Modélisation dynamique des modèles physiques etnumériques pour la simulation en électromagnétisme.Application dans un environnement de simulation intégrée :SALOME

Gilles David

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INP GRENOBLE
Spécialité : Génie Electrique

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

présentée et soutenue publiquement

par

Gilles DAVID

le 24 novembre 2006

Modélisation dynamique des modèles physiques et numériques pour la simulation en électromagnétisme. Application dans un environnement de simulation intégrée: SALOME

Directeurs de thèse : Gérard MEUNIER
Thierry CHEVALIER

JURY

Pr.	Patrick	Dular,	Rapporteur
Pr.	Laurent	Nicolas,	Rapporteur
Pr.	Maurizio	Repetto,	Examinateur
M.	Yves	Fricaud,	Examinateur
Pr.	Gérard	Meunier,	Directeur de thèse
MC.	Thierry	Chevalier,	Co-Directeur de thèse

Remerciements

Ce travail a été réalisé au Laboratoire d'Electrotechnique de Grenoble, au sein de l'ex-équipe Modélisation et CAO en électromagnétisme, nouvellement MAGE.

J'adresse mes sincères remerciements à :

Monsieur Patrick DULAR, Professeur à l'Université de Liège, qui m'a fait l'honneur d'être rapporteur,

Monsieur Laurent NICOLAS, Professeur et Directeur du CEGELY, qui m'a fait l'honneur d'être rapporteur,

Monsieur Maurizio REPETTO, Professeur au Politecnico di Torino, qui m'a fait l'honneur d'être membre du jury,

Monsieur Yves FRICAUD, Ingénieur chez OpenCascade, qui m'a fait l'honneur d'être membre du jury,

Monsieur Gérard MEUNIER, Directeur de Recherche au CNRS, qui m'a fait l'honneur d'être membre du jury,

Monsieur Thierry CHEVALIER, Maître de Conférence à l'INPG, qui m'a fait l'honneur d'être membre du jury.

Mes remerciements vont d'abord à ceux qui m'ont permis d'arriver jusqu'ici : je remercie tout d'abord Monsieur Yves BRUNET, Directeur du Laboratoire d'Electrotechnique de Grenoble, pour m'avoir accueilli au sein de ce laboratoire.
Bien sûr, je remercie chaleureusement Gérard MEUNIER pour avoir dirigé mes travaux et apporté ses connaissances dans le domaine de la simulation numérique. Son expérience et sa grande gentillesse ont largement contribués à rendre ces trois années très agréables.
Je tiens aussi à remercier Yves MARECHAL, non seulement pour m'avoir accueilli au sein de l'équipe qu'il dirige, mais aussi et surtout pour m'avoir présenté ses travaux, ce qui m'a conduit à m'engager dans cette thèse.
Enfin, last but not least, un grand merci à James ROUDET et Robert PERRET, dont les encouragements ont contribué à m'engager dans ces longues mais gratifiantes voies que sont la formation d'ingénieur à l'ENSIEG et la formation doctorale à l'INPG.

Ce travail de thèse n'aurait jamais pu atteindre ce résultat sans l'aide précieuse de Thierry avec qui j'ai passé de longs moments de programmation en Python. Merci de m'avoir encadré comme tu l'as fait, d'avoir passé tant de temps à proposer des solutions, et surtout de m'avoir encouragé !

Un grand merci à tous les acteurs du laboratoire qui l'aident à tourner rond : à Stéph' pour ta gentillesse, à Monique, Jacqueline, Danielle et Elise pour leur efficacité à la résolution des tracas administratifs, à Patrick pour son efficacité ...

Pour finir, je remercie du fond du cœur ma petite chérie d'avoir été si gentille, patiente, compréhensive et d'avoir cru en moi. Merci.

Je dédie cette thèse
à Nadège,
à mes parents, ma famille,
et à tous ceux qui comptent pour moi.

Table des matières

Table des figures

Liste des tableaux

Listings

Chapitre 1

Introduction générale

Ce travail de recherche s'inscrit dans le cadre du Réseau National de recherche et d'innovation en Technologies Logicielles (RNTL), mis en place fin 1999, par le ministère de la Recherche et le ministère de l'Industrie. Le RNTL a reçu comme mission de favoriser la constitution de projets innovants de recherche et de développement coopératif entre entreprises et équipes de la recherche publique [1]. Un des domaines soutenu par le RNTL et en plein essor en France est celui de la simulation numérique.

La conception et l'étude, voire l'optimisation, de produits ou dispositifs fortement technologiques passent par l'utilisation de logiciels de simulation numérique. De nombreux logiciels existent dans ce domaine (ex : ANSYS, Femlab, Fluent, ...) et la plupart d'entre eux sont spécialisés dans un type de problème particulier et sont associés à un ou plusieurs domaines physiques donnés. Plusieurs techniques permettent de résoudre les problèmes complexes (ex : les éléments finis, volumes finis, différences finies). Chacune de ces méthodes présente des avantages et des inconvénients en fonction du problème à traiter et du cadre d'utilisation. Par exemple la méthode des éléments finis, par sa souplesse d'emploi et sa grande généralité, est devenue indispensable pour la résolution des problèmes aux dérivées partielles en électromagnétisme et en mécanique.

Face à un problème de conception ou d'optimisation de dispositif physique, la modélisation et la simulation sont donc devenues des étapes quasi indispensables. Jusqu'à aujourd'hui la simulation s'est essentiellement concentrée sur le phénomène physique à l'origine du fonctionnement du dispositif. A présent ce n'est plus suffisant et les éventuels couplages avec d'autres phénomènes physiques annexes modifiant ou perturbant le comportement désiré doivent être pris en compte. Une des raisons principales est que les dispositifs physiques ont gagné en complexité (essentiellement grâce aux progrès de la miniaturisation) mais aussi en performances. Ces progrès obligent alors les concepteurs à prendre en compte la modélisation des phénomènes physiques dont les effets, jusqu'alors négligeables face aux performances de leur dispositif, deviennent désormais parties intégrantes du dispositif global. Dès lors, les outils de calculs numériques utilisés doivent permettre non seulement de modéliser précisément le phénomène physique pour arriver au fonctionnement désiré du dispositif, mais ils doivent aussi traiter des interactions éventuelles avec d'autres phénomènes physiques, c'est-à-dire prendre en compte des comportements fortement multiphysiques.

Comme dit précédemment, à l'origine, les outils de simulation ont été développés pour représenter le phénomène physique majeur qui est l'origine du fonctionnement du dispositif en question ; ils se limitent donc souvent à la physique concernée (magnétique pour les dispositifs électriques, mécanique des fluides pour les turbines, mécanique des structures pour les ouvrages et les pièces mécaniques, neutronique pour les cœurs de réacteur nucléaire, ...). La diversité des physiques ainsi exploitées a entraîné la diversité des méthodes utilisées, de part les spécificités intrinsèques à chaque physique. Parmi ces méthodes, certaines peuvent toutefois concerner de nombreuses physiques et sont largement diffusées. Citons par exemple la méthode des éléments finis utilisée en thermique, mécanique des structures, neutronique ou en électromagnétisme, ou celle des volumes finis utilisée en mécanique des fluides. Dès lors on voit apparaître des tentatives de couplage multiphysiques et multi-méthodes.

Les besoins multiphysiques évoqués plus haut ont conduit dans un premier temps

les concepteurs de logiciels à faire évoluer les codes de calculs existants pour représenter au mieux l'influence de ces phénomènes *parasites*. La complexité des codes est devenue telle qu'il devient d'après nous indispensable de considérer les physiques secondaires en tant que telles. Pour cela il faut être capable de réaliser des couplages multiphysiques performants pour une grande variété de physiques. Pour réaliser ces couplages, on peut être ammené à choisir entre une approche globale ou modulaire, selon le problème considéré.

Dans le cas de l'approche globale, le code de calcul utilisé est basé sur la représentation de l'ensemble des phénomènes physiques ainsi que de leurs couplages. On peut voir un code basé sur cette approche comme une boîte noire modélisant le dispositif concerné dans son ensemble, c'est-à-dire aussi bien le phénomène physique principal à l'origine de ce dispositif que les autres phénomènes physiques, non désirés, et leurs influences sur ce phénomène principal. Pour ce faire, une technique possible est de traiter les couplages d'un seul bloc en ne résolvant qu'un seul jeu d'équations regroupant les inconnues des différentes physiques entrant en jeu. L'intérêt de cette technique est de mieux prendre en compte les liens couplant les physiques directement au niveau des équations. En effet les termes de couplages apparaissent clairement dans la matrice à résoudre et les inconnues des physiques sont calculées en même temps. Un tel outil peut rapidement devenir très compliqué, tant en terme d'évolutivité que de maintenance. Par exemple, la modification de la modélisation d'une physique pour mieux la représenter peut amener à un grand nombre de modifications dans le code de calcul au niveau des couplages avec les autres physiques, voire de leur modélisation. L'ajout d'un phénomène supplémentaire peut se révéler également très compliqué car il s'effectue au cœur du code de calcul.

L'approche modulaire se base sur la séparation des descriptions des phénomènes physiques, de leurs résolutions et de leurs couplages. Elle peut par exemple être adaptée pour des problèmes où les échelles de temps ou de taille entre les physiques sont très différentes. La résolution du problème global passe alors nécessairement par un composant de supervision permettant d'effectuer des itérations entre les différents modules, mais surtout d'assurer une cohérence entre les données de ces codes. Pour ce faire, une technique possible est de traiter les physiques les unes après les autres, les solutions de l'une devenant des sources pour l'autre. Cette technique offre plus de souplesse sur les méthodes de résolutions des équations pour chacune des physiques concernées. Dans le cas des codes de calculs basés sur les éléments finis par exemple, chacune des physiques peut être traitée par un solveur dédié, avec un maillage et des paramètres de simulation adaptés. La modélisation du dispositif dans son ensemble passe alors par un enchaînement de simulations de chacune des physiques, les couplages étant assurés par un échange de données entre les codes de calcul concernés. La complexité réside alors dans l'interopérabilité des codes de calcul utilisés : beaucoup ne sont pas conçus pour échanger des données vers l'extérieur, mais si cela est possible, encore faut-il s'assurer de la compatibilité des données échangées.

La différence entre ces approches est à ne pas confondre avec celle entre couplage fort et couplage faible. Dans le dernier cas, les hypothèses faîtes rendent partiellement indépendantes les physiques et facilitent ainsi le couplage. Nous préférons faire la distinction entre résolutions globale et modulaire. En effet rien n'empêche de résoudre un problème multiphysiques avec l'une ou l'autre des approches, tout en gardant un couplage fort ; dans le cas de la méthode modulaire, des itérations successives entre les solveurs, et donc des échanges fréquents de données, seront alors nécessaires afin d'assurer le couplage

fort souhaité. Un avantage de la méthode modulaire est de pouvoir choisir un solveur spécialisé pour chaque physique, les méthodes de résolution de chacun des solveurs étant indépendantes les unes des autres. Il devient alors possible de bénéficier des savoir-faire de chacun des spécialistes. La mise au point d'un dispositif physique revient ainsi à la résolution d'un problème multiphysiques, multi-solveurs, voire multi-méthodes.

Les deux approches sont intéressantes. En fonction du problème traité, l'une ou l'autre présente des avantages mais la 2^{e} paraît incontournable : elle offre plus de possibilités de souplesse, d'évolutivité, de flexibilité, et elle présente une grande capacité de prototypage. La 1^{ière} permet en général de donner des résultats avec des temps de calcul réduits mais, à l'inverse, ne permet pas de facilement réaliser de nombreux prototypage.

De plus en plus de logiciels offrent la possibilité de réaliser certains couplages (par exemple : électromagnétisme/thermique), mais le choix des physiques à coupler reste encore réduit. On peut en effet comprendre qu'il soit difficile de réaliser, dans un logiciel spécialisé, des couplages avec des physiques qui ne soient pas dans le domaine de compétence du logiciel en question. En effet, développer un couplage coûte cher en terme de temps de programmation, et les investissements requis sont très importants.

Le travail de recherche présenté dans ce mémoire propose d'apporter des solutions au problème de couplage de différentes physiques en aidant et simplifiant le travail des développeurs pour la réalisation de codes de calculs, en particulier ceux dédiés à la résolution de problèmes multiphysiques. Pour cela, des démarches, méthodes et outils sont proposés pour permettre de réaliser à moindre coût des outils de calculs, multiphysiques ou monophysiques. L'objet est bien d'être capable de capitaliser au maximum (utilisation de logiciels existants) et d'explorer les nombreuses possibilités de couplage, le nombre de combinaisons possibles étant très important, avec un coût qui reste accessible. Un certain nombre d'éléments peuvent permettre de largement faciliter le travail des développeurs : par exemple la séparation des aspects de description des données (structure de données) et de leur traitement (supervision) est une voie à explorer.

Nous proposons de suivre les étapes suivantes pour exposer notre travail. Nous commencerons tout d'abord par un cas d'étude *significatif* de la problématique que nous aborderons. Il s'agit de la simulation du comportement magnéto-thermique d'un ruban supraconducteur. Dans ce cas le phénomène principal est magnétique mais il peut être perturbé par la température du système, elle-même éventuellement induite par les phénomènes électromagnétiques. Nous exposerons la démarche classique que l'on est amené à suivre lors de la mise en place de solution pour des problèmes de ce type. Ensuite nous nous appuierons sur cet exemple pour illustrer les éléments caractéristiques de la réalisation de ce type de couplage, ce qui permettra d'introduire les premiers concepts de notre approche.

Dans une deuxième partie, nous présenterons en détail les concepts que nous proposons et le cadre pour leur mise en œuvre. Nous détaillerons dans cette partie la mise en place d'un métalangage et les intérêts de cette démarche. A l'issue de cette partie l'ensemble des concepts que nous proposons auront été introduits et les outils à mettre en place autour de ces concepts seront définis.

La partie suivante sera consacrée aux choix technologiques et à la mise en œuvre des concepts exposés précédemment. Nous expliquerons en détail comment nous avons obtenu le comportement souhaité pour les modèles de données décrits par le métalangage

grâce à une utilisation intensive des concepts de métaprogrammation. La mise en œuvre des outils et services autour du métalangage sera aussi expliquée en détails.

Enfin toutes ces réalisations seront présentées et mises en œuvre dans une plate-forme open-source de simulation numérique. L'objectif est de conduire l'application des concepts jusqu'à la réalisation d'un environnement complet et utilisable de simulations multi ou mono-physiques. Nous présenterons deux exemples ayant conduit à l'intégration de solveurs dans cette plate-forme : un cas d'école de problème magnétostatique simple, et un problème d'interactions fluide-structure.

Chapitre 2

Cas d'étude : simulation d'un ruban supraconducteur HTC

Sommaire

2.1 Introduction

La méthodologie d'unification des modèles de données physiques présentée dans ce mémoire est un travail théorique orienté génie logiciel. Pour plus de clarté et afin d'améliorer la compréhension du lecteur, un exemple concret illustrera les notions développées. Cet exemple rentre dans les domaines d'application à la fois du génie électrique, génie thermique et des matériaux supraconducteurs. Différents modèles issus de plusieurs domaines physiques seront ainsi traités, ainsi que leurs éventuels couplages.

Ce chapitre traite de la résolution d'un problème couplé magnétothermique dans un ruban supraconducteur à haute température critique (SHTC). Cette résolution doit prendre en compte l'effet de la température sur les propriétés physiques du SHTC, mais doit aussi tenir compte des pertes par conduction (AC). La méthode de résolution est basée sur la séparation des modèles de données physiques, chacun étant associé à son propre solveur, et sur un composant logiciel de supervision. Ce dernier est chargé de l'enchaînement des résolutions des solveurs et de la transmission de données entre solveurs, afin de résoudre le problème global.

Après une description du problème étudié dans la partie 2.2, les équations à résoudre seront rappelées dans les parties 2.3.1 et 2.3.2. La méthode de couplage de ces équations afin de résoudre le problème global fera l'objet de la partie 2.4. Les résultats de simulation sur une géométrie simple font l'objet de la partie 2.5. Finalement, dans la dernière partie 2.6, nous analyserons la démarche de modélisation et nous montrerons comment elle peut s'appliquer pour divers problèmes (magnétique, thermique, magnétothermique, ...). Nous présenterons le problème de la généricité de cette démarche de modélisation, et nous introduirons alors la solution que nous proposons pour répondre à ce problème, à savoir l'utilisation d'un formalisme unique dédié à la description des modèles de données physiques.

2.2 Description du problème

Le domaine d'application des SHTC est potentiellement extrêmement important, des machines électriques à la fusion thermonucléaire contrôlée en passant par les câbles

d'énergie. Une modélisation numérique est indispensable pour dimensionner correctement et optimiser les dispositifs supraconducteurs. Des outils ont été développés mais ils considèrent en général un fonctionnement isotherme. A cause des pertes AC dans un supra, les conditions ne sont pas isothermes, d'où la nécessité de développer un modèle électrothermique. L'étude de la transition, ou perte de l'état supraconducteur (SC), qui est à la base du limiteur de courant de défaut, exige un traitement électrothermique également.

La circulation d'un courant (souvent sinusoïdal) $i(t)$ dans un solénoïde supraconducteur conduit à l'apparition d'une induction magnétique $B(t)$. Le problème consiste à calculer la répartition de la densité de courant J dans le matériau en fonction du temps. Dans notre cas, la résolution seule de l'équation du champ électromagnétique ne suffit pas. En effet cette équation fait intervenir la loi de comportement du matériau $\sigma = J(E)$ dont l'évolution dépend de la température. Or, l'induction magnétique $B(t)$ dans le matériau est la source de pertes par conduction [2]. Ces pertes nous amènent à tenir compte de l'évolution de la température qui ne restera pas uniforme. Pour connaître la répartition de $J(t)$, il s'agit alors de résoudre les équations du champ électromagnétique et de la chaleur, couplées entre elles au travers de la température T et des pertes p.

2.3 Les modèles physiques

2.3.1 Le modèle magnétique

Le modèle magnétique permet de calculer la densité de courant J à partir de l'alimentation électrique (courant ou tension) de la bobine, en tenant compte de la dépendance de la loi de comportement $J(E)$ du matériau envers la température. Il s'agit de résoudre l'équation du champ électromagnétique :

$$\overrightarrow{rot}\frac{1}{\mu_0}\overrightarrow{rot}\overrightarrow{A} + \sigma(E)\frac{\partial\overrightarrow{A}}{\partial t} = -\sigma(E)\overrightarrow{grad}\,V \tag{2.1}$$

La loi de comportement du matériau définit la relation entre champ électrique et densité de courant et peut s'exprimer selon le modèle en puissance [2, 3] (figure 2.1) :

$$J = \sigma(E)\cdot E = \frac{J_c}{E_c}\left(\frac{|E|}{E_c}\right)^{\frac{1}{n}-1} E \tag{2.2}$$

J_c est la densité de courant critique, atteinte pour $E = E_c$ (habituellement fixé à $100\mu V/m$).Le paramètre n définit la pente de la partie linéaire. Si on considère un comportement isotherme (figure 2.1), J_c est constante et égale à J_{c0} (densité de courant pour $T = T_0$, température initiale du solénoïde).

2.3.2 Le modèle thermique

Ce modèle permet de calculer la température T dans le matériau à partir des pertes AC p, considérées alors comme sources de chaleur. Pour cela, l'équation de la chaleur avec sources doit être résolue :

$$\rho C_p\frac{\partial T}{\partial t} - div(\lambda\cdot gradT) = p \tag{2.3}$$

ρ représente la masse volumique, C_p la chaleur spécifique, λ le coefficient de conduction, $T(r,z,t)$ la température et p les sources de chaleur.

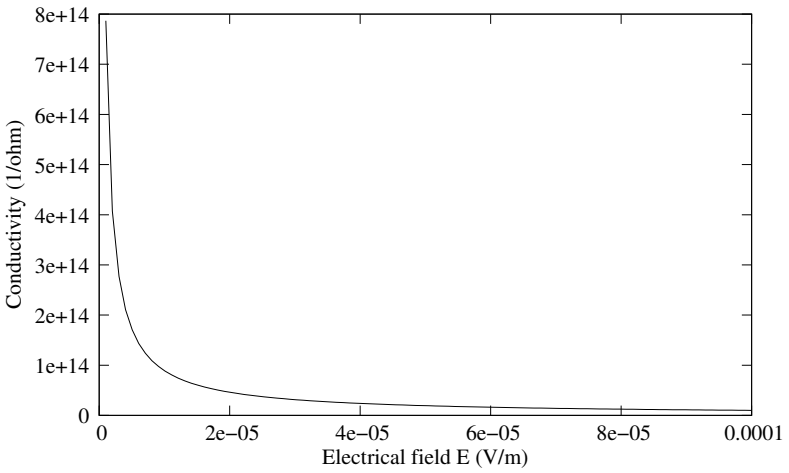


FIG. 2.1 – Loi de comportement $\sigma(E)$ du matériau SC (T=77K,$E_c = 100\mu V/m$,$J_{c0} = 10^9 A/m^2$, n=20)

2.4 Le couplage magnéto-thermique

2.4.1 Choix de la méthode

Pour résoudre ce problème en gardant un couplage fort entre les physiques, deux solutions s'offrent à nous :

- *la méthode globale :* résoudre en un bloc une seule équation regroupant les inconnues magnétique et thermique,
- *la méthode modulaire :* résoudre les équations magnétique et thermique de manière itérative et en satisfaisant aux relations de couplage.

Les constantes de temps mises en jeu par les phénomènes électriques et thermiques ont des ordres de grandeur différents. Dès lors, plutôt que de résoudre les équations magnétothermiques directement, on préfèrera la deuxième solution. De plus, cette méthode apporte la souplesse de pouvoir choisir des pas de temps différents pour chacun des problèmes, et offre la possibilité d'utiliser deux solveurs différents adaptés à chaque physique. Eventuellement, les maillages peuvent aussi être différents.

Concernant le modèle magnétique, la loi de comportement donnée par l'équation (2.2) considère une température uniforme dans le matériau. Or, nous considérons que la circulation du courant dans le matériau entraîne des pertes AC, causant ainsi un échauffement. Les variations locales de température influent alors sur les caractéristiques du matériau et la loi de comportement s'écrit alors ainsi :

$$J = \sigma(E) \cdot E = \frac{J_c}{E_c} \left(\frac{|E|}{E_c} \right)^{\frac{1}{n}-1} E \quad \text{avec} \quad J_c = J_{c_0} \frac{1 - \frac{T}{T_c}}{1 - \frac{T_0}{T_c}} \tag{2.4}$$

T_c est la température critique du supraconducteur. Ce modèle ne considère pas le régime au-delà de T_c. En effet, pour des températures supérieures, le matériau n'a plus le comportement supraconducteur. De la même manière, la relation entre J et E n'est valable que dans un intervalle limité dont on sort quand la température se rapproche de T_c.

En ce qui concerne le modèle thermique, nous considérons que les pertes AC peuvent être calculées à partir du champ électrique et de la densité de courant : $p = EJ$.

2.4.2 Principe de fonctionnement

Pour résoudre chacune des équations, on utilise une méthode éléments finis (EF) associée à un solveur. Chaque équation peut être résolue par un solveur spécifique adapté à son problème. Pour résoudre l'ensemble du problème, il faut être capable de piloter les solveurs EF et d'échanger des informations entre ceux-ci à tout instant. Pour cela, on réalise un composant logiciel (le superviseur) qui va piloter les solveurs EF et superviser l'ensemble de la résolution. Le superviseur présente le schéma de fonctionnement de la figure 2.2.

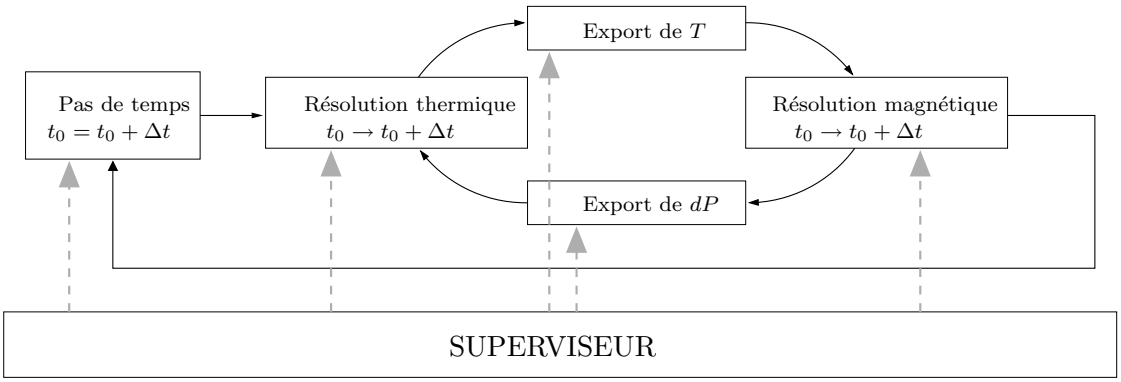


FIG. 2.2 – Schéma fonctionnel du composant de supervision

Il pilote les différentes étapes de la résolution : choix du pas de temps [1], import/export de données et séquencement des résolutions. Les résolutions sont effectuées par deux solveurs : un thermique pour le problème thermique et un magnétique pour le problème électromagnétique. Ces deux solveurs sont basés sur le moteur de résolution de Flux [4].

Le superviseur pilotant l'ensemble du séquencement, il peut décider de l'arrêt ou pas de la résolution, et également de post-traiter en temps réel les résultats de simulation (intéressant si la simulation est longue pour le suivi de résultats).

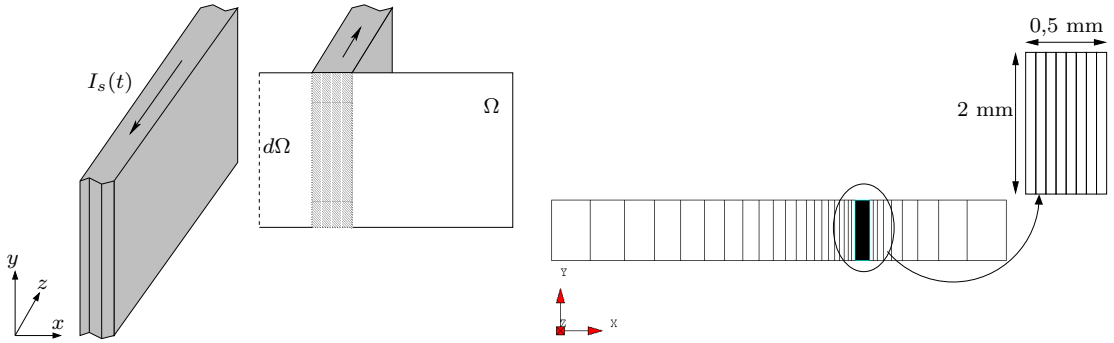
2.5 Résultats de simulation

2.5.1 Cas test

2.5.1.1 Géométrie et maillage

Le problème considéré est celui de deux plaques SHTC dont la géométrie et le maillage sont représentés sur la figure 2.3. Une plaque fait $0,5$ mm d'épaisseur. Elle est considérée infinie selon les axes y et z. Ainsi le champ est uniforme sur la hauteur de la plaque. Nous choisissons de résoudre le problème sur le domaine Ω (2 mm de hauteur) avec les conditions aux limites adéquates.

[1] Dans ce cas précis, l'évolution de la température était telle qu'il n'était pas nécessaire de choisir des pas de temps différents pour les solveurs thermique et magnétique.



(a) Géométrie (b) Maillage du domaine Ω

FIG. 2.3 – Géométrie et maillage du problème

2.5.1.2 Conditions aux limites

Du point de vue du problème magnétique, nous savons que le champ magnétique est tangent sur la frontière gauche $d\Omega$ (en pointillée). Afin de refléter la symétrie existante, nous appliquons une condition de type Dirichlet ($A = 0$) sur $d\Omega$. Par ailleurs, étant donné les dimensions de notre problème, la composante tangentielle du champ est considérée nulle sur le reste de la frontière du domaine Ω : nous appliquons alors une condition de type Neumann homogène ($\frac{dA}{dn} = 0$ ou $B_t = 0$).

Du point de vue du problème thermique, nous considérons que la frontière du supraconducteur est adiabatique pour notre domaine d'étude. En effet, le temps d'étude étant rapide, il y a peu de phénomène de convection au début de la simulation.

2.5.1.3 Données physiques

Une source de courant $I_s(t) = 500 \cdot \sin(2\pi ft)$ avec $f = 50$ Hz est appliquée sur le supraconducteur. Son sens de circulation est indiqué sur la figure 2.3(a). Le supraconducteur étudié présente une densité de courant critique initiale J_{c_0} de $10^9 \mathrm{A/m^2}$. La surface du domaine Ω étant $S = 0,5 \times 2 = 1$ mm^2, le courant maximal admissible dans le ruban est $I_{max} = 1000$ A. Le maximum du courant source étant de 500 A, dans le cas idéal le courant doit se répartir dans la première moitié du ruban, où il prendra les valeurs de $+S \times J_c$ quand $dIs/dt > 0$ et $-S \times J_c$ quand $dIs/dt < 0$.

En réalité, la relation $E(J)$ du matériau ne suit pas une loi parfaite : théoriquement la densité de courant est égale à $\pm\, J_c$, mais dans la pratique, elle peut dépasser ces valeurs, ce que reflète l'indice n dans l'équation (2.2). La figure 2.4 montre deux courbes $E(J)$ pour $n = 10$ et pour $n = \infty$ (le cas idéal). Le choix de la valeur de n se fait d'après les mesures expérimentales réalisées sur le matériau.

La température critique du supraconducteur est $T_c = 80$ K. Le bain dans lequel le ruban est plongé est à une température $T_0 = 77$ K. L'évolution de la température dans le ruban dépend des paramètres λ et C_p du matériau. Pour le supraconducteur étudié, elles sont les suivantes : $\lambda = 500\ W/m/K$ et $C_p = 2.10^6\ J/m^3/K$. Toutefois, avec une telle valeur de C_p, le matériau met beaucoup de temps pour chauffer. Pour éviter d'avoir des temps de simulation long, nous avons ramené cette constante à une valeur de 4000 $J/m^3/K$. Il faudra alors faire attention aux interprétations physiques suite à la modification de cette valeur.

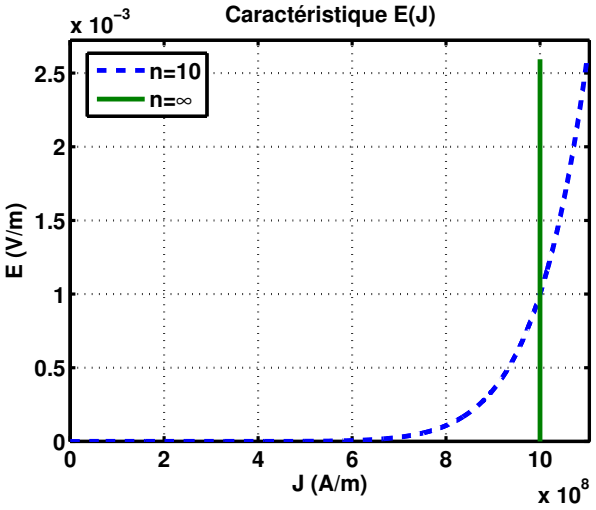


FIG. 2.4 – Tracés de la loi $E(J)$ pour $n = 10$ et $n = \infty$ (cas théorique)

2.5.1.4 Résolution du problème non couplé

Il s'agit maitenant de valider les simulations, mais aussi d'avoir une référence de comparaison pour les autres simulations. Nous résolvons le problème dans le cas où la dépendance de J_c avec la température n'est pas prise en compte : $J_c = J_{c_0}$. Les pertes magnétiques sont quant à elles toujours prises en compte comme sources de chaleur. La figure 2.5 présente l'évolution de la répartition de la densité de courant sur une demi-période.

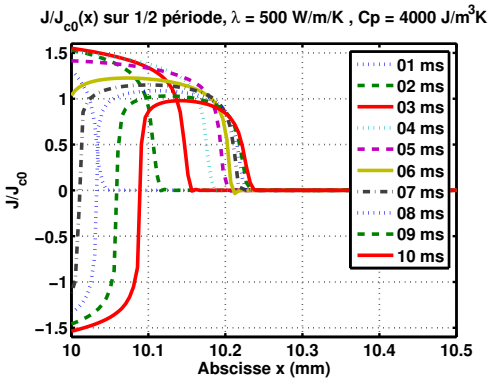


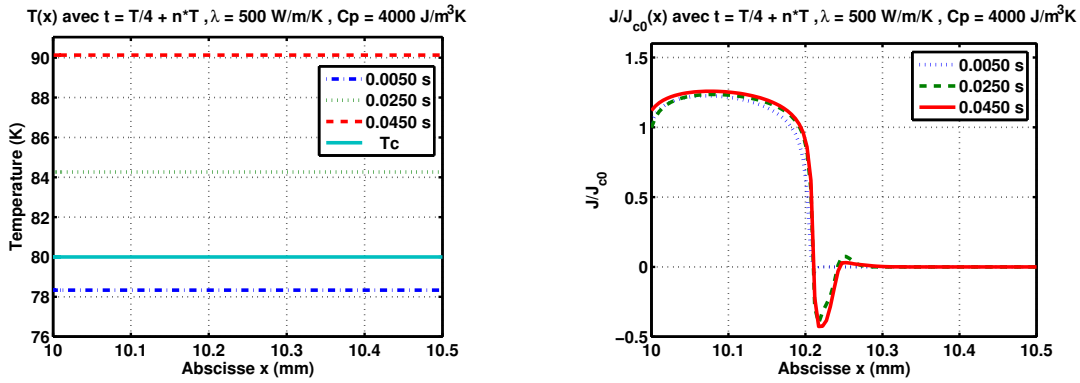
FIG. 2.5 – Evolution de J/J_{c_0} pendant une demi-période avec $J_c = J_{c_0}$

Nous avons le comportement normal d'un supraconducteur : le courant pénètre par l'intérieur de la plaque. Théoriquement, $J = J_c$ quand le courant source I_s passe de 0 à Imax, puis $J = -J_c$ quand I_s passe de Imax à 0. Cependant, nous observons que J dépasse J_c, conformément à ce qu'indique la figure 2.4. L'indice de transition résistive n considéré dans ces simulations est de 30. Ce dépassement au-delà de J_c conduit naturellement à ce que la densité de courant se répartisse sur moins de la moitié de la plaque quand le courant est maximum ($I_{max} = I_c/2$).

Par la suite, nous tracerons uniquement la répartition de J aux instants où I_s est

maximum, c'est-à-dire $t = \frac{T}{4} + nT$. Nous pourrons alors comparer l'évolution de cette répartition après plusieurs périodes de I_s.

La figure 2.6 montre les répartitions de J et de T dans la largeur du ruban pour les instants correspondant au maximum de courant source. J_c étant indépendant de T, J se répartit comme prévu de la même manière pour tous ces instants, sur moins de la moitié du ruban et avec un dépassement au-delà de J_c puisque $n = 30$. (figure 2.6(b)).



(a) Répartition de T dans la largeur du ruban

(b) Répartition de J/J_{c_0} dans la largeur du ruban

FIG. 2.6 – Répartitions de T et J/J_{c0} avec J_c indépendant de T

L'uniformité de la température à travers la plaque (figure 2.6(a)) est due à la diffusivité importante et à la faible épaisseur (La constante de temps $\tau = a^2 \frac{\lambda}{C_p} = 2\mu s$). On note cependant que sa valeur augmente dans le temps, allant même jusqu'à dépasser la température critique T_c du supraconducteur. Ceci étant impossible sans que le matériau passe à l'état non supra, nous en déduisons que cet échauffement, causé par les pertes AC, justifie pleinement la dépendance de J_c avec la température.

2.5.2 Validation du couplage et mise en évidence de son effet

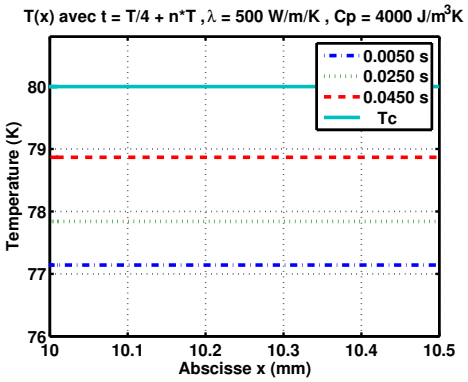
A présent, nous allons réaliser le couplage magnéto-thermique avec $\lambda = 500 W/m/K$ et $C_p = 4000 J/m^3/K$. La dépendance de J_c avec la température est alors prise en compte selon l'équation suivante :

$$J_c = J_{c_0} \frac{1 - \frac{T}{T_c}}{1 - \frac{T_0}{T_c}} \tag{2.5}$$

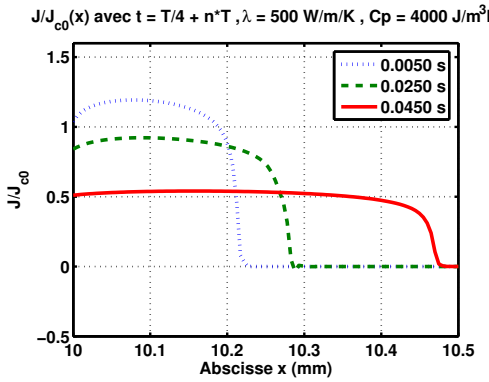
Nous représentons J/J_{c_0} et T pour t = 5, 25 et 45 ms, instants correspondants aux maxima du courant source Is (figure 2.7). La figure 2.7(b) montre que la valeur de la densité de courant diminue quand la température augmente (figure 2.7(a)) puisque J_c diminue. En conséquence, la répartition de J s'étale plus dans la largeur de la plaque. En effet, pour un même courant source I_s et avec une densité de courant critique J_c plus petite, il faut une section plus grande pour continuer à faire passer ce courant.

2.5.3 Etude de l'influence de la conductivité thermique λ

Après avoir mis en évidence l'effet de la température sur la répartition de J dans la plaque, nous allons nous intéresser à l'influence de la non-uniformité de la température. Avec une conductivité thermique $\lambda = 500\ W/m/K$ et une chaleur spécifique $C_p =$



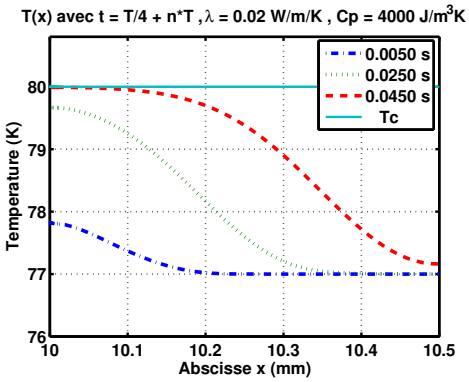
(a) Répartition de T dans la largeur du ruban



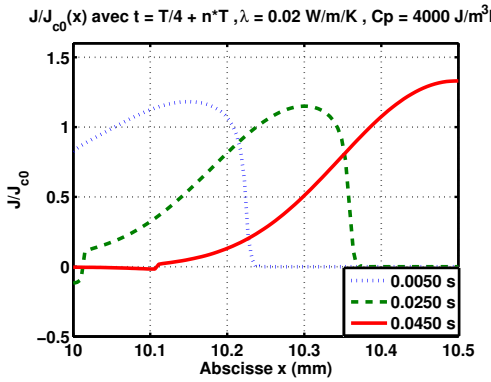
(b) Répartition de J/J_{c0} dans la largeur du ruban

FIG. 2.7 – Mise en évidence de l'effet du couplage

4000 $J/m^3/K$, la constante de temps du système thermique $\tau_{th} = a^2 \frac{\lambda}{C_p}$ vaut 2 μs : la température se répartit instantanément de manière uniforme dans la largeur du ruban. Afin de simuler une non-uniformité de cette répartition, nous modifions le paramètre λ à une valeur de $0,02$ $W/m/K$. La constante de temps du système thermique τ_{th} passe ainsi de 2 μs à 50 ms. La figure 2.8(a) nous montre en effet que la température diffuse beaucoup plus lentement que dans le cas précédent.



(a) Répartition de T dans la largeur du ruban



(b) Répartition de J/J_{c_0} dans la largeur du ruban

FIG. 2.8 – Influence de la conductivité thermique λ

En parallèle, la figure 2.8(b) montre bien que la densité de courant diminue dans la zone où la température est plus élevée : à $t = 45$ ms, la température atteint la température critique de 80 K sur près de $0,1$ mm de large ; la densité de courant est nulle au même endroit. Le courant se répartit ainsi plus profondément que prévu dans le ruban, jusqu'à atteindre le bord droit, avec des valeurs de J très importantes, alors que sur le bord gauche, plus aucun courant ne circule. Si on continue, la température continue d'augmenter, amenant alors tout le ruban HTC vers sa température critique, et donc hors de son état supraconducteur. Le modèle utilisé ne permettant pas de prendre en compte le passage état supra $\rightarrow$ état normal, nos simulations s'arrêtent avant ce moment.

2.6 Analyse de la démarche de modélisation

Nous venons de résoudre un problème multiphysique magnéto-thermique en suivant des étapes de modélisation et de résolution précises : la géométrie du problème a été décrite puis maillée, les différentes propriétés physiques requises par les solveurs ont été données, les équations ont été résolues selon des méthodes adaptées et enfin les résultats de calcul ont été exploités.

Cette démarche peut facilement être programmée dans un logiciel de simulation dédié à la résolution de problèmes magnétothermiques. L'inconvénient de cet outil est qu'il ne pourra pas traiter d'autre problème. Par exemple si nous considérons le cas d'un problème électromagnétique simple, le principe serait le même mais la démarche serait différente : les propriétés physiques et le processus de résolution ne sont plus les mêmes. Dans ce cas, nous devrions programmer un autre logiciel de simulation dédié à la résolution de problèmes électromagnétiques.

Nous pouvons ainsi programmer autant de logiciels de simulation qu'il n'y a de modèles de physiques. Dès lors, il suffirait de choisir le logiciel correspondant à notre problème pour pouvoir le résoudre. Cependant, si ce logiciel n'est pas disponible, il faudrait alors réécrire un logiciel adapté. Nous constatons que cette démarche n'est pas adaptée dans le cas où l'on souhaite traiter des cas très variés. Elle impose un coût important, notament en terme de développements et de maintenance de code.

Nous proposons de nous inscrire dans une autre démarche qui consiste en un descripteur général de problème numérique dans lequel toutes les étapes de modélisation, résolution et exploitation sont traitées :

- description de la géométrie,
- maillage,
- description des propriétés physiques,
- résolution des équations,
- exploitation des résultats.

