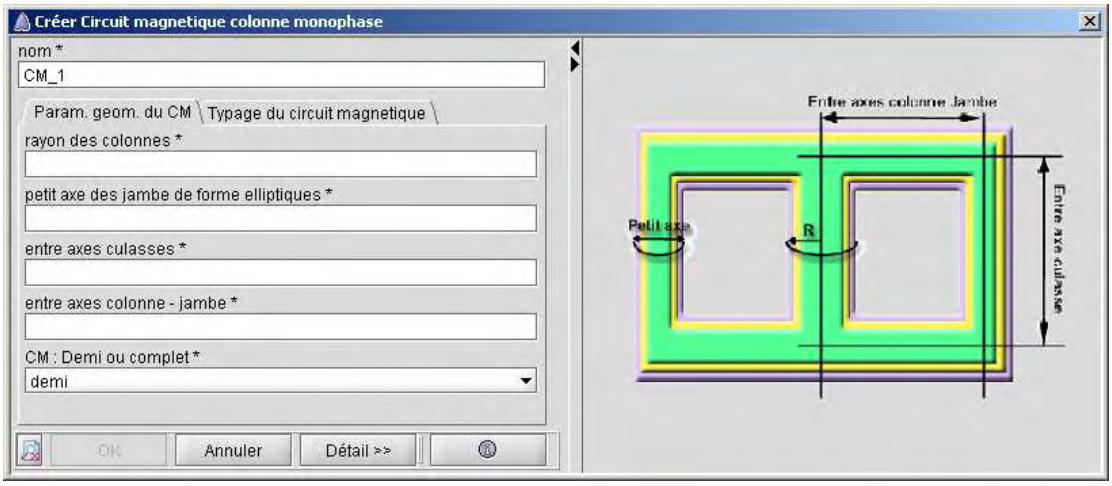


Figure 4.5 : Boîte de saisie d'un circuit magnétique

4.2.3 Algorithmes spécifiques

Le logiciel TransfoBuilder est développé en collaboration avec les experts du métier de conception de transformateur dans le but d'intégrer non seulement le vocabulaire précis du métier mais aussi des algorithmes spécifiques correspondant aux besoins réels des utilisateurs. Ainsi, ces derniers n'ont plus à créer point par point toute la géométrie du transformateur. L'affectation des paramètres physiques et électriques est elle aussi automatisée. Ces divers algorithmes constituent la force des logiciels métiers, ils permettent de soulager les utilisateurs dans leur démarche.

4.2.3.1 Création automatique de géométrie

Le premier algorithme développé et intégré dans le logiciel TransfoBuilder permet la création automatique de la géométrie du transformateur en fonction des paramètres saisis par les concepteurs. Pour cela, l'utilisateur, après avoir saisi les données de son transformateur, active une commande qui génère un fichier de commande pilotant la construction de la géométrie du transformateur. Ce fichier, écrit en langage Python (langage de commande des logiciels Flux), est ensuite exécuté dans le logiciel généraliste Flux.

Le principe de la construction de la géométrie est assez simple. Nous créons au préalable une bibliothèque Python permettant la création paramétrée des éléments du transformateur. Ces classes correspondent à tous les éléments constituant nos transformateurs. Ainsi, le fichier "EcranCol.py", par exemple, permet de créer un écran électrostatique en fonction d'un certain nombre de paramètres tels que la hauteur, le rayon ou la phase sur laquelle il devra être affecté… Tous ces fichiers Python respectent le même format, ils doivent décrire la fonction permettant la création, dans le logiciel Flux, de la géométrie de l'objet. Ainsi, tout le savoir du

corps de métier est contenu au sein de ces fichiers Python décrivant le paramétrage de tous nos types de transformateurs.

Le fichier de création de la géométrie complète du transformateur fait donc appel aux commandes décrites ci-dessus. L'algorithme commence toujours par créer le circuit magnétique puis il crée les repères de positionnement des conducteurs et enfin les enroulements et les écrans.

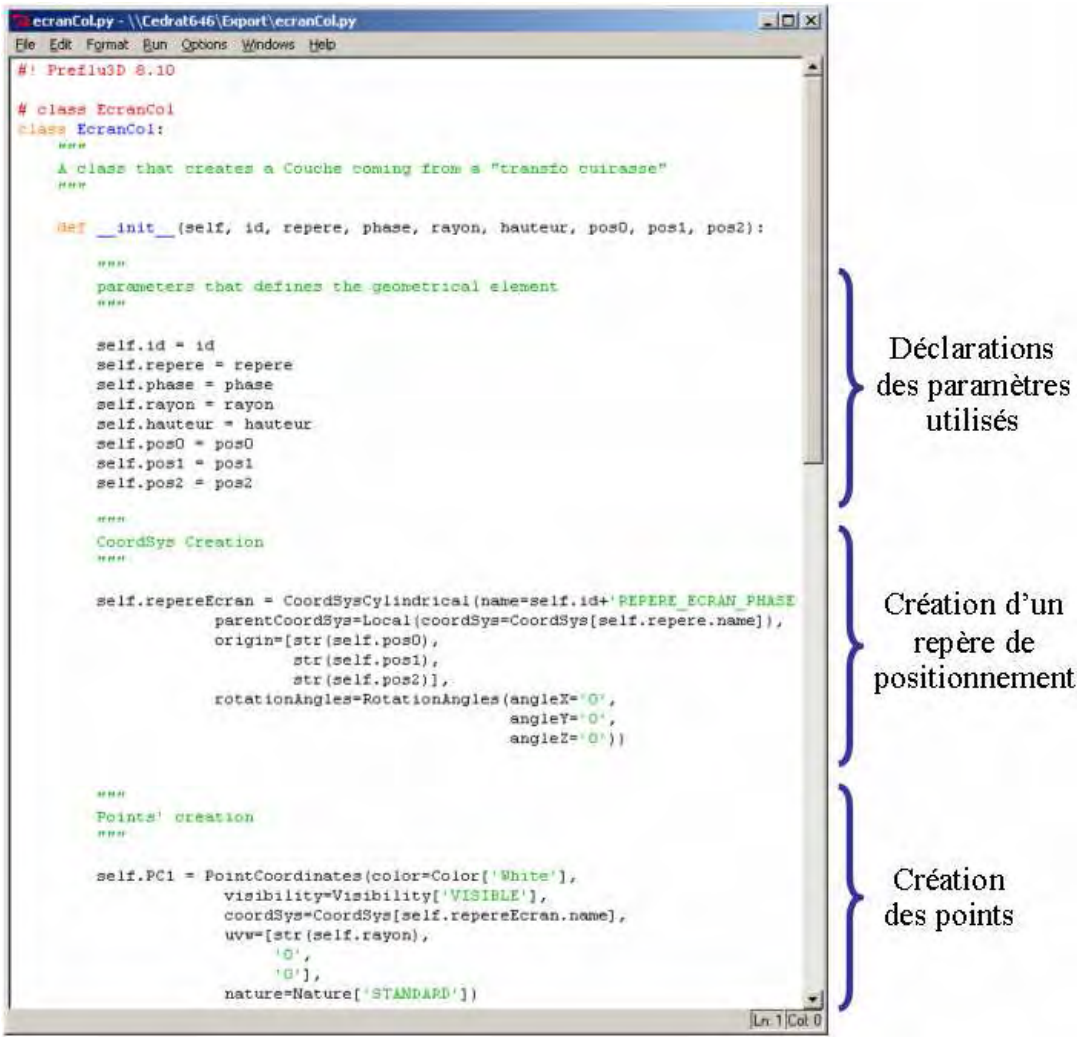


Figure 4.6 : Exemple de géométrie dans Flux : EcranCol.py

4.2.3.2 Affectation automatique des paramètres

Après avoir construit la géométrie de son transformateur le concepteur doit créer les matériaux correspondant à la partie physique de son étude puis affecter ces matériaux à sa géométrie. Cette phase de modélisation a aussi été automatisée, ainsi les experts ont associé aux volumes de la géométrie des transformateurs les matériaux dont les caractéristiques ont été saisies par l'utilisateur final.

D'autre part, le schéma électrique du transformateur doit lui aussi être affecté à la géométrie. Pour cela, une fenêtre de saisie spécifique a été conçue, toujours dans l'optique de simplifier la saisie pour l'utilisateur. Le concepteur de transformateur dispose d'une boîte de dialogue représentant la géométrie simplifiée de son transformateur et en définit le câblage par de simples clics de souris (voir figure 4.7).

