Traitement unifié des propriétés physiques dans unenvironnement d'analyse intégré

Stefan Giurgea

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
dans le cadre de **l'Ecole Doctorale « Electronique, Electrotechnique, Automatique, Télécommunications, Signal »**

présentée et soutenue publiquement

par

Stefan GIURGEA

le 17 Decembre 2003

Titre :

Traitement unifié des propriétés physiques dans un environnement d'analyse intégré

Directeurs de thèse :

Yves Maréchal
Thierry Chevalier

JURY

M. Mouloud Feliachi	, Rapporteur
M. Patrick Dular	, Rapporteur
M. Bernard Secher	, Examinateur
M. J. L. COULOMB	, Examinateur
M. Y. MARECHAL	, Directeur de thèse
M. T. CHEVALIER	, Directeur de thèse

Remerciements

Je tiens à remercier Yves Maréchal, Professeur à l'Institut National Polytechnique de Grenoble qui a été mon directeur de thèse durant ces trois années. J'ai été toujours impressionné par sa capacité d'abstraction. Il a toujours su être disponible pour faire progresser notre démarche scientifique, et la rendre accessible.

Je tiens aussi à remercier à Thierry Chevalier, Maître de conférences à l'ENSIEG, qui a co-encadré ce travail de thèse. Je le remercie tout particulièrement pour sa capacité de travail en équipe et pour sa grande disponibilité.

Je remercie vivement Monsieur le Professeur Jean Louis Coulomb de l'INPG, pour la confiance qu'il a placé en moi, en m'accueillant au sein de l'équipe Modélisation et pour m'avoir proposé ce sujet de thèse. Je le remercie pour ses conseils qui m'ont été fort utiles dans l'avancement de cette thèse.

Mes remerciements s'adressent également à :

Monsieur Mouloud Feliachi, professeur à l'Université de Saint Nazaire pour l'honneur qu'il m'a fait en acceptant de présider le jury de thèse.

Patrick Dular, chercheur Qualifié F.N.R.S. à l'Université de Liége pour avoir jugé ce travail en qualité de rapporteur du jury. J'ai pu également apprécier la qualité de ses remarques constructives lors de la correction du ce manuscrit.

Bernard Sécher, ingénieur de recherche au CEA Saclay, pour avoir participé au jury de thèse, mais aussi pour ses suggestions dans le cadre du projet SALOME.

Je remercie les participants du Projet SALOME avec lesquels j'ai eu de nombreux échanges scientifiques, en particulier Paul Rascale, Yves Fricaud, Nicholas Rejneri, Emmanuel Dorlet, Hubert Rolland.

Monsieur Patrice Labie pour son aide, mais aussi pour sa bonne humeur.

Madame Etienette Callegher, et Patrick Eustache, responsables du réseau informatique, pour leur grande disponibilité et sympathie.

Je remercie Gérard Meunier, Jean-Paul Ferrieux, Robert Perret, pour leurs conseils et pour leur disponibilité.

Je remercie également à tous mes professeurs de l'Université Polytechnique de Bucarest, en particulier Monsieur Constantin Ghita pour m'avoir initié au monde de la recherche.

Ma gratitude s'adresse aussi à ceux qui ont été tour à tour mes collègues dans la salle « Modélisation » : Mauricio Caldora Costa, Afef Slama, Singva Ma, Olivier Defour, Iszabela Klutsch, FleurJanet et Michael Joan, qui m'ont aidé à répondre à un certain nombre de questions pratiques, techniques, ou scientifiques, que je me suis posé. Je remercie aussi a tous mes collègues qui m'ont aide à un moment ou autre de la thèse : Gilles David, David Frey, Sebastien Gréhant, et à mes compatriotes : Catalin et Monika Ciocoiu, Cristina Pop, Ioana Fagarasan, Daniel Moraru, Adi Manescu, Constantin Surdu.

Enfin, je tiens à remercier tous ceux qui m'ont soutenu, et plus particulièrement ma femme Coralia qui a toujours su m'encourager dans les moments de stress, mes parents Ion et Rodica qui m'ont appris des principes dont je me servirai toute ma vie, et mes amis qui m'ont offert leurs conseils et sympathie.

1 Contexte et problématique

1.1 Le contexte

Une des préoccupations actuelles dans le domaine de la construction des dispositifs industriels est d'augmenter la rapidité de conception de nouveaux produits. Dans ce contexte, la modélisation des phénomènes physiques a comme objectif de déterminer la valeur et la variation des grandeurs physiques caractéristiques en fonction des propriétés du milieu où ces grandeurs sont définies et de la géométrie. Actuellement pour réaliser cette analyse, de nombreux logiciels de simulation numérique utilisent la méthode des éléments finis.
La modélisation est profitable, soit directement, pour tester numériquement les caractéristiques des dispositifs en cours de développement, soit indirectement, pour réaliser une optimisation automatique des paramètres qui caractérisent un dispositif en vue d'obtenir la meilleure réponse à un cahier des charges [Costa], [Cox].

Examinons les contraintes que doivent satisfaire ces logiciels de simulation.

1.1.1 Les besoins en couverture des disciplines de la physique

De nos jours, en milieu industriel, les besoins en simulation concernent généralement plusieurs disciplines de la physique à la fois : mécanique des structures et des fluides, électromagnétisme, électrique, thermodynamique, acoustique, nucléaire, biologie, ... Dans le domaine du Génie Electrique en particulier, les phénomènes thermiques, électromagnétiques, mécaniques et acoustiques sont généralement couplés.
La modélisation des problèmes multi-physiques constitue un enjeu très important dans l'avancement des techniques de conception en milieu industriel. Malheureusement, actuellement le concepteur (ou les concepteurs) utilisent généralement des outils différents en fonction de l'étude à mener, du produit à concevoir et aussi de leurs habitudes et savoir-faire.
Si pour chaque discipline, les solveurs de simulation numérique sont très performants dans leurs calculs, ils sont aussi souvent faiblement ouverts sur l'extérieur. Les programmeurs doivent toujours intervenir pour faire communiquer des logiciels distincts et les faire échanger des données.

1.1.2 Les besoins en méthodes de résolution

La méthode des éléments finis, si elle est communément admise comme étant multi-disciplinaire, n'est cependant pas toujours la mieux adaptée. En fonction du problème et des conditions de calcul, des méthodes plus spécialisées peuvent souvent s'appliquer avec parfois beaucoup de bénéfices en coût et/ou en précision. Pour atteindre cette hybridation des modèles numériques, une architecture multi-solveur doit être mise en place, afin de permettre d'utiliser différents solveurs travaillant avec des structures de données différentes [Smirnov].

1.1.3 La prise en compte des métiers et des niveaux d'utilisation

Pour un même solveur, son utilisation diffère en fonction du problème à traiter, des résultats souhaités, des contraintes industrielles, des contraintes de fabrication.
Par ailleurs, le niveau de compétence des utilisateurs est lui-même très variable, allant des experts aux utilisateurs occasionnels. L'utilisation des logiciels de simulation peut relever des objectifs très différents : comportement de méthodes pour l'utilisateur numéricien, construction de modèles pour l'utilisateur physicien, conception de dispositifs pour le chercheur ou le bureau d'études.

Tous ces profils d'utilisateurs et d'utilisations demandent des adaptations des logiciels de simulation à leurs spécificités.

1.2 Le concept de plate-forme d'intégration

Les logiciels de simulation numérique actuels disposent parfois de capacité de traitements multi physiques. Ils peuvent ainsi résoudre simultanément plusieurs aspects (par exemple, thermo-mécanique, hydro-mécanique, thermo-fluidique, etc), qui ont été prévus et réalisés par l'équipe de développement du code.
Le développement d'un nouveau couplage nécessiterait actuellement la mise en œuvre d'un nouveau module logiciel dédié ou des changements considérables dans l'environnement logiciel existant.

Le principal problème de la création d'un tel logiciel est le coût initial d'investissement et le coût de la gestion, car il devrait suivre les évolutions de ces domaines. D'autre part, la nécessité d'un tel logiciel est indiscutable - la résolution des problèmes multi-physiques nécessite une démarche systémique, mais l'interopérabilité des solveurs existants est très faible.

La construction d'un logiciel multi-physique générique ne peut pas être faite d'une manière « compacte et homogène » pour tous les domaines de la physique: c'est-à-dire de créer un seul logiciel concernant tous les domaines de la physique.

Une solution serait la construction d'un logiciel de type plate-forme qui pourrait incorporer les différents solveurs, donc une construction « intégrée ». Il aurait comme composants différents solveurs et utiliserait leurs fonctions critiques.

1.2.1 Mise en oeuvre

Pratiquement une plate-forme est construite par plusieurs composants logiciels, réalisés de manière à permettre à chacun d'intégrer son propre composant avec le minimum de code possible.

Une plate-forme dédiée à la simulation des problèmes multi-physiques, permettrait l'intégration des solveurs et des outils de discrétisation géométrique, dédiés pour chaque domaine étudié. Une telle plate-forme nécessite aussi la spécification des propriétés physiques pour les domaines donnés. Des composants pour offrir les services non critiques (description géométrique, visualisation, etc.) peuvent être fourni par la plate-forme.

L'utilisation de la plate-forme comporte deux actions distinctes :

- L'intégration du solveur dans l'environnement, qui est réalisée une fois pour chaque solveur. Cela est réalisé par un utilisateur intégrateur.
- L'utilisation effective de la plate-forme, par un utilisateur final. L'utilisateur doit décrire complètement un problème physique, le résoudre à l'aide du solveur intégré et analyser les résultats.

1.2.2 Les avantages d'une construction « intégrée »

Comme il s'agit d'un environnement logiciel qui peut intégrer plusieurs solveurs dédiés aux différents domaines de la physique, les coûts d'investissement pour intégrer des solveurs déjà existants à cet environnement logiciel sont nettement inférieurs à ceux supposés par un projet de création d'un nouveau solveur adapté au couplage. On utilise des ressources humaines, intellectuelles et matérielles à coût zéro.

Une diminution des coûts au moment de la création ou de l'amélioration des solveurs dédiés est obtenu, car on peut se concentrer sur les fonctions critiques du solveur. Il s'agit des coûts de qualité, comme des coûts financiers.
Elle permettra aussi une valorisation des solveurs, car une fois intégrés dans le contexte multi physique on augmente leur intérêt.

1.2.3 Les contraintes, les conditions

L'outil doit être générique pour permettre l'intégration de tout solveur de simulation numérique pour la résolution des problèmes physiques.

L'intégration de tout solveur n'est pas suffisante, car l'utilisateur final exige une interface de travail commune, du point de vue fonctionnel et structurel. En conséquent, il est nécessaire de disposer d'un modèle de données unifié et aussi d'un environnement logiciel générique. Le logiciel d'intégration devrait être utilisé par des spécialistes de plusieurs domaines de la physique, donc il doit offrir une interface facile à utiliser.

L'amélioration de l'exécution implique la création des simulations "systémiques" interdisciplinaires. Pour réaliser ceci, les groupes de codes doivent facilement être liés ensemble et partager un modèle commun de base de données.

La simulation systémique impose des rendements élevés : le codage doit exploiter les dispositifs de calcul parallèle.

On doit tenir compte d'intégration des nouveaux types de données. L'outil doit permettre de améliorer et d'actualiser les solveurs, le logiciel, le code - aux coûts de qualité et coûts financiers minimum. Donc les composants doivent avoir un grade élevé d'indépendance.

1.3 Le projet SALOME

Pour répondre aux exigences et aux besoins présentés ci-dessus, il a été initié le projet SALOME [Salome] (Simulation numérique par Architecture Logicielle en Open-source et à Méthodologie d'Evolution).

SALOME est une plate-forme Open-source conçue pour offrir un environnement global afin de résoudre des modèles numériques multi-physiques, et de favoriser la liaison CAO-Calcul. Les éléments non-critiques d'une application de Conception Assisté par Ordinateur (CAO) et simulation sont groupés sur cette plate-forme générique. La plate-forme fournit les éléments de description de données physiques, d'une manière unifiée et les interfaces nécessaires pour la description des modèles physiques et d'exploitation des résultats (pré et post processing).

1.3.1 Les objectifs de la plate-forme [Toumi]

1. **Flexibilité** - La plate-forme doit permettre une adaptation rapide et à faible coût, pour réaliser l'analyse des nouveaux types de problèmes. L'utilisateur intégrateur doit avoir accès facile à tous les paramètres de la modélisation pour créer les outils spécifiques aux domaines spécifiques.

2. **Productivité**: l'exécution du code doit être simple pour l'utilisateur et la réutilisation des composants doit être facile.

3. **Performance**: En permettant une exécution parallèle sur plusieurs machines qui communiquent en réseau, les nouvelles applications doivent être en mesure à simuler plus finement et plus rapidement des phénomènes complexes et dans des conditions de couplage physique.

4. **Evolutivité**: d'une part, les technologies logicielles et les architectures physiques évoluent rapidement par comparaison au temps d'élaboration d'une application scientifique et de sa validation tandis que d'autre part, un modèle de données adapté à la totalité d'échanges entre les composants peut progresser dans le temps. La plate-forme doit pouvoir suivre facilement ces développements.

1.3.2 Une plate-forme Open-source

Les initiateurs du projet SALOME ont décidé de réaliser la plate-forme sous licence « Open-source » (logiciel libre). Le concept Open-source est fondé sur un développement collectif du logiciel, à travers Internet.

L'utilisation des logiciels Open-source et la création d'une plate-forme Open-source a comme principaux avantages [Open] :

- La réutilisation des composants spécifiques Open-source qui ont été largement introduits dans la communauté informatique (Open Cascade, Corba, UML....).
- L'attraction d'une communauté large intéressée dans l'utilisation et le perfectionnement du logiciel, ce qui implique une évolution continue basée sur la contribution volontaire de ceux qui utilisent la plate-forme.

- Pour les producteurs des solveurs, un tel logiciel open-source leurs permet d'avoir accès à une technologie très avancée avec un coût moindre
- Offrir aux producteurs de solveurs la possibilité d'intégrer leurs logiciels dans le cadre de la plate-forme avec des coûts minimums (temps, ressources humaines, code et ressources financiers bien sur)
- Pour l'industrie, la plate-forme représente un outil qui est validé en permanence
- Une large utilisation de la plate-forme

1.3.3 Un projet fondé sur les techniques logicielles les plus récentes

Dans l'univers Open-source, on peut trouver des outils en pas avec les dernières technologies informatiques. Cela est le résultat d'une contribution mutuelle de milliers de participants à travers l'Internet. Les outils Open-source qui ont été choisis pour satisfaire les différents besoins de la plate-forme sont :

- Système d'exploitation : Linux Mandrake
- Librairie graphique : MESA 3D
- Architecture repartie basée sur CORBA : OmniORB.
- Géométrie, graphique : Open CASCADE, VTK
- Langages de commandes : Python
- Format de sauvegarde : HDF
- Interface Homme-Machine : Qt
- Wrapper C++/Python : Swig
- Visualisation des résultats : swig
- Générateur de documentation : doxygen.

Le choix de toutes ces bibliothèques a été fait après une étude rigoureuse, tout en tenant compte de leur degré de généralité
Figure 1-1). Il a été choisi d'utiliser intensivement les outils déjà existants en Open-source, afin de n'avoir à créer que les outils très spécifiques aux exigences de la plate-forme.

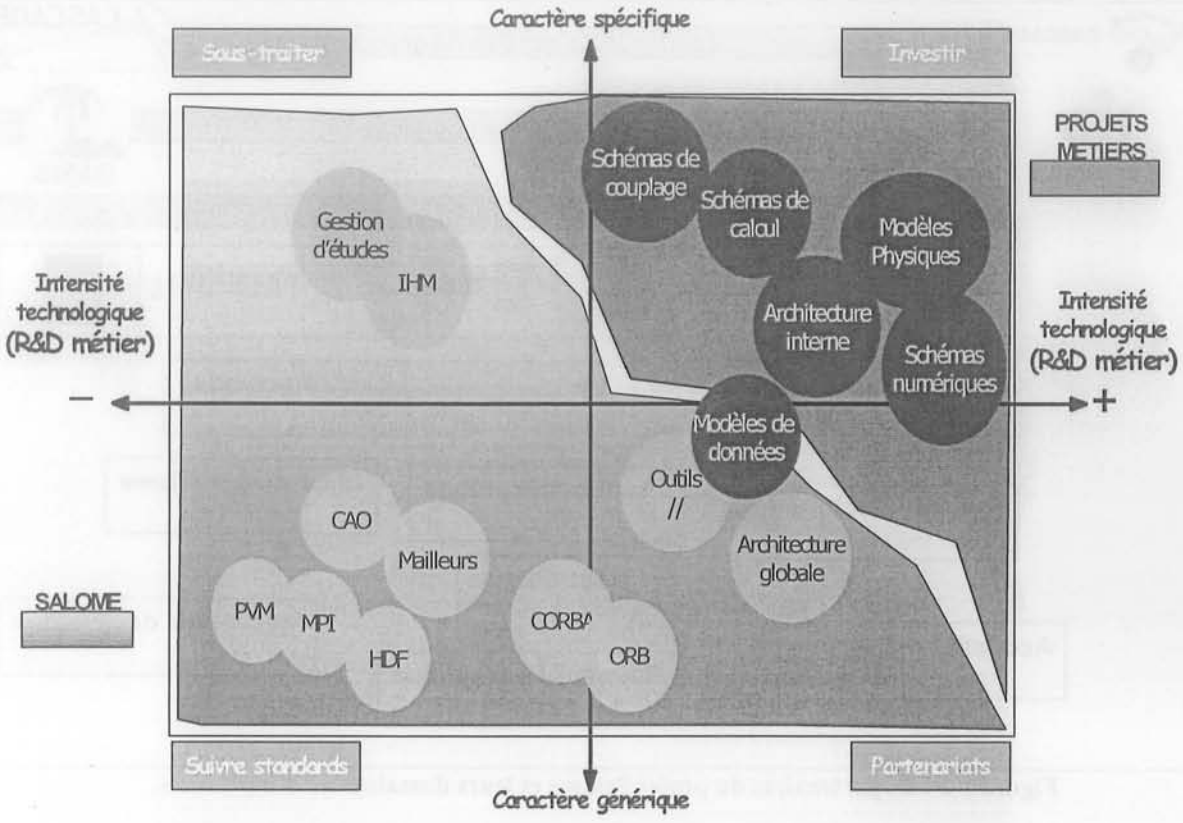


Figure 1-1 : la typographie des composants utilisés dans le projet Salome.

1.3.4 Les partenaires du projet Salome

Le projet a été démarré à l'initiative de Matra Datavision. Il rassemble plusieurs partenaires dont on peut citer :
MATRA DATAVISION, LIP6, CEA, EDF-R&D, AEROSPATIALE MATRA CCR, PRINCIPIA Services, CEDRAT, INPG/LEG, BUREAU VERITAS.

L'idée a été de réaliser trois groupes de savoir-faire apporté par les partenaires:

- Des développeurs informatiques qui apportent des solutions innovantes sur ***l'architecture logicielle*** de la plate-forme et sur les composants.
- Des concepteurs et des utilisateurs de logiciels de simulation, apportant une expérience dans la modélisation et la ***simulation numérique***.
- Des acteurs majeurs de plusieurs domaines industrielles (nucléaire, hydromécanique, génie électrique, aérospatial, etc.). Ces acteurs apportent ***l'expérience de gestion de grands projets***, pour valider des cas de calcul réels.

La Figure 1-2 présente les principaux partenaires et leur savoir-faire apporté.

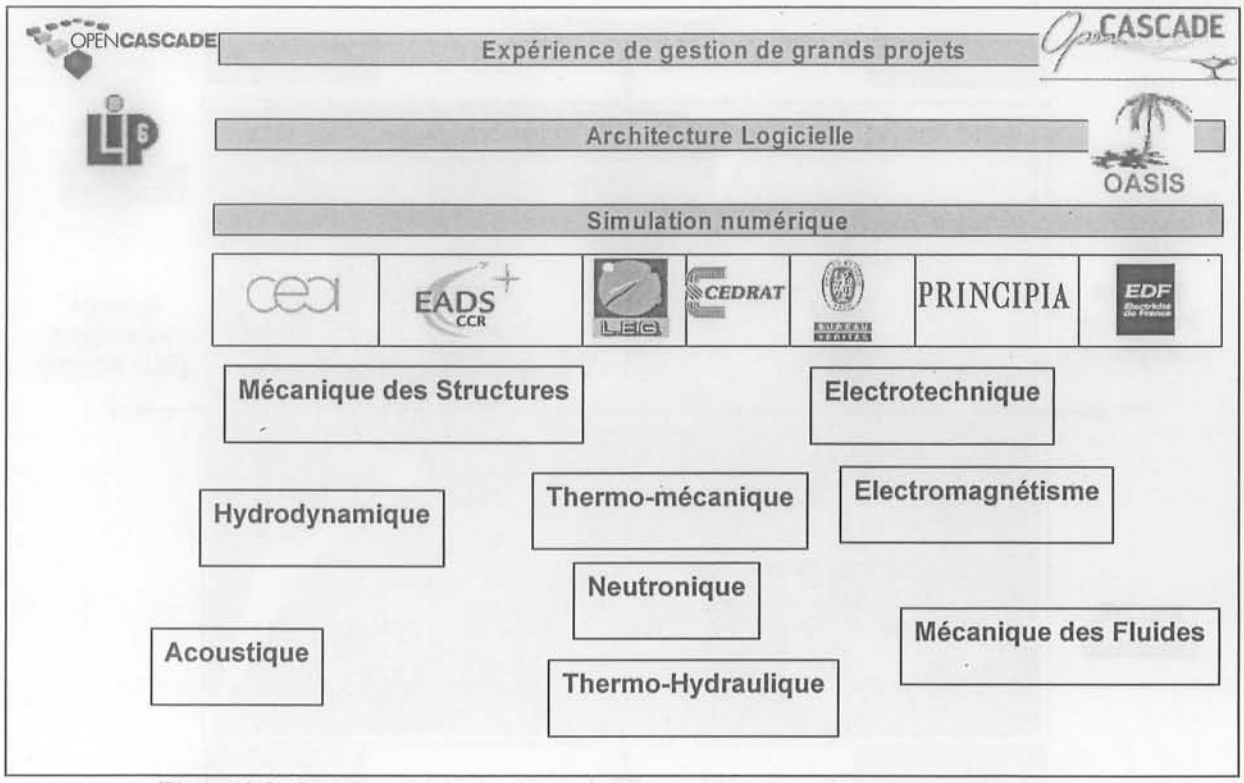


Figure 1-2 : les partenaires du projet Salome et leurs domaines de compétences.

1.3.5 La gestion du projet

La gestion du projet est réalisée d'après les règles de la recherche appliquée dans l'industrie, avec une planification claire, un management de projet. Cette coordination a nourrit en permanence la cohérence du projet SALOME, les objectifs à réaliser et donc les missions, les objectifs techniques à atteindre.

Par ailleurs, la présentation de notre travail de recherche continuera dans le chapitre suivant, chapitre 2, avec un cas d'utilisation, un exemple, qui constitue notre point de départ, nous permettant d'intégrer conceptuellement notre mission dans le cadre de l'architecture globale de la plate-forme SALOME.

La figure ci-dessous, Figure 1-3, présente le cycle de vie de la plate-forme, tel qu'il a été prévu par ses architectes et le calendrier à respecter, présentées dans le cadre de la réunion de travail de mars 2001 par un de notre partenaires, le CEA.

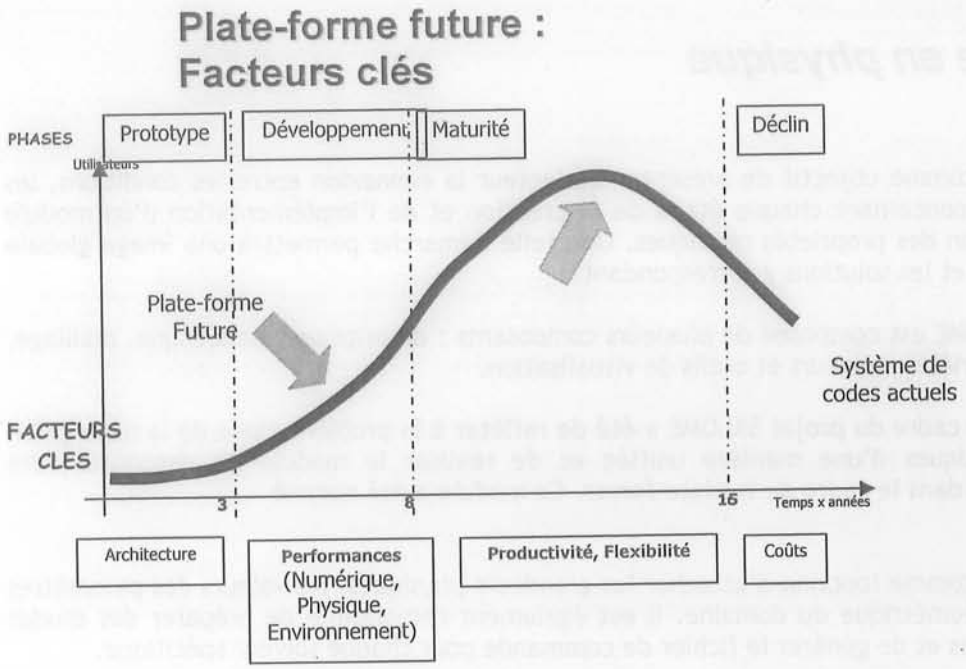


Figure 1-3 : évolution prévue de la plate forme Salome.

1.4 La mise en physique

Ce sous-chapitre a comme objectif de présenter au lecteur la connexion entre les conditions, les besoins, les limites concernant chaque étape de la création et de l'implémentation d'un module dédié à la description des propriétés physiques. Une telle démarche permettra une image globale de la problématique et les solutions y correspondant.

La plate-forme SALOME est composée de plusieurs composants : description géométrique, maillage, description des propriétés, solveurs et outils de visualisation.

Notre travail dans le cadre du projet SALOME a été de refléter à la problématique de la description des propriétés physiques d'une manière unifiée et de réaliser le module de description des propriétés physiques dans le cadre de la plate-forme. Ce module a été nommé « DATA ».

Le module DATA a comme fonction d'attacher les grandeurs physiques, les valeurs des paramètres à une description géométrique du domaine. Il est également responsable de préparer des études paramétriques simples et de générer le fichier de commande pour chaque solveur spécifique.

1.4.1 La mise en données des problèmes physiques pour être résolus en utilisant des solveurs éléments finis

Généralement, les lois de la physique appliquées sur un domaine permettent de décrire le comportement des grandeurs physiques, grâce aux équations aux dérivées partielles. La méthode des éléments finis [Touzot] est l'une des méthodes les plus utilisées pour résoudre ces équations numériquement.

Une telle analyse repose sur la description géométrique du système, mais aussi sur l'ensemble des propriétés physiques qui présentent un intérêt pour la description du domaine.

Réaliser la mise en données complémentaire après la description géométrique sera un deuxième pas nécessaire avant la résolution numérique.

Cette mise en données concerne les propriétés physiques du problème : grandeurs physiques, les lois et les théorèmes valables pour le domaine, propriétés de matériau, conditions limites, etc [Mocanu].

1.4.2 La réalisation du module DATA

Notre présentation a une structure symétrique : elle commence par un exemple de ce que le module DATA doit fournir et elle se termine par ce que le module DATA que nous avons réalisé fournit aujourd'hui.

Cette structure coïncide avec les étapes que nous avons parcourues pour réaliser le module DATA :

Le chapitre 2 présente un cas d'utilisation qui a précédé la conception du module DATA. Le cas d'utilisation a permis de mettre en évidence les besoins du module, lors de la mise en données d'un problème multi-physique.

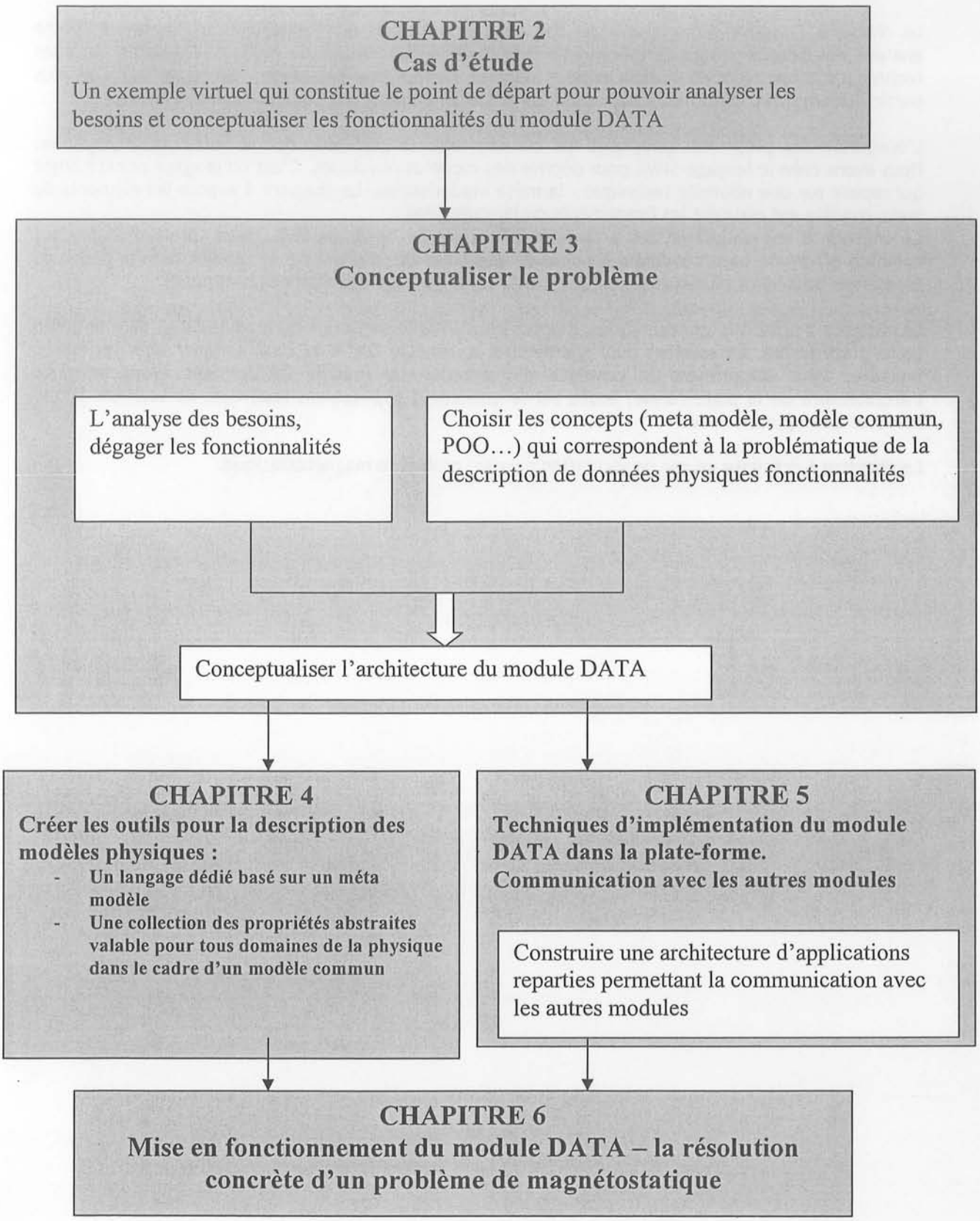
Le chapitre 3 repose sur une analyse des besoins examinés sur l'exemple du chapitre 2. Cette analyse des besoins pilote la spécification du modèle conceptuel du module. Il expose alors les techniques qui peuvent être utilisées pour mener à bien le développement du module DATA, et plus particulièrement la méta modélisation qui sera finalement retenue comme ligne directrice.

L'ensemble des propriétés physiques qui caractérisent le problème définit son modèle physique. Nous avons crée le langage SPML pour décrire des modèles physiques. C'est un langage orienté objet qui repose sur une nouvelle technique - la méta modélisation. Le chapitre 4 expose les éléments du meta-modèle qui assurent les fondements du langage SPML.
La création d'une collection des propriétés physiques de base, en SPML, sous forme d'un modèle commun offre une base commune à plusieurs domaines de la physique et permet dans le cadre de problèmes possédant plusieurs physiques couplées, le partage des données communes.

Le chapitre 5 présente les techniques utilisées dans l'implémentation du module DATA dans le cadre de la plate-forme, notamment pour permettre au module DATA de communiquer avec les autres modules. Pour comprendre le contexte d'interaction du module DATA, nous avons présenté l'architecture de la plate-forme, basée sur le concept d'applications reparties, et aussi les autres modules de la plate-forme.

Le chapitre 6 présente un cas de validation pour un problème magnétostatique.

Structure de la présentation notre travail de recherche



2 Un scénario d'utilisation

A l'origine du projet SALOME se trouve un ensemble de partenaires industriels confrontés aux calculs numériques par éléments finis. Les partenaires ont exprimé le besoin d'une plate-forme d'analyse commune et ouverte dédiée à la résolution des problèmes multi-physiques pouvant utiliser différents solveurs éléments finis.

Concepteurs ou utilisateurs de ces solveurs éléments finis, les partenaires du projet SALOME ont défini une exigence commune - la création d'un environnement qui permette :

- la description de la géométrie pour les dispositifs étudiés.
- la description des données physiques des problèmes.
- la résolution des problèmes multi physiques en utilisant différents solveurs éléments finis.
- l'exploitation et la visualisation des résultats.

Ce travail de thèse a contribué à la conception d'un des modules de la plate-forme (module DATA) dédié à la description et la gestion des données de la physique.
Les spécifications de la plate-forme, et en particulier du module DATA, ont débuté par la proposition de quelques problèmes représentatifs que la plate-forme et le module DATA devaient être capable de résoudre.

Nous exposons ici un des problèmes qui a constitué un scénario possible d'utilisation de la plate-forme. Il s'agit d'un cas d'analyse du phénomène de chauffage par induction pour réaliser un traitement thermique. Nous avons mis l'accent sur les fonctions à réaliser par le module DATA. Ce cas d'utilisation va nous permettre ultérieurement de conduire une analyse des besoins du module DATA et de réaliser une première maquette du module.

2.1 Définition d'un cas d'utilisation représentatif

On désire concevoir une pièce de transmission d'un ensemble mécanique. Cette pièce a pour fonction de transmettre des efforts et permettre le mouvement relatif des ensembles qu'elle joint. Elle doit répondre à la fois à des contraintes d'élasticité pour une solidité optimale et à des contraintes de rigidité locales pour une durée de vie suffisante.

La conception mécanique de la pièce a été réalisée par un bureau d'étude spécialisé. Nous sommes à présent en charge de l'optimisation du processus de traitement thermique à appliquer à la pièce brute pour lui conférer les propriétés mécaniques requises. Le traitement thermique doit être réalisé essentiellement en surface afin d'augmenter la résistance en surface sans altérer l'élasticité de l'ensemble.
Les propriétés à obtenir et leurs localisations sont fournies par le bureau d'étude mécanique spécialisé.

En ce qui concerne le traitement thermique de la surface, il existe de nombreux procédés pour le réaliser. Parmi ceux-ci, le chauffage par induction est bien adapté, car il permet d'obtenir des hautes valeurs de la densité de puissance de surface au niveau de l'objet chauffé (p_s>1000kW/m^2). La densité de puissance peut être ainsi 1000 fois supérieure que celle obtenue par radiation dans un four à 1000°C.

Cela fait de la méthode de chauffage par induction une des meilleures méthodes pour réaliser le traitement de durcissement thermique. D'ailleurs, par induction électromagnétique, le chauffage n'est produit que dans le matériau traité. Par conséquence l'inertie thermique est réduite, ce qui assure un niveau minime de pertes énergétiques. Pour ces raisons nous avons choisi le chauffage par induction, dans le but de réaliser le traitement thermique de la pièce mécanique.

Le chauffage par induction est le résultat du courant produit par induction électromagnétique dans un matériau conducteur situé dans un champ magnétique, variable dans le temps.
D'une manière pratique, le dispositif de chauffage comporte une bobine alimentée en courant alternatif : $i(t)=I_{max}\cdot\cos(\omega t)$. La bobine compose l'inducteur et la pièce à traiter, l'induit. Suite à l'effet pelliculaire, la densité du courant induit diminue lorsque l'on pénètre dans le matériau conducteur. Cette variation est dépendante de la fréquence et des propriétés magnéto-électrique du matériau : la conductivité électrique et la perméabilité magnétique. La Figure 2-1, présente le dispositif de chauffage par induction qui a été proposé.

Pour optimiser le processus de chauffage par induction, il est nécessaire de simuler la distribution de température dans la pièce étudiée.

Nous allons utiliser la plate-forme SALOME, pour étudier l'ensemble du processus de chauffage par induction d'une pièce massive en acier. On connaît la géométrie ; on connaît le matériau de la pièce qui est caractérisé par sa conductivité électrique, par sa perméabilité magnétique, par sa conductivité thermique. On souhaite déterminer via la densité de la puissance due aux courants induits dans la profondeur de la pièce, les valeurs de la température dans celle-ci.

Ce type de problème nécessite deux analyses couplées réalisées sur le même dispositif. On réalise une première simulation magnéto-dynamique pour déterminer les sources de chaleur à l'intérieur de la pièce. Puis ensuite on réalise une simulation de diffusion thermique avec les sources de chaleur précédemment évaluées. Eventuellement le cycle des résolutions peut être répété, car les paramètres électromagnétiques varient avec la température [FEL02]. Classiquement ces deux analyses seraient réalisées distinctement grâce à deux outils de simulation spécifiques. L'idée est

ici de pouvoir dérouler l'ensemble de ce processus de simulation de façon naturelle dans un seul environnement de simulation.

Simulation magnétique

Les équations qui définissent le régime magnétodynamique sont issues des équations de Maxwell et sont associés aux équations de composition des matériaux :

$$\begin{aligned} rot\,\vec{H} &= \vec{J} \\ div\,\vec{B} &= 0 \\ rot\,\vec{E} &= -\frac{\partial \vec{B}}{\partial t} \\ \vec{B} &= \mu \cdot \vec{H} \\ \vec{J} &= \sigma \vec{E} \end{aligned} \qquad (2.1)$$

Le champ magnétique est produit par le courant I(t) qui circule dans les inducteurs du système. La variation du champ magnétique génère des courants induits locaux (noté J) qui deviennent sources de chaleur pour la pièce étudiée. La valeur locale de la densité de puissance développée est donnée par :

$$p_J = \frac{1}{\sigma} J^2 \qquad (2.2)$$

Aux équations, il faut adjoindre des conditions aux limites qui imposeront une composante normale nulle de l'induction sur les limites du domaine (supposées suffisamment éloignées du dispositif) :

$$\vec{B} \cdot \vec{n} = 0\big|_{\Gamma} \qquad (2.3)$$

Pour la résolution du système d'équations différentielles précédents, nous utiliserons par exemple le logiciel Flux3D [Cedrat1]. Cette résolution nous fournira la distribution de densité de puissance thermique p_J dissipée dans la pièce. Cette densité de puissance servira en suite pour définir les sources de chaleur dans le cadre de l'analyse thermique.

Simulation thermique

La deuxième partie de l'analyse concerne l'évolution de la température dans l'intérieur de la pièce. Le phénomène à modéliser est la diffusion de la chaleur à l'intérieur de la pièce en fonction du temps connaissant les sources de chaleur. Il s'agit alors de résoudre l'équation classique de diffusion de la chaleur dans une pièce massive en présence des sources internes :

$$\frac{1}{\alpha}\frac{\partial \theta}{\partial t} = div\,grad\theta + \frac{p}{\lambda} \qquad (2.4)$$

Où la puissance thermique p est la puissance electrique obtenue après l'analyse magnétodynamique, la constante λ nommée conductivité thermique. Le paramètre α s'appelle diffusivité thermique, et est défini par :

$\alpha = \lambda / (\rho \cdot C_p)$ ou « ρ » est la densité du matériau et « C_p » sa chaleur spécifique.

L équation (2.4), ensemble avec l'équation de la convection de la chaleur dans l'environnement,

$$\frac{\partial \theta}{\partial n}(\vec{r}_\Sigma, t) = \alpha(\theta\big|_\Sigma - \theta_a) \qquad (2.5)$$

$$\theta = \theta_0\big|_{t=0} \qquad (2.5)$$

ainsi que les conditions initiales, détermine le champ de température dans la pièce. On a choisi pour résoudre les équations thermiques un solveur gratuit: le solveur Rheolef (disponible à http://www-lmc.imag.fr/lmc-edp/Pierre.Saramito/rheolef/.

2.2 Mise en œuvre dans un environnement de simulation unifié

Sur la base des spécifications du problème présenté ci-dessus, nous avons donc imaginé l'utilisation d'un environnement de simulation intégré capable de réaliser de façon naturelle l'ensemble des étapes nécessaires à la résolution de notre problème. Ces étapes sont les suivantes :

1. Chargement du fichier de CAO fourni par le bureau d'études de mécanique, concepteur de la pièce ;
2. Simplification de la géométrie pour obtenir une description cohérente avec les simulations à réaliser ;
3. Définition des propriétés électromagnétiques de la pièce à traiter ;
4. Définition d'un maillage adéquat ;
5. Sélection du solveur électromagnétique et programmation de la résolution ;
6. Résolution du problème électromagnétique et stockage des résultats obtenus ;
7. Définitions complémentaires des propriétés thermiques et chargement des résultats des puissances dissipées précédentes comme source de chaleur ;
8. Sélection du solveur thermique et programmation de la résolution ;
9. Résolution et stockage des résultats de température ;
10. Post-traitement de l'estimation des caractéristiques mécaniques obtenues.

Nous allons à présent analyser les étapes énoncées précédemment, en particulier **la définition des propriétés physiques**, cette responsabilité revenant au module DATA.

2.2.1 Description de la géométrie du problème (étape 1 et 2)

La description géométrique contient les lignes, les surfaces et les volumes. Dans le cas général, la plate-forme doit permettre la création et la modification de la description géométrique. La plate-forme doit avoir la possibilité d'importer des descriptions géométriques déjà existantes, dans les formats utilisés par l'industrie (par exemple STEP, IGES, BREP) [STEP].

La géométrie a été créée par le bureau d'étude. La Figure 2-1 présente la géométrie du dispositif étudié.

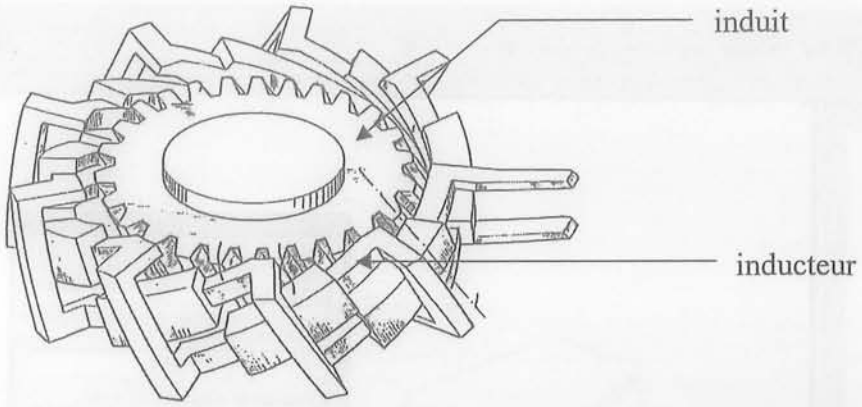


Figure 2-1 Description de la géométrie du dispositif de chauffage par induction

2.2.2 Description des propriétés physiques électromagnétiques (étape 3)

Après la création de la géométrie du système, l'utilisateur doit réaliser la description des propriétés physiques, à l'aide d'une interface homme machine graphique ou textuelle. Pour l'exemple choisi, nous n'avons considéré que l'interface graphique.

Le problème à traiter est multi-physique et nous allons dans un premier temps décrire les données pour le problème magnétodynamique et valider cette résolution avant de passer à la suite.