Cet outil est une plateforme logicielle qui ne propose pas de résoudre toutes les équations possibles de la physique, mais un environnement dans lequel il est possible de décrire complètement notre problème pour ensuite le résoudre avec le ou les solveurs appropriés. En effet parmi les étapes de modélisation, nous constatons que la description de la géométrie et le maillage sont des processus qui peuvent être réalisés indépendamment du problème à résoudre [2]. Vient ensuite la description de l'ensemble des données du problème à traiter. Cette partie est directement dépendante du type de problème que l'on souhaite traiter et des hypothèses que l'on choisi. C'est à cette partie que nous nous attachons en particulier car elle conditionne le reste de la chaîne. Ensuite vient la résolution qui doit pouvoir exploiter l'ensemble des données évoquées précédemment et qui est aussi dépendante des choix réalisés. Cette partie sera traitée du point de vue du modèle de données mais pas du reste (méthodes de calcul, formulation, ...). Finalement la dernière étape est le traitement des résultats, partie que nous ne traiterons pas ou peu, faute de temps.

Le problème magnéto-thermique que nous venons de résoudre synthétise de nombreux aspects de modélisation de phénomènes physiques. Tout d'abord, le procédé physique étudié faisait intervenir deux physiques couplées entre elles. Ce couplage a été

[2] Précisons toutefois que la phase de maillage peut être délicate, par exemple dans le cas de régions à faibles épaisseurs, et qu'elle nécessite alors des compétences liées au problème à résoudre.

modélisé d'un côté par la dépendance de la densité de courant J_c avec la température pour la partie magnétique, et d'un autre côté par la prise en compte des pertes de chaleur $p = EJ$ comme termes sources pour la partie thermique. Ensuite, ces choix de modélisation ont nécessité d'échanger des données entre les deux solveurs. Ce processus d'échange de données entre solveurs n'est pas trivial : en effet un solveur possède généralement sa propre structure de données interne, et des solveurs différents ont peu de chance d'avoir des structures compatibles entre elles.

A partir ce ce constat, les questions qui se posent sont les suivantes : comment avoir un outil générique permettant de décrire n'importe quel phénomène physique, indépendamment du solveur qui sera par la suite utilisé pour résoudre notre problème ? Comment faire en sorte que le solveur récupère les informations qui le concernent ? Comment faire pour que l'ajout d'une physique soit réduit en coût [5] ?

Une réponse possible à ces questions est de partir sur le principe que pour pouvoir plus facilement coupler des physiques, et donc leurs modèles associés, les développeurs de code de calcul doivent pouvoir s'appuyer sur une base commune, quels que soient les domaines physiques impliqués. Cette base commune servirait ainsi de support pour pouvoir échanger des données de manière unifiée.

La première idée qui peut venir à l'esprit lorsque l'on parle de base commune est de rassembler le plus de concepts communs à toutes les physiques dans un modèle unique, universel. On sait en effet que beaucoup de phénomènes physiques suivent des comportements modélisés par les mêmes familles d'équations mathématiques (elliptiques, paraboliques, aux dérivées partielles, ...). Il suffirait alors ensuite de dériver ce modèle universel vers la physique de notre choix en y ajoutant ses spécificités pour obtenir son modèle. Néanmoins, même si le maximum de concepts est rassemblé dans un modèle unique, il restera toujours des domaines physiques pour lesquels ce modèle ne conviendra pas comme base de modélisation. De plus, réaliser ce modèle universel pourrait revenir en fait à écrire une liste exhaustive d'équations mathématiques afin de couvrir le maximum de modèles possibles.

Nous proposons alors de réaliser un formalisme unique dont le rôle est non pas de décrire l'ensemble des propriétés physiques de toutes les physiques, mais de décrire les modèles de données, l'organisation de ces propriétés physiques. Ce formalisme se place ainsi à un niveau supérieur de modélisation et se comporte comme un modèle de modèles : c'est un métamodèle. Les modèles sont étroitement liés et dépendants des solveurs, mais le formalisme permettant de les écrire reste indépendant du problème, que celui-ci soit multiphysique ou monophysique. Des services autour de ce formalisme doivent alors être proposés afin de pouvoir traiter non seulement les modèles, mais aussi les données qui sont créées à partir de ces modèles. Ces services doivent aussi permettre aux solveurs de récupérer leurs données physiques respectives, et de se les échanger entre eux.

Le formalisme que nous proposons est réalisé sous la forme d'un langage de description des propriétés physiques : le SPML (Standard Physic Modeling Language). Les modèles de données physiques seront ainsi écrits avec le même langage, ce qui permet alors d'utiliser un outil générique pour les analyser. Ainsi n'importe quel modèle de données écrit avec ce langage pourra être interprété et utilisé pour rentrer les propriétés physiques avec le même logiciel.

2.7 Conclusion

Ce chapitre a présenté la modélisation, la résolution et l'exploitation des résultats d'un problème multiphysique magnéto-thermique appliqué à un matériau supraconducteur. Nous avons proposé une méthode qui consiste à séparer les modèles des physiques et à réaliser un couplage externalisé entre ces modèles. Les équations des modèles ont été résolues par leur solveur respectif. La résolution globale du problème a nécessité l'utilisation d'un composant de supervision permettant d'enchaîner les résolutions spécifiques et de gérer les échanges de données. Nous somme ainsi arrivé à des résultats montrant une réelle influence de la température sur la loi de comportement du matériau supraconducteur.

Après une analyse de la démarche de modélisation, nous avons remarqué que pour de nombreux problèmes de simulations numériques cette démarche était la même :
- description de la géométrie,
- description des propriétés physiques,
- maillage,
- résolution des équations,
- exploitation des résultats.

Nous avons été contributeurs d'une plate-forme qui propose de factoriser un maximum d'étapes de ce processus au sein d'un environnement unique, permettant ainsi de traiter de nombreux problèmes avec un même logiciel. Cette plateforme a été concrétisée avec le projet SALOME [6] et rassemble les étapes de géométrie, physique, maillage, résolution et résultats.

Pour permettre la description des propriétés physiques d'un problème indépendamment de la physique concernée, nous avons développé un formalisme unique, un métamodèle, permettant de décrire des modèles de données physiques et d'utiliser ensuite ces modèles pour décrire les propriété physiques et les transmettre vers le solveur associé. Ce formalisme a été concrétisé avec un langage de description, le SPML, ainsi qu'avec toute une panoplie de services autour permettant de gérer les modèles et les données. L'organisation de ce langage et ses fonctionnalités associées sont présentées dans le chapitre 3, tandis que sa mise en œuvre dans les détails est traitée dans le chapitre 4.

Chapitre 3

Méthodologie proposée

Sommaire

3.1 Introduction

Le chapitre 2 a présenté un exemple de résolution de problème numérique. Cet exemple a permit d'illustrer les différentes étapes suivies lors d'une démarche de modélisation et de résolution d'un tel problème. Après analyse de cette démarche, nous avons proposé l'utilisation d'un formalisme unique pour décrire les modèles de données physiques. Ce formalisme a été réalisé sous la forme d'un langage de description des propriétés physiques : le *Standard Physic Modeling Language* ou *SPML*.

Le *SPML* fait partie de la famille des langages de programmation orientés objets. Le chapitre 3.2 explique ce choix et pose le cahier des charges de ce langage. Autour de celui-ci, une liste de fonctionnalités et de services, allant de la vérification de la cohérence des modèles, jusqu'au traitement des données physiques, doit aussi être proposés. Les services que nous proposons sont expliquées dans le chapitre 3.3.

Pour pouvoir comprendre l'architecture et l'utilisation du SPML, le chapitre 3.4 sera consacré à l'explication des concepts et de l'organisation générale d'un langage de programmation orienté objet, avec comme base de travail le langage Python. Ensuite, le chapitre 3.5 expliquera en détail l'architecture et l'organisation du langage SPML et ses spécificités.

Enfin le chapitre 3.6 illustrera les concepts des chapitres précédents avec l'écriture en SPML d'un modèle de données d'un simple problème thermique.

3.2 Le langage SPML : une réalisation du formalisme unique

Comme nous l'avons évoqué précedemment, nous proposons de mettre en place un formalisme qui va permettre de décrire les modèles physiques que les développeurs utilisent pour réaliser leur logiciel de simulation numérique. Il existe en fait quasiment autant de modèles que de logiciels qui permettent de faire de la simluation numérique. Même si pour un même domaine physique (exemple : l'électromagnétisme), on pourrait s'attendre à trouver des choses communes entre deux logiciels il n'en est rien car les logiques de développement et les contraintes technologiques conduisent généralement à des choix très différents. Dans ce cadre, on comprend aisément que l'échange des données (et donc le couplage) entre différents outils sera très difficile. Cela le sera d'autant plus que les physiques sont différentes et que les habitutes des différentes communautés sont distinctes.

Tout cela nous conduit donc à proposer une démarche qui consiste à mutualiser non pas les structures (modèles) de données utilisés par les différents solveurs mais à mutualiser la « façon », la « manière » dont on va décrire, représenter ces données. Ceci n'aura d'intérêt bien sûr que si cette mutualisation facilite l'interoperabilité des données entre les outils. Il faut donc que la mutualisation de la « façon » de décrire les données soit accompagnée de mécanismes et d'outils qui faciliteront cette interopérabilité des données.

Nous proposons de concrétiser le formalisme pré-cité sous forme d'un langage formel qui sera réalisé sour forme de langage informatique. Ce langage sera donc utilisé pour décrire des modèles, en particulier des modèles de données pour la physique. Un modèle de données peut lui aussi être vu comme un langage car il décrit la « façon » dont un problème physique peut se composer. A ce titre nous parlerons aussi pour notre langage formel de métalangage (langage de langages). Nous appelerons ce langage le SPML (Standard Physic Modeling Language), il sera conforme à la notion de langage orienté objet et s'appuyera sur un langage déjà existant. Ce langage se base sur une grammaire qui sera décrite de manière pragmatique dans la suite du chapitre.

Le choix de définir un langage nouveau vient du fait que les langages existants ne couvrent pas l'ensemble des besoins de notre situation. Le point le plus important est que dans les langages existant, aucun mécanisme n'existe pour définir intrinséquement le comportement des objects vis à vis de leurs interactions avec l'utilisateur. C'est le point majeur qui distingue ce langage de ceux existants. D'autres points comme la double possibilité de typer ou non-typer les données sont aussi des différences. Cependant comme une grande partie des besoins sont aussi déjà couvert par des langages existants, nous avons choisi de nous appuyer sur l'un deux pour bénéficier de celles-ci à moindre coût. Nous avons pour cela choisi un langage ouvert et souple d'utilisation : le Python [7].

3.3 Fonctionnalités associées au SPML

La structure d'un langage ne suffit pas pour son utilisation, il doit aussi proposer des services qui permettent par exemple de manipuler les données ou d'interpréter les commandes. Les services disponibles autour du langage SPML sont surtout destinés à l'utilisateur intégrateur, c'est-à-dire la personne chargée d'écrire un modèle de données SPML. Ce chapitre détaille les services suivants :

– l'interprète de langage,

- la gestion des modèles,
- la gestion des données,
- la projection des données vers différents environnements,
- la mise à disposition des données sur un bus CORBA,
- la programmation du comportement des modèles.

3.3.1 L'interprète de commande

Le langage SPML est un langage de commande et en tant que tel, il doit être accompagné d'un interprète chargé d'analyser et d'exécuter les commandes qui lui sont adressées. Ces commandes sont par exemple destinées à la création de la structure d'un modèle SPML, ou encore à la gestion (lecture, écriture) des objets issus de ces modèles.

L'interprète SPML doit être capable de traiter aussi bien des instructions en ligne de commande (interprète interactif), qu'un fichier de script complet (c'est-à-dire généralement un modèle de données SPML). Chaque instruction est exécutée après analyse de sa syntaxe ; si celle-ci n'est pas correcte, l'interprète lève une exception indiquant le type d'erreur et l'instruction qui en est la source.

3.3.2 La gestion des modèles

Les modèles de données définissent une structure particulière pour décrire des propriétés physiques, souvent associée à la structure de données interne d'un solveur. Le langage SPML propose un service permettant de gérer la structure même de ces modèles. Il est ainsi possible d'organiser un modèle en sous-modèles plus fins, ou encore d'importer des modèles existants pour réutiliser leur structure.

3.3.3 La gestion des données

Quand un modèle de données est écrit avec le SPML, il peut alors être chargé par l'interprète SPML qui analysera et vérifiera les éventuelles erreurs de syntaxe (cf §3.3.1). Si le modèle est valide, un service propose alors de créer et de manipuler des objets dont la structure est définie dans le modèle. Par exemple, si un modèle définit la structure d'une classe SPML `Region`, il est ensuite possible de créer des instances de cette classe, de modifier cet objet en attribuant des nouvelles valeurs à ses attributs, ou encore de le détruire.

Les objets ainsi manipulés étant fortement typés, les opérations effectuées sont contrôlées. Un attribut SPML de type entier ne sera ainsi pas autorisé à recevoir toute autre type de valeur qu'un entier. Ainsi, si un type des donnée ne correspondent pas au type qui a été définit dans le modèle SPML, l'objet concerné sera considéré comme étant non valide du point de vue SPML.

Toutes les données sont organisées en jeux de données (*physical data sets*) et leur accès se fait par l'intermédiaire d'un point d'accès unique (le *physical data set reader*). Il est possible d'avoir plusieurs jeux de données dans une application pour pouvoir organiser les données.

3.3.4 La projection des données

Les modèles de données SPML sont décrits avec la même syntaxe, quel que soit leur domaine d'application. Les données issus de ces modèles représentent les propriétés physiques destinées à être utilisées dans le cadre d'une simulation d'un procédé physique.

Le langage SPML propose alors un service de projection de ces données dans des environnements ou elles pourront être traitées. Parmi les environnements de projections, les plus courants sont les interfaces graphiques, pour une manipulation plus intuitive des données, et bien sûr les solveurs associés aux modèles.

Cette projection est rendue possible grâce à la création d'une représentation arborescente neutre des jeux de données : un arbre abstrait. Cet arbre abstrait permet de faire le lien entre l'environnement de projection et les données crées à partir des modèles SPML. Par exemple, la projection d'un jeu de données vers une interface graphique pourra être matérialisée par une vue arborescente de ces données. En faisant apparaître un menu contextuel d'un objet, on souhaite voir apparaître toutes les actions réalisables sur cet objet (édition, suppression, information). L'arbre abstrait permet de connaître les actions possibles que l'on peut effectuer sur n'importe quel objet ainsi que sur leurs attributs. Il propose alors des méthodes permettant de récupérer ces informations. Toute modification des données entraîne la mise à jour dynamique de l'arbre abstrait, qui représente alors fidèlement un jeu de données avec son interaction vers un environnement extérieur.

3.3.5 La mise à disposition sur un bus de partage de données (ex : CORBA)

Le langage SPML propose un service permettant un accès réseau aux objets SPML. Pour cela, il utilise l'architecture CORBA [1] permettant le développement d'une application à partir de composants communiquant entre eux à travers une couche réseau. Ces composants peuvent être écrits dans des langages de programmation différents. Chaque composant est alors décrit sous la forme d'une interface écrite en langage IDL [2].

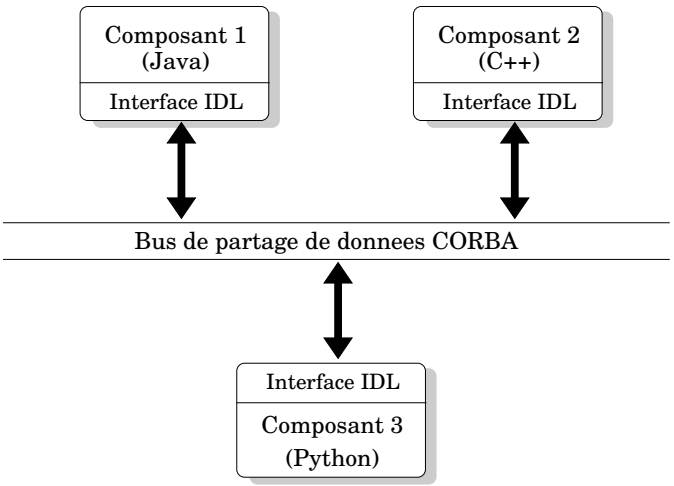


FIG. 3.1 – Composants logiciels communicant à travers un bus de partage de données CORBA

Le langage SPML propose ainsi un module CORBA avec son interface IDL permettant une communication avec d'autres modules externes. De plus, au chargement d'un modèle SPML, l'interface IDL correspondante à ce modèle est automatiquement générée. La structure de données devient alors disponible à travers la couche réseau, au-

[1] Common Object Request Broker Architecture
[2] Interface Definition Language

torisant ainsi la manipulation des données issues de ce modèle depuis un autre composant CORBA, conformément à la méthode expliquée en §3.3.3.

3.3.6 La programmation du comportement des modèles

En plus de la description brute de la structure des données dans un modèle SPML, il est possible de programmer un comportement plus évolué de ce modèle. Pour cela, le langage SPML propose de définir deux méthodes particulières pour chaque attribut SPML :

- `def spml_attr_bscontrols__(self,value)` : si elle existe, cette méthode est exécutée avant l'affectation de `value` à l'attribut `attr`. Elle doit renvoyer un objet du même type que `value`. Cette méthode permet par exemple de vérifier la cohérence physique d'une grandeur. Il est possible de lever une exception de type `spml_error` en cas de problème.
- `def spml_attr_ascontrols__(self,value)` : si elle existe, cette méthode est exécutée après l'affectation de `value` à l'attribut `attr`. Elle n'a pas de valeur de retour. Cette méthode permet par exemple de calculer un champ dérivé dépendant de cette valeur.

Le SPML propose aussi de définir des fonctions utilisateurs qui seront traitées comme des méthodes SPML. Il suffit pour cela de définir une méthode comme étant SPML, avec son dictionnaire de contrôle.

3.4 Concepts fondamentaux de programmation orientée objet (Exemple : Python)

La réalisation du métamodèle qu'est le SPML utilise beaucoup de mécanismes issus de la programmation orientée objet : héritage, polymorphisme, encapsulation. Ce chapitre va essayer d'expliquer les concepts nécessaires au SPML pour que le lecteur puisse comprendre comment ils ont ensuite été utilisés à notre profit. Nous illustrerons les notions grâce à des exemples écrits en Python. Cette partie est fortement basée sur le tutoriel de Shalabh Chaturvedi [8].

3.4.1 Concepts de base

Dans les versions de Python antérieures à la 2.2, il n'existait que trois sortes d'objets à manipuler :

- les types (`list`, `tuple`, `string`, ...),
- les classes,
- les instances

Les types et les classes étaient de nature vraiment différente et il était impossible de créer de nouveaux types. Le nouveau système à partir de la version 2.2 a changé tout cela et se rapporte plus au concept du tout objet.

Alors qu'est-ce qu'un objet ? Un objet est la brique de base du nouveau système et il est défini par le fait d'avoir :

- une identité unique (i.e. à partir de deux noms, on peut savoir si ce sont un seul même objet, ou non),

- des attributs (i.e. on peut accéder à d'autres objets avec `object-name.attributename`),
- des relations (ce sont des attributs, mais Python les reconnaît à part). Elles sont au nombre de deux :
  1. le *type* : chaque objet ne possède qu'un seul et unique *type*,
  2. les *bases* : des objets peuvent avoir une ou plusieurs *bases*.
- un nom (le nom de l'objet, certains peuvent en avoir plusieurs, d'autres peuvent ne pas en avoir).

Dans ce nouveau système, tout est objet : `list`, `tuple` et `string` sont des objets. Toutes les classes que l'on définit sont des objets, et bien sûr les instances de ces classes sont aussi des objets. Toutefois, tous les objets ne sont pas égaux comme nous allons le voir par la suite.

Les relations entre objets utilisées dans ce système sont des relations *type-instance* et *supertype-soustype*. Ces relations peuvent être affiliées respectivement aux termes *« est un genre de »* et *« est une instance de »* (figure 3.2) :

« est un genre de » : Relation plus connue sous le terme de spécialisation dans la syntaxe orientée objet. Cette relation existe entre deux objets quand l'un (le *soustype*) est une version spécialisée de l'autre (le *supertype*). Par exemple, un chat *est un genre de* mammifère. Il possède tous les traits d'un mammifère et d'autres traits spécifiques qui font de lui un chat.

« est une instance de » : Aussi connue comme instanciation, cette relation existe entre deux objets quand l'un (l'*instance*) est un exemple concret de ce que l'autre définit (le *type*). Je possède un chat de compagnie qui s'appelle Moka. Moka *est une instance de* chat.

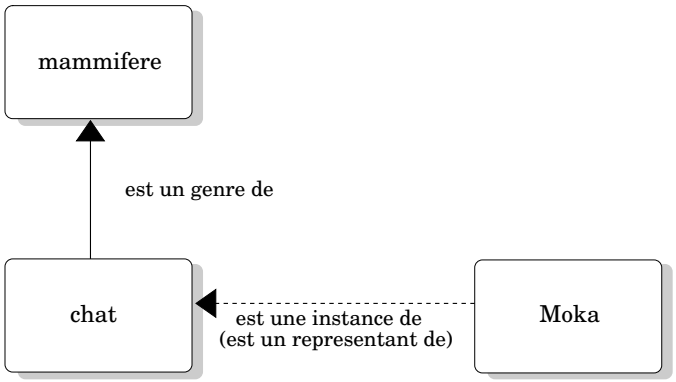


FIG. 3.2 – Relations entre objets

La relation *supertype-soustype* est en trait plein pour renforcer le fait que les objets sont plus proches que dans une relation *type-instance*. La définition de ces relations entraîne les propriétés de transition suivantes :

- si X est une instance de A, et A est un soustype de B, alors X est aussi une instance de B,

– si B est une instance de M, et A est un soustype de B, alors A est aussi une instance de M.

Ces propriétés sont illustrées sur la figure 3.3.

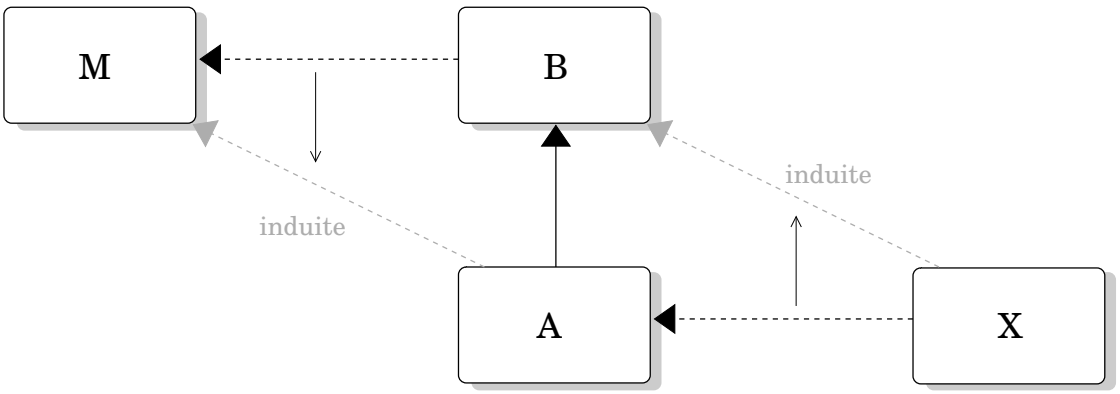


FIG. 3.3 – Transition des relations

D'après ces règles et ces relations, nous pouvons donc écrire :

1. Moka est une instance de chat (ou alors, le type de Moka est chat),
2. Moka est une instance de mammifère (ou alors, le type de Moka est mammifère).

Cependant, nous avons dit auparavant qu'un objet avait un seul type. Pourquoi Moka a alors deux types ? En fait, bien que les deux affirmations soient justes, l'une est plus juste que l'autre. Donc en réalité le type de Moka est chat, et Moka est une instance de chat et de mammifère. Avec la syntaxe Python, cela s'écrit comme suit :

– `Moka.__class__` is `chat` (l'attribut __class__ nous donne le type d'un objet),
– Les deux tests `isinstance(Moka,chat)` et `isinstance(Moka,mammifère)` sont vrais.

Des règles similaires existent pour la relation *supertype-soustype* :

– Si A est un soustype de B, et B est un soustype de C, alors A est aussi un soustype de C.

Un chat est un genre de mammifère et un mammifère est un genre d'animal. Un chat est donc un genre d'animal. Avec la syntaxe Python, cela s'écrit comme suit :

– `chat.__bases__` is `(mammifère,)` (l'attribut __bases__ nous donne un tuple contenant les supertypes d'un objet),
– les deux tests `issubclass(chat, mammifère)` et `issubclass(chat,animal)` sont vrais.

Il est possible pour un objet d'avoir plus d'une base.

3.4.2 Les objets Python

Maintenant que nous avons étudié les notions d'objet et de relations, penchons nous sur les objets Python, et d'abord sur les objets de base : `<type 'object'>` et `<type 'type'>` avec l'exemple suivant :

```
1  >>> object
2  <type 'object'>
3  >>> type
4  <type 'type'>
5  >>> object.__class__
6  <type 'type'>
7  >>> object.__bases__
8  ()
9  >>> type.__class__
10 <type 'type'>
11 >>> type.__bases__
12 (<type 'object'>,)
```

lignes 1 et 3 : Noms des objets dans Python.

lignes 5, 7, 9 et 11 : Les relations entre ces objets.

Ces deux objets sont à la base de tous les objets Python. Ils sont indissociables l'un de l'autre et ne peuvent pas être définis seuls. Ces objets et leur relations sont dessinés sur la figure 3.4.

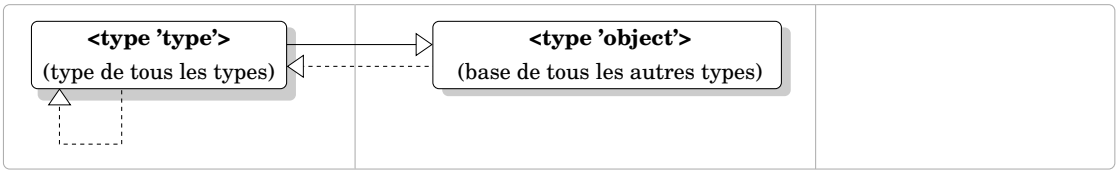


FIG. 3.4 – Les deux objets de base de Python : `type` et `object`

L'exemple suivant montre ce que donne l'instruction `isinstance` sur ces objets :

```
1  >>> isinstance(object, object)
2  True
3  >>> isinstance(type, object)
4  True
```

ligne 1 : Ici les propriétés de transitions ont été appliquées : `<type 'type'>` étant un soustype de `<type 'object'>`, les instances de `<type 'type'>` sont aussi des instances de `<type 'object'>`.

ligne 3 : Encore une fois ce sont les propriétés de transitions qui ont été appliquées : puisque `<type 'type'>` est un soustype de `<type 'object'>` et que `<type 'object'>` est aussi une instance de `<type 'object'>`, alors `<type 'type'>` est une instance de `<type 'object'>`.

Les objets qui ont été présentés jusqu'ici font partie de la catégorie d'objets appelés `type objects`. En effet un objet peut être un `type object` ou un `non-type object`. Les `type objects` peuvent participer à une relation `type-instance` aussi bien du côté `type` que du côté `instance`. En revanche les `non-type objects` ne peuvent participer

à ces relations que du côté `instance`.

Les `type objects` sont aussi les seuls à pouvoir être les `supertypes` d'un autre objet. Les `non-type objects` sont des objets tellement concrets qu'il est impossible qu'un autre objet en soit un soustype. Pour s'en rendre compte, peut on trouver un objet qui soit un genre de Moka ?

Nous venons donc de voir que Moka était un `non-type object` et que `chat` était un `type object`. Pour simplifier l'écriture, les `type objects` sont appelés plus simplement `types`. Tous les soustypes d'un `type object` sont aussi des `types`.

Nous pouvons résumer les dernières notions ainsi :

1. `<type 'object'>` est une instance de `<type 'type'>`,
2. `<type 'object'>` n'est le soustype d'aucun objet,
3. `<type 'type'>` est une instance de lui-même,
4. `<type 'type'>` est un soustype de `<type 'object'>`,
5. il n'existe que deux genres d'objets dans Python : les `type objects` et les `non-type objects`.

La figure 3.5 montre d'autres types prédéfinis dans Python. Examinons les avec des commandes Python de l'exemple suivant :

```
1  >>> list
2  <type 'list'>
3  >>> list.__class__
4  <type 'type'>
5  >>> list.__bases__
6  (<type 'object'>,)
7  >>> tuple.__class__, tuple.__bases__
8  (<type 'type'>, (<type 'object'>,),)
9  >>> dict.__class__, dict.__bases__
10 (<type 'type'>, (<type 'object'>,),)
11 >>>
12 >>>maListe = [1,2,3]
13 >>>maListe.__class__
14 <type 'list'>
```

ligne 1 : L'objet prédéfini `<type 'list'>`.

ligne 3 : Son type est `<type 'type'>`.

ligne 5 : C'est un supertype : `<type 'object'>`.

ligne 7 et 9 : Mêmes remarques pour `<type 'tuple'>` et `<type 'dict'>`. Un tuple est l'équivalent d'une liste mais elle n'est pas modifiable.

ligne 12 : Commande pour créer une instance de `<type 'list'>`.

ligne 14 : Enfin un objet ayant un autre objet que `<type 'type'>` comme type.

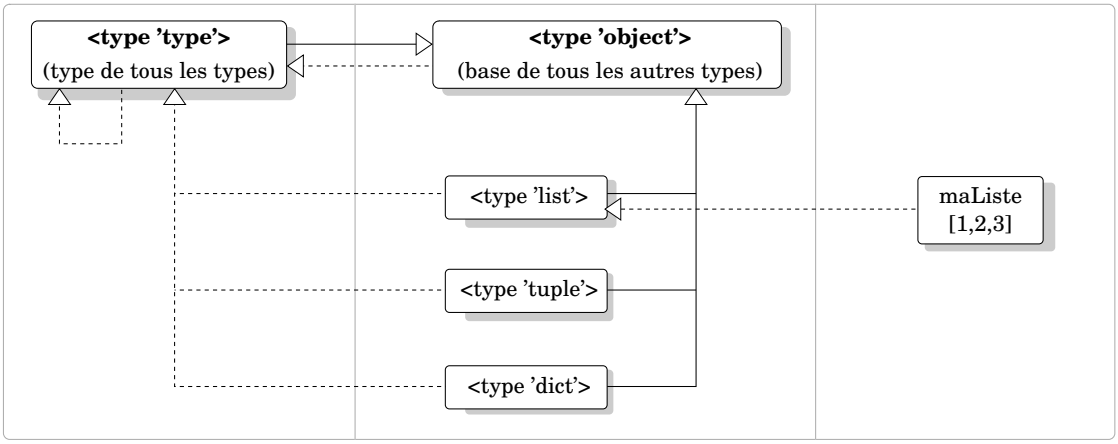


FIG. 3.5 – Quelques `types` prédéfinis dans Python

De la même manière que pour la création d'une liste, quand on crée un tuple ou un dictionnaire on obtient des instances de leurs types respectifs. Par contre, il n'est pas possible de créer une instance de `maListe` parce que l'objet `maListe` est un `non-type object`.

Nous pouvons donc maintenant mieux définir ce qu'est un `type object` : si un objet est une instance de `<type 'type'>`, alors c'est un `type object`, sinon c'est un `non-type object`.

3.4.3 Créer de nouveaux objets

De nouveaux objets peuvent être créés par deux méthodes : par spécialisation ou par instanciation :

– Création d'objet par spécialisation :

```
1 class C(object):
2   pass

3 class D(object):
4   pass

5 class E(C, D):
6   pass

7 class maListe(list):
8   pass
```

ligne 1 : Le mot-clé `class` informe Python de créer un nouvel objet `type` en spécialisant/soustypant un objet `type` existant.

ligne 2 : Le corps de la définition d'une classe contient généralement des méthodes et autres attributs.

ligne 5 : La spécialisation à partir de plusieurs `bases` est aussi possible.

ligne 7 : La plupart des types prédéfinis peuvent être soustypés.

Le terme *class* est traditionnellement utilisé pour parler d'un objet créé avec le mot-clé `class`. Cependant, les classes sont maintenant synonymes des types, et les types prédéfinis ne sont pas appelés des classes. Nous préferrerons alors utiliser le terme *type* pour à la fois les types prédéfinis et les nouveaux types créés.

– Création d'objet par instanciation :

```
1 obj = object()

2 cobj = C()

3 maListe = [1,2,3]
```

lignes 1 et 2 : l'opérateur *call* (`()`) créé un nouvel objet en instanciant un objet existant. Ce dernier doit être un `type object`. Selon le `type object`, l'opérateur *call* peut accepter des arguments.

ligne 3 : Python propose une syntaxe pour créer des objets à partir de types prédéfinis (par exemple les crochets créent une instance de `<type 'list'>`).

Les nouveaux objets créés dans les exemples précédents sont placés sur le diagramme de la figure 3.6.

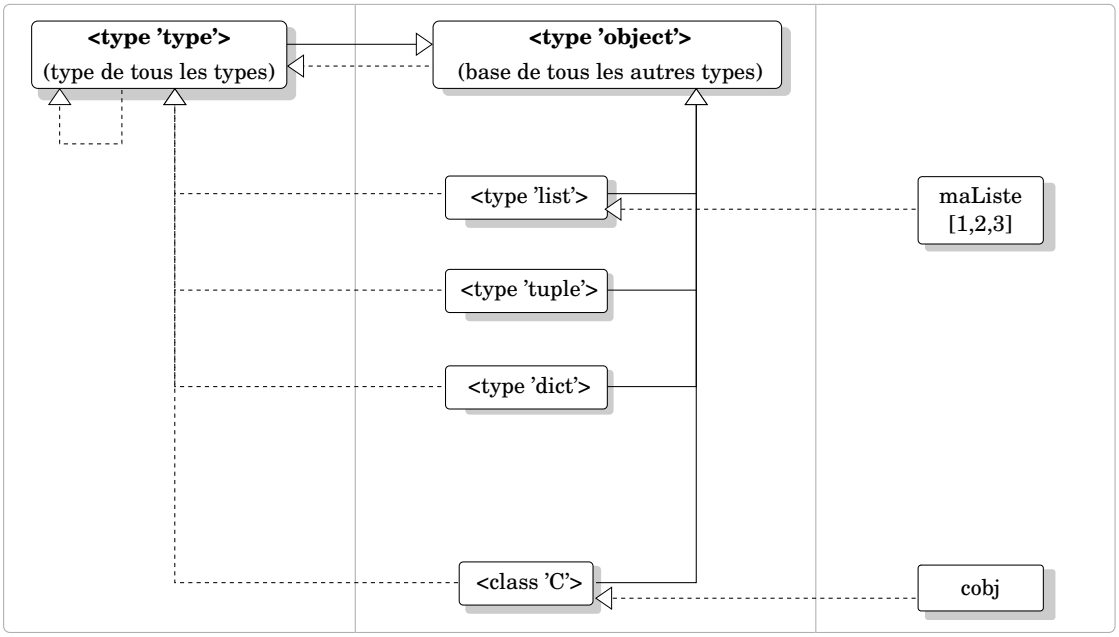


FIG. 3.6 – Objets créés par un utilisateur

Nous pouvons remarquer que le type `C` est automatiquement une instance de `<type 'type'>`, juste en soustypant `<type 'object'>`. Ceci est vérifié en regardant ce que donne l'instruction `C.__class__`. La prochaine partie explique pourquoi.

3.4.4 Le processus d'instanciation

En interne, quand Python créé un nouvel objet, il utilise toujours un `type object` et créé une instance de cet objet. Plus précisément, il utilise les méthodes `__new__()` et `__init__()` du `type object`. On peut dire que le `type object` sert d'usine à fabriquer de nouveaux objets, ce qui explique pourquoi tous les objets ont un type.

Quand on créé un nouvel objet en soustypant un autre objet, Python regarde l'objet base qui a été spécifié et utilise son type comme type pour le nouvel objet. Ainsi quand le type `C` a été créé en soustypant `<type 'object'>`, le type de `<type 'object'>` a été utilisé, c'est-à-dire `<type 'type'>`, pour créer `C`. Par ailleurs, ceci revient à dire que tout soustype de `<type 'object'>` (et leurs soustypes, ...) auront `<type 'type'>` comme type.

Il est possible de spécifier à Python quel type utiliser pour créer un objet. Il suffit pour cela d'utiliser l'attribut de classe `__metaclass__` comme le montre l'exemple suivant :

```
1 class Mon_C_Avec_Un_Type_Special(object):
2   __metaclass__ = Un_Type_Special
```

Pour cet exemple, Python va créer `Mon_C_Avec_Un_Type_Special` en instanciant `Un_Type_Special` et non `<type 'type'>`. Pour que cela fonctionne, il faut que `Un_Type_Special` soit un soustype du type de l'objet de base. Dans cet exemple :

- L'objet de base de `Mon_C_Avec_Un_Type_Special` est `<type 'object'>`.
- Le type de `<type 'object'>` est `<type 'type'>`.
- Donc `Un_Type_Special` doit être un soustype de `<type 'type'>`.

Si l'objet que l'on créé est un soustype de plusieurs objets et que l'attribut `__metaclass__` n'est pas spécifié, le type utilisé pour la création de cet objet dépend de ses bases : si toutes les bases ont le même type, alors celui-ci sera utilisé, sinon l'attribut `__metaclass__` doit être spécifié.

3.4.5 Classement des objets Python

La figure 3.7 classe tous les objets que nous avons vu jusqu'à maintenant en trois catégories : les *métaclasses*, les *classes* et les *instances*.

Ce classement nous montre quelques points remarquables :

1. Les flèches pointillées traversent les limites verticales, c'est-à-dire vont de *objet* à *métaobjet*. La seule exception est `<type 'type'>` qui est son propre *métaobjet*.
2. Les flèches continues ne traversent pas les limites verticales. La seule exception est la relation `<type 'type'>` → `<type 'object'>`.
3. Aucune flèche continue ne peut apparaître dans la colonne de droite : ces objets sont trop concrets pour être soustypés.
4. Les flèches pointillés ne peuvent avoir leur tête dans la colonne de droite : ces objets sont trop concrets pour être instanciés.
5. Les deux colonnes de gauche contiennent les `type objects` et la colonne de droite les `non-type objects`.
6. Si un objet est créé en soutypant `<type 'type'>`, il apparaîtrait dans la colonne de gauche et serait à la fois soustype et instance de `<type 'type'>`.

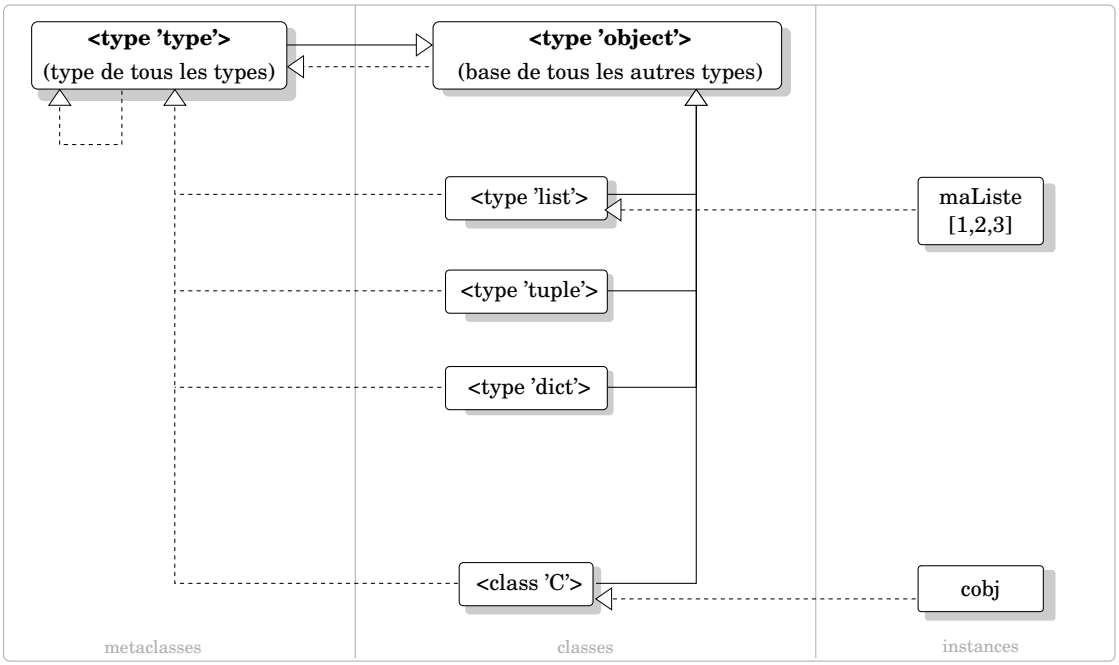


FIG. 3.7 – Classement des objets Python

On remarque aussi que `<type 'type'>` est le type de tous les types et que `<type 'object'>` est le supertype de tous les types (sauf lui-même).

Nous pouvons à présent résumer tous ce qui a été dit à propos des objets et types de Python :

- Il existe deux genres d'objets dans Python :
  1. Les `type objects` : peuvent créer des instances, peuvent être soustypés.
  2. Les `non-type objects` : ne peuvent pas créer d'instances, ne peuvent pas être soustypés.

- `<type 'type'>` et `<type 'object'>` sont deux objets clés du système.
- `objectname.__class__` existe pour tout objet et pointe vers le type de l'objet.
- `objectname.__bases__` existe pour tout `type object` et pointe vers les supertypes de l'objet. Seul `<type 'object'>` n'en possède aucun.
- Pour créer un objet en utilisant le sous-typage (ou spécialisation), il faut utiliser le mot-clé *class* et préciser les bases (et optionnellement le type) du nouvel objet. Ce mécanisme créé toujours un `type object`.
- Pour créer un objet en utilisant l'instanciation, il faut appeler l'opérateur `call` (`()`) du `type object` que l'on veut utiliser. Ce mécanisme peut créer un `type object` ou un `non-type object` en fonction du `type object` utilisé.
- Python offre des syntaxes particulières pour créer certains `non-type objects`. Par exemple `[1,2,3]` crée une instance de `<type 'list'>`.
- En interne, Python utilise *toujours* un `type object` pour créer un nouvel objet. Ce dernier est une instance du `type object` utilisé. Python trouve le `type object` d'un objet créé avec le mot-clé `class` en analysant les objets de bases et en trouvant leurs types.

– Le test `issubtype(A,B)` vérifie la relation supertype-soustype entre A et B. Il renvoie `True` si :
  1. `B` est dans `A.__bases__`, ou
  2. `issubtype(Z,B)` est vrai pour tous les `Z` dans `A.__bases__`.
– Le test `isinstance(A,B)` vérifie la relation type-instance entre A et B. Il renvoie `True` si :
  1. `B` est dans `A.__class__`, ou
  2. `issubtype(A.__class__,B)` est vrai.