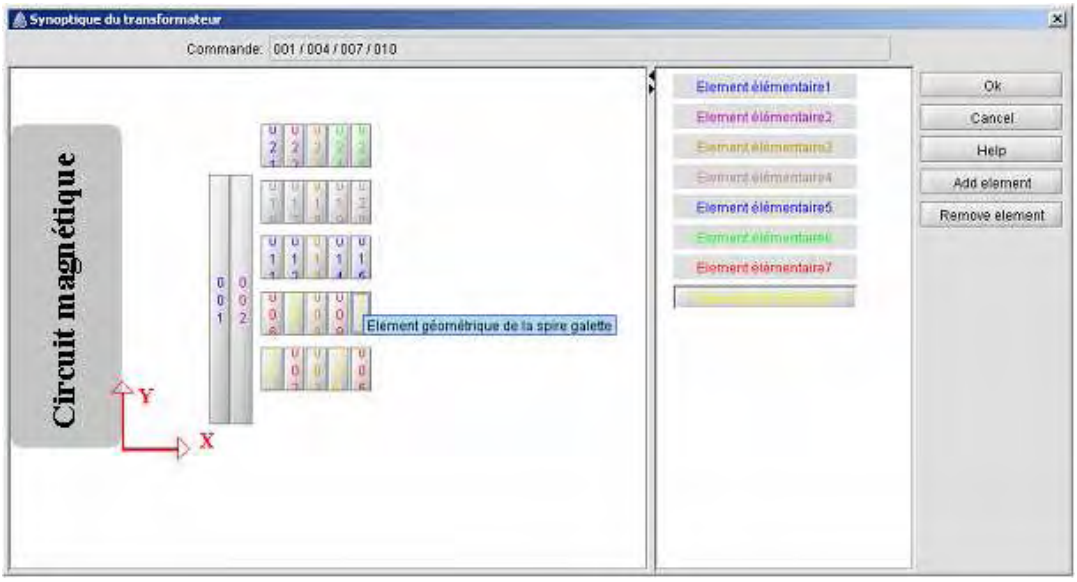


Figure 4.7 : Outil de saisie de circuit électrique

Cette commande spéciale a pour effet d'ajouter dans le fichier de commande Python les instructions de création des inducteurs et leurs affectations aux différents volumes de la géométrie du transformateur.

4.2.4 Evolutions et perspectives

Dans l'état actuel de TransfoBuilder, le maillage reste à la charge de l'utilisateur, qui doit pour cela, disposer de compétences d'utilisation des logiciels Flux. A terme, cette étape devrait elle aussi être automatisée. Dans ce cas, l'utilisateur de TransfoBuilder ne verra plus le logiciel Flux3D mais ne fera qu'utiliser les capacités de calcul des logiciels Flux.

L'évolution consistant à mailler de manière automatique les transformateurs est cependant compliquée à mettre en place. En effet, le maillage est le point clé qui permet d'obtenir des résultats satisfaisants tout en conservant un temps de résolution raisonnable. Ce compromis est très dur à trouver et dépend beaucoup de la géométrie du fait de la présence de couches minces. C'est dans ce compromis que le savoir-faire des ingénieurs modélisant des transformateurs intervient. Cette évolution devra sans nul doute nécessiter une forte concertation avec des experts réalisant régulièrement ce genre d'opération.

4.3 Inductance Calculation [InCa]

InCa (Inductance Calculation) est un logiciel de conception assisté par ordinateur dédié à l'analyse et à la détermination des inductances parasites dues au câblage. Cette application est basée sur une méthode analytique exacte qui permet d'obtenir très rapidement les valeurs des éléments parasites induits dans les dispositifs étudiés. Le domaine d'activité du logiciel métier InCa est très varié : Il s'étend de l'étude des circuits intégrés à l'analyse de convertisseurs statiques en passant par les bus barres, les barres massives, diverses connectiques ou encore les fusibles (voir figure 4.8).

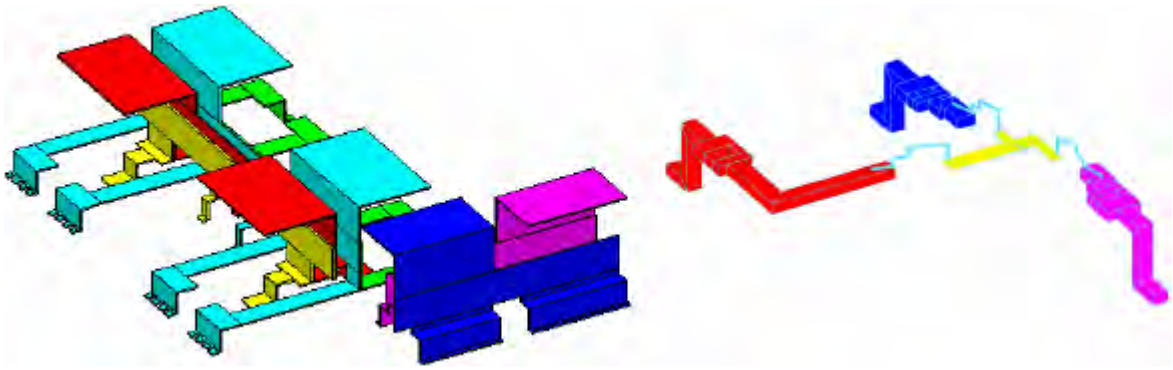


Figure 4.8 : Bus barres et Circuits intégrés

De plus, ce logiciel métier permet, d'après la description du dispositif, d'établir son circuit équivalent en fonction de la fréquence. Grâce au logiciel InCa, les concepteurs peuvent par exemple évaluer les évolutions de surtensions lors de la mise en parallèle de composants, mais aussi modéliser les couplages entre structures avec pour objectif l'intégrité d'un signal.

Ce logiciel est issu du Laboratoire d'Electrotechnique de Grenoble et concrétise de nombreuses années de recherche [CLAVEL-96] [SCHANEN-94] [GUICHON-01]. Récemment, un partenariat avec la société Schneider Electric a été mis en place afin de faire évoluer le logiciel InCa vers un logiciel plus intuitif et plus convivial. La société Cedrat a donc décidé d'appliquer notre méthode de conception par modélisation afin de décrire l'application InCa de la même manière que nos autres logiciels métiers.

Dans un premier temps, le modèle de données a été converti pour être compatible avec notre logiciel de description de modèles d'application. Puis, ce modèle a été enrichi de toutes les informations supplémentaires introduites par notre démarche. Ainsi, les informations de type interface utilisateur des entités et des commandes utilisées dans le logiciel métier InCa ont été décrites grâce au logiciel FluxBuilder. De plus, l'interface homme machine à elle aussi été complètement décrite. Le résultat obtenu est donc très proche de l'interface de nos autres

logiciels métiers. Ainsi, les utilisateurs déjà familiarisés avec ce type d'interface peuvent prendre en main le logiciel InCa très rapidement (voir figure 4.9).

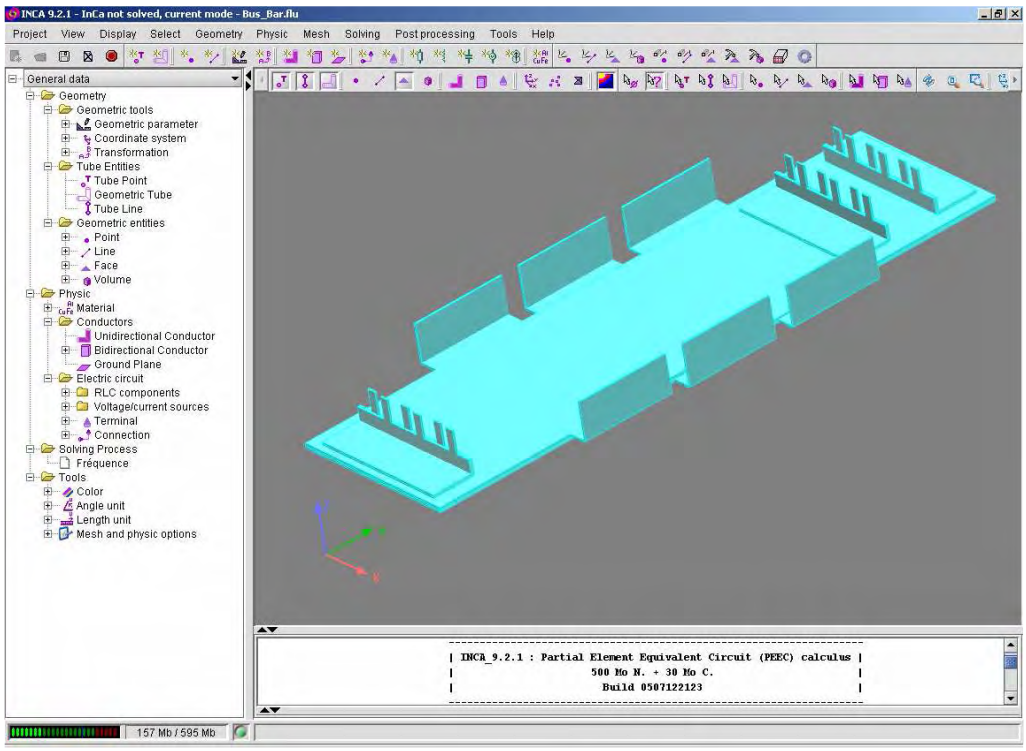


Figure 4.9 : Logiciel métier InCa

Le modèle de données ainsi que l'interface homme machine de cette application reprennent le langage des métiers de l'électrotechnique. Cela permet de faciliter grandement la prise en main des utilisateurs.