Typiquement, le problème magnétodynamique est composé des parties suivantes :

- les inducteurs dans lesquels circulent des courants variables en fonction du temps;
- des régions composées des matériaux magnétiques saturables ou non saturables, isolants ou conducteurs;
- des conditions limites spécifiques.
- au moins d'une formulation magnétodynamique, qui exprime la manière de résoudre les équations du système;
- l'air environnant les autres régions;

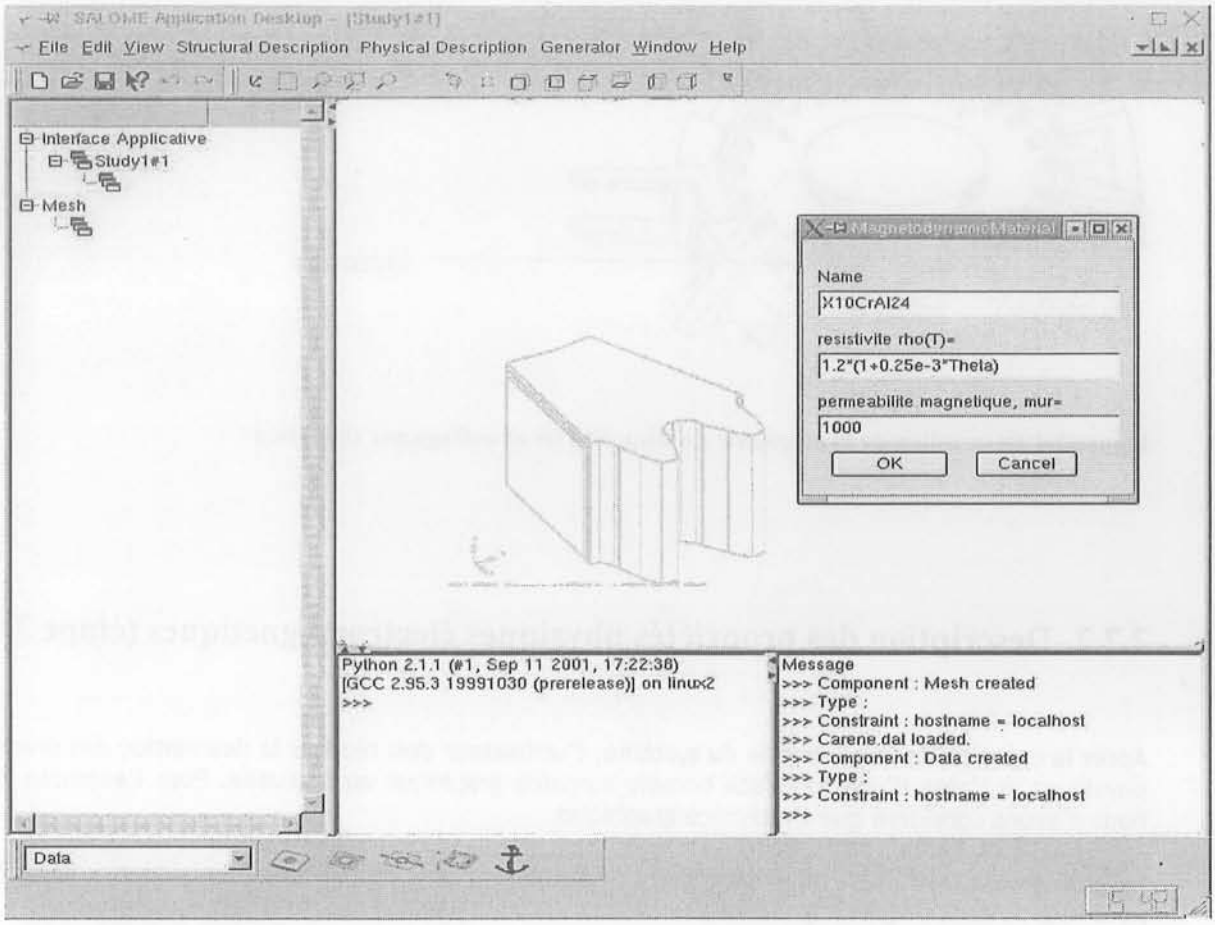


Figure 2-2 – Interface graphique pour la description des propriétés physiques

On peut par exemple s'intéresser à la description du matériau qui a été retenu par le bureau d'étude de mécanique pour le dispositif à modéliser (l'acier X10CrAl24). Le système d'équation (2.1.) montre que les paramètres qui présentent un intérêt pour la formulation magnétodynamique du problème sont :

- la résistivité : $\rho = 1.2(1 + 0.25 \cdot 10^{-3}\theta) \cdot 10^{-6} \quad [\Omega \cdot m]$
- la permittivité relative $\mu_r = 1000$

Notons qu'il faut fixer la température θ pour définir la résistivité.

L'interface graphique doit présenter une fenêtre de dialogue pour définir la fonction de la résistivité et la permittivité relative. Pour permettre une analyse du système en utilisant plusieurs matériaux de même type, cette fenêtre de dialogue doit aussi préciser le nom du matériau.

Pour réaliser ces fonctions nous avons imaginé l'interface graphique présentée dans Figure 2-2. Il est évident que cette fenêtre de dialogue utilisée pour la description du matériau X10CrAl24, est dédiée à des données de type « Magnetic Material » caractérisées par :

- un nom représenté par une chaîne de caractères,
- l'expression d'une fonction représentée par une chaîne de caractères,
- un paramètre réel pour définir la perméabilité magnétique.

Mais cette fenêtre de dialogue n'est que la partie « visible » lors de la création d'un élément de la description physique. Le contenu d'une donnée physique doit être accessible ultérieurement, afin d'être utilisé dans le cadre de la résolution du problème. Pour cette raison, l'existence d'une

structure de données qui peut stoker en mémoire chaque type d'éléments (tel que « MagneticMaterial ») est indispensable pour réutiliser les données spécifiées (telles que la spécification du matériau X10CrAl24).

Similairement, les autres données du problème magnétique seront décrites et chargées en mémoire. Chaque type de donnée de la physique nécessitera sa propre interface et une structure de données spécifique.

Une résolution qui utilise la méthode des éléments finis, impose la subdivision du domaine dans des régions homogènes. Les régions peuvent réunir la description des propriétés physiques : la description des propriétés du matériau qui se trouve à l'intérieur, les conditions limites et aussi la spécification des formulations physiques appliquées sur la région.

Dans le cadre du modèle magnétodynamique, on peut créer la **région** « Acier », ayant comme support géométrique le volume de la pièce étudiée. En plus cette région comporte une référence vers le matériau qui la compose (l'acier X10CrAl24) et la formulation utilisée par le solveur, dans le calcul du champ magnétique et des courants induits (par exemple une formulation magnétodynamique en potentiel vecteur).

Nous allons utiliser le terme de ***modèle physique*** pour définir l'ensemble des structures de données qui sont nécessaires pour assurer le déroulement de la mise en données, pour un type de problème appartenant à un domaine de la physique. On dit qu'un ***modèle physique*** est instancié quand le problème est complètement défini.

2.2.3 Générer le maillage (étape 4)

La pièce à chauffer est séparée en deux volumes en fonction de la profondeur de pénétration du champ magnétique. Ces deux volumes constitueront des supports géométriques pour des domaines ayant a priori des densités de courants relativement différentes, et pour cette raison, les deux volumes seront maillés d'une manière différente en termes de taille des mailles, qualité des éléments, etc. (Figure 2-3). Ce maillage est un maillage dédié à la résolution électromagnétique.

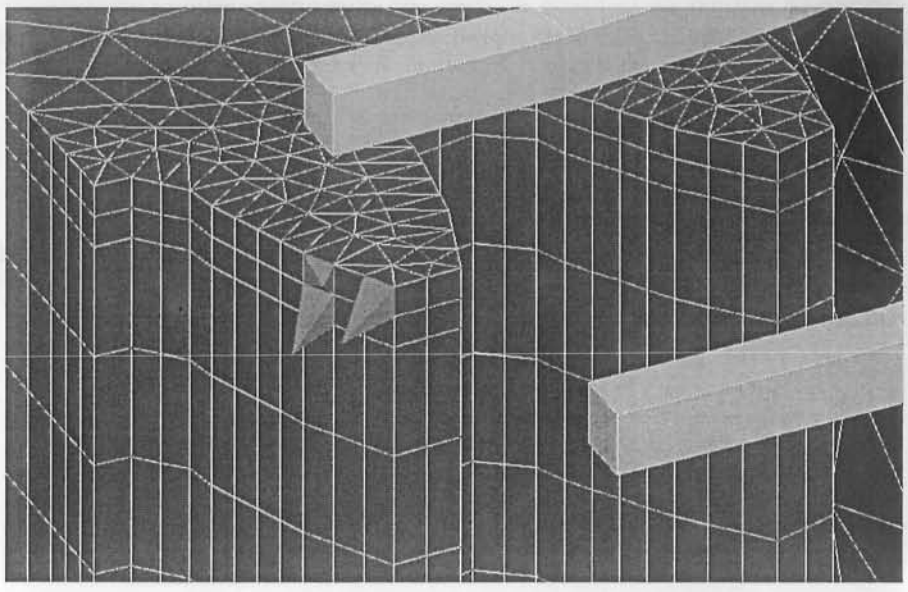


Figure 2-3 – Maillage pour la résolution magnétique

Le modèle physique instancié, ensemble avec la géométrie et le maillage du système, constituent les données d'entrée pour le solveur.

2.2.4 Sélection du solveur magnétique. Lancement de la résolution et persistance des résultats (étape 5 et 6)

La mise en données étant à présent entièrement réalisée, il ne reste plus qu'à sélectionner le solveur Flux pour permettre la résolution. Ensuite le calcul est effectué soit en direct soit en différé en fonction des besoins de l'utilisation.

En fin de résolution, les résultats obtenus sont stockés de manière persistante afin de pouvoir être exploités et utilisés par la suite. Il s'agit entre autres de la valeur des courants et de la densité de puissance dissipée dans la pièce à traiter.

2.2.5 Complément de la description du problème pour traiter le problème thermique (étape 7)

Après la spécification de tous les éléments concernant l'étude magnétodynamique, l'étude thermique du système nécessite l'instanciation d'un modèle thermique, sur la même géométrie, ou sur une géométrie similaire.

Dans ce modèle, conformément aux équations (2.3.-2.5.), l'acier X10CrAl24 est caractérisé par sa conductivité thermique qui a la valeur $\lambda = 14 \quad [W/m{\cdot}^{\circ}C]$.
En conséquence une fenêtre de dialogue semblable à celle présentée dans la Figure 2-4 aidera l'utilisateur à définir le matériau X10CrAl24 du point de vue thermique. Cela permet de compléter la description déjà réalisée et de ne pas avoir à redéfinir l'ensemble du problème pour le cas de la thermique.

D'autre part il faut pouvoir spécifier que les sources de chaleur qui seront utilisées pour la résolution du problème thermique sont les valeurs en chacun des points de la pièce à traiter de la puissance dissipée par le phénomène magnétique. Il faut donc pouvoir indiquer que ces sources sont le résultat obtenu par la simulation précédemment réalisée.

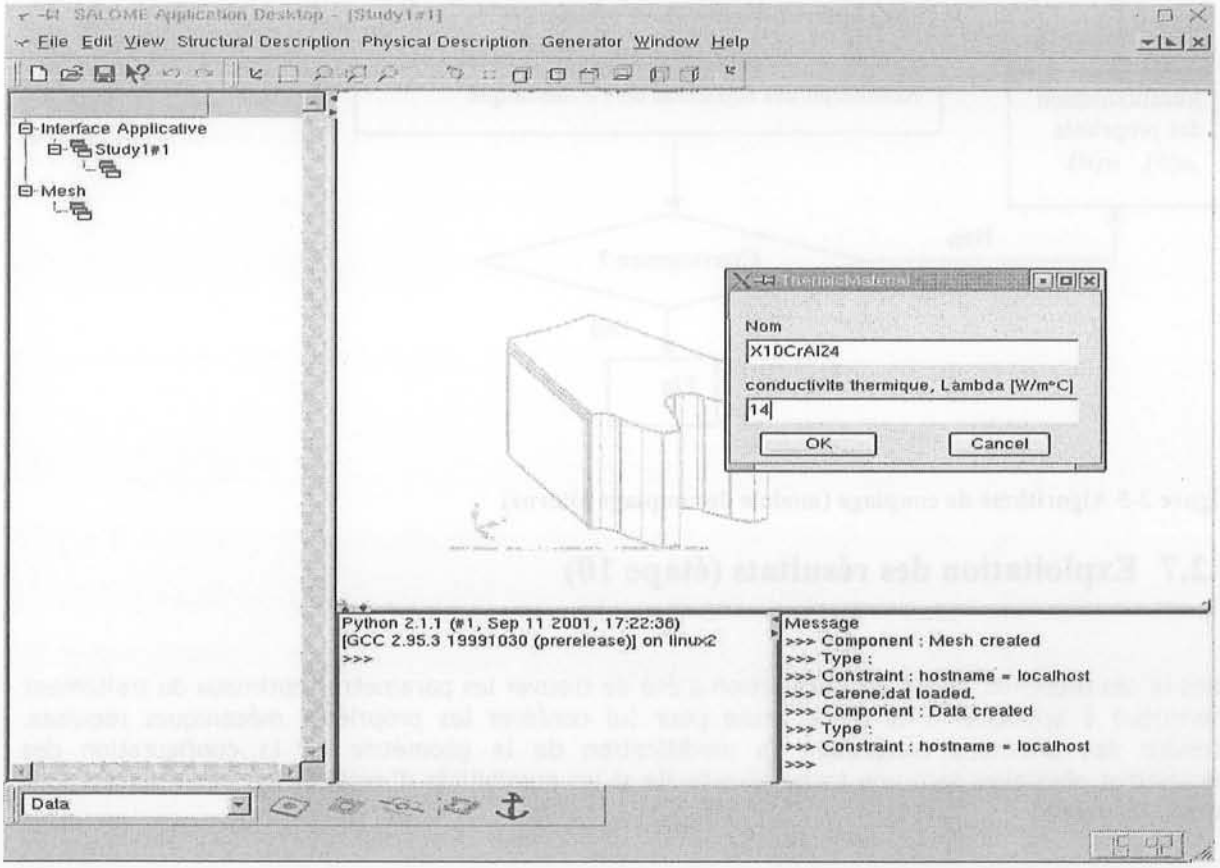


Figure 2-4

2.2.6 Sélection du solveur thermique et programmation de la résolution (étape 8 et 9)

Comme pour le cas de la magnétique, il faut sélectionner le solveur et lancer la résolution ainsi que prévoir le stockage des résultats. Ces résultats seront ceux qui nous intéressent dans ce cas, c'est-à-dire la répartition de la température dans la pièce à traiter.

Pour le cas présenté un couplage alterné [FEL02] permettra l'utilisation des deux solveurs différents. Dans ce cas, les équations aux dérivées partielles de chaque problème sont résolues séparément et le couplage est réalisé par le transfert des données d'un problème à l'autre. L'existence d'un module logiciel superviseur sera nécessaire pour assurer le transfert de données entre les solveurs.

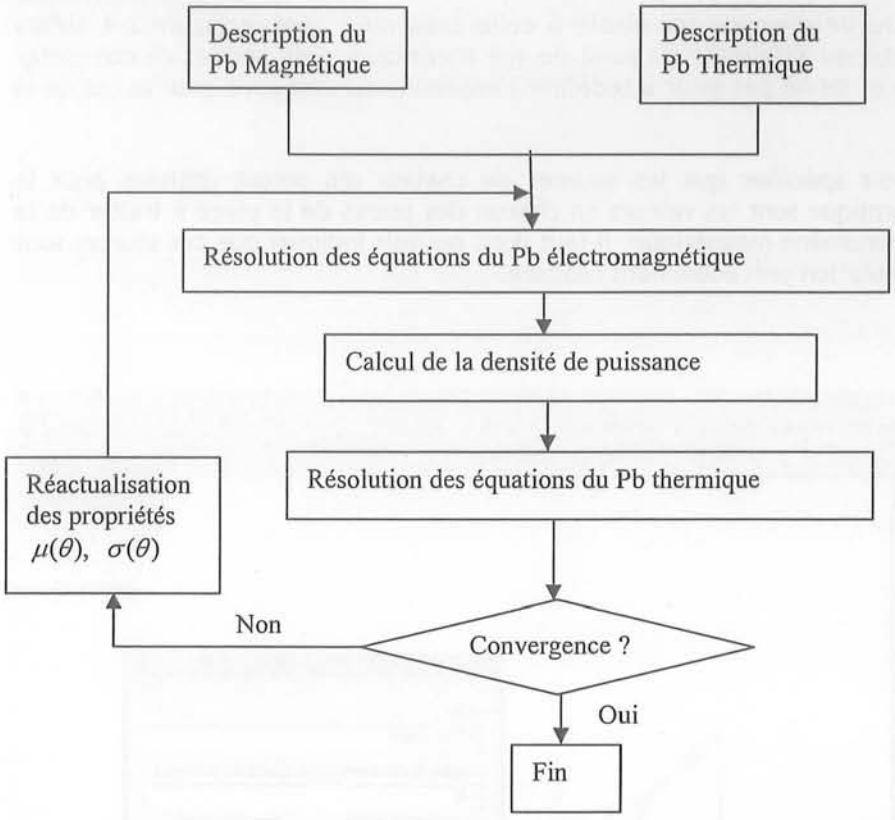


Figure 2-5 Algorithme de couplage (modèle de couplage alterné)

2.2.7 Exploitation des résultats (étape 10)

Dans le cas présenté, le but de l'utilisation a été de trouver les paramètres optimaux du traitement thermique à appliquer à la pièce brute pour lui conférer les propriétés mécaniques requises. Prendre des décisions concernant la modification de la géométrie ou la configuration des paramètres physiques sera une tache plus facile si les possibilités d'exploitation des résultats sont mieux adaptées.

Généralement les fonctionnalités d'exploitation sont offertes par le solveur, mais si le solveur a seulement comme rôle la résolution numérique du problème, la plate-forme, elle, doit offrir un module spécialisé pour l'exploitation des résultats, qui doit répondre aux nécessités suivantes [FluxTutorial]:

- l'accès direct au fichier des résultats généré par le solveur
- la visualisation des informations globales particulières au domaine étudié (par exemple la puissance électrique nécessaire pour réaliser le chauffage)

- la sélection d'un cas de calcul
- la visualisation sur le maillage des résultats disponibles : (par exemple la densité de courant $|\vec{J}|$, le champ de la température θ représentées dans la Figure 2-6.

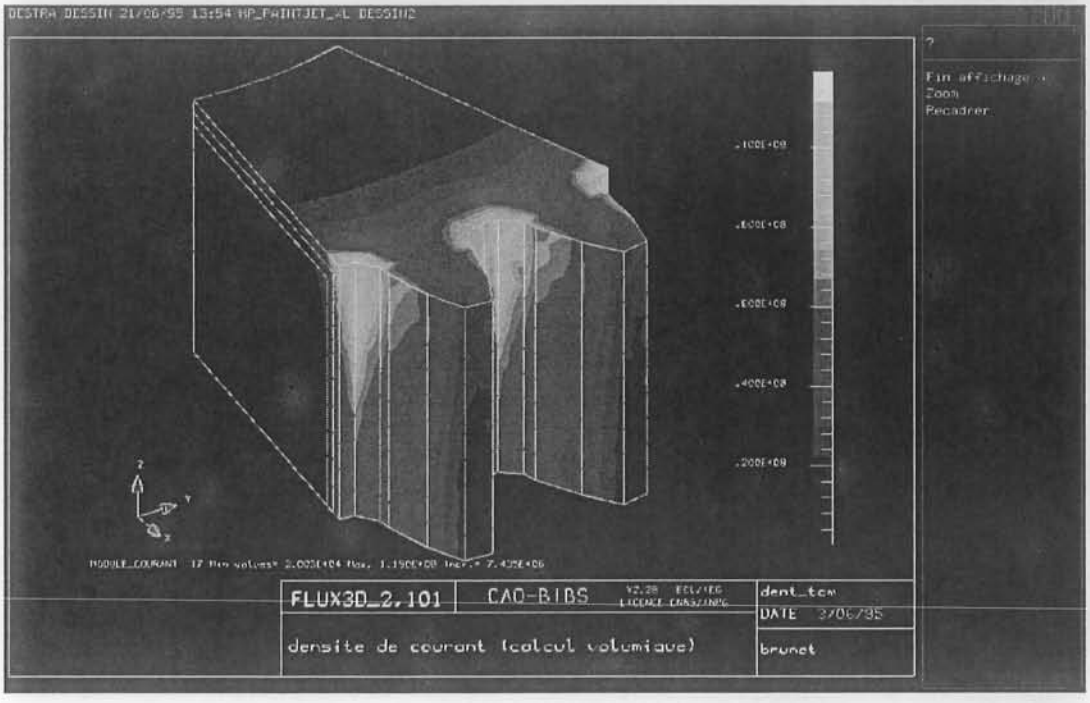


Figure 2-6 - Résultats de simulation thermique

2.3 Conclusion

Nous avons présenté un cas d'utilisation complet qui constituera la base de l'analyse des besoins, développée dans le chapitre suivant. Cette analyse nous aidera à définir les objectifs conceptuels de la réalisation du module DATA.

Ce cas d'utilisation met d'ores et déjà en avant un certain nombre de points intéressants liés à l'utilisation d'un environnement unifié dans le cadre de la multi-physique. Nous avons désormais une vision de ce que nous pouvons attendre du descripteur de propriétés physiques. Nous allons dans le chapitre suivant étudier les aspects théoriques et techniques qui permettent la mise en œuvre de notre projet.

3 Analyse théorique

Le but de ce chapitre est d'analyser, de spécifier et de définir les solutions théoriques et pratiques d'un module de description des propriétés physiques, dans le cadre de la plate-forme d'intégration SALOME.

Le point de départ de la mise en pratique du module de description des propriétés physiques est **l'analyse des besoins d'utilisation** de ce module. Nous distinguons les besoins issus des différents cas d'utilisation et ceux issus de l'interface homme machine et l'interface module DATA avec les autres modules.
Une fois ces besoins définis, nous pouvons déduire **les objectifs à atteindre et les blocs logiques** y correspondant. En conséquence, pendant cette deuxième étape, les objectifs seront structurés sous forme d'**un modèle conceptuel**. La structure de ce modèle représente la colonne vertébrale du module logiciel à implanter.

La troisième étape concerne **les concepts techniques** employés dans la phase mise en oeuvre du module DATA. Cette analyse technique nous permet de comparer les différentes solutions dont nous disposons et de trouver la plus adaptée à nos besoins (une architecture basée sur la méta modélisation).

L'architecture générique du système choisie après cette analyse technique va faire l'objet d'une étude séparée. En ce sens, on analysera l'état actuel de l'application de la solution générique (le méta modèle) dans le génie logiciel.
Finalement, on présente l'adaptation du méta modèle pour le cas spécifique de la description des données physiques.

3.1 *L'analyse*

3.1.1 L'analyse des besoins

L'analyse des besoins a pour objectif de définir la place du module DATA et les fonctionnalités que le module doit offrir. Cette analyse a fait l'objet de nombreuses discussions avec des utilisateurs potentiels du module, notamment des entreprises utilisatrices des solveurs éléments finis, mais aussi des entreprises productrices de logiciels de modélisation.

Nous pouvons séparer l'analyse des besoins en deux parties : une modélisation des cas d'utilisation et la description du dialogue [Sommerville], [JAC].

- La modélisation des cas d'utilisation emploie des acteurs, pour représenter les rôles qu'un utilisateur peut jouer, et des cas d'utilisation pour représenter ce que l'utilisateur doit être capable de faire avec le système.
- La description du dialogue offre une image d'ensemble des interfaces du module DATA avec l'utilisateur et avec le reste de la plate-forme.

a). Les cas d'utilisation

Concernant les acteurs qui interagissent avec le système, on peut en distinguer deux types :

- un utilisateur final qui utilise effectivement le module DATA et dont l'objectif est de réaliser des calculs ;
- un utilisateur intégrateur qui prépare la plate-forme afin que celle-ci soit prête à utiliser par l'utilisateur final.

Le scénario présenté dans le chapitre précédent constitue une première séquence potentielle de cas d'utilisation. Nous pouvons représenter les deux acteurs du scénario sous forme simplifiée à l'aide d'un diagramme des cas d'utilisation d'UML [Ketani], [Muller].

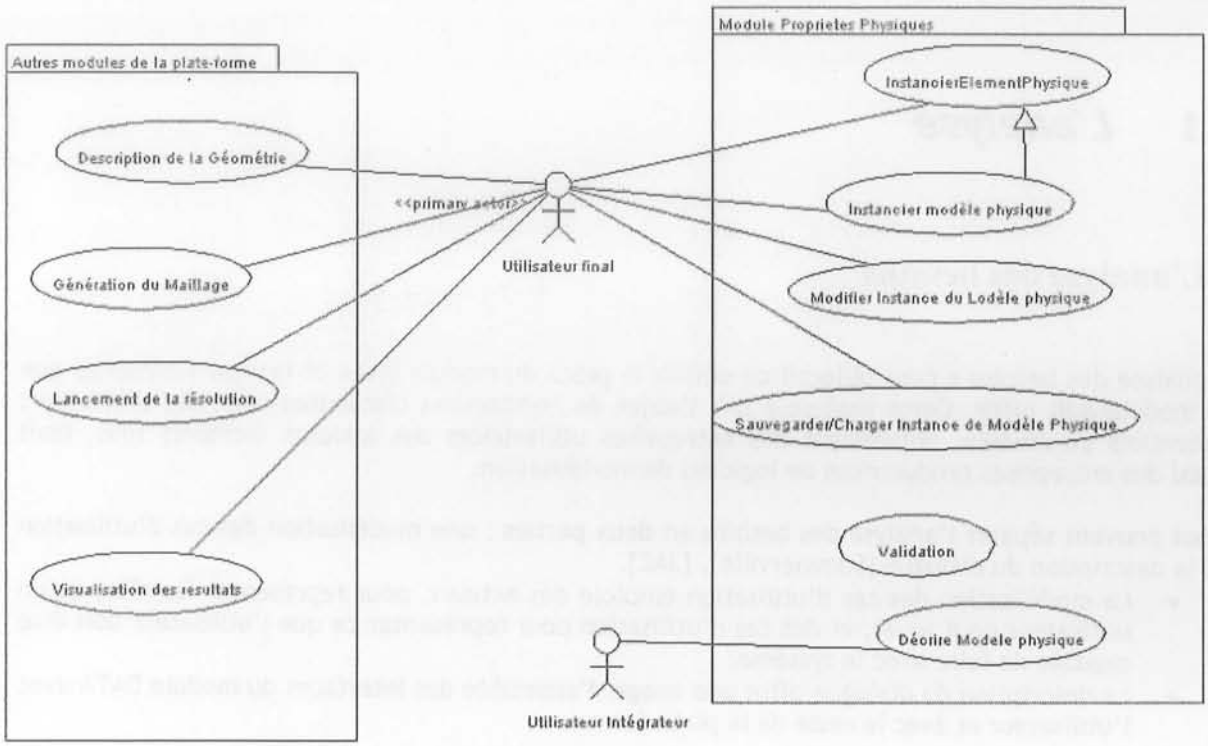


Figure 3-1: Cas d'utilisation de la plate-forme Salome

On peut décrire les cas d'utilisation représentés par le scénario comme suit :

- *Instancier une entité du modèle physique* par *l'utilisateur final*. En dialoguant avec le module, l'utilisateur doit apporter un nouvel élément à la description physique du problème. Par exemple, dans le scénario du chapitre précédant, l'utilisateur final a du créer différents types d'éléments physiques : des matériaux, des sources, des régions volumiques.
- *Instancier le modèle physique* par *l'utilisateur final* qui veut réaliser une description complète du problème, afin d'être envoyée vers le solveur. L'utilisateur doit créer toutes les entités physiques nécessaires pour la description du problème.
- *Modifier un élément du modèle physique* par l'*utilisateur final* ;
- *Sauvegarder ou charger une instance du modèle physique* par l'*utilisateur final* ;
- Décrire un *modèle physique* par *l'utilisateur intégrateur* est une étape qui conditionne l'utilisation finale. Cela parce que l'instanciation d'un modèle physique nécessite l'existence du modèle lui même. Dans le scénario du chapitre précédant, pour réaliser une analyse magnétodynamique / thermique, le module DATA a eu besoin d'une description d'un modèle physique spécifique approprié.

L'utilisateur final exige aussi d'autres fonctionnalités pour réaliser l'analyse d'un problème physique : description géométrique, génération du maillage, lancement de la résolution et la visualisation des résultats. Toutes ces fonctionnalités font l'objet d'études pour les autres modules de la plate-forme SALOME.

b).Les interfaces du module DATA

Pour les cas d'utilisation destinés à l'utilisateur final, le dialogue entre celui-ci et le module DATA passera par une interface homme machine (IHM). L'IHM peut assurer, entre l'utilisateur et le système, un dialogue textuel ou une interaction graphique, plus facile à utiliser [Dix]. Selon le cas d'utilisation, cette interface peut être plus ou moins complexe.

Instancier une entité du modèle physique à l'aide d'une interface graphique (Graphic User Interface) nécessite une interface à chaque entité. Le scénario de chapitre précédant présenté l'exemple de GUI pour instancier un matériau magnétodynamique / thermique (Figure 2-2).

Mais il n'y a pas que les interfaces avec l'utilisateur à analyser. Le module DATA doit communiquer aussi avec les autres modules de la plate-forme. Ainsi par exemple, le modèle physique instancié a comme support des entités géométriques, il a donc besoin de dialoguer avec le module Géométrie. De l'autre coté, les instances des modèles physiques doivent être accessibles par le solveur. On verra dans le chapitre 5 la solution choisie pour réaliser cet interfaçage entre les différents modules de la plate-forme.

3.1.2 L'analyse conceptuelle

Compte tenu de l'analyse de besoins faite ci-dessus, on peut dégager les objectifs suivants :

- on a besoin d'un modèle de données qui structure les propriétés physiques du problème ;
- il est nécessaire que ce modèle de données communique avec une base de données pour réaliser les opérations de sauvegarde et de restauration ;
- l'instanciation du module doit être accessible à l'aide d'une interface graphique ;
- l'interface graphique doit être adaptée à chaque physique ou chaque solveur. A priori cette dernière constatation semble laisser à la charge de l'utilisateur intégrateur le soin de réaliser cette interface graphique. Nous tenterons d'apporter tout de même des réponses génériques à ce besoin et c'est pourquoi, nous introduisons un moteur d'interface avec l'utilisateur
- les données physiques sont utilisées et utilisent d'autres modules. Il est donc nécessaires de disposer d'une interface de programmation pour le module DATA.

Ces objectifs nous conduisent à un modèle conceptuel dont la structure des composants a été représentée dans la Figure 3-2.

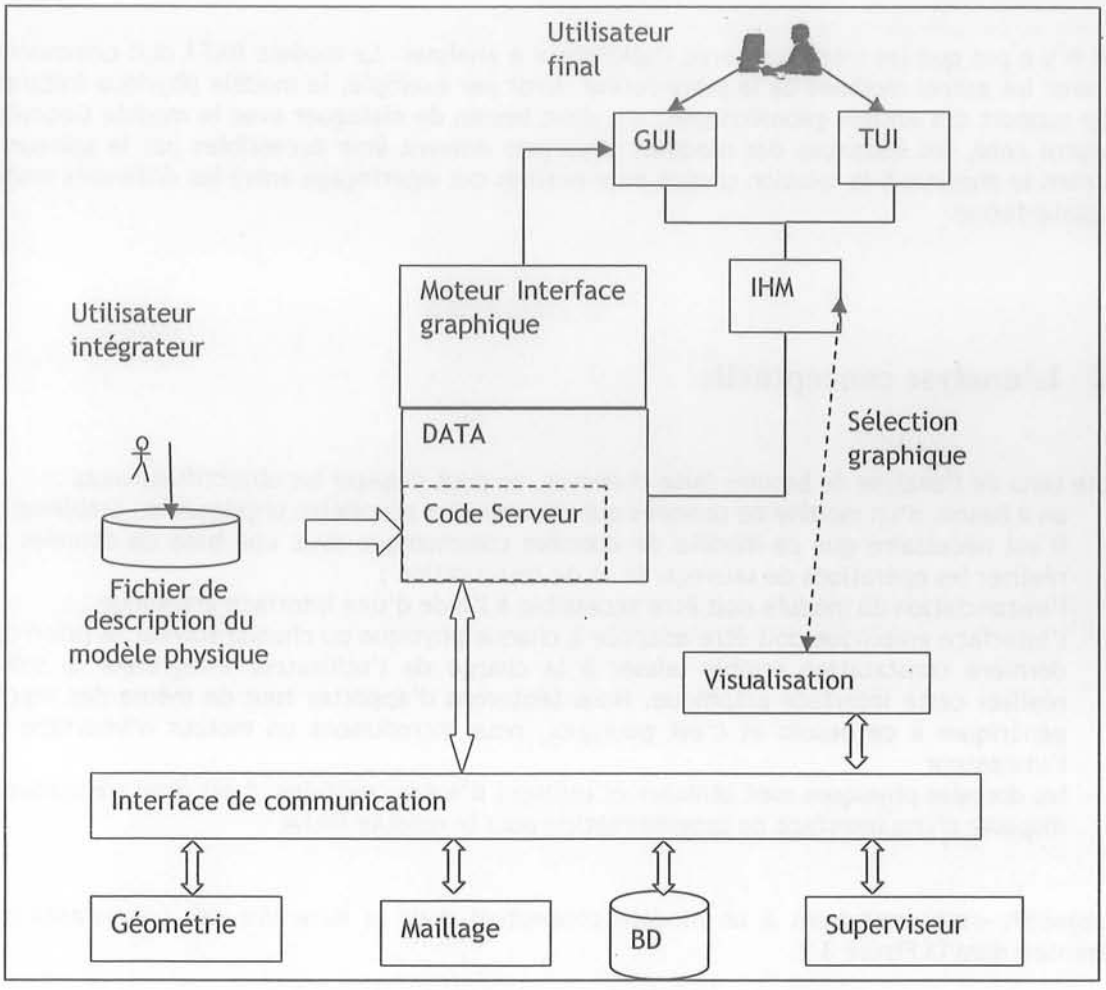


Figure 3-2: Modèle conceptuel du module DATA

La figure envisage la structure du module qui réalisera la description des propriétés physiques. Le rôle central est joué par un bloc qui implémente un modèle de données physiques. Ce bloc a des liens avec les interfaces utilisateur (IHM), avec une base de données de la description physique et avec les autres modules de la plate-forme.

3.2 *Le contexte de la programmation orientée objet*

La mise en donnée d'un problème de modélisation peut profiter des récentes évolutions de la technique informatique. Pour un solveur donné, la programmation orientée objet (POO) permet d'organiser les concepts et de définir leur comportement.

Ainsi, avant d'exposer les éventuelles approches théoriques ou pratiques, il nous a semblé souhaitable de donner quelques éléments de base sur les techniques de Programmation Orientée Objet, qui sont présentés dans la plupart des développements informatiques actuels. Le lecteur familier avec ces concepts pourra bien évidemment passer ce paragraphe. Pour les autres, nous allons essayer de donner, en quelques mots, ce que signifie et apporte cette technique de programmation.

3.2.1 Les clases et les objets

Les classes et les objets représentent les éléments de base de la programmation orientée objet [OMT].

Les classes jouent le rôle de modèle, d'archétype définissant des propriétés et des comportements. Les représentants d'une classe ont la structure et le comportement de leur classe. Conformément aux définitions de la programmation orientée objet, un représentant d'une classe est appelé un objet de cette classe. La structure et le comportement d'un objet sont donnés respectivement par les définitions des attributs et des méthodes de la classe.

On peut considérer que dans l'exemple magnétodynamique présenté dans le chapitre 2, l'utilisateur final a, à sa disposition, la classe `MagneticMaterial` et il a l'intention de créer une instance du celle-ci, nommée `X10CrAl24`, avec deux paramètres (représentés par les attributs de la classe), $\rho = 1.2(1 + 0.25 \cdot 10^{-3}\theta) \cdot 10^{-6}$ $[\Omega \cdot m]$, et la permittivité relative $\mu_r = 1000$ (Figure 3-3 et la Figure 2-2)

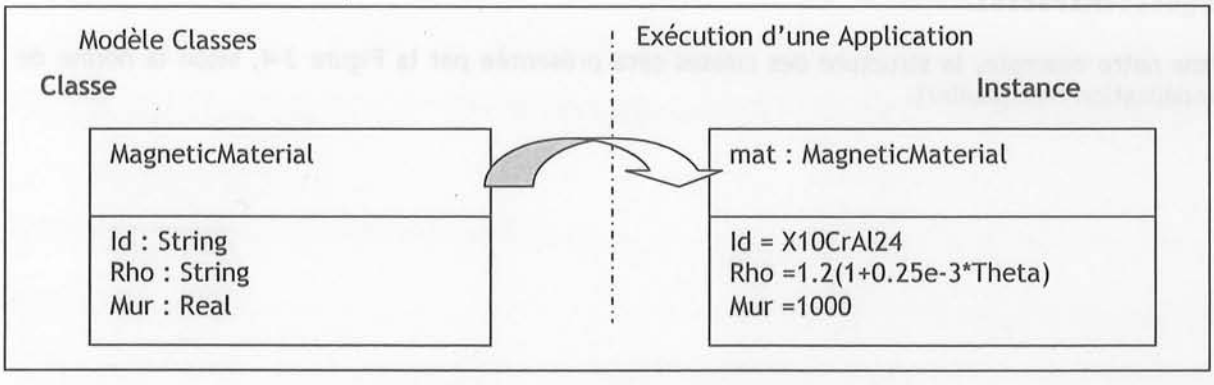


Figure 3-3 : exemple de classe et instance.

L'approche par classe est destinée à faciliter le travail du concepteur de logiciel, dont l'objectif est de créer un modèle informatique virtuel d'un objet réel. En ce sens, la démarche associée à la programmation orientée objet est relativement naturelle.

a). Créer le modèle de classes (abstraction)

Rendre possible la simulation numérique du dispositif par l'utilisateur final est, dans le contexte de la programmation orientée objet, conditionnée par l'existence d'un ensemble de classes. Cet ensemble de classes s'appelle le modèle de données.

La création du modèle de données est un processus d'abstraction [OMT] : partant d'un système réel, il s'agit d'en faire un modèle informatique virtuel à base de classes. Dans le développement d'un nouveau modèle, l'abstraction entraîne nécessairement une simplification : l'intérêt du concepteur ne se porte que sur les comportements utiles à l'application visée.

Dans le cadre de notre modélisation du chapitre précédent, les seules propriétés du matériau à prendre en compte lors de l'étude magnétodynamique sont la résistivité et la perméabilité. A partir de ces besoins, le concepteur a créé la classe `MagneticMaterial`, possédant les attributs correspondants à ces deux propriétés.

Mais la notion de matériau peut aussi intéresser un autre utilisateur, qui lui souhaite conduire une analyse thermique.Dans ce cas, le concepteur intégrateur du solveur devra proposer un modèle de matériau qui possède les propriétés de :

- conductivité thermique ;
- capacité calorifique ;
- densité ;

ce qu'il propose dans une autre classe, la classe `ThermicMaterial`, qui possède les trois attributs correspondants.

b).L'héritage

Bien sûr, dans certains solveurs, il existera des familles de problèmes pour lesquels les propriétés intéressantes des matériaux seront les propriétés magnétiques et d'autres pour lesquels ce seront les propriétés thermiques. Il faudra donc, au sein du modèle de donnée du solveur, disposer des types de matériaux.

Dans cette situation, une nouvelle abstraction est possible. Considérant que les 2 classes représentent des matériaux, quelles ont des propriétés et des comportements communs, il est possible de définir une nouvelle classe `AbstractMaterial`, classe qui sera dite abstraite car aucun représentant de cette classe ne peut réellement exister et qui rassemblera les attributs et les méthodes communes aux 2 classes précédemment définies `MagneticMaterial` et `ThermicMaterial`. En conséquence, les classes `MagneticMaterial` et `ThermicMaterial` ne modélisent plus que leurs comportements spécifiques et héritent d'une description commune par la classe `AbstractMaterial`. On dit aussi que la classe `AbstractMaterial` généralise la classe `MagneticMaterial`.

Dans notre exemple, la structure des classes sera présentée par la Figure 3-4, selon la norme de modélisation UML [Muller].

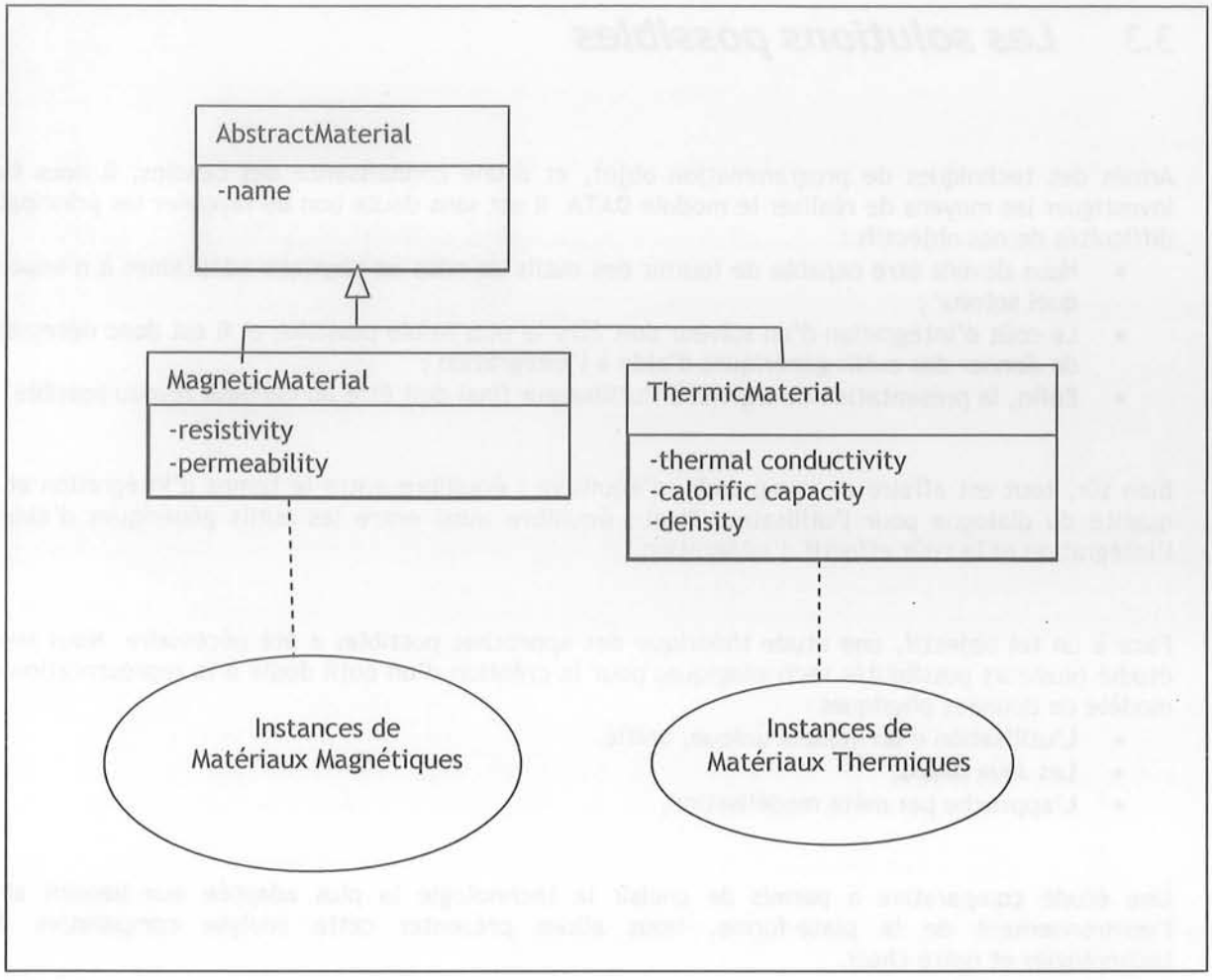


Figure 3-4

Selon l'héritage proposé, les instances de la classe `MagneticMaterial` peuvent être considérées comme des instances de la classe `AbtractMaterial`. La généralisation et l'héritage sont transitifs à un nombre indéfini de niveaux : une instance d'une classe sera simultanément instance de toutes ses clases de base. Les attributs d'une instance sont formés par l'union des attributs de toutes ses clases de base.

L'héritage permet donc de gagner en modularité et réduit considérablement l'effort de programmation.

En terme de démarche associée à l'approche orientée objet, la création d'une nouvelle classe dans un modèle de donnée sera précédée par une analyse des classes existantes, afin de réutiliser le maximum de code existant. A contrario, lors de la création de nouvelles classes dans le cadre d'un projet totalement nouveau, il faudra toujours penser la structure d'héritage en vue de concevoir des classes modulaires et réutilisables.

3.3 *Les solutions possibles*

Armés des techniques de programmation objet, et d'une connaissance des besoins, il nous faut investiguer les moyens de réaliser le module DATA. Il est sans doute bon de rappeler les principales difficultés de nos objectifs :

- Nous devons être capable de fournir des outils de mise en physique adaptables à n'importe quel solveur ;
- Le coût d'intégration d'un solveur doit être le plus faible possible, et il est donc nécessaire de donner des outils génériques d'aide à l'intégration ;
- Enfin, la présentation du logiciel à l'utilisateur final doit être du meilleur niveau possible.

Bien sûr, tout est affaire de compromis, d'équilibre : équilibre entre le temps d'intégration et la qualité du dialogue pour l'utilisateur final ; équilibre aussi entre les outils génériques d'aide à l'intégration et le coût effectif d'intégration.

Face à un tel objectif, une étude théorique des approches possibles a été nécessaire. Nous avons étudié plusieurs possibilités technologiques pour la création d'un outil dédié à la représentation du modèle de données physiques :

- L'utilisation d'un modèle unique, unifié.
- Les Java Beans,
- L'approche par méta modélisation,

Une étude comparative a permis de choisir la technologie la plus adaptée aux besoins et à l'environnement de la plate-forme. Nous allons présenter cette analyse comparative des technologies et notre choix.

3.4 *Le modèle de données unifié*

Face à un problème de cette envergure, une solution envisageable est de proposer un modèle de donnée unique, fédérateur.