Les mécanismes que l'on vient d'étudier ont été utilisé pour la réalisation d'un langage dédié à la description des propriétés physiques. Le chapitre 3.5 qui suit explique la structure de ce langage appelé *SPML* (Standard Physic Modeling Language). Les fonctionnalités proposées autour de ce langage sont traitées dans le chapitre 3.3.

3.5 Structure du SPML

La structure du SPML est conforme à l'architecture basée sur la métamodélisation proposée par l'UML. Le schéma conceptuel généralement accepté pour la métamodélisation est basée sur l'architecture à quatre couches suivante [9] :
– Méta-métamodèle
– Métamodèle
– Modèle
– Objets utilisateurs

Le rôle de ces couches est résumé par le tableau 3.1. Ce chapitre décrit plus en détails les couches du métamodèle et du modèle qui constituent le SPML.

Couche	**Description**	**Exemple**
Méta-métamodèle	méta-métamodèle	`type`
Métamodèle	Une instance du méta-métamodèle. Définit le langage pour spécifier un modèle.	`spml_type_type`, `spml_type_intrinsic`, `spml_type_class`
Modèle	Une instance du métamodèle. Définit un langage pour la description de propriétés physiques.	`spml_class`, `spml_int`, `spml_app_model`
Objets/données utilisateur	Une instance du modèle. Définit les propriétés d'un domaine spécifique.	`Identification`, `Region`, `ConditionIsotherme`

Tab. 3.1 – Architecture UML à 4 couches

3.5.1 Description générale

La structure globale du métamodèle du SPML est conforme aux concepts de programmation orientée objet précédents auxquels elle ajoute d'autres notions. Pour cela, le SPML définit deux objets principaux : `<type 'spml_type_type'>` et `<type 'spml_object'>`. Ces deux objets et leurs relations sont dessinés sur la figure 3.8.

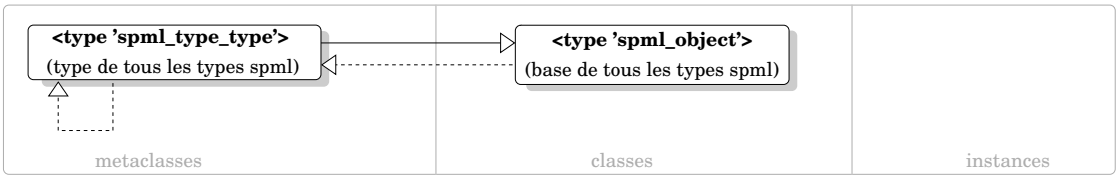


FIG. 3.8 – Objets de base du SPML

Les relations entre ces objets sont réalisées grâce à un mécanisme d'héritage des objets de base `<type 'type'>` et `<type 'object'>` (figure 3.9). Nous bénéficions ainsi de tous les mécanisme associés à ces deux objets et à leurs relations, et nous définissons alors le SPML comme une extension du langage défini par ces objets, dans notre cas Python.

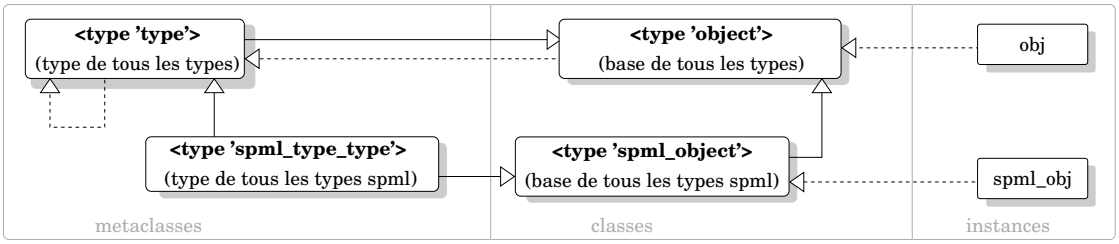


FIG. 3.9 – Réalisation des relations des objets de base du SPML par héritage

La structure globale du SPML est composée de 5 modules principaux représentés sur le diagramme de la figure 3.10 :

- le module *spml_type* décrit la structure des données,
- le module *spml_ui* concerne l'interface utilisateur,
- le module *spml_utility* décrit le comportement des collections et des méthodes,
- le module *spml_app* traite de la définition des applications et des modèles,
- le module *spml_exceptions* définit la structure des exceptions propres au SPML.

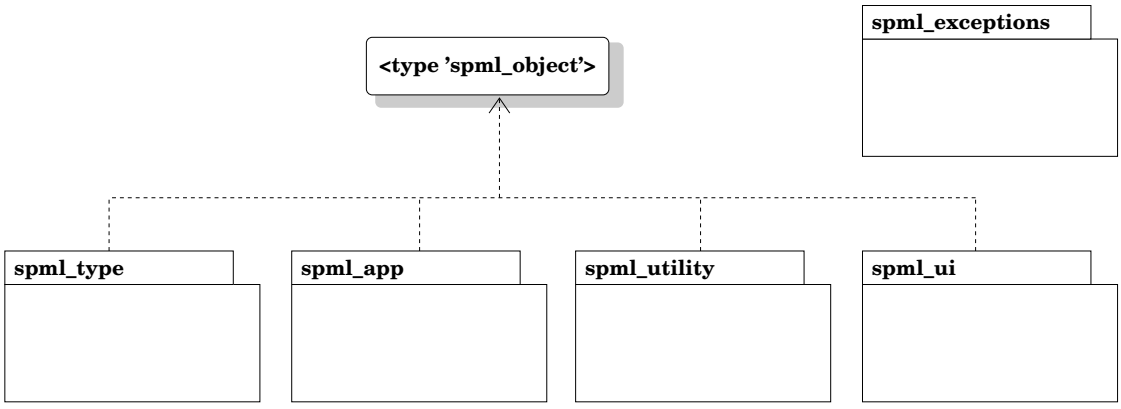


FIG. 3.10 – Organisation globale de la structure du SPML

Tous ces modules sont décrits dans les parties suivantes. Pour chacun, un schéma représentant l'organisation des classes est donné et commenté, associé à la définition et à la description de la structure des principales classes avec des exemples d'utilisation.

3.5.2 Le module spml_app

Cette structure est conçue pour permettre la définition d'une application et des modèles de données (figure 3.11). Toutes les classes héritent de la classe mère abstraite `<type 'spml_app_object'>`, elle-même héritant de `<type 'spml_object'>`.

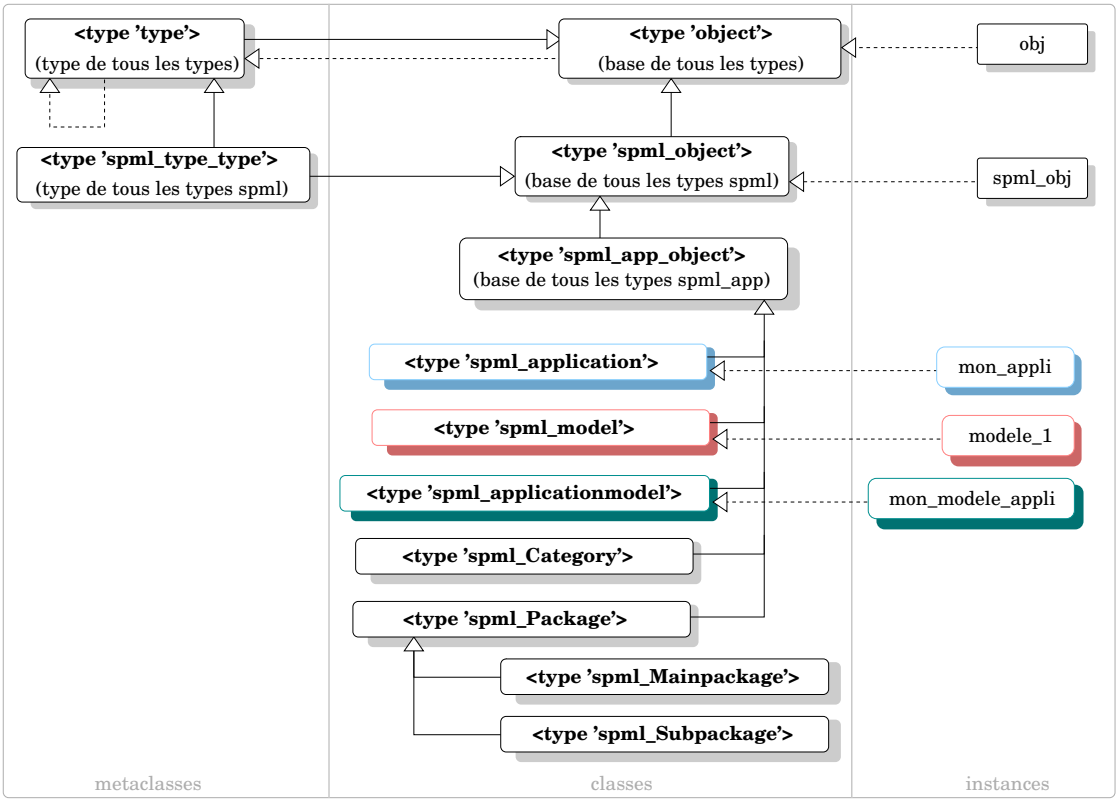


FIG. 3.11 – Organisation des types du module *spml_app*

3.5.2.1 L'application : `<type 'spml_application'>`

L'application est l'objet exécutable. Elle représente un jeu de fonctionnalités clairement identifiés par l'utilisateur final. Il ne peut y avoir qu'une seule application pour un problème. Elle est composée des attributs obligatoires suivants :

- un identifiant unique,
- un numéro de version,
- un modèle d'application `spml_applicationmodel` qui regroupe les modèles de données physiques.

Le type `<type 'spml_application'>` possède les méthodes suivantes, permettant de traiter les modèles SPML :

`compileIDL(self,path=None,verbose=1)` : compile les modèles de données de l'application vers un fichier IDL (Interface Definition Language) [3]. Par défaut le chemin de ce fichier est le répertoire courant, sauf si `path` est précisé.

`getIDLFileName(self)` : renvoie le nom du fichier IDL associé à l'application. Ce nom est créé à partir du nom de l'application et de sa version.

`addInstanciableObjects(self,dict)` : fait appel à la méthode du même nom sur tous les modèles de données définis pour l'application. Cette méthode remplit le dictionnaire `dict` avec les classes concrètes des modèles utilisés par l'application.

`getApplicationModel(self)` : renvoie le modèle d'application associé.

La syntaxe pour déclarer une application `mon_appli` est donnée dans l'exemple 3.1.

```
1 mon_appli = spml_application(id="Mon_application",
2                               version = "v1.0",
3                               models = mon_modele_appli)
```

Exemple 3.1 – Déclaration d'une application (`spml_application`)

La création d'une instance d'application suppose que le modèle de l'application `mon_modele_appli` ait auparavant été créé. La structure d'un modèle d'application est expliquée dans la partie suivante.

3.5.2.2 Le modèle de l'application : `<type 'spml_applicationmodel'>`

Le modèle d'application est défini pour une application donnée. Il regroupe les modèles de données physiques utilisés dans l'application. Il est composé des attributs obligatoires suivants :

- un identifiant unique,
- un numéro de version,
- une liste de modèles de données.

La syntaxe pour déclarer un modèle d'application `mon_modele_appli` est données dans l'exemple 3.2.

```
1 mon_modele_appli = spml_applicationmodel(id="Application_model",
2                                          version = "v1.0",
3                                          models = [modele_1,
                                              modele_2])
```

Exemple 3.2 – Déclaration d'un modèle d'application (`spml_applicationmodel`)

Le modèle de l'application regroupe les modèles de données `modele_1` et `modele_2` qu'il faut donc avoir déjà déclarés. La structure d'un modèle de données est expliquée dans la partie suivante.

[3] L'IDL est un langage de description dont la syntaxe est très proche de celle du langage `C++`. Il est utilisé comme interface pour la communication entre différents programmes écrits dans des langages différents.

3.5.2.3 Le modèle de données : <type 'spml_model'>

Un modèle de données est un ensemble de structures de données réutilisables. Elles regroupent un ensemble de concepts associés à un domaine d'activité particulier. Les structures de données sont normalement associées à un seul modèle de données mais elles peuvent référencer des structures de données définies dans d'autres modèles de données. Une application est définie par une combinaison de plusieurs modèles de données. Un modèle de données est facilement dérivé ou spécialisé, de la même manière que le font les classes, pour étendre ou restreindre un domaine d'activité. Ces relations d'héritage amènent à un arbre appelé l'arbre d'héritage des modèles de données.

Un modèle de données est défini par un identifiant unique. Par convention cette identifiant est entièrement en minuscule, chaque mot étant séparé par un trait souligné. Le modèle de données peut aussi avoir des sous-modèles de données, réalisant ainsi l'arbre d'héritage.

La syntaxe pour déclarer les modèles de données `modele_1` et `modele_2`, avec `modele_2` sous-modèle de `modele_1` est données dans l'exemple 3.3.

```
1 modele_1 = spml_model(id="mon_premier_modele")
2 modele_2 = spml_model(id="mon_deuxieme_modele",supertype=modele_1)
```

Exemple 3.3 – Déclaration de modèles (`spml_model`)

3.5.3 Le module spml_type

Cette structure définit les types de données proposés par le métamodèle. Elle est détaillée sur la figure 3.12. On y retrouve entre autres les types intrinsèques, les énumérations, les interfaces et les classes. Toutes les classes héritent de la classe mère abstraite `<type 'spml_type_type'>`. Cette dernière hérite non seulement de `<type 'spml_object'>`, mais aussi du type de base Python `<type 'type'>`. De fait, toutes les classes définies dans cette structure sont donc des métaclasses, ou types, qui serviront pour l'instanciation d'autres classes, comme expliqué dans le chapitre 3.4.4. Ces types définissent le comportement de leurs instances et définissent ainsi les principales caractéristiques du langage SPML.

Le langage SPML est accompagné d'une bibliothèque de classes abstraites, instances de ces types. L'exemple 3.4 illustre ceci avec la définition de la classe abstraite `spml_int`.

```
1 class spml_int(int):
2    __metaclass__ = spml_type_intrinsic
```

Exemple 3.4 – Une instance de type `spml_type_intrinsic` : `spml_int`

Cette classe est un type intrinsèque, comme indiqué par la valeur de son attribut `__metaclass__`. Lorsque le moteur de Python va créer une telle classe, il passera alors par `<type 'spml_type_intrinsic'>` comme *type* au lieu de l'habituel `<type 'type'>`. Pour utiliser le comportement d'un intrinsèque de type `spml_int`, il suffira alors de soutyper cette classe, et de renseigner les attributs requis par `<type 'spml_type_intrinsic'>`. Les structures des types intrinsèques et des classes SPML sont expliquées dans les deux parties suivantes.

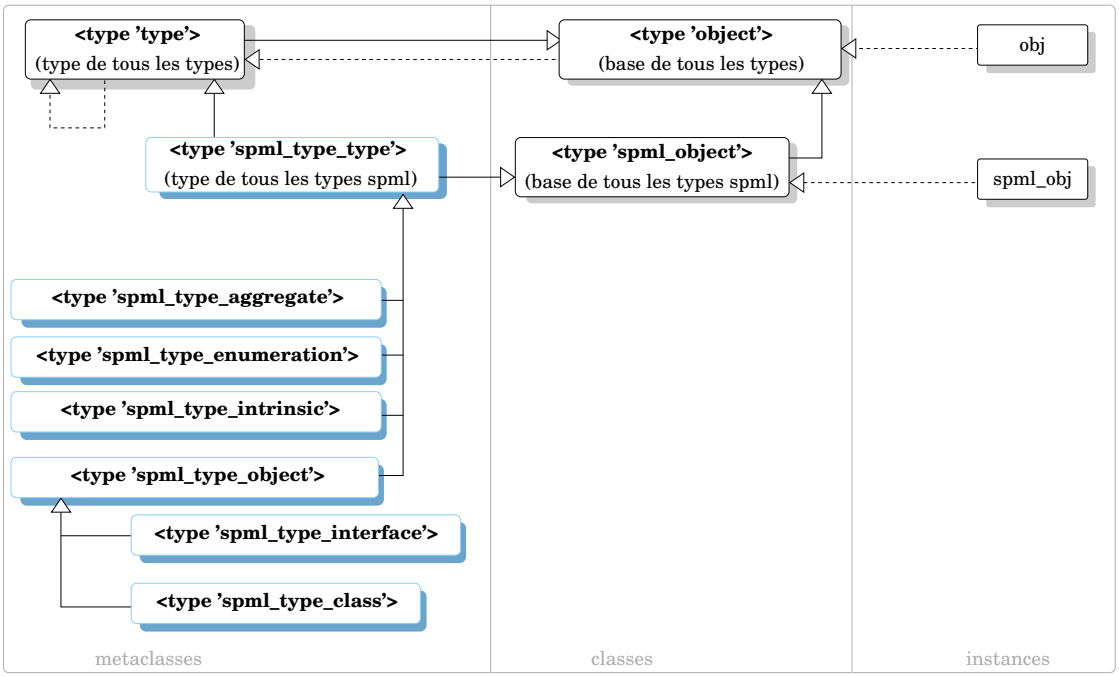


FIG. 3.12 – Organisation des types du module *spml_type*

3.5.3.1 Les types SPML intrinsèques

Les types SPML intrinsèques définissent les structures de données les plus simples pouvant être manipulées par une application. Les types de base tels que les entiers, réels, chaînes de caractères ou booléens font parties de ces types intrinsèques. Les types intrinsèques peuvent aussi être utilisés pour définir de nouveaux types de base : URL, image, noms, ... Un type intrinsèque SPML possède les attributs suivants :

- Un identifiant unique,
- Une référence vers un modèle de données,
- Un type de base parmi la liste suivante : *integer, real, numeric, string, numeric_or_string, boolean, void*.

Pour définir une classe comme étant un type SPML intrinsèque, celle-ci doit hériter de l'un des types proposées par le SPML : `<type 'spml_int'>`, `<type 'spml_long'>`, `<type 'spml_float'>`, `<type 'spml_complex'>` et `<type 'spml_str'>`. La figure 3.13 montre bien que ces types sont tous des instances de `<type 'spml_type_intrinsic'>`.

Une classe ainsi définie possède alors des méthodes particulières, permettant principalement une introspection sur elle-même :

`getRelatedType(self)` : renvoie le type de base,

`is_a(self,aString)` : renvoie 1 si le type de base est `aString`, 0 sinon,

`checkTypeOfValue(self,value)` : vérifie la cohérence entre le type de `value` et le type de base de l'intrinsèque testé. Renvoie 1 si le type de `value` correspond au type de base,

`getPythonEquivalentTypes(self)` : renvoie le type Python correspondant au type de base (ex : `IntType` est le type Python équivalent de `integer`).

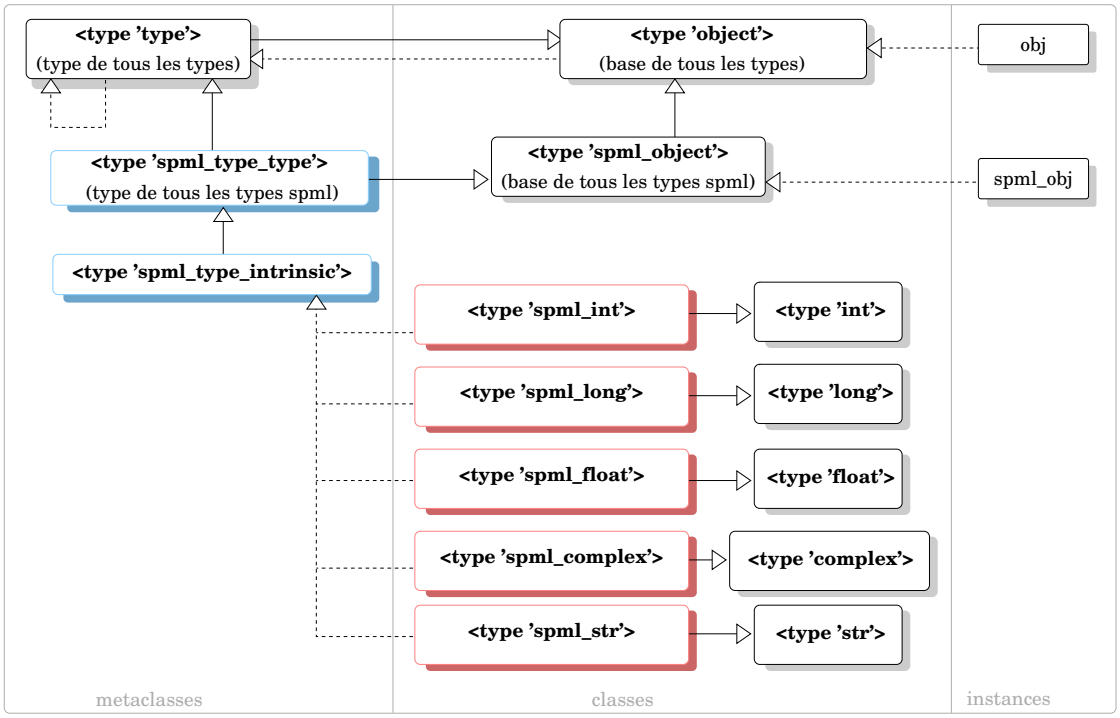


FIG. 3.13 – Les types intrinsèques du module *spml_type*

L'exemple 3.5 donne les instructions qui permettent de créer une classe SPML de type intrinsèque dont le rôle est de fournir à un objet SPML une identification unique.

```
# définition de la classe
class Identification(spml_str):
    spml_type__ = "STRING"
    spml_modelowner__ = monModel
```

Exemple 3.5 – Création d'une classe SPML de type intrinsèque

```
# instanciation de la classe
mon_id = Identification()
```

Cette classe et son instance sont représentées sur le diagramme de la figure 3.14

3.5.3.2 Les classes SPML

Les classes SPML sont les éléments clés utilisés par une application. Chaque classe SPML de l'application est défini par ses données (attributs) et services (méthodes). Une classe SPML peut implémenter un jeu d'interfaces et étendre au plus une classe SPML de base. Ceci conduit à un schéma d'héritage simple pour les classes SPML, et à de multiples implémentations d'interfaces. Une classe SPML comporte aussi un *stéréotype* parmi la liste suivante :

ABSTRACT **:** pour une classe SPML abstraite au sens de la programmation orientée objet et pour une architecture nécessitant du polymorphisme,

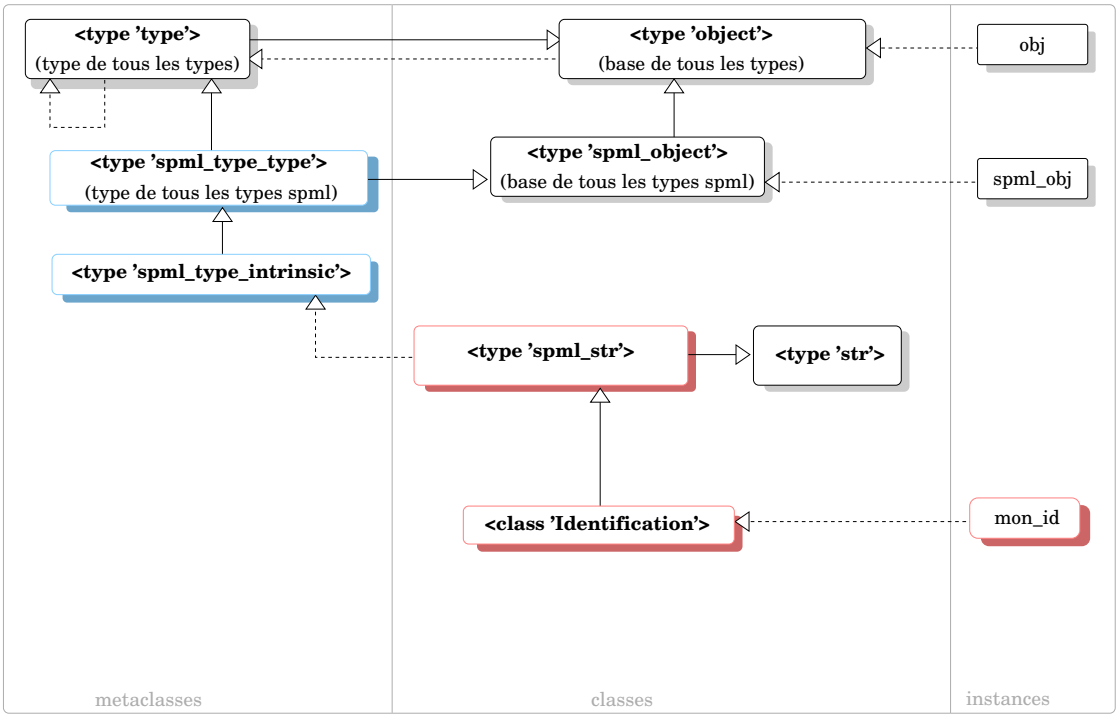


FIG. 3.14 – Exemple de type intrinsèque SPML

CONCRETE **:** pour une classe SPML concrète,

NODE **:** similaire à *ABSTRACT*, sauf qu'une classe SPML *NODE* ne déclare pas tous les comportements que ses classes SPML *CONCRETE* définissent,

WRAPPER **:** pour une classe SPML destinée à reformater ou encapsuler un jeu de comportements.

Concrètement, les stéréotypes *ABSTRACT* et *CONCRETE* sont les seuls à être utilisés pour le moment. En ce qui concerne la structure, une classe SPML est définie par :

- un identifiant unique,
- un stéréotype (ABSTRACT, CONCRETE, NODE, WRAPPER),
- une référence vers un modèle de données,
- une référence vers un package (optionnel),
- une référence vers une classe SPML qu'elle étend (optionnel),
- un jeu d'interfaces qu'elle implémente (optionnel),
- un jeu de méthodes (optionnel),
- un jeu d'attributs (optionnel),
- des informations d'interface utilisateur (optionnel).

Pour définir une classe comme étant SPML, il faut la faire hériter de la classe `spml_class` fournit par la librairie SPML (figure 3.15). Cette classe est une implémentation de la métaclasse `spml_type_class` du métamodèle.

Cette métaclasse rajoute des comportements spécifiques à ceux des traditionnelles classes Python :

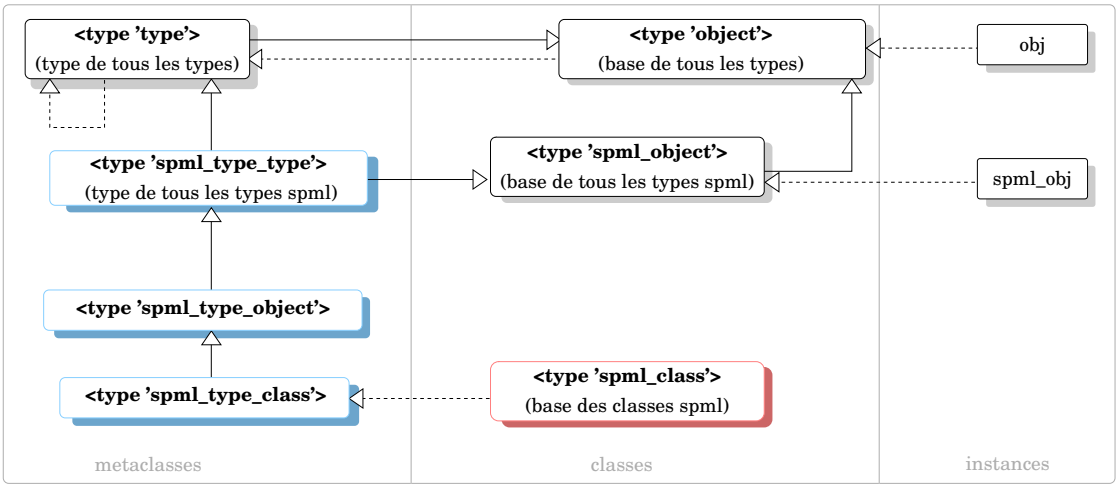


FIG. 3.15 – Les classes SPML du module *spml_type*

- les classes SPML ont un *stéréotype* (abstract, concrete, node, wrapper),
- certains attributs ont du contrôle de type : leur nom est de la forme `spml_*`. En fait, un attribut `spml_monattr` va générer un attribut SPML `monattr` dont le type est contrôlé, alors que l'attribut `spml_monattr` n'a aucun contrôle de type (attribut Python classique),
- les attributs `spml_*` existent toujours pour une instance de `spml_class` et les objets cibles peuvent être valides ou non,
- les attributs `spml_*` ont des relations particulières avec leur classe (association, composition, agrégation, identification),
- les attributs `spml_*` ont une définition de leur comportement (forcé, optionnel, dérivé, interne),
- les attributs `spml_*` peuvent avoir des comportements particuliers, ceci pour des besoins d'interface utilisateur par exemple.

La syntaxe pour créer une classe SPML `Region` est présentée dans l'exemple 3.6. Cette classe est représentée sur la figure 3.16.

```
1 # définition de la classe
2 class Region(spml_class):
3   #
4   # définition du comportement de la classe
5   #
6   spml_modelowner__ = modele_1
7   spml_stereotype__ = "CONCRETE"
```

Exemple 3.6 – Syntaxe pour la création d'une classe SPML `Region`

Une classe SPML est avant tout une classe Python, elle peut donc avoir des attributs et méthodes Python classiques. L'exemple 3.7 montre ainsi qu'on peut ajouter un attribut `attr` à la classe SPML `Region`.

```
1 # définition de la classe
2 class Region(spml_class):
```

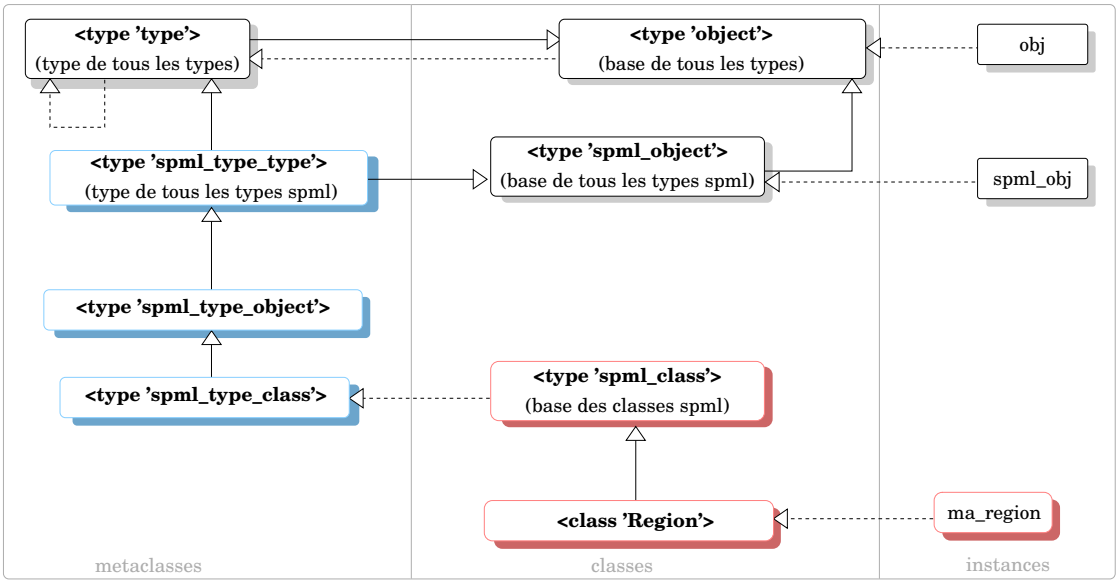


FIG. 3.16 – Exemple de classe SPML utilisateur

```
  #
  # définition du comportement de la classe
  #
  spml_modelowner__ = modele_1
  spml_stereotype__ = "CONCRETE"
  #
  # définition de l'attribut Python 'attr'
  #
  attr = "un_attribut_Python"
```

Exemple 3.7 – Définition d'un attribut Python classique pour la classe SPML `Region`

Maintenant, si l'on veut ajouter l'attribut SPML `name` à cette même classe, il faut lui ajouter une information indiquant au métamodèle que l'attribut possède un comportement SPML. Pour cela, il doit être déclaré sous la forme `spml_name` et un dictionnaire de contrôle `spml_name_controls__` doit lui être associé (exemple 3.8).

```
# définition de la classe
class Region(spml_class):
  #
  # définition du comportement de la classe
  #
  spml_modelowner__ = modele_1
  spml_stereotype__ = "CONCRETE"
  #
  # définition de l'attribut Python 'attr'
  #
  attr = "un_attribut_Python"
  #
  # définition de l'attribut SPML 'name'
  #
  spml_name = None
```

```
16   spml_name_controls__ = {}
```

Exemple 3.8 – Définition d'un attribut SPML pour la classe SPML `Region`

Le dictionnaire de contrôle permet de définir le comportement de l'attribut SPML correspondant. Les attributs SPML sont détaillés dans la partie 3.5.4.2 du module spml_utility.

3.5.4 Le module spml_utility

Ce module regroupe la définition des collections avec leurs bornes et des attributs et méthodes des classes SPML.

3.5.4.1 Les collections et les bornes

Une collection définit un type de liste d'objets. Il y a deux genres de collections possibles :

- `<type 'spml_arrayCollection'>` définit une collection à taille fixe,
- `<type 'spml_variableSizeCollection'>` définit une collection à taille variable (`set`, `list` ou `bag`).

Les collection sont définies par des bornes inférieure et supérieure. Ces bornes sont des instances de `<type 'spml_bound'>`. Pour une collection à taille fixe, les bornes sont nécessairement de type `<type 'spml_integerBound'>` (c'est-à-dire des entiers). Pour une collection à taille variable, elles peuvent être de type `<type 'spml_integerBound'>` ou `<type 'spml_populationDependentBound'>` (c'est-à-dire flottante).

L'exemple 3.9 montre la syntaxe pour créer une collection de type liste avec une borne inférieure nulle et une borne supérieure flottante.

```
1  maListe = spml_listCollection(lowerBound=spml_integerBound(
       boundValue=0),
2                                upperBound=
                                    spml_populationDependentBound())
```

Exemple 3.9 – Syntaxe pour la création d'une collection de type liste

La figure 3.17 présente l'organisation de ces classes.

3.5.4.2 Les attributs et méthodes

Comme nous l'avons déjà dit, les attributs des classes SPML sont typés. Ce module propose donc des types permettant de représenter les attributs et les méthodes des classes SPML. La figure 3.18 présente l'organisation de ces types.

Ces types sont organisés en trois grands groupes :

- les types représentant les champs de valeur inverses (de type simple ou collection),
- les types représentant les commandes (fonction, macro, méthode et procédure),
- les types représentant les attributs SPML (de type simple ou collection). Parmi ceux-ci, on distingue :
  - les types représentant les arguments,
  - les types représentant les champs de valeur.

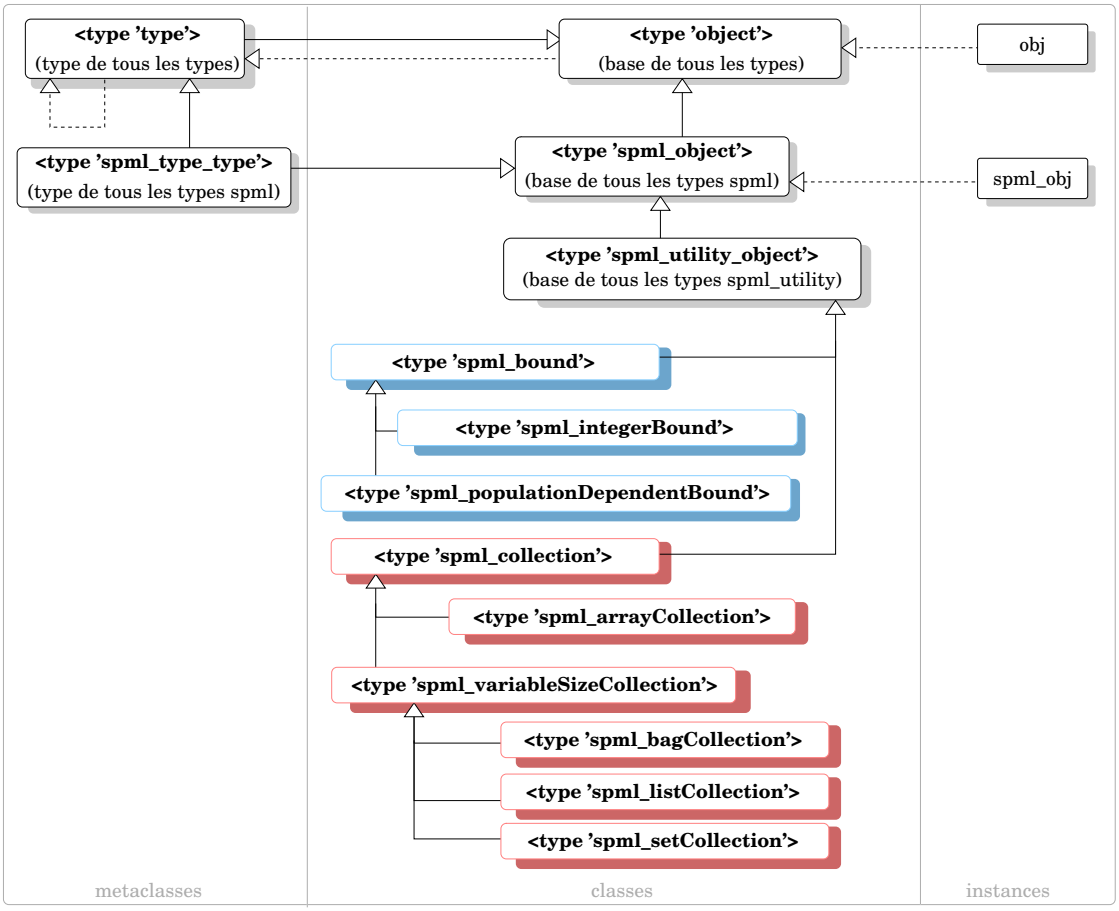


FIG. 3.17 – Organisation des types du module spml_utility : collections et bornes.

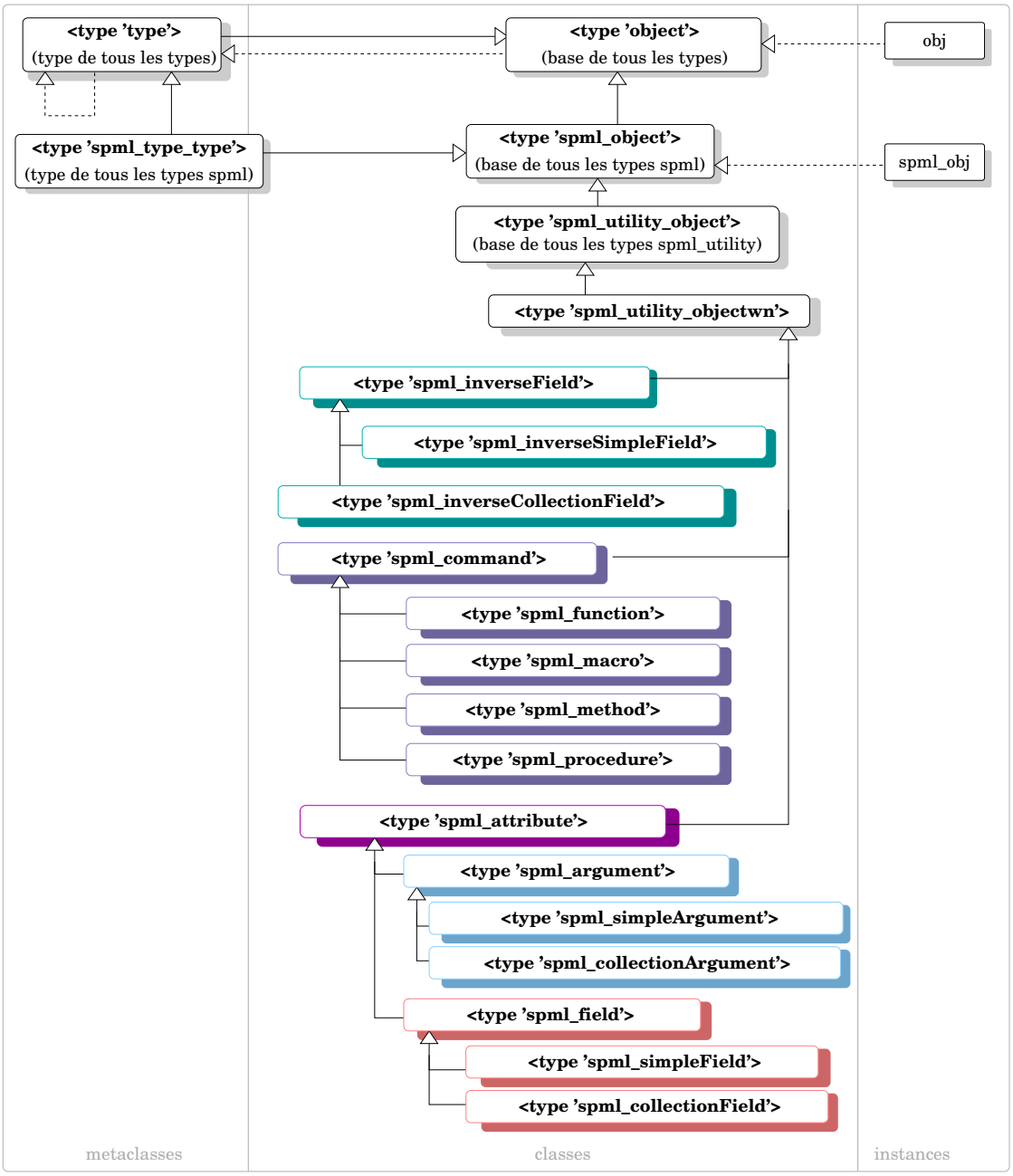


FIG. 3.18 – Organisation des types du module spml_utility : attributs et méthodes

Comme nous l'avons vu dans le chapitre 3.5.3.2 traitant des classes SPML, les attributs SPML possèdent certaines caractéristiques leur permettant de se différencier des attributs classiques :

- Un attribut SPML représente les données d'une entité physique, elle-même représentée par sa classes SPML. Un attribut donné appartient à une seule classe et ne peut pas être partagé avec d'autres.
- Le type relié (*relatedType*) des attributs leur permet de référencer une structure de données précise.
- Les attributs sont de type simple ou collection. Dans ce dernier cas, un attribut représente la relation entre un objet et une collection d'objets.
- Un attribut SPML possède un mode de définition parmi les suivants :

  FORCED : l'utilisateur doit affecter une valeur à l'attribut pour que l'objet soit valide.