Enfin, l'application de notre démarche à la création du logiciel InCa permet non seulement de réduire grandement sa maintenance mais aussi de pouvoir bénéficier de toutes les évolutions de la machine virtuelle : tous nos logiciels métiers évoluent de la même manière et restent tous très semblables dans leur maniement.

4.4 FluxMotor

FluxMotor est un logiciel dédié au métier de motoriste. Développé au sein de la société Cedrat, il est issu d'un partenariat avec le Laboratoire d'Electrotechnique de Grenoble et le laboratoire Speed de l'université de Glasgow [CHANTREL-04] [LACOMBE-05]. L'application FluxMotor n'est pas un logiciel métier à part entière. Cette application correspond plutôt à un ensemble d'objets métiers intégrés dans les logiciels généralistes Flux. Le logiciel FluxMotor correspondant se présente donc de la même manière que le logiciel Flux2D, cependant, il propose une multitude d'objets métiers permettant de modéliser différents types de moteur.

Ces objets, dédiés au métier de motoriste, ont été créés dans le but de simplifier et d'aider les concepteurs à modéliser leurs moteurs dans l'environnement généraliste Flux. Le logiciel FluxMotor est donc basé sur une bibliothèque d'objets moteurs créée et enrichie par les experts du domaine des moteurs électriques (voir figure 4.10).

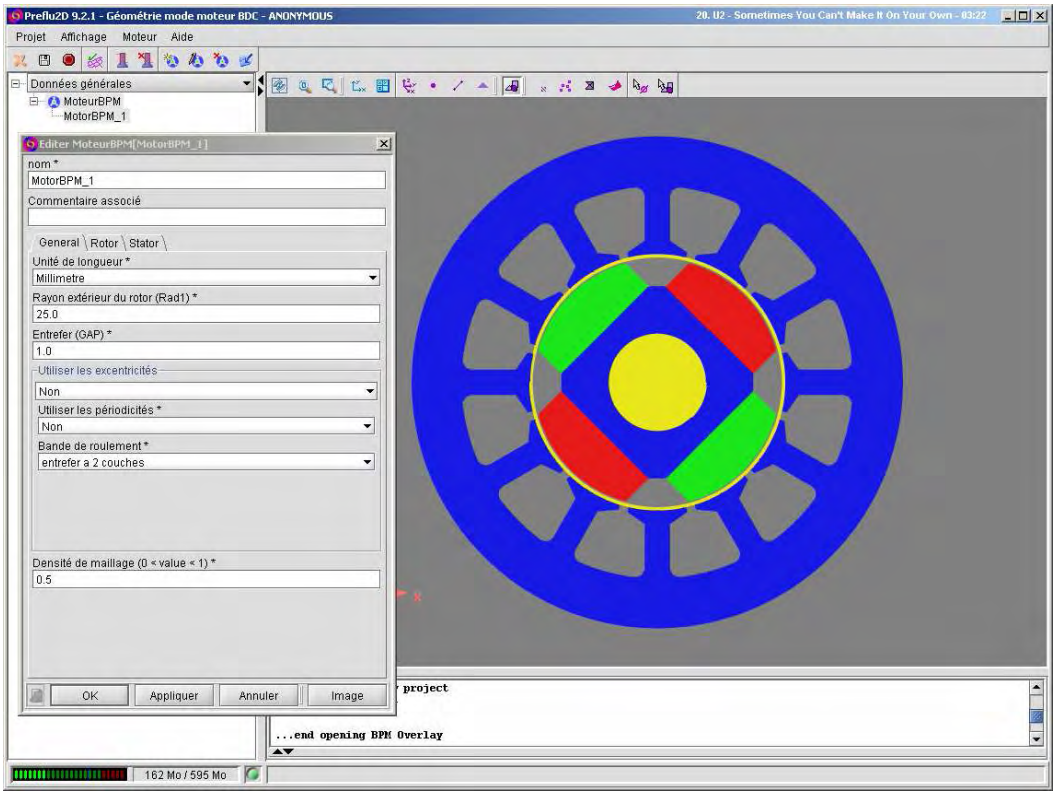


Figure 4.10 : Objets métiers moteurs

En un sens, cette application diffère assez sensiblement des logiciels métiers précédents. Jusqu'à présent, les applications métiers devaient être réalisées par un informaticien ayant des compétences métier. Dans le cadre de l'application FluxMotor, constituée d'une bibliothèque d'objets métier, nous avons souhaité offrir à l'utilisateur de Flux, disposant des compétences métiers et d'une bonne connaissance de Flux, la capacité d'introduire ces propres objets métiers. Pour cela, nous avons développé une approche de conception d'objets métiers, dérivé de l'approche de conception d'applications métiers.

Dans un premier temps, nous allons expliquer cette démarche de conception d'objets métiers. Puis, afin d'exposer le résultat obtenu par l'application de cette approche au métier de concepteur de moteur, nous décrirons le fonctionnement du logiciel FluxMotor.

4.4.1 Démarche de conception d'objet métier

Pour formaliser le plus possible la création d'objets métiers, nous avons donc développé une démarche de conception spécialement dédiée à ceux-ci. Nous avons tout d'abord défini un patron ("pattern") spécifique pour décrire les opérations de base de nos objets, à savoir la création, la modification et la suppression. C'est le motif de base sur lequel peut s'appuyer toute introduction d'objets métiers (voir figure 4.11).

```
#! Preflu2D 9.21
# class FluxMotor

#-------------------------------------------------------
# HISTORY :
# 05/08/29 start of the history (GL)
#-------------------------------------------------------

class FluxTruc(FluxRepresentation):

    def __init__ (self, paramDict):
        FluxRepresentation.__init__(self)
        #réglage des paramètres
        self.createParameters(paramDict)
        #construction
        self.build()

    ###################################################
    # Method:          # createParameters()
    # Purpose:         # Creates the geometric parameters
    # Parameters:      # None
    # Returns:         # None
    ###################################################
    def createParameters(self, paramDict):
        pass

    ###################################################
    # Method:          # build()
    # Purpose:         # build the flux object
    # Parameters:      # None
    # Returns:         # None
    ###################################################
    def build(self):
        pass

    ###################################################
    # Function:        # deleteFluxObject()
    # Purpose:         # Completely deletes the
    # Parameters:      # None
    # Returns:         # None
    ###################################################
    def deleteFluxObject(self):
        pass

    ###################################################
    # Function:        # changeParameterValue(a
    # Purpose:         # Changes parameter
    # Parameters:      # attributeId - paramete
    # Returns:         # None
    ###################################################
    def changeParameterValue (self, attributeId
        pass
```

Figure 4.11 : Patron d'objet métier

Un expert concevant un objet relatif à son métier doit non seulement exprimer son modèle de données, mais également écrire les fonctions composant le patron présenté ci-dessus.

4.4.1.1 Patron d'objet métier

Le premier objectif de notre démarche est donc la définition avec précision d'un patron en langage Python pour la création future de nos objets métiers. Ce patron doit donc être la base de toutes les classes d'objets qui seront créées. Dans ce but, nous avons intégré dans le logiciel de description une commande permettant la création de fichier Python. Celle-ci est entièrement basée sur notre patron et correspond aux données du modèle. Ainsi, après avoir créé son modèle, le concepteur obtient automatiquement ces fichiers Python correspondants. Le patron que nous avons réalisé est constitué des composants suivants :

- Un constructeur **__init__** qui sert à initialiser l'objet métier. Cette méthode appelle successivement les deux méthodes qui suivent.
- Une méthode **createParameters** qui sert à initialiser les paramètres de l'objet métier.