Cette solution est simple, au moins en apparence. Tout solveur qui souhaite intégrer la plate-forme Salomé doit se conformer au modèle de donnée du module DATA. Bien sûr, ce modèle de donnée devra être bien pensé, en particulier pour rendre l'intégration la plus rapide possible. Au besoin, si certains concepts manipulés par le solveur en voie d'intégration ne sont pas couverts par le modèle de données fédérateur du module DATA, l'intégrateur pourra étendre ce modèle et éventuellement proposer tout ou partie de cette extension comme évolution du modèle fédérateur.

L'industrie de la CAO, en mal de standard d'échanges de données à proposé une telle approche dans le cadre beaucoup plus vaste de la norme STEP (Standard for The Exchange of Product model data) [STEP]. Il existe même une partie de cette norme qui pourrait répondre partiellement à nos besoins, puisque s'intéressant à la simulation numérique par éléments finis.

Examinons, à la lumière des apports de cette norme quels sont les contraintes et les bénéfices de cette approche.

3.4.1 Principes d'un modèle physique unifié

Un modèle unique utilisé pour la description de données de la physique doit se conformer aux contraintes imposées par les deux disciplines qui collaborent à sa création, le génie physique et le génie logiciel [Ma] :

- la séparation entre données, modèle de données, représentation des données.
- l'utilisation intensive de la POO.
- l'élaboration d'objets métier (propre à chaque physique).
- éviter les multiples définitions des types de données.

En s'appuyant sur les principes qui ont conduit à la norme STEP, notre modèle de données unifié devrait respecter les conditions suivantes :

- Le modèle unifié doit permettre d'isoler les entités appartenant à un certain domaine de la physique. Cette réduction simplifie le travail de l'utilisateur intégrateur qui peut limiter ses efforts au domaine étudié.
- Le modèle unifié doit posséder une ou plusieurs parties communes, que tout solveur doit implanter et qui permet le transfert d'information entre solveurs. Ce modèle commun est aussi un atout pour l'utilisateur intégrateur, à qui il apporte un noyau de classes totalement réutilisables.
- La modification du modèle unifié ne doit pas gêner l'utilisateur intégrateur.
- L'utilisateur intégrateur peut étendre le modèle fédérateur et proposer sa contribution comme extension du modèle unifié à une nouvelle discipline pas encore traitée. Il doit pour cela obtenir l'aval de sa communauté et aussi du reste des utilisateurs, car ses ajouts doivent à la fois être représentatifs des besoins de sa discipline et ne pas entraver le développement des autres disciplines.

3.4.2 Les contraintes d'une approche par modèle de données unifié

Globalement, cette méthode se prête bien à des domaines établis : la modélisation géométrique en est un exemple. Elle a fait l'objet déjà de nombreuses normes par le passé et à ce titre, est le fruit d'une lente maturation. Du coup son adoption par la communauté, condition indispensable de réussite d'une entreprise de normalisation, est relativement rapide.
Le domaine du calcul scientifique est encore éloigné de cette maturité, même si les méthodes éléments finis constituent un socle commun à de nombreuses disciplines.

L'autre difficulté importante réside dans l'architecture du modèle de données unifié. Pour des raisons évidentes d'évolutivité, elle ne doit pas être monolithique, mais constitué de parties, comme la norme STEP l'a proposé, relativement indépendantes les unes des autres. Cet objectif demande encore une fois un recul qu'il est sans doute difficile de trouver actuellement dans la communauté du calcul scientifique.

Enfin, clairement, le travail de l'intégrateur n'est que peu facilité. Il doit construire et maintenir des interfaces entre son modèle de donnée natif et celui qu'il a exprimé dans le modèle de données unifié.

Par contre notre travail, en tant que réalisateur du module DATA, est relativement balisé.

3.5 *Les Java Beans*

A l'opposé de l'approche fédératrice du modèle unifié, les Java Beans sont plutôt des boites à outils de « petits » objets, associables à la demande, pour constituer les objets complexes de la discipline considérée.

L'origine des Java Beans est issue des méthodes de conception des interfaces graphiques utilisateur (GUI). Ce domaine a bénéficié de l'utilisation des librairies de classes dédiées. Par exemple pour développer des interfaces graphiques en C ou C++, QT offre une librairie de classes et un *environnement de développement intégré* (IDE) qui facilite la construction de boites de dialogues.

La technologie Java a pris une voie légèrement différente en proposant les Java Beans. Les composants Java Beans sont des composants développés par des programmeurs avancés, pour être utilisés aussi par des programmeurs relativement novices. Ces derniers utilisent visuellement les composants de type Java Beans, dans *un outil de développement* (OD), voir Figure 3-5 [Bean] [Bonjour].

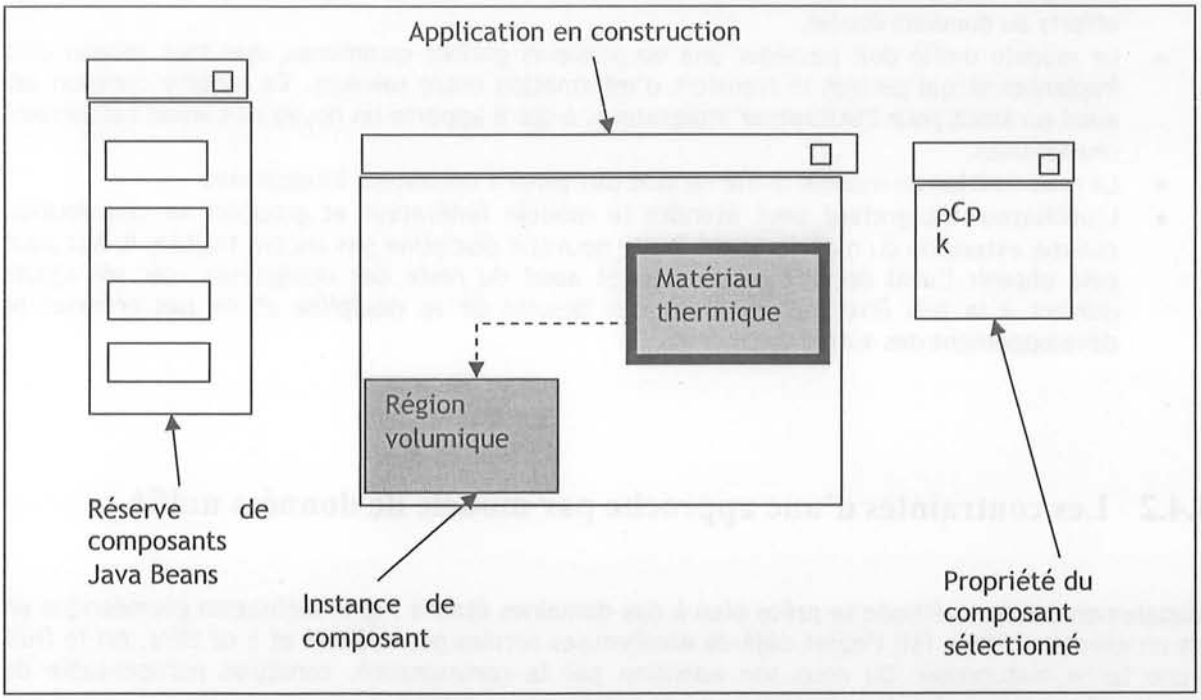


Figure 3-5 : utilisation des Java Beans dans le cadre du module DATA

Implémentée dans notre module DATA cette technologie serait bien pratique, car elle conduirait à une conception graphique des modèles physiques par l'utilisateur intégrateur. Dans la Figure 3-5 nous présentons comment on peut consulter un composant de dialogue pour un Matériaux thermique : nous avons sélectionné graphiquement un objet de type « matériau thermique » pour construire par assemblage le dialogue de saisie d'une région volumique.

3.5.1 La structure du Java Beans

Les Java Beans sont en fait des classes Java ordinaires qui possèdent un constructeur sans argument. La majorité des composants Java Beans ont dans leur composition des fonctions *set()* (mutateurs), pour modifier des attributs, et des fonctions *get()* (assesseurs), pour obtenir la valeur contenue dans leur attributs [Vartiainen]. Les valeurs données par les assesseurs représentent les propriétés du composant Java Beans. La structure du composant Java Beans contient aussi des évènements reconnus et des méthodes de traitement de ces évènements.

3.5.2 Les fonctionnalités du composant Java Beans [Englander],[Bean]

a). Introspection

Toute classe Java peut être interrogée au sujet des ses propriétés et de ses comportements en utilisant les classes du paquetage *java.reflect*. Cette faculté s'appelle introspection et elle est utilisée avec les outils de conception graphique pour permettre l'édition des propriétés d'une manière graphique. En plus, l'introspection peut être utilisée pour déterminer quels types d'événements seront utilisés pour l'interaction du composant Java Beans avec le reste de l'application.

b). Personnalisation

L'aspect et le comportement d'un composant Java Beans peuvent être modifiés en utilisant ses mutateurs. Cette modification est possible en utilisant des éditeurs des propriétés spécialement adaptées pour les composants Java Beans.

c). Manipulation d'événement.

Les composants Java Beans communiquent en envoyant des événements. D'autres composants Java Beans peuvent s'enregistrer en tant qu'écouteur de ces événements. Quand l'événement se produit, les méthodes appropriées des composants Java Beans récepteurs, inscrits pour ce type d'événement, sont appelées.

d). Persistance.

Les composants de Java Beans sont persistants. Ceci signifie que l'état d'un composant peut être sauvegardé et reconstitué. La manière la plus facile de réaliser la persistance est d'implémenter l'interface *java.io.Serializable*. Dans ce mécanisme de sauvegarde le composant est transformé en forme binaire après quoi il peut être stocké sur le disque ou être envoyé au-dessus du réseau et plus tard être reconstitué juste comme il était avant ce processus de sérialisation.

3.5.3 Analyse de la technologie des Java Beans dans le cadre du projet Salome

Si les Java Beans constituent une approche élégante à bien des points de vue, ils possèdent cependant plusieurs inconvénients dans le cadre du projet de module DATA.

C'est tout d'abord une solution propriétaire, même si les Java Beans constituent plus une démarche que de réels outils. C'est, à ce titre, en contradiction avec la volonté de fournir un environnement Open Source.

Par ailleurs, les Java Beans mettent l'accent sur la présentation graphique des composants construits, par contre le modèle de données sous jacent n'est pas exprimé. Le modèle de données n'étant pas complètement maîtrisé, la réutilisabilité des composants construits sous cette forme n'est pas certaine.

Enfin la connexion par envois et capture d'évènements entre composants est relativement déstructurée et peu propice à la lisibilité.

En conclusion, l'approche par construction graphique est séduisante, mais ne peut pas être conçue sans un support solide sur un modèle de données clairement exprimé.

3.6 *La méta modélisation*

La méta modélisation trouve sa place dans le cadre de la POO.

La relation entre un objet et la classe qui représente son modèle, oppose deux entités de nature différente : la classe a une existence statique liée au code qui la définit, dont l'objet est une instance. L'existence de l'objet est située à un niveau « inférieur », celui de l'exécution de l'application.

Il est facile de définir une classe. Mais son évolution temporelle est limitée par la nécessité de changer le code qui implémente la classe : l'utilisateur a l'idée de changer le modèle ; la difficulté est que pour changer le modèle, il doit changer le code qui implémente les classes du modèle (Figure 3-6)

Les objets sont des entités dynamiques qui n'existent que pendant l'exécution de l'application qui les créent. Au gré des idées de l'utilisateur final, les objets peuvent être modifiés à volonté, en attribuant différentes valeurs à ses attributs (par exemple il peut modifier la valeur du paramètre ρ du matériau thermique X10CrAl24), ou en faisant agir les différentes méthodes de classe.

La base de la méta modélisation est de réaliser une abstraction du modèle orienté objet sous la forme d'un autre modèle qui se trouve à un niveau hiérarchique supérieur, de telle sorte que les classes deviennent des objets à ce niveau hiérarchique.

Avec la méta modélisation, le modèle est une instance du méta modèle, et les éléments du modèle sont des éléments du méta modèle, de la même manière que les objets sont générés par les classes.

Le modèle gagne de cette manière la liberté temporelle qu'il n'avait pas dans la conception POO classique. Le modèle peut être modifié dynamiquement, selon les idées d'un utilisateur intégrateur, par modification des attributs des objets du méta modèle (Figure 3-6).

Théoriquement le méta modèle peut être considéré comme une instance d'un méta méta modèle, et ainsi de suite, tout peut avoir « un niveau méta ». En pratique, seulement 4 niveaux s'avèrent nécessaires [Atkinson] [Sellers1].

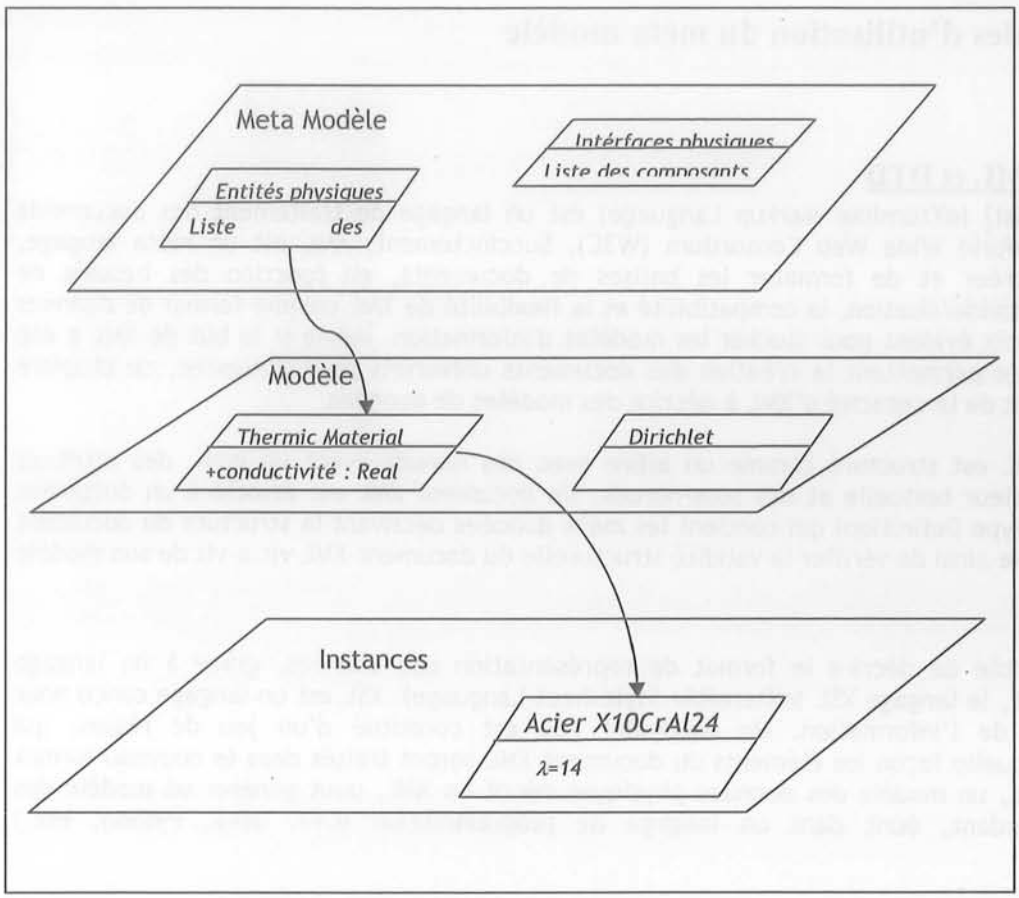


Figure 3-6 : Modélisation comportant 3 niveaux hiérarchiques : méta, modèle et projet

3.6.1 L'approche par méta modélisation

Un *méta modèle (de données) décrit les constituants d'un modèle (de données)* et leurs rapports.

Ainsi le méta modèle est constitué des types utilisés dans la description d'un modèle. En un sens, un méta modèle fourni des informations similaires à celles issues de l'introspection des Java Beans : quelles sont les propriétés et les comportements de cette classe, ... Mais ces informations ne sont plus limitées au contenu strictement utile dans le cadre de la programmation orientée objet. Puisque nous maîtrisons désormais la structure du méta modèle, nous pouvons aussi ajouter des informations supplémentaires : comment s'appelle cette classe ou cet attribut dans différentes langues, quelle est la page d'aide associée à telle classe, cette classe doit elle être connue de l'utilisateur, ... ?

Mais le principal avantage de la méta modélisation réside surtout dans la capacité de spécifier et de d'écrire les comportements du niveau méta. Par exemple, nous pourrons considérer que la restauration/sauvegarde du projet de l'utilisateur final est un service générique, dont la responsabilité incombe au niveau méta. Nous pouvons aussi envisager de bâtir, toujours en s'appuyant fortement sur le niveau méta, des générateurs de dialogue graphique ou textuel génériques.Ces outils du niveau méta s'appliquent pour tous les modèles de données et facilitent donc grandement le travail de l'intégrateur.

3.6.2 Exemples d'utilisation du méta modèle

a). La norme XML et DTD

XML [Harold] [Vlist] (eXtensible Markup Language) est un langage de traitement des documents proposé par le World Wide Web Consortium (W3C). Succinctement, XML est un méta langage, permettant de créer et de formater les balises de documents, en fonction des besoins de l'utilisateur. La standardisation, la compatibilité et la flexibilité de XML comme format de données font de lui un choix évident pour stocker les modèles d'information. Même si le but de XML a été d'offrir un langage permettant la création des documents universels pour l'internet, ce chapitre traitera seulement de la capacité d'XML à décrire des modèles de données.

Un document XML est structuré comme un arbre avec des noeuds ayant un nom, des attributs contenant une valeur textuelle et des sous-nœuds. Un document XML est associé à un document DTD (Document Type Definition) qui contient les méta données décrivant la structure du document XML. Il est possible ainsi de vérifier la validité structurelle du document XML vis-à-vis de son modèle DTD.

Il est aussi possible de décrire le format de représentation des données, grâce à un langage compagnon d'XML, le langage XSL (eXtensible Stylesheet Language). XSL est un langage conçu pour la manipulation de l'information. Un document XSL est constitué d'un jeu de règles, qui déterminent de quelle façon les éléments du document XML seront traités dans le nouveau format de données. Ainsi, un modèle des données physiques décrit en XML, peut générer un modèle des classes correspondant, écrit dans un langage de programmation (C++, Java, Python, etc.) [Georgescu].

L'exemple suivant envisage la description du modèle thermique en utilisant XML.

b).Exemple d'un modèle physique:

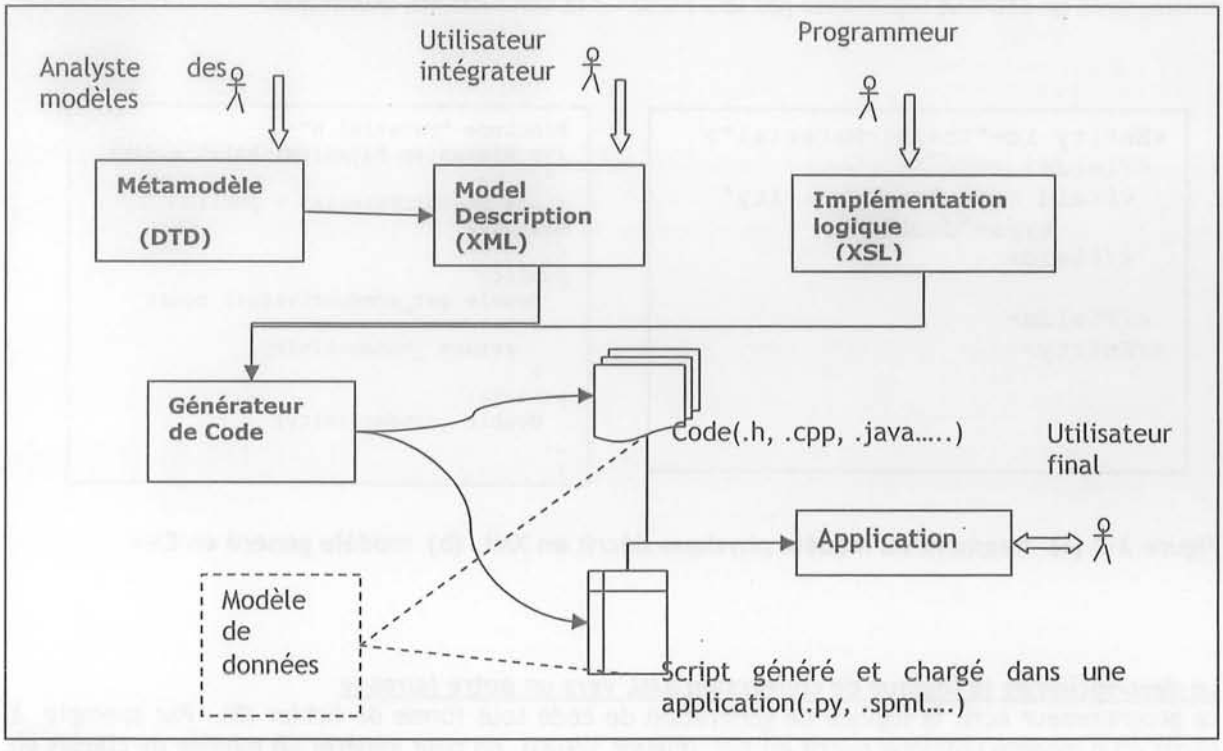


Figure 3-7 méta modélisation en utilisent DTD et XML

La description du Méta modèle

L'analyste de modèles physiques crée le modèle conceptuel qui indique la structure des données pour le générateur de code sous forme de méta données de XML dans un dossier de DTD
Le fragment du méta modèle proposé est composé des entités physiques, avec des champs et des interfaces associées.Une **entité physique** a un nom et des champs optionnelles de type « Field » (Figure 3-8)

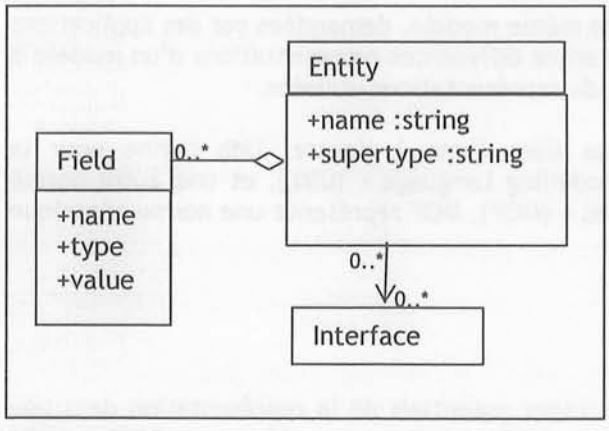


Figure 3-8(a)

```
<!ELEMENT Entity (fields?,
  interfaces?)>

<!ATTLIST Entity
 name NMTOKEN #REQUIRED
 supertype CDATA #REQUIRED>

<!ELEMENT Fields (Field*)>
<!ATTLIST Field
 name NMTOKEN #REQUIRED
 type CDATA #REQUIRED
 value CDATA #REQUIRED>
```

(b) fragment du métamodèle en format DTD

La description du modèle

L'utilisateur intégrateur peut décrire le modèle spécialisé pour la thermodynamique, dans un fichier XML, tout en représentant la structure du méta modèle, décrit antérieurement dans en format DTD(Figure 3-8(b)). Par exemple, dans le modèle thermodynamique, ThermicMaterial est une entité, avec un attribut représenté par le « Field » : la conductivité thermique.

```
<Entity id="ThermicMaterial">
 <Fields>
  <Field name="conductivity"
       type="double">
  </Field>

 </Fields>
</Entity>
```

```
#include "Material.h"
/** Stores an ThermicMaterial's data.
 */
class ThermicMaterial : public
Material
{
public:
  double get_conductivity() const
  {
    return _conductivity;
  }
private:
  double _conductivity;
....
}
```

Figure 3-9 (a) fragment du modèle physique décrit en XML (b) modèle généré en C++

La description de la logique de conversion XML vers un autre langage

Le programmeur écrit la logique de génération de code sous forme de fichier XSL. Par exemple, à partir d'un modèle physique décrit en XML (Figure 3-9(a)), on peut générer un modèle de classes en C++ (Figure 3-9(b)). Le fichier XLS doit contenir le schéma de mappage d'un élément de type « Entity », dans le format d'une classe C++.

c). UML et MOF

Il y a plusieurs manières de représenter un modèle. Par exemple, un modèle (le modèle thermodynamique) peut être représenté par :
- un diagramme UML
- une instance d'un document de type DTD (Document Type Definition) qui peut être instancié.
- un jeu de classes écrites dans un langage O.O. ,(C++, Java, Python), qui décrivent et manipulent des données appartenant au modèle.

Toutes sont les représentations des expressions d'un même modèle, demandées par des applications différentes. La nécessité de créer des interactions entre différentes représentations d'un modèle a suggéré la création d'une base commune aux types de représentations utilisées.

En 1997, l'OMG adoptait deux nouvelles normes liées l'une à l'autre. Une norme pour la modélisation orientée-objet, nommée « Unified Modelling Language » (UML), et une autre norme décrivant fondement d'UML, « Meta Object Facilities » (MOF). MOF représente une norme générique pour les abstractions de haut niveau.

Les points de départ de MOF

La formalisation d'UML a mis en évidence les avantages potentiels de la représentation dans une base générique d'un langage de représentation d'UML (MOF) : Une structure légère et simple, ayant en plus la possibilité d'autodéfinition (MOF doit être son propre méta modèle [MOF]).

MOF est très abstrait dans la mesure où il est conçu comme un méta modèle pouvant représenter d'autres métamodèles.
MOF poursuit un objectif spécifique qui consiste à fournir une plate-forme de méta modélisation adaptée pour le développement des métadonnées et la création d'outils génie logiciel.

(Par exemple un outil de modélisation, basée sur une extension d'UML qui inclue des diagrammes du flux de données, peut être représenté et intégré avec UML en utilisant MOF ; voir notre exemple)

UML est un langage de modélisation objet graphique, conçu pour prendre en charge une large gamme d'opérations spécifiques pour toutes les phases du développement d'un logiciel, dont l'analyse des exigences, la modélisation du domaine et la génération de code pour l'implémentation.

Facilités du MOF

Le haut niveau d'abstraction du MOF a permis la génération d'autres métas modèles. Par exemple CORBA IDL pour UML a été généré à partir d'une description UML basé sur MOF.

Cette abstraction offre aussi des nouvelles possibilités d'interopérabilité des outils de modélisation, particulièrement dans le domaine d'échange de modèles. Même si UML est un excellent outil pour décrire schématiquement n'importe quel système objet, il ne peut pas interchanger des modèles [Tardiveau]. Par contre, un autre standard, basé sur XML, a été développé spécialement pour faciliter l'échange des modèles entre applications, le standard XMI (XML Metadata Interchange)

Ce mécanisme permet pour tous les systèmes basés sur le méta modèle MOF d'échanger des modèles avec autres systèmes.Comme le méta modèle UML, XMI est une instance du méta modèle MOF.
Cette facilité peut constituer un des éléments qui faciliterait les échanges des modèles physiques décrites pour le module DATA.

Structure du MOF

D'un point de vue structurel, MOF se présente sous la forme d'un modèle de données très restreint qui sous sa représentation UML est donnée par la Figure 3-10 [MOF].

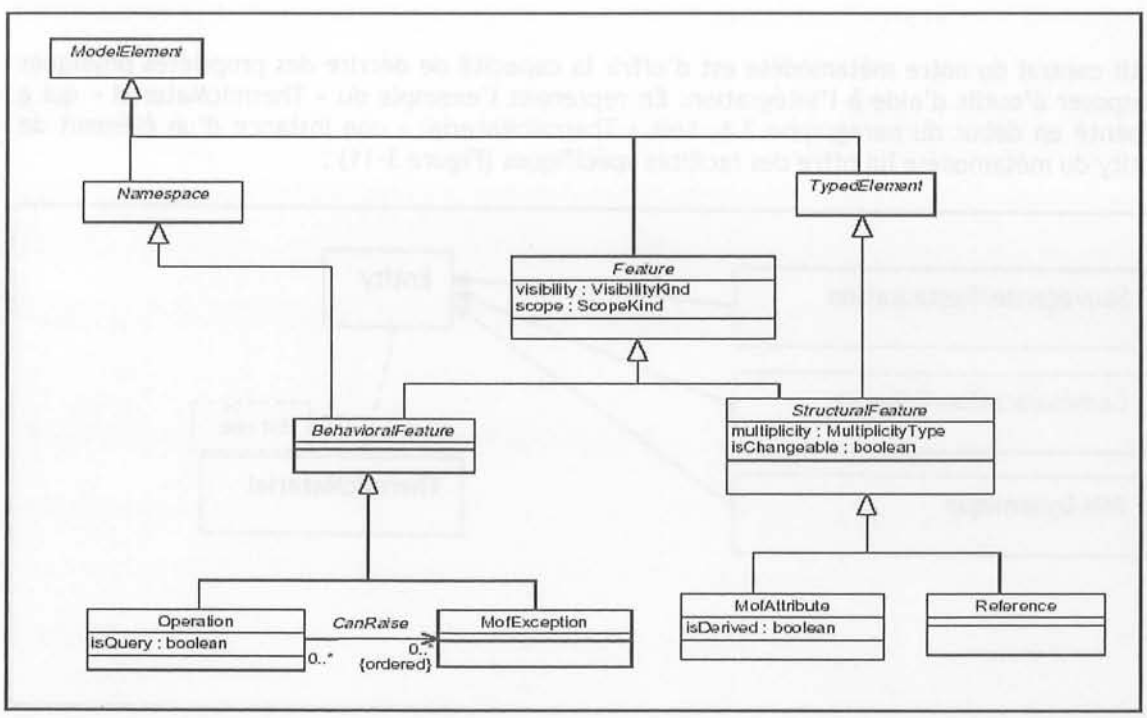


Figure 3-10

3.7 *La solution retenue*

3.7.1 Analyse des forces des 3 approches

A l'issue de cette analyse il est souhaitable de faire la synthèse des principaux avantages que chaque technique pourra amener en voue de réaliser la description des propriétés physiques sur la plate-forme SALOME:

- Le développement interactif
- L'IHM intégrée définie. ---------- Java Beans
- Forte structuration des données
- Dynamicité des modèles de données physiques ------ Métamodèle
- Standardisation des données
- Echange de données physiques ---------- Modèle de données unifié

L'idée a été de trouver une solution technologique médiane qui fédère l'ensemble des points forts de ces trois approches. Mais nous avons tout de même privilégié la méta modélisation qui impose une forte structuration des données et procure la dynamicité des modèles de données.

En conséquence nous avons proposé la création d'un métamodèle dédié pour la description des propriétés physiques, et nous proposons aussi de fournir des mécanismes annexes pour assurer les autres fonctionnalités :

- un moteur de génération d'une IHM dynamique, à partir des informations contenues dans le métamodèle
- un modèle commun décrit à l'aide du métamodèle permettra d'éliminer la redondance et de réaliser des liens entre plusieurs modèles physiques.

L'objectif central du notre métamodèle est d'offrir la capacité de décrire des propriétés physiques et de disposer d'outils d'aide à l'intégration. En reprenant l'exemple du « ThermicMaterial » qui a été présenté en début du paragraphe 3.6. Soit « ThermicMaterial » une instance d'un élément de type Entity du métamodèle lui offre des facilités spécifiques (Figure 3-11) :

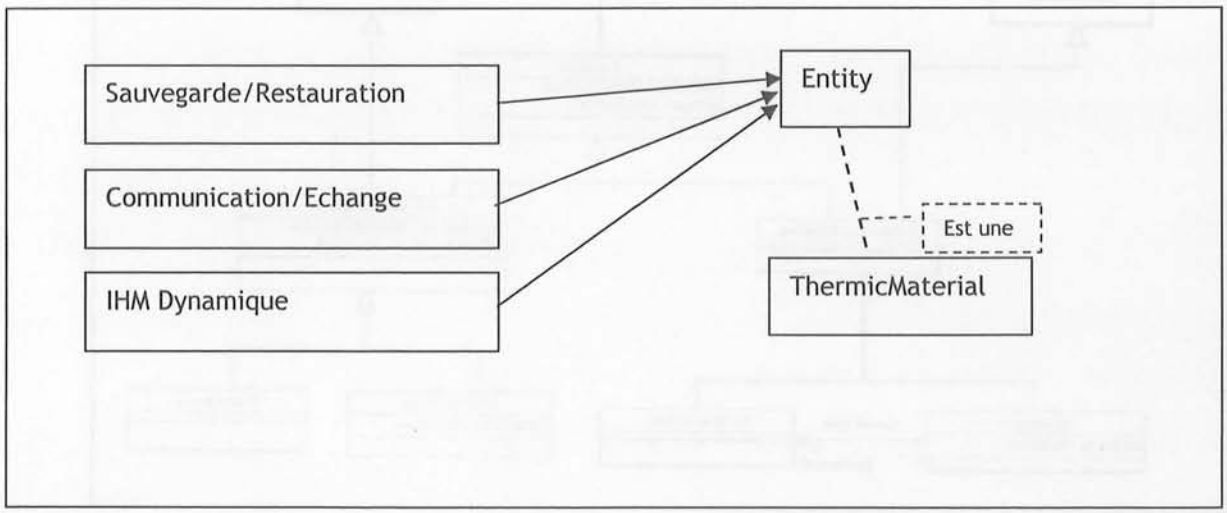


Figure 3-11

Dans l'état actuel de notre réflexion, l'approche retenue ne permet pas une description graphique du modèle physique, comme les Java Beans le permettaient. Nous n'avons pas retenu XML comme formalisme car la complexité des structures que nous souhaitons décrire interdit son utilisation. Enfin notre métamodèle, faute de temps n'a pas été exprimé dans le langage MOF.
Ces trois aspects pouvant être retravaillés par la suite.

3.8 *Conclusion*

Suite à l'analyse des besoins module DATA, nous avons imaginé le modèle conceptuel de ce module.

Il est centré autour d'un composant qui implémente un modèle physique comportant les différents types de données de la physique utilisée. Ce composant gère aussi les instances de ce modèle.

Parmi plusieurs solutions techniques analysées, nous avons choisi de réaliser ce composant en utilisant un nouveau concept, la méta modélisation. En étudiant plusieurs exemples de métamodèles qui sont utilisés actuellement, nous avons pu dégager les spécifications que notre métamodèle doit réunir.

La description effective des propriétés physiques est fondée sur l'existence du métamodèle et sur un langage basé sur le métamodèle, qui seront présentés dans le chapitre suivant.

A part sa fonction de base de description des éléments des propriétés physiques, le module doit réaliser d'autres fonctionnalités - notamment la communication avec les autres modules et l'existence d'une interface avec l'utilisateur. La mise en œuvre de ces autres fonctions est traitée dans le chapitre 5.

4 La description des propriétés physiques

Nous allons présenter dans ce chapitre les principes de base la description des propriétés physiques pour la plate-forme SALOME.

La première partie est dédiée au langage SPML, spécialement conçu pour décrire les propriétés physiques. Nous analyserons ses principaux éléments sémantique et syntaxique.

La deuxième partie présente le « Modèle Commun », modèle de données réutilisable qui a pour vocation d'être une base de travail pour tout solveur souhaitant s'insérer dans l'environnement Salome.

Le chapitre se terminera enfin par un exemple destiné à mettre en évidence la capacité de décrire un problème multi-physiques dans Salome avec le formalisme SPML.

4.1 *SPML : un langage de description des propriétés physiques*

Dans le chapitre précédent, l'analyse des besoins et des solutions nous a permis de conclure sur l'intérêt d'une approche par méta modèle pour la description de tous les éléments du modèle physique à créer.

Dans une conception basée sur la méta modélisation, le méta modèle permet de définir un langage de spécification des modèles [Atkinson]. Si le méta modèle offre la sémantique du modèle, en définissant sa structure, la forme des expressions dans ce langage sera définie par sa syntaxique.

Partant de ce point de vue, nous proposons un langage de spécification pour les propriétés physiques. Ce langage est nommé SPML (SALOME Physics Modelling Language) [SALOME Data Spec]. Nous présenterons successivement la sémantique du langage SPML et sa syntaxe dans cette partie, que nous conclurons par un exemple.

4.1.1 Un méta modèle dédié à la description des propriétés physiques

La modélisation en Génie logiciel peut se découper en quatre couches dont la structure hiérarchique est représentée par la figure suivante [Mateo], [Carlson]:

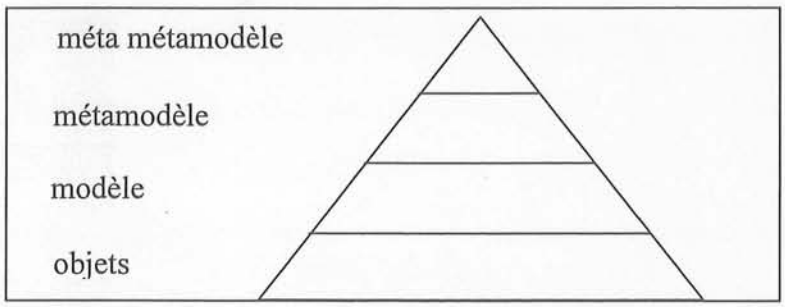


Figure 4-1

Le méta méta modèle : il définit l'infrastructure pour l'architecture du méta modèle et le langage de description du méta modèle. C'est généralement le modèle le plus compact.

Le méta modèle : il définit le langage du spécification du modèle. Par exemple, des entités, des attributs ...

Le modèle : il définit le langage de description pour un domaine d'information. Par exemple des entités spécifiques pour faire la description d'un domaine hydrodynamique : des régions fluides, des matériaux fluides, ... C'est le niveau qui définit complètement l'application à réaliser et en ce sens, c'est à ce niveau que le concepteur du logiciel travaille le plus souvent.

Les objets créés par l'utilisateur, constitué par des instances du modèle. Par exemple, un matériau fluide qui caractérise l'eau de mer.

Pour terminer avec cette description sommaire des niveaux de modélisations, remarquons d'une part, que le niveau « méta méta » est terminal, si il permet de se décrire lui-même, et d'autre part, qu'on passe d'une couche à l'autre par instantiation : le modèle est constitué d'instances du méta modèle, les objets sont des instances des types décrits dans le modèle, ...

Pour notre part, nous nous sommes en réalité limité à une structure en 3 couches, en fusionnant les couches « méta méta » et « méta ». Il nous reste donc à nous concentrer sur le niveau « méta », afin d'apporter à l'intégrateur de solveur les bons outils.

La structure du méta modèle doit définir toutes les entités et toutes les relations qui peuvent exister dans le cadre du langage issu du méta modèle. Dans notre cas le méta modèle doit couvrir les entités et les relations qui peuvent être décrites avec le langage SPML, dédié à la description des modèles de la physique.

Nous allons présenter les éléments du méta modèle qui forment le fondement du langage SPML selon la progression suivante :

a). les éléments de base du méta modèle
b). les éléments réservés à la description des principaux types du langage
c). les structures utilisées à la description de l'interface utilisateur

Les structures de données qui suivent sont répertoriées par leur nom dans le langage SPML. Vu le souhait de diffusion de cette technologie, nous avons utilisé des termes en langue Anglaise.

a). Les éléments de base

Les 3 structures principales du méta modèle permettent de définir un solveur, ses modèles de données et le regroupement physique des structures de données.

Model représente un ensemble de structures de données cohérent. Il a la capacité de réunir les concepts d'un domaine d'activité particulier. Une structure de données est liée à un seul modèle, mais elle peut référencer des structures des données qui appartiennent à d'autres modèles. Un modèle des données peut être dérivé ou spécialisé, en utilisant le mécanisme de l'héritage, pour généraliser ou pour limiter son domaine d'activité.

Application modélise un solveur particulier. Elle peut s'appuyer sur plusieurs ***Model***, afin de les rendre accessibles dans le cadre de la plate-forme. Pour faire cela, ***Application*** a deux propriétés fondamentales :

- Elle regroupe des types ***Model***
- Une structure de type ***Application*** est une entité exécutable, et par son exécution dans le cadre du module DATA, elle rend accessible les modèles contenus par l'application.

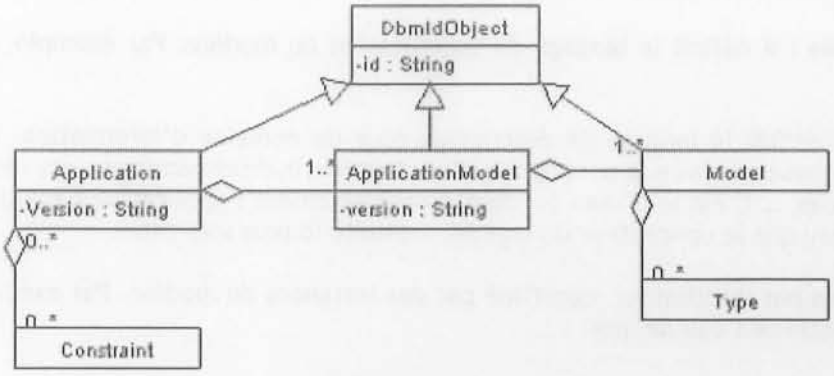
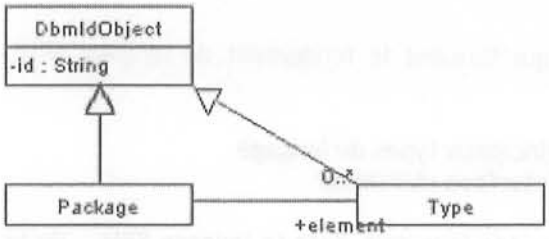


Figure 4-2

Le modèle de données de ces 2 structures de base est présent à la Figure 4-2 en utilisant la représentation de diagrammes de classes UML [Muller].

Package offre la possibilité de structurer les entités du modèle dans une structure arborescence semblable aux répertoires. Cette structure est destinée à gérer la structuration physique du code. L'exemple Figure 4-3(b) montre un choix de structuration physique par solveur.



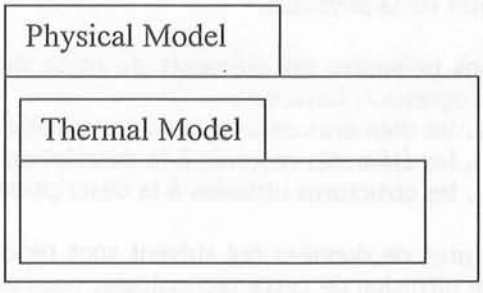


Figure 4-3 (a) (b)

La structure UML du *Package* est représentée dans la Figure 4-3(a). On peut observer la structure récursive des packages. Ainsi un *Package* pourra en contenir d'autres.

b). Les éléments destinés à décrire les principaux types du langage

Dans le méta modèle, les types qui peuvent être modélisés sont (Figure 4-4) :

- Les types intrinsèques
- Les énumérations
- Les interfaces
- Les entités

Tous ces types appartiennent obligatoirement à un modèle de données (instance du type *Model*) et optionnellement à une *Hierarchy* (cf § c). Pour réaliser ce comportement commun, ils héritent d'une classe abstraite *Type* constituée de:

- un identificateur (obligatoire)
- une référence vers un modèle de données (obligatoire)

- une référence vers une hiérarchie (optionnelle)

Les types ***Intrinsic*** représentent les types de base manipulés par l'***Application*** : réels, entiers, chaînes, image, identificateur, ...
Un type ***Intrinsic*** hérite de ***Type*** et est caractérisé par un type de base choisi parmi : entier, réel, numérique, chaîne de caractères, numérique ou chaîne, booléen, void, ou référence vers objet, qui caractérise son formatage interne.

Les types ***Enumeration*** sont utilisés pour définir des collections de valeurs nommées. Une énumération a en plus des attributs déjà décrits dans ***Type***, une liste d'éléments sous forme de chaînes de caractères.