  FINAL : la valeur donnée par l'utilisateur ne peut plus être changée une fois affectée à l'attribut. Ce mode est utile pour un identifiant.

  OPTIONAL : l'utilisateur n'est pas obligé d'affecter une valeur à l'attribut. Si aucune valeur n'est donnée, une valeur par défaut est affectée.

  DERIVED : la valeur de cet attribut n'est pas donnée directement par l'utilisateur. Elle est le résultat d'un processus d'évaluation interne. Par exemple, si un utilisateur définit un cercle à partir de trois points, le programme pourrait aussi avoir besoin des coordonnées du cercle pour réaliser des calculs. Le centre du cercle est lors un attribut dérivé.

  INTERNAL : un attribut interne ne reçoit pas de valeur de la part de l'utilisateur et n'est pas utilisé dans le programme, sauf pour du traitement interne.

- Les attributs sont utilisés pour définir des relations. Ils sont alors classés dans l'une des quatre catégories de relations suivantes (classement conforme au formalisme UML) :

  ASSOCIATION : c'est la relation par défaut quand aucune autre relation n'est utilisée. Elle associe une instance de classe à une ou plusieurs instances d'une autre classe.

  AGGREGATION : c'est une relation du genre *'fait partie de'* ou *'est composé de'*. Il n'y a cependant pas de relation d'existence entre les objets composants et l'objet composé.

  COMPOSITION : c'est une relation d'agrégation dans laquelle, cette fois-ci, il y a une relation d'existence entre les objets composants et l'objet composé. Les premiers sont comme des objets privés, internes à l'objet plus complexe. Dit autrement, la relation inverse a une cardinalité maximum et minimum de 1 [4].

  IDENTIFICATION : un attribut définissant une relation d'identification est unique et peut être considéré comme un index.

[4] La relation entre un objet et un type intrinsèque ou une énumération est considérée comme une composition. Dans ce cas, une valeur par défaut peut être donnée. Dans les cas autres qu'avec un type intrinsèque ou une énumération, aucune valeur par défaut n'est utilisée.

– Les attributs possèdent un mode d'évaluation qui peut être utilisé dans le cas d'un processus d'évaluation. Ce dernier peut être réalisé pendant la création ou la modification d'une instance. La valeur du mode d'évaluation change la manière dont le processus d'évaluation est effectué.

NONE : l'édition de l'instance utilisée dans cet attribut ne requiert pas de nouvelle évaluation de l'instance possédant cet attribut.

BACK-PROPAGATION : l'édition de l'instance utilisée dans cet attribut implique une nouvelle évaluation de l'instance possédant cet attribut.

– Enfin, un attribut possède un mode de stockage (soit PERSISTENT, soit TRANSIENT) qui surcharge le mode de stockage de la classe.

Ces caractéristiques sont précisées dans le dictionnaire de contrôle qui est associé à chaque attribut SPML. Si nous reprenons l'exemple 3.8 de la définition de la classe SPML `Region` (page 42) en spécifiant que l'attribut SPML `name` est de type `Identification`, nous obtenons le code de l'exemple 3.10.

```
# définition de la classe
class Region(spml_class):
  #
  # définition du comportement de la classe
  #
  spml_modelowner__ = modele_1
  spml_stereotype__ = "CONCRETE"
  #
  # définition de l'attribut Python 'attr'
  #
  attr = "un_attribut_Python"
  #
  # définition de l'attribut SPML 'name'
  #
  spml_name = None
  spml_name_controls__ = {"relatedType":Identification,
                          "stereotype":"IDENTIFICATION"}
```

Exemple 3.10 – Définition d'un attribut SPML par son dictionnaire de contrôle

3.5.5 Le module spml_ui

Ce module permet de définir les informations utilisateurs associés aux différentes entités. Il permet ainsi de préciser et de rendre visible de manière compréhensible des informations à propos des objets et de leurs champs pour l'utilisateur final. Par exemple, l'intégrateur d'un modèle thermique pourra préciser l'unité de température requit pour un champ représentant la température ambiante. Cette information sera rattachée au champ de l'objet et accessible grâce à une méthode.

Le module est divisé en trois parties : la définition des contextes et profils utilisateurs (§3.5.5.1), les filtres (§3.5.5.2) et les informations utilisateurs (§3.5.5.3).

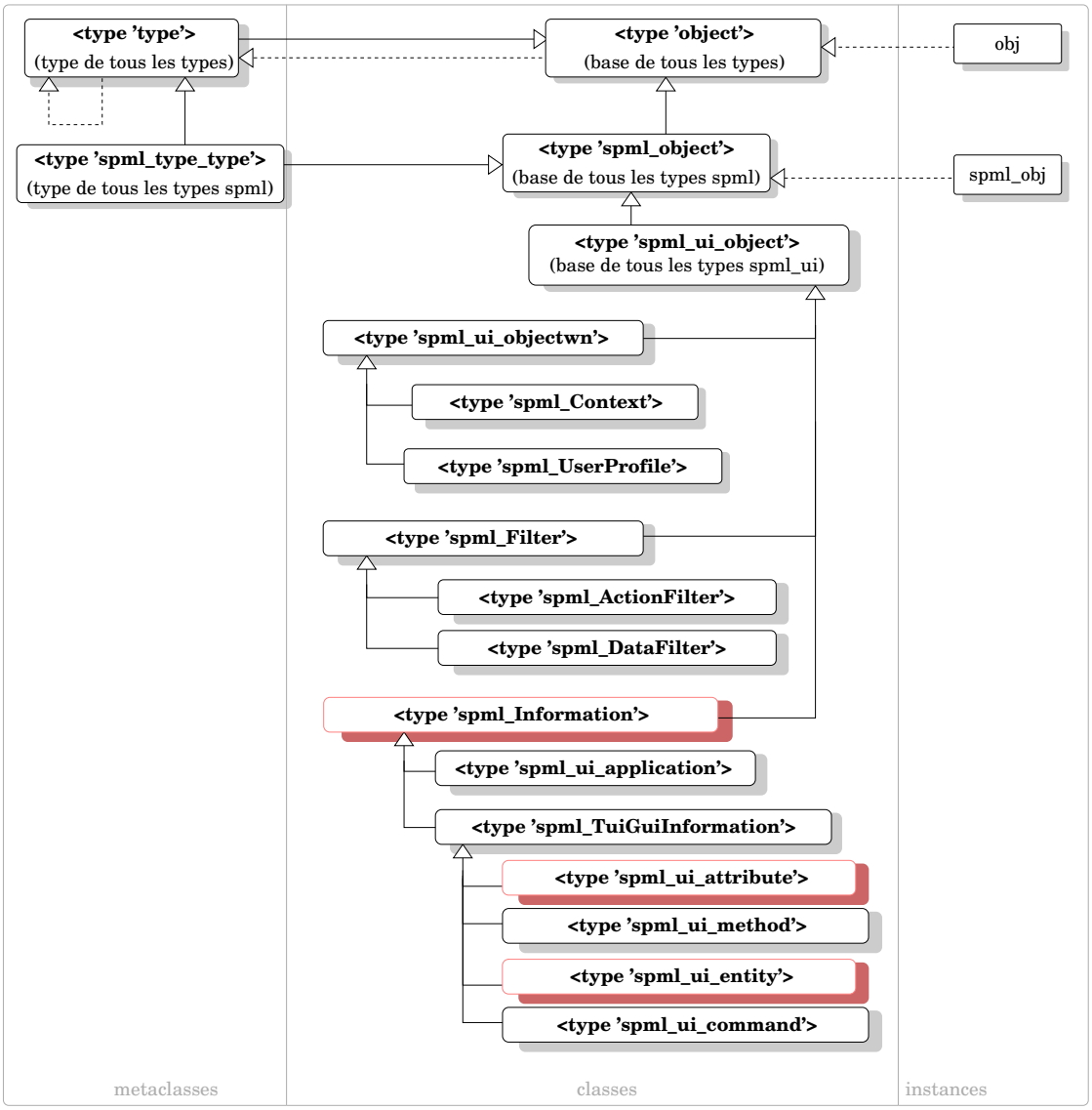


FIG. 3.19 – Organisation des types du module spml_ui

3.5.5.1 Les contextes et profils utilisateurs

Les types `<type 'spml_Context'>` et `<type 'spml_UserProfile'>` héritent tous les deux du type `<type 'spml_ui_objectwn'>`, ce qui signifie que leurs instances possèdent un identifiant unique.

Un contexte permet de définir une vue particulière pour l'application en cours. Par exemple, on peut imaginer créer dans une application autant de contextes que de modèles de données. Pour chaque contexte, on affichera par exemple sous forme arborescente toutes les instances des classes de ce modèle uniquement. Le type `<type 'spml_Context'>` possède les attributs suivants :

- `id` : l'identifiant unique,
- `defaultLabel` : une étiquette,
- `additionalLabels` : une liste d'étiquettes supplémentaires,
- `localHistory` : une liste d'historiques,
- `underlyingModel` : un modèle d'application.

La définition de profils utilisateurs permet éventuellement ensuite de personnaliser une application en fonction de ces profils. Le type `<type 'spml_UserProfile'>` possède les attributs suivants :

- `id` : l'identifiant unique,
- `defaultLabel` : une étiquette,
- `additionalLabels` : une liste d'étiquettes supplémentaires,
- `localHistory` : une liste d'historiques,

3.5.5.2 Les filtres

Les types `<type 'spml_Filter'>` sont destinés à appliquer des filtres de visibilité de données ou d'actions possibles sur des objets SPML. Ces types sont d'ailleurs liés à un contexte et éventuellement à un profil d'utilisateur. Selon le contexte, on peut ainsi décider d'afficher ou pas des actions ou des données selon que l'on autorise ou non leur utilisation. Les types `<type 'spml_Filter'>` possèdent donc les attributs suivants :

- `contextForCustomisation` : le contexte associé au filtre,
- `profileOfUsers` : une liste de profils d'utilisateurs.

On distingue deux catégories de filtres : le filtre de données (`<type 'spml_DataFilter'>`) et le filtre d'actions (`<type 'spml_ActionFilter'>`). Ces deux types possèdent les attributs supplémentaires suivants :

- `rightOfUsers` : le type d'actions possible avec ce filtre (aucun droit, lecture seule, exécution, ...).

3.5.5.3 Les informations utilisateur

Les types permettant de définir les informations utilisateurs sont les plus importantes du module. Elles permettent de préciser, par exemple dans des boîtes de dialogue graphique, les définitions des différentes classes et de leurs attributs dans les modèles de données SPML.

Tous ces types, par héritage de `<type 'spml_information'>`, sont composés des attributs suivants :

- `defaultLabel` : une étiquette,

- `defaultComment` : un commentaire,
- `defaultTooltip` : le texte d'une bulle d'aide,
- `additionalLabels` : une liste d'étiquettes supplémentaires,
- `additionalComment` : une liste de commentaires supplémentaires,
- `additionalTooltip` : une liste de bulles d'aide supplémentaires.

Le type `spml_ui_application` permet de spécifier les informations utilisateurs pour l'application. Il possède les attributs spécifiques suivants :
- `contexts` : une liste de d'instances de `<type 'spml_Context'>`,
- `profiles` : une liste de d'instances de `<type 'spml_UserProfile'>`,
- `userManual` : une adresse réseau vers un manuel d'utilisation,
- `tutorial` : une adresse réseau vers un tutorial.

Tous les type héritant de `<type 'spml_TuiGuiInformation'>` possèdent d'autres attributs :
- `commandName` : nom de la commande correspondante,
- `commandShortName` : nom abrégé de la commande correspondante,
- `category` : catégorie correspondante,
- `helpPageUrl` : une adresse réseau vers une page d'aide,
- `iconUrl` : une adresse réseau vers un icône,
- `iconGroupName` : un nom de groupe d'icône.

Parmi ces derniers types, nous distinguons :
- `<type 'spml_ui_attribute'>` : informations utilisateurs associé à un attribut SPML ; ce type possède les attributs supplémentaires suivants :
  - `contextFiltering` : liste de filtres de données (<type 'spml_DataFilter'>),
  - `reentrantMode` : réutilisation ou non de la valeur saisie pour le prochain attribut SPML,
  - `itemCount` : un entier.

- `<type 'spml_ui_method'>` : informations utilisateurs associé à une méthode SPML ; ce type possède les attributs supplémentaires suivants :
  - `contextFiltering` : liste de filtres d'actions (<type 'spml_ActionFilter'>),
  - `reentrantMode` : réutilisation ou non de la valeur saisie pour la prochaine méthode SPML.

- `<type 'spml_ui_entity'>` : informations utilisateurs associé à une classe SPML ; ce type possède les attributs supplémentaires suivants :
  - `contextFiltering` : liste de filtres de données (<type 'spml_DataFilter'>),
  - `inputMode` : liste de modes d'entrée,
  - `reentrantMode` : réutilisation ou non de la valeur saisie pour la prochaine classe SPML,
  - `itemCount` : un entier.

- `<type 'spml_ui_command'>` : informations utilisateurs associé à une commande SPML ; ce type possède les attributs supplémentaires suivants :
  - `contextFiltering` : liste de filtres d'actions (<type 'spml_ActionFilter'>).

Ainsi l'exemple 3.11 reprend celui de la définition de la classe SPML `Region` (Exemple

3.10 page 47) en ajoutant un nouvel attribut SPML *comment* de type `str`, optionnel et avec des informations utilisateurs.

```
# définition de la classe
class Region(spml_class):
  #
  # définition du comportement de la classe
  #
  spml_modelowner__ = modele_1
  spml_stereotype__ = "CONCRETE"
  #
  # définition de l'attribut Python 'attr'
  #
  attr = "un_attribut_Python"
  #
  # définition de l'attribut SPML 'name'
  #
  spml_name = None
  spml_name_controls__ = {"relatedType":Identification,
                          "stereotype":"IDENTIFICATION"}
  #
  # définition de l'attribut SPML 'comment'
  #
  spml_comment = None
  spml_comment_controls__ = {"relatedType":str,
                         "definitionMode":"OPTIONAL",
                         "uiInformation":spml_ui_attribute(
                           defaultLabel="Commentaire",
                           defaultComment= "Ce commentaire est
                              optionnel")}
```

Exemple 3.11 – Définition d'informations utilisateur sur un attribut SPML

3.6 Application à un problème thermique

Nous avons expliqué la structure du langage SPML, avec ses spécificités destinées à la description de modèles de données physiques. Nous allons maintenant montrer comment ce langage peut être utilisé dans le cadre d'un exemple simple de problème thermique. Nous partons pour cela de l'étude du modèle de données thermique. Ce modèle de données est ensuite traduit dans une représentation orientée objet afin de pouvoir ensuite le représenter avec le langage SPML. Cette représentation objet ressemble à celle d'un diagramme de classe UML, avec la différence que nous ajoutons des informations sur les classes qui ne sont pas reconnues par l'UML, comme par exemple des informations utilisateurs sur une classe ou un attribut SPML.

3.6.1 Représentation objet du modèle thermique

Le modèle de données du solveur thermique est assez simple. En effet nous considérons que le matériau supraconducteur présente des caractéristiques thermiques linéaires et isotropes. Nous proposons alors de définir un objet `Region` portant ces informations physiques, ainsi qu'un lien vers la région géométrique correspondante autravers d'un objet

de type `GeomReference`. Nous définissons aussi un objet `BoundaryCondition` permettant de définir les conditions aux limite de notre problème. Ces conditions aux limite sont séparées en deux types : `Adiabatic` et `Isotherm`. En considérant qu'un jeu de données est composé de régions thermiques et de conditions limites, nous obtenons le diagramme de classe de la figure 3.20.

Tous ces objets possèdent des informations utilisateurs, et certains attributs possèdent en plus des contraintes : les températures d'une région et d'une condition isotherme ne peuvent pas être négatives.

Enfin, le diagramme fait apparaître trois types de données intrinsèques :
- Identification : une simple chaîne de caractères, utilisée comme nom de l'objet,
- DataString : une simple chaîne de caractères,
- DataFloat : un simple réel.

3.6.2 Description SPML du modèle

La description de ce modèle en langage SPML comporte troix parties distinctes :

1. un préambule, dans lequel sont définis les modèles utilisés, le modèle d'application et l'application,
2. la description des types intrinsèques,
3. la description des classes du modèle.

Chacune de ces parties sera détaillée, en accord avec le diagramme de la figure 3.20 qui représente les classes du modèle, leurs attributs avec leur type et éventuellement une contrainte, et les informations utilisateurs (UI) des classes et attributs.

3.6.2.1 Le préambule

Les structures des classes du préambule sont détaillées dans le paragraphe §3.5.2.

```
1 basicThermicModel = spml_model(id="BasicThermicModel")

3 appli = spml_application(id"BasicThermicProblem",
4                          version="v1.0",
5                          models=spml_applicationmodel(
6                                    id="monApplicationModel",
7                                    version="v1.0",
8                                    models=[basicThermicModel])
```

Exemple 3.12 – Modèle SPML thermique simple : préambule

L'exemple 3.12 définit un modèle de données appelé *`BasicThermicModel`* et une application utilisant ce modèle. Toutes les classes SPML définies par la ensuite appartiendront à ce modèle.

3.6.2.2 Les types intrinsèques

Les types intrinsèques du modèle sont : `<type 'Identification'>`, `<type 'DataString'>`, `<type 'DataFloat'>` et `<type 'GeomReference'>`. La structure d'un type intrinsèque est détaillée dans le paragraphe §3.5.3.1. L'exemple 3.13 montre le code pour la déclaration de ces types intrinsèques.

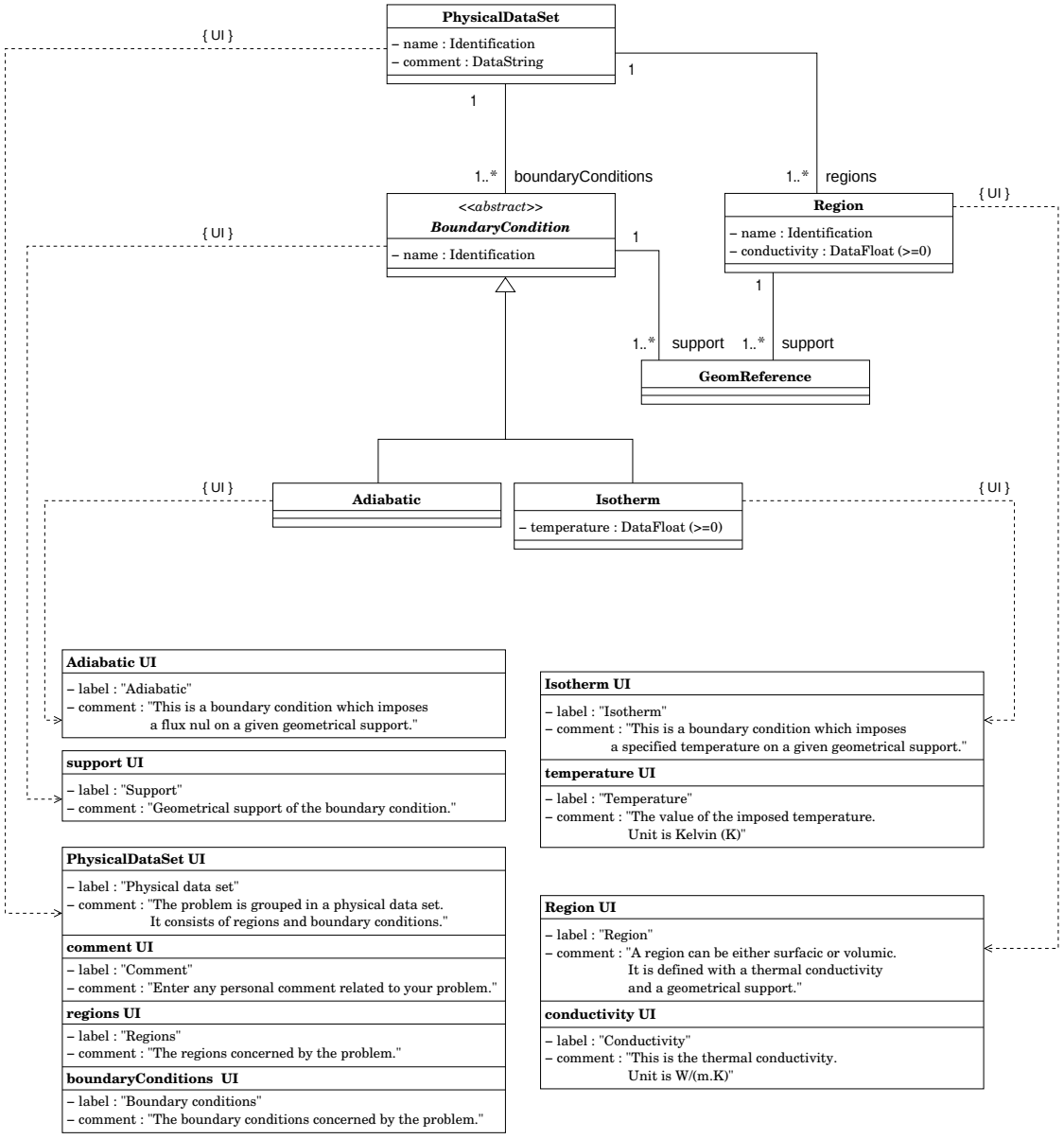


FIG. 3.20 – Représentation objet d'un modèle de données thermique simple

```
9  #----------------------------------------
10 # Types Intrinsèques
11 #----------------------------------------
12 class Identification(spml_str):
13   spml_type__ = "STRING"
14   spml_modelowner__ = basicThermicModel

16 class DataString(spml_str):
17   spml_type__ = "STRING"
18   spml_modelowner__ = basicThermicModel

20 class DataFloat(spml_float):
21   spml_type__ = "REAL"
22   spml_modelowner__ = basicThermicModel

24 class GeomReference(spml_salomeRef):
25   spml_modelowner__ = basicThermicModel
```

Exemple 3.13 – Modèle SPML thermique simple : types intrinsèques

Pour obtenir une référence vers une entité extérieure (ici une géométrie), le méta-modèle a été étendu et propose le type `<type 'spml_salomeRef'>`. Ce type permet d'encapsuler un objet extérieur disponible sur un bus de partage de données (ex : CORBA).

3.6.2.3 Les classes

Comme indiqué en §3.5.3.2, pour définir une classe comme étant SPML, elle doit hériter du type `<type 'spml_class'>` et des attributs de classe obligatoires doivent être renseignés. Notre modèle compte cinq classes : `PhysicalDataSet`, `Region`, `BoundaryCondition`, `Adiabatic`, `Isotherm`.

1. La classe `PhysicalDataSet` :

```
26 #----------------------------------------
27 # Classe PhysicalDataSet
28 #----------------------------------------
29 class PhysicalDataSet(spml_class):
30   #
31   # définition du comportement de la classe
32   #
33   spml_modelowner__ = basicThermicModel
34   spml_stereotype__ = "CONCRETE"
35   spml_uicontrols__ = spml_ui_entity(
36     defaultLabel = "Physical data set",
37     defaultComment = "The problem is grouped in a physical data "+
38                      "set. It consists of regions and boundary "+
39                      "conditions.")
```

Exemple 3.14 – Modèle SPML thermique simple : la classe `PhysicalDataSet` (1/5)

Les informations utilisateur de la classe `PhysicalDataSet` sont affectés à l'attribut `spml_uicontrols__` par l'intermédiaire d'une instance du type `spml_ui_entity`,

conformément au paragraphe §3.5.5.3.

```
40  #
41  # définition de l'attribut SPML 'name'
42  #
43  spml_name = None
44  spml_name_controls__ = {"relatedType":Identification,
45                          "stereotype":"IDENTIFICATION"}
```

Exemple 3.15 – Modèle SPML thermique simple : la classe `PhysicalDataSet` (2/5)

Nous définisson ici un attribut SPML : son nom commence par `spml_`. Par la suite, l'attribut SPML sera accessible par `PhysicalDataSet.`*`name`* et non pas `PhysicalDataSet.`*`spml_name`*. Le dictionnaire de contrôle `spml_name_controls__` lui est associé. Ce dernier précise le comportement et les caractéristiques de l'attribut.

La valeur par défaut de l'attribut `name` est `None`, c'est-à-dire qu'il est vide. Si une valeur est affectée puis supprimée, la nouvelle valeur de l'attribut sera celle par défaut.

L'attribut est de type `<type 'Identification'>`; ceci est réalisé en associant ce type à la clé `"relatedType"` de son dictionnaire de contrôle. Si la valeur de l'attribut n'est pas compatible avec son type, alors l'attribut est non valide.

```
46  #
47  # définition de l'attribut SPML 'comment'
48  #
49  spml_comment = None
50  spml_comment_controls__ = {"relatedType":DataString,
51    "definitionMode""OPTIONAL",
52    "uiInformation":spml_ui_attribute(
53      defaultLabel="Comment",
54      defaultComment="Enter any personal comment"+
55                     "related to your problem.")}
```

Exemple 3.16 – Modèle SPML thermique simple : la classe `PhysicalDataSet` (3/5)

L'attribut `comment` est de type `<type 'DataString'>`. Il est définit comme étant optionnel : même vide, il est valide. Les informations utilisateur de l'attribut `comment` sont affectés à la clé `"uiInformation"` de son dictionnaire de contrôle par l'intermédiaire d'une instance du type `spml_ui_attribute`, conformément au paragraphe §3.5.5.3.

```
56  #
57  # définition de l'attribut SPML 'regions'
58  #
59  spml_regions = None
60  spml_regions_controls__ = {
61    "relatedType":Region,
62    "stereotype":"ASSOCIATION",
63    "listMode":"CF",
```

```
64      "typeOfCollection":spml_listCollection(
65        lowerBound=spml_integerBound(boundValue=0),
66        upperBound=spml_populationDependentBound()),
67      "uiInformation":spml_ui_attribute(
68        defaultLabel="Regions",
69        defaultComment= "The regions concerned by the problem.")}
```

Exemple 3.17 – Modèle SPML thermique simple : la classe `PhysicalDataSet` (4/5)

L'attribut `regions` est de type `<type 'Region'>`. La relation entre l'attribut `regions` et sa valeur est définie comme étant une association. Si elle n'est précisée, la valeur par défaut de cette relation est une composition. L'attribut `regions` est défini comme étant de type *liste*. Le type de liste est précisé : c'est une collection de bornes $[0; \infty[$.

```
70    #
71    # définition de l'attribut SPML 'boundaryConditions'
72    #
73    spml_boundaryConditions = None
74    spml_boundaryConditions_controls__ = {
75      "relatedType":BoundaryCondition,
76      "stereotype":"ASSOCIATION",
77      "listMode":"CF",
78      "typeOfCollection":spml_listCollection(
79        lowerBound=spml_integerBound(boundValue=0),
80        upperBound=spml_populationDependentBound()),
81      "uiInformation":spml_ui_attribute(
82        defaultLabel="Boundary conditions",
83        defaultComment="The boundary conditions concerned"+
84                       " by the problem.")}
85  #----------------------------------------
86  # Fin classe PhysicalDataSet
87  #----------------------------------------
```

Exemple 3.18 – Modèle SPML thermique simple : la classe `PhysicalDataSet` (5/5)

Les caractéristiques de l'attribut `boundaryConditions` sont identiques à celles de `regions`, à part le type qui est `<type 'BoundaryCondition'>` dans ce cas.

2. La classe `Region` :

```
88  #----------------------------------------
89  # Classe Region
90  #----------------------------------------
91  class Region(spml_class):
92    #
93    # définition du comportement de la classe
94    #
95    spml_modelowner__ = basicThermicModel
96    spml_stereotype__ = "CONCRETE"
97    spml_uicontrols__ = spml_ui_entity(
98      defaultLabel = "Physical Regions",
```

```
99      defaultComment ="A region can be either surfacic or volumic."+
100                     "It is defined with a thermal conductivity "+
101                     "and a geometrical support.")
102   #
103   # définition de l'attribut SPML 'name'
104   #
105   spml_name = None
106   spml_name_controls__ = {"relatedType":Identification,
107                           "stereotype":"IDENTIFICATION"}
108   #
109   # définition de l'attribut SPML 'conductivity'
110   #
111   spml_conductivity = 80.2
112   spml_conductivity_controls__ = {
113     "relatedType":DataFloat,
114     "uiInformation":spml_ui_attribute(
115       defaultLabel="Conductivity",
116       defaultComment="This is the thermal conductivity."+
117                      "Unit is W/(m.K)")}
118   # méthode appelée avant d'attribuer une valeur
119   # à l'attribut SPML 'conductivity'
120   def spml_conductivity_bscontrols__(self,value):
121       if value<0:
122           print "value of conductivity should be positive"
123           raise spml_error, "value of conductivity should be > 0"
124       #return the same value
125       return value
126   # méthode appelée après l'attribution de la valeur
127   # à l'attribut SPML 'conductivity'
128   def spml_conductivity_ascontrols__(self,value):
129       pass
```

Exemple 3.19 – Modèle SPML thermique simple : la classe `Region` (1/2)

Nous définissons une méthode permettant d'affiner le comportement du modèle, comme indiqué dans le paragraphe des services §3.3.6 : `spml_conductivity_bscontrols__`. Cette méthode est appelée avant l'affectation d'une nouvelle valeur à l'attribut `conductivity` et après vérification de la validité de cette valeur. Dans notre cas, elle test la valeur de l'attribut conformément à sa contrainte : si la valeur est négative, une exception `spml_error` est levée et la valeur n'est pas affectée à l'attribut ; dans le cas contraire (valeur positive ou nulle), la valeur est affectée à l'attribut.

De la même manière, nous pouvons définir une 2e méthode qui sera cette fois-ci appelée après l'affectation de la valeur à l'attribut `conductivity` : `spml_conductivity_ascontrols__`. Ce mécanisme permet par exemple de réaliser des calculs, ou de définir la valeur d'un attribut dérivé. Dans notre cas, nous ne faisons rien.

```
130   #
131   # définition de l'attribut SPML 'support'
132   #
133   spml_support = None
```

```
134   spml_support_controls__ = {
135     "relatedType":GeomReference,
136     "stereotype":"ASSOCIATION",
137     "listMode":"CF",
138     "typeOfCollection":spml_listCollection(
139       lowerBound=spml_integerBound(boundValue=0),
140       upperBound=spml_populationDependentBound()),
141     "uiInformation":spml_ui_attribute(
142       defaultLabel="Support",
143       defaultComment= "Geometrical support of the region.")}
144   #
145   # definition of classic attributes (non SPML)
146   #
147   not_spml_var = "not_spml_var"
148   def not_spml_func(self):
149       print "This is not an spml function"
150 #----------------------------------------
151 # Fin classe Region
152 #----------------------------------------
```

Exemple 3.20 – Modèle SPML thermique simple : la classe `Region` (2/2)

Nous montrons dans l'exemple 3.20 que nous pouvons tout à fait définir des attributs et méthodes classiques, c'est-à-dire que leur nom ne commence pas par `spml_` et qu'ils n'ont pas de dictionnaire de contrôle. Ces attributs seront pris en charge par le langage sur lequel le SPML se base (c'est-à-dire Python). Ce mécanisme est donc gratuit pour le SPML etpermet par exemple de définir des variables de calcul internes.

3. La classe `BoundaryCondition` :

```
153 #----------------------------------------
154 # Classe BoundaryCondition
155 #----------------------------------------
156 class BoundaryCondition(spml_class):
157   #
158   # définition du comportement de la classe
159   #
160   spml_modelowner__ = basicThermicModel
161   spml_stereotype__ = "ABSTRACT"
```

Exemple 3.21 – Modèle SPML thermique simple : la classe `BoundaryCondition` (1/2)

La classe `BoundaryCondition` est définie comme étant abstraite : elle ne peut donc pas être instanciée. Par contre elle définie une structure commune à ses classes dérivées.

```
162   #
163   # définition de l'attribut SPML 'name'
164   #
165   spml_name = None
166   spml_name_controls__ = {"relatedType":Identification,
```

```
                          "stereotype":"IDENTIFICATION"}
  #
  # définition de l'attribut SPML 'support'
  #
  spml_support = None
  spml_support_controls__ = {}
  spml_support_controls__["relatedType"]=GeomReference
  spml_support_controls__["stereotype"]="ASSOCIATION"
  spml_support_controls__["listMode"]="CF"
  spml_support_controls__["typeOfCollection"]=spml_listCollection(
    lowerBound=spml_integerBound(boundValue=0),
    upperBound=spml_populationDependentBound())
  spml_support_controls__["uiInformation"]=spml_ui_attribute(
    defaultLabel="Support",
    defaultComment="Geometrical support of "+
                   "the boundary condition."
#----------------------------------------
# Fin classe BoundaryCondition
#----------------------------------------
```

Exemple 3.22 – Modèle SPML thermique simple : la classe `BoundaryCondition` (2/2)

Nous montrons dans l'exemple 3.22 une autre méthode pour remplir un dictionnaire de contrôle, de la forme `dict[key]=value`.

4. La classe `Adiabatic` :

```
#----------------------------------------
# Classe Adiabatic
#----------------------------------------
class Adiabatic(BoundaryCondition):
  #
  # définition du comportement de la classe
  #
  ##spml_modelowner__ = basicThermicModel
  spml_stereotype__ = "CONCRETE"
  #
  # définition de l'attribut SPML 'name'
  #
  spml_name = None
  spml_name_controls__ = {"relatedType":Identification,
                          "stereotype":"IDENTIFICATION"}
  spml_uicontrols__ = spml_ui_entity(
    defaultLabel = "Adiabatic",
    defaultComment="This is a boundary condition "+
                   "which imposes thermal flux=0 "+
                   "on a given geometrical support.")
#----------------------------------------
# Fin classe Adiabatic
#----------------------------------------
```

Exemple 3.23 – Modèle SPML thermique simple : la classe `Adiabatic`

La classe `Adiabatic` hérite de la classe abstraite `BoundaryCondition`, c'est donc aussi une classe SPML et elle hérite de tous ses attributs, SPML ou non. La définition du modèle de la classe est commentée car ce modèle est le même que sa superclasse : il n'est donc pas nécessaire de le redéfinir. La classe est définie comme étant concrète. Elle peut ainsi être instanciée, contrairement à sa superclasse.

5. La classe `Isotherm` :

```
209 #-----------------------------------------
210 # Classe Isotherm
211 #-----------------------------------------
212 class Isotherm(BoundaryCondition):
213   #
214   # définition du comportement de la classe
215   #
216   ##spml_modelowner__ = basicThermicModel
217   spml_stereotype__ = "CONCRETE"
218   #
219   # définition de l'attribut SPML 'name'
220   #
221   spml_name = None
222   spml_name_controls__ = {"relatedType":Identification,
223                           "stereotype":"IDENTIFICATION"}
224   spml_uicontrols__ = spml_ui_entity(
225     defaultLabel = "Isotherm",
226     defaultComment="This is a boundary condition "+
227                    "which imposes a specified temperature "+
228                    "on a given geometrical support.")
229   #
230   # définition de l'attribut SPML 'temperature'
231   #
232   spml_temperature = None
233   spml_temperature_controls__ = {
234     "relatedType":DataFloat,
235     "uiInformation":spml_ui_attribute(
236       defaultLabel="Temperature",
237       defaultComment="The value of the imposed temperature."+
238                      "Unit is Kelvin (K)")}
239   # méthode appelée avant d'attribuer une valeur
240   # à l'attribut SPML 'temperature'
241   def spml_temperature_bscontrols__(self,value):
242     if value<0:
243       print "Value of temperature should be positive (K)"
244       raise spml_error,"Value of temperature should be > 0 (K)"
245     #return the same value
246     return value
247 #-----------------------------------------
248 # Fin classe Isotherm
249 #-----------------------------------------
```

Exemple 3.24 – Modèle SPML thermique simple : la classe `Isotherm`

Nous défissons la 2e condition limite `Isotherm` en héritant une fois encore de `BoundaryCondition`. Une méthode permettant d'affiner le comportement du

modèle est de nouveau proposée. Cette méthode teste la valeur de l'attribut `temperature` conformément à sa contrainte : si la valeur est négative, une exception `spml_error` est levée et la valeur n'est pas affectée à l'attribut ; dans le cas contraire (valeur positive ou nulle), la valeur est affectée à l'attribut.

3.7 Conclusion

Le formalisme de description de modèles de données — le métamodèle — est concrétisé sous la forme d'un langage informatique : le SPML, Standard Physic Modeling Language. Ce langage permet de décrire les structures des modèles de données de solveurs, c'est-à-dire l'organisation des données physiques et les relations qui les lient.

Nous avons décidé de définir le langage SPML comme un langage orienté objet. Cependant étant donné que beaucoup de langages permettent de répondre à une majorité des besoins, nous avons choisi de nous baser sur l'un d'entre eux afin de minimiser les coûts de réalisation du langage. Le SPML ajoutera des spécificités supplémentaires permettant entre autres de définir le comportement des objets vis à vis des interactions avec l'utilisateur, mais aussi de définir des données typées ou non-typées.

Ce langage sera associé à des outils et services permettant de manipuler les modèles et les données physiques. Le cahier des charges des fonctionnalités de ce modèle est le suivant :

- un interprète de commandes, pour l'exécution et la vérification de la syntaxe des instructions SPML,
- un système de gestion des modèles et des données, pour la vérification de la cohérence des données par exemple,
- un outil permettant l'intéraction entre les données et un environnement extérieur, par exemple le solveur sur lequel se base le modèle, ou une interface graphique,
- une mise à disposition des données pour d'autres applications à travers une interface réseau,
- la possibilité d'affiner le comportement des modèles pour les rendre plus efficaces.

Les concepts fondamentaux de la programmation orientée objet, sur lesquels est basée la structure du langage SPML, ont été donné. Nous avons alors définit les objets de base à partir desquels sont définis tous les autres objets, mais aussi les mécanismes permettant de créer de nouveaux objets ou encore de créer des représentants de tels objets, c'est-à-dire des instances.

Ensuite nous avons détaillé la grammaire de notre langage de manière pragmatique. Celui-ci est ainsi structuré en quatre modules dont l'architecture suit le schéma traditionnel à quatre couches : méta-métamodèle, métamodèle, modèle et objets/données. L'originalité de cette structure réside dans le module dédié à l'interface utilisateur qui permet de prendre en compte, dans un modèle de données, les intéractions avec un environnement extérieur.

Finalement, l'exemple d'un modèle décrit avec le SPML a permit de rendre ce langage plus concret, mais aussi d'expliquer la structure type d'un modèle :

- le préambule, qui définit le ou les modèles qui vont être décrit, mais aussi l'application qui contiendra ces modèles,
- les types de données intrinsèques particulières au modèle (références extérieurs, url, identifiants, ...),

– les classes, qui permettent de décrire au mieux la structure des données.

Le chapitre suivant est consacré à la mise en œuvre du langage dans lequel nous détaillons le fonctionnement des mécanismes du langage et des outils associés.

Chapitre 4

Mise en œuvre des concepts

Sommaire

4.1 Introduction

Après avoir présenté dans le chapitre 2 un exemple de problème numérique pour lequel nous avons montré comment l'utilisation du langage pouvait être utilisé pour la description des modèles de données physiques, nous avons détaillé la structure de ce langage dans le chapitre 3. Ce chapitre détaille la réalisation du langage et de ses fonctionnalités.

Après une brève justification du choix de Python comme langage de base pour le SPML, nous expliquerons dans le chapitre 4.2 comment nous avons étendu la sémantique et les fonctionnalités de Python pour obtenir le comportement désiré et ainsi réaliser la structure décrite dans le chapitre 3.5. Ensuite nous expliquerons comment ont été réalisées les fonctionnalités associées au langage : le chapitre 4.3 traitera de l'interprète, nous expliquerons la gestion des modèles et des données dans le chapitre 4.4, le chapitre 4.5 concernera la projection des données, finalement la mise à disposition sur un bus de partage des données sera détaillée dans le chapitre 4.6.

Toutes ces fonctionnalités réunies forment un moteur de description des propriétés physiques dont l'architecture générale est donnée avec le schéma E.1 en annexe E.

Remarque : les noms des types et supertypes du SPML donnés dans le chapitre 3 ont parfois été simplifiés pour des raisons de lisibilité. Ce chapitre utilisera les noms exacts tels qu'ils sont définis dans les fichiers sources (exemple : `spml_type_type` → `spml_type_type__`).

4.2 La sémantique SPML : mécanismes

Nous allons expliquer dans cette partie comment nous avons utilisé les mécanismes de la programmation orientée objet, et plus particulièrement ceux de Python, pour définir les objets SPML et obtenir le comportement défini et décrit dans le chapitre 3.5.

4.2.1 Python : un langage adapté

Nous avons besoin d'écrire un interprète qui soit orienté objet et qui nous permettent d'implémenter les concepts que nous avons évoqués précédemment. La ré-écriture complète d'un interpréte est une solution envisageable. Il existe des outils qui facilitent énormément ce travail (ex : lex, yacc [10], antlr [11]). Cependant le travail à réaliser reste conséquent. Nous avons donc analysé la solution qui consiste à repartir d'un langage existant et à compléter ce langage avec les concepts dont nous avons besoin et qui manquent. Pour cela il faut que le langage de départ réunisse plusieurs conditions. Qu'il soit orienté objet, possède déjà un interpréte et surtout soit très facilement extensible. Il existe de nombreux langage orienté objet tels que : Ada, CommonLisp, Ruby, Python, Scala, C++, Smalltalk, etc... Nous ne détaillerons pas ici tous les langages possibles mais ce qui a prévalu au choix de Python est la facilité d'étendre les concepts du langage pour élargir la sémantique de l'interprète (voir paragraphe §3.5)

4.2.2 Personnalisation du comportement des classes et types intrinsèques SPML

De la même manière qu'une classe est appelée par sa méthode `__call__` (ou encore avec des parenthèses `()`) pour créer une instance, une métaclasse est appelée par sa méthode `__call__` pour créer une classe. Cette métaclasse est considérée comme un supertype, et une classe devient alors un objet comme un autre. Par exemple, prenons la classe `X` suivante :

```
class X(object):
  a=10
```

Pour créer une instance de cette classe, il faut d'abord créer la classe elle-même. Pour cela l'interprète Python, dont le rôle est justement de réaliser toutes ces opérations, utilise le type de `X`. Le chapitre 3.4 explique que `type(X)` est recherché parmi les types des bases de la classe. La classe `X` ayant pour base `(object,)`, le type de `X` est le type de `object`, soit `<type 'type'>`.