- Une méthode **build** qui correspond à la construction dans Flux des points, des lignes … de l'objet métier.
- Une méthode **deleteFluxObject** qui permet de détruire tous les éléments constituant l'objet métier.
- Une méthode **changeParameterValue** qui correspond à la gestion de la modification des paramètres.

Il est à noter qu'une bonne utilisation de la méthode changeParameterValue permet une gestion incrémentale de l'objet. En effet, la modification de certains paramètres ne nécessite pas la reconstruction complète de l'objet. Par exemple, le changement du type d'encoche d'un stator entraîne sa reconstruction complète, alors que la modification de la profondeur d'une encoche ne nécessite que la modification du paramètre correspondant.

4.4.1.2 Template Editor

Si le concepteur est un expert de son métier, il n'est pas forcément expert dans l'utilisation de notre logiciel de description de modèle nommé FluxBuilder. Donc, dans le but de faciliter la création de modèle de données, nous avons décidé de concevoir un logiciel simple d'utilisation et spécialisé dans la description d'objet métier.

Ce logiciel simplifié s'appuie sur FluxBuilder et sur le mécanisme de masquage exposé dans un chapitre précédent. Ainsi, seules les informations utiles à la description d'objets métiers sont présentes dans le logiciel Template Editor (voir figure 4.12).

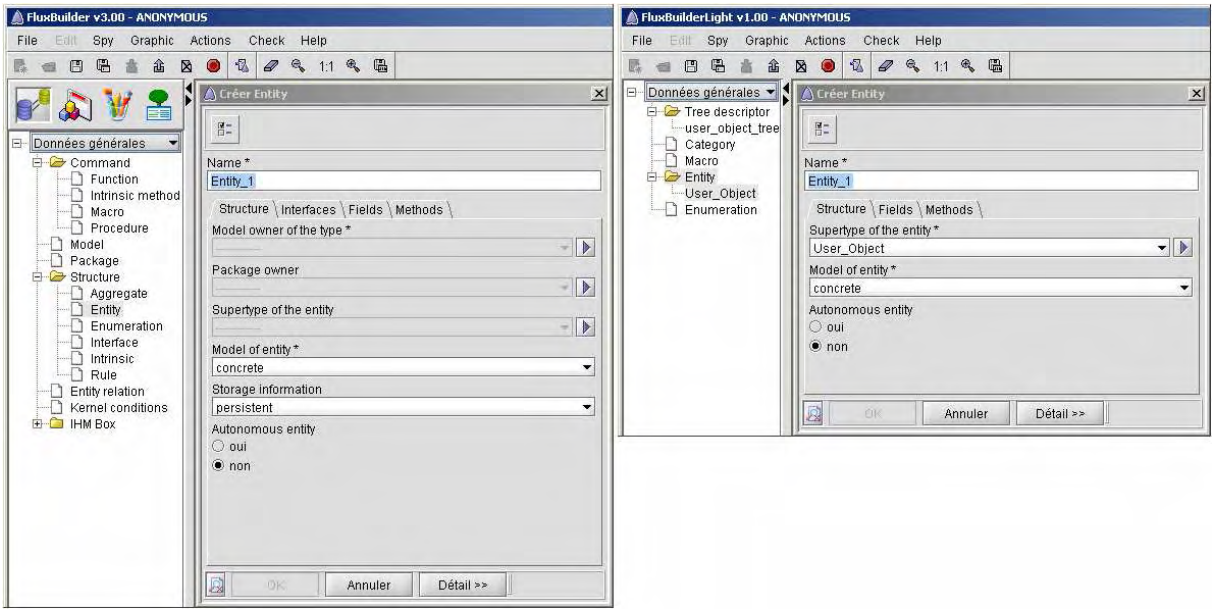


Figure 4.12 : FluxBuilder et Template Editor

De plus, le logiciel Template Editor permet la création automatique des patrons en langage Python correspondants aux modèles d'objets métiers saisis. Le concepteur d'objets métiers n'a plus qu'à écrire le code correspondant à son objet dans le patron ainsi créé.

4.4.2 Intégration dans le logiciel généraliste

Le lien entre les objets métiers créés grâce au logiciel Template Editor et les logiciels généralistes Flux est réalisé grâce à un changement de cas d'utilisation. Le basculement sur le cas d'application FluxMotor entraîne le chargement dynamique du modèle de données ainsi que du modèle de l'interface homme machine de l'application FluxMotor (voir figure 4.13).

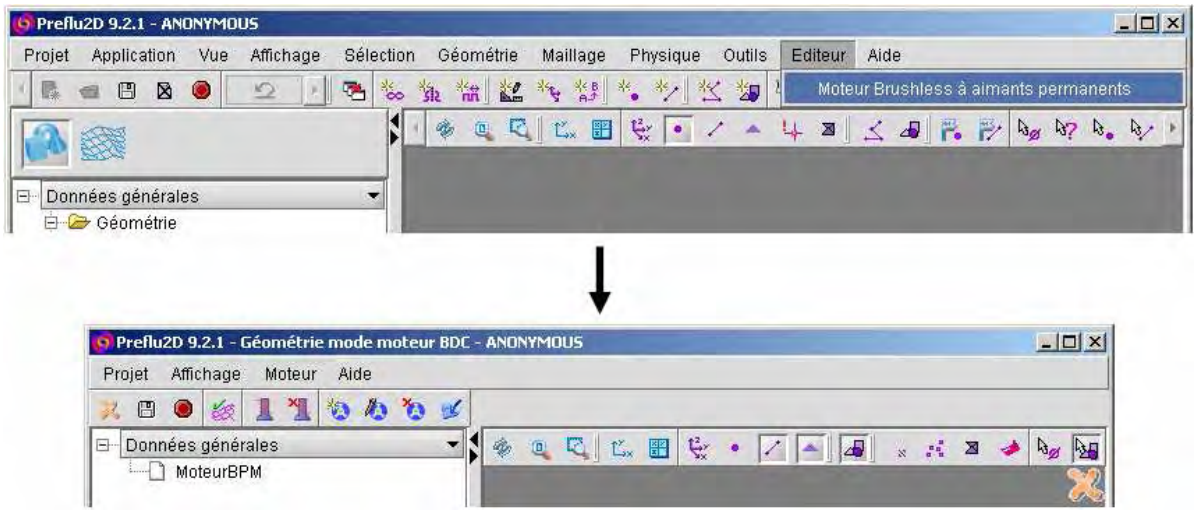


Figure 4.13 : Importation des objets moteurs dans le logiciel généraliste Flux2D

De plus, cette méthode permet aussi de charger dynamiquement les patrons en langage Python correspondant aux opérations de base des objets métiers. Ainsi, les méthodes de création, de modification et de suppression seront donc accessibles dans ce nouveau cas d'utilisation.

Ensuite, la sortie du cas d'utilisation FluxMotor entraîne la suppression de la base de données "méta" du modèle complet de l'application FluxMotor ainsi que le déchargement des patrons en langage Python des objets métiers créés. De retour dans l'environnement du logiciel généraliste Flux, l'utilisateur récupère les objets Flux (points, lignes, éléments du maillage, matériaux, …) créés par les algorithmes métiers.

4.4.3 Algorithmes spécifiques

Le logiciel FluxMotor est basé sur des algorithmes spécifiques de création de différents types de moteur. Ainsi, le savoir-faire du métier est intégré non seulement dans le modèle de

données des objets moteurs, mais aussi dans les algorithmes de création, de modification et de suppression écrits en langage Python. Ces algorithmes permettent à la fois la génération de la géométrie de nos objets moteurs, mais aussi l'affectation du maillage de manière automatique. D'autres algorithmes, comme l'importation de fichiers au format Speed [SPEED] ou la création et affectation du bobinage des moteurs, ont aussi été développés dans le but de rendre plus intuitif et plus facile l'utilisation des objets métiers moteurs.

Les algorithmes les plus spécifiques au métier de motoriste sont ceux qui permettent la création automatique de la géométrie et du maillage de nos objets métiers moteurs.

Pour cela, notre partenariat avec l'université de Glasgow nous a permis de bénéficier des connaissances des experts du laboratoire Speed. Grâce à eux, nous avons pu utiliser les paramètres définissant les différents types de moteurs à aimants permanents définit dans le logiciel Speed.