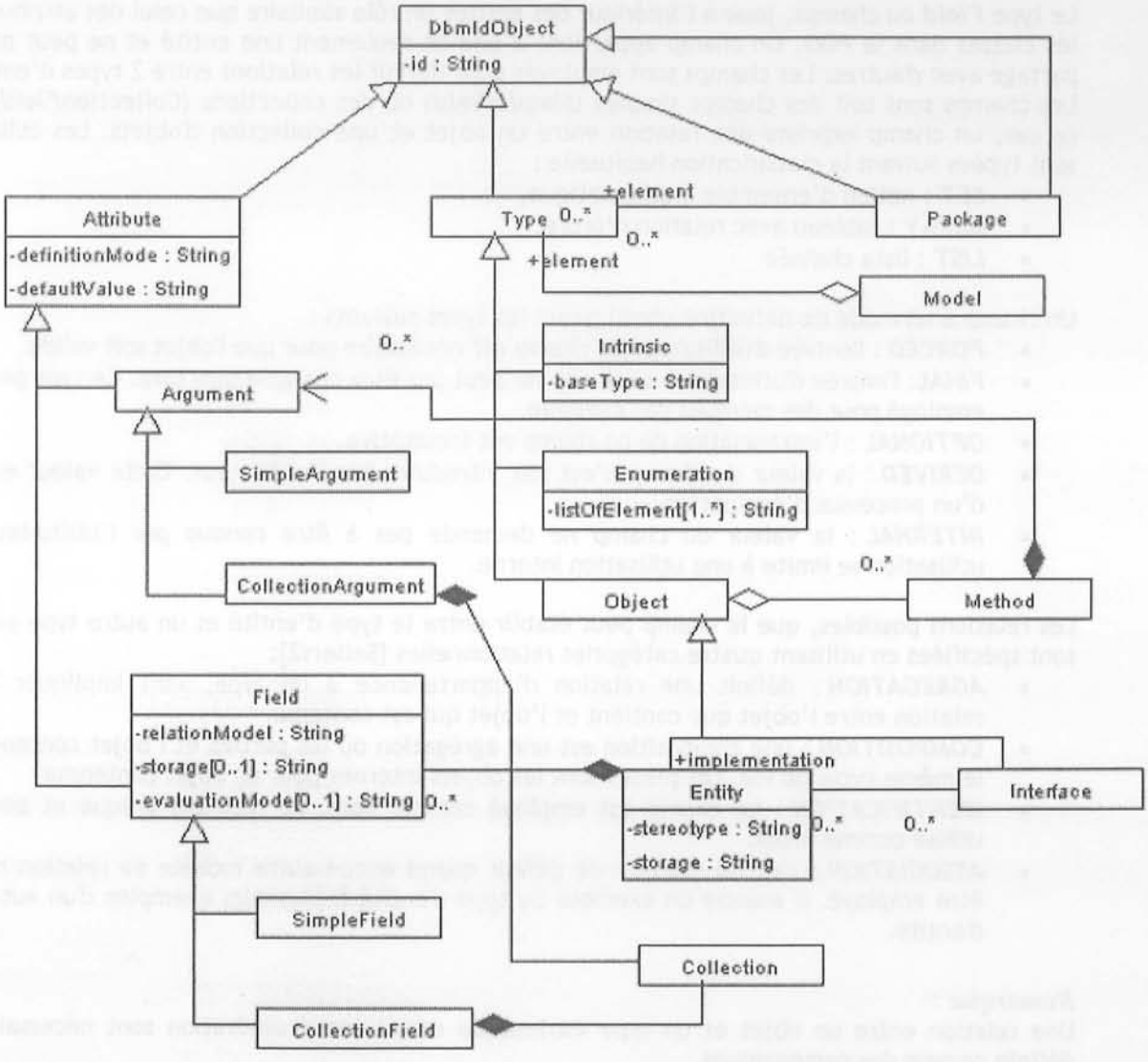


Figure 4-4

Le type ***Interface*** est utilisé pour décrire le concept d'interface, en conformité à la programmation orientée objet (POO). Les interfaces disposent seulement des déclarations des méthodes modélisant le comportement. Une interface peut étendre une ou plusieurs autres interfaces.

Le type ***Entity*** représente le concept de base dans la description d'un modèle de données. Une entité est définie par ses données (***fields***) et par ses services (***methods***).

Cette structure est évidemment similaire à la notion de classe de la POO. A ce titre, elle dispose de l'information concernant les interfaces implantées par le type d'entité. Un type d'entité dispose également d'un stéréotype. Les principaux stéréotypes proposés sont ***ABSTRACT*** et ***CONCRETE***, mais d'autres stéréotypes sont définis matérialisant des comportements plus fins et plus évolués.

Si la notion de type d'entité est assimilable à la notion de classe de la POO, elle l'étend et l'adapte aussi pour d'une part permettre la gestion du grand nombre de type d'entités : c'est le rôle de l'ajout d'une référence vers un modèle, et d'autre part, comme nous le verrons par la suite, pour faciliter l'intégration rapide dans l'environnement SALOME : cela se concrétise par l'introduction d'informations destinées à l'interface utilisateur et au stockage sur disque des données.

Le type ***Field*** ou champs, joue à l'intérieur des entités un rôle similaire que celui des attributs pour les classes dans la POO. Un champ appartient à une et seulement une entité et ne peut pas être partagé avec d'autres. Les champs sont employés pour définir les relations entre 2 types d'entités.
Les champs sont soit des champs simples (***SimpleField***) ou des collections (***CollectionField***). Dans ce cas, un champ exprime une relation entre un objet et une collection d'objets. Les collections sont typées suivant la classification habituelle :

- ***SET*** : notion d'ensemble mathématique,
- ***ARRAY*** : tableau avec relation d'ordre,
- ***LIST*** : liste chaînée

Un champ a un mode de définition choisi parmi les types suivants :

- ***FORCED*** : l'entrée d'utilisateur du champ est nécessaire pour que l'objet soit valide.
- ***FINAL***: l'entrée d'utilisateur du champ ne peut pas être changée plus tard. Ce type peut être employé pour des marques par exemple.
- ***OPTIONAL*** : l'instanciation de ce champ est facultative.
- ***DERIVED*** : la valeur du champ n'est pas introduite par l'utilisateur. Cette valeur est issue d'un processus d'évaluation.
- ***INTERNAL*** : la valeur du champ ne demande pas à être connue par l'utilisateur. Son utilisation se limite à une utilisation interne.

Les relations possibles, que le champ peut établir entre le type d'entité et un autre type existant, sont spécifiées en utilisant quatre catégories relationnelles [Sellers2]:

- ***AGREGATION*** : définit une relation d'appartenance à un type, sans impliquer aucune relation entre l'objet que contient et l'objet qui est contenu.
- ***COMPOSITION*** : une composition est une agrégation où les parties et l'objet conteneur ont le même cycle de vie. Les pièces sont les objets internes pour un objet conteneur.
- ***IDENTIFICATION*** : Le champ est employé comme nom. Le nom est unique et peut être utilisé comme index.
- ***ASSOCIATION*** : c'est la relation de défaut quand aucun autre modèle de relation ne peut être employé. Il associe un exemple du type d'entité à plusieurs exemples d'un autre type d'entité.

Remarque :
Une relation entre un objet et un type intrinsèque et/ou une énumération sont nécessairement définie comme des compositions.

En résumant, un champ est composé de :

- Un type relatif à choisir parmi tous les types existants (obligatoires);
- Une information pour préciser ***SimpleField*** ou ***CollectionField***.
- Un mode de définition à choisir entre ***FORCED***, ***FINAL***, ***OPTIONAL***, ***DERIVED***, ***INTERNAL*** (obligatoire);

- Un modèle de relation à choisir parmi : ***ASSOCIATION***, ***AGREGATION***, ***COMPOSITION***, ***IDENTIFICATION*** (obligatoire);
- Une information supplémentaire nécessaire pour le stockage à choisir entre ***TRANSIENT*** et ***PERSISTENT*** (obligatoire)
- Une valeur par défaut du champ en cas de non remplissage (facultative);

Le type ***Method*** joue le rôle des méthodes dans la POO. Une méthode appartient à un et un seul type d'entité. Elle définit les actions réalisables par un type d'entité. Comme dans le cas des types d'entités, les méthodes étendent la notion de méthode connue dans la POO en particulier par l'introduction d'informations destinées à l'interface utilisateur. Les méthodes sont définies par des *Arguments*.

Le type ***Argument*** représente les paramètres des méthodes. Un argument particulier appartient à une et seulement une méthode et ne peut pas être partagé par d'autres. Les arguments peuvent être de type ***SimpleArgument*** ou de type ***CollectionArgument***. Leur structure s'approche de celle des champs.

c). Les structures pour la description de l'interface utilisateur

En plus de toutes les données précédentes, il est possible et utile d'ajouter de l'information supplémentaire liée à la gestion de l'interface utilisateur. Un moteur d'interface utilisateur va être réalisée qui interprétera ces informations pour créer dynamiquement le dialogue avec l'utilisateur.

Ce dialogue automatisé devra pouvoir prendre deux aspects:

- Une interface utilisateur textuelle (TUI) sous forme de lignes de commandes
- Une interface utilisateur graphique (GUI) sous forme de boites de dialogue

Les deux modes sont réalisés grâce à l'information d'interface utilisateur (UI) associée aux structures de données précédentes. Ainsi l'information d'UI apporte principalement la possibilité de décrire les textes qui apparaîtront dans les boites de dialogue et cela dans différentes langues, ainsi que les mots réservés du langage de commande. Toutes ces informations sont généralement facultatives. Si elles ne sont pas fournies, ce sont les noms de structure de données qui sont employés, mais, dans ce cas, le dialogue est beaucoup moins explicite et naturel.

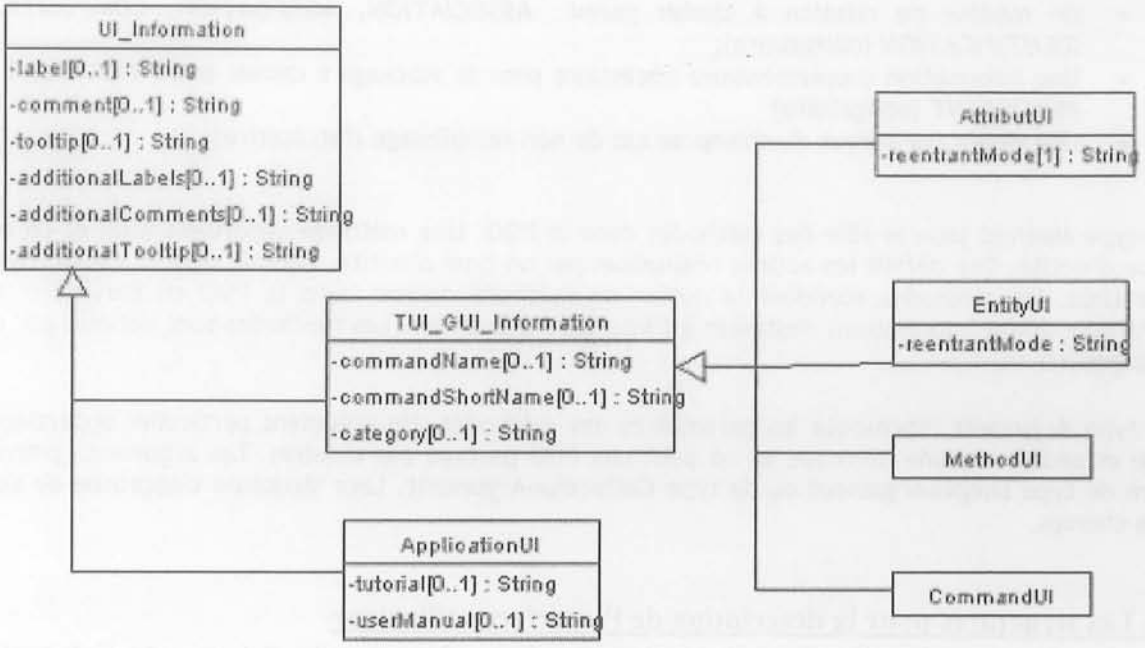


Figure 4-5

Les informations nécessaires pour la description de l'interface utilisateur peuvent être séparées en informations communes pour tous les types du modèle et informations spécialisées par type.

La Figure 4-5 montre la structure de données permettant de définir les UI, représentés par ***UI_Information***, ***TUI_GUI_Information***, ***Application_UI*** qui sont déclinés dans des types spécialisés pour spécifier l'interface utilisateur pour des types spécifiques du modèle, tel que ***EntityUI*** pour le type ***Entity***, ou ***AttributeUi***, pour le type ***Field***.

En complément de cela, la structure ***Hierarchy*** permet de définir une arborescence principalement utilisée pour la présentation de l'arbre des données dans SALOME.

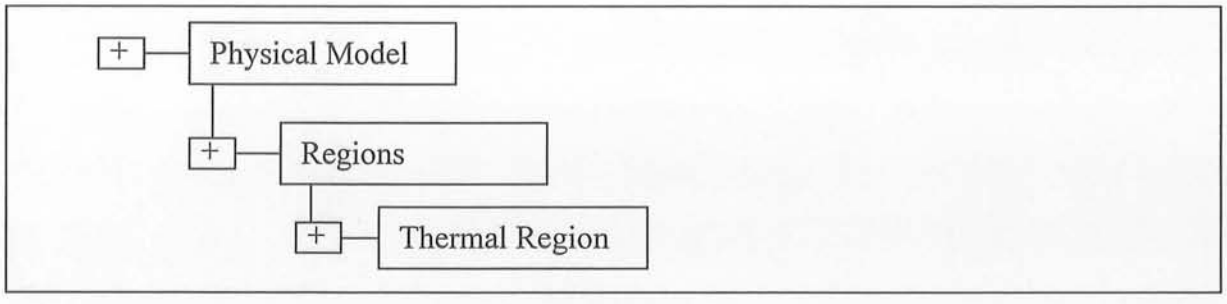


Figure 4-6

Cette structuration de l'arbre permet ainsi de retrouver les éléments d'un modèle physique, sans connaître la structure précise du modèle de données. Il sera suffisant de regarder dans la hiérarchie d'un type abstrait, par exemple une région, pour accéder à la particularisation de celle-ci (par exemple une région thermique, représentée par «Thermal Region » dans la Figure 4-6).

d). Schéma global

Nous avons organisé les éléments du méta modèle décrit ci-dessus dans une structure fonctionnelle issue de la structure du MOF [MOF], [Tardiveau].

Un premier niveau de structuration sépare le méta modèle en 2 paquetages (Figure 4-7) :

- le paquetage qui permet de décrire structure des données ***DataPackage***
- le paquetage qui contient la structure de l'interface utilisateur ***UserInterfacePackage***.

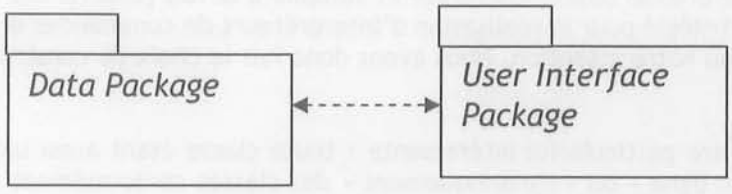


Figure 4-7 : Les paquetages principaux du méta modèle

Le paquetage des données est divisé lui-même divisé en trois sous paquetages : ***ApplicationModelPackage***, ***TypeAttributePackage*** et ***CollectionPackage***(Figure 4-8).

- ***ApplicationModelPackage*** contient les structures nécessaires pour implémenter la colonne vertébrale de la description d'un modèle physique sur laquelle seront attachés tous les autres éléments d'un modèle. Il contient principalement les structures ***Model*** et ***Application***.
- ***TypeAttributePackage*** rassemble une grande partie des éléments réservés à la description des principaux types du SPML.
- ***CollectionPackage*** contient les structures nécessaires pour décrire les types des collections.

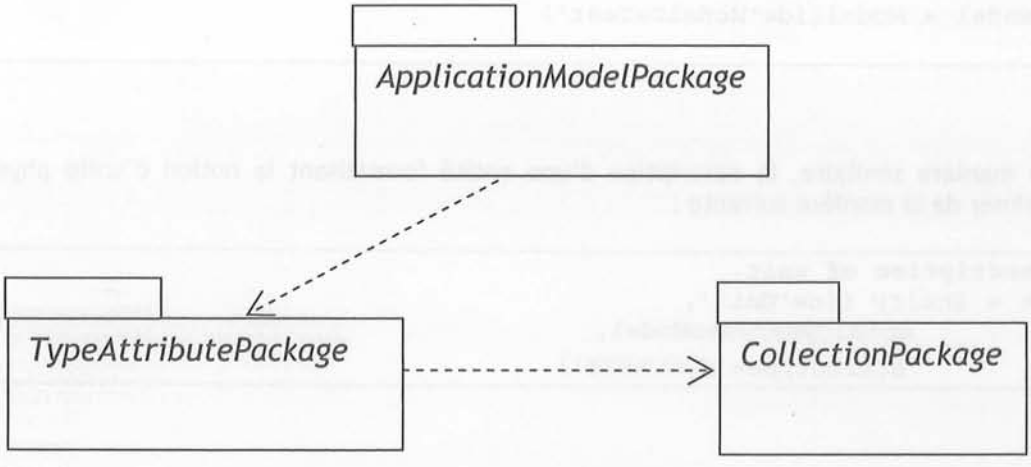


Figure 4-8 : les sous paquetages

4.1.2 La syntaxe du SPML

Le méta modèle précédent est un premier pas vers un langage, le langage SPML, qui sera utilisé par l'utilisateur intégrateur pour décrire son propre solveur. Partant du méta modèle décrit précédemment, il reste à définir une syntaxe pour complètement spécifier le SPML. Pour cela nous nous sommes fortement inspiré du langage de programmation Python.

Python est un langage orienté objet, interprété et compilé à la fois [Lutz1]. Ces deux particularités en font un langage privilégié pour la réalisation d'interpréteurs de commandes et c'est tout d'abord cet aspect qui a retenu notre attention. Nous avons donc fait le choix de construire un interpréteur de SPML en Python.

Mais Python a une autre particularité intéressante : toute classe étant aussi un objet, il est donc possible de créer « en ligne » ou « dynamiquement » des classes conformément aux objectifs de la méta modélisation. Dans notre cas, cette spécificité permettra de changer dynamiquement le modèle de données décrit par l'utilisateur intégrateur. Le langage Python a donc été aussi choisi pour réaliser l'implémentation du méta modèle.

Restons en là concernant Python momentanément, sachant que le choix du langage Python sera complètement motivé dans le chapitre 5.

La sémantique du SPML est malgré tout influencée par le fait que l'interpréteur de SMPL est réalisé en Python. En gardant la logique de la méta modélisation, le modèle physique sera composé d'instances du méta modèle. Ainsi, l'approche retenue pour le langage SPML repose principalement sur des appels vers les constructeurs des classes du méta modèle, suivant la syntaxe Python.

Par exemple, la description d'un nouveau modèle a une syntaxe qui correspond à l'instanciation de la classe ***Model***, du méta modèle :

```
monModel = Model(id='ModelDeTest')
```

D'une manière similaire, la description d'une entité formalisant la notion d'unité physique peut s'exprimer de la manière suivante :

```
# Description of unit.
unit = Entity (id='Unit',
               modelOwner=monModel,
               stereotype='CONCRETE')
```

Nous allons par suite présenter un exemple de modélisation de la physique en langage SPML, qui illustrera les notions présentées jusqu'à présent.

4.1.3 Illustration du langage SPML : description d'un modèle électrostatique

Nous avons choisi d'illustrer le langage SPML avec l'exemple des problèmes d'électrostatique. Plus spécifiquement, et afin de limiter cet exemple, nous nous contenterons d'une modélisation bidimensionnelle de base, capable de calculer par exemple le comportement d'un condensateur plan considéré comme infini dans sa troisième dimension, ce qui revient à calculer le potentiel électrique dans un volume éventuellement chargé situé entre deux surfaces conductrices fixées à deux valeurs de potentiel électrique.

Pour réaliser cette étude, l'utilisateur doit spécifier les propriétés physiques des régions étudiées et les valeurs des potentiels des électrodes.

Du point de vue du module DATA de SALOME, il est nécessaire de décrire un modèle dédié au domaine de l'électrostatique, ce qui revient à décrire l'ensemble des concepts contenus dans ce domaine.

Nous allons parcourir les étapes de la description de ce modèle physique.

a). L'analyse du domaine de l'électrostatique

Seul le résultat de cette analyse sera exposé. Ce résultat fait en particulier apparaître des entités abstraites qui évidemment sont le fruit d'une réflexion intégrant des contraintes de réutilisation dépassant le cadre strict de l'électrostatique. Par exemple des régions qui peuvent comporter des matériaux et des formulations physiques. Ces régions seront modélisées par une classe nommée ***Region***, abstraite, concrétisée par la classe ***SurfacicRegion*** pour les problèmes en 2 dimensions. Les régions surfaciques électrostatiques possèdent une densité de charge ***rho***.

Une classe abstraite pour la description des matériaux (***Material***) nous permet d'introduire plusieurs types de matériaux.
La description d'un matériau diélectrique s'effectue par la classe concrète ***DielectricMaterial*** , qui contient un attribut définissant la permittivité électrique.

Le concept général de condition aux limites peut être modélisé à l'aide d'une classe abstraite ***BoundaryCondition***. Deux autres classes abstraites ***DirichletCondition*** et ***Neumann Condition*** héritent de la classe ***BoundaryCondition*** et modélisent les conditions aux limites classiques de Neumann et Dirichlet.
La description d'un potentiel électrique uniforme impose la création d'une nouvelle classe ***ElectricDirichletCondition***.

Enfin, le solveur électrostatique a aussi besoin de types numériques « spécifiques ». Par exemple la création de la classe ***RealConstantScalar*** permet l'association d'une unité à une constante réelle.

La Figure 4-9 présente la représentation UML des classes utilisées dans le modèle électrostatique.

b). La description de chaque classe avec le langage SPML

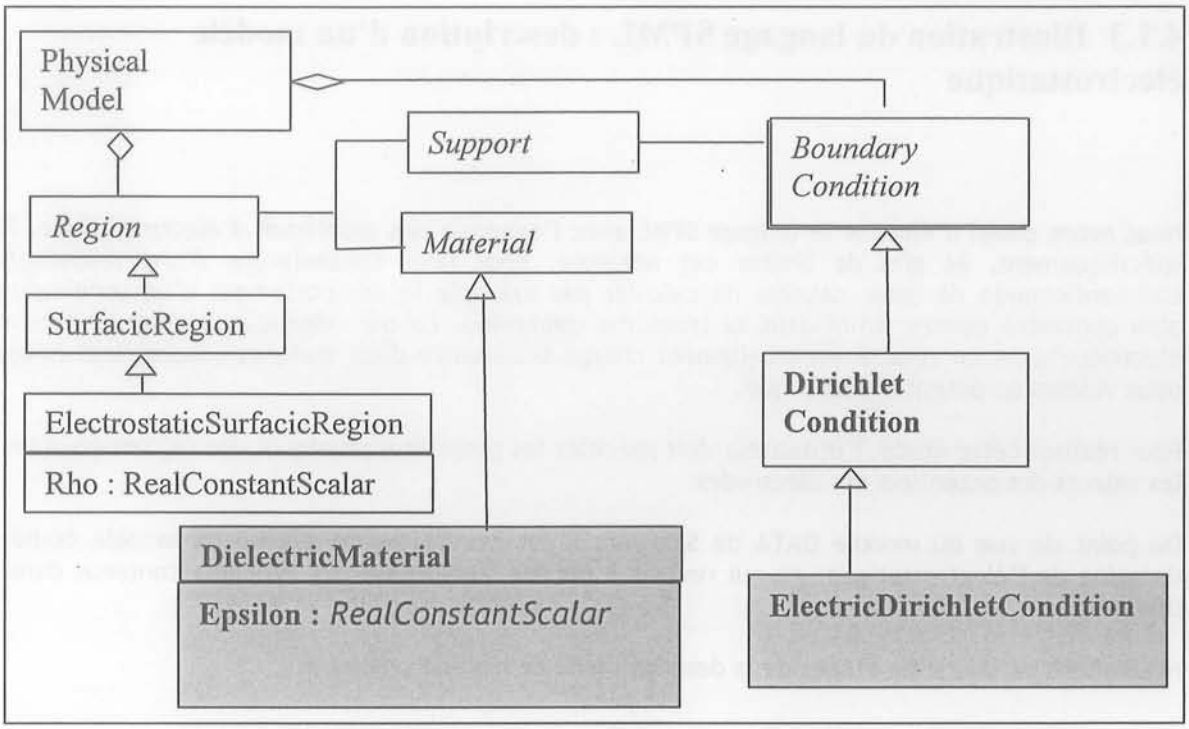


Figure 4-9 La structure du modèle électrostatique crée utilise des classes abstraites. Les carrés gris représentent les classes analysées ici en détail

```
#-------------------------------------------------------------------------
# a specific dielectric material, used in electrostatic formulation
# Material
#       |
#       |---DielectricMaterial
#                   +numericalFunction: epsilon

dielectric_material = Entity(id='DielectricMaterial',
            modelOwner=ElectrostaticModel,
            stereotype='CONCRETE',
            supertype=Entity['Material'])

dielectric_material_epsilon=SimpleField(id='epsilon',
                        relatedType=RealConstantScalar,
                        definitionMode='FORCED',
                              stereotype='AGREGATION',
                              uiInformation=AttributeUi(defaultLabel='Permitivity',
                                                    reentrantMode='NOT_REENTRANT'),

dielectric_material.fields=[ dielectric_material_epsilon]

dielectric_material.uiInformation = EntityUi(defaultLabel='Dielectric Material',
                                  reentrantMode='REENTRANT')
```

Listing 1 : Description complète d'un matériau dielectrique.

Le listing 1 présente l'implémentation en SPML de l'intégralité du modèle électrostatique.

Pour mieux comprendre le langage SPML, étudions en détails une suite d'instruction, par exemple la description du type ***DielectricMaterial*** .

Elle commence par la définition du nouveau type nommé par le paramètre ***id*** du constructeur :

```
dielectric_material = Entity(id='DielectricMaterial',
```

```
            modelOwner=ElectrostaticModel,
            stereotype='CONCRETE',
            supertype=Entity['Material'])
```

Pour être utilisable, l'entité doit appartenir à un modèle physique. Le paramètre ***modelOwner*** nous permet de définir ce modèle. Pour notre exemple, l'entité ***DielectricMaterial***, appartient au modèle physique électrostatique ***ElectrostaticModel***.
Le paramètre ***supertype*** spécifie l'entité qui est héritée. Ici l'entité ***DielectricMaterial*** hérite la classe abstraite ***Material***.

La suite de l'implémentation réalise la définition de la structure de l'entité ***DielectricMaterial*** en présentant ses attributs (***fields***)

```
dielectric_material_epsilon=SimpleField(id='epsilon',
                        relatedType=RealConstantScalar,
                        definitionMode='FORCED',
                        stereotype='AGREGATION',
```

L'attribut ***epsilon*** exprime la permittivité électrique du matériau, son type est défini par le paramètre ***relatedType*** , ici représenté par un constant scalaire réel (***RealConstantScalar***).

L'attribut « ***stereotype*** » va spécifier la relation qui existe entre l'attribut epsilon et l'entité qui le contient.

La description concernant l'interface graphique de l'attribut peut spécifier les noms utilisés pour définir les champs dans la fenêtre de dialogue (« Permitivity »).

```
                        uiInformation=AttributeUi(defaultLabel='Permitivity',
                                                  reentrantMode='NOT_REENTRANT'),
```

Les définitions des attributs, ceux-ci sont attribués à l'entité :

```
dielectric_material.fields = [dielectric_material_epsilon]
```

Enfin, le texte proposé à l'utilisateur dans sa langue native est renseigné, pour le type ***DielectricMaterial***

```
dielectric_material.uiInformation = EntityUi(defaultLabel='Dielectric Material',
                                  reentrantMode='REENTRANT')
```

L'information graphique ***reentrantMode='REENTRANT'*** précise qu'après la création d'une instance de matériaux diélectrique, le gestionnaire de l'interface graphique restera dans le mode de création d'une instance de matériaux diélectrique.

4.2 *Le modèle Commun*

L'un des moyens pour minimiser les coûts de l'intégration est la possibilité de réutiliser des éléments déjà existants dans la description des nouveaux modèles physiques. La constitution d'une bibliothèque éléments généraux et réutilisables est une solution largement utilisée dans la POO, par exemple dans le cas des librairies de classes dédiées.

Dans notre cas, on peut augmenter l'efficacité de l'intégrateur en recherchant les éléments réutilisables qui sont communs conceptuellement pour toutes les disciplines de la physique.

Comme pour toute autre description d'un modèle physique, dans le cadre de la plate-forme SALOME, la description du modèle commun sera réalisée en langage SPML. Pour réutiliser le modèle commun, un nouveau modèle doit d'une part référencer le Modèle Commun dans sa description, et d'autre part, lorsque cela s'avère possible, hériter du Modèle Commun.

Le Modèle Commun a été développé en identifiant les besoins d'utilisateur dans différents domaines physiques. Pour être compatible avec n'importe quel modèle physique, le modèle commun est petit (environ 20 classes), essentiellement abstrait et constitué de 3 paquets, liés à trois fonctionnalités différentes :

- une partie qui concerne les aspects numériques de base,
- une décomposition s'appuyant sur la structure de l'objet à analyser,
- une décomposition visant à décrire la physique de l'objet à analyser.

Dans les paragraphes suivants nous allons présenter ces 3 paquets.

4.2.1. Les définitions des types numériques

Un modèle destiné au calcul physique se doit de connaître des types de base comme des formules ou des unités. En ce sens, ces types font naturellement partie du Modèle Commun. La Figure 4-10 présente la structure des types numériques de base.

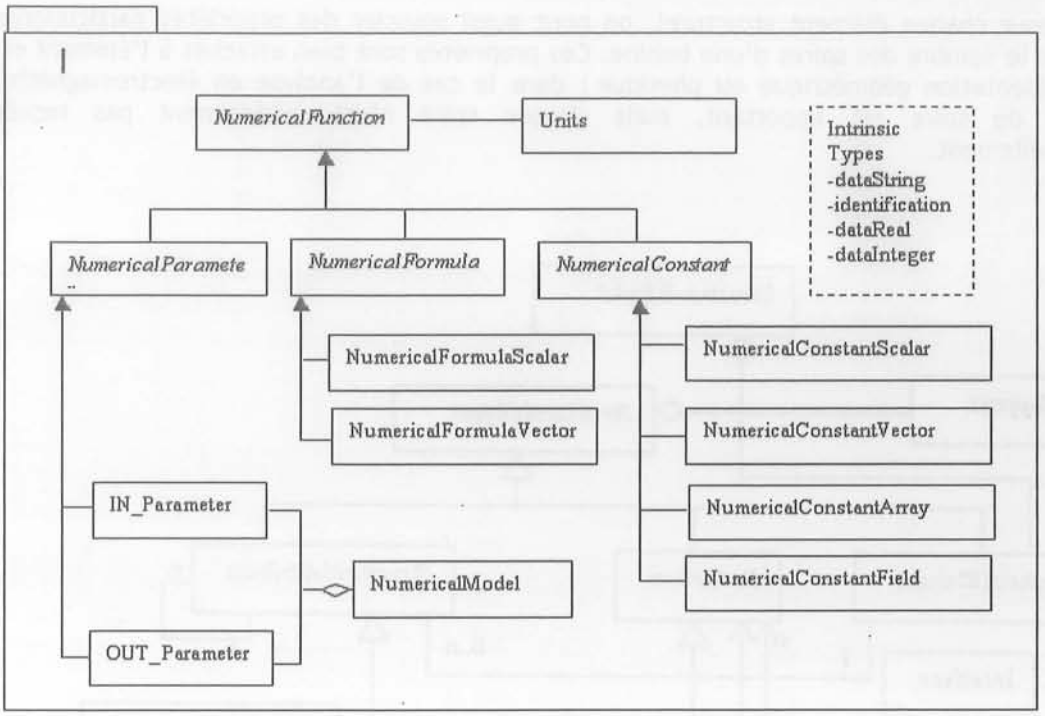


Figure 4-10

L'entité « ***NumericalFunction*** » représente le super type de toute entité de la partie numérique, susceptible d'être inclue dans une description physique ou structurelle. L'entité ***NumericalConstant*** est le cas particulier d'une constante numérique. Le type ***NumericalFormula*** permet d'apporter la gestion de formules dans la description de la physique.

4.2.2 La décomposition structurelle de l'objet à analyser

Au delà de toute physique, un objet à analyser se décompose en pièces : c'est sa décomposition structurelle. Elle exprime la structure interne du système à analyser. Cette décomposition structurelle est intéressante à exprimer, même si elle est généralement négligée. C'est en effet le seul modèle stable dans une analyse multi physique, la géométrie et le modèle physique lui-même étant susceptibles de changer d'une physique à l'autre.

Les éléments du modèle structurel représentent une partie du dispositif étudié. Ils peuvent avoir une référence optionnelle vers un support constitué d'un élément géométrique. L'aspect facultatif de la présence d'une géométrie peut paraître surprenante. Mais en réalité, bon nombre de pièces ne sont pas réellement représentées dans le modèle géométrique utilisé pour conduire une analyse : soit il s'agit de détails non modélisables, soit leur modèle est simplifié (remplacement d'une partie de la géométrie par une masse condensée possédant des propriétés mécaniques : moments d'inertie, ...). Un élément structurel peut aussi disposer d'une référence vers un matériau contenu dans une bibliothèque.
Un élément structurel doit définir ses relations avec les autres éléments de la structure. Ainsi il existe un diagramme de relations entre les éléments qui composent le dispositif : c'est à proprement parler le modèle structurel du dispositif.

Enfin, pour chaque élément structurel, on peut aussi associer des propriétés particulières : par exemple le nombre des spires d'une bobine. Ces propriétés sont bien attachés à l'élément et non à sa représentation géométrique ou physique : dans le cas de l'analyse en électromagnétisme, le nombre de spire est important, mais chaque spire n'est évidemment pas représentée individuellement.

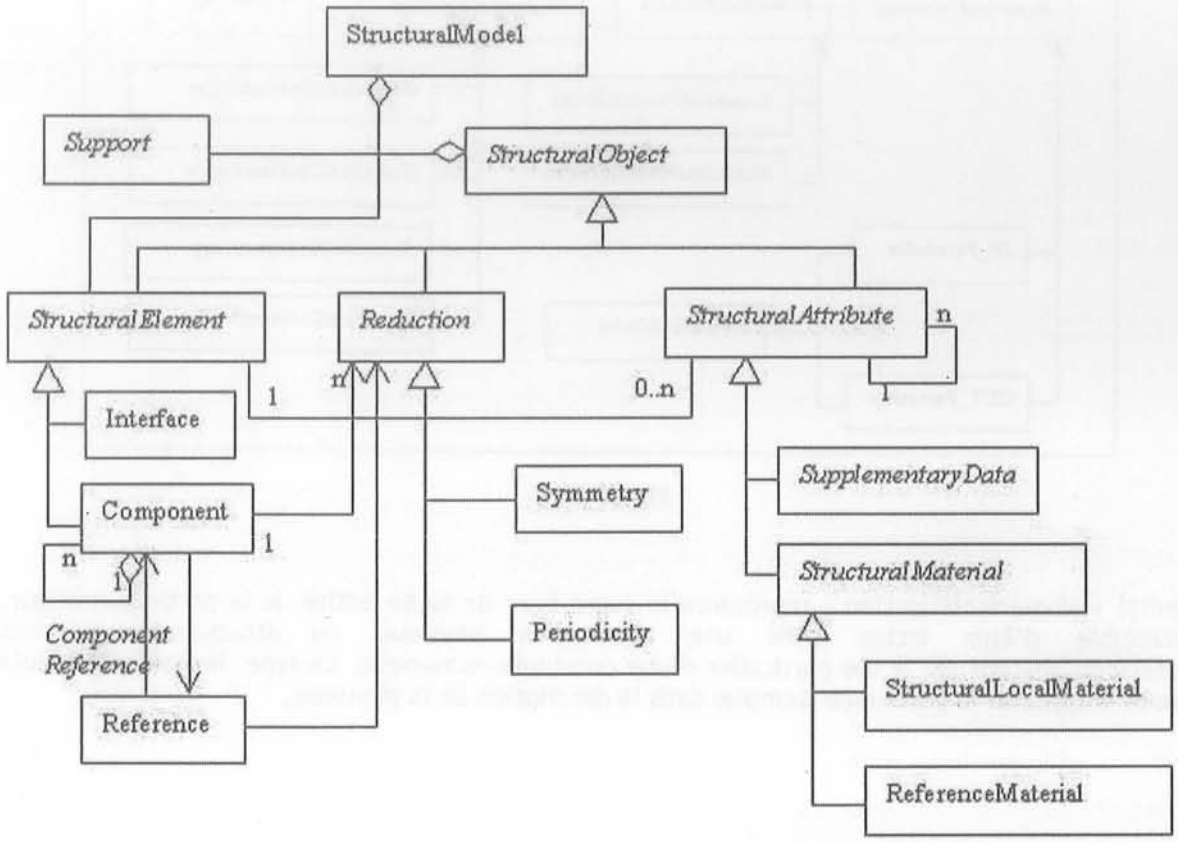


Figure 4-11 : Modèle de données pour la description de la décomposition structurelle de l'objet à analyser.

La Figure 4-11 présente la partie du Modèle Commun dédié à la description structurelle dans le cadre d'un modèle physique. La classe ***StructuralObject*** représente la généralisation des types structurels. Son héritage donne trois directions pour la description structurelle. On peut distinguer ainsi

- une description des éléments qui composent le dispositif.
- une description des symétries ou périodicités,
- une description des attributs de la structure.

La description des éléments du système repose sur 2 catégories : les éléments (***Component***) à proprement parler et les interfaces (***Interface***) entre ces éléments. La présence d'interface est importante et joue souvent un rôle majeur dans la modélisation physique.

La description des symétries. Dans le cadre du Modèle Commun, la classe ***Reduction*** permet de préciser la topologie du dispositif complet ainsi que de chacun de ses constituants en termes de symétries ou périodicités. Cela permet de réduire l'analyse à la partie élémentaire du dispositif ou de ses constituants.

La description des attributs de la structure permet d'associer un matériau référencé dans une bibliothèque à un élément de structure, et aussi d'ajouter des propriétés qui sont utiles à la modélisation de l'élément.

Pour rendre plus concrets ces concepts, prenons un exemple. Dans le cas d'un système de chauffage par induction d'engrenages, la décomposition structurelle pourrait être la suivante : «Gear » pour l'engrenage à chauffer, « Coil » et « Air » pour l'environnement qui contourne la pièce. Ces composants seront des objets de type ***Component*** et ont comme support géométrique les volumes qui définissent l'engrenage, l'inducteur, et l'air.

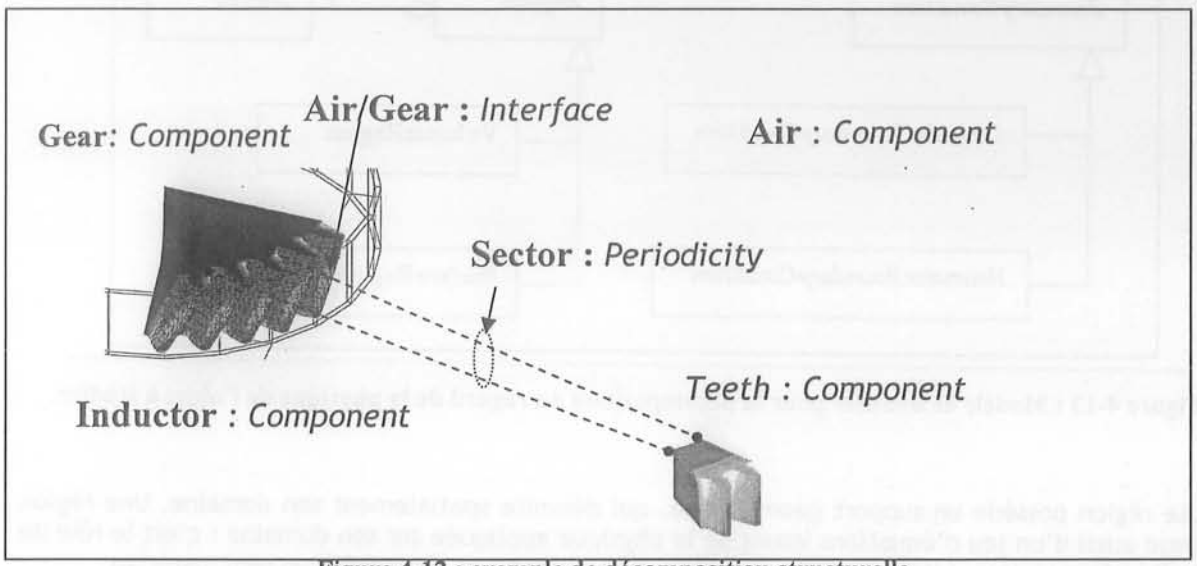


Figure 4-12 : exemple de décomposition structurelle.

Dans la Figure 4-12 nous pouvons observer la représentation des composants « Coil » et « Air ». Un objet « Air/Gear » de type ***Interface***, ayant comme support géométrique la surface de délimitation entre l'engrenage et l'air ambiant pourra définir l'interface entre les composants « Air » et « Gear », interface utile à la modélisation des échanges thermiques.

Il est possible de définir le composant « Gear » en utilisant un composant « Teeth », associé à un objet de type ***Periodicity***, qui définit la périodicité du motif d'une dent dans l'engrenage.

4.2.3 La décomposition au regard de la physique de l'objet à étudier

La décomposition physique joue le rôle de liant entre la description géométrique et structurelle et l'application des lois de la physique sur ce système. Cette description achèvera la spécification du problème pour être envoyée au solveur.

La généralisation composée des principaux types de données communs à toutes les physiques donne lieu aux concepts de régions, conditions aux limites, et sources.

La Figure 4-13 présente les principaux éléments de la décomposition physique.

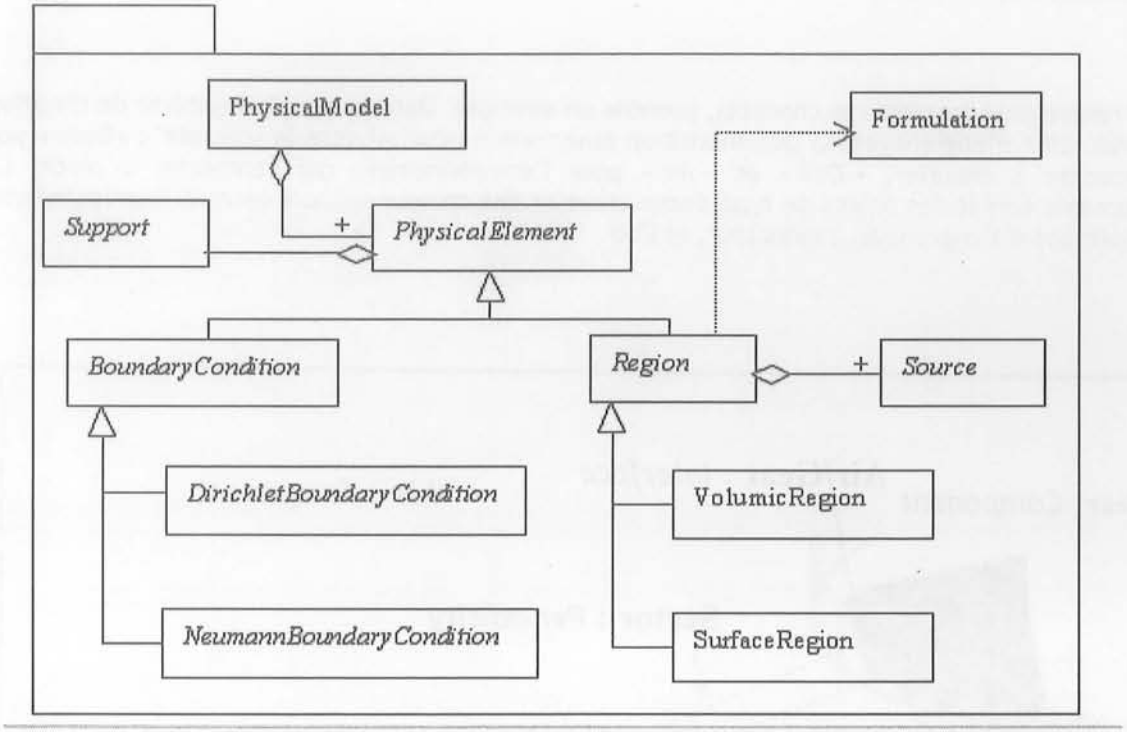


Figure 4-13 : Modèle de données pour la décomposition au regard de la physique de l'objet à étudier.

Toute région possède un support géométrique, qui délimite spatialement son domaine. Une région dispose aussi d'un jeu d'équations issues de la physique appliquée sur son domaine : c'est le rôle de l'attribut formulation. L'intérieur d'une région est homogène et composé d'un seul matériau.

Généralement le solveur contient la description des formulations ainsi que les méthodes numériques adéquates. Dans l'état actuel du module DATA, la description d'une région doit référencer une formulation disponible dans le solveur utilisé. Pour la suite il est envisageable de réutiliser et intégrer les facilités offertes par le logiciel getDP [Dular98], [Dular97], afin de permettre la description des formulations dans le cadre de la plate-forme SALOME.

Les différents types de régions utilisées, pour des modèles spécialisés, peuvent être spécialisés en fonction de la dimension de leur support géométrique (région volumique, région surfacique, ...) ou en fonction du domaine de la physique dans lequel est réalisé l'étude de la région. Ce dernier type est présent dans les modèles spécialisés. Par exemple, dans un modèle thermique, nous pouvons avoir des régions spécifiques, telles que ***ThermicRegion***, qui représente une région volumique contenant un matériau décrit d'un point de vue thermique.

Le type abstrait ***BoundaryCondition*** représente l'abstraction de toutes les classes qui permettent la description des conditions aux limites. Une condition aux limites imposera des variables à l'aide des fonctions numériques spécifiques. Le modèle commun offre aussi des sous classes dédiées à des conditions limites de type Dirichlet ou Neumann. Ces classes seront spécialisées lors de leur utilisation dans différents modèles physiques.

4.3 *Modélisation de la physique en présence du Modèle Commun*

Nous avons maintenant à notre disposition deux éléments fondamentaux pour réaliser la description des modèles physiques : un langage spécialisé dans la description des propriétés physiques, SPML, basé sur un méta modèle, et une collection des types abstraits qui sont a priori valables pour de nombreux domaines de la physique. L'intégrateur peut ainsi créer des modèles physiques en SPML, tout en bénéficiant d'une bibliothèque standard de « classes » abstraites.

En utilisant le Modèle Commun, les nouvelles classes qu'il sera amené à créer pour décrire l'environnement de son solveur, hériteront des classes du Modèle Commun. Ce mécanisme offre la facilité de réutiliser les champs et les méthodes communes.