Une fois le type de `X` trouvé, l'interprète fait appel à sa méthode `__call__` selon la syntaxe suivante : `type(`*`name, bases, dict`*`)`. La variable *`name`* correspond au nom de la classe à créer, *`bases`* à un tuple contenant les bases de la classe et *`dict`* à l'environnement d'exécution ayant été défini dans la classe (ie. un dictionnaire contenant les attributs et méthodes). L'instruction pour créer la classe `X` est donc : `type('X',(object,),{'a':10})`.

Notre objectif est de modifier le comportement des objets SPML afin qu'ils soient traités différement des objets Python classiques. En effet, le rôle des objets SPML étant de modéliser des structures de données physiques, ils subissent des contraintes spécifiques (tests de validité, contrôles de types, ...). Le module *spml_type* a été créé dans ce sens, et par exemple la métaclasse (le supertype) `<type 'spml_type_class__'>` permet de contrôller spécifiquement le comportement des classes SPML.

Pourquoi le supertype `<type 'spml_type_class__'>` ? Et bien parce que toute classe SPML doit hériter de `<type 'spml_class'>`, et que ce type est défini pour être créé à partir du supertype `<type 'spml_type_class__'>` grâce à l'utilisation de l'attribut `__metaclass__` (cf §3.4.4). La commande Python pour la création d'une classe SPML est alors : `spml_type_class__(name,bases,dict)`. Ce mécanisme est le même pour les autres types définis dans le métamodèle SPML.

La séquence normale lors de l'appel à la méthode `__call__` de `<type 'spml_type_class__'>` est donnée dans l'exemple 4.1.

```
class spml_type_class__(spml_type_object__):
  def __call__(cls,*args,**kw):
    # this function is called for creation
    # of an instance of spml_class
    inst = cls.__new__(cls,*args,**kw)
    assert inst.__class__ is cls
    cls.__init__(inst,*args,**kw)
    #return new instance
    return inst
```

Exemple 4.1 – Séquence normale d'une méthode `__call__`

La méthode `__new__` informe au supertype de créer un nouveau type ; la méthode `__init__` permet quant à elle de personnaliser la création et l'initialisation de ce type. C'est donc dans cette dernière que sont effectués les contrôles et tests pour obtenir le comportement désiré des types SPML selon un schéma de principe qui est le même pour tous les métatypes :

1. récupération de la valeur d'un attribut SPML ; si cet attribut n'existe pas une exception `AttributeError` est levée.
2. contrôle éventuel de la valeur ; si le contrôle échoue une exception `spml_error_` est levée.
3. si une exception a été levée, on crée l'attribut SPML avec une valeur par défaut.

Selon les cas, ce schéma peut être complété par d'autres vérifications ou modifications.

Nous détaillons cette personnalisation de comportement pour les différents métatypes du métamodèle SPML dans les sous-parties suivantes.

4.2.2.1 Personnalisation commune aux types SPML : `spml_type_type__.__init__`

Le supertype `spml_type_type__` est le type des classes et types intrinsèques SPML. C'est donc dans sa méthode `__init__` que sont réalisées les personnalisations communes à ces objets :

1. on vérifie que le nom de l'objet en création n'est pas interdit (c'est-à-dire n'est pas dans la liste des mots réservés, ou qu'il contient un espace),
2. on récupère le modèle de l'objet avec l'attribut `spml_modelowner__` :

   ```
   spml_modelowner__ = getattr(cls,"spml_modelowner__")
   ```

   L'objet récupéré est de type `spml_app_model__`.

   (a) si le modèle existe, on met à jour la liste des types du modèle en y ajoutant la classe en création :

```
spml_modelowner__.addTypes(cls)
```

(b) sinon, le modèle de la classe en création sera celui par défaut (pour l'instant : aucun)

4.2.2.2 Personnalisation lors de la création des types intrinsèques SPML : spml_type_intrinsic__.__init__

Dans la méthode `__init__` du supertype `spml_type_intrinsic__` on récupère la valeur de l'attribut `spml_type__`. L'objet récupéré est de type `string`. On vérifie que cette `string` est un des mots autorisés pour cet attribut : `'integer'`, `'real'`, `'numeric'`, `'string'`, `'numeric_or_string'`, `'boolean'`, `'void'`, `'INTEGER'`, `'REAL'`, `'NUMERIC'`, `'STRING'`, `'NUMERIC_OR_STRING'`, `'BOOLEAN'`, `'VOID'`. Si ce n'est pas le cas, on lui affecte la valeur par défaut `'string'`.

4.2.2.3 Personnalisation lors de la création des classes SPML : spml_type_class__.__init__

La personnalisation des classes SPML passe par les méthodes `__init__` des métatypes `spml_type_object__` et `spml_type_class__`. Dans le 1er cas sont traitées les méthodes SPML, alors que dans le 2e ce sont les attributs SPML.

Le principe de traitement est le suivant : pour chaque attribut ou méthode SPML, son dictionnaire de contrôle est analysé, puis une instance de `spml_utility_method__` ou `spml_utility_field__` selon le cas est créée avec les arguments du dictionnaire de contrôle. La classe SPML possèdera alors une liste de ces instances.

Analysons le traitement des méthodes et attributs SPML :

1. *Les méthodes SPML (`spml_type_object__.__init__(cls,name,bases,dict)`)* :

   Nous récupérons d'abord la liste des instances de `spml_utility_method__` en utilisant le dictionnaire `dict` passé en argument qui contient la liste des attributs et méthodes de la classe :

   ```
   cls.spml_methods__ = dict["spml_methods__"]
   ```

   Si cette clé `"spml_methods__"` n'existe pas alors nous initialisons `cls.spml_methods__` à une liste vide `[]` et nous cherchons des méthodes SPML. Pour cela nous analysons toutes les entrées du dictionnaire `dict` et cherchons les méthodes SPML qui ont été déclarées dans la classe : le nom commence par `spml_` (= attribut ou méthode SPML) et le type est `types.FunctionType` (= fonction attaché à une classe = méthode). Pour chaque méthode ainsi trouvée :
   - son nom est débarrassé du préfixe « `spml_` » (« `spml_maMethode` » devient ainsi « `maMethode` ») :

     ```
     name_end = name[5:]
     ```

   - les méthodes dont le nom se termine par « `__` » sont des méthodes particulières qui ne sont pas traitées ici,
   - le dictionnaire de contrôle associé à la méthode est récupéré (rappel : pour une méthode SPML « `spml_maMethode` », le dictionnaire de contrôle s'appelle « `spml_maMethode_controls__` ») » :

```
method_controls = getattr(cls,"spml_"+name_end+"_controls__"
    )
```

– si les informations d'interface utilisateur de la méthode existent on ne fait rien, sinon on les crée :

```
if method_controls.has_key("uiInformation"):
  pass
else:
  method_controls["uiInformation"] = spml_ui_method__(\
                                defaultLabel=name_end+"_UI")
```

– finalement, nous créons une instance de `spml_utility_method__` à partir des informations recueillies et nous l'ajoutons à la liste `cls.spml_methods__` :

```
newField = spml_utility_method__(id=name_end,
      uiInformation=method_controls["uiInformation"])
cls.spml_methods__.append(newField)
```

2. *Les attributs SPML (`spml_type_class__.__init__(cls,name,bases,dict)`)* :

Le traitement des attributs SPML débute avec ceux utilisés pour définir le comportement de la classe elle-même, selon le schéma expliqué plus haut (récupération, contrôle, éventuellement attribut forcé à une valeur par défaut – cf tableau en annexe A.1) :
– le stéréotype :

```
spml_stereotype__ = getattr(cls,"spml_stereotype__")
```

– le supertype :

```
cls.spml_supertype__ = bases[0]
```

– Les informations UI :

```
spml_uicontrols__ = getattr(cls,"spml_uicontrols__")
```

Si les informations UI ne sont pas données, elles sont créées :

```
cls.spml_uicontrols__ = spml_ui_entity__(
  defaultLabel = name,
  defaultComment= 'no comment available',
  defaultTooltip = '')
```

Ensuite nous parcourons le dictionnaire `dict` pour trouver les attributs ayant été déclarés SPML : le nom commence par `spml_` et le type n'est pas `types.FunctionType`. Pour chaque attribut ainsi trouvé :
– son nom est débarrassé du préfixe « `spml_` » (« `spml_monAttribut` » devient ainsi « `monAttribut` ») :

```
name_end = name[5:]
```

- les attributs dont le nom se termine par « `__` » sont des attributs particuliers qui ne sont pas traités ici,
- le dictionnaire de contrôle associé à l'attribut est récupéré (rappel : pour un attribut SPML « `spml_monAttribut` », le dictionnaire de contrôle s'appelle « `spml_monAttribut_controls__` ») :

```
field_controls = getattr(cls,"spml_"+name_end+"_controls__")
```

- l'existence des clés du dictionnaire de contrôle correspondantes aux éléments du tableau en annexe B.1 est vérifiée. Pour chaque clé inexistante, celle-ci est créée et lui est affectée la valeur par défaut correspondante (donnée dans le même tableau B.1).

4.2.3 Personnalisation du comportement des instances de classes SPML

Une fois que les classes ont été créées comme indiqué précédemment, la création d'objets instances de ces classes doit aussi être personnalisée pour que leur comportement soit conforme à celui désiré. Ainsi les instances de classes SPML doivent par exemple faire en sorte qu'un attribut SPML ayant été déclaré comme `spml_unAttributSPML` puisse être accédé depuis l'extérieur comme `unAttributSPML`.

Un autre exemple : les attributs SPML pouvant être typés (ce type étant donné par la valeur du `"relatedType"` de leur dictionnaire de contrôle), l'instance de classe doit vérifier lors de l'affectation d'une valeur si le type de cette valeur correspond à celui de l'attribut ; selon le cas, l'attribut doit alors être marqué comme étant valide ou non. Une conséquence directe est que si une instance de classe SPML possède un attribut marqué comme non valide, alors cette instance n'est pas valide non plus. La notion de validité d'un objet SPML porte ainsi une information de cohérence des données face au modèle de données.

Le type `spml_class` est nécessairement hérité par toutes les classes SPML, c'est donc dans ses méthodes que nous avons défini le comportement des classes SPML et de leurs instances. L'initialisation d'une instance de classe est prise en charge par la méthode `__init__`. Dans le langage Python, les mécanismes de lecture et d'écriture des valeurs des attributs d'un objet peuvent être modifiés. Nous utilisons cette propriété en surchargeant les méthodes `__getattr__` pour la lecture et `__setattr__` pour l'écriture.

4.2.3.1 L'initialisation des instances de classe SPML : `spml_class.__init__`

Cette méthode est appelée lors de la création d'une instance de classe SPML, par exemple : `maRegion = Region()` en supposant que `Region` soit défini comme une classe SPML. Elle permet d'initialiser des variables importantes :

- *`self.__realdict__ = {}` :* ce dictionnaire contient les noms de tous les attributs de l'instance (sans le préfixe `spml_` pour les attributs SPML) avec leur valeur.
- *`self.__dictOfFieldValidity = {}` :* ce dictionnaire établit le listing de tous les attributs SPML de l'instance avec leur validité (1 : valide, 0 : non valide).
- *`self.spml_garbageable__ = 0` :* cette variable permet de supprimer des objets SPML lorqu'ils ne sont plus utilisés. Pour cela un processus de ramasse-miettes est exécuté et parcourt les objets SPML en regardant la valeur de cette variable. Si elle est à 1, alors l'objet est supprimé.

– *self.__inited = 1 :* `__inited` est une variable de classe initialisée à 0 à la création d'une classe SPML. Lors de la création d'une instance de cette classe, la valeur de `__inited` pour l'instance passe à 1. Ce mécanisme permet d'avoir deux comportements différents de la méthode `__setattr__` lors des phases de création d'une classe et de celle d'une instance de classe (voir §4.2.3.3).

La signature de la méthode `__init__` est la suivante : `def __init__(self,*args,**kw)`. L'étoile devant l'argument `args` indique que c'est un tuple (liste non modifiable). Il contient les éventuels arguments passés en paramètre de création d'une instance de classe. A l'heure actuelle ces arguments ne sont pas pris en compte. Les deux étoiles devant l'argument `kw` indiquent que c'est un dictionnaire. Il contient les éventuels attributs donnés à la création d'une instance de classe, avec leur valeur initiale. Chaque valeur est affectée à son attribut en passant par la méthode `__setattr__` dont le rôle est justement d'affecter les valeurs aux attributs :

```
for key in kw.keys():
  self.__setattr__(key,kw[key])
```

Finalement, la dernière étape consiste à compléter l'instance en vérifiant que tous les attributs définis dans la classe existent. Si ce n'est pas le cas, alors ils sont créés avec une valeur fixée à `None`, c'est-à-dire vides. On est alors certain que l'instance possède bien tous les attributs qui ont été définis dans sa classe, même si certains de ces attributs peuvent être vides, et donc éventuellement non valides.

4.2.3.2 La personnalisation de l'opération de lecture d'un attribut : surcharge de `spml_class.__getattr__`

Dans Python, l'opération de lecture d'un attribut est normalement réalisée grâce à la méthode `__getattr__` de cette manière :

```
def __getattr__(self,name):
  try:
    return self.__dict__[name]
  except KeyError:
    raise AttributeError, name
```

Tous les attributs sont stockés dans le dictionnaire `self.__dict__`, si l'attribut cherché n'y est pas, c'est qu'il n'existe pas et une exception de type `AttributeError` est alors levée. Dans notre cas, nous ne voulons pas passser par ce dictionnaire puisqu'il contient tous les attributs définis dans la classe, y compris ceux avec les préfixes `spml_` qui définissent le comportement de la classe SPML. Pour cette raison, nous choisissons de définir un nouveau dictionnaire `self.__realdict__` que nous initialisons à la création de l'instance et que nous remplissons lors de l'écriture de valeurs dans les attributs (méthode `__setattr__`). L'opération de lecture est alors réalisée ainsi :

```
def __getattr__(self,name):
  try:
    return self.__realdict__[name]
  except KeyError:
    raise AttributeError, name
```

Nous obtenons alors un comportement similaire mais contrôlé par notre dictionnaire `self.__realdict__`.

4.2.3.3 La personnalisation de l'opération d'écriture d'un attribut : surcharge de `spml_class.__setattr__`

Dans Python, l'opération d'écriture d'un attribut est normalement réalisée ainsi :

```
def __setattr__(self,name,value):
  self.__dict__[name] = value
```

Comme nous l'avons dit dans le cas du `__getattr__`, c'est le dictionnaire `self.__dict__` qui est normalement chargé de stocker tous les attributs avec leur valeur. Python étant un langage non typé (en réalité le typage est réalisé à la volée, dynamiquement), l'opération d'écriture d'un attribut consiste alors simplement à remplacer sa valeur dans le dictionnaire.

Le langage SPML est quant à lui fortement typé, ceci nous oblige alors à réaliser certains contrôles avant d'affecter la valeur à l'attribut. Mais tout d'abord, nous devons distinguer la phase de création d'une classe à celle d'une instance de classe. La variable `self.__inited` nous permet de faire cette différence :

- si `self.__inited = 0`, c'est que nous sommes encore en phase de création de la classe par l'interprète. Dans ce cas, l'appel à la méthode `__setattr__` vient du supertype `spml_type_class__`. A ce moment du processus de création de la classe, ce sont pour les attributs définissant le comportement de la classe que l'on souhaite affecter une valeur. Ces attributs ne sont donc pas destinés à être accédés ni même modifiés par un utilisateur. L'affectation de leur valeur est donc simplement réalisée en écrivant leur valeur dans le dictionnaire `self.__dict__` :

  ```
  self.__dict__[name] = value
  ```

- si `self.__inited = 1`, c'est que nous somme en phase de création d'une instance de classe SPML. Dans ce cas, l'appel à la méthode `__setattr__` porte sur les attributs « normaux », c'est-à-dire ceux destinés à être lus et modifiés par un utilisateur. On commence alors le processus de vérification portant sur l'attribut et sa nouvelle valeur. Dans tous les cas, ceux-ci sont stockés dans le dictionnaire `self.__realdict__` :
  - d'abord, le nom de l'attribut de doit pas commencer par `spml_` : ces attributs sont réservés pour définir le comportement de la classe, entre autres.
  - ensuite trois situations sont possibles :
    1. ce n'est pas un attribut SPML, donc on reproduit le `__setattr__` Python avec notre dictionnaire `self.__realdict__` :

       ```
       self.__realdict__[name] = value
       ```

    2. c'est un attribut SPML mais la nouvelle valeur n'est pas du bon type. Dans ce cas, pour toute autre valeur que `None`, une exception `SPMLAttributeError` est levée. Sinon, l'attribut est considéré comme non valide et le dictionnaire `self.__dictOfFieldValidity` est mis à jour :

       ```
       self.__dictOfFieldValidity[name]=0
       ```

       Le dictionnaire `self.__realdict__` est aussi mis à jour : si l'attribut SPML est un type intrinsèque dont le stéréotype est `"IDENTIFICATION"`, sa valeur définit le nom de l'instance, nous choisissons alors une valeur par défaut :

```
defName = self.getKlass().getId()+"__"+str(self.getUid())
self.__realdict__[name] = defName
```

sinon l'attribut est associé à `None` :

```
self.__realdict__[name] = None
```

3. c'est un attribut SPML et la nouvelle valeur est du bon type.
Si cette valeur est `None`, elle est remplacée par la valeur par défaut de l'attribut :

```
if value is None:
  try:
    dval = field.getDefaultValue()
    value = dval
  except:
    pass
```

Ensuite, on essaie de récupérer la méthode permettant de modifier la valeur avant son affectation (cf §3.3.6) :

```
bscontrols = getattr(self,"spml_"+name+"_bscontrols__")
```

Si la méthode existe, elle est exécutée et la valeur est modifiée en conséquence, sous réserve d'erreurs de syntaxe et que le type de la valeur modifiée reste conforme à celui de l'attribut :

```
try:
  value = bscontrols(value)
  if self.getKlass().checkField(name,value) is not 0:
    raise spml_error, "type error after controls "+
              "value modification for field",name
except spml_exception:
  print "spml_exception catch"
  raise
except:
  print "Error in spml_"+name+"_bscontrols__ method
     coding"
  raise
```

Ensuite, nous considérons l'attribut comme valide et le dictionnaire `self.__dictOfFieldValidity` est mis à jour :

```
self.__dictOfFieldValidity[name]=1
```

Le dictionnaire `self.__realdict__` est aussi mis à jour :

```
self.__realdict__[name] = value
```

Cependant nous devons considérer le cas où le type de l'attribut en cours de modification est une instance de classe SPML. Dans ce cas de figure, si la la nouvelle valeur est bien une instance de `spml_class` (et pas `None`), deux aspects sont à examiner :

(a) la validité de l'attribut dépend de celle de l'instance :

```
self.__dictOfFieldValidity[name] = value.isValid()
```

(b) si le stéréotype est `"COMPOSITION"`, l'ancienne valeur ne peut plus exister et doit donc être supprimée par le mécanisme de ramasse-miettes. Pour cela on modifie la valeur de son attribut `self.spml_garbageable__` :

```
try:
  # l'ancienne valeur est récupérée
  oldField=getattr(self,name)
  if oldField != None:
    oldField.setGarbageable()
    # dans le cas d'un "CollectionField", cette
    # opération est répétée pour tous les éléments
    # de la liste
except AttributeError:
  # cette exception signifie que l'attribut
  # n'a pas d'ancienne valeur : on ne fait rien
  pass
except:
  raise
```

Enfin, on finit par essayer de récupérer la méthode permettant d'exécuter du code après l'affectation de la nouvelle valeur :

```
ascontrols = getattr(self,"spml_"+name+"_ascontrols__")
```

Si la méthode existe, elle est exécutée, sous réserve d'erreurs de syntaxe.

4.3 Réalisation de l'interprète SPML

L'interprète est chargé d'exécuter les commandes qui lui sont données. Ce travail est très compliqué, et développer un interprète complet prendrait trop de temps. Nous avons donc décidé d'utiliser l'interprète de Python, et ainsi de bénéficier des avantages qu'apporte un tel outil : la syntaxe Python, et donc la syntaxe SPML, sont nativement reconnues.

L'interprète n'est pas exactement le même selon qu'il soit en ligne de commande, ou qu'il interprète un fichier SPML. Pour cela, nous avons créé une structure permettant de faire face à ces cas, le diagramme de classe UML de la figure 4.1 la représente.

La classe de base `SPMLInterpreter` est chargée d'analyser les lignes de commandes qui lui sont passées en paramètre et de gérer des dictionnaires représentant des environnements d'exécution.

La classe `SPMLConsole` gère un tampon pour les commandes en entrée de l'interprète et lui envoie les commandes.

Les classes `SPMLInteractiveConsole` et `SPMLEnhancedConsole` émulent le plus fidèlement possible l'interprète intéractif Python et ajoutent la sémantique SPML.

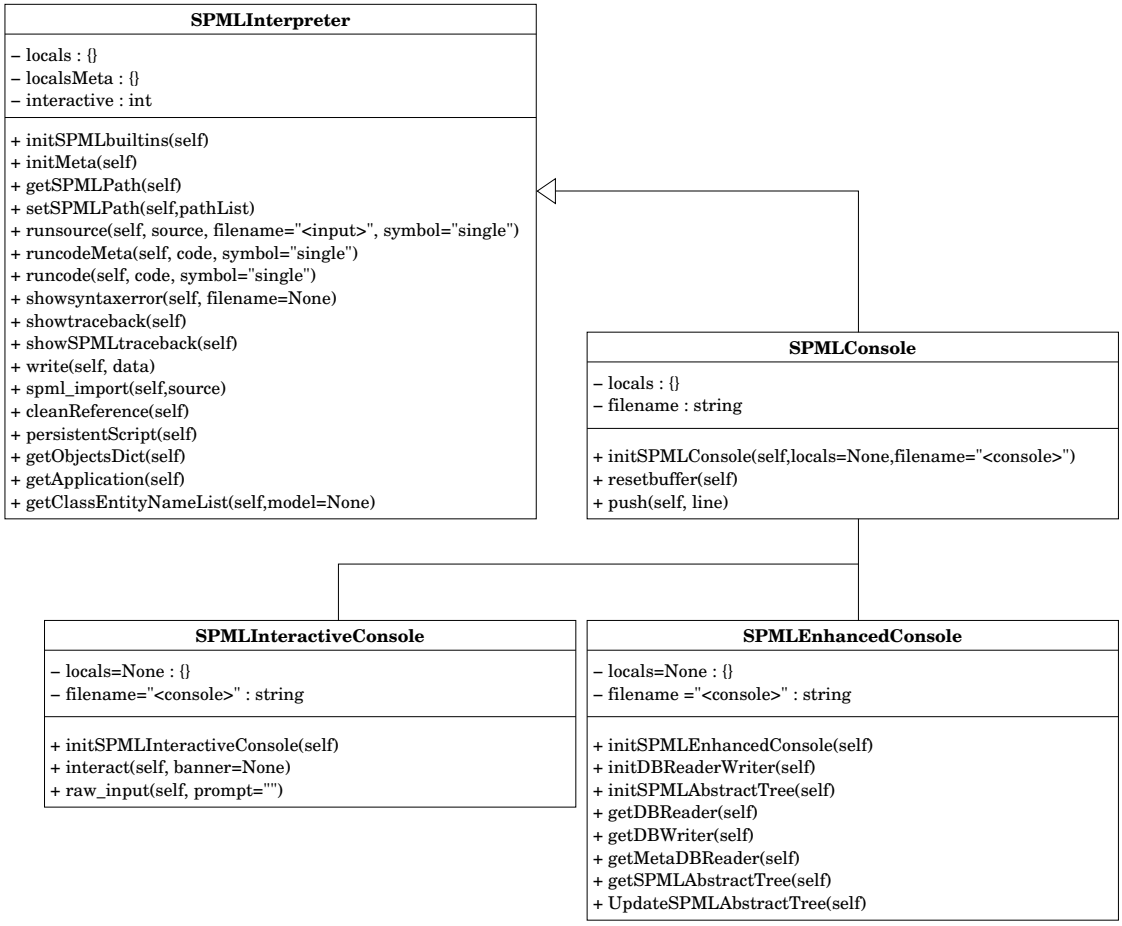


FIG. 4.1 – Diagramme de classe UML de la structure des interprètes SPML

4.3.1 L'interprète de base : SPMLInterpreter

4.3.1.1 Initialisation de l'interprète

Un interprète de base peut être initialisé avec deux dictionnaires Python : `locals` et `localsMeta`. Ceux-ci sont en fait des environnements d'exécution dans lesquels du code peut être exécuté. Les variables créées par ces lignes de code sont ainsi stockées dans ces dictionnaires. Le code suivant montre un exemple de commande (`a='uneVariableLocale'`) exécuté dans un environnement d'exécution précis (`locals`) :

```
locals = {}
exec "a='uneVariableLocale'" in locals
```

Si aucun dictionnaire `locals` n'est donné à la création de l'interprète, il est créé avec les clés `"__name__"` et `__doc__` et leurs valeurs respectives à `"__console__"` et `None`. De même pour le dictionnaire `localsMeta`.

```
if localsMeta is None:
  localsMeta = {"__name__": "__console__", "__doc__": None}
self.localsMeta = localsMeta
if locals is None:
```

```
  locals = {"__name__": "__console__", "__doc__": None}
self.locals = locals
```

Le dictionnaire `locals` correspond à l'environnement d'exécution par défaut dans lequel sera exécuté le plupart du code. Le dictionnaire `localsMeta` correspond à un environnement d'exécution particulier, réservé à l'exécution de code SPML dit « natif ». Ce code SPML « natif » est historique et possède une syntaxe qui n'est pas du Python. Il correspond en fait aux premières versions du langage SPML. Lors du passage à la syntaxe Python, nous avons voulu garder une compatibilité avec les anciens fichiers de modèles de données, ce qui explique la présence du dictionnaire `localsMeta`. Pour assurer la compatibilité entre les versions, une simple redirection est réalisée entre les noms des anciennes classes et les nouvelles. Par exemple, l'ancienne syntaxe de `spml_class` est `ClassEntity`. Nous passerons sous silence tout ce qui a été fait pour assurer la compatibilité avec les versions précédentes de syntaxe du langage, ce qui n'altère en rien les explications données.

Une fois les environnements d'exécution initialisés, l'interprète crée un objet particulier destiné à compiler des commandes en code Python :

```
self.compile = CommandCompiler()
```

Ce compilateur renvoie un objet `code` si la commande est complète et valide ; `None` si la commande est incomplète ; il lève l'exception `SyntaxError` si la commande est incomplète et contient une erreur de syntaxe, ou `OverflowError` or `ValueError` si la commande contient un terme invalide. C'est donc par ce compilateur que passeront toutes les commandes à exécuter.

L'initialisation de l'interprète continue avec la définition d'un mode intéractif ou non. Ce mode n'est utilisé que pour la redirection des erreurs :

```
#definition of output for reporting error
self.interactive = interactive
```

Enfin le niveau méta du langage doit être initialisé pour que les utilisateurs puissent créer des objets SPML. Ceci est réalisé par la méthode `self.initSPMLbuiltins()` qui importe le fichier contenant la définition du métamodèle dans le dictionnaire `locals` :

```
exec "from DATA_SPML import *" in self.locals
```

Le fichier `DATA_SPML.py` contient la définition de `spml_class` et d'autres types intrinsèques, mais surtout elle importe aussi le fichier `DATA_SPMLtype` contenant lui la définition de toutes les métaclasses du métamodèle.

De la même manière, le niveau méta du langage pour l'ancienne syntaxe, est initialisé. L'environnement d'exécution concerné est cette fois-ci `localsMeta` et c'est la méthode `self.initMeta()` qui réalise cette initialisation :

```
exec "from DATA_MetaModel import *" in self.localsMeta
```

Le fichier `DATA_MetaModel` contient toutes les redirections du métamodèle entre l'ancienne et la nouvelle syntaxe.

4.3.1.2 Les fonctionnalités de l'interprète

Une fonctionnalité principale est l'analyse et l'exécution de code source. Cette fonctionnalité est réalisée par la méthode `runsource(self, source, filename="<input>", symbol="single")`. La signature est la même que celle de l'objet `self.compile` (instance de `CommandCompiler`) : `source` est la chaîne de caractère source ; `filename` est un nom de fichier optionnel depuis lequel le code source est lu ; et `symbol` est un symbole optionnel pour la grammaire dont la valeur est soit `"single"`, soit `"eval"`.

Durant la phase d'analyse du code source, l'interprète vérifie d'abord la présence de la commande `spml_import` qui signifie que la commande souhaite importer le code SPML contenu dans un fichier :

```
if source.startswith("spml_import"):
  try:
    if verboseMode: print "In runsource, found spml_import"
    try:
      self.openedStreams_ = self.openedStreams
      self.streamLine_ = self.streamLine
      self.importMode_ = self.importMode
    except:
      pass

    self.spml_import(source)
    if verboseMode: print "Model",source,"imported"

    try:
      self.openedStreams = self.openedStreams_
      self.streamLine = self.streamLine_
      self.importMode = self.importMode_
    except:
      pass
  except SPMLInterpreterError, e:
    if not self.interactive:
      raise
    map(self.write,e.trace)
```

Dans ce cas, il fait appel à la méthode `spml_import(self,source)` dont le rôle est d'ouvrir en lecture et d'exécuter le code SPML du fichier spécifié après l'instruction `spml_import`. La récursivité de ce mécanisme (une instruction `spml_import` appel un fichier qui contient aussi l'instruction `spml_import`, et ainsi de suite) est complétement gérée grâce à la variable `self.openedStreams` qui joue le rôle de pile pour les fichiers à importer. Cette pile est vidée au fur et à mesure que les fichiers sont importés.

Une fois passé ce test d'import, l'analyse continue :

```
else:
  mode = 1 # 1 is normal, 0 is need to use localsMeta
  # test if source line is SPML specific and needs
  # particular treatment
  if source.startswith("spml_native"):
    #delete "spml_native keyword"
    srclist = source.split()
    del srclist[0]
    source = " ".join(srclist)
    mode = 0
```

```
  #standard python execution
  try:
    code = self.compile(source, filename, symbol)
  except (OverflowError, SyntaxError, ValueError):
    # Case 1
    self.showsyntaxerror(filename)
    return False

  if code is None:
    # Case 2
    return True

  # Case 3
  if mode:
    self.runcode(code)
  else:
    self.runcodeMeta(code)
return False
```

Nous devons définir deux modes d'exécution du code pour d'un côté l'ancienne syntaxe SPML, et de l'autre la nouvelle syntaxe conforme à celle de Python. La première syntaxe est signalée par l'instruction `spml_native` en début de commande et se traduit par la définition d'un mode à 0. La deuxième syntaxe (Python) se traduit par la définition du mode à 1. Une fois ce mode défini, le code est analysé et exécuté à proprement parlé grâce à un compilateur `self.compile`.

- Si ce compilateur lève une exception, le message d'erreur est envoyé à l'utilisateur et la méthode renvoie la mention `False`.
- Si le compilateur renvoit `None`, la méthode renvoie la mention `True`.
- Si le compilateur renvoie du code Python, alors celui-ci est exécuté dans l'environnement d'exécution contenant les variables adaptées : pour un mode à 1, la méthode `self.runcode` est appelée et elle exécute le code dans `self.locals` ; pour un mode à 0, la méthode `self.runcodeMeta` est appelée et elle exécute le code dans `self.localsMeta`. Une fois le code exécuté, la méthode `runsource` renvoie la mention `False`, sauf si une exception est levée.

-

La valeur de retour de cette méthode permet de choisir quel prompt afficher dans le cas d'un interprète intéractif : `>>>` pour une nouvelle commande ou `...` pour une commande incomplète (cf §4.3.3).

Une autre fonctionnalité proposée par l'interprète est la gestion des erreurs. Les méthodes `showsyntaxerror(self, filename=None)`, `showtraceback(self)` ou `showSPMLtraceback(self)` sont appelées dès qu'une erreur, de syntaxe par exemple, apparaît. Ces méthodes récupèrent les informations de la dernière exception levée grâce à l'appel de la méthode `sys.exc_info()`. Ensuite soit l'exception est propagée (en mode non intéractif), soit elle est écrite sur l'erreur standard (mode intéractif). Le mode non intéractif permet par exemple de déclencher l'apparition d'une boîte de dialogue dans une interface utilisateur avec des informations sur l'erreur ayant provoquée l'exception.

L'interprète permet aussi, avec la méthode `persistentScript(self)`, de sauvegarder tous les objets en mémoire dans un script. Si ce script est exécuté dans un nouvel environnement d'exécution, tous les objets qui ont été sauvegardés seront recréés. Pour cela une

liste est créée avec tous les objets instances de `spml_class`. Cette liste est ordonnée de telle sorte que les objets qui sont référencés apparaissent avant ceux qui les référencent. Le script est alors construit en faisant appel à la méthode `getExecutableRepresentation()` de tous ces objets :

```
def persistentScript(self):
  """return the sorted list of all objects

  spmlclass : a spml_class of which the objects should be
  (to be implemented)

  return: a script
  """
  sList = []
  for obj in self.locals.values():
    if isinstance(obj,DATA_SPML.spml_class):
      add = 0
      #add obj in sList only if it is not already in
      if obj not in sList:
        #for all object in the sorted list
        for sObj in sList:
          if obj.hasReferenceTo(sObj):
            sList.insert(sList.index(sObj),obj)
            add = 1
            break
      if not add:
        sList.append(obj)
  sList.reverse()
  #write code for each object
  stream=""
  for object in sList:
    string="automaticGenerateName__"+str(object.getUid())+"="
    string=string+object.getExecutableRepresentation()
    stream = stream+string+"\n"
  return stream
```

Enfin, l'interprète propose des méthodes permettant :
- de récupérer le dictionnaire `self.locals` : `getObjectsDict(self)`,
- de récupérer l'application si elle est définie : `getApplication(self)`,
- de récupérer la liste des objets par leur noms, avec la possibilité de se limiter à un modèle : `getClassEntityNameList(self,model=None)`.

4.3.2 La console de base : SPMLConsole

Cette console est un interprète SPML dont le rôle est d'émuler au plus près l'interprète de Python. Cette console n'est pas intéractive par défaut, elle se contente de gérer les commandes à analyser et exécuter, ligne par ligne. Elle définit pour cela une mémoire tampon d'entrée contenant les lignes de la commande : `self.buffer`. Le fichier à partir duquel sont lues les commandes est enregistré dans une variable : `self.filename`. La gestion d'une commande à analyser est effectuée dans la méthode `push(self, line)` :

```
def push(self, line):
  """Push a line to the interpreter.
```

```
The line should not have a trailing newline; it may have
internal newlines.  The line is appended to a buffer and the
interpreter's runsource() method is called with the
concatenated contents of the buffer as source.  If this
indicates that the command was executed or invalid, the buffer
is reset; otherwise, the command is incomplete, and the buffer
is left as it was after the line was appended.  The return
value is 1 if more input is required, 0 if the line was dealt
with in some way (this is the same as runsource()).
"""
self.buffer.append(line)
source = "\n".join(self.buffer)
more = self.runsource(source, self.filename)
if not more:
  self.resetbuffer()
return more
```

Cette méthode fait appel à la méthode `runsource` pour analyser les lignes d'une commande. En fonction de la valeur de retour, on sait si la commande est terminée (`more=False`), ou si elle continue (`more=True`).

4.3.3 La console intéractive : SPMLInteractiveConsole

Cette console est intéractive et permet d'utiliser un interprète en lignes de commandes. Elle définit les deux méthodes nécessaires pour un interprète de type Python :

- `interact(self, banner=None)`
  Cette méthode définit ce que va écrire la console à l'écran en fonction de ce qui est écrit par l'utilisateur. La ligne débute par un prompt qui sera soit `>>>` pour une nouvelle commande ou `...` pour l'attente de la suite d'une commande incomplète. Le prompt affiché dépend de la valeur de retour de la méthode `push(line)` où `line` est la ligne de commande donnée par l'utilisateur.
- `raw_input(self, prompt="")`
  Cette méthode permet de récupérer une commande en lisant une ligne sur l'entrée standard et en la convertissant en chaîne de caractère.

4.3.4 La console évoluée : SPMLEnhancedConsole

Cette console améliorée est destinée à être utilisée depuis une application évoluée dans laquelle la gestion et l'accès aux objets doivent être encadrés. Elle met ainsi à disposition d'une telle application les objets suivants (leurs structures sont détaillées dans les paragraphes §4.4 et §4.5) :

- `self.reader` : cet objet propose un accès en lecture contrôlé aux objets SPML du dictionnaire `self.locals`.
- `self.writer` : cet objet propose un accès en écriture contrôlé aux objets SPML du dictionnaire `self.locals`.
- `self.metaReader` : cet objet propose un accès en lecture contrôlé aux objets SPML des dictionnaires `self.locals` et `self.localsMeta`.
- `self.spmlAbstractTree` : cet objet propose une représentation ordonnée et évoluée des objets SPML. Cette représentation gère entre autres les intéractions entre l'utilisateur et les objets SPML qui ont lieu à travers une interface extérieure (interface graphique par exemple). L'interprète amélioré permet de mettre à jour cette représentation grâce à une méthode dédiée : `UpdateSPMLAbstractTree(self)`.

4.4 Réalisation de la gestion des modèles et des données

Le SPML étant un langage qui dérive de Python, l'accès à ses objets peut très bien être fait directement, selon le mécanisme `valeur = objet.attribut`. Cependant, contrairement à Python, le SPML est un langage avec des contraintes de types. Nous proposons alors un module pour gérer l'accès aux objets SPML, que ce soit pour la lecture ou l'écriture. Ce module s'appelle le `DATA_DBReaderWriter` et est composé de trois classes : le `DataDBReader`, le `DataDBWriter` et le `DataMetaDBReader`.

Toutes ces classes sont accessibles depuis une interface CORBA ; elles proposent donc des méthodes publiques, décrites dans une interface IDL, à partir desquelles une application extérieure peut interroger ce que l'on va appeler la base de données, c'est à dire non seulement les objets SPML instanciés, mais aussi la structure de ces objets (via le métamodèle). Ces méthodes publiques font ensuite appel aux méthodes privées des classes. Ce sont les méthodes privées qui réalisent les opérations demandées.

Les méthodes publiques (celles qui sont accessibles depuis une interface CORBA) sont généralement utilisées comme couches intermédiaires pour accéder aux méthodes privées depuis une interface CORBA. Leurs arguments ne peuvent alors pas être des objets Python ; ce sont des types CORBA : `string`, `long`, `double`, `boolean` ou `any`. Le type CORBA « `any` » porte sur lui la valeur de la variable ainsi que son type. Il permet de définir des variables dont on ne connaît pas le type par avance.

Le module `DATA_DBReaderWriter` définit la fonction `getEntityWithUid(entityUid,objDict)` dont le rôle est de retrouver un objet à partir de son identifiant unique (`entityUid`) et d'un environnement d'exécution (`objDict`). Cette fonction est utilisée par les méthodes publiques : prenons l'exemple d'une méthode `X` chargée de récupérer la valeur d'un attribut `attr` d'un objet `obj` en mémoire. Un des arguments de cette méthode sera nécessairement l'objet `obj` en question. Le problème se pose si un programme ou une application souhaite utiliser cette méthode `X` à travers une couche CORBA. Cette application pourra pas donner un objet Python comme argument à la méthode car ces objets ne peuvent pas transiter à travers une couche CORBA. Une solution sera alors de passer par une méthode publique `X_publique` qui fera le lien entre la méthode `X` et la couche CORBA. Les arguments de cette méthode `X_publique` pourront transiter à travers la couche CORBA, car ce seront des types simples (entiers, string,...). L'objet `obj` sera alors représenté par un entier `objUid` correspondant à son uid et la méthode publique retrouvera l'objet `obj` grâce à la fonction `getEntityWithUid(entityUid,objDict)`. Elle pourra alors réaliser l'opération demandée sur cet objet.

Les classes `DataDBReader` et `DataDBWriter` héritent de la même classe abstraite `DataDB`. Cette dernière définie une méthode qui permet de récupérer le type d'un attribut SPML à partir d'un objet SPML et de l'attribut. Cette méthode est déclinée en deux versions, l'une privée, l'autre publique :

- `def getFieldType_P(self,object,field):` *(privée)*
- `def getFieldType(self,entityUid,fieldName):` *(publique)*

4.4.1 Le DataDBReader

Le `DataDBReader` est initialisé à partir d'un environnement d'exécution contenant les objets SPML (`self.objectsDict`). Nous présentons ici la liste de ses méthodes.

Pour chacune d'entre elles, nous préciserons son caractère privé ou public. Pour chacune des méthodes publiques, nous indiquerons la ou les méthodes privées qui peuvent être appelées.

Le `DataDBReader` propose les méthodes suivantes :

- `def getField_P(self,object,field)` : *(privée)*
  Renvoie la valeur du `spml_field field` de `object`.

- `def getCField_P(self,object,field,index=None)` : *(privée)*
  Renvoie la valeur du `spml_field field` de `object`. Cette valeur est une liste. Si `index=None`, la liste est renvoyée, sinon c'est l'élément à la position `index` qui est renvoyé.

- `def getEntityType_P(self,object)` : *(privée)*
  Renvoie le type de l'instance de classe SPML `object`.

- `def getField_Integer_P(self,object,field)` : *(privée)*
  Renvoie la valeur du `spml_field field` de `object`. Cette valeur est un entier.

- `def getField_Double_P(self,object,field)` : *(privée)*
  Renvoie la valeur du `spml_field field` de `object`. Cette valeur est un réel.

- `def getField_Boolean_P(self,object,field)` : *(privée)*
  Renvoie la valeur du `spml_field field` de `object`. Cette valeur est un booléen.

- `def getField_String_P(self,object,field)` : *(privée)*
  Renvoie la valeur du `spml_field field` de `object`. Cette valeur est une chaîne de caractères.

- `def getField_Entity_P(self,object,field)` : *(privée)*
  Renvoie la valeur du `spml_field field` de `object`. Cette valeur est une instance de classe SPML.

- `def getEntity_P(self,klass)` : *(privée)*
  Renvoie la liste des objets instances de `klass`.

- `def getObjects_P(self,spmlclass=None)` : *(privée)*
  Renvoie la liste ordonnée des instances dont la classe est `spmlclass`. La liste est rangée de telle sorte que les instances qui sont référencées (par association par exemple) sont placées avant celles qui les référencent. Si `spmlclass` est à `None`, c'est la liste de toutes les instances de `spml_class` qui est renvoyée.

- `def getRepresentation_Entity_P(self,entityList)` : *(privée)*
  Renvoie la liste des noms des instances dont les uids sont donnés en entrée. La liste renvoyée est formée ainsi : [str(uid1), str(obj1), str(uid2), str(obj2), ...]