La bibliothèque d'objets moteurs que nous avons ainsi développée pour le logiciel FluxMotor propose un ensemble de quatre-vingt-trois rotors et vingt-six stators permettant de construire plus de deux mille moteurs à aimants permanents différents.

L'algorithme de création automatique de la géométrie de nos moteurs intègre aussi l'affectation automatique des discrétisations ponctuelles et linéiques. Le résultat obtenu est donc une géométrie de moteur entièrement maillée, le tout réalisé de manière automatique. Ces algorithmes développés par la société Cedrat sont tous intégrés dans les patrons permettant un formatage commun et donc une utilisation dynamique de nos objets métiers.

4.4.4 Evolutions et perspectives

Le logiciel FluxMotor a déjà reçu de nombreuses améliorations depuis sa création. Néanmoins, de nouvelles évolutions comme l'intégration de nouveaux types de moteurs, sont déjà à l'étude.

De plus, le but final des objets métiers moteurs est de rendre possible l'utilisation de ces objets métiers tout au long d'une étude Flux, depuis la création jusqu'au post-traitement, en passant par la physique et les études "métiers". Ces différentes études, tels que des essais à vide ou en court-circuit correspondent au cœur du métier de motoriste. Pour cela, le logiciel FluxMotor devra devenir totalement indépendant. Il correspondra alors réellement à un logiciel métier et n'utilisera plus que les algorithmes de calcul du module Flux. Son interface lui sera propre et pourra même devenir un logiciel de type "assistant" de conception de moteur.

4.5 Les logiciels Flux

Les logiciels Flux sont co-développés par la société Cedrat et le Laboratoire d'Electrotechnique de Grenoble. Ces applications généralistes de modélisation des phénomènes électromagnétiques par la méthode des éléments finis sont destinées à l'industrie du génie électrique. Elles sont utilisées de par le monde, en Europe, aux Etats Unis ou en Asie du Sud Est, par un millier de sociétés et d'universités. Ces logiciels de calcul par éléments finis constituent une référence mondiale dans le dimensionnement et l'analyse en deux ou trois dimensions de dispositifs électromagnétiques et thermiques, ces dispositifs pouvant être des moteurs électriques, des alternateurs ou encore des contacteurs, des actionneurs, des transformateurs …

La démarche de conception par modélisation que nous avons exposée dans les chapitres précédents a été mise en œuvre ces dernières années dans le but d'améliorer la présentation générale des logiciels Flux (voir figure 4.14).

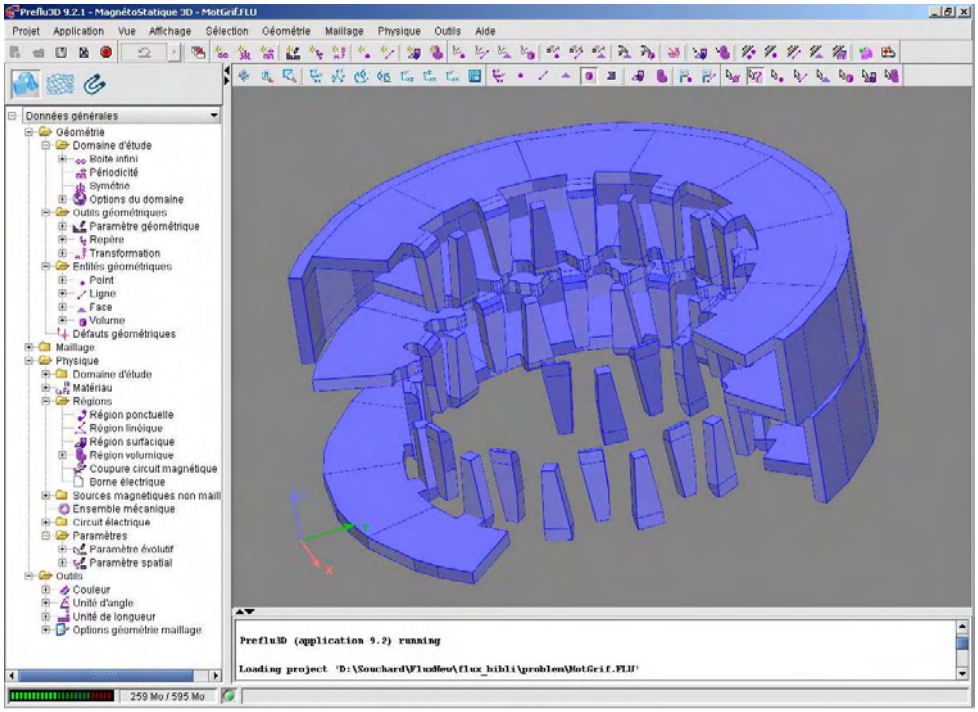


Figure 4.14 : Nouvelle interface des logiciels Flux

Grâce à la démarche proposée, la complexité de l'application est facilement. Ceci permet un développement plus rapide et plus structuré ainsi qu'une maintenance extrêmement réduite. La taille de l'application Flux (plus d'un millier de classes modélisées) a imposé un travail conséquent sur l'amélioration des performances de la machine virtuelle Application Modeling Language [AML].

4.5.1 Modèles des logiciels Flux

Les logiciels Flux ainsi mis au goût du jour comprennent plus d'un millier d'entités et plusieurs centaines de commandes associées.

Un grand nombre de cas d'utilisation ont aussi été décrits avec leurs contextes, arbres de représentation, barres d'outils, menus et vues graphiques correspondants. Tous ces cas d'utilisations couvrent une grande variété de domaines comme les études magnéto harmonique, magnéto statique ou magnéto transitoire, mais aussi la conduction électrique, la simulation thermique …

Tous les contextes ainsi décrits par le modèle des logiciels Flux correspondent aux différentes étapes que les utilisateurs doivent réaliser pour mener à bien leurs études. Les contextes géométrie et maillage permettent la construction géométrique du projet puis son maillage en vue de sa résolution. Le contexte physique sert à la définition des propriétés physiques et électriques du problème. Et enfin, les contextes résolution et exploitation permettent de résoudre le problème grâce aux différents algorithmes basés sur les éléments finis puis d'exploiter ces résultats dans différents formats de visualisation.

L'évolution du modèle de données des logiciels Flux nous a permis d'augmenter la structure et le nombre d'objets modélisables pour nos applications. De plus, nos interfaces homme machine sont désormais conformes aux standards créés de nos jours. Tout cela permet de pouvoir bénéficier d'une maintenance à moindre coût et d'une évolution grandement facilitée pour tous les aspects des logiciels Flux.

4.5.2 Evolutions et perspectives

Ces dernières années, les logiciels Flux ont subit une refonte complète visant à les repositionner dans les standards des logiciels développés de nos jours. Les évolutions apportées à nos logiciels ont touché à la fois la partie calcul, programmée en langage Fortran, mais aussi la partie interface avec l'utilisateur.

Le remodelage du noyau des applications Flux est réalisé grâce à la mise en pratique de notre démarche de conception de logiciel par modélisation et repose sur la machine virtuelle Application Modeling Language [AML]. Cette métamorphose s'est opérée par étapes dans les versions successives de nos logiciels réalisés en parallèle à ces travaux de thèse. Ainsi, la version 8.1 des logiciels Flux a intégré la modélisation des données et commandes de nos objets. Cela a permis de valider les algorithmes de génération automatiques de boîtes de

dialogue ainsi que celui d'interprétation automatique de ces boîtes. De plus, le langage Python est devenu, à partir de cette version, le langage de commande utilisé dans les logiciels Flux.

La version 9.1 a vu l'avènement de la modélisation des interfaces homme machine ainsi que des applications, de leurs cas d'utilisations et de leurs contextes. Tous les menus, barres d'outils, arbres de représentation et vues graphiques ont été décrits dans notre modèle d'application. Le contexte physique a été modélisé, cependant il reste encore à intégrer les contextes résolution et exploitation pour finaliser totalement la refonte des logiciels Flux. Le contexte résolution proposera non seulement le réglage des options de résolution mais aussi une analyse paramétrique. Cette dernière permettra de résoudre un problème plusieurs fois en faisant varier des paramètres géométriques, physiques ou du maillage. Le dernier contexte appelé exploitation permettra de visualiser les résultats de la résolution sous différents formats de représentation.

L'ajout de ces deux derniers contextes est prévu pour la sortie de la version 10. Celle-ci constituera le point final de cette mutation.