La stratégie de développement d'un nouveau modèle se déroule alors suivant ces 3 étapes :

1) L'analyse des nécessités du modèle permet l'identification des entités du modèle, nécessaires pour réaliser une description complète du problème.
2) Pour chaque nouvelle entité, trouver son super type parmi les types abstraits du Modèle Commun.
3) Réaliser la description des nouvelles entités dans le langage SPML, en utilisant au maximum les champs et les méthodes du Modèle Commun.

4.3.1 Un exemple de problème multi-physiques

Pour illustrer l'intégralité de cette démarche, nous allons analyser le cas d'utilisation exposé au chapitre 2, à la lumière du langage SPML et du Modèle Commun.

Le problème proposait l'étude du chauffage par induction d'une pièce d'engrenage. Il s'agit d'un problème couplé magnétodynamique, et thermique. Pour la spécification des données physiques de chaque problème, la solution proposée était la création de deux modèles : un modèle magnétodynamique et un modèle thermique.

Pour représenter ces deux modèles, leurs concepts peuvent être dérivés à partir des types abstraits du Modèle Commun (Figure 4-14).

Nous allons reprendre l'exemple du matériau choisi pour le dispositif à modéliser (l'acier non magnétique X10CrAl24). Les paramètres magnétiques et électriques qui présentent un intérêt pour la formulation magnétodynamique du problème sont :

- la dépendance en fonction de la température de la résistivité : $\rho = 1.2(1 + 0.25 \cdot 10^{-3}\theta) \cdot 10^{-6}$ $[\Omega \cdot m]$
- la permittivité relative $\mu_r = 100$

Pour modéliser le matériau, nous avons créé la classe ***MagnetodynamicMaterial***. Les deux paramètres concernant la résistivité (Rhô) et la permittivité relative (Mur) sont représentés grâce aux types numériques du modèle commun.

L'étude thermique du système nécessite l'instanciation d'un modèle thermique, sur la même géométrie. Dans ce modèle, l'acier X10CrAl24 est caractérisé par sa conductivité thermique $\lambda = 14 \quad [W/m{\cdot}°C]$. La classe ***ThermicMaterial*** décrit le même matériau, mais d'un point de vue thermique.

L'héritage polymorphique illustré pour la description des matériaux permet d'aborder et de résoudre un dispositif pour deux physiques différentes.

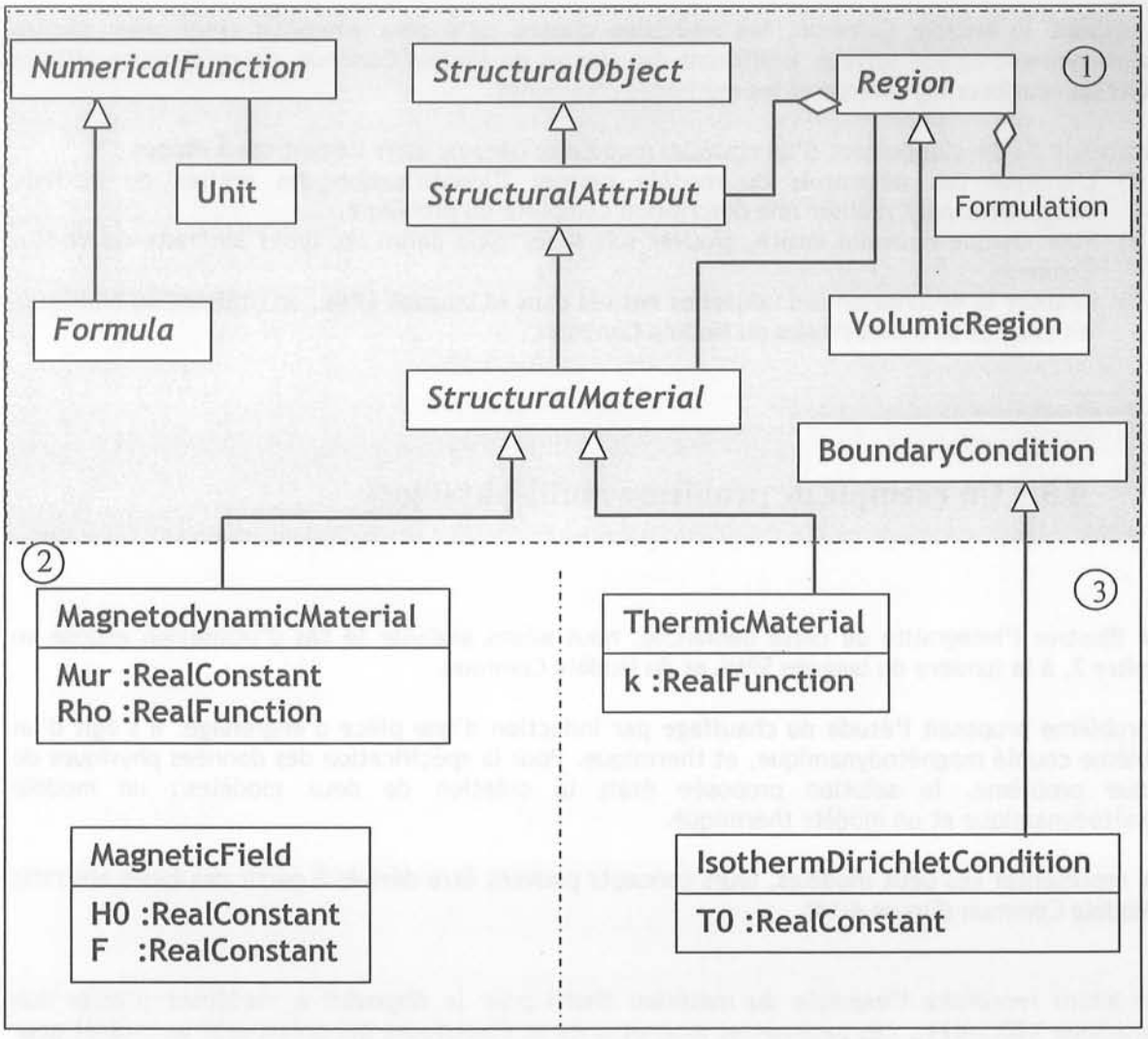


Figure 4-14 Représentation simplifiée du Modèle commun (1). Spécialisation du modèle commun par un modèle magnétodynamique (2) puis par un modèle thermique (3)

Cet exemple illustre la coexistence de plusieurs modèles physiques. L'utilisation du Modèle Commun par les 2 modèles physiques apporte implicitement une propriété intéressante : un type appartenant au modèle commun peut être référencé partout. Par conséquent, au niveau des descriptions physiques, il devient aussi possible de partager des informations entre les 2 physiques.

On constate que deux objets de type ***VolumicRegion*** ayant comme support le même volume géométrique du dispositif (Figure 4-15), bénéficieront d'une description magnétodynamique pour

l'étude du modèle magnétodynamique et d'une description thermique pour l'étude du modèle thermique.

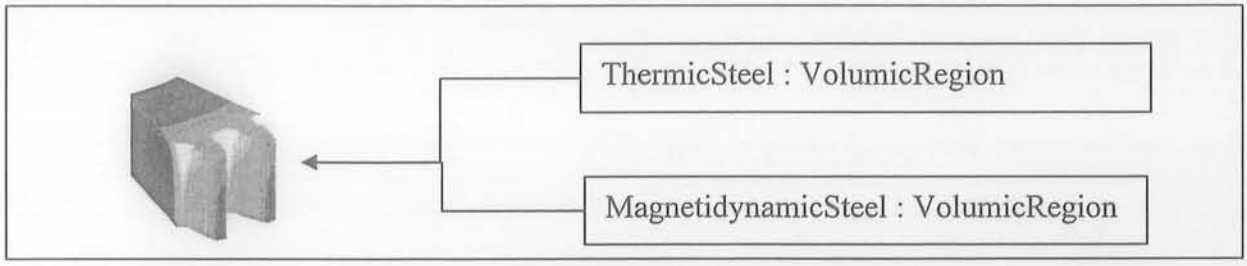


Figure 4-15

CONCLUSION

Nous avons dans le cadre de ce chapitre défini un langage de description de propriétés physiques : le SPML, a la fois au regard de sa sémantique que de sa syntaxe. Nous avons illustré l'utilisation de ce langage sur quelques exemples simples. Nous avons enfin montré la nécessité et l'apport d'un Modèle Commun, tout particulièrement dans le cadre d'analyse multi physiques.

5 Mise en oeuvre

Comme nous avons déjà précisé, le projet SALOME a été démarré pour offrir un environnement logiciel de type « Open Source » qui permet une analyse complète des problèmes physiques, en utilisant des solveurs basés sur la méthode des éléments finis.

Au cours de l'analyse des besoins du chapitre 2, plusieurs étapes nécessaires à la résolution de ces types des problèmes ont été évoquées. Parmi ces étapes, on peut rappeler :

- la description géométrique
- la discrétisation de la géométrie
- l'analyse et la description de la physique
- la résolution des équations de la physique sur le domaine géométrique
- l'exploitation graphique et synthétique des résultats

La réalisation des étapes énumérées exige plusieurs outils adaptés, spécifiques pour chaque fonction :

- Un outil de conception géométrique (CAD) qui réalise la description de la géométrie
- Un mailleur pour réaliser la discrétisation de la géométrie
- Des outils dédiés à la visualisation des résultats et des informations physiques
- Un outil dédié à la description des propriétés physiques

A ces fonctions spécifiques correspondent des modules dédiés.

Pour rendre les développement modulaires, ils sont réalisées sous la forme d'un environnement multi composant et reparti.

Dans la première partie du chapitre nous allons exposer la notion de « Environnement d'applications reparties », qui est à la base de l'architecture de la plate-forme SALOME. Nous allons insister sur les applications reparties basées sur la norme CORBA. Nous présenterons ensuite l'ensemble des modules en insistant sur les interactions avec le module DATA.

Dans le cadre de la plate-forme, notre rôle a été de concevoir et de mettre en œuvre un outil générique de description physique. Nous avons déjà traité dans le chapitre 4, le langage SPML que nous avons spécialement créé pour la description des propriétés physiques. Ce chapitre expliquera l'infrastructure logicielle, qui assure d'un point de vue technique les fonctionnalités du SPML afin de pouvoir réaliser la mise en données pour les différents problèmes physiques.

Une description très rapide des autres modules de la plate-forme permettra au lecteur d'avoir une image d'ensemble de la plate-forme et de mieux comprendre les interactions de notre module DATA avec les composants de la plate-forme.

5.1 *Environnement d'applications reparties*

Pour la réalisation de la plate-forme multi composants, il a été décidé la mise en œuvre d'une architecture d'applications reparties.

La communication au niveau basique (en utilisant des routines de communication, des mémoires partagées, etc.) aurait été une solution coûteuse en terme de développement.
D'autres solutions plus flexibles sont proposées par des environnements dédiés à la communication dans les environnements d'applications reparties. Parmi ces environnements on peut citer CORBA, GLOBE ou DCOM [Tan].

Une application repartie permet d'avoir dans le cadre de la plate-forme un niveau élevé d'indépendance des modules qui composent la plate-forme. Ajouter des nouveaux modules sans changer les autres, était une des conditions requises pour la plate-forme SALOME. Le concept d'applications reparties est représenté par le découpage d'une application en plusieurs modules qui communiquent entre eux, leur exécution pouvant se réaliser sur une ou sur plusieurs machines [Tan]. La communication entre les différents modules repose sur la spécification de services requis et fournis.
Le terme d'application client-serveur exprime d'une manière plus explicite deux rôles véritablement distincts dans le cadre de cette communication:

- le client - celui qui exprime les besoins
- le serveur - celui qui répond aux besoins du client

5.1.1 Environnement d'applications réparties basé sur CORBA

CORBA comporte un modèle orienté objet Client/Serveur d'abstraction et de coopération entre les applications reparties [Geib].

Si nous avons plusieurs applications (modules orientés objet) les modules peuvent exporter leur objets en format d'objets CORBA qui vont communiquer entre eux. Les interactions entre les modules seront matérialisées par des invocations à distance des méthodes des objets. Le rôle du client ou du serveur joué par un objet est définit lors de l'utilisation de l'objet : l'application qui implante un objet constitue le serveur et l'application qui invoque une méthode d'un objet représente le client. Une application peut être à la fois client et serveur.

CORBA est un standard. L'architecture proposée par le standard CORBA fait collaborer des applications sur des machines, des environnements et des langages différents.
Cette architecture est basée sur un bus, appelé ORB (bus de requêtes objet), chargé d'assurer la communication entre les applications (voir la Figure 5-1).

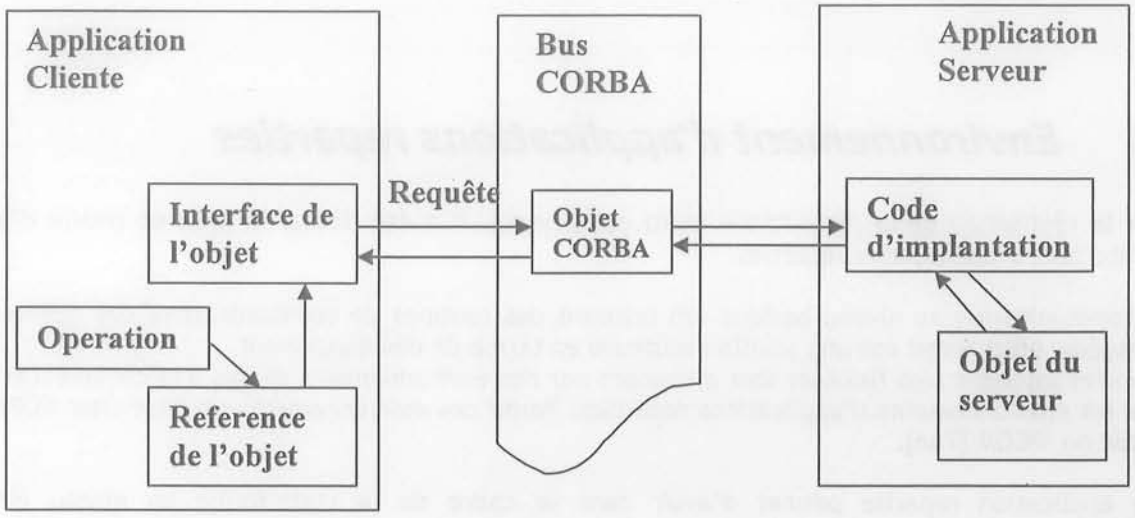


Figure 5-1– Présentation de l'architecture avec des applications réparties dont communication client-serveur est basée sur l'utilisation du bus CORBA

D'une manière statique, les communications client -serveur sur le bus ORB nécessitent la création des interfaces pour réaliser l'émission et la réception des messages entre les clients et les serveurs.

Une première interface, dénommée « souche » (stub) est utilisée par le client pour la transmission via ORB de la demande d'exécution d'une fonction du serveur. Le serveur utilise une seconde interface appelée « squelette » (skeleton) qui récupère la demande sur le bus et la transmet à la fonction demandée de l'objet du Serveur.

Il existe aussi une manière dynamique [Geib], [Schmidt] pour réaliser la communication sur le bus ORB, en utilisant du côté « client » une Interface d'Invocation Dynamique (DII), indépendante de l'objet cible, et en utilisant du côté serveur un skeleton dynamique.

La conception de la plateforme SALOME a prévu des composants bien définis, qui n'évoluent pas pendant leur exécution. En conséquence, les interfaces des composants SALOME sont prédéfinies. Ainsi pour assurer une communication fiable entre les composants statiques du SALOME, la définition statique des interfaces s'est imposé.

5.1.2 IDL

Les interfaces de communication (squelettes et souches) doivent être spécifiées dans un langage dédié à la définition des interfaces, IDL (Interface Definition Language). Simple et proche des langages de programmation comme C++ ou Java, IDL n'entre pas dans la catégorie des langages de programmation. IDL est un langage de description.

Grâce à sa possibilité de définir des structures de données et des fonctions, IDL permet de spécifier la collaboration entre les objets distribués d'une application en séparant l'interface de l'implantation des objets.

Pour relier les objets dans un ensemble d'objets repartis, on doit créer les interfaces de ces objets par type d'objets.
IDL permet de regrouper les fonctionnalités et les données dans des interfaces d'objets. Cela permet de spécifier dans une interface certaines fonctions ou attributs d'une classe afin de les rendre utilisables à l'extérieur des objets. L'implémentation effective d'un objet CORBA peut être faite dans un langage standard de programmation (C++, Python, Java...).

La description IDL peut être projetée dans les interfaces spécifiques à la communication client/serveur. Ainsi les souches seront projetées dans l'environnement de programmation du client et les squelettes seront projetés dans l'environnement de programmation du serveur. Cette projection permet de relier des clients et des serveurs écrits dans des langages de programmation différents.

La projection IDL dans un langage de programmation est réalisée par un pré-compilateur, dépendant du langage cible.
Dans la plate-forme SALOME, nous avons utilisé largement les langages C++ , Java et Python. Pour réaliser la projection des interfaces en C++ et Python, nous avons utilisé le pré-compilateur offert par OmniORB [Grisby], capable de générer les souches et les squelettes dans ces langages. Les interfaces « IDL » sont également compilées en Java, à l'aide du pre-compilateur « idlj » offert par « Sun » et sont complètement compatibles avec les précédents.

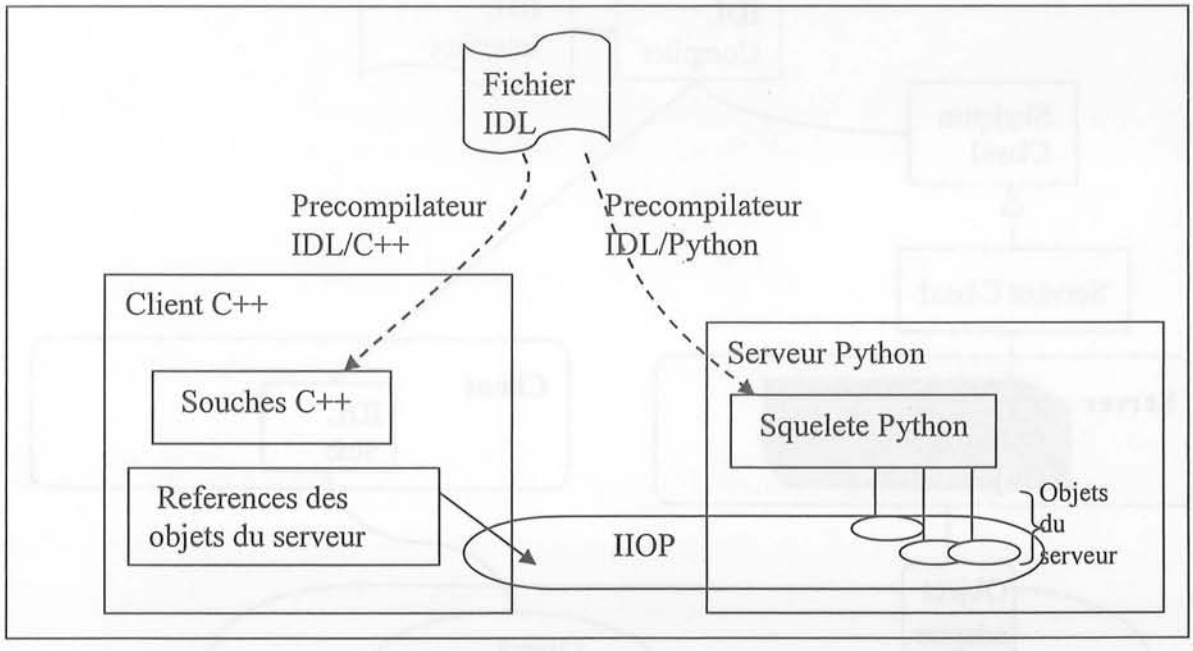


Figure 5-2 La projection de la description IDL d'une communication client/serveur

La Figure 5-2 illustre la projection de la description IDL d'une communication client/serveur. Ainsi nous avons notre Serveur écrit en Python, qui bénéficie d'un squelette Python, mais aussi des bibliothèques de classes de l'ORB en Python. De l'autre côté, le client réalisé en C++ utilise les souches générées en C++, ainsi que des bibliothèques de l'ORB en C++ offertes par la bibliothèque omniORB [Grisby].

Au niveau bas, l'information est échangée dans un format indépendant du langage de programmation, de l'ORB et du système d'exploitation. Ce modèle d'échange est basé sur un protocole de communication nommé IIOP (Internet Inter-ORB Protocol), qui utilisé les couches TCP/IP. Dans l'exemple présenté, le client C++ envoie des données codées en format IIOP vers le

serveur Python. Ce serveur réalise la décodification, ensuite il traite les demandes du client et il renvoie les résultats codés aussi en format IIOP.

Le code généré des squelettes peut être utilisé pour implanter les classes qui définissent les attributs et les fonctions du serveur.

Implémenter effectivement un **objet CORBA** dans un langage de programmation est possible par la création d'une classe qui réutilise les facilitées de la *classe skeleton* en héritant celle-ci. En pratique cette classe est appelée **Servant**. Son rôle est d'implémenter une fonction pour chaque opération de l'interface.

Un client d'un objet repartis possède une **référence d'objet** qui permet l'accès à l'objet. Une **référence d'objet** identifie un objet CORBA, et contient l'information d'adresse. L'adresse permet au client d'invoquer des opérations de l'objet dans le cadre d'un système d'applications reparties orientées objet. La souche (stub) représente pour le client l'objet CORBA au niveau local, dans son langage d'implémentation [Schmidt], [CORBA].

L'objet CORBA de référence de l'ORB est adapté (casting) [Grysbi] dans le format de la souche (stub). En conséquence la souche (stub) a accès au format de l'objet de référence et elle est en contact avec le bus ORB pour réaliser des invocations (Figure 5-3).

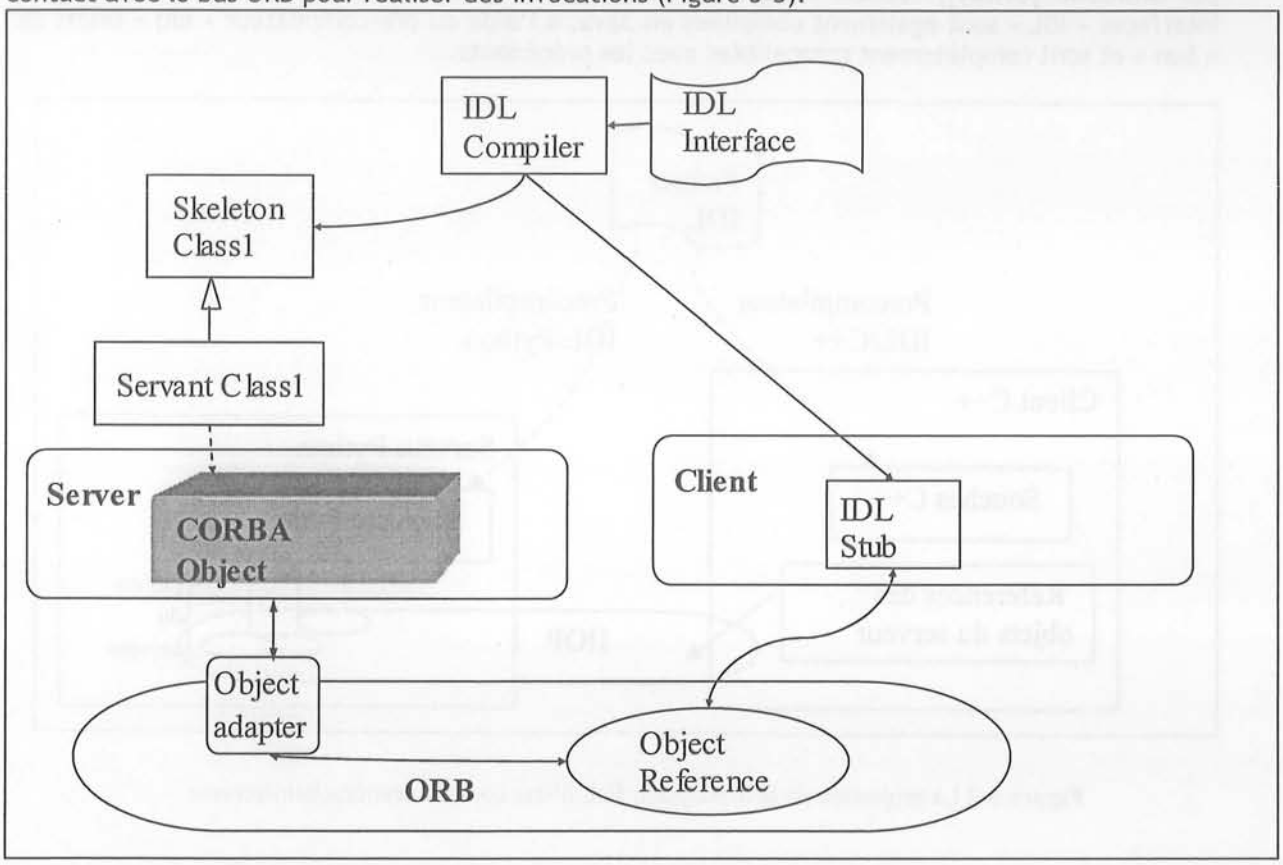


Figure 5-3 - Présentation du processus d'implémentation d'un objet CORBA :

Côté Serveur

1- créer la classe skeleton

2- créer une classe Servant

3- créer un objet CORBA du type de la classe Servant

4- associer un objet adapter pour faciliter la communication

Côté Client

5- générer la souche (stub)

6- obtenir une référence sur l'objet CORBA

7- le client envoie des requêtes sur l'objet CORBA implémenté

5.1.4 Les services CORBA

Le standard CORBA offre également un jeu de services pour faciliter l'intégration et l'interopérabilité des objets distribués. Dans la plate-forme SALOME, un rôle essentiel est joué par le service de nommage qui permet de référencier les objets CORBA par de noms choisis au moment de leur création dans un serveur .
Un autre service, « Object life cycle », définit comment les objets CORBA sont crées, supprimés ou copiés.

5.2 *Architecture générale SALOME*

SALOME se présente sous la forme de plusieurs composants logiciels construits de façon à permettre l'intégration de solveurs ainsi que d'algorithmes de maillage existants et la spécialisation des propriétés physiques pour un domaine donné. La plateforme repose sur une architecture d'applications reparties. Le lien entre les composants de la plate forme est réalisé par le protocole CORBA.

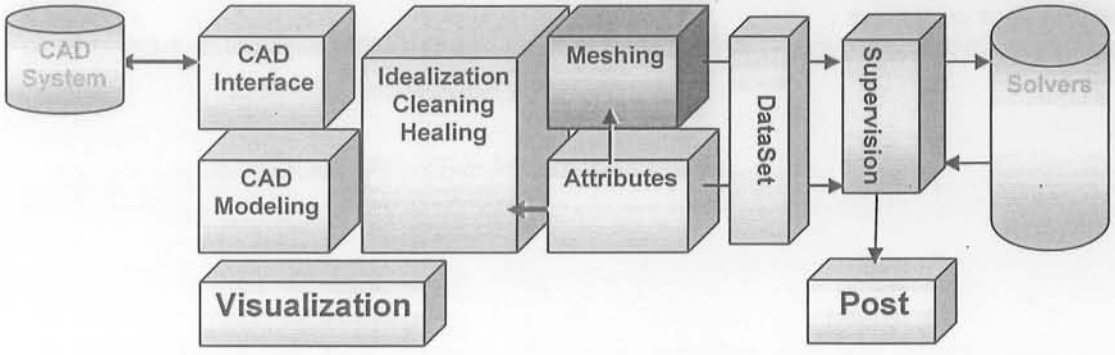


Figure 5-4 Présentation des composants répartis qui jouent des rôles spécifiques : description géométrique, maillage, description des propriétés, solveurs et outils de visualisation

Nous allons présenter dans ce paragraphe les fonctionnalités des modules de la plate-forme.

5.2.1 Présentation des composantes de la plate-forme SALOME

Module Noyau

Dans le cadre de la plate-forme SALOME le module Noyau assure un ensemble de services indispensables au fonctionnement des autres modules (GEOM, DATA,...). Une fois chargé, il peut initialiser le bus CORBA et charger sur ce bus les autres composants.

- Ce module joue le rôle central pour le système d'objets distribués de la plate-forme, en dirigeant les communications client- serveur entre les objets qui représentent les composants de la plate- forme.
- Ce module assure aussi des fonctions non liées à la communication, par exemple la génération d'une interface homme machine (IHM) commune et la persistance.

Chaque module du SALOME contient un moteur qui lui permet de créer des objets distribués propres à leur fonctionnalité. Ce moteur (serveur CORBA) est crée par une « factory » (ou container) qui est également un serveur CORBA. Un moteur est un serveur CORBA lancé soit sur la machine locale, soit sur une machine distante. Le serveur CORBA associé au moteur peut être réalisé par une librairie chargeable dynamiquement dans le processus du container. Si le moteur est constitué à partir d'un code exécutable préexistant, la librairie se résume à un wrapper qui encapsule le code qu'elle lance dans un processus séparé.

Les containeurs sont des objets distribués avec le rôle de créer d'autres objets distribués. Ici le conteneur a son interface IDL, il est implémenté comme tout autre serveur. Les différents clients, tel que l'IHM du SALOME, ou bien un autre module déjà crée (DATA, GEOM, ...,) demandent au Conteneur de créer et d'activer le serveur du module sur le bus CORBA (voir Figure 5-5).

Le Noyau offre les principaux services de communication spécifiques à CORBA. Un rôle essentiel est joué par le service de nommage Salomé. Ce service fournit un système de désignation qui retrouve dynamiquement, à partir de noms symboliques, des références sur les objets Salomé répartis, sans information sur leur localisation. Le service de nommage peut accéder aux services du noyau, aux containers et aux instances des composants SALOME

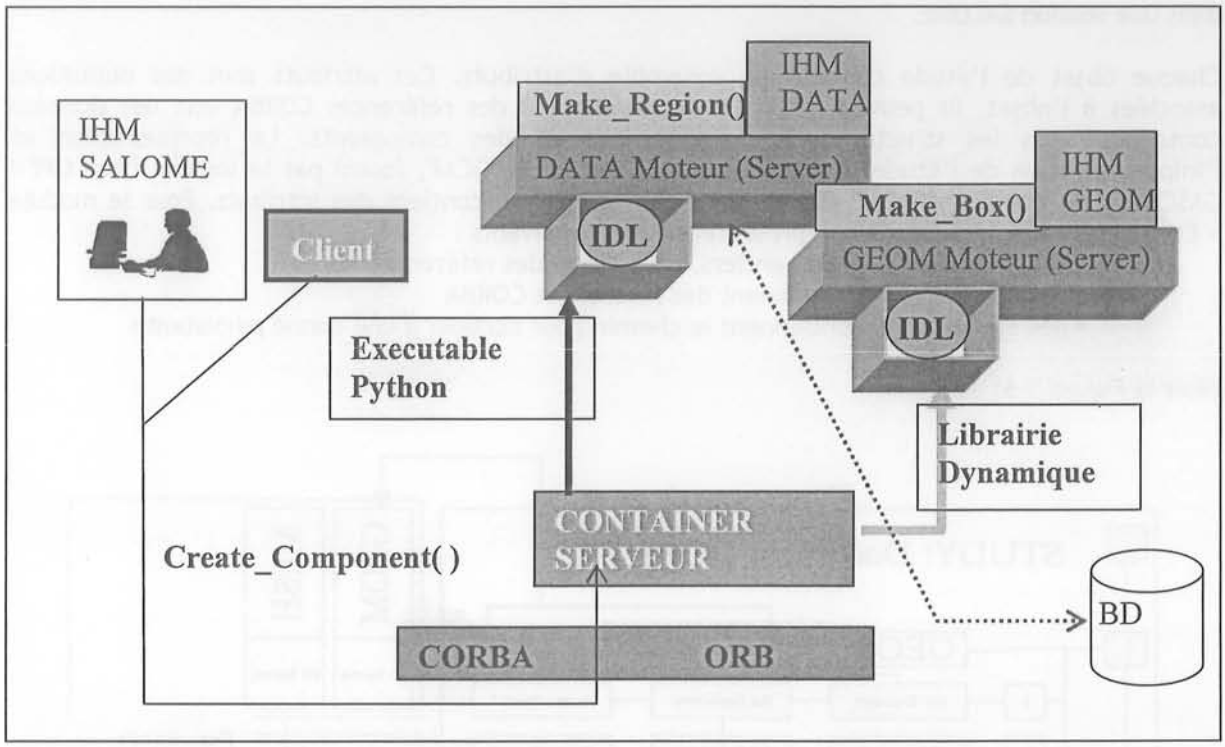


Figure 5-5 : activation des serveurs des modules à partir d'une commande IHM

Interface Applicative

L'utilisateur accède aux fonctions des différents composants de SALOME, soit en mode interactif graphique (GUI), soit en mode textuelle commande (TUI), via un interpréteur Python (Figure 5-7).

L'interface utilisateur textuelle en Python [Swig] est réalisée à l'aide de SWIG. SWIG est un outil du domaine public permettant la génération automatique d'interfaces de codes C ou C++ dans différents langages de script, dont Python

L'interface utilisateur graphique est réalisée avec Qt [Qt1], [Qt2], qui représente un ensemble de classes C++ dédiées à la conception des interfaces graphiques.
L'interface graphique SALOME est dynamique et elle change sa structure en fonction du module chargé. Cette fonctionnalité est basée sur des fichiers XML de description des menus, écrits pour chaque module. De plus, les composants peuvent avoir des interfaces graphiques propres (fenêtres de dialogue, etc.) pour décrire le processus de création des leurs objets spécifiques.

Etude

Grâce aux services du Noyau, chaque composant (module) de la plate-forme sera chargé comme serveur dans le cadre de la plate-forme, lors de son utilisation. A son tour, un composant produira d'autres objets spécifiques, qui seront accessibles sur le bus CORBA.

Pour gérer les objets appartenant à des composants différents, il a été introduit un composant nommé « Etude ». Une gestion commune permet de définir des relations entre les objets produits par les composants de la plate-forme. Le composant Etude permet aussi l'association à chaque objet d'un identificateur unique pour avoir un accès précis à l'objet d'étude, et il permet aussi de donner accès au composant qui a produit l'objet.

L'Etude se présente sous une forme arborescente (arbre de l'étude, Figure 5-7) et il contient l'ensemble des objets de l'étude, qui est une représentation des objets crées par les composants dans une session SALOME.

Chaque objet de l'étude contient un ensemble d'attributs. Ces attributs sont des définitions associées à l'objet, ils peuvent contenir des valeurs ou des références CORBA vers des données contenues dans les structures de données internes des composants. La représentation et l'implémentation de l'Etude sont basées sur le document OCAF, fourni par le logiciel libre OPEN CASCADE. Un document OCAF est un arbre des nœuds qui contient des attributs. Pour le module « Etude », les attributs peuvent représenter les types suivants :

- attributs standard, (les entiers, les strings, des références etc....)
- des attributs qui contiennent des références CORBA
- des attributs qui contiennent le chemin pour accéder à une donné persistante

(Voir la Figure 5-6)

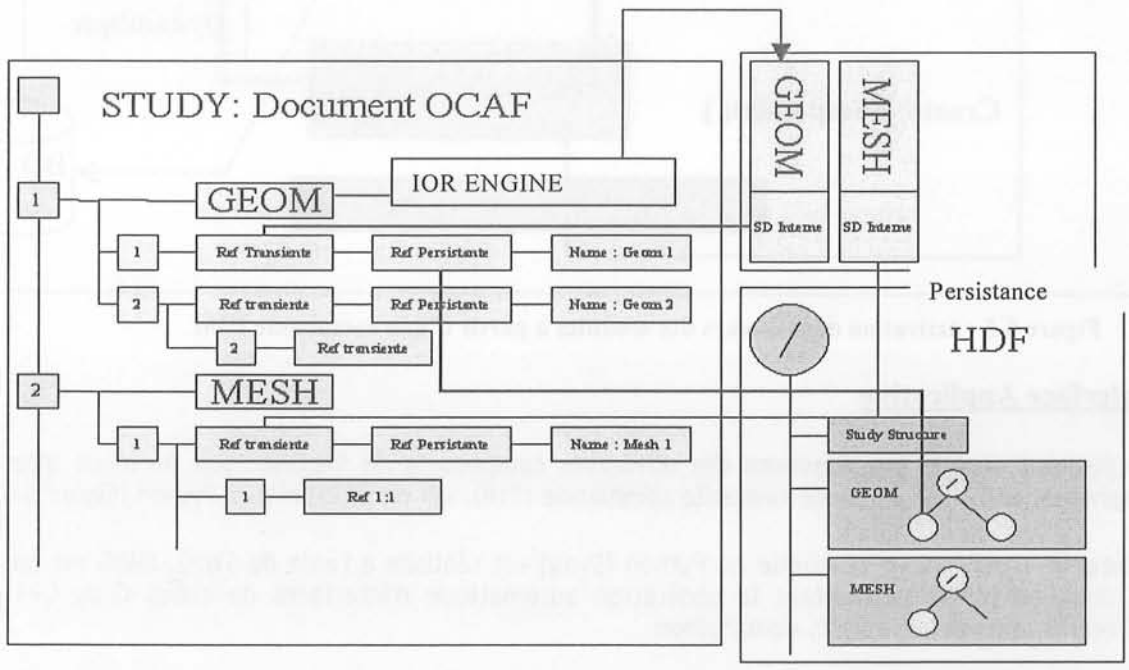


Figure 5-6 : le fonctionnement de l'Etude est basées sur le document OCAF

Chaque nœud de l'arbre d'un document OCAF contient un numéro d'ordre (tag). Le parcours de l'arbre OCAF, de la racine jusqu'un nœud, fournit une séquence de tag qui constitue un

identificateur unique. Cette séquence est appelée « Entry » et contient des tags séparés par « : ». (Par exemple, dans l'arbre de la Figure 5-6, l' « Entry » du Mesh1 est 2 :1)

La mise en œuvre des fonctionnalités nécessaires pour assurer la persistance de l'Etude a été réalisée à l'aide de HDF, un outil qui permet de représenter les données persistantes sous forme d'arborescence [HDF].

Module GEOM

Le module GEOM a pour fonction d'éditer, de créer et de modifier une description géométrique. Cette description doit être exploitable par les modules maillage et DATA. Le module GEOM doit aussi assurer l'importation et l'exportation de la géométrie sous formats standardisées BREP, STEP et IGES.

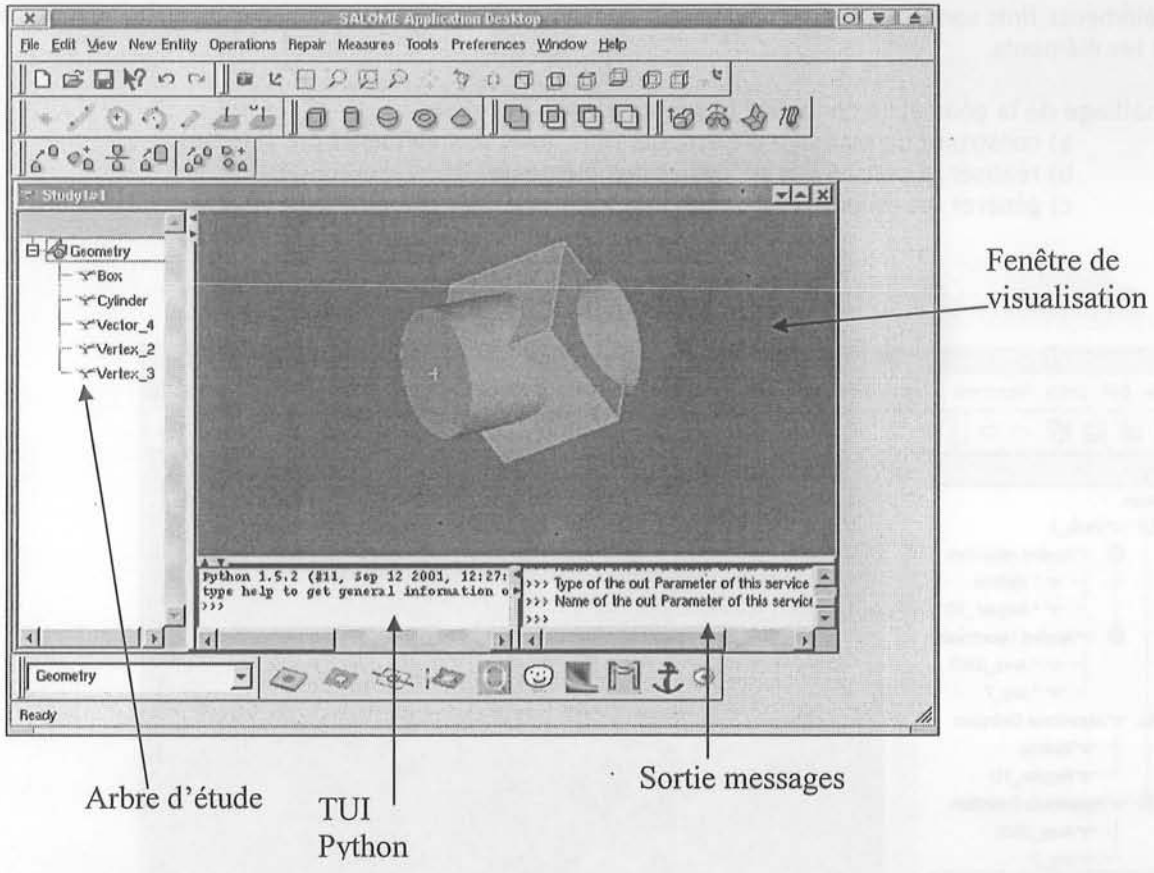


Figure 5-7 l'interface graphique du module GEOM , l'interface applicative et l'étude

Le module utilise largement les facilités du modeleur géométrique OpenCASCADE, qui fonctionne en mode 2D et 3D.

La Figure 5-7 présente l'aspect de l'interface graphique du module GEOM dans le cadre de la plateforme SALOME. Dans le cadre de cette interface nous observons la possibilité de développer la géométrie à l'aide d'une interface graphique. Une fenêtre de dialogue textuel en langage Python permet aussi de créer la géométrie textuellement.

On peut observer aussi la représentation des objets géométriques crées dans un Etude. Cet arbre d'étude permettra aux objets géométriques d'être référencés par un identificateur unique comme

présenté ci-dessus. Par cet identificateur, les autres modules peuvent utiliser les objets de la géométrie, notamment pour y associer des propriétés physiques.

La plate-forme permet aussi la visualisation de la géométrie à l'aide des outils de visualisation basés sur les viewers OpenCASCADE et VTK.

Module Maillage

La plate-forme SALOME a été conçue pour intégrer des **solveurs éléments finis**. Un premier pas dans l'analyse d'un problème avec la **Méthode de l'Eléments Finis**, consiste dans la **discrétisation** de la géométrie du domaine dans une collection d'éléments géométriques prédéfinis (éléments finis) [Reddy]. L'ensemble des éléments finis est nommé aussi **maillage**.

Les éléments finis sont interconnectés, en assurant la continuité des variables sur les frontières qui lient les éléments.

Le maillage de la géométrie comporte plusieurs étapes à réaliser :

a) construire un maillage d'éléments finis, avec des éléments pré sélectés
b) réaliser la gestion des nœuds et des éléments
c) générer les propriétés géométriques (calculer les coordonnées, les intersections, etc.)

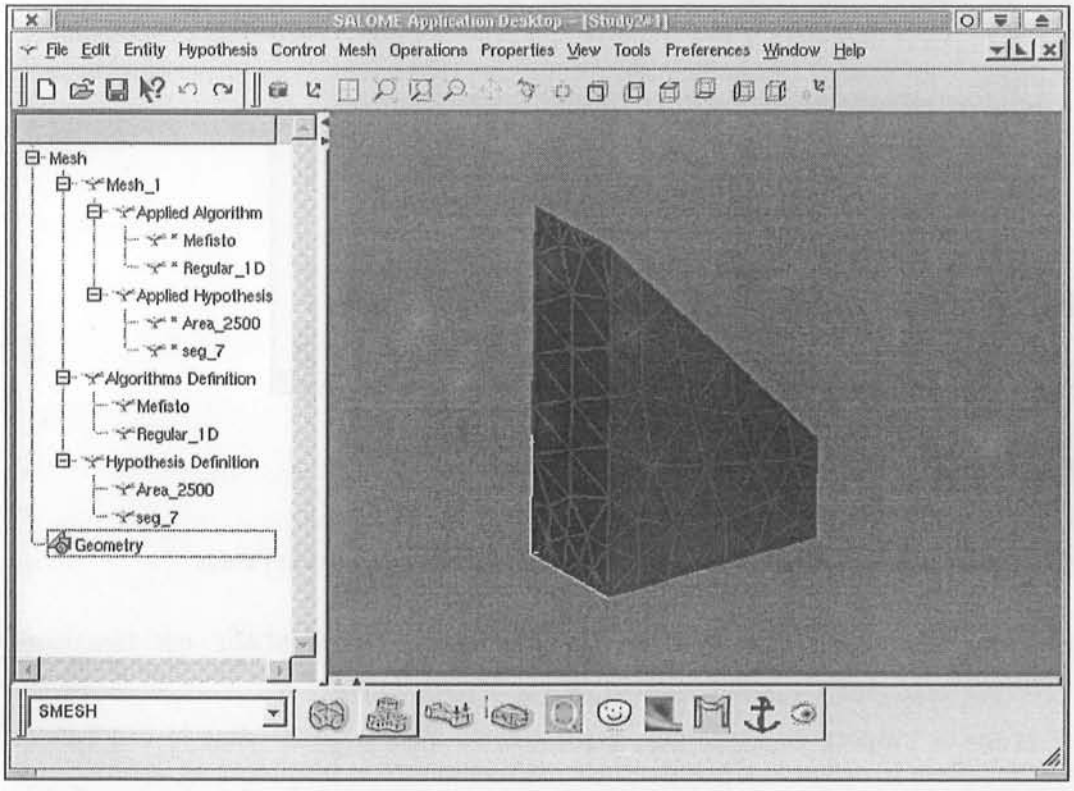


Figure 5-8 - L'aspect du module maillage

Dans le cadre de la plate-forme SALOME, cet objectif est réalisé à l'aide des plusieurs algorithmes de maillage : structuré, non structuré, triangle/tétraèdres, quadrilatères/hexaèdres, « advancing front », Veronoi- Delaunay, etc. Le module permet aussi le pilotage d'autre mailleur.