- `def getPotentialValues_Entity_P(self,field)` : *(privée)*
  Renvoie une liste des instances dont le type correspond au type de l'attribut `field`. Par exemple si `field` est de type « `Region` », alors la méthode retournera la liste de toutes les instances de cette classe « `Region` ». La liste sera en fait composée

de la représentation de ces instances (`str(obj)`).

- `def getListOfFields_P(self,object): :` *(privée)*
  Renvoie la liste des noms des attributs de l'objet `object`.

- `def getListOfFields(self,entityUid):` *(publique)*
  Cette méthode renvoie la liste des attributs SPML d'une instance de classe SPML. Elle utilise pour cela la méthode `getGenericFields()` sur le type de l'instance (soit `spml_type_class__` ou un type dérivé).

- `def getEntityType(self, entityUid):` *(publique)*
  ⟶ *getEntityType_P*

- `def getField(self,entityUid,fieldName):` *(publique)*
  ⟶ *getFieldType_P*
  ⟶ *getField_Entity_P*
  ⟶ *getField_Integer_P*
  ⟶ *getField_Double_P*
  ⟶ *getField_String_P*
  ⟶ *getField_Boolean_P*
  Cette méthode doit renvoyer la valeur de l'attribut `fieldName` de l'instance identifiée par son uid `entityUid`. La difficulté réside dans le fait qu'elle est appelée quel que soit le type de l'attribut. Ne connaissant pas par avance le type de l'attribut dont la valeur est demandée, nous passons donc par le type CORBA `any`.

- `def getEntity(self,entityType):` *(publique)*
  ⟶ *getEntityType_P*

- `def getObjects(self,spmlclass=None):` *(publique)*
  ⟶ *getObjects_P*

- `def getRepresentation_Entity(self,entityUidList):` *(publique)*
  ⟶ *getRepresentation_Entity_P*

- `def getPotentialValues_Entity(self,entityUid,fieldName):` *(publique)*
  ⟶ *getPotentialValues_Entity_P*

4.4.2 Le DataDBWriter

Tout comme le `DataDBReader`, le `DataDBWriter` est initialisé à partir d'un environnement d'exécution contenant les objets SPML (`self.objectsDict`). Le rôle de cette classe est de créer, modifier et détruire des objets SPML. Nous présentons ici la liste de ses méthodes. Pour chacune d'entre elles, nous préciserons son caractère privé ou public. Pour chacune des méthodes publiques, nous indiquerons la ou les méthodes privées qui peuvent être appelées.

Le `DataDBWriter` propose les méthodes suivantes :

- `def createEntity_P(self,klass):` *(privée)*
  Cette méthode permet de créer une instance de la classe SPML `klass` si celle-ci

existe, c'est-à-dire si elle est dans le dictionnaire `self.objectsDict`. Une fois cette instance créée, elle est rajoutée dans le dictionnaire `self.objectsDict` sous la clé `"__uid_"+str(newInstance.getUid())+"__dataServer"`. Connaître la manière dont est construite cette clé est important car cela permet par la suite de retrouver une instance à partir de son uid dans l'environnement d'exécution.

- `def createEntityArgs_P(self,klass,args,kw):` *(privée)*
  Cettte méthode a le même rôle que la précédente, avec la possibilité de préciser la valeur de certains attributs grâce au dictionnaire `kw` (ex : `createEntity(Region,() ,{"name":"maRegion"})`). Le tuple `args` n'est pour l'instant pas pris en compte.

- `def deleteEntity_P(self,object):` *(privée)*
  Cette méthode permet de supprimer une instance de classe SPML. Si l'instance en question est référencée par une autre instance, cette suppression n'est possible que si la relation est de type association (et non pas composition ni aggregation). La suppression consiste à faire appel à la méthode `setGarbageable()` de l'instance puis à nettoyer le dictionnaire des objets avec la fonction `cleanObjectReference( self.objectsDict)` du métamodèle.

- `def setField_P(self,object,field,values):` *(privée)*
  Cette méthode permet d'affecter une ou plusieurs valeurs à un attribut d'une instance de classe SPML. Pour cela, la méthode `__setattr__` de l'instance est utilisée.

- `def setField_Entity_P(self,object,field,values):` *(privée)*
  Cette méthode permet d'affecter une ou plusieurs valeurs à un attribut d'une instance de classe SPML et dont le type correspond à une instance de classe SPML. La valeur est l'uid de cette instance. L'instance concernée est donc d'abord recherchée grâce à la fonction `getEntityWithUid` avant d'être affectée à l'attribut via la méthode `__setattr__` de l'instance à modifier.

- `def setCField_P(self,object,field,values,index=None):` *(privée)*
  Cette méthode permet d'affecter une ou plusieurs valeurs à un attribut d'une instance de classe SPML et dont le mode de liste est `CollectionField`. Si `index` est à `None` la liste de valeurs est affectée à l'attribut, si `index` est à `-1` elle est concaténée à la liste existante de l'attribut, et sinon l'élément à la position `index` est remplacé par la valeur.

- `def setCField_Entity_P(self,object,field,values,index=None):` *(privée)*
  Cette méthode permet d'affecter une ou plusieurs valeurs à un attribut d'une instance de classe SPML, dont le type correspond à une instance de classe SPML et dont le mode de liste est `CollectionField`.

- `def deleteField_P(self,object,field):` *(privée)*
  Cette méthode permet de supprimer la valeur de l'attribut `field` de l'instance `object`.

- `def deleteCField_P(self,object,field,index):` *(privée)*
  Cette méthode permet de supprimer la valeur de l'attribut `field`, dont le mode de liste est `CollectionField`, de l'instance `object`. Si `index` est à `-1`, la valeur

de l'attribut, c'est-à-dire la liste entière, est supprimée. Sinon c'est l'élément à la position `index` qui est supprimé.

- `def createEntity(self,aEntityName):` *(publique)*
  ⟶ *createEntity_P*

- `def createEntityArgs(self,aEntityName,args,kw):` *(publique)*
  ⟶ *createEntityArgs_P*

- `def deleteEntity(self,entityUid):` *(publique)*
  ⟶ *deleteEntity_P*

- `def setField(self,entityUid,fieldName,values):` *(publique)*
  ⟶ *getFieldType_P*
  ⟶ *setField_Entity_P*
  ⟶ *setField_P*
  Cette méthode permet d'affecter une ou plusieurs valeurs à un attribut d'une instance de classe SPML. L'instance est repérée par son identifiant unique `entityUid`, l'attribut par son nom `fieldName`, et le type des valeurs `values` est le type CORBA `any`. Elles portent ainsi leur valeur et leur type réel.

4.4.3 Le DataMetaDBReader

Le `DataMetaDBReader` est initialisé à partir du dictionnaire des objets SPML `self.objectsDict` et aussi avec le dictionnaire des objets définis au niveau méta `self.metaObjectsDict`. Cette classe permet d'accéder aux méthodes des superclasses du métamodèle. Nous présentons ici la liste de ses méthodes. Pour chacune d'entre elles, nous préciserons son caractère privé ou public. Pour chacune des méthodes publiques, nous indiquerons la ou les méthodes privées qui peuvent être appelées.

Le `DataMetaDBReader` propose les méthodes suivantes :

- `def getPotentialTypes_P(self,klass):` *(privée)*
  Cette méthode renvoie la liste de toutes les classes concrètes de type `klass`. Les classes héritant de `klass` sont aussi concernées.

- `def getType_P(self,klass,field):` *(privée)*
  Cette méthode renvoie une liste de `string` informant sur le type de l'attribut `field` de la classe `klass`. Cette liste ne contient qu'un seul élément et est construite de la manière suivante :

```
#entity check
if field.getType().is_a("Entity"):
  return ["classSPML"]
#intrinsic check
elif field.getType().is_a("INTEGER"):
  return ["integer"]
elif field.getType().is_a("DOUBLE"):
  return ["double"]
elif field.getType().is_a("STRING"):
  return ["string"]
elif field.getType().is_a("BOOLEAN"):
  return ["boolean"]
```

```
elif field.getType().is_a("NUMERIC"):
  return ["double"]
elif field.getType().is_a("NUMERIC_OR_STRING"):
  return ["string"]
elif field.getType().is_a("VOID"):
  raise AttributeError
```

- `def getClassUiInformations_P(self,klass):` *(privée)*
  Cette méthode renvoie les informations d'interface utilisateur à propos de la classe `klass`. Elle interroge pour cela la méthode `getUiInformations()` de la classe concernée. La valeur de retour est une liste de `string` composée des valeurs de `label`, `comment` et `tooltip`.

- `def getAttributeUiInformations_P(self,field):` *(privée)*
  Cette méthode renvoie les informations d'interface utilisateur à propos de l'attribut `field`. Elle interroge pour cela la méthode `getUiInformations()` de l'objet `spml_field__` correspondant à l'attribut concerné. La valeur de retour est une liste de `string` composée des valeurs de `label`, `comment` et `tooltip`.

- `def getPotentialTypes(self,aClass):` *(publique)*
  ⟶ *getPotentialTypes_P*

- `def getType(self,aClass,fieldName):` *(publique)*
  ⟶ *getType_P*

- `def getClassUiInformations(self,aClass):` *(publique)*
  ⟶ *getClassUiInformations_P*

- `def getAttributeUiInformations(self,aClass,fieldName):` *(publique)*
  ⟶ *getAttributeUiInformations_P*

4.5 Réalisation de la projection des données

Nous voulons offrir la possibilité d'intéragir dynamiquement avec les objets de manière intelligente depuis un programme extérieur. Pour cela, nous avons réalisé une structure appelée « arbre abstrait » qui organise les données d'une application SPML sous forme arborescente. Cet arbre, qui est une instance de la classe `SPMLAbstractTree`, est constitué de feuilles particulières : des instances de `SPMLFeather`. Ceci permettra par exemple de présenter à l'utilisateur les valeurs des paramètres qu'il a saisis et de lui proposer de les modifier.

Une feuille `SPMLFeather` est créée pour chaque objet SPML de l'application et les relations entre les objets se traduisent par des relations parents-enfants entre les feuilles. Chaque feuille porte des informations liées à l'objet SPML associé : nom, valeur, validité, référence,... En plus de ces informations basiques, nous proposons d'associer à chaque feuille des méthodes évoluées permettant d'exécuter des actions précises sur l'objet SPML associé (édition, suppression, ...). Ces méthodes évoluées sont des intances des classes `FunctionWrapper` ou `BoundMethodWrapper` et seront appelées mappeurs dans la suite du document.

4.5.1 Le mapping des fonctions : `FunctionWrapper` et `BoundMethodWrapper`

Le rôle des classes `FunctionWrapper` et `BoundMethodWrapper` est de fournir des objets permettant d'exécuter des fonctions ou des méthodes sur des objets SPML, avec éventuellement des arguments en entrée et une ou plusieurs valeurs de retour. La première classe permet de mapper des fonctions alors que la deuxième mappe des méthodes (c'est-à-dire des fonctions rattachées à des instances de classes). De tels mappeurs possèdent un identifiant et optionnellement un convertisseur d'argument. Ce convertisseur permet de forcer le type d'un argument passé en entrée. Enfin, l'exécution de la fonction mappée est effectuée par la méthode `mappeur.run(self,args,kw)`.

Voici comment fonctionnent ces mappeurs : nous voulons par exemple mapper l'accès en lecture d'un attribut pour un objet SPML. Si nous le faisons directement, l'instruction est la suivante :

```
value = getattr(object,attrName)
```

Nous voulons reproduire ce comportement mais avec un mappeur de méthode `BoundMethodWrapper`. Ce mappeur sera identifié par le nom `"GetValue"`, la méthode à wrapper est `__getattr__` sur l'objet `object` (cet objet est une instance de `spml_class`). Cette méthode possède un argument `name` que nous renseignons avec le nom de l'attribut `attrName`. Cela donne le code suivant :

```
# 'object' est un objet SPML (instance de spml_class)

getValue = BoundMethodWrapper("GetValue",
                              getattr(object,"__getattr__"),
                              {"name":"attrName"})
```

L'appel à la méthode `run` de l'objet getValue provoquera l'exécution de la méthode `__getattr__` sur l'objet `object` avec comme argument le nom de l'attribut `attrName`.

L'intérêt de ces classes est de pouvoir proposer une méthode sur des objets SPML qui s'appellera par exemple toujours `"GetValue"`, mais dont la réalisation sera adaptée à chaque objet SPML. Ainsi on voudrait pouvoir récupérer une des valeurs d'un attribut SPML qui est de type collection avec une méthode dont le nom serait toujours `"GetValue"`. La méthode mappée serait cette fois-ci différente de celle de l'exemple précédent car il ne suffira pas de faire appel à la méthode `object.__getattr__` :

```
# 'reader' est une instance de DATA_DBReader
# 'object' est un objet SPML (instance de spml_class)
# 'field' est une instance de spml_utility_collectionField__
# => La valeur de 'field' est donc une liste []
# 'fieldCount' est l'index de la valeur à récupérer dans cette
   liste

getValue = BoundMethodWrapper("GetValue",
                              getattr(reader,"getCField_P"),
                              {"object":object,
                               "field":field,
                               "index":fieldCount}))
```

Nous obtenons ainsi un mappeur de méthode nommé `"GetValue"` comme précédemment, mais qui mappe cette fois-ci l'opération suivante :

```
value = reader.getCField_P(object=object,
                           field=field,
                           index=fieldCount)
```

Ces mappeur de méthodes sont faits pour être associés à des feuilles SPML `SPMLFeather`, et donc à l'objet SPML qui lui est associé. De cette manière, une application peut demander l'exécution de la méthode `"GetValue"` sur n'importe qu'elle feuille possédant une telle méthode sans avoir besoin de savoir comment est exécutée cette action.

4.5.2 Les feuilles SPML : SPMLFeather

Une feuille SPML sert à représenter un objet SPML (instance de `spml_class` ou `spml_utility_field__`) avec des fonctionnalités associées. Elle porte des informations qui renseignent sur les modes d'intéractions possibles entre l'objet qu'elle représente et un utilisateur extérieur potentiel de cet objet. Une feuille SPML sert ainsi d'interface entre un objet SPML et le monde extérieur.

Une feuille SPML possède un identifiant (composé de deux parties : `id0` et `id1`), une valeur (un objet SPML), est liée à une feuille parente et peut avoir plusieurs feuilles enfants. A la création, une feuille est par défaut vide, c'est-à-dire sans valeur (`self.value = None`), sans feuille parent (`self.parent = None`) et sans feuille enfant (`self.children = []`). Une feuille possède aussi un dictionnaire de mappeurs de fonctions (`FunctionWrapper`) ou de méthodes (`BoundMethodWrapper`) : `self.methods`. Les clés sont les noms des mappeurs et les valeurs les mappeurs même.

L'API principale d'une feuille SPML est sa méthode `run(self,method,args,kw)` qui permet d'exécuter la méthode du même nom sur le mappeur de méthode passé en argument.

Une feuille SPML est l'élément de base d'un arbre « abstrait » dont le rôle et la structure sont définis dans la partie qui suit.

4.5.3 L'arbre abstrait : SPMLAbstractTree

Un arbre abstrait est constitué de feuilles SPML (`SPMLFeather`) représentant des objets SPML d'un environnement d'exécution donné et dont les relations parent-enfant traduisent les relations entre ces objets. Cet arbre abstrait est généralement initialisé par un interprète SPML avec une instance de `DATA_DBReader` (`self.reader`), une instance de `DATA_DBWriter` (`self.writer`) et une instance de `DATA_DBMetaReader` (`self.metaReader`). Durant l'initialisation sont aussi définis un dictionnaire `self.dictOfProjectedObject` (`={}`) et une feuille SPML vide `self.rootFeather` à laquelle on attribue le nom `'DATA'`. Cette feuille SPML sera la feuille parente de toutes les autres feuilles et constituera le point d'entrée pour accéder aux autres feuilles de l'arbre.

Un arbre abstrait joue le rôle d'interface ordonnée entre les objets SPML, représentés par des feuilles SPML, et un programme extérieur. L'analyse d'un arbre abstrait permet de représenter les objets dans un environnement d'intéractions avec un utilisateur, comme par exemple une interface graphique. Nous allons expliquer concrètement quel est le principe de la création d'un arbre abstrait.

L'idée est de d'abord créer une feuille SPML pour chaque instance de `spml_class` de l'environnement d'exécution. Pour cela, nous récupérons la liste de toutes ces instances grâce au DBReader :

```
objectList = self.reader.getObjects_P()
```

Ensuite, pour chaque objet de la liste, nous essayons de récupérer sa feuille SPML associée si elle existe. Cela est possible grâce au dictionnaire `self.dictOfProjectedObject` qui permet d'associer un objet à sa feuille. Si une feuille est trouvée, nous mettons à jour son identifiant :

```
# Create tree
for object in objectList:
  try:
    #update feather
    feather = self.dictOfProjectedObject[str(object.getUid())]
    #Set second part of feather id
    feather.setId1(object)
```

Si l'objet n'a pas de feuille associée, elle est créée et le dictionnaire `self.dictOfProjectedObject` ainsi que l'identifiant de la feuille sont mis à jour :

```
  except KeyError:
    #create feather
    feather = SPMLFeather(self.rootFeather)
    #Set first part of feather id
    feather.setId1(object)
    #add feather in dict
    #key is uid, value is feather
    self.dictOfProjectedObject[str(object.getUid())] = feather
```

La feuille nouvellement créée est définie comme étant une feuille fille de la feuille de base de l'arbre `self.rootFeather`.

Ensuite, après avoir retrouvé ou créé la feuille, nous la mettons à jour avec les informations de l'objet auquel elle est associée :

```
  #Set feather value
  feather.setValue(object)
  #Indicate that feather is projected
  feather.indicator = 1
  #Set feather SPML validity
  feather.setSPMLValidity(object.isValid())
```

L'attribut `feather.indicator` permet de savoir si une feuille a été mise à jour ou non.

Enfin, chaque feuille ainsi obtenue est projetée dans l'arbre abstrait :

```
  #Project object and its fields
  self.project(object,feather)
```

Cette projection consiste à créer d'autres feuilles SPML pour chacun des attributs SPML de l'objet, ainsi qu'à ajouter des mappeurs de méthodes aux feuilles. Le mécanisme de projection est détaillé dans l'annexe D.

Finalement, une fois tous les objets projetés dans l'arbre, le dictionnaire `self.dictOfProjectedObject` est mis à jour en supprimant les feuilles qui n'ont plus lieu d'être :

```
# Remove the feathers which are no more child
# of self.rootFeather
```

```
for key,feather in self.dictOfProjectedObject.items():
  if feather.indicator is 0:
    self.rootFeather.removeChild(feather)
    del self.dictOfProjectedObject[key]
```

4.5.4 La projection vers un environnement extérieur

Les objets SPML doivent pouvoir être représentés dans un environnement extérieur. Pour cela, nous proposons une démarche dont le principe repose sur la définition d'un nouvel arbre de projection héritant de l'arbre abstrait. Ce nouvel arbre doit être capable de projeter les objets SPML dans son environnement. Par exemple, pour pouvoir utiliser notre outil de description de la physique dans l'interface graphique de la plate-forme SALOME (cf chapitre 5), nous avons étendu la classe `SPMLAbstractTree` vers une nouvelle classe `SalomeTree`. La définition d'une méthode `self.projectSalome(self)` dans le nouvel arbre `SalomeTree` permet d'adapter la projection vers ce nouvel environnement. Toute la structure existe dans l'arbre abstrait mais il s'agit ici d'ajouter des particularités liées à la vue graphique (forme, couleur,...)

De nouveaux types de données peuvent aussi apparaître avec le nouvel environnement et des relations peuvent exister entre ces données et les objets SPML. Ainsi un type de données `SalomeRef` est apparu avec la plate-forme SALOME pour représenter des objets géométriques ou des objets de maillage.

L'arbre abstrait ne connaisant pas la structure de ces données, il ne sait pas les projeter. Par contre il suppose que ces données sont associées à un arbre de projection, lequel sera alors chargé de projeter ces données avec une méthode spécifique : `self.projectField(object,field,feather)`. Les arguments de cette méthode représentent l'objet SPML en cours de projection, l'attribut dont le type n'est pas connu par l'arbre abstrait, et la feuille `SPMLFeather` associée à l'objet en projection.

L'appel à cette méthode dans l'arbre abstrait est réalisé dans sa méthode `project(self,object,feather)` de cette manière :

```
def project(self,object,feather):
  #...
  for field in object.getKlass().getGenericFields():
    #...
    if (field.getStereotype() in ('identification',\
      'IDENTIFICATION')):
      #...
    else:
      #get the type of the field
      fType = field.getType()
      if fType.is_a("Intrinsic") or fType.is_a("Entity"):
        #
        # Projection standard
        #
      else:
        #
        # Field type is not in MetaModel:
        # call its projectField() method
        #
        self.projectField(object,field,feather)
```

Pour pouvoir projeter les données de type `SalomeRef`, l'arbre de projection `SalomeTree` pour la plate-forme SALOME implémente donc la méthode `self.projectField(object,field,feather)`, dont la structure ressemble beaucoup à la structure de la méthode `project(self,object,feather)`. De plus, les données `SalomeRef` ont été représentées par une nouvelle classe de feuille (`SalomeFeather`). Cette classe hérite bien sûr de `SPMLFeather`.

4.6 Réalisation du partage des données

Nous voulons offrir la possibilité d'accéder aux objets SPML en mémoire, depuis une application extérieure et à travers un bus de partage de données. Plusieurs spécifications existent et celle que nous utilisons est CORBA. Le chapitre 4.6.1, basé sur un article du site Easter-eggs [12], présente les principes, points forts et limites de CORBA. Ensuite le chapitre 4.6.2 montrera comment, sans connaître à l'avance leur structure de données, nous pouvons dynamiquement mettre à disposition les données à travers un bus CORBA.

4.6.1 Présentation de CORBA

4.6.1.1 Principe

CORBA (Corba Object Request Broker Architecture) est une spécification définie par l'OMG [13]. L'OMG est un consortium international réunissant un grand nombre d'entreprises. Ce consortium a été formé dans les années 80 pour réfléchir à des questions générales se posant de manière récurente dans l'industrie informatique : distribution des traitements entre machines, utilisation de plate-formes hétérogènes, standardisation d'une modélisation *objet* des processus ... La spécification CORBA constitue une tentative pour apporter des réponses à ces questions.
En particulier, CORBA définit :

- un langage IDL (Interface Definition Language) permettant la description d'objets CORBA. Chaque objet est décrit de manière externe par un « contrat » qu'il est censé remplir (ce « contrat » est écrit en langage IDL). De plus, CORBA spécifie la manière de « mapper » le langage IDL vers un grand nombre de langages informatiques existants actuellement (C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, et IDLscript),
- un protocole de communication commun ; la version IP de cet protocole est « IIOP » (Internet Inter-ORB Protocol).
- des Services de bases (Service de Nommage, Service de Persistance, Service de Transactions, ...) : ces services sont définis en langage IDL.

Exemples de Services :

- le Service de Nommage permet de mettre en place un système de nommage des objets CORBA hébergés par le système. Chaque objet peut alors être retrouvé par son nom ; il est inutile de connaître la localisation physique de l'objet (nom de machine, paramètres serveur ...).
- le Service de Transaction permet de mettre en place un mécanisme de transactions distribuées (similaire à une transaction sur une base de donnée, mais répartie sur plusieurs machines).

Une implémentation CORBA est une implémentation des spécifications CORBA sur une plate-forme donnée. Il existe un grand nombre d'implémentations CORBA. En principe,

chaque implémentation comprend :

- un compilateur d'IDL ; il permet de mapper le langage IDL vers un ou plusieurs langages cibles.
- un ORB (Object Request Broquer) : c'est lui qui traîte les requêtes CORBA. Il peut être vu comme un « bus » commun à tous les objets CORBA.
- quelques Services. En principe, toutes les implémentations incluent le Service de Nommage.

Exemples d'implémentations CORBA :

- propriétaires :
  - Iona Orbix [1]
  - ORBAcus [2] (Open Source)
  - VisiBroker [3]
- libres :
  - omniORB [4] (licence GPL/LGPL)
  - ORBit [5] (licence GPL/LGPL)
  - TAO [6]

4.6.1.2 Utilisations possibles de CORBA

- Spécifications d'une application. Ces spécifications peuvent être réalisées en langage UML par exemple (le langage UML est aussi une spécification de l'OMG).
- Description en langage IDL des objets CORBA constituant l'application.
- Choix techniques :
  - langages cibles pour les objets CORBA et les processus utilisateurs de ces objets,
  - plate-formes hébergeant les objets : machines, systèmes d'exploitation, réseaux,
  - choix d'une ou plusieurs implémentations CORBA.
- Codage : les objets décrits en langages IDL sont mappés vers leur langage cible. Chaque objet CORBA (« servant ») est implémenté dans son langage cible. Le code permettant d'utiliser les « servants » en mode client-serveur est complétement généré par le compilateur d'IDL.

4.6.1.3 Points forts de CORBA

- CORBA intègre toutes les étapes qui vont des spécifications formelles d'une application à son implémentation physique ; la description formelle d'une application est facilitée par le langage IDL.
- CORBA permet l'intégration d'applications déjà existantes avec des technologies nouvelles : par exemple, une application Python peut être « interfacée » par un objet CORBA et utilisée par des clients Java.
- Grande robustesse à l'utilisation ; par rapport à des solutions client-serveurs « classiques », les solutions CORBA sont plus robustes à l'utilisation et plus simples à administrer.
- Grande souplesse pour faire évoluer une application : en taille et en fonctionnalités.

[1] `http://www.iona.com/`
[2] `http://www.ooc.com/ob/`
[3] `http://www.borland.com/visibroker`
[4] `http://www.uk.research.att.com/omniORB/omniORB.html`
[5] `http://cvs.gnome.org/viewcvs/ORBit/`
[6] `http://www.cs.wustl.edu/~schmidt/TAO.html`

4.6.1.4 Limites de CORBA

- Technologie complexe et difficile à maîtriser.
- Le langage IDL offre une vision objet « limitée » (pas de surcharge) : ces limitations sont inhérentes au langage IDL et à la possibilité de mapper ce langage vers un grand nombre de langages informatiques.
- Mauvaise utilisation de CORBA dans les projets actuels : souvent, seul l'aspect « client-serveur » de CORBA est utilisé ; dans ce cas, mieux vaut utiliser des technologies plus « légères » (RPC, serveur socket ...).
- Les implémentations actuelles de CORBA sont souvent limitées en fonctionnalités : peu de services CORBA implémentés ; mappage de l'IDL vers un nombre restreint de langages de programmation (C, C++, Java) ; implémentations incompatibles les unes avec les autres ...

4.6.2 Le partage des données SPML

Les outils dédiés à la description des propriétés physiques que nous avons développés sont accessibles grâce à la définition de deux composants CORBA : `DATAModule` et `SPMLDBReaderWriter`. Ces composants mettent à disposition un certains nombre de services auxquels une application peut faire appel à travers un bus CORBA. Ces services sont décrits dans deux fichiers IDL : `DATA_Gen.idl` pour le premier, `DATA_SPMLDBReaderWriter.idl` pour le deuxième.

Le principe de ces modules est de permettre d'intéragir avec les objets SPML, comme par exemple la création ou la modification de données, mais ils permettent aussi une intéraction avec les modèles de données, en proposant des services de chargement et d'introspection de ces modèles.

4.6.2.1 Le composant SPMLDBReaderWriter

Ce composant permet tout simplement d'accéder au `DATA_DBReaderWriter` à travers une couche CORBA, et ainsi de pouvoir manipuler des objets SPML. Il est donc composé de quatre interfaces correspondants aux quatres classes de ce module :

- `DataDB`
- `DataDBReader`
- `DataDBWriter`
- `DataMetaDBReader`

Ces quatres interfaces proposent un accès aux méthodes publiques de leurs classes respectives (cf §4.4).

4.6.2.2 Le composant DATAModule

Ce composant est créé pour chaque application qui existe. Il permet d'effectuer des actions au niveau des modèles et de l'arbre abstrait associés à cette application. Il permet aussi de récupérer les instances des `DataDBReader`, `DataDBWriter` ou `DataMetaDBReader` liés à l'application.

Plusieurs application pouvant être créées, nous identifions chacune d'entre elles par un numéro d'étude. Concrètement, une étude consiste généralement en la résolution d'un problème, avec les étapes de description de la géométrie, de maillage, de description des propriétés physiques, de résolution et de traitementdes résultats.

Pour pouvoir expliquer le rôle de ce composant, nous allons suivre les différentes étapes qui composent la partie description des propriétés physiques pour une étude. En effet, les méthodes que propose ce module, au travers de l'interface `DATA_Gen`, vont permettre à l'utilisateur de pouvoir décrire la physique de son problème et de manipuler les données qu'il aura créées, ceci à n'importe qu'elle étape du processus de résolution de son problème.

1. Définition d'une application :
   Une application est définie pour un problème donné. Elle est composée de modèles de données physiques qui permettent de décrire la structure de ces données. Le modèle de l'application, qui rassemble tous ces modèles, est alors chargé par notre composant. Les méthodes relatives à cette étape sont les suivantes :
   - `string importSPMLModel(in string ApplicationFileName,in long studyId)`
     Charge un modèle de données en mémoire. Le modèle est stocké dans le fichier `ApplicationFileName` et est associé à l'étude n°`studyId`.
   - `ListOfString getModelNames(in long studyId)`
     Renvoie la liste des modèles définis dans l'application.
   - `string getParentModelName(in long studyId,in string modelName)`
     Renvoie le modèle parent du modèle donné.
   - `ListOfString getClassEntityNameList(in long studyId,in string modelName)`
     Renvoie la liste des classes définies dans un modèle.
   - `void setCurrentStudyId(in long studyId)` Définit un numéro d'étude pour l'application en cours (une application = une étude)
   - `long getCurrentStudyId()`
     Renvoie le numéro d'étude en cours

2. Démarrage des services :
   Une fois que le modèle de l'application est chargé par notre composant, la structure de données de l'étude en cours est alors configurée et plusieurs services sont démarrés :
   - une instance de console évoluée `SPMLEnhancedConsole` (cf §4.3.4) est créée et configurée avec le modèle de données chargé au début.
   - des instances de `DataDBReader`, `DataDBWriter` et `DataMetaDBReader` sont créées par la console. Le composant CORBA permet de les récupérer :
     - `DataDBReader getDBReader(in long studyId)`
     - `DataDBWriter getDBWriter(in long studyId)`
     - `DataMetaDBReader getMetaDBReader(in long studyId)`

3. Mise à disposition de la structure de donnée sur le bus CORBA :
   Pour pouvoir manipuler les données physiques à travers une couche CORBA, il faut que la structure de ces données soit décrite dans un fichier IDL. Ce dernier est donc généré dynamiquement, ainsi que son implémentation en Python, au chargement du modèle. Le module CORBA permet de retrouver le nom du fichier IDL ainsi généré :
   - `string getModelIDLFileName(in long studyId)`
     Renvoie le nom du fichier IDL associé au modèle de l'étude n°`studyId`

   Pour accéder au données physiques, nous proposons de passer par un point d'entrée unique : un jeu de données. Ce jeu de données doit être une instance de `PhysicalDataSet`, classe qui doit donc être définie dans le modèle de données et qui rassemblera les propriétés physiques d'un problème. Le module

met ensuite un composant à disposition des utilisateurs ; c'est une instance de `DATA_PDSReader` et s'enregistre dans l'annuaire CORBA sous le chemin `DATASer-vices/DATA_PDSReader__"+str(studyId)`. Il permet de récupérer tous les jeux de données qui auront été créés grâce à la méthode `def getPhysicalDataSetList( self)`.

4. Manipulation des données :
   La manipulation des données se fait par l'intermédiaire de l'arbre abstrait et des composants d'accès à la base de données. Les méthodes disponibles sont alors les suivantes :
   - `void displayObjectBrowser(in long studyId)`
     Met à jour l'arbre abstrait et le projette vers son environnement de projection (interface graphique par exemple)
   - `ListOfString run(in string idFeuille,in long studyId, in string methodName, in ListOfString args)raises (ExecutionError)`
     Execute une action sur un objet ou sur un attribut. Les actions possibles sont déterminées par les mappeurs de méthodes attachés aux feuilles SPML représentant ces objets ou attributs.
   - `ListOfString getMethods(in string idFeather,in long studyId)`
     Renvoie la liste des mappeurs de méthodes disponibles sur un objet ou un attribut.
   - `ListOfString getMethodsArgs(in string idFeather,in string methodName,in long studyId)`
     Renvoie la liste des arguments d'un mappeur de méthode.
   - `ListOfString getException(in long studyId)`
     Renvoie la dernière exception levée.
   - `boolean saveDataInPythonScript(in string fileName,in string mode,in long studyId)`
     Sauvegarde les données dans un script Python.
   - `boolean loadPythonScript(in string fileName,in long studyId)`
     Charge des données sauvegardées dans un script Python.

4.7 Conclusion

Ce chapitre a permis de comprendre comment ont été réalisés le métalangage SPML et ses fonctionnalités associées. Nous nous sommes basés sur un langage existant, Python, pour obtenir un langage de programmation orienté objet puissant et destiné à la description de modèles de données physiques. Le fait de se baser sur un langage existant permet de bénéficier d'une structure complète et de réduire les coûts de développement pour la réalisation des fonctions indispensables à tout langage de programmation (gestion de la mémoire, interprète de commandes, ...).

L'écriture de modèles de données avec notre langage permet de bénéficier d'une gestion évoluée des données :

- accès aux données contrôlé par les composants `DBReader`, `DBWriter` et `MetaDBReader`,
- projection des données dans un environnement extérieur grâce à l'arbre abstrait,
- intéractions entre l'utilisateur et les données depuis un environnement extérieur

prises en charge par l'arbre abstrait,
– accès aux données possible dans une application répartie avec une communication de type client-serveur basée sur le protocole CORBA.

Notre langage a été utilisé dans un composant dédié à la description de propriétés physiques au sein d'une plate-forme de simulation numérique : SALOME. Cette plate-forme est présentée dans le chapitre 5, dans lequel nous présentons aussi l'intégration de solveurs et la résolution de problèmes, couplés ou non, dans cette plate-forme.

Chapitre 5

Application au sein d'une plateforme de simulation numérique : SALOME

Sommaire

5.1 Introduction

Le langage SPML et tous ses outils associés forment un moteur de description des propriétés physiques générique, qui peut être utilisé pour n'importe quel modèle de données physiques. Ce moteur fait partie intégrante de la plate-forme de simulation numérique open-source SALOME dans laquelle il est chargé de gérer l'apect « physique » du processus de modélisation classique (géométrie, maillage, physique, résolution et post-traitment).

La partie 5.2 est consacrée à la présentation de la plate-forme SALOME : nous exposerons son principe de fonctionnement en modules répartis et nous nous intéresserons particulièrement au module DATA consacré à la description de la physique.

Ensuite la partie 5.3 sera l'occasion d'expliquer la méthode d'intégration d'un solveur dans la plate-forme : nous montrerons comment nous avons encapsulé un solveur magnétique dans un module pour la résolution d'un problème simple de magnétostatique. Le solveur en question ne proposant pas d'API qui permette de l'intégrer directement, nous montrerons comment nous avons mis en place une couche réseau de haut niveau pour pouvoir dialoguer avec le logiciel intégrant le solveur. Cette méthode illustre très bien les possibilités de résolution de problèmes multiphysiques avec des solveurs placés sur des machines déportées.

Enfin la partie 5.4 traitera d'une intégration complète d'un solveur de calculs d'intéractions fluide-stucture : description de son modèle de données avec le module DATA, puis calcul de la déformation d'une plaque excitée par un détachement tourbillonnaire. Ce solveur fournit une API et une librairie qui permettent de l'intégrer directement dans la plate-forme.

5.2 Introduction à la plate-forme Salome

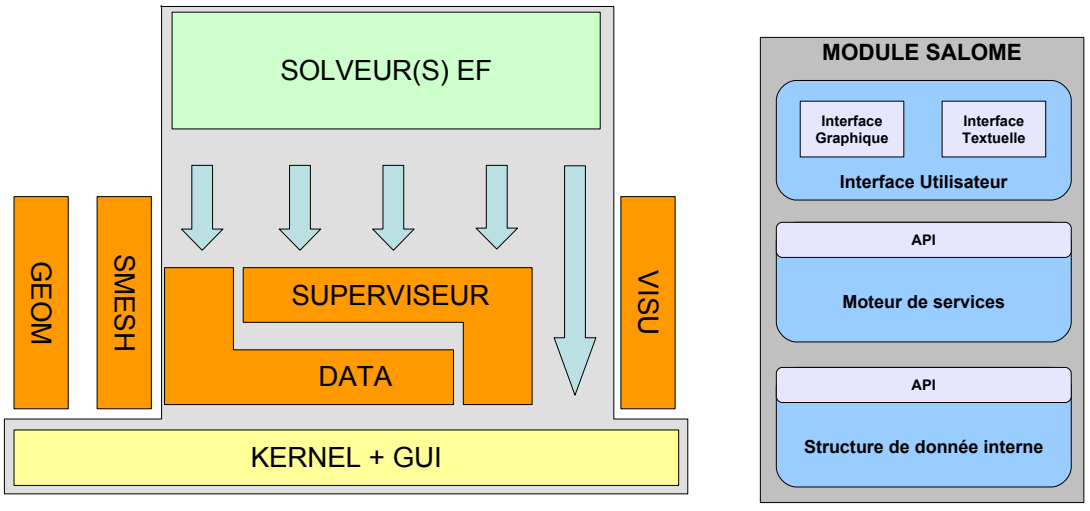
5.2.1 Présentation générale

La plate-forme SALOME [6] est née dans le courant de l'année 2000 suite à un constat de manque de solution logicielle répondant à certains besoins :

1. besoin d'une solution pour la résolution de problèmes multiphysiques qui soit efficace, pour laquelle les couplages multiphysiques seraient simplifiés, et qui offrirait un accès simple aux utilisateur,
2. besoin de faciliter l'intégration de logiciels de calculs existants au sein d'une solution unique et d'offrir une interopérabilité entre codes de calculs.

A partir de ce constat a débuté en septembre 2000 le projet RNTL pour le développement de la plate-forme SALOME. Cette plate-forme est conçue pour offrir un environnement global afin de résoudre des modèles numériques multiphysiques et de favoriser la liaison CAO-calcul. Elle regroupe les éléments non critiques d'une application de conception assistée par ordinateur (CAO) et de simulation. Elle fournit aussi les éléments de description de donnés physiques d'une manière unifiée et les interfaces nécessaires pour la descripion des modèles physiques et pour l'exploitation des résultats (pré et post-processing) [6, 14].

Techniquement, la plate-forme se présente sous la forme de composants logiciels – des modules – construits de façon à permettre une intégration aisée d'algorithme de maillages et de solveurs existants ; elle repose sur une architecture d'applications réparties, c'est-à-dire que les modules qui la composent montrent un grand niveau d'indépendance, tout en gardant une cohérence dans leurs réalisations. Ces composants communiquent entre eux selon un schéma de relations client-serveur et selon le protocole standard CORBA [15, 16, 13].



(a) Organisation de la plate-forme SALOME

(b) Structure interne d'un module SALOME

FIG. 5.1 – La plate-forme SALOME : une organisation en modules

La figure 5.1(a) représente un synaptique de l'organisation des différents modules qui composent la plate-forme SALOME. Chaque module a un rôle bien précis (géométrie,

mailage, ...) qui correspond aux différentes étapes d'un processus de modélisation classique. La documentation de ces modules est disponible sur le site de la plate-forme [6] et se destine aussi bien aux utilisateurs finaux qu'aux utilisateurs intégrateurs chargés de développer des modules pour l'intégration de solveurs.

Parmi ces modules, le module DATA permet de spécifier les propriétés physiques (propriétés de matériaux, conditions aux limites, conditions initiales) liées à la résolution des simulations numériques. Pour cela, il se base sur le métamodèle que nous avons développé, c'est-à-dire le langage SPML et les outils de gestion et de projection des données qui l'accompagnent. La structure de ce module est conforme aux spécifications de la plate-forme et se compose de trois principaux composants (figure 5.1(b)) :

- une interface utilisateur graphique (DATAGUI),
- un moteur qui implémente les services de description des propriétés physiques,
- une structure de donnée interne proposant les services de persistance et de transfert de données entre modules.

5.2.2 Fonctionnement du module de description des propriétés physiques : DATA

Le module DATA est donc une application du métamodèle dans la plate-forme SALOME. En ce sens, il doit pouvoir offrir une intéraction avec les autres composants tels que le module chargé de la description géométrique et celui qui s'occupe du maillage. Ainsi dans l'exemple donné dans la partie 3.6, la définition de la classe SPML `Region` montre qu'elle possède un attribut `support` dont le rôle est d'associer un support géométrique à une région. Ce support géométrique sera représenté dans notre cas par un objet de la plate-forme SALOME, et plus particulièrement du module GEOM. Le métamodèle a donc été étendu pour être utilisé dans cet environnement et propose le nouveau type intrinsèque `spml_salomeRef` pour représenter les objets de la plate-forme. Cette extension s'est répercutée sur tous les outils (interprète, projection, gestion des données) qui ont alors été étendus à leur tour de manière à prendre en compte ce nouveau type.

Du point de vue de l'utilisateur final, c'est-à-dire celui qui utilisera un solveur intégré à la plate-forme pour résoudre son problème numérique, le module DATA a été conçu pour être très simple à prendre en main. Une fois les phases de géométrie et maillage passées, les étapes à suivre pour la partie « physique » du processus de modélisation sont les suivantes :

- Configurer le module avec le modèle de données physiques du problème traité. Cette configuration consiste à lire un fichier SPML dans lequel le modèle est complétement décrit. La structure du modèle est alors chargée dynamiquement en mémoire grâce à l'interpréte SPML comme expliqué dans la partie 4.3.
- Décrire les propriétés physiques de l'étude en créant modifiant les objets nécessaires proposés par le modèle (Région thermique, matériau diélectrique, ...).
- Rassembler toutes ces propriétés physiques dans un objet représentant un jeu de données : un `PhysicalDataSet`, dont la structure doit bien sûr être précisée dans le modèle.

Toutes ces actions se font par l'intermédiaire de boîtes de dialogues et de menus contextuels dont le contenu est dynamiquement rempli avec les informations utilisateurs associés aux objets SPML.

Cette simplicité a un coût, et celui-ci se place au niveau de l'intégration du solveur, où réside la difficulté. Cette tâche est confiée à un utilisateur intégrateur dont le rôle est multiple :

- il est d'abord chargé de décrire le modèle de données du solveur en utilisant le langage SPML (la phase la plus compliquée restant plutôt celle de la conceptualisation que celle de l'écriture à proprement parler),
- il lui faut ensuite créer un module qui encapsulera le solveur dans la plate-forme et dont la structure sera conforme à celle présentée dans la figure 5.1(b). Ce module sera chargé au minimum de configurer et de démarrer le solveur avec des paramètres de simulations donnés, mais surtout avec les propriétés physiques déclarées dans l'étude.