4.6 Conclusion

Le développement d'une plate-forme de conception de logiciel par modélisation a pour but de permettre la création de nombreuses applications dédiées aux métiers du génie électrique. Pour valider notre démarche, nous avons présenté plusieurs types de logiciels et d'objets métiers. Ce chapitre comporte un panel représentatif d'objets et logiciels développés grâce à la méthode de conception que nous avons exposée dans les chapitres précédents.

Pour commencer, nous avons décidé de présenter le logiciel TransfoBuilder. Ce logiciel dédié à la caractérisation et à la conception de transformateurs est un véritable logiciel métier construit entièrement grâce à notre approche. Le modèle de cette application à été saisi grâce au logiciel de description de modèle, FluxBuilder. L'application TransfoBuilder repose uniquement sur la machine virtuelle FluxCore pour s'exécuter. De plus, ce logiciel possède sa propre interface homme machine, il peut donc être totalement dédié au métier de concepteur de transformateurs.

Le deuxième exemple permettant de valider notre démarche est le logiciel InCa. Cette application est dédiée à l'analyse et à la détermination des inductances parasites dues au câblage. La réalisation du logiciel métier InCa nous a permis de tester notre démarche pour l'évolution d'un logiciel déjà existant. Dans ce cas de figure, nous avons pu constater le gain d'efficacité et de temps pour réaliser la migration de cette application.

Ensuite, nous avons souhaité exposer un autre type d'application que notre démarche permet de développer, à savoir les objets métiers. Afin d'expliquer ce nouveau type de conception, nous présentons le logiciel FluxMotor constitué par l'assemblage d'objets métiers moteurs et du logiciel généraliste Flux2D. Pour concevoir ces objets métiers, nous avons mis en place une nouvelle démarche de conception spécialement adaptée à la création d'objets métiers. Ainsi, tous les objets métiers conçus grâce à cette démarche seront compatibles et pourront être intégrés dans les logiciels généralistes Flux. De plus, le logiciel FluxMotor intègre des algorithmes spécifiques au métier de motoriste comme une aide à la création du bobinage de nos moteurs ainsi qu'une importation de fichiers au format Speed [SPEED].

La dernière réalisation que nous avons présentée n'est pas une application métier mais n'est autre que le logiciel généraliste Flux. Notre démarche de conception permet, en effet, d'optimiser la construction de nos logiciels métiers, mais tous les avantages de notre approche de conception peuvent aussi s'appliquer aux logiciels généralistes. Ainsi, la refonte des logiciels généralistes Flux entreprise ces dernières années est grandement facilitée par une architecture éprouvée. Ceci permet un développement plus rapide et plus structuré ainsi qu'une maintenance extrêmement réduite. De plus, la complexité des logiciels Flux a permis de valider totalement notre démarche de conception par modélisation.

Conclusion générale

Le nombre grandissant d'utilisateurs de logiciels de simulation, ainsi que la diversification de ses utilisateurs, constituent un nouvel environnement pour les logiciels de simulation. En effet, les utilisateurs ne sont plus nécessairement des experts. La demande évolue vers des logiciels spécialisés par métier ou par domaine d'application. Ce constat nous a entraîné à reconsidérer notre démarche de conception d'applications scientifiques pour l'adapter aux changements du marché des logiciels de simulation.

Durant ces trois dernières années, nous avons développé une démarche de conception par modélisation pour nos logiciels de simulation. Outre l'apport des techniques de Programmation Orientée Objet, les récentes évolutions dans le domaine du génie logiciel et de la méta modélisation contribuent beaucoup à la conception rapide et fiable d'applications complexes. Nous avons proposé une démarche de conception de logiciels originale utilisant ces dernières évolutions en les adaptant à nos contraintes.

Notre approche est basée sur la description des modèles de données en Unified Modeling Language complétée d'informations de présentation. Nous avons appelé le langage résultant l'Application Modeling Language.

Le modèle Application Modeling Language est rendu exécutable grâce à une machine virtuelle réutilisable. Ce programme intègre tous les services nécessaires à l'interprétation et à l'exécution du modèle du logiciel. L'avantage principal de cette machine virtuelle est que tous les services développés sont réutilisables : ils ne dépendent pas d'un modèle de données spécifique et sont disponibles dans tous les logiciels développés selon l'approche de modélisation.

Cette construction d'applications, via un modèle, présente une excellente solution en terme de temps de développement et de modularité. Elle permet en effet d'offrir rapidement, sur la base de logiciels généralistes, un grand nombre de logiciels de simulation numérique spécialisés, plus faciles d'accès et adaptés à un type précis d'utilisation. Le gain obtenu est double :

- En temps tout d'abord, car la modification du modèle de données est beaucoup plus rapide qu'une programmation classique.
- En coût enfin car seules la saisie du modèle et l'implémentation des algorithmes sont effectivement réalisées, le reste des services étant factorisé dans un interpréteur générique. Quant à la maintenance, elle est elle aussi grandement facilitée.

L’ensemble de notre démarche a été validé par la refonte complète de deux logiciels de simulations numériques ainsi que par la création de deux exemples concrets de logiciels métiers.

Perspectives

Le travail présenté dans ce manuscrit est inscrit dans une démarche globale de conception de logiciels par modélisation. Dans un futur proche, de nombreuses améliorations devraient permettre d'améliorer les différents modèles de nos logiciels ainsi que la machine virtuelle grâce à l'intégration de nouveaux composants.

A court terme, les améliorations attendues pour nos modèles d'application sont les suivantes :

- La conversion du mode de "persistance fichier" de notre modèle de données du langage STandard for the Exchange of Product model data [STEP] au langage eXtensible Markup Language [XML]. Cette perspective de refonte du module base de données a pour but de permettre la collaboration entre bases de données pour la gestion de nos futurs nombreux objets métiers. Notre langage de modélisation devra lui aussi évoluer pour devenir compatible avec ce concept de "modèle composite". Des réflexions ont déjà été menées et le langage eXtensible Markup Language [XML] semble être le plus propice à nos besoins.
- L'amélioration de la modélisation dynamique. Il faudra d'une part éprouver le modèle de "workflow" sur des applications de bonne taille et d'autre part compléter le formalisme utilisé par l'introduction de modèles de type "état transition" en particulier, pour automatiser le masquage dynamique d'actions en fonction du contexte (menu grisé si aucun fichier n'est ouvert par exemple).
- L'ajout de règles pour les objets du modèle de données. Le formalisme Object Contraint Language [OCL], développé par l'Object Management Group [OMG] et intégré à l'Unified Modeling Language [UML], permet de formaliser l'expression de contrainte. Intégrée dans notre démarche, cette nouvelle notion nous permettra de gérer dynamiquement toutes les contraintes liant nos objets.
- La modélisation des tests dans le modèle de l'application. A partir du modèle de l'application, il doit être possible de déterminer une première liste de tests à réaliser en conformité avec le plan d'assurance qualité.

Dans le domaine du développement de la machine virtuelle, les améliorations que nous envisageons sont :

- La mise en place d'une modélisation et d'un affichage de type Wizard. Ce nouveau type d'application pourra permettre de créer des logiciels guidant l'utilisateur pas à pas dans la démarche de réalisation de son étude. Cette évolution est très liée à la modélisation dynamique de nos applications. En effet, le développement d'application de type Wizard nécessite de pouvoir modéliser les enchaînements et les comportements des fenêtres du logiciel.
- La réalisation de la version finale des logiciels Flux (version 10). La dernière étape du développement des logiciels Flux est l'intégration des contextes résolutions et exploitations des résultats. Dans cette dernière étape, la société Cedrat compte développer de nouveaux composants basés sur une représentation très synthétique des données. La création de tels composants graphiques perfectionnés pour la phase d'exploitation permettra de simplifier et de clarifier la présentation des résultats aux utilisateurs.

La majorité de ces évolutions s'inscrit dans la phase de construction massive de logiciels métiers. Des études de marché et de besoins utilisateur devront nous guider dans nos choix de développement de logiciels spécialisés. Après les métiers de motoriste et de concepteur de transformateur, les métiers du génie électrique ciblés pourraient être la conception de contacteur, l'étude de Micro Electro Mechanical Systems [MEMS] ou encore le contrôle non destructif.

Notre démarche devra perpétuellement s'adapter aux différentes évolutions autant commerciales que techniques. La veille technologique sur les domaines du génie logiciel et du génie informatique permettront d'apporter des améliorations correspondant aux nouveaux besoins des utilisateurs de logiciels de simulation.