La relation entre les éléments finis et leurs nœuds peut être représentée conventionnellement dans une matrice booléenne de connectivité.

Un format standard a été adopté pour SALOME, pour la représentation du maillage : MED. Ce format a été proposé par EDF.

Module Superviseur

Le module Supervision de la plate-forme SALOME permet d'exécuter et de contrôler un ensemble des composants numérique répartis. Les schémas qui spécifient l'enchaînement des calculs et les données échangées, peuvent être de deux types :

1. Les schémas de calcul de type data flow. Ces schémas sont représentés par des graphes orientés sans boucle ou opération de contrôle. Un moteur de supervision permet d'exécuter simultanément certains composants et de gérer le cycle de vie des composants. La composition des schémas peut être construite d'une manière graphique. Un éditeur graphique permet la création facile des schémas de calcul de type data flow (voir la Figure 5-9).

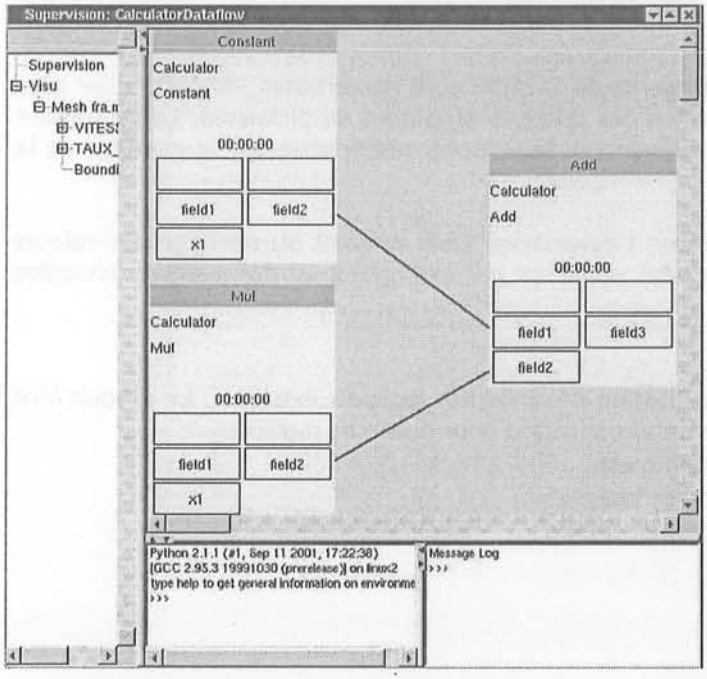


Figure 5-9 Aspect de l'interface graphique du module superviseur

2. Des schémas de calcul contenant de l'algorithmique. Ces schémas permettent de définir des chaînages ou des couplages complexes de composants numériques, à l'aide de boucles et de tests, en s'appuyant sur un langage de script. Il est possible ainsi de définir l'enchaînement de résolutions des différents solveurs pour réaliser des études multi physiques. La description des schémas de calcul contient de l'algorithmique de la syntaxe et des commandes du langage de script.

En plus, le module Supervision est également chargé de leur exécution et de leur suivi. Ainsi le module peut lancer les calculs spécifiés dans le schéma de calcul, mais aussi de réaliser le transfert des données entre différents calculs. Une IHM spécifique permettra à l'utilisateur de visualiser

l'évolution de l'exécution d'un composant et d'intervenir en arrêtant ou en reprenant l'exécution du schéma de calcul ou d'un composant.

Module VISU

Le composant VISU (Visualisation) a été conçu pour produire les outils de représentation graphique et d'exploitation des résultats issus des codes de simulation numérique intégrés à la plate-forme SALOME. Les résultats fournis par le solveur peuvent prendre les formes suivantes :

- **Information globale** issue du calcul : bilan énergétique, résultat condensé sur un point remarquable. Ce type de résultat peut prendre une forme textuelle et il ne nécessite pas des outils spécifiques pour être représenté.
- **Champ** : ensemble de résultats de même type, affectés à des mailles, des nœuds ou des points. Les champs peuvent être des types suivants (ou combiner plusieurs de ces types) :
  - scalaires
  - vecteurs
  - tenseurs
  - matrices
  - chaînes de caractères

Pour un post-traitement efficace, les données produites par le module DATA doivent être accessibles à l'aide des outils du module VISU.

Les interfaces entre VISU et les autres composants de SALOME sont nombreuses, en particulier avec le module DATA pour permettre la visualisation des données physiques du problème. Le composant VISU nécessite aussi un support pour ces représentations, support matérialisé par le maillage de la géométrie.

L'utilisation des fichiers de type .med permet l'association dans le point du maillage des valeurs correspondantes au champ étudié. Ainsi on peut visualiser par exemple avec des nuances associées aux valeurs (voir la Figure 5-10).

La conception en logiciel libre a permis l'utilisation des modules logiciels existants. Le produit VTK [VTK] a été retenu dans le cadre du module de visualisation pour plusieurs motifs :

- possibilité de gérer des graphiques
- traitement et visualisation des images
- il est disponible en version open source

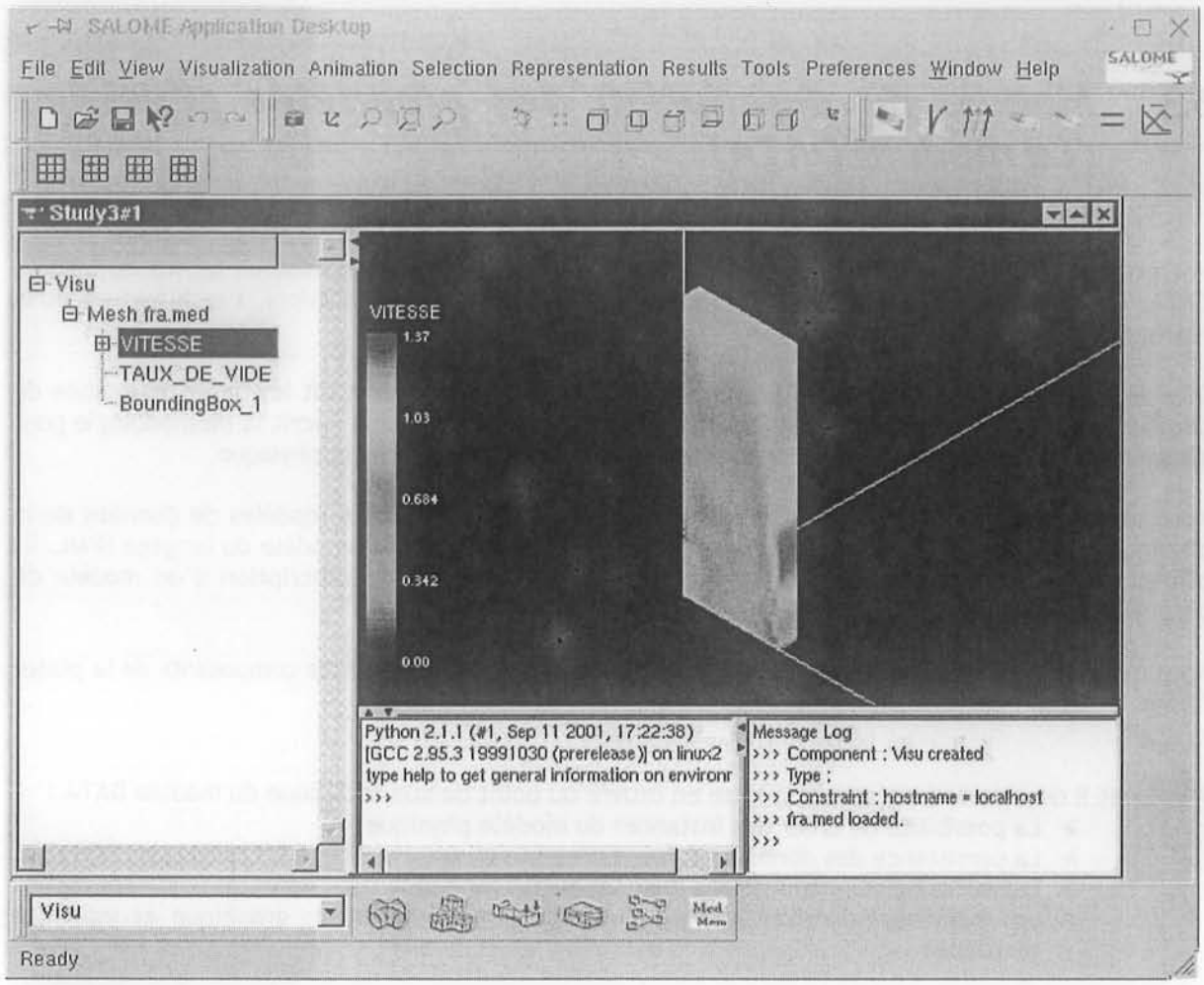


Figure 5-10 – La fenêtre VTK du module VISU

Le fait que les applications basées sur VTK peuvent être décrites en C++, ou en Python, a été un facteur positif pour son intégration dans la plate-forme. Etant utilisé en C++ orienté objet, VTK offre l'ensemble des fonctionnalités de traitement de l'image exigées par la plate-forme.

L'IHM de la plate-forme étant réalisé en Qt. En conséquence, il a été impératif d'afficher l'image VTK dans la fenêtre Qt de l'IHM. Cela a été permis par l'utilisation des classes VTKQGL [VTK].

5.3 *Module de description des propriétés physiques (Le module DATA)*

Notre travail dans le cadre du projet SALOME a été la conception et la mise en œuvre du module DATA. Ce travail avait pour objectif de fournir à la plate-forme les concepts, l'architecture et la réalisation de ce module.

Dans le 4ème chapitre, nous avons présenté comment nous avons construit les bases du module de description des propriétés physiques (le module DATA). En ce sens, on a décrit la méthodologie pour réaliser des nouveaux modèles de données spécifiques à un domaine de la physique.

Nous avons créé un langage spécialement conçu pour la description des modèles de données de la physique, SPML. Les éléments de ce langage se retrouvent dans le métamodèle du langage SPML. Ils offrent des services génériques qui sont exigés par le processus de description d'un modèle de propriétés physiques.

Nous avons vu aussi quels sont les mécanismes de communication entre les composants de la plate-forme.

A présent il nous reste à présenter la mise en œuvre du point de vue technique du module DATA :

- La possibilité de créer des instances du modèle physique
- La persistance des données
- Les outils de communications avec les autres modules
- Les interfaces du module avec l'utilisateur final (interface graphique et interface textuelle)

Une fois décrit en SPML, un modèle physique pourra être utilisé effectivement par un utilisateur final. C'est lui qui créera des données physiques pour réaliser la mise en données d'un problème. Les données physiques seront conformes aux types présentés dans la description SPML.

Les données physiques décrites par l'utilisateur final devront être accessibles pour les autres composants, dans le cadre de l'exécution de la plate-forme. La fonction de sauvegarde des données sous une forme persistante permettra une utilisation ultérieure des données.

Nous allons traiter dans ce paragraphe la mise en œuvre technologique des concepts développés jusqu à présent.

5.3.1 Mise en œuvre par une approche statique

Afin de mieux comprendre le fonctionnement, nous avons ilustré par un fragment de la description SPML du modèle électrostatique, qui a été présenté dans le chapitre 4.1.3. Nous avons considéré dans le Listing 1 seulement les éléments de base de la description d'un matériau diélectrique

```
material1 = Entity(id='DielectricMaterial',
           supertype=Material)

material1_epsilon=SimpleField(id='epsilon',
                    relatedType= Real,
                    stereotype='AGREGATION')
                  )
material1.fields=[dielectricMaterial_epsilon]
```

Listing 1 description SPML d'un matériau diélectrique

En adoptant une approche statique, l'implémentation du module DATA suppose la génération d'une représentation du modèle dans un langage de programmation. Ainsi à partir de la description SPML, un compilateur assure la génération d'un modèle de classes dans le cadre d'un fichier Python.

On peut observer dans le fragment d'un fichier généré pour la description électrostatique Listing 2 qu'une classe DielectricMaterial est générée pour l'élément de type « Entity ».

```
class DielectricMaterial(Material):
        def __init__(self,__name,__epsilon):
            materiau.__init__(self,name=__name)
             self.epsilon=__epsilon      #type : Real
     ......
     def geteps(self):
    ...
     def inscriptObjectInSalomeStudy(self,studyBuilder,dataComponent,study):
    ...
     def writeReexecutableCode(self,alreadySavedObjectDict):
    ...
```

Listing 2 la génération d'une classe Python DielectricMaterial

On peut observer que la classe est munie des méthodes de type « ancestor » et « mutateur » pour ses attributs, mais qu'elle dispose aussi de méthodes spécifiques pour s'intégrer dans la plate-forme

Un compilateur génére aussi les interfaces (IDL) du modèle qui permettent la récupération des données du modèle des classes instantiées et de les rendre accessibles sur un serveur CORBA.

```
interface DielectricMaterial:Material{
    attribute float eps;
    float   geteps();
    short   seteps(in epsilon eps);
    };
```

Listing 3 – Fragment du fichier IDL généré

Le Listing 3 montre la génération de l'interface IDL qui correspond à l'entité DielectricMaterial.
Techniquement le compilateur génére automatiquement le modèle de données et l'interface IDL. Il doit aussi générer le code qui va permettre de réaliser la conexion entre le skeleton CORBA (obtenu par la compilation de l'interface IDL) et l'objet représenté par la classe générée (dans notre exemple la classe DielectricMaterial).

Le Listing 4 montre la manière de la génération automatique d'une classe servant qui hérite d'une souche (classe « skeleton ») généré par le pré-compilateur OmniOrb.

```
class Servant_DielectricMaterial(Model__POA.DielectricMaterial,Material):
    def __init__(self,realObject):
        self.realObject=realObject
 def getepsilon(self):    #type:real
        realepsilon=self.realObject.getepsilon()
```

```
        Srv_epsilon=eps(realepsilon)
        return Srv_epsilon._this()
```

Listing 4

On observe que le code de liaison généré, dispose du référence à un objet existant (realObject) et fournit via CORBA un accès direct à cet objet.

La Figure 5-11 reprend l'ensemble des actions et générations effectuées à partir du fichier de description SPML pour obtenir le code à intégrer dans la plate-forme.

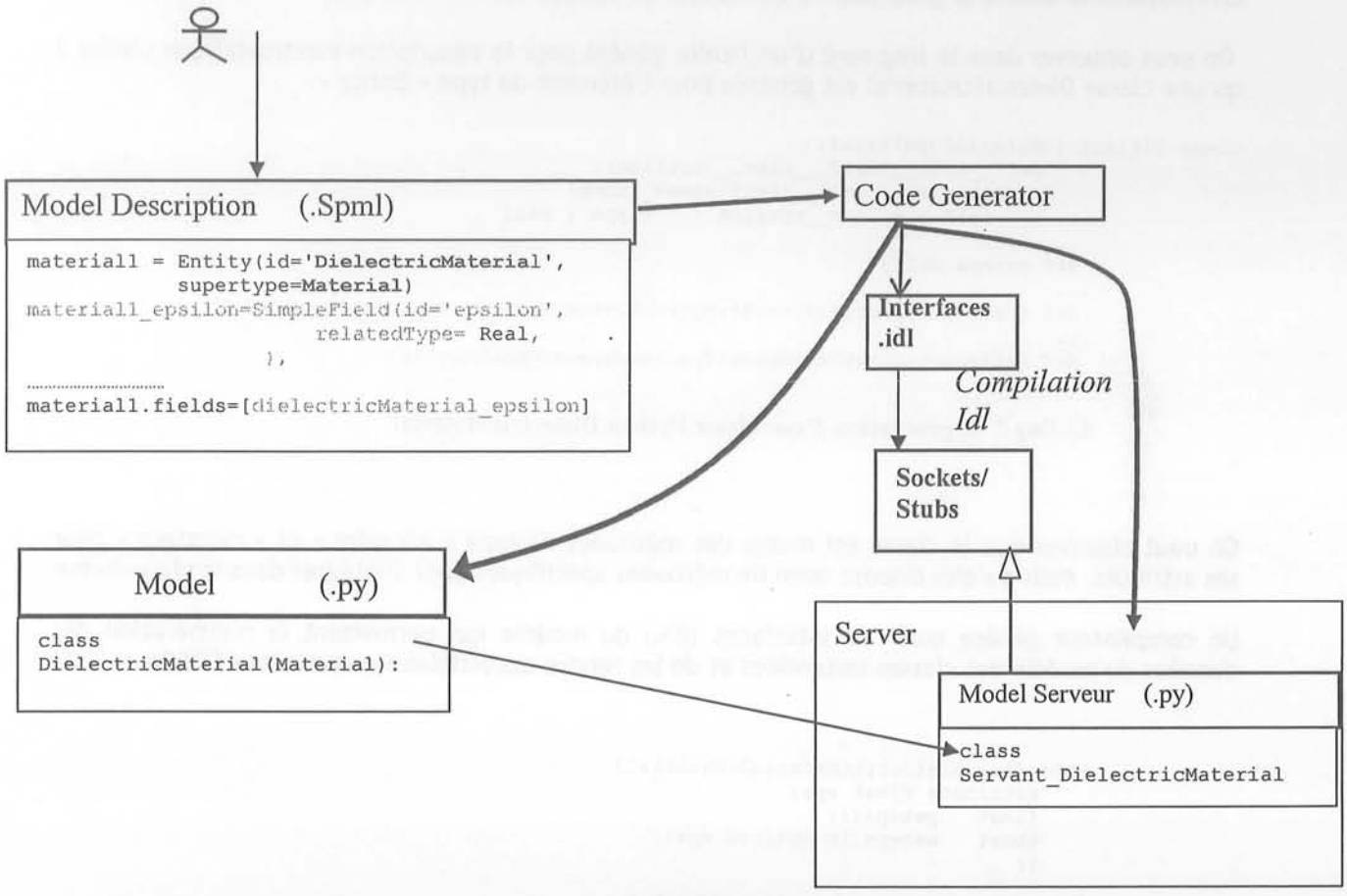


Figure 5-11 le déroulement des actions pour l'approche statique.

Nous avons réalisé un compilateur capable de générer aussi les classes Servant à partir de la description .Spml du modèle. Les classes Servant générées représentent la même structure que les classes du modèle. La différence consiste dans leur rôle :

- les classes du Modèle servent à créer des nouvelles entités du modèle lors de son instanciation,
- les classes Servant peuvent rendre publics sur le bus ORB des objets ayant une structure similaire avec les classes du modèle.

5.3.2 Passage à une solution dynamique

Même si la solution statique était simple et robuste, elle nécessitait la génération de code et la réutilisation du code généré ; cela rendait difficile l'intégration de plusieurs modèles physiques notamment lors de la croissance en complexité des applications de la plate-forme SALOME.

La modélisation statique du modèle était constituée d'un fichier Python qui regroupait toutes les entités du modèle dans des classes qui les caractérisent. La solution dynamique cherche à trouver une utilisation plus efficace du métamodèle. En ce sens, le modèle doit être une instance du métamodèle, comme il a été expliqué d'une manière détaillée dans le paragraphe 3.4.1.

L'idée de la solution dynamique est de charger directement en mémoire la structure du modèle sans passer par du code généré. Il s'agit d'émuler par des objets en mémoire le comportement des classes générées dans la solution précédente.

Cela nous permet de **réaliser notre objectif** : Obtenir un modèle de données au moment de l'interprétation de la description SPML du modèle.

Nous devons réaliser un parseur SPML pour créer une représentation en mémoire du modèle de données.
Comme la description choisie est conforme à la syntaxe Python, nous allons pouvoir utiliser l'interpréteur Python pour parser le fichier SPML. A cela il faut ajouter un programme Python pour interpréter la partie sémantique.
Les types décrits en SPML seront parsés en Python, dans le cadre d'un module.

Les phases de fonctionnement du module, lors de la mise en données d'un problème sont :

- la création d'un espace de noms qui possèdent un dictionnaire de noms
- l'importation du métamodèle dans l'espace de noms du dictionnaire
- l'importation de la description SPML dans l'espace de noms
- **la création des instances du modèle dans le cadre de l'espace de noms**

La dernière action sera réalisée d'une manière textuelle dans le langage défini par le métamodèle, tout en utilisant les types décrits dans la description SPML du modèle.

Python est un langage de programmation orienté objets. **Son modèle de classes offre des notions avancées comme le polymorphisme, la surcharge d'opérateurs et l'héritage multiple.** Son interpréteur permet d'exécuter directement le code écrit en Python, sans avoir besoin de passer par les phases spécifiques de compilation comme dans le cas des langages compilés (C++, Java, etc.).

a). Chargement dynamique du modèle

Nous avons vu que pour la solution statique, le code des classes du modèle est généré dans un fichier Python. L'exécution de ce fichier assure l'existence des classes. La description du modèle en SPML est constituée du script d'instanciation des classes du métamodèle. La création des instances, mais aussi des classes pourra être faite dans un espace de mémoire privé (module).

Pour la solution dynamique, le module DATA peut récupérer dans un fichier le code qui implémente les entités du modèle, comme instances du métamodèle. Dans notre cas, ce fichier est représenté par la chaîne du code contenue dans le fichier de description SPML.

Cette chaîne est ensuitée exécuté directement, en utilisant la fonction du Python « exec() » [Lutz1] . La fonction intégré « exec » du Python permet d'exécuter du code Python quelconque. Cette fonction accepte des arguments optionnels. Si un simple argument dictionnaire est fourni (après le mot « in », obligatoire), il est utilisé comme espace de noms pour le code exec :

```
def loadModelFileinDict(path,file,dict):
    try:

      modelFileStream=open(string.join(loadFileStack,"/")+"/"+file+".spml","r")

      exec modelFileStream in dict
```

Listing 5 : chargement d'une description SPML dans un dictionnaire

Cette action chargera le modèle dans une zone de memoire spécifique (en pratique, un dictionnaire Python, conçu pour contenir toutes les entités du modèle, ainsi que les instances qui y sont associées (Listing 5). Après le chargement du modèle, la création du composant Data dans le module Etude, rendra accessible un objet de référence pour le serveur Data.

Finalement, après la récupération du nom du fichier IDL, le module DATA lancera automatiquement au fur et à mesure de leur création le serveur de données du module DATA qui rendra accessible les données physiques.

En reprenant l'exemple du Listing 1, l'importation du métamodèle ouvre la possibilité de créer des objets de type Entity dans l'espace mémoire privé. Ces objets de type Entity, ont implémenté un comportement similaire à une classe Python qui est réalisé assez facilement grâce à Python. Il suffit de redéfinir la fonction __call__ de la classe Entity (Listing 6).

```
class ClassEntity(ClassObject):
  def __init__(self,id,supertype=....
      ..........................................
  def __call__(self,*args,**kw):
      instance = InstanceEntity(self,args,kw)
      return instance
      ..........................
```

Listing 6 : __call__() permet en Python de redéfinir l'opérateur ' () '

Lors du chargement du modèle électrostatique, l'exécution de la description SPML du type DielectricMaterial, va créer l'objet DielectricMaterial, qui est de type Entity :

```
DielectricMaterial = Entity(id='DielectricMaterial',
                            supertype=Material)
```

Cet objet peut être appelé comme s'il était une classe. Par l'appel de la commande :

```
Mat1= DielectricMaterial (epsilon=6.15)
```

Ceci est équivalent d'un point de vue syntaxique, avec l'appel d'une fonction constructeur pour une classe appelée DielectricMaterial. Cet appel aura comme effet la création d'un objet de type « InstanceEntity » du métamodèle par l'appel de son constructeur dans la fonction __call__() de la classe Entity.

L'objet Mat1, issue de la classe InstanceEntity possède un attribut « klass » qui contient le nom de l'instance de type « Entity », DielectricMaterial qui a été crée précédemment et permet de garder une référence vers la classe à laquelle il appartient.
La classe InstanceEntity permet l'enregistrement des attributs avec leurs valeurs dans un dictionnaire. Cet enregistrement est précédé par la vérification de la conformité du type des valeurs avec les arguments acceptés par l'entité spécifié en SPML.

Dans le cas présenté, DielectricMatérial accepte un seul attribut de type réel, « epsilon » qui aura comme valeur 6.15.

Nous avons réalisé ainsi l'objet Mat1, tout en utilisant la syntaxe d'un constructeur d'une classe nommée « DielectricMaterial ». Les attributs de l'objet correspondent à la structure définie par DielectricMaterial. Plusieurs fonctions de la classe InstanceEntity offrent à l'instance « Mat1 » des facilités spécifiques de la POO, tel que l'introspection et des méthodes de type « ancestor » et « mutator ».

```
#class that will manage the instance of entity object
class InstanceEntity:
    __uid = 0
    __inited = 0
    def __init__(self,klass,args,kw):
        #set the uid of the instance entity
        self.__uid=InstanceEntity.__uid
        InstanceEntity.__uid = InstanceEntity.__uid+1
        #set the class
        self.klass = klass
        self.__realdict__ = {}
        #initialized the dict for SalomeStudyObject for fields
        self.__dictOfFieldSalomeStudyObject={}
        #set the field not garbageable
        self.__garbageable = 0
        #set initiation is finished
        self.__inited = 1
        if args != ():
            raise AttributeError, "keywords are required for the moment"
        for key in kw.keys():
            self.__setattr__(key,kw[key])
-----------------------------
    def getKlass(self):
        return self.klass

    def __setattr__(self,name,value):
        #print "setattr", name
        if not self.__inited:
            self.__dict__[name] = value
    def generatePythonCode(self,withAutomaticGenerateName):
        #create the name and add the class name
        if withAutomaticGenerateName:
-----------------------------
    def inscriptObjectInSalomeStudy(self,studyBuilder,dataComponent,study):
        #call the class method that is able to realize the study inscription
        self.klass.inscriptObjectInSalomeStudy(self,studyBuilder,dataComponent,study)
        #return the SObject created in the salome study
        return self.getSalomeStudyObject()
```

ancestors, introspection

Persistance, et inscription dans l'étude

Listing 7 la classe InstanceEntity

D'autres fonctions offrent des méthodes spécifiques pour s'intégrer dans la plate-forme tel que la persistance, et l'inscription dans l'étude (Listing 7). Ce mécanisme assure lors du chargement du fichier SPML, la création des objets de type Entity, qui ont la propriété de émuler les entités du fichier SPML, dans des objets à un comportement similaire aux classes Python évoquées en 5.3.1

En conclusion :

Nous avons vu le principe du chargement d'un nouveau modèle dans un espace de noms directement en mémoire. En suite nous avons présenté en détail les étapes du chargement d'une description SPML, et les mécanismes qui assurent les fonctionnalités spécifiques pour le modèle de données :

1) Récupérer une référence vers l'arbre d'étude.

2) Création d'un nouveau dictionnaire Python qui contiendra toutes les entités du modèle et ses instances.
3) Création d'une application qui contiendra le modèle.
4) Récupérer une référence vers le fichier SPML utilisé, à partir de son nom.
5) Le chargement dans l'espace de noms du dictionnaire du module contenu dans le cadre du fichier SPML, qui décrit le modèle.
6) La création dans l'arbre de l'étude d'un nouveau composant qui sera l'origine d'un sous arbre pour acquérir graphiquement les objets créés lors de la mise en données physique.
7) Démarrer un serveur qui fournira :
   - Une référence vers le fichier SPML.
   - Une référence vers l'arbre d'étude.
   - Une référence CORBA, vers l'application exécutable qui contient le modèle chargé.

   Ce serveur est destiné à être utilisé par l'interface graphique du module DATA. Ainsi l'interface graphique aura accès aux spécifications concernant GUI, présentées dans le cadre du fichier SPML qui décrit le modèle physique.
8) Démarrer un serveur des données physiques qui peut fournir les objets créés comme instance du modèle physique, lors de la mise en données par l'utilisateur final. Ce dernier serveur joue un rôle clé dans la communication avec les autres modules, notamment pour réaliser la récupération des données destinées au solveur.

La création des instances du modèle physique (Utilisateur final)

Pour la création d'une instance du modèle physique nous avons conçu dans le cadre du module DATA la fonction « runCommandData() ».
L'exécution de la commande est réalisée dans le même espace de memoire qui a chargé les entités du modèle :

```
def runCommandData(self, command, studyName,mode="creationMode"):
        dataObjectDict=self.dictOfDataObjectDict[studyName]

                expression = 'lastObject='+command

                exec command in dataObjectDict
```

Listing 8

De cette manière, le dictionnaire contient les classes du métamodèle, les instances de ces classes, représentées par les entités du modèle, et les instances du modèle qui sont les données physiques elles-mêmes du problème. On retrouve donc tous les niveaux du méta regroupés dans le même endroit du mémoire (Figure 5-12).

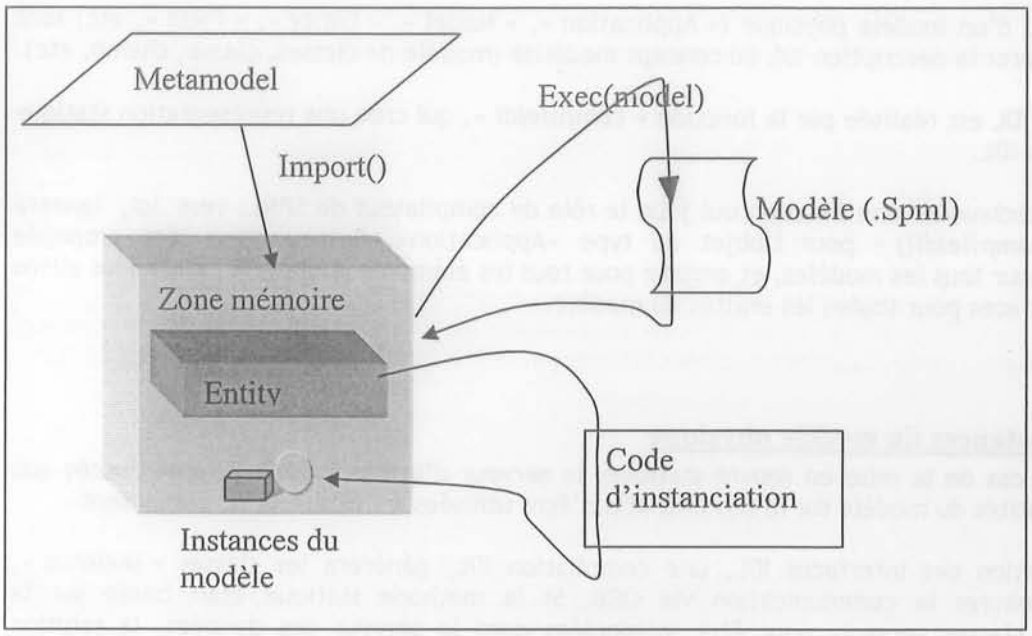


Figure 5-12 la création d'une instance du modèle physique

c). La persistance

Une fois le modèle instancié, les instances des entités du modèle peuvent être isolées et récupérées, grâce à la possibilité d'introspection offerte par la fonction Python « isInstance() ». Des méthodes spécifiques offertes par le métamodèle offrent la possibilité de sauvegarder les instances du modèle sous une forme de « script ». Le Listing 9 montre l'utilisation dans le cadre du fonction writePythonScript() du module DATA des fonctions generatePythonCode() qui sont associées aux entités du metamodèle :

```
def writePythonScript(dataObjectDict,stream):
    objectList=[]
    for key in dataObjectDict.keys():
        if isinstance(dataObjectDict[key],InstanceEntity):
            if objectList.count(dataObjectDict[key])==0:
                objectList.append(dataObjectDict[key])

    #write code for each object
    for object in objectList:
        string=object.generatePythonCode(1)
```

Listing 9

Dans ce cas, le rechargement des instances sauvegardées peut être réalisé simplement par l'exécution du script sauvegardé.

Génération des interfaces IDL

Grâce à sa complexité, le langage SPML inclut toutes les informations qui peuvent caractériser l'interaction des types du modèle. Les éléments du métamodèle qu'on peut retrouver dans la

description SPML d'un modèle physique (« Application », « Model », « Entity », « Field », etc) sont capables de générer la description IDL du concept modélisé (modèle de classes, classe, champ, etc).

La génération d'IDL est réalisée par la fonction « ***compileIdl*** », qui crée une représentation statique de la description IDL.

D'une manière technique, une fonction qui joue le rôle de compilateur de SPML, vers .Idl, lancera la fonction « ***compileIdl()*** » pour l'objet de type «Application». Cette action sera propagée récursivement pour tous les modèles, et ensuite pour tous les éléments du modèle. Ainsi nous allons obtenir les interfaces pour toutes les entités du modèle.

Serveur des instances du modèle physique

Comme dans le cas de la mise en œuvre statique, le serveur d'instances doit assurer l'accès aux instances des entités du modèle sur le bus ORB afin d'être utilisées à l'extérieur du composant.

Apres la génération des interfaces IDL, une compilation IDL, générera les classes « skeleton », responsable d'assurer la communication via ORB. Si la méthode statique était basée sur la génération des classes servant, pour être instanciées dans le serveur des données, la solution dynamique génère le code des classes servant dans une chaîne de caractères qui ensuite est exécuté dans un dictionnaire du serveur des données (wrapperObjectDict). Ce code va créer toutes les classes servant qui correspondent aux classes du modèle (voir Listing 10).

```
def __init__(self,applicationFullName,classEntityNameList,dataObjectDict):
    print "enter in the DataIdlSrv __init__"
    self.dataObjectDict=dataObjectDict
    self.wrapperObjectDict={}
    #try to read the file containing the compiled python idl file
    applicationName=applicationFullName+"__POA"
    try:
        self.wrapperObjectDict[applicationName]=__import__(applicationName)
        #exec "import "+applicationName in self.wrapperObjectDict
    except:
        raise "Error while importing the module : "+applicationName

    for name in classEntityNameList:
        #create a new class to assure inherence
        tab="    "
        str="class "+name+"("+applicationName+"."+name+",InstanceEmulator):\n"
        str=str+tab+"def __init__(self):\n"
        str=str+tab+tab+"InstanceEmulator.__init__(self,'"+name+"')\n"

        exec str in self.wrapperObjectDict
```

(a) (b) (c)

Listing 10: (a) création du code pour importer les classes skeleton ; (b) création du code pour les classes servant ; (c) exécution du code pour chaque classe servant.

Emulation des commandes d'une interface textuelle

Il est normal pour un utilisateur final qu'il veuille créer des nouvelles instances du modèle physique (matériaux, régions, etc.), écrire des scripts de commande de forme :

```
Unit(nom= 'Kelvin', abréviation='K')
```

Ce fait se traduira par l'envoi d'une commande vers le serveur de données :

```
runCommandData('unit1=Unit(nom='Kelvin', abréviation='K')')
```

Nous avons vu que l'exécution d'une commande runCommandData a comme résultat la création d'une instance du modèle. La définition de l'instance est donnée comme paramètre pour runCommandData.

Une classe émulateur offre la possibilité de créer des instances portant des noms des classes existantes dans le modèle. La surcharge de l'opérateur '()' à l'aide de la fonction «__call__() », du Python, rend possible l'appel de la fonction __call__() par une séquence de code contenant le nom de l'objet suivi par '()'. Par exemple si Unit est un objet de la classe Emulator, le code Unit(arg1, arg2) appelle la fonction __call__() avec les arguments considérés. La structure de la classe Emulator est donnée dans le Listing 11 :

```
class EmulatorClassEntity:
    def __init__(self,id):
        self.__id=id
    def __call__(self,*args,**kw):
        #create a new emulator instance entity
        newInstanceEntity=EmulatorInstanceEntity(self,args,kw)
        #creation of the string to send to the data server
        commandString=repr(newInstanceEntity)+"="+self.__id+"("
        #alternative is to send as name the string give in the script

        for key in kw.keys():
                commandString=commandString+","+key+"="+repr(kw[key])
        commandString=commandString+")"
        result = dataComponent.runCommandData(commandString,myStudyName)
    def getId(self):
        return self.__id
```

Listing 11

Pour émuler toutes les classes du modèle, dans un espace d'une application séparée, la solution a été de créer tous les objets possibles de type Emulator, portant le nom des classes existantes dans le modèle.

De cette manière, en utilisant un TUI séparé de l'espace de mémoire du modèle, avec une syntaxe identique à un appel d'entité du modèle, il est possible par la surcharge de la fonction __call__() d'appeler la création des instances réelles du modèle.

Pour mieux comprendre le fonctionnement de TUI, nous allons presenter un fragment de l'instanciation textuelle du modèle électrostatique dont la description en SPML a été présentée en détail dans le paragraphe 4.1.3.b :

```
>>> ur = Unit (name= 'relative unit', abreviation='ru')
>>> f1 = Formulation (Id= 'ES3SCA')
>>> mat_dielectric_PE = DielectricMaterial (name='PE', epsilon=RealConstantScalar(value=8,51, unit = ur))
>>>BCond1 = ElectricDielectricCondition (name='BCond1', support = [1 :2 :1 :1, 1 :2 :2], value=0)
>>>DielectricRegion=ElectrostaticVolumicRegion (name = PE_Region, Material= mat_dielectric_PE,
                    support= '1 :2 :1 :0')
-----------------------------------
>>>dataSet = PhysicalDataSet ( ListOfRegions= [DielectricRegion, AirRagion],
            ListOfBoundaryConditions=[BCond1, BCond2]
```

GUI

Lors de l'analyse d'un problème de la physique, par un utilisateur final, le rôle de l'interface graphique (GUI) est de faciliter la mise en données du problème, d'une manière graphique. Pour caractériser les propriétés physiques d'un problème, l'utilisateur doit créer des instances du modèle physique dont le problème est dépendant (description des matériaux, des conditions aux limites, etc.). La diversité des types de données qui composent des modèles différents, mais aussi la manière graphique différente de la création des instances, appartenant à des types différents, impose l'auto adaptation du GUI.

Cette auto adaptation représente le principal attribut du GUI du module DATA. Elle demande une adaptation aux différents modèles (par exemple prendre en compte seulement les types qui peuvent intervenir dans un problème électrostatique).

Une autre adaptation est de générer des fenêtres de dialogue spécifiques à chaque type du modèle. Cela impose plusieurs parties pour GUI :

1) un arbre de recherche, qui aide l'utilisateur à trouver le type qu'il veut créer ;
2) un générateur de fenêtres de dialogue, qui sont adaptables automatiquement en fonction du type à créer ;
3) un générateur de TUI, à partir d'une fenêtre de dialogue.

1. Arbre de recherche

Pour créer des nouvelles instances du modèle physique, l'utilisateur peut avoir des difficultés pour retrouver le bon type. Par exemple, on considère la mise en données de la physique pour un problème d'électrostatique. On peut utiliser le modèle électrostatique, qui a été présenté au 4.1.3 pour décrire une condition limite de type Dirichlet, qui impose un potentiel constant sur une frontière du domaine.

Le modèle décrit nous offre le type EquipotentialDirichletCondition. L'utilisation du TUI suppose une bonne connaissance des types offerts par le modèle physique et de leur syntaxe. Pour l'interface graphique, nous avons introduit une facilité pour retrouver d'une manière intuitive et logique, les types contenus dans le modèle physique, chargé par le module DATA au moment de l'exécution de la plate-forme. Cette facilité est offerte par un menu contextuel dynamique, à une structure arborescente, qui s'adapte d'une manière automatique au modèle physique utilisé.

L'utilisateur part d'une catégorie générique de type « Physical » ; ensuite, il peut choisir une « Boundary Conditions » (condition limite), parmi les types physiques de base. L'arbre de recherche avance à chaque pas avec un nouveau menu contextuel (voir Figure 5-13). Finalement entre les conditions Dirichlet, l'utilisateur peut choisir, parmi les conditions de type Dirichlet du modèle spécialisé, le type EquipotentialDirichletCondition. La sélection graphique de ce type concret, aura comme effet l'ouverture d'une fenêtre de dialogue adaptée pour créer des éléments de type EquipotentialDirichletCondition. Cet arbre de recherche est basé sur des éléments de type « Hierarchy » du SPML. Nous avons vu dans le paragraphe 4.1.1 (a) qu'une hiérarchie est destinée à structurer logiquement les entités du modèle physique. Une hiérarchie peut contenir des entités du modèle et d'autres hiérarchies. L'appartenance à une hiérarchie spécifie les entités qu'on veut créer à l'aide d'une fenêtre de dialogue GUI. Dans l'exemple analysé nous avons avancé dans l'arbre de 3 hiérarchies : « Physical », « Boundary Conditions », et nous avons choisi finalement l'entité EquipotentialDirichletCondition.

En esprit avec le caractère orienté objet du SPML, la hiérarchie d'un élément de type « Entity » sera héritée par ses sous classes jusqu'à sa redéfinition. Au moment du chargement du modèle, chaque hiérarchie sera actualisée avec les entités qui lui appartiennent.

La solution technique que nous avons développée comporte plusieurs étapes :

1) Construire la structure hiérarchique du modèle, dans un document en format XML. Cela a été réalisé par des fonctions getXmlString( ) qui sont présentes dans le type « Hierarchy » du SPML. Cette fonction offre les informations sur la composition d'une hiérarchie. En suite une fonction du métamodèle getXMLTreeString() générera une fichier XML qui réalise la description des hiérarchies du modèle

2) Obtenir dynamiquement à partir du stream XML généré, un menu contextuel qui joue le rôle d'un arbre de recherche (Figure 5-13) :

```
<DEFANGED_application-hierarchy>
<submenu label-id="Structural">
</submenu>
<submenu label-id="Numerical">
</submenu>
<submenu label-id="Physical">
<submenu label-id="BoundaryConditions">
<popup-item label-id="DirichletVectorBoundaryCondition"/>
<popup-item label-id="FlotantScalarBoundaryCondition"/>
<popup-item label-id="EquipotentialDirichletCondition"/>
<popup-item label-id="DirichletScalarBoundaryCondition"/>
</submenu>
<popup-item label-id="Formulation"/>
<popup-item label-id="Reg_vol"/>
....
</submenu>
<submenu label-id="DataSet">
<popup-item label-id="PhysicalDataSet"/>
</submenu>
</application-hierarchy>
```

Listing 12

On a réalisé ce module, en utilisant un parseur XML de type Sax [Harold]. Finalement, les menus contextuels ont été implémenté en utilisant les facilités de la bibliothèque graphique Qt [Qt, ORiely].

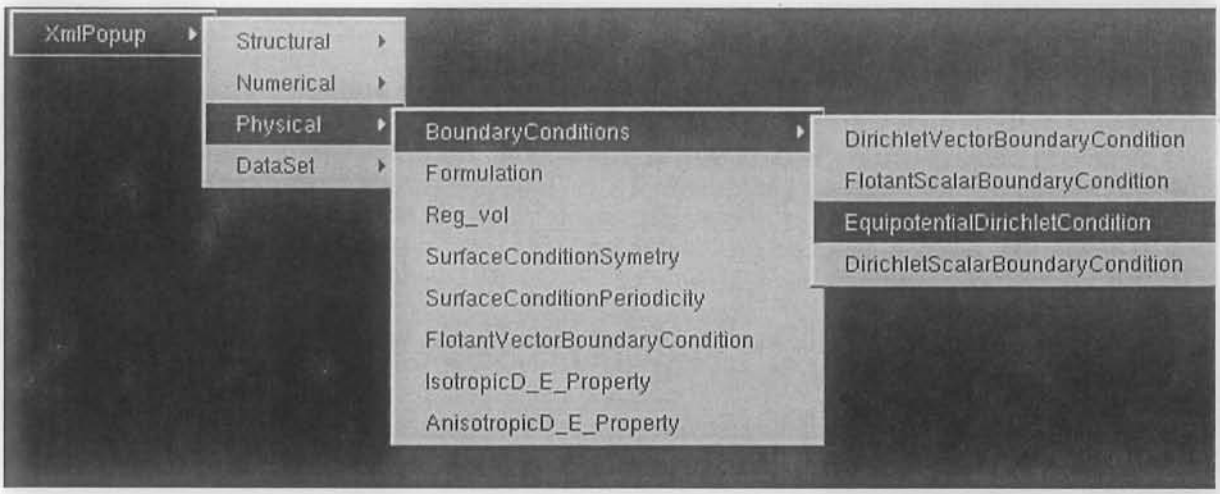


Figure 5-13 : arbre de recherche des types disponibles dans le modèle courant

3) Envoyer une requête pour le serveur qui offre la génération des fenêtres de dialogue. Le client constitué par une extension de la classe QMenuPopup, du « Qt », va demander la génération d'une fenêtre de dialogue qui correspond au type sélectionné.