La partie qui nous concerne est celle de la récupération des propriétés physiques, néanmoins nous pouvons assister les intégrateurs à créer leur module, ce processus étant très redondant d'un solveur à l'autre.

La récupération des données est réalisée par le biais du réseau CORBA, puisque c'est le mode de communication inter-modules dans SALOME. Un avantage de ce type de communication est la possibilité d'interroger un serveur situé à un autre endroit géographique, et ainsi de pouvoir disposer des capacités de calcul d'un site distant.

Quand l'utilisateur final a chargé son modèle de donnée, le module DATA a automatiquement créé un fichier IDL représentant la structure de ce modèle et à lancé un serveur `PDSReader` pour accéder à cette structure à travers une couche CORBA. Ce serveur ne propose qu'une seule méthode, `getPhysicalDataSetList()`, dont le rôle est de retrouver et de récupérer toutes les instances des jeux de données `PhysicalDataSet` dans l'étude. On voit ici la nécessité de créer cette structure dans le modèle de données. Il suffit alors ensuite d'itérer sur toutes les instances de `PhysicalDataSet` ainsi retrouvées (il y en a généralement qu'une seule) et de récupérer les valeurs des attributs qui sont les propriétés physiques de notre problème (régions, conditions limites, ...). L'exemple suivant illustre toutes ces étapes :

1. Récupération d'un objet CORBA correspondant au serveur `PDSReader` :

```
# Get the PhysicalDataSet Reader
name = [CosNaming.NameComponent("DATAServices","dir"),
        CosNaming.NameComponent("DATA_PDSReader__"+str(studyId
           ),"object")]
try:
  obj = self._nameService.resolve(name)
except CosNaming.NamingContext.NotFound, ex:
  print "DATA_PDSReader not found in Naming Service"
  raise SALOME.SALOME_Exception("DATA_PDSReader not found in
     Naming Service")
```

2. Import de l'implémentation Python du fichier IDL du modèle de données :

```
# Import the IDL file
try:
  exec "import "+ self.idlFileName
  print "IDL file",self.idlFileName,"imported"
except Exception,e:
```

```
  print "Error: the IDL file",self.idlFileName,"could not be
     loaded"
  print e
  raise e
```

3. On réalise une opération de tranformation de type de l'objet CORBA en `PDSReader` :

```
# Get a reference on the PDSReader
try:
  exec "pdsReader = obj._narrow("+self.idlFileName+".PDSReader
     )"
except Exception,e:
  print "Exception with command: exec pdsReader = obj._narrow(
     "+self.idlFileName+".PDSReader)"
  print e
  raise e
print "pdsReader: ",pdsReader
```

4. Récupération et parcourt de la liste des jeux de données :

```
# Get the list of Physical DataSets
self.pdsList = pdsReader.getPhysicalDataSetList(); #is a
   ListOfPhysicalDataSet
print "pdsList: ",self.pdsList

# Parsing the PDS list
for pds in self.pdsList:
```

5. L'attribut `regions` est accessible avec la méthode `_get_regions()` (syntaxe CORBA) :

```
  try:
    listOfRegions = pds._get_regions()
  except Exception,e:
    print "Exception while getting regions of pds:",e
    return
  print "List of regions:",listOfRegions
```

Le solveur est alors alimenté par les données physiques ainsi récupérées ; la manière dont le solveur est renseigné dépendant étroitement de sa structure, à ce stade du processus de modélisation aucune méthode générique ne peut être donnée pour réaliser cette fonction. C'est pourquoi nous présentons dans la suite deux exemples d'intégration de solveurs dans la plate-forme, selon des méthodes adaptées à leurs structures. Ces exemples concernent l'intégration d'un solveur magnétique basé sur le code de calcul FLux [4], puis l'intégration d'un solveur de calcul des intéractions fluide-structure développé par le CSTB [17].

5.3 Intégration d'un solveur magnétique

5.3.1 Définition du problème

Nous souhaitons résoudre un problème simple de magnétostatique 3D : calculer la répartition d'un champ magnétique dans une circuit magnétique avec la présence d'un

aimant. La géométrie de la pièce est donnée dans la figure 5.2 et les dimensions dans le tableau 5.1. Les propriétés des matériaux sont données dans le tableau 5.2.

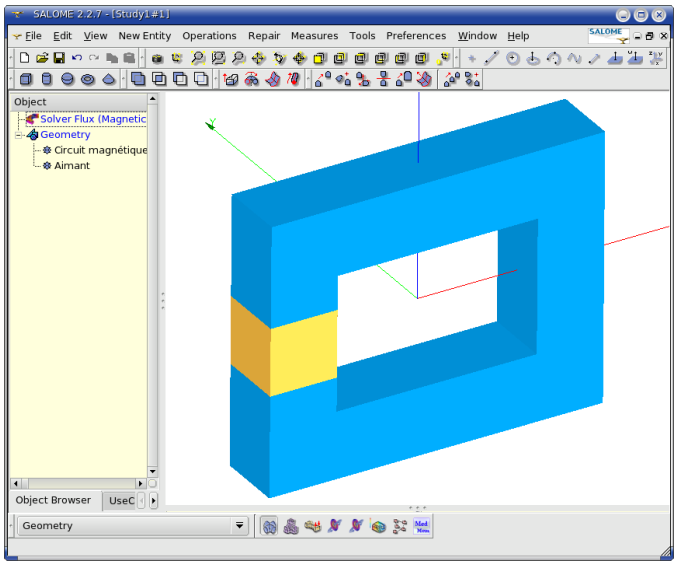


FIG. 5.2 – Vue 3D de la géométrie du problème

	Aimant			Circuit magnétique	
	Min	Max		Min	Max
X	0	100	X	0	500
Y	0	100	Y	0	400
Z	150	250	Z	0	100

TAB. 5.1 – Dimensions de la pièce d'acier

Aimant				Circuit magnétique	
perméabilité		aimantation résiduelle		perméabilité	
μ_{rX}	1	b_{rX}	0	μ_r	100
μ_{rY}	1	b_{rY}	0		
μ_{rZ}	1	b_{rZ}	1.5		

TAB. 5.2 – Propriétés des matériaux

5.3.2 Principes de l'encapsulation du solveur

Nous voulons résoudre ce problème avec le logiciel Flux configuré avec le modèle magnétostatique 3D. Ce solveur ne propose pas d'API directement accessible que l'on pourrait utiliser depuis la plate-forme pour le piloter. Cependant, il existe une autre méthode qui consiste à demander à Flux d'exécuter des commandes que l'on écrit dans un fichier. Les commandes ainsi envoyées sont des commandes Python. Nous nous baserons sur cette méthode pour piloter le logiciel Flux depuis SALOME.

Un autre point de difficulté à mentionner est le fait que le solveur n'est physiquement pas installé sur la même machine que SALOME. Nous avons donc décidé de faire passer les commandes à travers le réseau informatique. Pour cela, un serveur est installé sur la

machine Flux ; ce serveur est en attente de commandes qui lui sont envoyées depuis la machine SALOME, puis quand il en reçoit une, il demande à Flux de l'exécuter. Nous ne rentrerons pas ici dans les détails de cette configuration client-serveur, l'important étant de savoir qu'il est possible d'envoyer des commandes au logiciel Flux depuis une machine déportée, et de savoir si cette commande a été correctement réalisée ou non.

5.3.3 Principes du modèle de données

Pour pouvoir configurer et piloter FLux, il faut définir un modèle SPML reflétant la structure du modèle de données utilisé (Magnétostatique 3D). Nous ne détaillons pas ici ce modèle qui est assez compliqué, mais nous expliquons comment nous avons utilisé la possibilité de définir des attributs dérivés pour générer les commandes pour Flux.

L'idée est de définir, pour toutes les classes du modèle, un même attribut SPML qui contiendra une commande Flux. Le mode de définition de cet attribut est `DERIVED`. Nous utilisons ensuite les méthodes `spml_xxx_ascontrols__` pour réévaluer cette commande après chaque modification d'un attribut SPML. Par exemple, nous avons défini dans le modèle une classe représentant une propriété $B(H)$ linéaire isotrope caractérisée par une perméabilité μ_r :

```
class LinearIsotropic(spml_class):
  spml_modelowner__ = propertyBHModel
  spml_stereotype__ = "CONCRETE"
  spml_uicontrols__ = spml_ui_entity_t(
    defaultLabel='Linear isotropic',
    defaultComment= Isotrop Linear with relative permeability.\n')
  #
  # field 'fluxString' definition
  #
  spml_fluxString = None
  spml_fluxString_controls__ = {"relatedType":DataString,
    "definitionMode":"DERIVED",
    "uiInformation":spml_ui_attribute_t(
      defaultLabel='Flux String',
      defaultComment='This is the corresponding Flux String.\n')}
  #
  # field 'mur' definition
  #
  spml_mur = None
  spml_mur_controls__ = {"relatedType":DataFloat,
    "uiInformation":spml_ui_attribute_t(
      defaultLabel='Relative Permeability',
      defaultComment='This is the relative permeability'+
                     ' of the material.\n')}
  #
  # An after-set control function for mur
  #
  def spml_mur_ascontrols__(self,value):
    try:
      self.fluxString = self.getFluxString()
    except Exception,e:
      print e
      pass
  def getFluxString(self):
    return "PropertyBhLinear(mur='%s')"%str(self.mur)
```

Après une modification de l'attribut `mur`, la méthode `spml_mur_ascontrols__` est exécutée et met à jour la valeur de l'attribut `fluxString`.

Les classes SPML du modèle de données portent ainsi sur elles, à travers leur attribut `fluxString`, la commande Flux permettant de définir les propriétés physiques qu'elles définissent. Cette commande est automatiquement mise à jour après toute modification.

5.3.4 Le composant magnétostatique

Le solveur Flux est représenté dans la plate-forme par un module SALOME. Ce module est chargé au démarrage de SALOME et il permet de réaliser plusieurs choses :

- configurer les paramètres réseau (nom d'hôte, numéro de port) pour communiquer avec le serveur de commandes de Flux (figure 5.3),
- envoyer des commandes Flux « à la main »,
- configurer le solveur avec les paramètres physiques de l'étude et lancer la résolution.

Les figures 5.4 et 5.5 présentent des captures d'écran de ces différentes étapes.

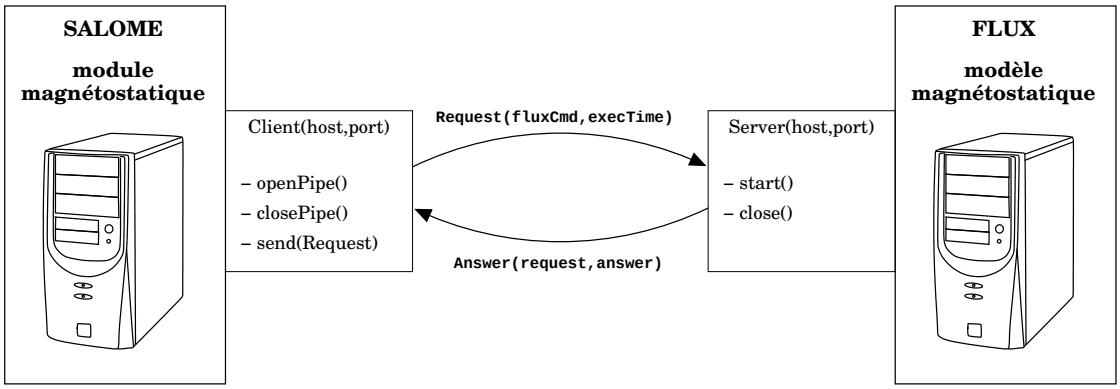
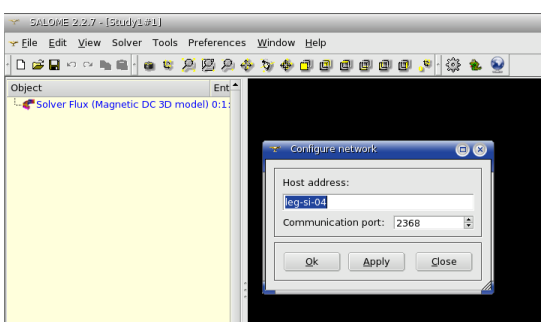
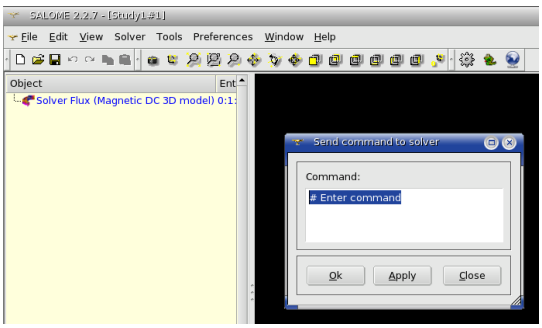


FIG. 5.3 – Composant magnétostatique – Principe de communication avec le solveur : une relation de type client-serveur est établie. Des objets « Request » et « Answer » sont échangés entre eux. L'objet « Request » porte la commande « `fluxCmd` » que Flux doit exécuter.

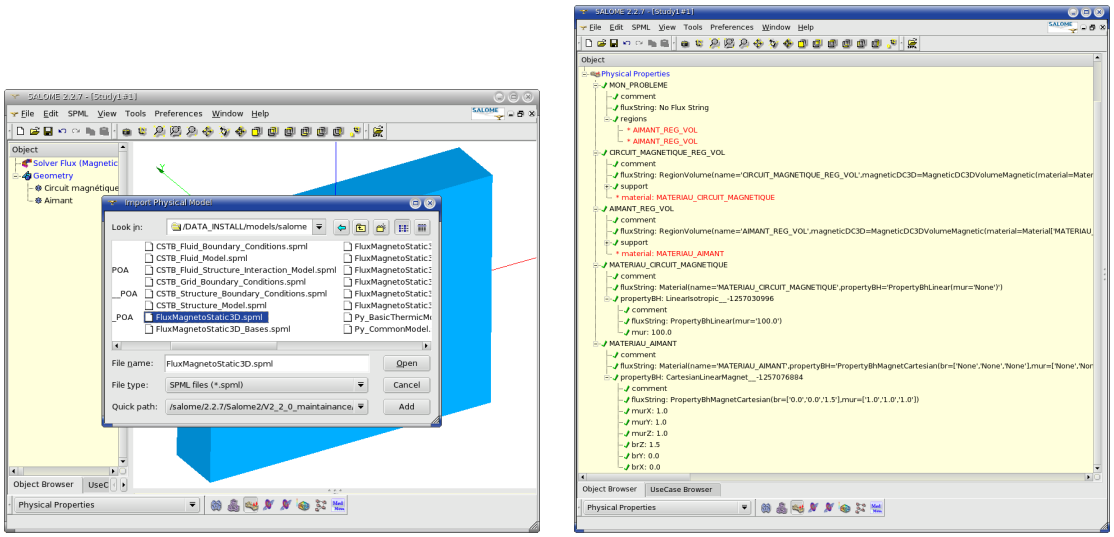


(a) Configuration des paramètres réseaux



(b) Edition de commande Flux

FIG. 5.4 – Configuration des paramètres réseau et édition de commande Flux pour le composant solveur magnétostatique 3D dans SALOME



(a) Chargement du modèle de données SPML (b) Edition des paramètres physiques

FIG. 5.5 – Définition des propriétés physiques du problème avec le module DATA

5.3.5 Résultats de simulation

La figure 5.6 montre le résultat de simulation avec la répartition du champ magnétique dans le circuit magnétique.

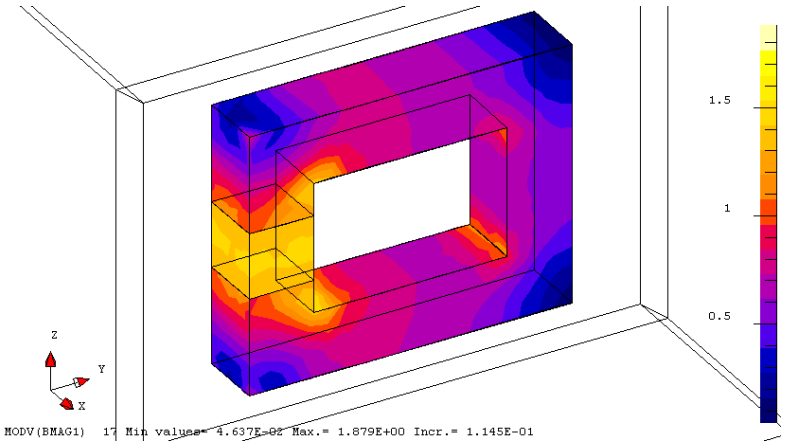


FIG. 5.6 – Répartition du champ magnétique dans le circuit magnétique

Le post-traitement est effectué dans le logiciel FLux3D. La plate-forme SALOME dispose elle aussi d'un module de post-traitement des résultats mais elle se base sur un format de fichier particulier (le format MED [18]) que Flux ne sait pas (encore ?) gérer. Nos tentatives d'ajouter la librairie MED à l'exécutable de Flux3D n'ont pas abouti ; la cause est sûrement une incompatibilité entre les versions de Fortran utilisés par ces programmes (Flux3D est basé sur Fortran77 alors que la librairie MED propose une API basée sur Fortran90).

5.4 Résolution d'un problème d'intéraction fluides-structures

Ce problème nous a été soumis par le CSTB qui était intéressé par la plate-forme SALOME et qui désirait utiliser le module DATA pour décrire les propriétés physiques pour la résolution de problèmes d'intéractions fluide-structure. Le problème qui est traité ici est un cas test qui permet de prédire le comportement dynamique d'une plaque élastique située dans le sillage d'un cylindre à section carrée. A fort nombre de Reynolds ($Re > 200$) des lâchés tourbillonnaires se produisent dans le sillage du cylindre à une certaine fréquence et excitent la plaque qui se met alors à osciller.

5.4.1 Définition du problème

Soit une plaque plane déformable fixée derrière un cylindre rigide et immobile de section carrée ; les deux sont plongés dans un écoulement de fluide visqueux incompressible. Toutes les dimensions et grandeurs physiques et numériques du problème sont représentées sur la figure 5.7 et le tableau 5.3 p.108.

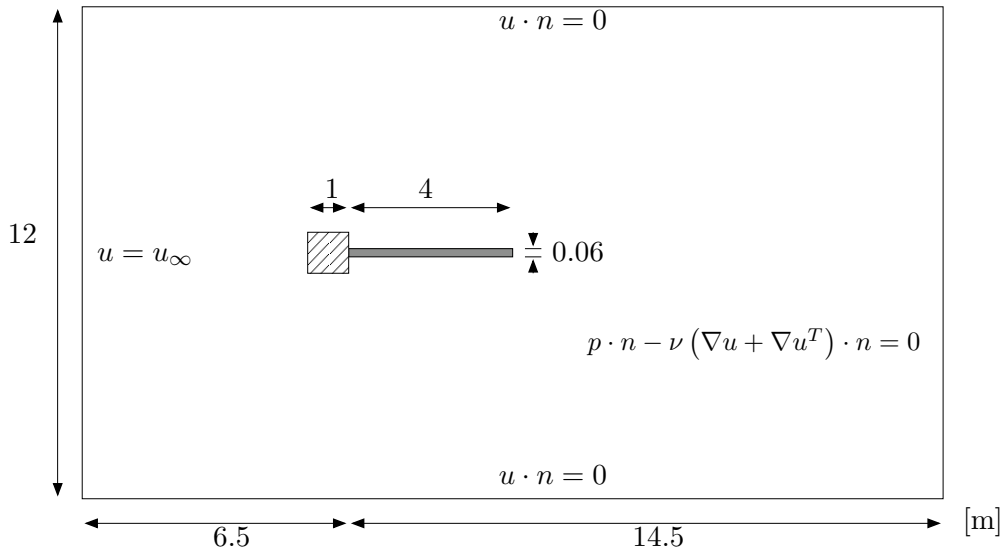


FIG. 5.7 – Plaque excitée par détachement tourbillonnaire – Définition du problème

5.4.2 Principes de l'encapsulation du solveur

Le solveur utilisé par le CSTB doit pouvoir être configuré et lancé depuis la plate-forme SALOME, c'est-à-dire, en ce qui nous concerne, depuis un programme Python. Or, le solveur a été développé avec le langage C, puis compilé dans un exécutable. Ce programme n'était pas fait au départ pour recevoir des commandes extérieures, et travaillait avec des fichiers dans lesquels étaient stockées les valeurs des paramètres, la géométrie et le maillage. C'est cette rigidité qui a conduit le CSTB a vouloir utiliser SALOME pour la définition de la géométrie, du maillage et des paramètres physiques, mais aussi pour l'exploitation des résultats.

Il est possible de faire appel à un programme C depuis un programme Python : cela nécessite de mapper les fonction C du programme vers un nouveau module Python [19]. Il est possible de le faire manuellement, mais il existe aussi des outils permettant de réalisser ce mapping (boost [20], swig [21]). Nous avons utilisé le programme SWIG qui

domaine fluide	3956 nœuds – 7636 éléments
domaine solide	621 nœuds – 998 éléments
éléments finis	triangles P1/P1
Formulation Navier Stokes	GLS
Modélisation de la turbulence	Filtre de Smagorinsky
Solveur linéaire	BICGSTAB
Préconditionneur	ILU
Convergence du solveur	$\|\text{résidu}\| < 10^{-8}$
Convergence non linéaire	$\|\text{résidu}\| < 10^{-4}$
Convergence vers état stationnaire	$\frac{\|u^{n+1}-u^n\|}{\|u^n\|} < 10^{-10}$
Viscosité dynamique du fluide μ	1.82×10^{-3}
Densité du fluide	1.18×10^{-3}
Densité du solide	0.1
Module de Young	2.5×10^{6}
Coefficient de Poisson	0.35
Fréquence du premier mode propre	3.73
Vitesse du fluide en amont	51.3
Pas de temps de calcul	0.001

TAB. 5.3 – Plaque excitée par des tourbillons – Paramètres de calcul

permet, à partir d'un fichier dans lequel sont définis les fonctions à mapper, d'obtenir un fichier en C correspondant au mapping du programme vers Python. La compilation de ce fichier nous donne ensuite une librarie (fichier `.so`) considéré par Python comme un module, et que nous pouvons alors importer et utiliser depuis notre programme Python.

5.4.3 Le modèle de données

Le modèle de données SPML a été défini à partir du diagramme de classes de la figure F.1 en annexe F. Les différentes captures d'écran des figures 5.8, 5.9 et 5.10 illustrent les différentes entités que l'on peut créer dans le module DATA après avoir chargé le modèle SPML.

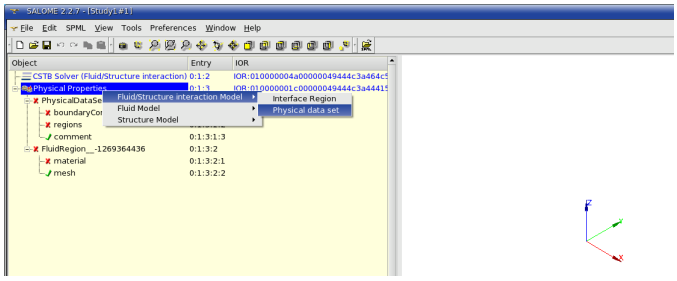


FIG. 5.8 – Plaque excitée par des tourbillons – Définition de la physique – Classes du modèle principal

5.4.4 Le composant solveur d'intéractions fluide-structure

Les équations régissant le mouvement de la plaque et l'écoulement du fluide sont résolues par un seul solveur. Nous sommes donc dans une situation de démarche globale

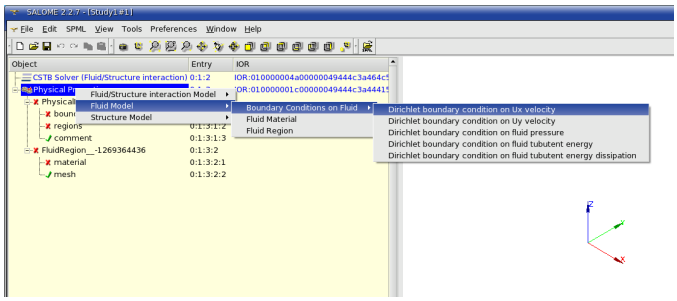


FIG. 5.9 – Plaque excitée par des tourbillons – Définition de la physique – Classes du modèle du fluide

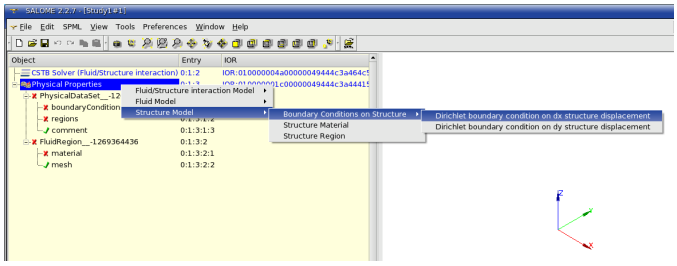


FIG. 5.10 – Plaque excitée par des tourbillons – Définition de la physique – Classes du modèle de la structure

(cf §2.6) pour laquelle il n'est pas nécessaire d'utiliser de composant de supervision pour itérer entre deux solveurs et échanger des données.

De la même manière que pour l'exemple précédent, ce solveur a donc été encapsulé dans un composant SALOME. Il permet de modifier les paramètres de simulations pour le solveur depuis une fenêtre de dialogue ; ces même paramètres peuvent aussi être récupérés depuis un fichier. Il permet bien sûr de lancer la simulation avec les paramètres et les propriétés physiques donnés. Si des paramètres physiques manquent (pas de région fluide ou solide), la résolution ne se lance pas et un message d'avertissement est affiché (figures 5.11 et 5.12).

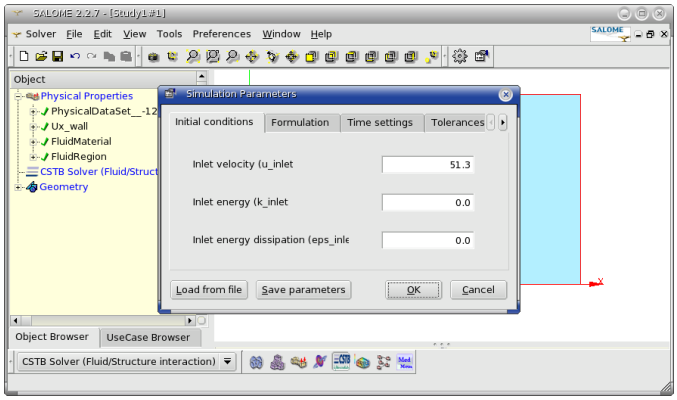


FIG. 5.11 – Composant SALOME pour le calcul d'intéractions fluide-structure – Modifications des paramètres du solveur

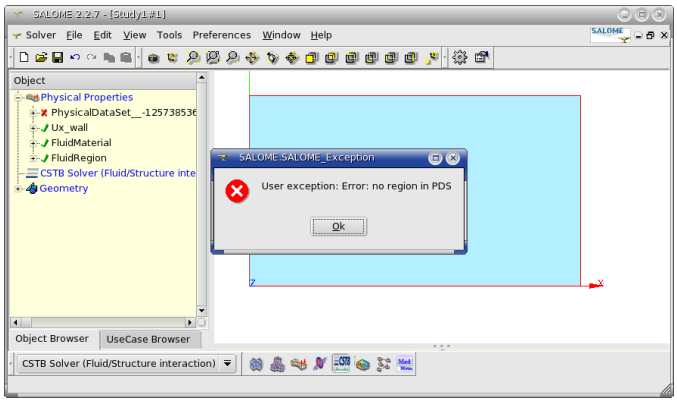


FIG. 5.12 – Composant SALOME pour le calcul d'intéractions fluide-structure – Message d'erreur : la physique n'est pas complète

5.4.5 Résultats de simulation

Après avoir défini la géométrie et la physique du problème, celui-ci a été résolu. La figure 5.13 montre la déformation de la plaque et donc du maillage. Les tracés des lignes de pression et des vecteur de vitesse obtenus sont présentés sur les figures 5.14 et 5.15. A noter que l'exploitation des résultats est entièrement réalisée avec le module de post-traitement VISU de SALOME. Les figures 5.16 et 5.17 présentent les maillages des régions du fluide et de la plaque.

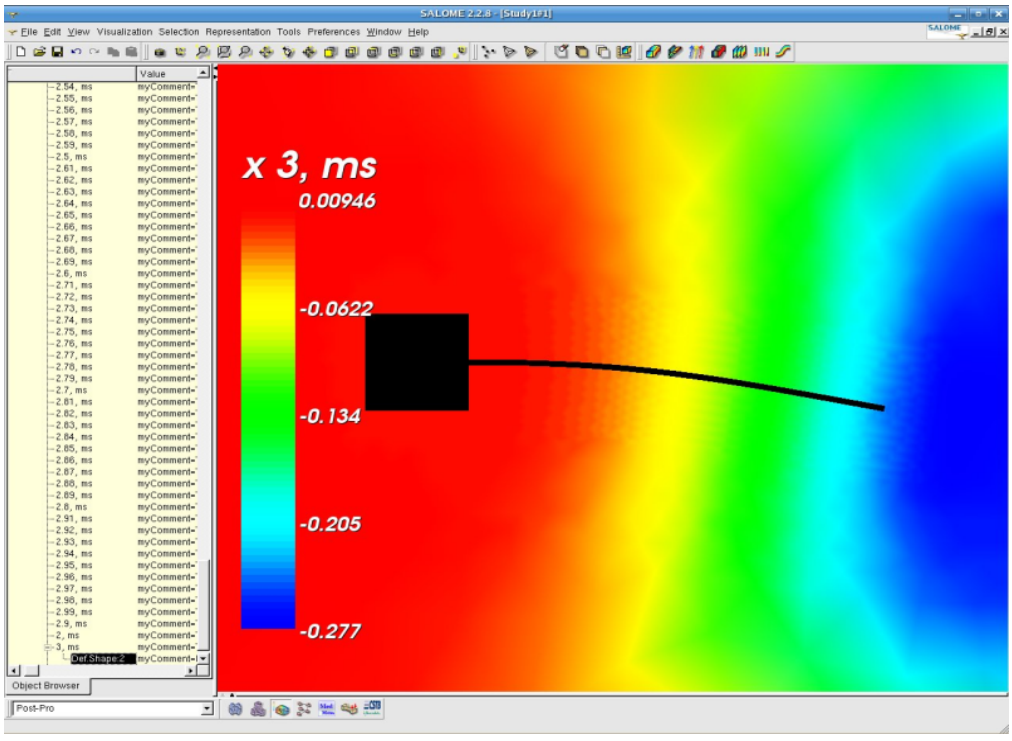


FIG. 5.13 – Calcul d'intéractions fluide-structure – Déformation du maillage

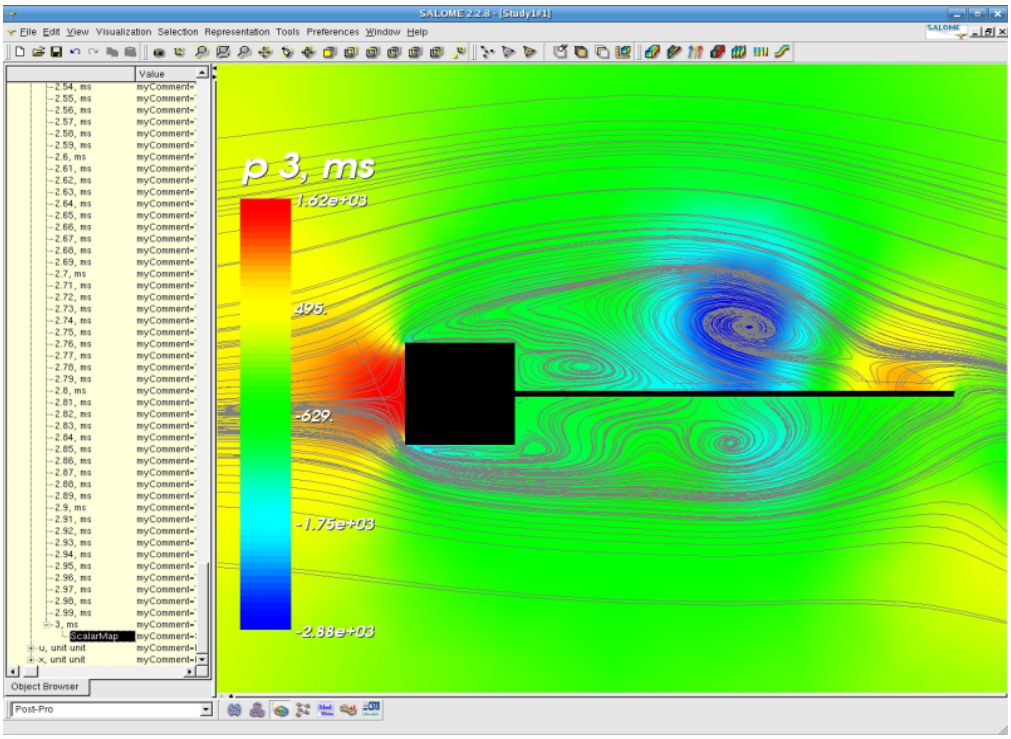


FIG. 5.14 – Calcul d'intéractions fluide-structure – Résultats – Champ de pression

5.5 Conclusion

Le métamodèle a été implémenté dans la plate-forme de simulation open-source SALOME avec le module DATA. Cette plate-forme est destinée à fournir un environnement d'intégration de composants destinés à la simulation numérique. Elle propose des modules spécialisés, CAO et maillage par exemple, pour faciliter le travail de description des problèmes. Le module DATA permet de décrire les propriétés physiques des problèmes étudiés avec la plate-forme. Il s'adresse à deux types d'utilisateurs :

- *l'utilisateur final :* c'est la personne qui réalise une étude et qui doit pouvoir décrire les propriétés physiques de son problème,
- *l'utilisateur intégrateur :* c'est la personne qui est chargée d'intégrer un ou des solveurs dans la plate-forme SALOME. Il lui faut pour cela décrire la structure de données du ou des solveur(s) dans un modèle SPML. Elle doit aussi programmer la transmission des données physiques vers le solveur concerné.

Nous mettons ainsi à la disposition d'une communauté un outil générique permettant de décrire les physiques de manière unifiée. Ce outil a été utilisé dans deux applications : un problème magnétostatique simple, et un problème couplé d'interactions fluide-structure. Ces exemples illustrent l'utilisation de la plate-forme SALOME comme environnement de pré et post-traitement des données. Ils illustrent aussi la généricité du module DATA, avec lequel les domaines de l'électromagnétisme, de la mécanique et de la mécanique des fluides ont pu être abordés.

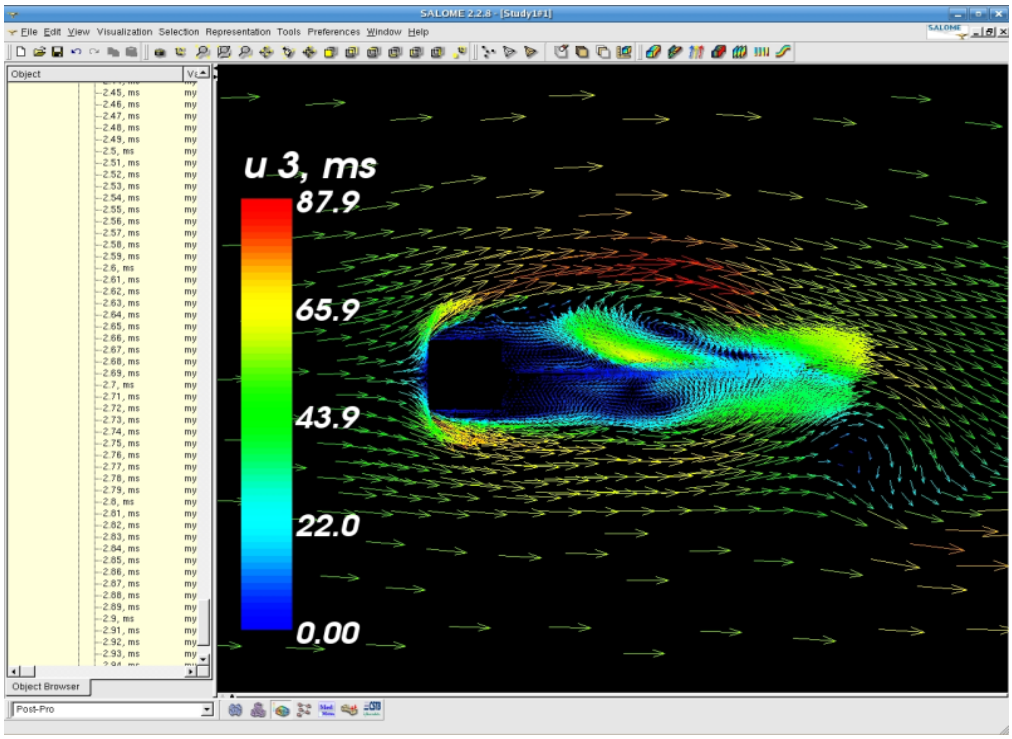


FIG. 5.15 – Calcul d'intéractions fluide-structure – Résultats – Vecteurs de vitesse

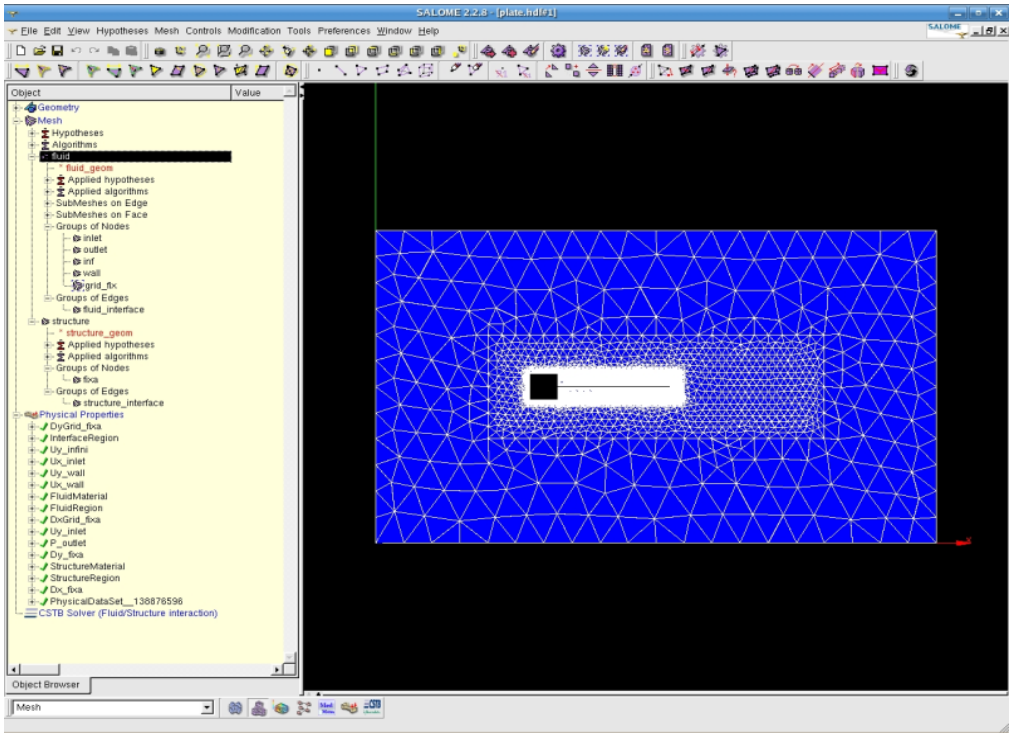


FIG. 5.16 – Calcul d'intéractions fluide-structure – Maillage de la région du fluide

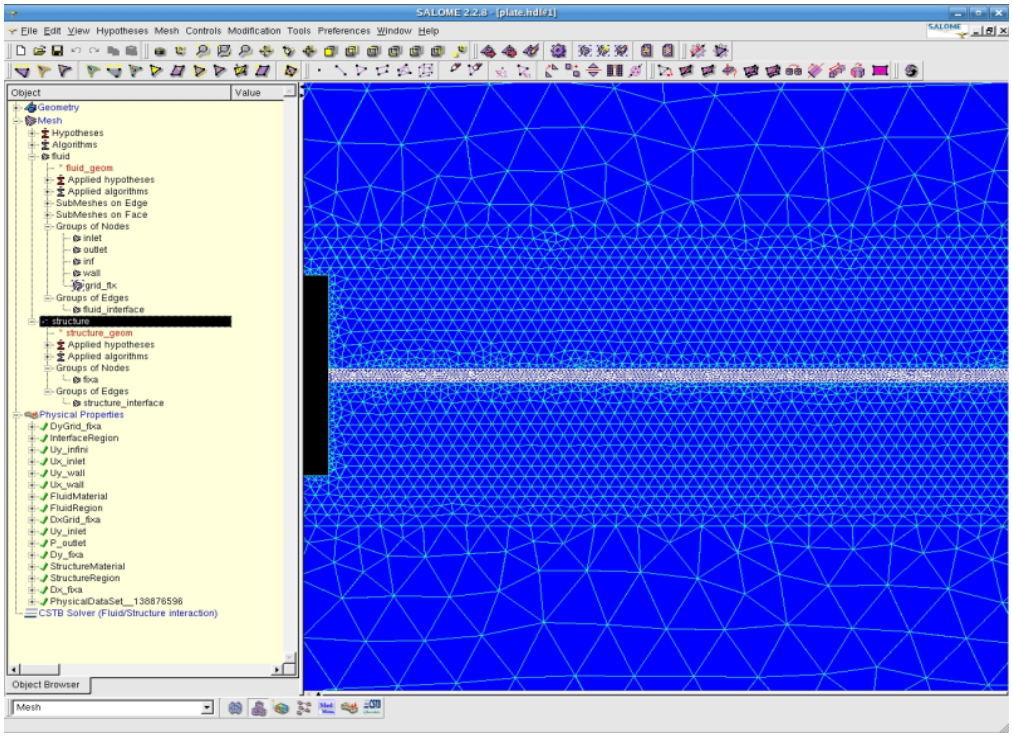


FIG. 5.17 – Calcul d'intéractions fluide-structure – Maillage de la région de la structure

Chapitre 6

Conclusion générale

Ce travail de recherche s'inscrit dans une problématique de modélisation de dispositifs multiphysiques du point de vue des développpeurs. Nous avons montré qu'une des difficultés principales rencontrées concerne le couplage des physiques. En effet, si de nombreux logiciels de simulation numérique existent, la plupart sont conçus pour résoudre un type de problème précis dans lequel les intéractions entre phénomènes physiques sont encore peu répandues.