Liste des publications

[SOUCHARD-04] Y. SOUCHARD, G. JEROME, Y. MARECHAL, "Benefits of Model Driven Architecture for simulation software design", Conference on Electromagnetcic Field Computation, CEFC 2004, Seoul, South Korea, June 2004.

[LACOMBE-05] G. LACOMBE, Y. SOUCHARD, C. CHANTREL, E. LECATHELINAIS, G. JEROME, X. BRUNOTTE, Y. MARECHAL, J. WALLACE, D. DORELL, "Benefits of Model Driven Architecture for simulation software design, application for dedicated motor tool", Conference on the Computation of Electromagnetic Fields, Compumag 2005, Shenyang, Liaoning, China, June 2005.

[MARECHAL-05] Y. MARECHAL, Y. SOUCHARD, G. JEROME, "Adaptive Object Models Architecture for simulation software design", International Symposium on Electromagnetic Fields in Mechatronics, Electrical and Electronic Engineering, ISEF 2005, Baiona, Spain, September 2005.

Bibliographie

[ACHARYA-04] Sharad ACHARYA, "Data Driven Presentation Layer for Dynamic and Configurable Web Systems", Pattern Languages of Programs, PLoP 2004, Monticello, Illinois, USA, September 2004.

[ALLARD-05] Bertrand ALLARD, "Création d'un pré processeur de géométrie 3D complexe pour Flux3D. Développement d'un outil métier de modélisation de transformateurs", Master de Recherche, Institut National Polytechnique de Grenoble, 2005.

[AOM] Adaptive Object Model, http://www.adaptiveobjectmodel.com

[BELAUNDE-99] Mariano BELAUNDE, "A Pragmatic Approach for Building a User-friendly and Flexible UML Model Repository", The Unified Modeling Language - Beyond the Standard, Second International Conference, UML'99, Fort Collins, Colorado, USA, October 1999

[CHANTREL-03] Cédric CHANTREL, "Etude et réalisation d'un générateur d'objets métiers pour Flux. Application aux moteurs", Master de Recherche, Institut National Polytechnique de Grenoble, 2003.

[CLAVEL-96] Edith CLAVEL, "Vers un outil de conception de câblage : le logiciel INCA", Thèse de doctorat, Institut National Polythechnique de Grenoble, 1996.

[DELCROIX-04] Jean-Baptiste DELCROIX, "Création d'un pré processeur de géométrie 3D complexe pour Flux3D. Développement d'un outil métier de modélisation de transformateurs", Master de Recherche, Institut National Polytechnique de Grenoble, 2004.

[DULAR-98] Patrick DULAR, W. LEGROS, A. NICOLET, "Coupling of local and Global Quantities in various Finite Element Formulations And its Application to Electrostatics, Magnetostatics and Magnetodynamics", IEEE Transaction on Magnetics, Vol. 34, No. 5, September 1998.

[DULAR-05] Patrick DULAR, Christophe GEUZAINE, GetDp : A General Environment for the Treatment of Discrete Problems, 2005.

[FLUX] Flux, a finite element simulation environment for Electrical Engineering, http://www.cedrat.com

[GUICHON-01] Jean-Michel GUICHON, "Modélisation, caractérisation et dimensionnement de jeux de barres", Thèse de doctorat, Institut National Polythechnique de Grenoble, 2001.

[INCA] InCa, a finite element simulation environment for inductance calculation, http://www.cedrat.com

[JACOBSON-93] Ivar JACOBSON, "Génie Logiciel Orienté Objet", Addison-Wesley France, 1993.

[JAVABEANS] Java Beans, http://java.sun.com/products/javabeans

[JEANNIARD-03] Luc JEANNIARD, "Création d'un pré processeur de géométrie 3D complexe pour Flux3D. Développement d'un outil métier de modélisation de transformateurs", Diplôme d'étude approfondie, Institut National Polytechnique de Grenoble, 2003.

[KHALED-04] Osama Mabrouk KHALED, Hoda M. HOSNY, "A pattern language for Developing Database-Driven Applications", Pattern Languages of Programs, PLoP 2004, Monticello, Illinois, USA, September 2004.

[LACOMBE-03] Guillaume LACOMBE, "Pilotage à distance des logiciels Flux", Master de Recherche, Institut National Polytechnique de Grenoble, 2003.

[MA-01] Singva MA, "Définition d'un protocole d'application STEP pour la Simalution en Electromagnétisme", Thèse de doctorat, Institut National Polythechnique de Grenoble, 2001.

[MANOLESCU-01] Dragos MANOLESCU, "Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development", Ph.D. Thesis and Computer Science Technical Report, University of Illinois at Urbana-Champaign, Urbana, Illinois, 2001.

[MARECHAL-01] Yves MARECHAL, "Vers une nouvelle génération de logiciels de simulations pour l'électromagnétisme et les disciplines connexes", Habilitation à Diriger des Recherches, Institut National Polythechnique de Grenoble, 2001.

[MAI-00] W. MAI, G HENNEBERGER, "Object-Oriented Design of Finite Element Calculations with Respect to Coupled Problems", Institute of Electrical and Electronics Engineers, IEEE Transactions on Magnetics, vol. 36, no. 4, July 2000.

[MDA] Model Driven Architecture, http://www.omg.org/mda

[MOF] Meta Object Facilities, V1.4, http://www.omg.org/mof

[MULLER-00] Pierre-Alain MULLER, Nathalie GAERTNER, "Modélisation objet avec UML", Eyrolles, 2000.

[OBJECTDIRECT] ObjectDirect company, http://www.objetdirect.com

[OMG] Object Management Group, http://www.omg.org

[POOLE-00] John D. POOLE, "The Common Warehouse Metamodel as a Foundation for Active Object Models in the Data Warehouse Environment", OMG CWM Working Group, April 2000.

[RIEHLE-00] Dirk RIEHLE, Michel TILMAN, Raph JOHNSON, "Dynamic Object Model", Pattern Languages of Programs, PLoP 2000, Monticello, Illinois, USA, August 2000.

[SCHANEN-94] Jean-Luc SCHANEN, "Intégration de la Compatibilité Electromagnétique dans la conception de convertisseurs en Electronique de Puissance", Thèse de doctorat, Institut National Polythechnique de Grenoble, 1994.

[SILVA-94] E. SILVA, R. MESQUITA, R. SALDANHA, P. PALMEIRA, "An object-oriented finite-element program for electromagnetic field computation", Institute of Electrical and Electronics Engineers, IEEE Transactions on Magnetics, vol. 30, no. 5, September 1994.

[SPEED] Scottish Power Electronics and Electric Drives consortium, software for electric motors and drives, Speed Laboratory, Glasgow, Scotland, http://www.speedlab.co.uk

[UML] Unified Modeling Language, V2.0, http://www.uml.org

[YODER-00] Joseph W. YODER, Reza RAZAVI, "Adaptive Object-Models", Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2000, Minneapolis, Minnesota, USA, October 2000.

[YODER-01] Joseph W. YODER, Federico BALAGUER, Ralph JOHNSON. "Architecture and Design of Adaptive Object-Models", Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa Bay, Florida, USA, December 2001.

Glossaire

- **Adaptive Object Model [AOM] :** L'Adaptive Object Model est un système qui représente les classes, les attributs, et les relations en tant que méta données. Ce modèle est basé sur les instances plutôt que sur les classes. De ce fait, quand les utilisateurs modifient des méta données pour refléter des changements d'état, ils modifient le comportement du système.
- **Application Modeling Language [AML] :** L'application Modeling Language est un langage de modélisation dérivé de l'Unified Modeling Language [UML]. Il comporte en plus des notions déjà présentes dans l'Unified Modeling Language [UML], des concepts de modélisation des informations de type interface homme machine d'un système. Ainsi, ce langage permet de décrire les informations de type interface utilisateurs des données d'une application ainsi que son interface homme machine.
- **Génie Logiciel :** Le génie logiciel est l'ensemble des activités de conception et de mise en oeuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi. Autrement dit, le génie logiciel est l'art de produire de bons logiciels au meilleur rapport qualité prix.
- **Graphical User Interface [GUI] :** La Graphical User Interface est une interface qui profite des capacités graphiques des ordinateurs actuels afin de rendre l'utilisation d'un logiciel la plus facile possible. En effet, une interface utilisateur graphique bien conçue peut permettre un meilleur apprentissage de langage de commande complexe.
- **Meta Object Facility [MOF] :** Reconnu comme standard de méta modélisation, le Meta Object Facility est un ensemble d'interfaces standards permettant de définir et de modifier des méta modèles et leurs modèles correspondants. C'est un moyen de définir la syntaxe et la sémantique d'un langage de modélisation. Il a été créé par l'Object Management Group [OMG] afin de définir la notation Unified Modeling Language [UML], par exemple.
- **Micro Electro Mechanical Systems [MEMS] :** Le terme Micro Electro Mechanical Systems correspond à l'intégration d'éléments mécaniques, des capteurs, des mécanismes de positionnement, des composants électroniques, sur un même substrat en silicone, par le biais de techniques de mico-fabrication.
- **Model Driven Architecture [MDA] :** Le Model Driven Architecture est comme son nom l'indique une architecture basée sur les modèles. Cette norme proposée et soutenue par

l'OMG permet une conception structurée de logiciels. L'idée fondamentale est que les fonctionnalités du système à développer sont définies dans un modèle indépendant en utilisant un langage de spécification approprié, puis traduites dans un ou plusieurs modèles spécifiques pour l'implémentation concrète du système. L'utilisation de ce concept se fait normalement à l'aide d'outils automatisés.