Générateur de fenêtres de dialogue

Ce module a été crée pour assurer la description graphique des instances du modèle physique. L'adaptation de la fenêtre de dialogue est réalisée en concordance avec la description SPML du type qu'on veut créer. Par exemple, pour créer un matériau de type DielectricMaterial, dont sa description SPML a été donnée au 4.1.3. (listing1), l'interface graphique génère la fenêtre de dialogue présentée ci-dessous (Figure 5-14)

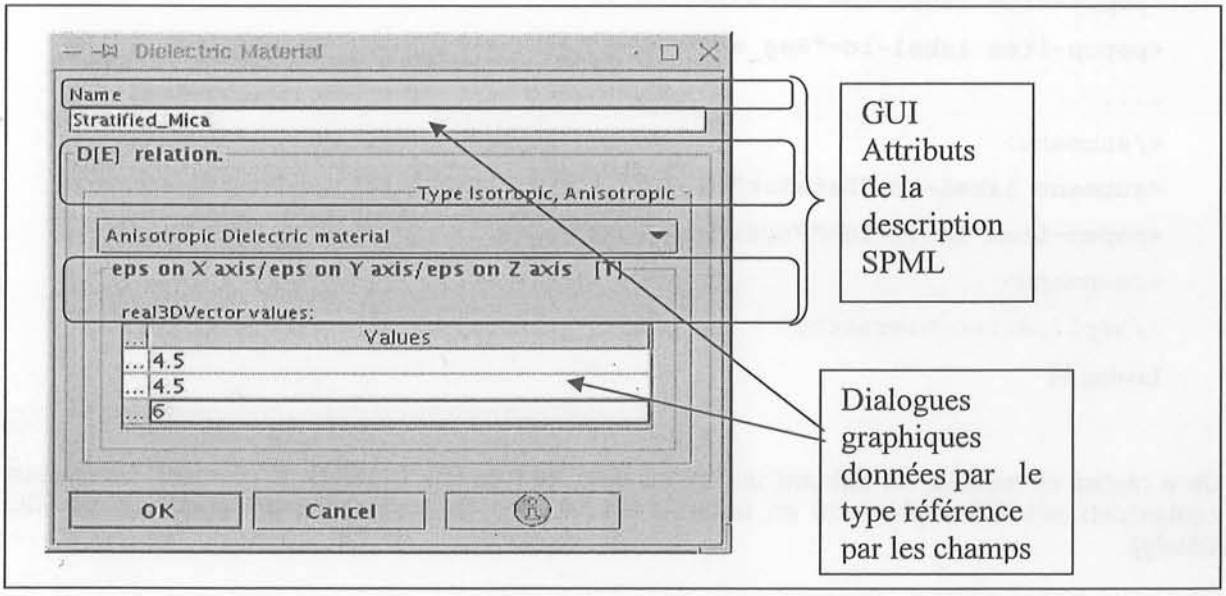


Figure 5-14 La structure est donnée par les champs de l'entité et par le type référence par ces champs

On peut observer par exemple pour un type réel qu'un éditeur de ligne est une solution. Par contre, pour instancier un vecteur de réels il nécessite une liste de lignes. Les informations graphiques du SPML (uiInformation) jouent un rôle supplémentaire, concernant les champs de documentation de la fenêtre, la position des champs ou la redondance de la fenêtre.

Etant un langage orienté objet, SPML supporte le polymorphisme. En conséquence, le générateur des fenêtres de dialogue a été adapté à cette fonctionnalité. Pour cela l'utilisation d'un champ abstrait apparaît dans la fenêtre de dialogue par la possibilité de choisir entre les sous-types concrets du type abstrait représentant le champ. La Figure 5-15 montre ce mécanisme.

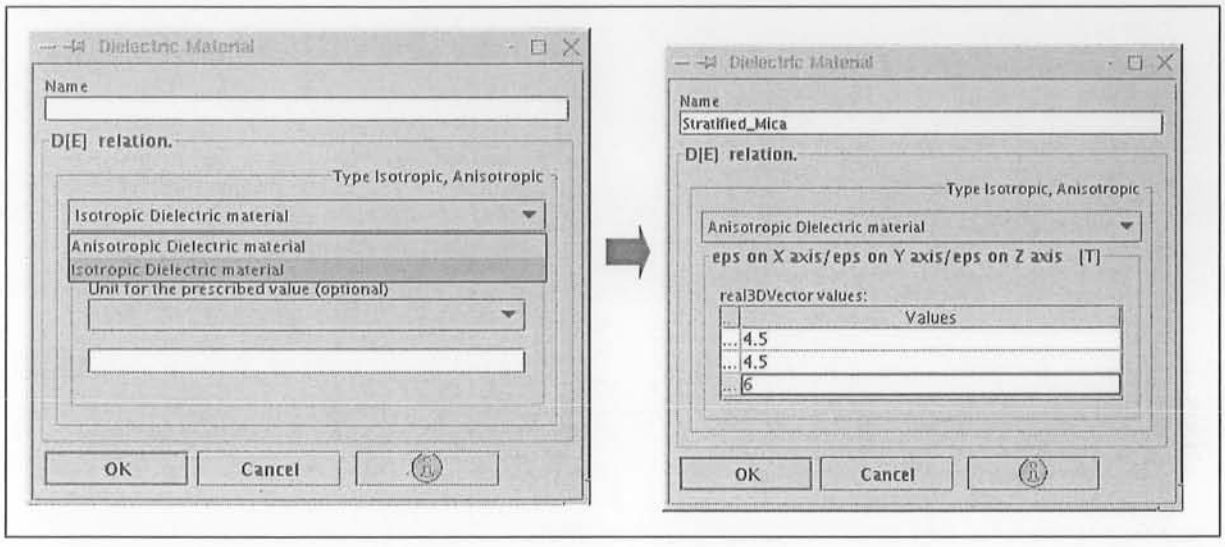


Figure 5-15 Polymorphisme du GUI, exemplifié pour deux matériaux diélectriques

On peut observer que le champ abstrait D_E_Property du matériau DielectricMaterial, pourra être instancié avec un des sous types du D_E_Property : Anisothropic_D_E_ Property ou Isothropic_D_E_Property.

Pour réaliser l'adaptation des fenêtres de dialogue, le module GUI du DATA a besoin d'une référence vers le fichier SPML qui décrit le modèle utilisé. Cette référence sera donnée par un serveur démarré au moment du chargement de la description du modèle dans le module DATA.
La majorité des données physiques contiennent d'autres instances de modèle physique qui ont été créés précédemment. Par exemple, une « Region » peut contenir des matériaux de type « DielectricMaterial ». Pour répondre à ce besoin, le serveur DATA, offre deux fonctions avec lesquelles le module GUI peut interrogé DATA :

- une fonction donne la liste des instances déjà crées
  getProjectInstances(in string study),
- une fonction qui donne le type d'une instance
  (getInstanceType(..)

Un générateur de TUI, à partir d'une fenêtre de dialogue

Apres l'instanciation de la fenêtre de dialogue, pour transformer les données saisies dans les champs de la fenêtre, les données seront regroupées dans une commande de type TUI, qui sera envoyée aux serveur de données du module DATA.

6 Exemple Complet : Environnement de calcul SALOME-FLUX pour la magnétostatique

Exemple Complet : Environnement de calcul SALOME-FLUX pour la magnétostatique

Nous avons décrit précédemment la réalisation et le contexte d'un module de description des propriétés physiques dans la plate-forme SALOME.

Avant de créer un modèle dédié pour un domaine physique particulier, nous allons aborder d'une manière générale la problématique qui intervient pour réaliser la description dans le langage SPML d'un modèle pour caractériser un domaine de la physique.

Pour valider le module, nous avons choisi un problème magnétostatique afin d'être testé sous la plate-forme SALOME. Ce test inclut deux actions distinctes :

- préparer la plate-forme, en intégrant le solveur FLUX
- utiliser effectivement la plate-forme pour : réaliser la mise en données du problème magnétostatique, lancer la résolution, et exploiter les résultats.

La préparation de la plate-forme commence avec la création d'un modèle magnétostatique pour adapter le module aux particularités du domaine magnétostatique.
Ensuite le chapitre propose de présenter les étapes de réalisation d'une interface entre le module DATA et le solveur Flux, qui a été choisi pour cette validation. Le couplage du solveur a bénéficié du serveur des données CORBA, qui offre le modèle de données accessibles sur le bus ORB de la plate-forme.

Finalement nous présentons l'utilisation effective de la plate-forme et particulièrement la mise en données du problème magnétostatique réalisé du point de vue de l'utilisateur final.

6.1 *Représenter un modèle physique*

Les systèmes physiques, leurs phénomènes, ainsi que les interactions et les transformations qui y sont associées, peuvent être décrites par des propriétés physiques. Ces propriétés peuvent analyser une série des données expérimentales sur le domaine étudié. Pour certaines propriétés on peut associer des entités mathématiques (vecteurs, scalaires, etc.), pour les caractériser quantitativement. On appelle grandeur physique, une propriété physique qui est caractérisée quantitativement.

La définition des entités mathématiques fait l'objet de la théorie des structures algébriques, qui est basée sur des relations et des propriétés mathématiques. Nous avons représenté et implémenté dans le cadre du modèle commun les principales entités mathématiques (vecteurs, scalaires, tenseurs,etc.). Ainsi nous avons déjà les entités mathématiques pour les associer aux propriétés physiques.

En fonction de l'importance qui leur est accordée par la théorie appliquée sur un domaine de la physique, les grandeurs physiques se classifient en grandeurs physiques primitives et grandeurs physiques dérivées.

Les lois et les théorèmes s'expriment par des équations entre les représentations numériques de diverses espèces de grandeurs physiques qui définissent le système. Généralement on peut représenter ces équations par :

$$m = \Im(a, b, \ldots n) \tag{6.1}$$

Ou m est la valeur numérique d'une grandeur physique de type M et, d'une même manière, a, b,...n sont des valeurs des grandeurs physiques de types A, B,...N. $\Im$ est un opérateur qui appliqué aux a, b,...n, détermine la valeur numérique m.

Dans le cadre des lois et des théorèmes peuvent intervenir des constantes de proportionnalité, qui peuvent être des constantes universelles ou des constantes de matériel.

Le modèle physique représente une synthèse du domaine physique pour regrouper les éléments suivants :

- les grandeurs physiques primitives
- les grandeurs physiques dérivées, issues de l'interaction avec les primitives (dans les équations des lois et des théorèmes)
- les constantes qui interviennent dans le cadre des lois et des théorèmes
- les équations du système qui sont représentées par les lois et les théorèmes valables pour le domaine de la physique considéré
- les diverses formulations dérivées à partir des équations du système (lois + théorèmes), qui permettent une solution numérique
- les conditions imposées sur les limites du domaine, qui associée aux systèmes des équations, assurent l'unicité de la solution pour la grandeur physique étudiée pour tout point du domaine.

Ces entités du modèle physique représentent les types des données d'entrée d'un solveur élément finit. Nous avons vu que dans la plate-forme SALOME, tous les types du modèle physique sont décrits d'une manière « orienté objet » dans le cadre d'un modèle de « classes ».

Pour adapter la plate-forme SALOME pour l'analyse d'un domaine de la physique, il faut créer ce modèle de classes et implémenter ce modèle de classes dans le langage SPML, dédié pour la description des propriétés physiques dans la plate-forme SALOME. Pour faciliter le travail de développement, on peut utiliser la notation UML pour représenter les diagrammes de classes du modèle, lors de la description du modèle, avant de passer à la phase d'implémentation.

En conclusion, pour réaliser la description d'un nouvel domaine de la physique, dans le but d'adapter la plate-forme SALOME à l'analyse du ce domaine, on doit débuter par un étude théorique du domaine. Cette étude permettra l'identification de toutes les entités du modèle physique (grandeurs physiques, constantes, conditions limite, etc.).

Une fois le modèle physique spécifié, on peut réaliser sa description dans le langage SPML. Cette description est nécessaire pour être utilisée dans le cadre de la plate-forme SALOME et elle sera réalisée par quelques étapes :

- il faut représenter chaque entité dans un format équivalent à une classe, classe capable de spécifier son type. Simplifier la classe à créer, en utilisant les classes du modèle commun, déjà disponibles.
- L'implémentation des nouvelles classes en SPML, tenant compte seulement de besoins du modèle.
- la finition du modèle, pour ajuster l'adaptation automatique de l'interface graphique.

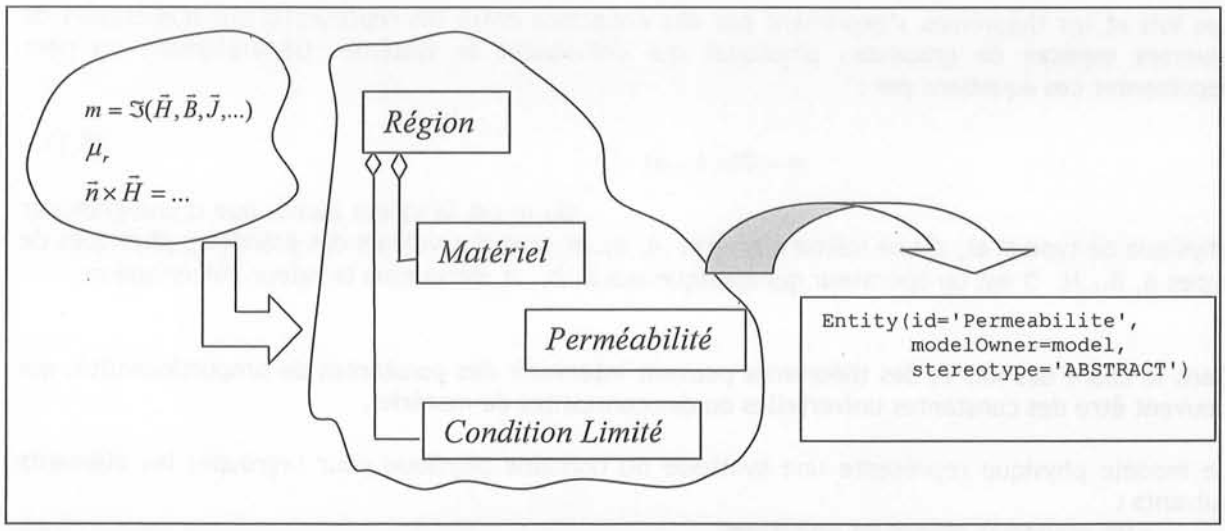


Figure 6-1 La description SPML à partir de l'analyse d'un domaine de la physique

6.2 *La description du modèle magnétostatique*

Pour réaliser la description du modèle magnétostatique, nous allons appliquer les étapes de description d'un modèle physique, présentées à la fin du paragraphe précédant. Nous allons commencer par l'étude théorique de la magnétostatique, qui nous donnera une spécification du modèle magnétostatique. Ensuite nous allons suivre les étapes présentées pour arriver à une description du modèle en format SPML.

6.2.1 Etude théorique de la magnétostatique

La magnétostatique constitue un cas particulier d'étude du champ électromagnétique. Dans ce sens, si le champ électromagnétique était caractérisé par quatre vecteurs :

- l'intensité du champ électrique $\vec{E}(P,t)$
- l'induction électrique $\vec{D}(P,t)$
- l'intensité du champ magnétique $\vec{H}(P,t)$
- l'induction du champ magnétique $\vec{B}(P,t)$,

la magnétostatique nécessite seulement la considération des grandeurs physiques qui concernent le champ magnétique stationnaire $\vec{H}(P,t)$ et $\vec{B}(P,t)$ (pour chaque point du domaine considéré P, à chaque instant t.)

Les relations fondamentales du champ magnétique stationnaire résultent par la particularisation des lois générales et les lois de matériau, qui caractérisent le champ électromagnétique, pour les conditions suivantes :

-les objets du système sont immobiles : $v = 0$

-les grandeurs physiques ne sont pas variables en temps : $\partial D/\partial t = 0$, $\partial B/\partial t = 0$, $\partial \rho_v/\partial t = 0$

Ces relations sont issues des lois générales de Maxwell

$$\nabla \times \vec{H} = \vec{J} \quad \text{(loi du circuit magnétique)} \qquad (6.1)$$

$$\nabla \cdot \vec{B} = 0 \quad \text{(loi du flux magnétique)} \qquad (6.2)$$

et l'adaptation de la loi de dépendance entre l'induction, l'intensité magnétique et l'induction rémanente dans la présence d'un champ magnétique

$$\vec{B} = \mu(H)\vec{H} + \vec{B}_r \qquad (6.3)$$

Nous considérons aussi les formes intégrales de ces équations.

Si nous considérons un domaine limité par la surface Σ, contenant à l'intérieur des éléments qui conviennent aux conditions imposées par la magnétostatique:

-des conducteurs immobiles parcourus par des courants continus de densité $\vec{J}$, donnée ;

-des aimants permanents d'induction rémanente $\vec{B}_r$ donnée
-des régions composées des matériaux magnétiques saturables ou non saturables

$\vec{H}$ et $\vec{B}$ à l'intérieur du domaine v_Σ sont uniquement déterminés par des conditions limites qui sont imposées soit par la composante tangentielle de H, soit par la composante normale de B sur la frontière du domaine v_Σ.

a). Spécifications du modèle

Les résultats de cette analyse montrent que les entités à représenter dans le modèle physique sont : l'intensité du champ magnétique $\vec{H}(P,t)$, l'induction du champ magnétique $\vec{B}(P,t)$, les conditions limites de type Dirichlet et Neumann, et les constantes locales $\vec{B}_r$ et μ qui dépendent du matériau qui compose la région. La forme intégrale introduit en plus la tension magnétique sur une courbe fermée U_m, l'intensité du courant dans un conducteur I et le flux magnétique sur une surface Ψ_m.

Nous analyserons ensuite les possibilités de représentation pour chaque entité du modèle physique :

- les grandeurs physiques primaires, $\vec{B}(P,t)$ et $\vec{H}(P,t)$, ont une valeur de type ***vecteur*** qui peut être représentée par une valeur $\Re^3$ dans un ***système de coordonnées***. Les deux grandeurs physiques ont des ***supports géométriques ponctuels***.
- les conditions limites de type Dirichlet et Neumann : Etant considérées à valeur nulle pour le modèle magnétostatique, ces conditions Dirichlet et Neumann seront spécifiées seulement par leur support géométrique, support représenté par des surfaces appartenant à la frontière du domaine.
- la constante locale μ : en fonction du medium (environnement, matériau) , la perméabilité pourra se voir associé un scalaire réel et positif dans le cas d'un matériau linéaire et isotrope, ou bien une valeur de type vecteur réel tridimensionnel dans le cas d'un matériau linéaire et anisotrope. Le cas des matériaux non linéaires imposera des fonctions numériques réelles ou vectorielles.
- la constante locale $\vec{B}_r$ concerne le matériau des aimants qui sont des éléments à une structure anisotrope. En conséquence sa valeur associée est un vecteur réel ($\Re^3$).

b). Simplifications et considérations techniques

En fonction des besoins d'étude nous pouvons considérer des hypothèses de simplification. Par exemple nous pouvons considérer le comportement du μ seulement pour des matériaux linéaires.
Les constantes locales peuvent être considérées comme des propriétés appartenant à des matériaux qui constituent les régions homogènes (Voir modèle commun).

Pour les constantes locales μ et $\vec{B}_r$ on peut classifier les matériaux comme des matériaux utilisés pour les aimants, caractérisées par μ et $\vec{B}_r$, et des matériaux non magnétiques caractérises seulement par μ.

Supplémentairement, chaque domaine de la physique peut avoir des dispositifs prédéfinis, qui ne sont pas issus de leur modèle physique. Il s'agit des entités structurelles qui ne correspondent pas à une description physique ou géométrique. Dans le cas de la magnétostatique nous avons les inducteurs dont description contient une valeur entière qui spécifie le nombre de spires et une représentation géométrique optionnelle.

6.2.2 Représentation du modèle des classes (OO)

A partir des entités du modèle exposées précédemment, nous avons envisagé les classes nécessaires pour le modèle. Le Tableau 1 expose les classes à créer :

Classe	Entité représentée	Champs	Types du champ
Induction	$\vec{B}(P,t)$; [T]	Valeur	RealConstantVector3D
		Support géométrique	Point
InténsitéChamp	$\vec{H}(P,t)$	Valeur	RealConstantVector3D
DensitéCourant	$\vec{J}$; [A/m]	Valeur	RealConstantVector3D
FluxMag	Ψ_m ; [Wb]	Valeur	RealConstant
		Support géométrique	Surface
TensionMag	U_m	Valeur	Real
		Support géométrique	Courbe
ConditionDirichlet	Condition Dirichlet	Support géométrique	Surface
ConditionNeumann	Condition Neumann	Support géométrique	Surface
MatériauMagnétostatique		Propriété_BH	Propriété_BH
Propriété_BH	Abstract ; μ et $\vec{B}_r$		-
BH_Lin_Iso	μ	Mur (μ)	RealConstant
BH_Lin_Aniso	μ	Mur (μ)	RealConstantVector3D
BH_AimantCartIso	μ et $\vec{B}_r$	Mur (μ)	RealConstantVector3D
		Br (μ)	RealConstantVector3D
Inductor (bobine)	(Structural)	Nbr Spires	Entier

Tableau 1

Le tableau d'analyse expose l'emploi des types du *Modèle Commun* dans la description des nouvelles entités. Certaines entités du modèle utilisent les types définis dans le *Modèle Commun*. Par exemple RealConstant contient déjà une unité associée à une valeur réelle.

L'autre manière d'utilisation du *Modèle Commun* est l'héritage. La Figure 6-2 présente la représentation d'une partie des classes crées selon le Tableau 1.

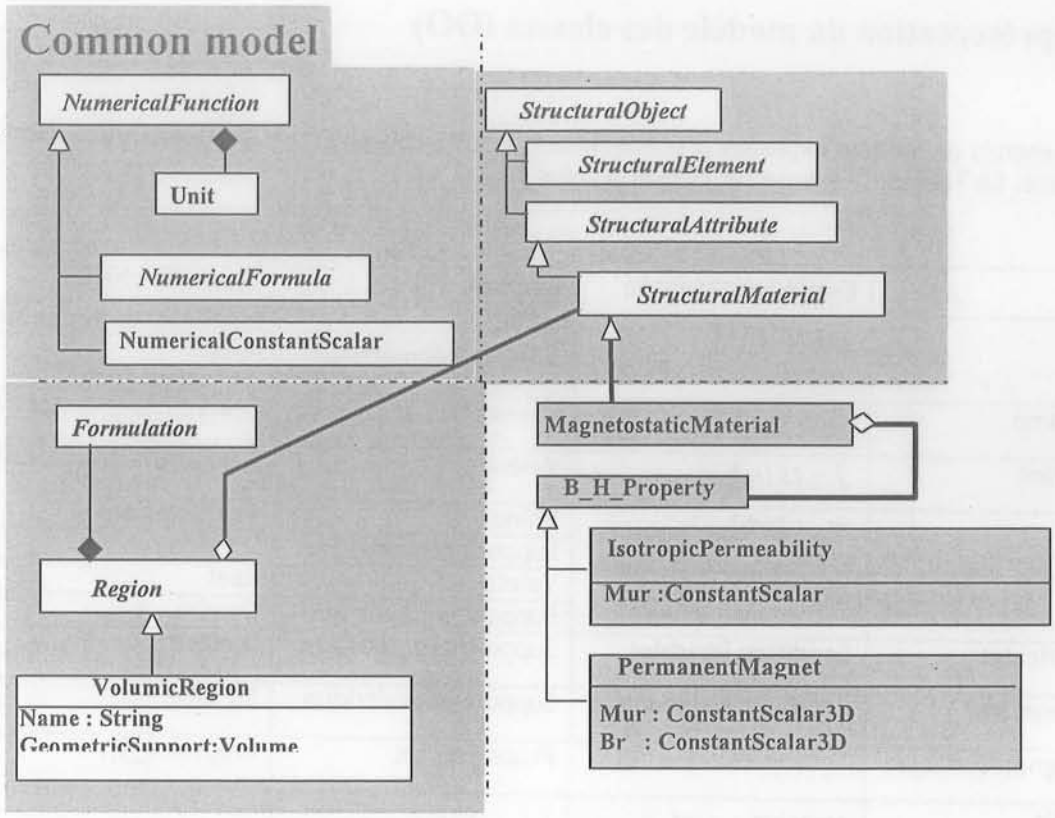


Figure 6-2 L'utilisation du modèle commun pour décrire un modèle magnétostatique

En magnétostatique, les matériaux sont définis par leur dépendance B(H) ; Dans ce sens nous avons introduit une classe B_H_Property et une série des classes qui l'héritent, pour exprimer la grande diversité des matériaux magnétiques (vide, aimants, matériaux non isotropes, matériaux non linéaires, etc.).

Nous allons présenter ici le cas d'un aimant permanent, et le cas des matériaux linéaires et isotropes. L'aimant permanent nécessite deux vecteurs pour définir sa perméabilité relative et son induction rémanente (comme nous pouvons le constater dans le tableau 1). Le matériau linéaire et isotrope nécessite seulement un seul paramètre real (la perméabilité relative), pour exprimer sa relation B(H).

Le code du Listing 1 et Listing 2 représente le code SPML de description des propriétés B(H) pour ces deux types de matériaux

```
#-------------------------------------------------------------------------------
#propMagnetique_HB(2386)
#     |--- permeabiliteLinIso (2645)
#     |
#     |--- AimantCartIso(2652)
#
Entity_propMagnetique_HB = Entity(id='propMagnetique_HB',
                    modelOwner=model,
                    stereotype='ABSTRACT')

Entity_propMagnetique_HB.uiInformation = EntityUi(defaultLabel='propMagnetique_HB',
                                    defaultComment='Type ISOTROPE, ANISOTROPE ou
AIMANT',
                                    reentrantMode='NOT_REENTRANT')
#-------------------------------------------------------------------------------
#propMagnetique_HB(2386)
#     |--- permeabiliteLinIso (2645)
#     |
Entity_permeabiliteLinIso = Entity(id='permeabiliteLinIso',
                    modelOwner=model,
                    stereotype='CONCRETE',
                   supertype = Entity_propMagnetique_HB,
                   hierarchy=Hierarchy_Physical)
permeabiliteLinIso_mur = SimpleField(id='mur',
                                    relatedType=realScalarConstant,
                                    definitionMode='FORCED',
                                    stereotype='AGREGATION',
```

Listing 1

```
#-------------------------------------------------------------------------------
#propMagnetique_HB(2386)
#     |--- AimantCartIso(2652)
#

Entity_AimantCartIso = Entity(id='AimantCartIso',
                    modelOwner=model,
                    stereotype='CONCRETE',
                   supertype = Entity_propMagnetique_HB,
                   hierarchy=Hierarchy_Physical)

AimantCartIso_field_br=SimpleField(id='br',
                                    relatedType=real3DVector,
                                    definitionMode='FORCED',
                                    stereotype='AGREGATION',
                                    uiInformation=AttributeUi(defaultLabel='br',

defaultComment='Aimantation remanante//Br axe X (T)/Br axe Y (T)/Br axe Z (T)',

reentrantMode='NOT_REENTRANT'),
                                    )
```

Listing 2

On peut distinguer dans le code SPML la relation d'héritage entre les classes (**supertype**), les champs qui composent l'entité (**fields**) et l'information qui donnera le format de l'entité dans l'interface graphique.

La classe B_H_Property constitue le type pour la définition du champ « propriété magnétique » de la classe qui définit les matériaux pour la magnétostatique. Il est important que la classe MagnetostaticMaterial hérite la classe StructuralMaterial non seulement pour les informations communes sur les matériaux offertes par la classe StructuralMaterial; mais aussi pour le fait que les autres classes du modèle commun ou les classes dérivées du modèle commun peuvent avoir des références sur la nouvelle classe MagnetostaticMaterial (Figure 6-2).

Par exemple la classe VolumicRegion du Modèle Commun bénéficiera indirectement de la possibilité de description des matériaux d'un point de vue magnétostatique qui sont décrits par la classe MagnetostaticMaterial. La classe VolumicRegion du modèle commun offre la possibilité de regrouper des régions qui sont homogènes comme matériaux et formulations.

L'entité PhysicalDataSet a été créée pour regrouper toutes les régions et toutes les conditions limite qui seront analysées. Cette solution offre un objet complexe qui contient touts les éléments utiles pour une résolution du problème par un solveur Elément Finit. De cette manière, une résolution exigera un seul objet de type PhysicalDataSet, et cet objet référencera toutes les autres données physiques du modèle qui sont utiles à la résolution.

6.3 *La création d'une interface entre la plate-forme SALOME et le solveur éléments finis Flux*

La description en SPML pour le modèle magnétostatique permet de faire la description complète d'un problème de magnétostatique. Une fois le problème spécifié, les données concernant la géométrie et les données physiques seront envoyées comme données d'entrée pour le solveur élément finit capable de résoudre des problèmes de la magnétostatique (le solveur Flux pour notre exemple). Il faut noter que chaque solveur peut avoir son format de données propre à lui.

Nous avons vu dans le chapitre 5 que les données du modèle physique instancié sont accessibles sur le bus CORBA. Il est accessible ainsi le modèle de données physiques, mais dans un format spécifié par la description SPML du modèle. Pour réaliser la connexion entre le modèle physique de la plate-forme et un solveur, une interface sera indispensable pour transmettre les données physiques au solveur.

Sur le bus CORBA, l'accès aux données du problème, sera simplifié grâce à un objet de type PhysicalDataSet. Dans cette manière, pour récupérer toutes les données physiques nécessaires à la spécification du problème, nous avons besoin d'un seul nom, d'un objet PhysicalDataSet qui suffira pour référencer sur le bus CORBA tous les éléments de la description physique.

Dans notre exemple un module spécial a été crée pour récupérer le modèle de données réalisé par le descripteur physique (module Data). L'interface doit réaliser aussi l'adaptation des données du modèle physique dans le format spécifique du solveur Flux, pour permettre au solveur de retourner la solution numérique.

Pour arriver à ce résultat, quelques fonctionnalités (services) seront nécessaires :

- assurer la communication avec le module de description physiques de la plate-forme SALOME (le module DATA).
- un module tampon du côté solveur pour récupérer le modèle de données dans un format du modèle physique similaire celui spécifié pour le module DATA.
- la représentation du modèle de données dans le format spécifique du solveur.

On peut observer dans la Figure 6-3 les modules chargés pour réaliser ces fonctionnalités. La communication avec le module de description des propriétés physiques sera assurée par un client wrappeur qui, à partir d'un élément de type PhysicalDataSet, récupère tous les éléments de la description physique. Ces éléments seront chargés dans le module tampon (Buffered model).

Un module spécifique (Format Converter) réalisera la conversion des données contenues dans le module tampon dans le format spécifique propre au solveur. Ce processus demande deux phases distinctes : une phase de matérialisation, suivie par une phase de composition. [Xexpress]

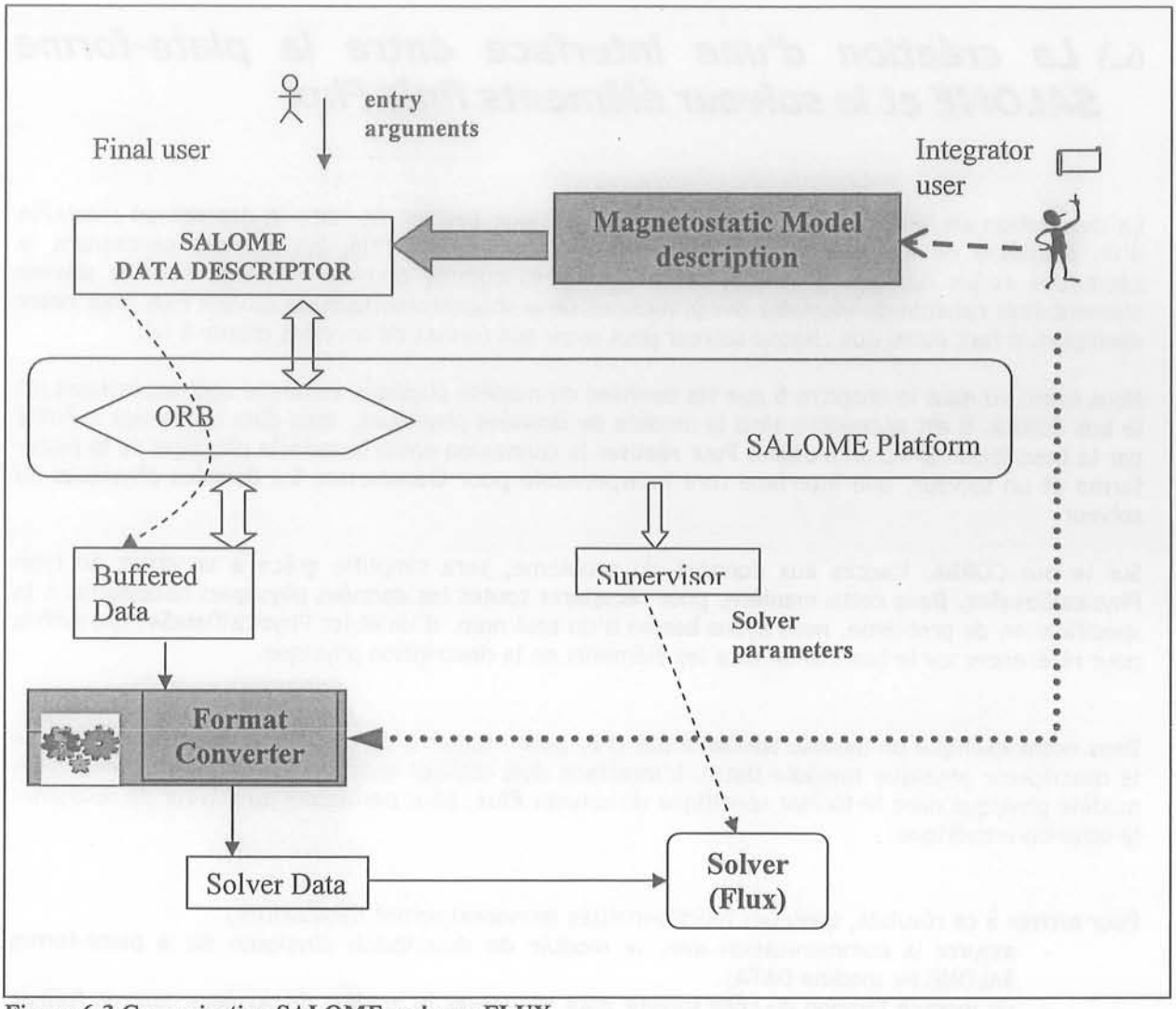


Figure 6-3 Comunication SALOME- solveur FLUX

Dans la phase de matérialisation seront crées les entités de base du solveur, qui dépendent seulement des entités existantes dans le module de description physique de la plate-forme. La création des entités qui dépendent des nouvelles entités du solveur n'est pas possible dans cette phase. La deuxième phase, de composition va créer les entités qui possèdent des attributs qui dépendent d'autres instances du solveur.

6.4 *Utilisation de l'environnement de calcul SALOME-FLUX ; problème magnétostatique*

Une fois le modèle magnétostatique décrit en SPML, après la création de l'interface qui permet de coupler le solveur dans la plate-forme SALOME, l'exemple de la résolution d'une simple problème de magnétostatique permettra de tester la fonctionnalité de la plate-forme, spécialement la communication entre le module de description des propriétés physiques (module DATA) et le solveur.

Le problème exposé demande le calcul du champ magnétique dans le système décrit pour un dispositif électromagnétique. Schématiquement, le circuit magnétique est représenté à la Figure 6-4.

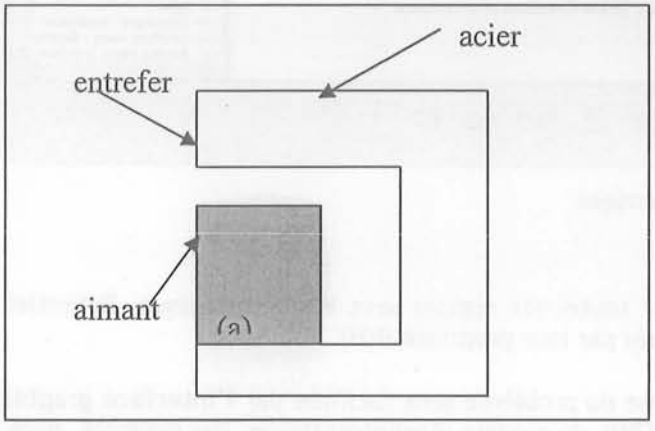


Figure 6-4 : le circuit magnétique du problème

Le modèle considéré dans cet exemple comporte trois régions composées respectivement de trois matériaux : l'air, l'acier et un aimant permanent.

Premièrement la description géométrique du système sera faite à l'aide du descripteur géométrique de la plate-forme SALOME comme on peut voir dans la Figure 6-5.

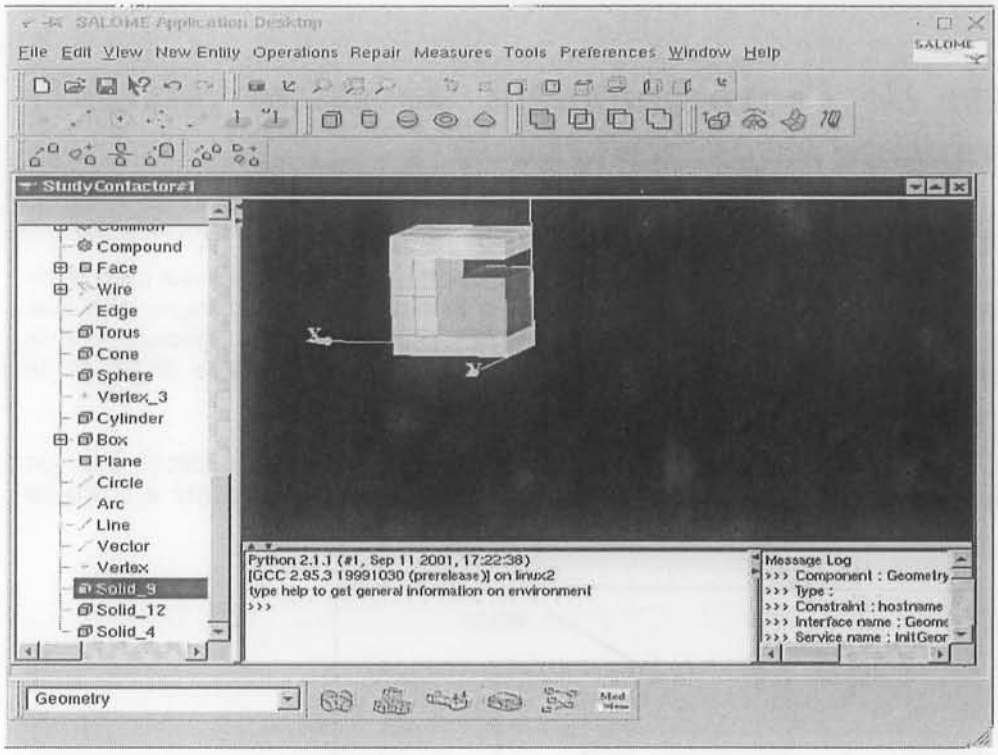


Figure 6-5 Description géométrique

La formulation choisie pour toutes les régions sera Magnétostatique -Potentiel Scalaire Total. Les matériaux seront caractérisés par leur propriété B(H).

La description de la physique du problème sera facilitée par l'interface graphique auto générée à partir de la description en SPML du modèle magnétostatique. Par exemple, nous voudrons décrire le matériau qui caractérise la région « Aimant ». Pour cette région, nous avons choisi un matériau de type aimant permanent, caractérisé par une induction rémanente $B_r = \begin{bmatrix} 0 \\ 0 \\ 0.85 \end{bmatrix} [T]$, et par le vecteur qui définit sa perméabilité relative $\mu_r = \begin{bmatrix} 1 \\ 1 \\ 1.15 \end{bmatrix}$.

Pour créer un nouveau matériau magnétostatique, l'utilisateur est aidé par une liste arborescente qui respecte la hiérarchie des entités du modèle (Figure 6-6a). Le matériau caractérisant l'aimant permanent isotrope, dont sa description SPML a été présenté en 6.2.2. Cette description SPML va générer une boite de dialogue présentée à la (Figure 6-6b) :

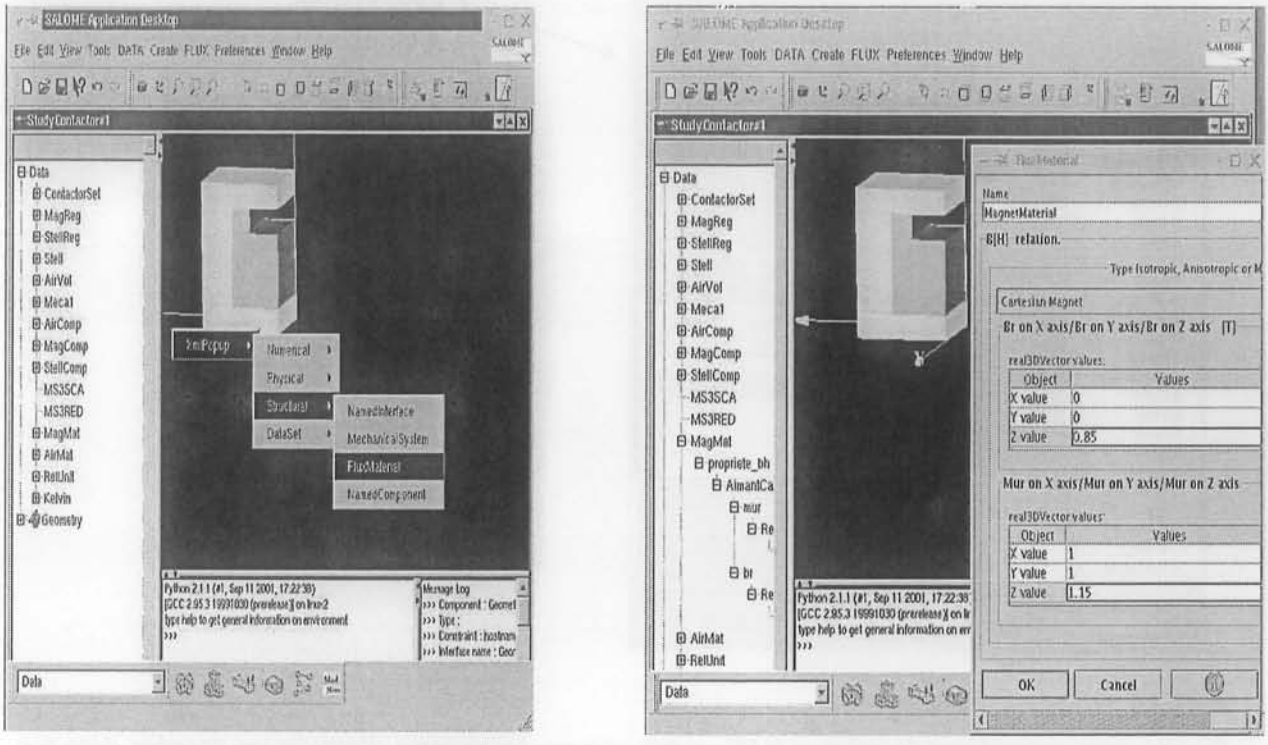


Figure 6-6 (a) : menu contextuel du modèle (b) : Création d'un matériau de type aimant permanent

L'utilisateur a eu la possibilité de choisir le type du matériau Aimant permanent dans une liste de types de matériaux existants dans le modèle magnétostatique. Cette action adaptera les champs de la boite de dialogue pour créer le matériau ayant la propriété B(H) correspondante à un aimant permanent. (Figure 6-6b).

Ensuite la région « Aimant » a été créée, en utilisant le matériau décrit précédemment. La région a créé ainsi un lien entre les propriétés physiques du matériau et le support géométrique, représenté par la géométrie de l'aimant « Solid7 » (Figure 6-7). Pour associer le composant géométrique, il a été utilisé l'arbre de l'étude, qui contient les composants crées par toutes les modules de la plate-forme.

Les autres éléments du problème seront crées similairement (les autres régions, les conditions limites, etc.). La spécification du problème est finalisée par le regroupement de tous les éléments descriptifs dans un élément de type PhysicalDataSet. L'élément de type PhysicalDataSet rendra les éléments de la description physique accessibles sur le bus CORBA de la plate-forme.

Après cette description complète des propriétés physiques, l'utilisateur peut demander la conversion du modèle physique, dans un format propre au solveur. Pour ainsi dire, cela est possible par l'utilisation de la commande « Convert SALOME/FLUX », présente dans l'interface graphique du module DATA (Figure 6-7) Cette action va démarrer le processus de conversion de données du format SALOME, dans le format accepté par le solveur FLUX, dont le mécanisme a été expliqué dans la Figure 6-3.