- **Object Contraint Language [OCL] :** Ce standard est intégré à l'Unified Modeling Language [UML] version 2.0. Il permet de formaliser l'expression des contraintes. Ce concept permet de régler des pré et post-contraintes, des conditions invariantes et autres.
- **Object Management Group [OMG] :** L'Object Management Group est un consortium de l'industrie de l'informatique entre plus de 850 membres. Il a été créé en 1989 dans l'objectif de standardiser et promouvoir le modèle objet sous toutes ses formes. Avec des standards reconnus sur l'ensemble du secteur des logiciels, l'Object Management Group permet une approche complète du cycle de vie, des solutions allant jusqu'à l'intégration en entreprise. Basés sur le Model Driven Architecture [MDA] et la Meta Object Facility [MOF], les standards de l'Object Management Group couvrent à la fois la conception des applications et leur implémentation. Ces Standards incluent par exemple l'Unified Modeling Language [UML], l'eXtensible Markup Language [XML], l'Object Contraint Language [OCL], …
- **Programmation Orientée Objet [POO] :** La Programmation Orientée Objet est aujourd'hui la méthode de développement d'application la plus utilisée dans le monde. Ses points forts sont la réutilisabilité, la modularité et l'extensibilité, là où la programmation impérative comporte le risque d'enfermer le code sur lui-même. Le langage Java est un exemple de langage orienté objet qui permet un gain de simplicité et de productivité.
- **Python [PYTHON] :** Python est un langage de programmation interprété. Il autorise la programmation impérative structurée, orientée objet, et fonctionnelle. Il est doté d'un typage dynamique, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions.
- **Remote Method Invocation [RMI] :** La Remote Method Invocation est une interface de programmation en langage Java permettant de manipuler des objets distants de manière transparente pour l'utilisateur, c'est-à-dire de la même façon que si l'objet était sur son poste local. Ainsi, un serveur permet à un client d'invoquer des méthodes à distance. Deux machines virtuelles sont donc nécessaires, une sur un poste serveur et une sur un poste client et l'ensemble des communications se fait en langage Java.

- **Standard Data Access Interface [SDAI] :** La Standard Data Access Interface est une interface de dialogue avec une base de données. Ce concept représente la norme pour l'accès des données au format Standard for the Exchange of Product model data [STEP]. Le principal avantage de ce standard est qu'il n'est pas adhérant à la mise en oeuvre matérielle de la structure de données ou à la technologie de stockage.
- **STandard for the Exchange of Product model data [STEP] :** Le STandard for the Exchange of Product model data est un standard de représentation et d'échange de données. Il a pour objectif d'intégrer les processus de conception, de développement, de fabrication et de maintenance d'un logiciel. Cette notion de conception permet donc de définir une représentation des données d'une application, interprétable par tout système informatique, et couvrant tout le cycle de vie de cette application. La mise en place de cette norme implique la définition d'un format neutre, interprétable par tout système informatique et indépendamment du système ayant généré les données.
- **Textual User Interface [TUI] :** La Textual User Interface est un concept qui a été inventé en même temps que la Graphical User Interface [GUI]. Cette notion est différente des interfaces de type ligne de commande car, comme la Graphical User Interface [GUI], elle utilise l'écran entier et ne fournit pas nécessairement de sortie ligne par ligne. Cependant, la Textual User Interface utilise seulement du texte et des symboles disponibles sur une console texte classique, alors que la Graphical User Interface [GUI] utilise uniquement des composants graphiques.
- **Unified Modeling Language [UML] :** L'Unified Modeling Language est un langage visuel permettant de modéliser des besoins d'un système à l'aide de diagrammes et de textes. Il permet aussi de décrire des architectures, des solutions ou des points de vue. Le langage Unified Modeling Language se présente actuellement comme le standard de spécification, de construction et de documentation de modèles Model Driven Architecture [MDA].
- **XML Metadata Interchange [XMI] :** Ce concept est un standard d'échange de données Unified Modeling Language [UML] basé sur eXtensible Markup Language [XML].Ce standard, créé par l'Object Management Group [OMG], est un procédé de sérialisation d'objets Meta Object Facility [MOF] sous la forme eXtensible Markup Language [XML].
- **eXtensible Markup Language [XML] :** L'eXtensible Markup Language est un méta langage permettant de décrire un modèle. Ce langage est un standard qui sert de base pour la création d'autre langage balisé spécialisé : c'est un méta langage.

- **XML Schema Description [XSD] :** Le XML Schema Description est un langage de description de format de document eXtensible Markup Language [XML] permettant de définir la structure d’un document eXtensible Markup Language [XML]. La connaissance de la structure d'un document XML permet notamment de vérifier la validité de ce document. Un fichier XML Schema Description est donc lui-même un document eXtensible Markup Language [XML]. Ce langage de description de contenus de documents eXtensible Markup Language [XML] est lui-même défini par un schéma, dont les balises de définition s'auto définissent (c'est un exemple de définition récursive).
- **eXtensible Stylesheet Language [XSL] :** L’eXtensible Stylesheet Language est le langage de description de feuilles de style associé à Extensible Markup Language [XML]. Une feuille de style eXtensible Stylesheet Language est un fichier qui décrit comment doivent être présentés (affichés, imprimés, ...) les documents XML basé sur un même XML Schema Description [XSD].

Résumé :

Les travaux présentés dans ce manuscrit ont pour objectif la conception d'une plate-forme de conception de logiciels métiers dérivés des logiciels Flux. Ces recherches ont été menées en partenariat avec la société Cedrat. Les logiciels de simulation sont de plus en plus complexes et donc nécessitent une conception de plus en plus structurée. C'est pourquoi les développeurs s'appuient sur de nouveaux concepts développés dans le domaine du génie logiciel. L'exploration de ces idées a permis d'évaluer les avancées possibles tout en tenant compte de nos contraintes. Ainsi, ce manuscrit présente les trois grandes phases de notre démarche de conception : La modélisation de nos applications grâce à de nouveaux concepts de méta modélisation, la génération du code et l'exécution de l'application grâce à l'interprétation des modèles créés. La présentation d'exemples validant notre démarche fait l'objet de la dernière partie de ce manuscrit. La description dynamique de nos applications via un modèle de données présente une solution en terme de temps de développement et de modularité. Elle nous permet d'offrir rapidement, sur la base du logiciel généraliste Flux, un ensemble de logiciels de simulation numérique spécialisés par métier plus facile d'accès et adapté aux utilisateurs.

Mots clés :

Logiciel Métier, Modélisation Objet, Méta Modèles, Model Driven Architecture, Simulation Numérique, Eléments Finis.

Abstract:

This paper presents the development of a meta level programming technology devoted to simulation software. Besides the Object Oriented Programming paradigm, this recent evolution of the computer science domain has proved to bring several benefits and allow rapid and robust application design. Our approach relies on an augmented Unified Modeling Language description of the application's component model and a reusable framework. This approach provides a high level of dynamicity. Hence, the customisation of general purpose software and even its encapsulation inside a dedicated simulator is very fast and requires only little programming and maintenance. For the end user, resulting software exhibits no difference compared to hard coded tailor software, and the benefits are the same : few inputs, high level and devoted commands, vocabulary and simulation process adapted to the user habits. The last part of this paper is devoted to some realisations which validate our approach of software developing.

Keywords:

Dedicated Tools, Object Modelling, Meta Models, Model Driven Architecture, Numerical Simulation, Finite Elements.