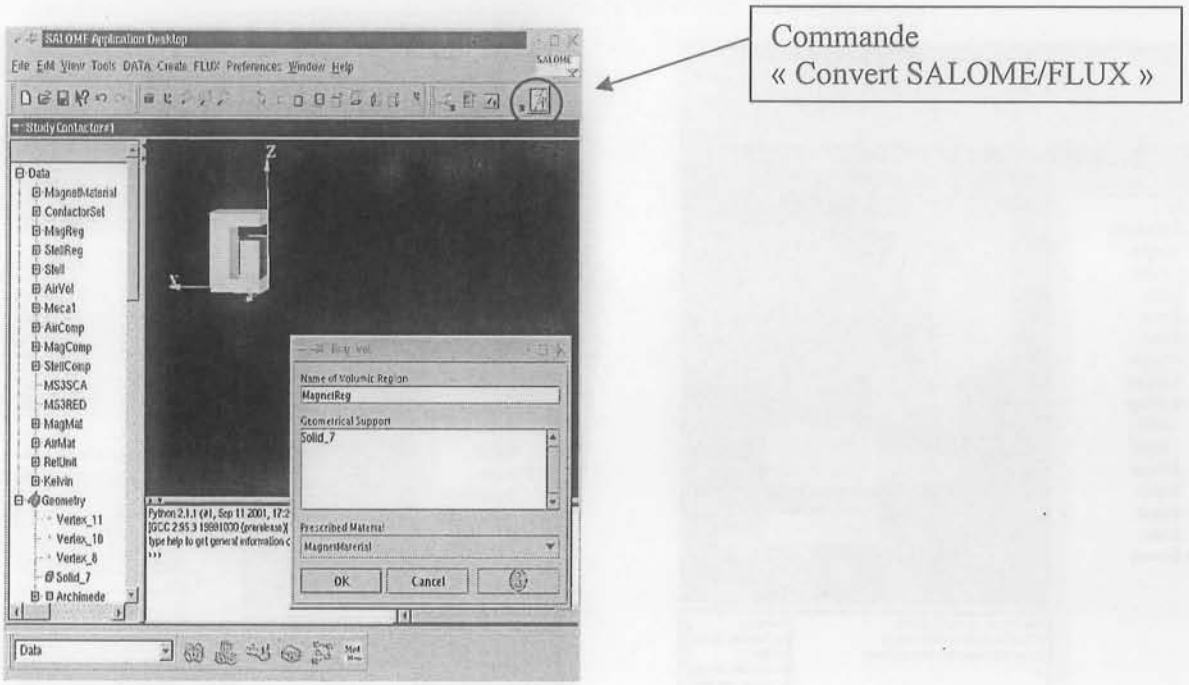


Figure 6-7 Création d'une région.

Ensuite nous avons lance le solveur FLUX, ayant comme paramètres d'entrée les données dans son propre format, que nous venons d'obtenir précédemment. Nous avons ainsi résolu le problème magnétostatique proposé, et les résultats de cette simulation sont montrés à la Figure 6-8

Cette résolution a validé le fonctionnement de notre module de description des propriétés physiques, qui a été utilisé pour la mise en données du problème, et qui a montré aussi sa capacité à intégrer un solveur dans la plate-forme.

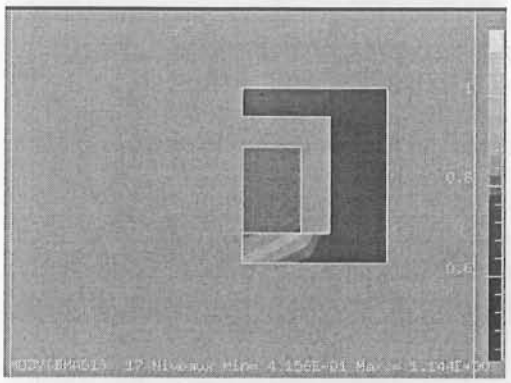


Figure 6-8 les résultats obtenus après la résolution

Il reste que le module superviseur réalise le lancement du solveur FLUX d'une manière automatique, ayant comme paramètres d'entrée les données en son propre format. Ultérieurement, le superviseur peut récupérer les résultats du solveur dans un fichier en format MED, fait qui va permettre la visualisation des résultats en utilisant le module VISU de la plate-forme (Figure 6-9).

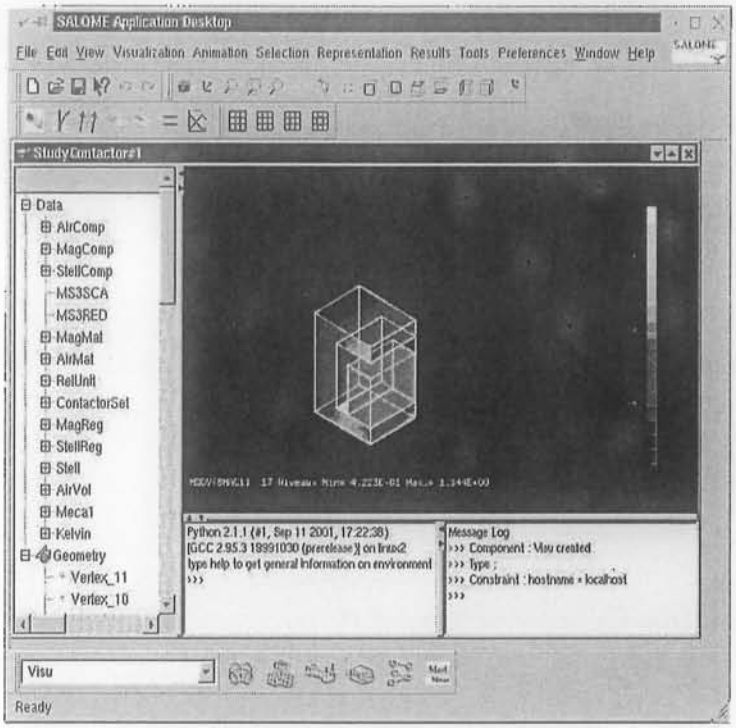


Figure 6-9 Visualisation en utilisant le module VISU

Conclusion

L'intégration du solveur magnétostatique a nécessité deux étapes. Une première étape consiste en description d'un modèle magnétostatique qui contient toutes les entités nécessaires pour la spécification d'un problème magnétostatique. Deuxièmement, la création d'une interface entre le solveur Flux et la plate-forme SALOME, permet de récupérer le modèle physique instancié et d'adapter ce modèle au format demandé par le solveur Flux.

La description en langage SPML du modèle magnétostatique a été facilitée par l'héritage des classes abstraites du *Modèle Commun*, mais aussi par l'utilisation des entités du *Modèle Commun* dans la composition des classes qui représentent la propriété B(H), dans le cas d'aimant permanent. En plus, cette extension du *Modèle Commun* développe l'héritage polymorphique des classes du *Modèle Commun*. Par exemple la classe VolumicRegion du *Modèle Commun* bénéficiera indirectement de la possibilité de description des matériaux d'un point de vue magnétostatique.

Les deux étapes ont permis l'adaptation du solveur Flux dans la plate-forme pour résoudre des problèmes de magnétostatique.
Un exemple d'analyse complète d'un problème de magnétostatique a validé le fonctionnement de la plate-forme ensemble avec un solveur éléments finis.

Conclusion générale

SALOME est une plate-forme générique Open Source de Pré-Post traitement destinée à être spécialisée pour y intégrer des codes de calcul existants et construire des solutions de simulation spécifiques « métier ». Elle intègre plusieurs modules dans une architecture de composants distribués basée sur CORBA. L'ensemble des modules a été conçu pour permettre l'intégration de solveurs ainsi que d'algorithmes de maillage et la spécialisation des propriétés physiques pour un domaine donné.

Notre mission dans le cadre du projet SALOME a été la conception et la mise en œuvre technologique du module DATA, dédié à la description des propriétés physiques.

La description effective des propriétés physiques d'un problème est conditionnée par l'existence d'un modèle de données qui dépend du domaine de la physique étudié. En ce sens, nous avons réalisé un nouveau langage dédié à la description des modèles de données physiques : le SPML (SALOME Physics Modelling Language). Un métamodèle dédié à la description des propriétés physiques offre au langage SPML sa base sémantique.

La syntaxe du SPML est issue et conforme au langage Python. Pour réaliser une partie commune de communication entre des modèles représentant différents domaines de la physique, il a été développé un Modèle de Données Commun. Le Modèle de Données Commun a été matérialise par une librairie SPML réutilisable.

Nous avons ainsi réalisé un outil complet de description des propriétés physiques : un langage dédié et une librairie standard.

La réalisation de l'IHM, notamment l'adaptation automatique de l'interface graphique aux modèles physiques décrits en SPML, font du module DATA un outil performant, qui permet une adaptation facile de la plate-forme, pour tout domaine de la physique.
Des fonctionnalités supplémentaires implémentées assurent la persistance et la communication des données à travers le bus CORBA.
Les facilités du langage Python ont permis l'implémentation des outils qui assurent la dynamicité du module DATA. Cela permet le chargement de la description SPML d'un modèle physique directement dans un espace de mémoire privée, sans passer par du code généré.

La connexion d'un solveur à la plate-forme SALOME repose sur deux étapes préalables, effectuées par l'intégrateur :

- Une première étape consiste dans la description du modèle de données propre au domaine de la physique concerné par le problème.
- Deuxièmement, la création d'une interface entre le solveur et la plate-forme SALOME, permettant la récupération du modèle physique instancié et d'adapter ce modèle au format demandé par le solveur.

Nous avons réalisé une première connexion du solveur Flux, dans le cadre de la plate-forme pour des analyses magnétostatiques sur des problèmes décrits en SALOME.

La description en SPML du modèle Magnétostatique nous a permis de tester le fonctionnement du langage SPML, ainsi que les facilités du Modèle Commun. L'adaptation des données concernant la description physique réalisée sous la plate-forme dans le format accepté par le solveur a permis le développement d'une méthodologie de connexion des solveurs en général. La récupération des données du côté solveur a validé aussi le mécanisme de communication entre le module DATA et le reste de la plate-forme.

Un des objectifs de SALOME est bien sûr de favoriser les couplages entre ces solveurs. Plus la partie commune des Modèles de Données des solveurs à coupler sera importante, plus les couplages seront aisés et robustes. Ainsi l'intégration d'autres solveurs permettra de faire évoluer le Modèle de Données commun. Sans viser un inaccessible modèle de données universelles, il est souhaitable de proposer un modèle de données commun plus riche que celui existant et une méthodologie de dérivation de ce modèle.

Suite au premier projet RNTL SALOME, l'ensemble des programmes sources de la plate-forme SALOME est maintenant diffusé en Open Source (www.salome-platform.org).

Un nouveau projet RNTL SALOME-2 a démarré et sur les 3 ans qui viennent ce projet va continuer d'enrichir la plate-forme en nouvelles fonctionnalités, principalement au niveau des algorithmes de maillage, du calcul parallèle et de la liaison avec la réalité virtuelle.

BIBLIOGRAPHIE

[Atkinson] C. Atkinson, T. Kuhne and B. Henderson-Sellers « To Meta or Not to Meta », JOOP vol. 13, no. 8, pp. 32-35, December 2000

[Bonjour] M. Bonjour, G. Falquet, J. Guyot, A. Le Grand. « Java: de l'esprit à la méthode », Vuibert, 1999, pp. 544

[Bean] Greg Voss : « Java Beans », disponible à http/java.sun.com

[Carlson] David Carlson : « Modelling XML Applications with UML », Addison-Wesley, 2001

[Cascade] Matra Datavision : site Open Cascade accessible à http://www.opencascade.org/

[CORBA] OMG Group : « The Common Object Request Broker: Core Specification (CORBA) v 3.0 », disponible à www.omg.org

[Costa] Maurício CALDORA COSTA, « Optimisation de dispositifs électromagnétiques dans un contexte d'analyse par la Méthode des Éléments Finis », Thèse de doctorat, I.N.P. Grenoble, juin 2001

[Cox] Jordan J. Cox, « Product Templates : *A parametric approach to mass customisation* , apparue dans l'ouvrage « CAD Tools and algorithms for product design », pp.3-16, sous la direction du P. Brunet

[Dix] Alan Dix - Janet Finlay -Gregory Abowd - Russell Beale , « Human-Computer Interaction », Prentice Hall, 1998

[Dular98] P. Dular, C. Geuzaine, F. Henrotte , W. Legros", "A general environment for the treatment of discrete problems and its application to the finite element method", IEEE TR MAG,V34, 5, pp. 3395-3398, 1998.

[Dular99] P. Dular, C. Geuzaine, F. Henrotte , W. Legros", "An evolutive environnement for teaching the finite element method in electromagnetism", IEEE TR MAG,V35, 5, pp. 1682-1685, 1999

[Englander] R. Englander « Developing JavaBeans », O'Reilly and Associates, 1997

[Eyraud] L.Eyraud, Y. Friteau, P Conrad « Conversion de l'energie. Transferts thermiques »

[Fel 02] Mouloud Féliachi « Couplage magnétothermique » ; chapitre apparu dans « Electromagnétisme et problèmes couplés » (Traité EGEM, série Génie électrique: électromagnétisme et éléments finis 3), Auteur: MEUNIER Gérard Date de parution: 12-2002.

[Fireteanu] V.Fireteanu "Procesarea electromagnetica a materialelor", Editura Politehnica, 1995

[Flux] Societé CEDRAT « Flux3D : Manuel de référence », 2002

[Geib] Jean-Marc Geib, Christophe Gransart, Philippe Merle, « CORBA : des concepts à la pratique », Inter-Editions, France, 1997

[Georgescu] Cristian Georgescu : « Code generation Templates using XML and XSL », C/C++ Journal Jan 2002, pp. 6-19

[Gheysens] R. Gheysens, J.Gosse, P.Petitdidier « Enseignement de l'electrothermie », DOPEE '85, 1987

[Golovanov] Golovanov C. « Dévéloppement de formulations éléments finis 3D, en potentiel vecteur magnétique: application à la simulation de dispositifs électromagnétiques en mouvement », Thèse Génie électrique, INPGrenoble 1997

[Grisby] Sai-Lai Lo, David Riddoch, Duncan Grisby: « The omniORB version 3.0User's Guide » AT&T Laboratories Cambridge, disponible à : http://www.uk.research.att.com/omniORB/doc/3.0/omniORB/index.html

[Harold] Elliotte Rusty Harold, W. Scott Means « XML in a Nutshell », O'Reilly & Associates, June 2002

[HDF] The NCSA HDF Home Page « Hierarchical Data Format (HDF)» disponible à http://hdf.ncsa.uiuc.edu/

[JAC] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard « Object-Oriented Software Engineering: A Use Case Driven Approach », Addison-Wesley, 1992

[Ketani] N. Ketani, D.Mignet, P. Paré « De Merise à UML », Eyrolles 1998

[Lutz1] Mark Lutz, David Ascher : « Learning Python » O'Reilly & Associates, April 1999

[Lutz2] Mark Lutz « Programming Python » , O'Reilly & Associates, 2001

[Ma] Singva Ma : « Définition d'un Protocole d'Application STEP pour la Simulation en Electromagnétisme », Thèse de doctorat, I.N.P. Grenoble, juin 2001

[Matteo] I.Diaz and A Matteo "Objectory Process Stereotypes", JOOP, vol. 12, no. 3, pp. 29-38, June 1999

[Mocanu] Mocanu C.I. "Bazele electrotehnicii, Teoria campului electromagnetic", Editura didactica si pedagogica, Bucuresti 1991

[MOF] OMG : Meta Object Facility (MOF)SpecificationDocument September 1, 1997, disponible à www.omg.org

[Muller] P.A. Muller « Modélisation objet avec UML », Eyrolles 1998

[OMT] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W. « Object-Oriented Modelling and Design », Prentice-Hall, Englewood Cliffs, N.J., 1991

[Open] Open Source Initiative « The Open Software License v. 1.1 » (disponible à http://opensource.org/licenses)

[Python] site : www.Python.org

[Qt1] Matthias Kalle Dalheimer « Programming with Qt », O'Reilly & Associates, 2002

[Qt2] Trolltech « Qt Reference Documentation », disponible à http://www.trolltech.com/

[Reddy] Reddy, J. N. « An Introduction to the Finite Element Method », McGraw-Hill Book Co., New York, 1984, 1993, 495 pp

[Roques] P. Roques, F. Vallée « UML en action », Eyrolles, 2003

[Salome] The SALOME projet SALOME, Spécifications on-line, disponible à http://www.opencascade.org /SALOME/Documents

[Schmidt] I. Pyarali and D. C. Schmidt "An Overview of the CORBA Portable Object Adapter", ACM StandardView, vol. 6, Mars 1998 (disponible à http://www.cs.wustl.edu/~schmidt/PDF/POA.pdf)

[Sellers1] B. Henderson-Sellers « Enhancing the OPF Repository »

[Sellers2] Jean-Michel Bruel, Brian Henderson-Sellers, Franck Barbier, Annig Le Parc, et Robert B. France « Improving the UML metamodel to rigorously specify aggregation and composition », IEEE Comunications Magazine, Vol 35, No 2, February 1997

[Sommerville] Ian Sommerville « Software Engineering », Ed. Addison Wesley, 1996

[STEP] « STEP on a Page », accessible à http://www.mel.nist.gov

[Straton] Straton J.A « Electromagnetic theory », Mc Graw-Hill Bock Company, New York, 1941

[Swig] SWIG 1.1 Users Manual disponible à http://www.swig.org

[Tanenbaum] Andrew S. Tanenbaum, Maarten van Steen « Distributed Systems: Principles and Paradigms », ed. Hardcover 2000

[Tardiveau] Max Tardiveau : « The Meta Object Facility : The final frontier of modelling », Journal of Object-Oriented Programming, June 1999, pp 8-11 September 1, 1997

[Toumi] Imad Toumi, Emmanuel Dorlet, Sana Abou-Haidar, Pierre Bruno « Le projet SALOME, plate-forme en Open Source de liaison CAO/Calcul », article présenté à MICAD, 2001

[Touzot] G. Dhatt, G.Touzot « Une présentation de la méthode des éléments finis », Ed. Maloine, PARIS

[Vartiainen] Tuukka Vartiainen : « JAVA BEANS AND ENTERPRISE JAVA BEANS », disponible à http://www.cs.helsinki.fi/u/campa/teaching/tukka-final.pdf

[Vinoski] Steve Vinoski « Integrating Diverse Applications Within Distributed Heterogeneous Environments », IEEE Communications Magazine, Vol. 14, No. 2, February, 1997

[Vlist] Eric van der Vlist « XML Schema », O'Reilly & Associates, June 2002

[VTK] Kitware : site accessible à http://www.kitware.com/

Liste de publications concernant le travail de thèse

« Unified physical properties description in a multi-physics open platform » S. Giurgea, T. Chevalier, J.L. Coulomb, Y. Maréchal, article accepté pour la revue « IEEE Transactions on magnetics» Vol. 39, No.3, Mai 2003, et présente à la conférence : «The Tenth Biennial IEEE Conference on Electromagnetic Field Computation » - Perugia, Italie, Juin 16-19 2002.

S.Giurgea, T. Chevalier, J.L. Coulomb, Y. Maréchal: "SALOME, a multi-physics open platform. Integration of an electromagnetic solver", Przeglad Elektrotechniczny R. LXXIX 10/2003 pp 728-731, article invité à la conférence « International Symposium on Theoretical Electrical Engineering », Varsovie , Pologne, Juillet 6-9 2003

«The SALOME project, an OpenSource framework linking CAD/Computation to easily develop and integrate electromagnetic solvers» T. Chevalier, J.L. Coulomb, Y.Maréchal, S. Giurgea, article présenté à la conference: «The Tenth Biennial IEEE Conference on Electromagnetic Field Computation » - Perugia, Italie , Juin 16-19 2002

« Conception d'une plate-forme générique de simulation multi physique; intégration d'un solveur électromagnétique» S.Giurgea, Y. Maréchal , T. Chevalier, J.L. Coulomb, présenté à : « 4ème Conférence Européenne sur les Méthodes Numériques en Electromagnétisme » Toulouse, 28 - 29 et 30 Octobre 2003

Annexe

Salome Physics Modeling Language, SPML Semantics

version 1.0

21 May 2002

By T.Chévalier, S.Giurgea, Y.Maréchal

Contents

1 Introduction

This document is the reference manual of the language that is dedicated to definition of models for the physics domains used in SALOME environnement.

The SPML specification consists of following interrelated parts:

- The SPML semantics (definition)
- The SPML data structure
- The SPML notation (textual command language)

2 Language architecture

The SPML is conformed to the four-layer metamodel architecture proposed by the UML (See: UML v1.1 Semantics document)

The generally accepted conceptual framework for metamodeling is based on an architecture with four layers:

- meta-metamodel
- metamodel
- model
- user objects

These functions of these layers are summarized in the following table:

Layer	Description	Example
meta-metamodel	The infrastructure for the metamodeling architecture. Defines the language for specifying metamodels	MetaClass, MetaAttribute, ...
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model	Entity, Attribute, ...

model	An instance of a metamodel. Defines a language to describe an information domain	Region, Units, …
user objects (user data)	An instance of a model. Defines a specific information domain	AirRegion, Meter, …

This document concerned the description of the metamodel layer.

3 Global Overview

The Figure 1 presents the global structuration of the SPML metamodel. It is composed of two package, one containing data structure and an other containing user interface (UI) structure. The data package is itself devided in tree subpackages. ApplicationModelPackage containing all structures to describe applications and models. TypeAttributePackage containing structures to describe objects with their attributes and methods. CollectionPackage containing structures to describe behaviour of collections such as array, list, etc. The 5 previous described packages are used to give a more readable view of the SPML metamodel and are not used for namespace. (This mean "TypeAttributePackage.Entity" syntax is not valid and you should simply used "Entity").

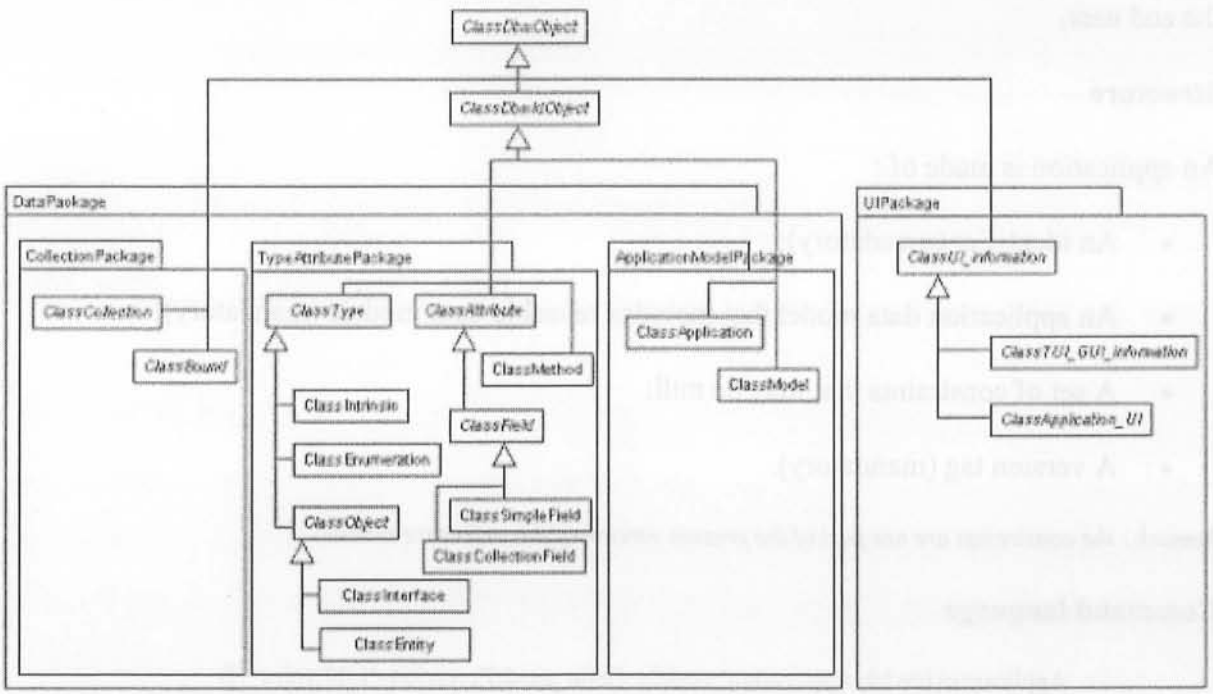


Figure 1 - Global Overview

4 The data strutures used for application and data models definitions.

These structures are designed to enable definition of an application, including the data model and command model. They are derived from the UML formalism, which is extended when necessary.

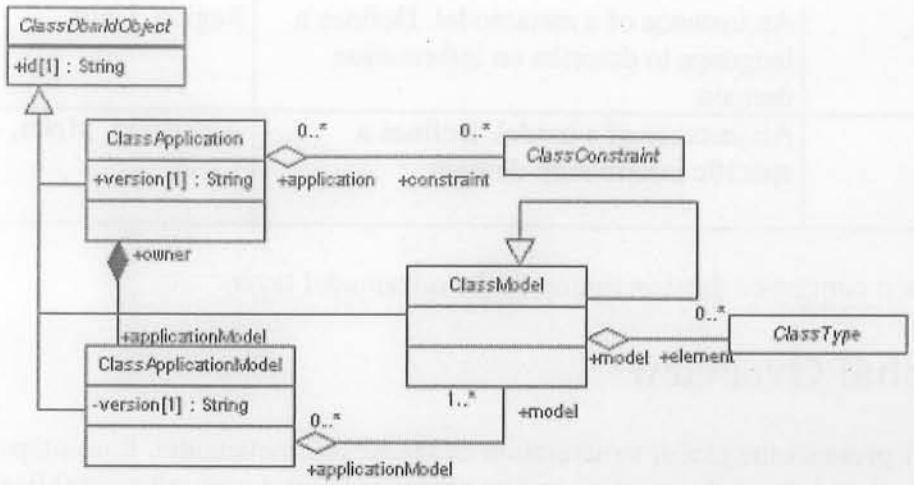


Figure 2 - Application and Models structure

4.1 The application

Definition

An application is the executable item. It represents a set of functionnality clearly identified for the end user.

Structure

An application is made of :

- An identifier (mandatory);
- An application data model that includes reusable data models (mandatory);
- A set of constraints that may be null;
- A version tag (mandatory).

Remark : the constraints are not part of the present version of this reference manual.

Command language

```
Application(id='MyApplication',models=[mon_model],version='1.00',rules=[])
```

Related links

The application definition will be enriched by user interface and regional information.

Future evolutions

4.2 The data models

Definition

A data model is a set of reusable data structures. They group a set of concepts related to a particular domain of activity. The data structures are normally related to one and only one data model but they can reference data structures defined in other data models. An application is defined by a combination of several data models.

A data model can be simply derived or specialised, the same way that entities do, to extends or restrict a domain of activity. These inheritance relations lead to a tree called the inheritance data model tree.

Structure

A data model is defined by a unique mandatory identifier. By convention, this name is full lower case, each word being separated by an underscore.

Command language

Related links

Future evolutions

The use of external data structure inside a data model may be allowed and checked by introducing a attribute imports, that is a list of data models.

4.3 The package

Remark: package are supported but not competely implemented

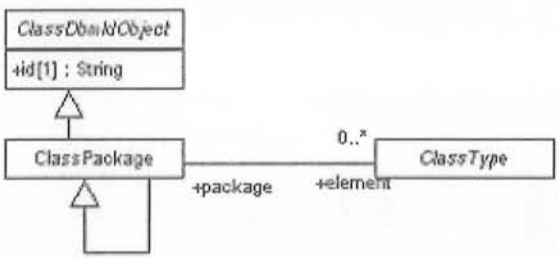


Figure 3 - Package structure

Definition

The package defines the physical layout of the source code of the entities. The packages are organised like directories, and thus have a tree structure.The head of the tree is a main package, whereas the other package are named sub packages.

The tree defined by the packges and the tree defined by the data models are usually different since the packages are the physical layout whereas datamodel is the logical organisation

Any data structure belongs to at most on package (either main package or sub package).

Structure

A package is defined by a name (mandatory). By convention, the name in in lower case and each constituvie word is separated by an underscore character.

A sub package also posseses a reference towards its father package.

Data structure of packges

Command language

Related links

Future evolutions

5 The type definitions

The type that can be modeled are :

- the intrinsic types
- the enumerations
- the interfaces
- the entities

All these types are related to a data model and a an optional package.

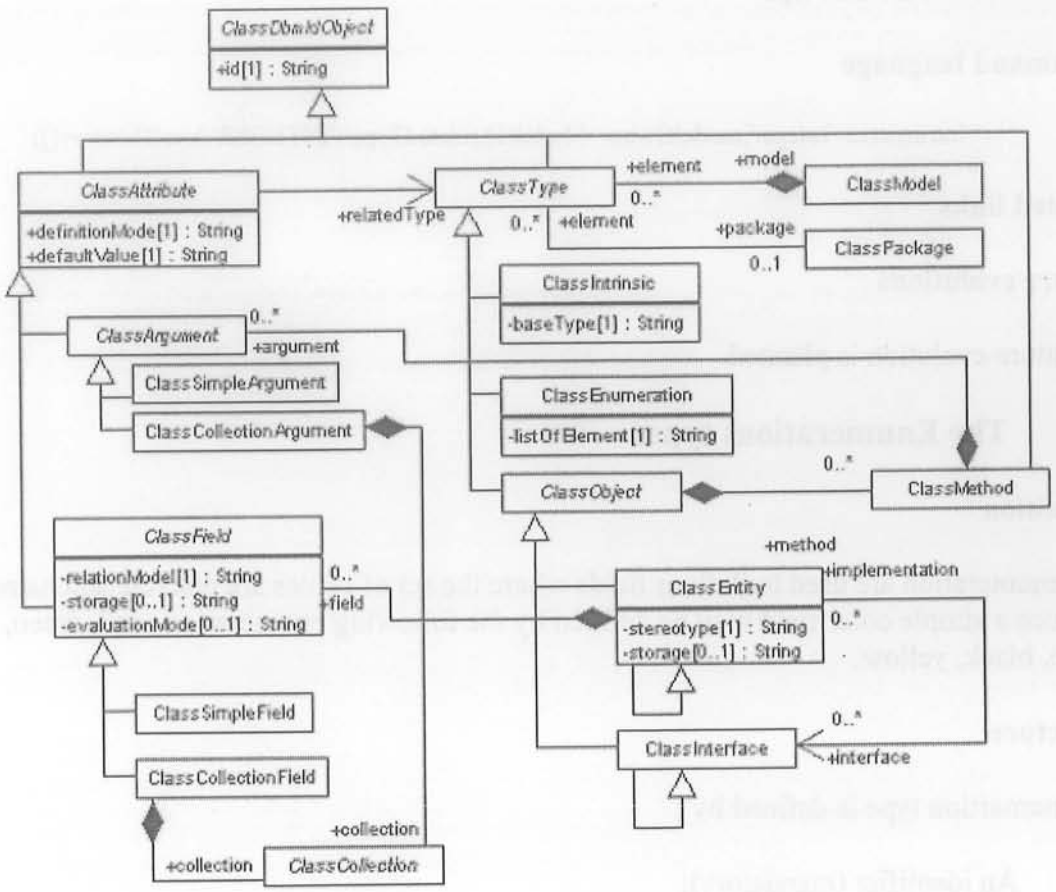


Figure 4 - Type and Attribute structure

5.1 The intrinsic types

Definition

The intrinsic types defines the simplest data structure that can be manipulated by an application. Among the instrinsic types, are the standard base types : integers, reals, strings, voids, booleans,... Intrisic types are also used for the definition of dedicated new base types : URL, image, name, ...

Structure

An intrinsic type is defined by :

- An identifier (mandatory);
- A reference to a data model (mandatory);
- A reference to a package (optional);
- An underlying base type chosen among interger, real, numeric, string, numeric_or_string, boolean, void and salome_object_reference (mandatory).

Data structure of intrinsic type

Command language

```
Intrinsic(id='Integer',modelOwner=Model(1),relatedType='INTEGER',localHistory=[])
```

Related links

Future evolutions

No future evolution is planned.

5.2 The Enumerations types

Definition

The enumeration are used to defines fields where the set of values are discrete and named. For instance a simple color field may be defined by the following enumeration : red, green, blue, white, black, yellow.

Structure

An enumartion type is defined by :

- An identifier (mandatory);
- A reference to a data model (mandatory);
- A reference to a package (optional);
- Astring composed of a list of semicolon separated name (mandatory).

Data structure of enumeration type

Command language

```
Enumeration(id='DiscreteColor',modelOwner=Model(1),val='red;green;blue;yellow;white;black')
```

Related links

Future evolutions

In the future, the tags in the val field of any enumeration should be multi language.

5.3 The interface types

Definition

The interface type is used to defined the object oriented interface concept: tehy are behaviours that can be implementd by entities. The interfaces only have the methods declaration that are used to express the behaviour modeled. An interface can extend several other interfaces.

Structure

An interface type is defined by :

- An identifier (mandatory);
- A reference to a data model (mandatory);
- A reference to a package (optional);
- A set of interfaces that are extended (optional or eventually null)
- A set of methods (optional or eventually null).

Data structure of Interface type

Command language

```
Observable = Interface(id='Observable',modelOwner=Model(1))

Interface(id='Listener',modelOwner=Model(1),supertypes=[Observable],

methods=[Method(id='notify',arguments=[SimpleArgument(id='objectId',relatedType=Type(2),def
initionMode='FORCED'],returns=Type(2))])
```

Related links

See also Method, Entity.

Future evolutions

The interface notion is not used yet. It has been kept for those who want to generate code from meta description.

Remarks

This concept of interface is translated into a pure abstract class in C++ and interface in Java.

5.4 The entity types

Definition

The entities are the key concept used in an Application. Any specific entity type is defined by its data (fields) and services (methods). An entity may implement a set of Interfaces and extends at most one base entity type. This leads to a simple inheritance of entities and multiple implementation of interfaces scheme.

An entity type also includes a stereotype that is chosen among:

- ABSTRACT : if the entity type is abstract in the OOP sense and that this abstract entity type is used for polymorphism aspect. For instance a Circle will be decalred abstract, if it is derived into Circle_3_points and Circle_center_radius.
- . CONCRETE : if the entity type is concrete.
- NODE : Similar to ABSTRACT except that a NODE entity type does not declares all the behaviour that its CONCRETE entity type defines.
- WRAPPER : A WRAPPER entity type is used to reformat or encapsulate a set of behaviour.

ABSTRACT and CONCRETE are the 2 major stereotypes.

Structure

An entity type is defined by :

- An identifier (mandatory);
- A stereotype chosen between CONCRETE, NODE, ABSTRACT, WRAPPER (mandatory)
- A reference to a data model (mandatory);
- A reference to a package (optional);
- A reference on the entity type extended (optional);
- A set of interfaces that are implemented (optional or eventually null);
- A set of methods (optional or eventually null);
- A set of fields (optional and eventually null);
- Additional storage information (either PERSISTENT or TRANSIENT) (optional);

Data structure of Entity type

Command language

```
myModel = MainModel(id='MyDataModel',version='1.00')

point = Entity(id='Point',modelOwner=monModel,stereotype='CONCRETE')

point.fields=[SimpleField(id='color',relatedType=Type(3),definitionMode='FORCED',stereotype='
ASSOCIATION',datatype='PERSISTENT',evaluationMode='NONE')]
```

Related links

See also Field, Method, Interface.

Future evolutions

Stereotype list may be different and user defined.

Remarks

- The fields also have a storage information that overloads the information given in the entity type.
- The Entity type data structure will be enhanced by UI information.

5.5 The fields

Definition

The fields are the data of the entity types. A particular field belongs to one and only one entity and may not be shared by others.

The fields are used to define the relations. They reference a data structure that is the related type in the field.

The fields are either simple or collection fields. In the latter case, a field express a relation between an object and a collection of objects.

A field has a definition mode which is chosen among :

- FORCED : the user input of the field is necessary for the object to be valid.
- FINAL : the user input of the field is complusory and may not be changed later on. This type may be used for identifiers for instance.
- OPTIONAL : the user input is optional for this field. If no input is provided, a default value is given.
- DERIVED : the value of the field is not a user input but a result of an evaluation process. It may however be useful to a user. For instance, if the user defines a Circle by 3 points, the program may also need the coordinates of this Cricle to perform computations. The center is then a derived field.
- INTERNAL : an internal field is not input by the user nor it is deemed useful at any moment except for internal treatement.

A field is also classified into 4 categories according to an extended UML formalism.

- ASSOCIATION : it is the default relation when no other relation model may be used. It associates an instance of the entity type to ro sevearl instances of an other entity type.

- AGGREGATION : An AGGREGATION defines a relation of « part of » type. It can be seen as a « is made of » relation. However, there is no life cycle relation between the parts objects and the bigger object.

- COMPOSITION : A composition is an aggregation where both part and complex objects have same life cycle. The parts are somehow private internal objects of the complex object. In other words, the inverse relation has a maximum and minimum cardinality of 1.

- IDENTIFICATION : The field is used as an identifier, i.e. The identifier is unique and may be considered as an index.

Remark : relations between an object and an intrinsic type or an enumeration is considered as a composition.In this case, a default value may be given.In cases other than relation with intrinsic or enumeration types, default values are not used.

The fields have an attribute « evaluation mode » that may be used in the evaluation process. During instance creation or modification, an evaluation may be performed. The value of the evaluation mode flag changes the way the evaluation process runs.

- NONE : the edition of the instance used in this field does not require a new evaluation of the instance having this field.

- BACK-PROPAGATION : the edition of the instance used in this field induces a new evaluation process for the instance having this field.

End, the fields have an storage falg, either PERSISTENT or TRANSIENT, that overloads the storage flag of the entity.

Structure

A field is made of :

- An identifier (mandatory);
- A choice between simple field, set, map, array or list. In the case of collection fields, minimum and maximum number of instances are also needed (mandatory);
- A related type to be chosen among all existing types (mandatory);
- A definition mode to be chosen between FORCED, FINAL, OPTIONAL, DERIVED, INTERNAL (mandatory);
- A relation model to be chosen among ASSOCIATION, AGREGATION, COMPOSITION, IDENTIFICATION (mandatory);
- Additional storage information (either PERSISTENT or TRANSIENT) (optional);
- A default value (optional);

- A validity domain (not used yet);
- An evaluation mode to be chosen between NONE and BACK_PROPAGATION (optional).

5.6 The Arguments

Definition

The arguments are the data of the methods types. A particular argument belongs to one and only one method and may not be shared by others.

The arguments are either simple or collection arguements.

A argument has a definition mode which is chosen among :

- FORCED : the user input of the argument is necessary for the method to be valid.
- FINAL : the user input of the arguement is complusory and may not be changed later on.
- OPTIONAL : the user input is optional for this arguement. If no input is provided, a default value is given.
- DERIVED : the value of the argument is not a user input but a result of an evaluation process. It may however be useful to a user.
- INTERNAL : an internal argument is not input by the user nor it is deemed useful at any moment except for internal treatement.

Structure

An argument is made of :

- An identifier (mandatory);
- A choice between simple simpleArgument or collectionArgument. In the case of collection argument, minimum and maximum number of instances are also needed (mandatory);
- A related type to be chosen among all existing types (mandatory);
- A definition mode to be chosen between FORCED, FINAL, OPTIONAL, DERIVED, INTERNAL (mandatory);
- A default value (optional);
- A validity domain (not used yet);

5.7 The Collection

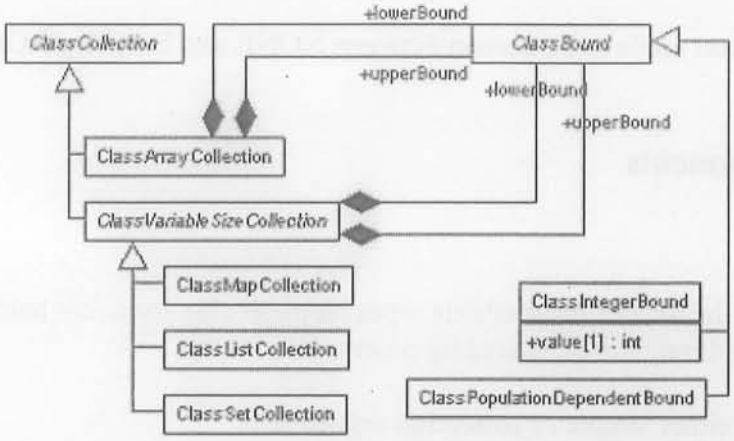


Figure 5 - Collection structure

Definition

- The collection definition the kind of list a field or an argument can be. Collections can be of two kind. Fixed value collection called ArrayCollection and variable size collection called VariableSized. Each collection needs an upper bound and lower bound. For ArrayCollection Bound are necessary IntegerBound. For other collection type bound can be

Structure

A collection is made of :

- On lower bound (mandatory);
- On upper bound (mandatory

6 The User Interface information

In addition to all previous data, it is possible and useful to add some extra information related to the User Interface management. A UI engine will interpret these information to create dynamically the dialog with the user.

The dialog may take 2 aspects : Textual User Interface (TUI) and Graphic User Interface (GUI). Both modes are interpreted using the UI information associated to the preious data structure. Thus the UI information will define manage the regional information (dialog in several languages), part of the layout in the dialog boxes and the command names that will be used in the TUI. All these information are generally optional. If not provided, then the data structure names are used, and the dialog will be less user friendly.

The TUI information may either be some new data structures or additionnal optional information added to the previous structures.

Remark : The UI information are described after the data model for conviniency only. In a normal session, both data model and UI information can be defined at the same time

6.1 The common information

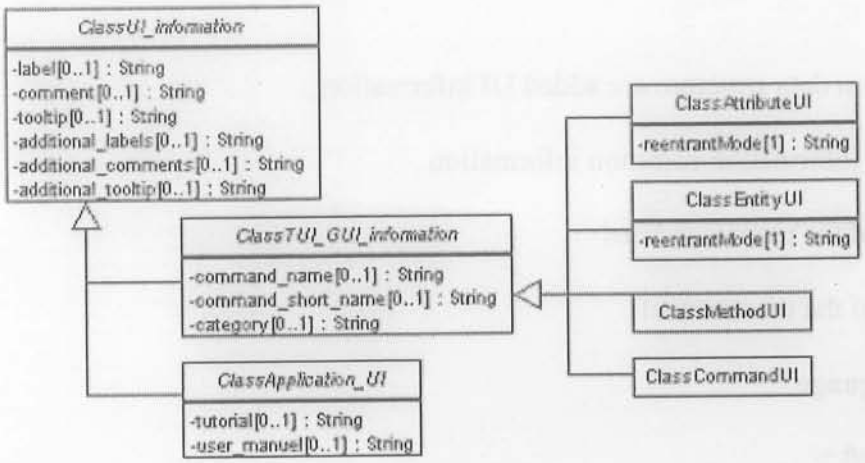


Figure 6 - UI information structure

The UI_information

The UI_information type is the base type for any UI information. It includes the following fields :

- label the default language label
- comment the default language comment
- tooltip the default language tooltip
- additional_labels the other languages labels
- additional_coments the other languages comments
- additional_tooltips the other languages tooltips

TUI_GUI_information

The TUI_GUI_information derives the UI_information type and is designed to be used by any entity that is to be used in a GUI or TUI mode.

It adds the following fields :

- command_name the name to be used for command language. If not provided, the data structure name is used.
- command_short_name the abbreviation of the command name if any (not used yet);
- category this field is used to dispatch the attributes into tabs in the GUI dialog.

Data structure of UI_information and TUI_GUI_information

6.2 The applications

Structure

To the application data structure are added UI information

- The UI_information common information
- An url of the tutorial manual
- An url of the user manual

Command language

```
mySoft =
Application(id='MyApplication',models=[mon_model],localHistory=[],version='1.00',rules=[])

mySoft.uiInformation = ApplicationUi(defaultLabel='my
soft',userManual='../Documentation/MySoft.html',tutorial='../Documentation/MySoftTutorial.html'
))
```

Full data structure of the applications including UI information

6.3 The categories

Definition

The categories allow to dispatch the fields in tabs. They are only used in the GUI dialog.

Structure

The categories are quite simple structures. They are made of 2 fields.

- The label in the default language
- The label in the other languages

Data structure of the categories

6.4 The entity type

Structure

To the entity type data structure are added all information comming from the TUI_GUI_information common information. These information have in that case some specific behavior:

- The label becomes the identifier of the type as soon as the user as to use this entity type.
- The comment is used as a label in the widget displaying all the derived types.

- The category field is used as default category. It can be overloaded by the input a different category in the fields of this entity type.

Some additional information are also provided:

- The reentrance mode tells wether a single instance or multiple instances creation mode is prefered for this entity type. The values are :
- REENTRANT : multiple instances creation mode
- NOT_REENTRANT : single instance creation mode

Full data structure of the entity type

6.5 The fields

Structure

To the field type data structure are added all information comming from the TUI_GUI_information common information. These information have in that case some specific behavior:

- The label becomes the identifier of the field as soon as the user as to use this field.
- The category field is used to dispatch the fields in tabs. It overloades the input of the category for the related entity type if any.

Some additional information are also provided:

- The reentrance mode tells wether in a multiple instances creation mode, the previous value of the field is proposed as a default value. The values are :
- REENTRANT : the previous value of the field is kept in a multiple instances creation mode
- NOT_REENTRANT : the previous value of the field is not the default value for the next creation operation.

Full data structure of the field type

6.6 The methods

The UI information for method and attribute types are the same as the entity and fields respectively.