Or, les progrès technologiques tels que l'intégration de composants et la miniaturisation ont entraîné une complexité grandissante des dispositifs physiques toujours plus performants. Les phénomènes qui jusqu'alors étaient considérés comme négligeables ont commencé à perturber voire modifier le comportement de ces dispositifs. Ces phénomènes sont ainsi devenus incontournables et leur influence doit désormais être prise en compte au moins tout autant que celle du phénomène physique principal dans le processus de modélisation que l'on peut désormais qualifier de multiphysique.

Deux grandes approches permettent de réaliser des couplages multiphysiques : l'approche globale et la modulaire. Si la première peut être considérée comme une boîte noire modélisant tout un dispositif dans son ensemble, la deuxième est basée sur la séparation des modèles de données physiques et l'utilisation d'un composant de supervision chargé de coordonner les résolutions et les transferts de données. Si ces deux approches sont incontournables, la 2^{e} offre des possibilités de souplesse, d'évolutivité et de flexibilité très intessantes, et elle présente une grande capacité de prototypage.

Nous proposons dans ce travail une méthode originale pour aider et simplifier le travail des développeurs confrontés à ce problème de couplage de différentes physiques. Cette méthode est basée sur l'utilisation d'un formalisme unique dont le but est de fournir un moyen de décrire de manière unifiée des modèles de données physiques. Placé à un niveau supérieur de modélisation, ce formalisme se comporte alors comme un modèle de modèles : c'est un métamodèle.

Des outils et services sont proposés autour de ce métamodèle afin de pouvoir gérer efficacement les modèles de données, mais aussi les données physiques issues de ces modèles et qui concernent des problèmes particuliers. La métamodèle a pour sa part été réalisé sous la forme d'un langage de programmation : le SPML (Standard Physic Modeling Language). Les modèles de données peuvent ainsi être décrits avec un même langage, quelle que soit le domaine de physique concerné. Ce langage est basé sur un langage de programmation orienté objet existant : Python. Il offre en plus des fonctionnalités variées comme le contrôle de cohérence des modèles et des données, une gestion évoluée pour l'accès à ces données, ou encore la possibilité de diffuser les données vers des environnements extérieurs (interfaces graphiques, solveurs).

Ces développements sont, entre autres, mis en œuvre dans une plate-forme open-source de simulation numérique : SALOME. Cet environnement permet de traiter de nombreux problèmes dans un logiciel complet qui rassemble les étapes de géométrie, physique, maillage, résolution et résultats. Deux exemples de problèmes multiphysiques, couplage magnéto-thermique et intéraction fluide-structure, sont donnés et permettent de démontrer l'utilité et la faisabilité de notre démarche.

Le travail que nous avons réalisé s'inscrit dans la continuité du travail de thèse de Stephan Giurgea [14] dans lequel il a participé à l'ébauche du métamodèle et d'un langage. Notre but était de mener à terme ces réflexions et de produire des outils concrets utilisables par la communauté de la simulation numérique. Dans ce cadre, la mise en œuvre était en soi une partie importante du travail. Dans cette optique, notre contribution s'est

portée sur plusieurs points importants :

- Nous avons finalisé le langage sous forme d'un langage cohérent : le SPML. Les concepts de base du langage ont été complétés et structurés de manière cohérente.
- Nous avons ajouté des services de gestion et de projection des données grâce à des notions d'accès contrôlé (Reader, Writer) et d'arbre abstrait.
- Nous avons mis en place un vrai interprète pour le SPML, grâce à la réutilisation d'un interprète existant. Cet interprète autorise désormais les commandes traditionnelles d'algorithmique (par exemple les boucles « `for` ») et est capable d'exécuter les concepts spécifiques au SPML.
- La syntaxe du SPML a été redéfinie pour la rendre plus simple d'utilisation et de compréhension. Elle est désormais entièrement conforme à la syntaxe Python, ce qui la rend plus propice à une diffusion aisée au sein de la communauté des développeurs d'outils de simulation numérique.

Perspectives

Malgré tout le travail accompli, beaucoup de choses restent à faire [1] : tout d'abord, la structure du métamodèle est posée, mais elle pourrait être complétée par certains `types` comme les énumérations, ou les interfaces. La définition de propriétés de matériaux à partir de valeurs expérimentales est un autre exemple de type de donnée que le métamodèle devra gérer. Toutes ces mises à jour du métamodèle entraineront bien sûr celles des outils associés.

Ensuite, un travail important, qui a été initié avec l'intégration de deux solveurs dans la plate-forme SALOME, reste celui de l'utilisation du SPML pour la description de modèles de données, qui entrainera alors des retours d'information de la part des utilisateurs du langage. Ces critiques seront nécessaires pour améliorer le métamodèle et ses outils, mais aussi pour proposer le développement d'autres outils ou services. Pour cela une collaboration étroite avec les physiciens et les spécialistes des formulations est nécessaire.

Un lien étroit avec le pilotage des solveurs doit être effectué puisqu'un premier travail sur la supervision a été présenté mais reste à continuer et à rationaliser.

Enfin, la maintenance du métamodèle au sein des outils où il est utilisé (SALOME, sous la forme du module DATA ; couplage magnéto-thermique) est un point critique pour son utilisation et sa diffusion. Il est très important de suivre les évolutions de version de SALOME pour proposer un module performant pour la description des propriétés physiques.

[1] pour autant que ce travail puisse être terminé un jour ...

Annexes

Annexe A

Attributs réservés des classes SPML

Nous décrivons ici les attributs de classe réservés à la description du comportement des classes ou des types intrinsèques SPML. Les noms de ces attributs ont tous les caractéristiques suivantes :

- ils commencent par `spml_`,
- et se terminent par `__`.

Pour chaque attribut nous donnons une définition, le type de la valeur qu'il reçoit, son éventuelle valeur par défaut, et son caractère optionnel. Le tableau A.1 concerne les classes SPML, c'est-à-dire celles qui héritent de `spml_class`. Le tableau A.2 concerne les classes intrinsèques SPML, c'est-à-dire celles qui héritent de `spml_int`, `spml_long`, `spml_float`, `spml_complex` et `spml_str`.

Attribut	**Définition**	**Type**	**Valeur par défaut**	**Opt**
`spml_modelowner__`	Modèle SPML	Instance de `spml_model`	—	Non
`spml_stereotype__`	Mode de relation	`str`	`"CONCRETE"`	Oui
`spml_supertype__`	Classe parente	`<type object>`	`None`	Oui
`spml_uicontrols__`	Informations pour interface utilisateur	Instance de `spml_ui_entity`	defaultLabel= *nomClasse* defaultComment = `"No comment available"`	Oui

Stéréotypes possibles : {`"ABSTRACT"`, `"CONCRETE"`}.

Tab. A.1 – Attributs réservés pour une classe SPML

Attribut	**Définition**	**Type**	**Valeur par défaut**	**Opt**
`spml_type__`	Type intrinsèque	`str`	`"STRING"`	Non
`spml_modelowner__`	Modèle SPML	Instance de `spml_model`	—	Non

Types intrinsèques possibles : {`"INTEGER"`, `"REAL"`, `"NUMERIC"`, `"STRING"`, `"NUMERIC_OR_STRING"`, `"BOOLEAN"`, `"VOID"`}.

Tab. A.2 – Attributs réservés pour un intrinsèque SPML

Annexe B

Attributs de contrôle des attributs SPML

Les attributs de contrôle permettent de définir les caractéristiques des attributs ou méthodes SPML. Ces attributs de contrôle sont des dictionnaires Python (couples clés/valeur). Nous donnons ici la description des éléments de ces dictionnaires de contrôle. Pour chaque élément (ou clé), nous donnons sa signification, le type de valeur, une éventuelle valeur par défaut et son caractère optionnel.

Clé	Signification	Type
`"relatedType"`	Type de l'attribut	`<type object>`
`"stereotype"`	Mode de relation	`str`
`"listMode"`	Mode de liste	`str`
`"typeOfCollection"`	Type de collection	Instance de `spml_utility_collection`
`"definitionMode"`	Mode de définition	`str`
`"uiInformation"`	Informations interface utilisateur	Instance de `spml_ui_attribute`

Clé	Valeur par défaut	Opt
`"relatedType"`	`None`	Non
`"stereotype"`	`"COMPOSITION"`	Oui
`"listMode"`	`"SF"`	Oui

`"typeOfCollection"`	`spml_listCollection(`
	`lowerBound=spml_integerBound(boundValue=0)`
	`upperBound=spml_populationDependentBound())`
		Oui

`"definitionMode"`	`"FORCED"`	Oui

`"uiInformation"`	`spml_ui_attribute(`
	`defaultLabel=`*`nomAttribut`*
	`defaultComment="No comment available"`
		Oui

– Stéréotypes possibles : {`"AGREGATION"`, `"COMPOSITION"`, `"ASSOCIATION"`, `"IDENTIFICATION"`}.
Attention : pour une classe donnée, il ne peut y avoir qu'un seul attribut avec le stéréotype `"IDENTIFICATION"`.
– Modes de liste possibles : {`"SF"` (= `SimpleField`), `"CF"` (= `CollectionField`)}.
– Modes de définition possibles : {`"FORCED"`, `"OPTIONAL"`, `"DERIVED"` (`"FINAL"` et `"INTERNAL"` existent mais ne sont pas implémentés)}.

Tab. B.1 – Contrôle des attributs SPML : liste des éléments

Annexe C

Les types SPML instanciables du métamodèle : attributs et méthodes

Liste détaillée des types SPML avec leurs attributs et méthodes. Le supertype d'un type est donné entre parenthèses ; les attributs et méthodes hérités ne sont alors pas redonnés.

C.1 Module *spml_app*

TAB. C.1: Attributs et méthodes des types du module *spml_app*

Attributs/Méthodes	Type/Objectif
`<type 'spml_app_object'>`	
id	`string`
getId(self)	renvoie self.id
`<type 'spml_app_category(spml_app_object)'>`	
defaultLabel	`string`
localHistory	liste d'instances de `spml_utility_ClassLocalCheckinFact`
additionalLabels	List de `str`
`<type 'spml_app_application(spml_app_object)'>`	
version	`string`
models	instance de `spml_applicationmodel`
localHistory	liste d'instances de `spml_utility_ClassLocalCheckinFact`
rules	liste d'instances de `spml_utility_Rule`
uiInformation	instance de `spml_ui_application`
getIDLFileName(self)	renvoie le nom du fichier IDL construit à partir de self.id et self.version
getApplicationModel(self)	renvoie self.models
getVersion(self)	renvoie self.version
getRules(self)	renvoie self.rules
`<type 'spml_app_applicationmodel(spml_app_object)'>`	

TAB. C.1: (suite)

Attributs/Méthodes	Type/Objectif
version	`string`
models	liste d'instances de `spml_model`
localHistory	liste d'instances de `spml_utility_ClassLocalCheckinFact`
rules	liste d'instances de `spml_utility_Rule`
uiInformation	instance de `spml_ui_application`
getModels(self)	renvoie self.models
getVersion(self)	renvoie self.version
`<type 'spml_app_model(spml_app_object)'>`	
supertype	instance de `spml_model`
localHistory	liste d'instances de `spml_utility_ClassLocalCheckinFact`
`<type 'spml_app_package(spml_app_object)'>`	
localHistory	liste d'instances de `spml_utility_ClassLocalCheckinFact`
`<type 'spml_app_mainpackage(spml_app_package)'>`	
`<type 'spml_app_subpackage(spml_app_package)'>`	
derivesFrom	instance de `spml_app_package`

C.2 Module *spml_utility*

TAB. C.2: Attributs et méthodes des types du module *spml_utility*

Attributs/Méthodes	Type/Objectif
`<type 'spml_utility_object'>`	
`<type 'spml_utility_objectwn(spml_utility_object)'>`	
id	`string`
getId(self)	renvoie self.id
`<type 'spml_utility_bound(spml_utility_object)'>`	
`<type 'spml_utility_populationDependentBound(spml_utility_bound)'>`	
`<type 'spml_utility_integerBound(spml_utility_bound)'>`	
boundValue	`integer`
getBoundValue(self)	renvoie self.boundValue
`<type 'spml_utility_collection(spml_utility_object)'>`	
`<type 'spml_utility_arrayCollection(spml_utility_collection)'>`	
lowerIndex	instance de `spml_utility_bound`
upperIndex	instance de `spml_utility_bound`
getLowerIndex(self)	renvoie self.lowerIndex
getUpperIndex(self)	renvoie self.upperIndex

TAB. C.2: (suite)

Attributs/Méthodes	**Type/Objectif**
`<type 'spml_utility_variableSizeCollection(spml_utility_collection)'>`	
lowerBound	instance de `spml_utility_bound`
upperBound	instance de `spml_utility_bound`
getLowerBound(self)	renvoie self.lowerBound
getUpperBound(self)	renvoie self.upperBound
`<type 'spml_utility_setCollection(spml_utility_variableSizeCollection)'>`	
`<type 'spml_utility_listCollection(spml_utility_variableSizeCollection)'>`	
`<type 'spml_utility_bagCollection(spml_utility_variableSizeCollection)'>`	
`<type 'spml_utility_attribute(spml_utility_objectwn)'>`	
relatedType	`type`
definitionMode	`string`
defaultValue	variable (`string`, `float`, ...)
range	`string`
uiInformation	instance de `spml_ui_attribute`
getType(self)	renvoie self.relatedType
getDefinitionMode(self)	renvoie self.definitionMode
getDefaultValue(self)	renvoie self.defaultValue
getRange(self)	renvoie self.range
getUiInformations(self)	renvoie self.uiInformation
`<type 'spml_utility_field(spml_utility_attribute)'>`	
stereotype	`string`
datatype	`string`
evaluationMode	`string`
inverseField	instance de `spml_utility_inverseField`
owner	instance de `spml_type_class`
getStereotype(self)	renvoie self.stereotype
checkTypeOfValue(self,value)	renvoie 0 si `value` est du bon type, 1 sinon
compilePythonLinesSet(self)	Persistance du field–renvoie une liste de `string`
`<type 'spml_utility_simpleField(spml_utility_field)'>`	
`<type 'spml_utility_collectionField(spml_utility_field)'>`	
typeOfCollection	instance de `spml_utility_collection`
`<type 'spml_utility_inverseField(spml_utility_objectwn)'>`	
directField	liste d'instances de `spml_utility_field`
uiInformation	instance de `spml_ui_attribute`
setDirectField(self,directField)	met à jour self.directField
`<type 'spml_utility_inverseSimpleField(spml_utility_inverseField)'>`	
`<type 'spml_utility_inverseCollectionField(spml_utility_inverseField)'>`	
typeOfCollection	instance de `spml_utility_collection`

TAB. C.2: (suite)

Attributs/Méthodes	Type/Objectif
`<type 'spml_utility_argument(spml_utility_attribute)'>`	
commandOwner	instance de `spml_utility_command`
`<type 'spml_utility_simpleArgument(spml_utility_argument)'>`	
`<type 'spml_utility_collectionArgument(spml_utility_argument)'>`	
typeOfCollection	instance de `spml_utility_collection`
`<type 'spml_utility_command(spml_utility_objectwn)'>`	
localHistory	liste d'instances de `spml_utility_ClassLocalCheckinFact`
modelOwner	instance de `spml_app_model`
arguments	liste d'instances de `spml_utility_argument`
rules	liste instance de `spml_utility_Rule`
packageOwner	instance de `spml_app_package`
uiInformation	instance de `spml_ui_command`
addArgument(self,argument)	met à jour self.arguments
addRule(self,rule)	met à jour self.rules
getModelOwner(self)	renvoie self.modelOwner
getArguments(self)	renvoie self.arguments
getRules(self)	renvoie self.rules
getPackageOwner(self)	renvoie self.packageOwner
getUiInformations(self)	renvoie self.uiInformation
`<type 'spml_utility_macro(spml_utility_command)'>`	
script	`string`
getScript(self)	renvoie self.script
`<type 'spml_utility_procedure(spml_utility_command)'>`	
`<type 'spml_utility_function(spml_utility_command)'>`	
returns	`type`
getReturns(self)	renvoie self.returns
`<type 'spml_utility_method(spml_utility_command)'>`	
returns	`type`
uiInformation	instance de `spml_ui_method`
compilePythonLinesSet(self)	persistance de la méthode ; renvoie une liste de `string`
`<type 'spml_utility_Rule(spml_utility_object)'>`	
ruleType	`string`
`<type 'spml_utility_GlobalCheckinFact(spml_utility_object)'>`	
name	`string`
revisionObject	`string`
date	`string`
producer	`string`
reviewer	`string`
getName(self)	renvoie self.name
getRevision(self)	renvoie self.revisionObject
getDate(self)	renvoie self.date

TAB. C.2: (suite)

Attributs/Méthodes	**Type/Objectif**
getProducer(self)	renvoie self.producer
getReviewer(self)	renvoie self.reviewer
`<type 'spml_utility_ClassLocalCheckinFact(spml_utility_object)'>`	
reasonOfModification	instance de `spml_utility_GlobalCheckinFact`
localComments	liste de `string`

C.3 Module *spml_ui*

TAB. C.3: Attributs et méthodes des types du module *spml_ui*

Attributs/Méthodes	**Type/Objectif**
`<type 'spml_ui_object'>`	
`<type 'spml_ui_objectwn(spml_ui_object)'>`	
id	`string`
getId(self)	renvoie self.id
`<type 'spml_ui_userProfile(spml_ui_objectwn)'>`	
defaultLabel	`string`
additionalLabels	liste de `string`
localHistory	liste d'instances de `spml_utility_ClassLocalCheckinFact`
`<type 'spml_ui_Context(spml_ui_objectwn)'>`	
defaultLabel	`string`
additionalLabels	liste de `string`
localHistory	liste d'instances de `spml_utility_ClassLocalCheckinFact`
underlyingModel	instance de spml_app_model
`<type 'spml_ui_Filter(spml_ui_object)'>`	
contextForCustomisation	instance de `spml_ui_Context`
profileOfUsers	liste de `spml_ui_userProfile`
`<type 'spml_ui_ActionFilter(spml_ui_Filter)'>`	
rightOfUsers	`string`
`<type 'spml_ui_DataFilter(spml_ui_Filter)'>`	
rightOfUsers	`string`
`<type 'spml_ui_information(spml_ui_object)'>`	
defaultLabel	`string`
additionalLabels	liste de `string`
defaultComment	`string`
additionalComments	liste de `string`
defaultTooltip	`string`
additionalTooltips	liste de `string`
getDefaultLabel(self)	renvoie self.defaultLabel
getDefaultComment(self)	renvoie self.defaultComment
getDefaultTooltip(self)	renvoie self.defaultTooltip
getAdditionalLabels(self)	renvoie self.additionalLabels

TAB. C.3: (suite)

Attributs/Méthodes	Type/Objectif
getAdditionalComments(self)	renvoie self.additionalComments
getAdditionalTooltips(self)	renvoie self.additionalTooltips
`<type 'spml_ui_application(spml_ui_information)'>`	
contexts	liste d'instances de `spml_ui_Context`
profiles	liste d'instances de `spml_ui_userProfile`
userManual	`string`
tutorial	`string`
`<type 'spml_ui_TuiGuiInformation(spml_ui_information)'>`	
commandName	`string`
commandShortName	`string`
category	instance de `spml_app_category`
helpPageUrl	`string`
iconUrl	`string`
iconGroupName	`string`
`<type 'spml_ui_command(spml_ui_TuiGuiInformation)'>`	
contextFiltering	liste d'instances de `spml_ui_ActionFilter`
`<type 'spml_ui_attribute(spml_ui_TuiGuiInformation)'>`	
contextFiltering	liste d'instances de `spml_ui_DataFilter`
inputMode	liste de `string`
reentrantMode	`string`
itemCount	`integer`
`<type 'spml_ui_entity(spml_ui_TuiGuiInformation)'>`	
contextFiltering	liste d'instances de `spml_ui_DataFilter`
inputMode	liste de `string`
reentrantMode	`string`
itemCount	`integer`
`<type 'spml_ui_method(spml_ui_TuiGuiInformation)'>`	
contextFiltering	liste d'instances de `spml_ui_ActionFilter`
reentrantMode	`string`

C.4 Module *spml_type*

TAB. C.4: Attributs et méthodes des types du module *spml_type*

Attributs/Méthodes	Type/Objectif
`<type 'spml_type_type(type,spml_object)'>`	
spml_id__	`string`
spml_modelowner__	instance de `spml_app_model`
spml_localhistory__	liste d'instances de `spml_utility_ClassLocalCheckinFact`
spml_users__	liste d'instances de `spml_ui_userProfile`
spml_packageowner__	liste d'instances de `spml_app_package`
getId(self)	renvoie self.spml_id__

TAB. C.4: (suite)

Attributs/Méthodes	**Type/Objectif**
getModelOwner(self)	renvoie self.spml_modelowner__
getPackageOwner(self)	renvoie self.spml_packageowner__
setPackageOwner(self,packageOwner)	met à jour self.spml_packageowner__
addUsers(self,users)	met à jour self.spml_users__
is_a(self,aString)	renvoie 1 si `str(type(self))= aString`, 0 sinon
`<type 'spml_type_intrinsic(spml_type_type)'>`	
spml_type__	`string`
getRelatedType(self)	renvoie self.spml_type__
is_a(self,aString)	renvoie 1 si `str(type(self))= aString`, 0 sinon
checkTypeOfValue(self,value)	renvoie 0 si `value` est du bon type, 1 sinon
getPythonEquivalentTypes(self)	renvoie l'équivalent Python de self.spml_type__
`<type 'spml_type_enumeration(spml_type_type)'>`–*Incomplet*	
is_a(self,aString)	renvoie 1 si `str(type(self))= aString`, 0 sinon
checkTypeOfValue(self,value)	renvoie 0 si `value` est du bon type, 1 sinon
`<type 'spml_type_aggregate(spml_type_type)'>`–*Incomplet*	
is_a(self,aString)	renvoie 1 si `str(type(self))= aString`, 0 sinon
getType(cls)	renvoie cls.spml_controls__[“type”]
getCollectionType(cls)	renvoie cls.spml_controls__[“collectionType”]
`<type 'spml_type_object(spml_type_type)'>`	
spml_methods__	liste d'instances de `spml_utility_method`
is_a(self,aString)	renvoie 1 si `str(type(self))= aString`, 0 sinon
addMethods(cls,methods)	met à jour cls.spml_methods__
getMethods(cls)	renvoie cls.spml_methods__
getGenericMethods(cls)	renvoie spml_methods__ de cls et de ses supertypes
`<type 'spml_type_interface(spml_type_object)'>`	
`<type 'spml_type_class(spml_type_object)'>`	
spml_stereotype__	`string`
spml_supertype__	`string`
spml_fields__	liste d'instances de `spml_utility_field`
spml_interfaces__	liste d'instances de `spml_type_interface`
spml_storage__	`string`
spml_uicontrols__	instance de `spml_ui_entity`
getStereotype(cls)	renvoie cls.spml_stereotype__
spml_getStereotype__(cls)	renvoie cls.spml_stereotype__
getUiInformations(cls)	renvoie cls.spml_uicontrols__
addFields(cls,fields)	met à jour cls.spml_fields__
getFields(cls)	renvoie cls.spml_fields__
getGenericFields(cls)	renvoie spml_fields__ de cls et de ses supertypes
is_a(cls,aStr)	renvoie 1 si aStr correspond au type ou stereotype de cls, 0 sinon
spml_is__(cls,aStr)	renvoie 1 si aStr correspond au type ou stereotype de cls, 0 sinon

TAB. C.4: (suite)

Attributs/Méthodes	Type/Objectif
isklassOrSubklassFromName (cls,entityType)	renvoie 0 si `entityType` correspond à l'id de cls ou de ses supertypes, 1 sinon
isklassOrSubklass(cls,classInstance)	renvoie 0 si `classInstance` correspond à cls ou un de ses supertypes, 1 sinon
checkField(cls,fieldName,fieldValue)	renvoie 0 si le field `fieldName` existe et `fieldValue` est du bon type, 1 si le field `fieldName` n'existe pas, 2 si le field `fieldName` existe et `fieldValue` n'est pas du bon type
getFieldByName(cls,fieldName)	renvoie l'instance de `spml_utility_field` avec `id=fieldname`
getFieldDefaultValue(cls,fieldName)	renvoie la valeur par défaut du field `fieldname`
checkTypeOfValue(cls,value)	renvoie 0 si `value` est du bon type, 1 sinon
_getFieldsName(cls)	renvoie une liste de string
compileIdl(cls)	renvoie une string correspondant au corps IDL de cls
compileHeadIdl(cls)	renvoie une string correspondant à l'en-tête IDL de cls

Annexe D

Mécanisme détaillé de la projection des objets SPML dans un arbre abstrait

D.1 La projection des instances de `spml_class` : `self.project(object,feather)`

Cette méthode est chargée de projeter l'instance de `spml_class` `object` dans l'arbre abstrait à partir de la feuille `feather`.

Le principe est de créer des mappeurs de méthodes et de les affecter à la feuille de l'objet en projection, de créer des feuilles SPML pour chaque attribut SPML de l'objet, et enfin de créer des mappeurs de méthodes et de les affecter à ces dernières feuilles.

Les mappeurs de méthode de la feuille associée à l'instance `object` doivent permettre de récupérer ses informations UI ou de supprimer l'instance. Les éventuelles méthodes personnalisées que l'utilisateur peut associer à une classe SPML doivent aussi être mappées.

Nous présentons ici la création de ces mappeurs de méthode et leur affectation à la feuille SPML représentant l'objet `object` en cours de projection.

Création et affectation à la feuille `feather` du mappeur `"GetClassUiInfo"` permettant de récupérer les infos UI :

```
# Create and assign method GetUiInfo
feather.addMethod(DATA_FunctionWrapper.BoundMethodWrapper(\
    "GetClassUiInfo",
    getattr(self.metaReader,"getClassUiInformations_P"),
    {"klass":object.getKlass()}))
```

Création et affectation à la feuille `feather` du mappeur `"Delete"` permettant de supprimer l'instance :

```
if "Delete" not in feather.getMethods():
  # Create and assign method Delete
  feather.addMethod(DATA_FunctionWrapper.BoundMethodWrapper(\
    "Delete",
    getattr(self.writer,"deleteEntity_P"),
    {"object":object}))
```

Création et affectation à la feuille `feather` des mappeurs pour les méthodes personnalisées :

```
#Adding custom methods
for method in object.getKlass().getGenericMethods():
  if method.getArguments() is None:
    print "Adding custom method :",method.getId()
    if method.getId() not in feather.getMethods():
      label = method.getUiInformations().getDefaultLabel()
      if label is None:
        label = method.getId()
      aMethod = DATA_FunctionWrapper.BoundMethodWrapper(\
                  label,
                  getattr(object,"spml_"+method.getId()),
                  {})
      feather.addMethod(aMethod)
  else:
    ##Methods with args not yet implemented
    pass
```

Une fois ces mappeurs créés, nous analysons et projetons à leur tour les attributs de l'objet.

D.2 La projection des attributs de chaque instance

Pour chacun des attributs, une feuille SPML `fChild`, fille de la feuille `feather`, est retrouvée ou créée. Ensuite elle est mise à jour en fonction du type et de la valeur de l'attribut.

Nous parcourons la liste des attributs de l'instance et pour chacun d'entre eux nous récupérons leur valeur :

```
for field in object.getKlass().getGenericFields():
  #get the value of the field
  try:
    fieldValue=getattr(object,field.getId())
  except AttributeError:
    #mean field is not given (if is optional it is normal)
    #test should be done
    pass
  except:
    raise
```

L'instance `object` peut posséder un attribut particulier permettant de lui donner un nom. Cet attribut est caractérisé par son stéréotype dont la valeur est `"IDENTIFICATION"`. Dans ce cas, la feuille `feather` se verra ajouter deux mappeurs permettant de lire (`"GetName"`) ou de modifier (`"SetName"`) cet attribut, c'est-à-dire le nom de l'instance :

```
  if (field.getStereotype() in ('identification',
                                'IDENTIFICATION')):
    # if object has an identification field,
    # add GetName and SetValue methods
    # Create and assign method GetName
```

```
    feather.addMethod(\
      DATA_FunctionWrapper.BoundMethodWrapper("GetName",
        getattr(object,"__getattr__"),
        {"name":field.getId()}))
    # Create and assign method SetName
    feather.addMethod(\
      DATA_FunctionWrapper.BoundMethodWrapper("SetName",
        getattr(self.writer,"setField_P"),
        {"object":object,"field":field}))
  else:
    #
    # Create methods related to the fields
    #
```

Pour tous les autres attributs dont le stéréotype n'est pas `"IDENTIFICATION"`, des mappeurs sont créés permettant de lire et modifier leur valeur, de les supprimer, de récupérer leurs infos UI, ou de récupérer leur type.

Commençons par le mappeur `"GetValue"` :

```
    # Create method GetValue
    getValue=DATA_FunctionWrapper.BoundMethodWrapper(\
      "GetValue",
      getattr(object,"__getattr__"),
      {"name":field.getId()})
```

Nous avons ensuite les mappeurs `"SetValue"`. Tous les types d'attributs doivent pouvoir être modifiés et pour cela, il y a autant de mappeurs pour modifier les valeurs que de types d'attributs. Les mappeurs sont spécialisés pour un type de valeur grâce à l'utilisation de convertisseurs. En effet, lorsque l'utilisateur donnera la nouvelle valeur d'un attribut, il le fera depuis une interface (graphique ou non). Quel que soit le type de l'attribut, ce sera une `string` que l'utilisateur enverra aux mappeurs. Ces derniers utiliseront alors leurs convertisseurs pour convertir cette `string` vers leur type de valeur (int, float ou instance de `spml_class`).

Mappeur `"SetValue"` pour une `string` (pas besoin de convertisseur) :

```
    # Create methods SetValue:
    # SetValue for string
    setValue=DATA_FunctionWrapper.BoundMethodWrapper(\
      "SetValue",
      getattr(self.writer,"setField_P"),
      {"object":object,"field":field})
```

Mappeur `"SetValue"` pour un entier :

```
    # SetValue for integer
    setValueInteger=DATA_FunctionWrapper.BoundMethodWrapper(\
      "SetValue",
      getattr(self.writer,"setField_P"),
      {"object":object,"field":field})
    setValueInteger.setArgsConverter(2,int)
```

Mappeur `"SetValue"` pour un réel :

```
    # SetValue for float
    setValueFloat=DATA_FunctionWrapper.BoundMethodWrapper(\
      "SetValue",
      getattr(self.writer,"setField_P"),
      {"object":object,"field":field})
    setValueFloat.setArgsConverter(2,float)
```

Mappeur `"SetValue"` pour un objet SPML :

```
    # SetValue for Entity
    setValueEntity = DATA_FunctionWrapper.BoundMethodWrapper(\
      "SetValue",
      getattr(self.writer,"setField_Entity_P"),
      {"object":object,"field":field})
    # This converter creates a new entity and returns its uid
    setValueEntity.setArgsConverter(2,self.writer.createEntity)
```

Nous avons ensuite le mappeur `"Delete"` permettant de supprimer la valeur de l'attribut :

```
    # Create method Delete
    deleteField=DATA_FunctionWrapper.BoundMethodWrapper(\
      "Delete",
      getattr(self.writer,"deleteField_P"),
      {"object":object,"field":field})
```

Ensuite le mappeur `"GetFieldUiInfo"` permettant de récupérer les infos UI de l'attribut :

```
    # Create method GetUiInfo
    getFieldUiInfo=DATA_FunctionWrapper.BoundMethodWrapper(\
      "GetFieldUiInfo",
      getattr(self.metaReader,"getAttributeUiInformations_P"),
      {"field":field})
```

Enfin le mappeur `"GetType"` permettant de récupérer le type de l'attribut :

```
    # Create method GetType
    getType=DATA_FunctionWrapper.BoundMethodWrapper(\
      "GetType",
      getattr(self.metaReader,"getType_P"),
      {"klass":object.getKlass(),"field":field})
```

Une fois les mappeurs créés, le type, le stéréotype et le mode de liste de l'attribut conditionnent le traitement de la feuille SPML `fChild` qui lui est associée. Au moment de l'écriture de ces lignes, seuls les types intrinsèques (type `"Intrinsic"`) et les instances de `spml_class` (type `"Entity"`) sont pris en compte.

Nous cherchons d'abord à récupérer la feuille `fChild` associée à l'attribut grâce au dictionnaire `self.dictOfProjectedObject`. Pour cela on la cherche à la clé `key = str( object.getUid())+field.getId()`. Si on la trouve, elle doit être mise à jour, sinon elle doit être créée :

```
exist = 1
try:
  #key is object uid + field name
  fChild=self.dictOfProjectedObject[\
    str(object.getUid())+field.getId()]
  if fChild.getParent() is not feather:
    raise KeyError
except KeyError:
  exist = 0
if exist:
  #--------------------------
  #update child feather
  #--------------------------
else:
  #--------------------------
  #create child feather
  #--------------------------
```

Que ce soit pour la mise à jour ou pour la création de la feuille `fChild`, nous distinguons les modes de liste de l'attribut : SimpleField ou CollectionField. Dans le premier cas nous avons affaire à un attribut simple, dans le 2^{e} c'est une liste et nous devons alors itérer sur chacun de ses éléments. Nous obtenons donc les cas de figures suivants :

1. création d'une feuille
   (a) pour un attribut Simplefield,
   (b) pour un attribut CollectionField,
2. mise à jour d'une feuille,
   (a) pour un attribut Simplefield,
   (b) pour un attribut CollectionField.

Chacun de ces cas de figure est détaillé :

1. *Création d'une feuille*
   La feuille `fChild` est créée comme enfant de la feuille `feather` et le dictionnaire `self.dictOfProjectedObject` est mis à jour. La première partie de l'id de `fChild` est aussi définie :

```
else:
  #--------------------------
  #create child feather
  #--------------------------
  fChild = SPMLFeather(feather)
  self.dictOfProjectedObject[\
    str(object.getUid())+field.getId()] = fChild
  #Set first part of feather id
  fChild.setId0(field.getId())
```

   Ensuite nous distinguons les modes de liste de l'attribut pour lequel la feuille est créée (Simplefield ou Collectionfield).

   (a) *Pour un attribut Simplefield :*

```
    if isinstance(field,\
      DATA_SPMLtype.spml_utility_simpleField__):
```

Nous commençons par mettre à jour la valeur et l'id de la feuille `fChild` et nous lui ajoutons le mappeur permettant de récupérer la valeur de l'attribut :

```
      fChild.setValue(fieldValue)
      fChild.setId1(fieldValue)
      # Assign methods to fChild
      fChild.addMethod(getValue)
```

Ensuite plusieurs situations se présentent :

– La valeur n'est pas `None` et le mode de définition de l'attribut n'est pas `"DERIVED"` ; dans ce cas nous pouvons ajouter le mappeur permettant de supprimer la valeur de l'attribut :

```
      if fieldValue is not None and not fieldIsDerived:
        fChild.addMethod(deleteField)
```

– L'attribut est de type `"Intrinsic"` et le mode de définition de l'attribut n'est pas `"DERIVED"` ; dans ce cas nous ajoutons les mappeurs permettant d'affecter une valeur en fonction du type exact de l'attribut :

```
      if fType.is_a("Intrinsic") and not fieldIsDerived:
        if fType.is_a("INTEGER") or fType.is_a("BOOLEAN"):
          fChild.addMethod(setValueInteger)
        elif fType.is_a("DOUBLE") or fType.is_a("NUMERIC"):
          fChild.addMethod(setValueFloat)
        else:
          fChild.addMethod(setValue)
```

– L'attribut est de type `"Entity"` et son stéréotype est soit `"ASSOCIATION"`, soit `"AGREGATION"`. Si la valeur n'est pas `None`, nous retrouvons la feuille de l'instance référencée et nous mettons à jour la feuille `fChild` avec cette feuille ; sinon la référence de `fChild` est mise à `None` :

```
      #
      #If field type is a reference entity
      #(ASSOCIATION or AGREGATION)
      #
      elif fTypeIsEntityReference:
        if fieldValue is not None:
          #Set reference
          ref = self.dictOfProjectedObject[\
            str(fieldValue.getUid())]
          fChild.setReference(ref)
          #Set last part of feather id
          #fChild.setId1(str(ref))
        else:
          fChild.setReference(None)
```

Dans tous les cas nous ajoutons à `fChild` un mappeur permettant de récupérer la liste des instances potentielles ; et si le mode de définition de l'attribut n'est pas `"DERIVED"`, nous ajoutons à `fChild` un mappeur pour affecter une instance à l'attribut :

```
        # Assign reference methods to fChild
        fChild.addMethod(\
          DATA_FunctionWrapper.BoundMethodWrapper(\
            "GetPotentialValues",
            getattr(self.reader,"getPotentialValues_Entity_P"),
            {"field":field}))
        if not fieldIsDerived:
          fChild.addMethod(\
            DATA_FunctionWrapper.BoundMethodWrapper(\
              "SetValue",
              getattr(self.writer,"setField_Entity_P"),
              {"object":object,"field":field}))
```

– L'attribut est de type `"Entity"` et son stéréotype est `"COMPOSITION"`. Si la valeur n'est pas `None`, nous reprojetons la valeur, qui est alors une instance de `spml_class`, avec la méthode `self.project` et depuis la feuille `fChild` :

```
      #
      #If field type is a composition entity
      #
      elif fTypeIsEntityComposition:
        if (fieldValue is not None):
          #Project object and its fields
          self.project(fieldValue,fChild)
          #Set last part of feather id
          fChild.setId1(str(fieldValue))
```

Si la valeur est `None`, alors nous ajoutons un mappeur permettant de créer une nouvelle instance du type de l'attribut, à condition que son mode de définition ne soit pas `"DERIVED"`. On ajoute aussi un mappeur permettant de récupérer la liste des types possible à instancier (types et sous-types de l'attribut) :

```
        else:
          if not fieldIsDerived:
            setValueEntity = \
              DATA_FunctionWrapper.BoundMethodWrapper(\
                "SetValue",
                getattr(self.writer,"setField_Entity_P"),
                {"object":object,"field":field})
            # This converter creates a new entity and
            # returns its uid
            setValueEntity.setArgsConverter(2,\
              self.writer.createEntity)
            fChild.addMethod(setValueEntity)
        fChild.addMethod(\
          DATA_FunctionWrapper.BoundMethodWrapper(\
            "GetPotentialValues",
            getattr(self.metaReader,"getPotentialTypes_P"),
            {"klass":fType}))
```

(b) *Pour un attribut CollectionField :*

```
    elif isinstance(field,\
      DATA_SPMLtype.spml_utility_collectionField__):
```

Dans le cas d'une collection, la valeur de l'attribut est une liste d'éléments homogènes. Nous choisissons de créer une feuille SPML pour chaque élément de la liste. Nous définissons une variable `index` qui indiquera au mappeur `"addElement"` s'il faut créer une nouvelle liste (`index = None`), ou ajouter un élément à la liste existante (`index = -1`).
Si la valeur est `None`, `index` est mis à `None`.

```
if fieldValue is None:
  index = None #mean addElement = create new list
```

Sinon, la liste existe : `index` est mis à `-1` et nous parcourons cette liste. Nous définissons un compteur `fieldCount` qui va nous permettre de connaître la position de chaque élément dans la liste. Pour chacun, une feuille SPML `fSubChild` est créée avec une id et une valeur correspondant à l'élément en cours et le dictionnaire `self.dictOfProjectedObject` est mis à jour. La feuille `fChild` sera le feuille parente de toutes ces feuilles `fSubChild` :

```
else:
  # If the field is not empty, it is a list
  index = -1 #mean addElement=append existing list
  # Counting elements
  fieldCount = 0
  # Parsing the elements in the field
  for fieldElement in fieldValue:
    # Create feathers
    fSubChild = SPMLFeather(fChild)
    if fTypeIsEntityComposition:
      self.dictOfProjectedObject[\
        str(object.getUid())+\
        field.getId()+\
        str(fieldElement.getUid())] = fSubChild
      fSubChild.setSPMLValidity(fieldElement.isValid())
    else:
      self.dictOfProjectedObject[\
        str(object.getUid())+\
        field.getId()+\
        str(fieldCount)] = fSubChild
      fSubChild.setSPMLValidity(\
        object.isFieldValid(field.getId()))

    fSubChild.setId1(fieldElement)
    fSubChild.setValue(fieldElement)
    fSubChild.indicator = 1
```

On ajoute à chaque feuille `fSubChild` les mappeurs `"GetFieldUiInfo"` et `"GetType"` et un mappeur permettant de récupérer l'élément en cours ; si l'attribut n'est pas un champ dérivé, un mappeur`"Delete"` permettant de supprimer l'élément en cours est aussi ajouté :

```
    # Assign methods to fSubChild
    fSubChild.addMethod(getFieldUiInfo)
    fSubChild.addMethod(getType)
    if not fieldIsDerived:
```

```
            fSubChild.addMethod(\
              DATA_FunctionWrapper.BoundMethodWrapper(\
                "Delete",
                getattr(self.writer,"deleteCField_P"),
                {"object":object,
                  "field":field,
                  "index":fieldCount}))
          fSubChild.addMethod(\
            DATA_FunctionWrapper.BoundMethodWrapper(\
              "GetValue",
              getattr(self.reader,"getCField_P"),
              {"object":object,
                "field":field,
                "index":fieldCount}))
```

Nous devons ensuite ajouter le mappeur `"SetValue"` permettant de modifier l'élément en cours. De la même manière que pour un attribut SimpleField, ce mappeur sera différent selon le type de l'attribut et ne sera ajouté que si l'attribut n'est pas un champ dérivé.

Voici le mappeur `"SetValue"` pour un attribut de type `"Intrinsic"` (auquel est ajouté un convertisseur pour les entiers et les réels) :

```
          if fType.is_a("Intrinsic") and not fieldIsDerived:
            setCValueIntrinsic=\
              DATA_FunctionWrapper.BoundMethodWrapper(\
                "SetValue",
                getattr(self.writer,"setCField_P"),
                {"object":object,
                  "field":field,
                  "index":fieldCount})

            if fType.is_a("INTEGER") or fType.is_a("BOOLEAN"):
              # SetValue for integer
              setCValueIntrinsic.setArgsConverter(2,int)
            elif fType.is_a("DOUBLE") or fType.is_a("NUMERIC"):
              # SetValue for double
              setCValueIntrinsic.setArgsConverter(2,float)
            fSubChild.addMethod(setCValueIntrinsic)
```

Voici le mappeur `"SetValue"` pour un attribut de type `"Entity"` et défini en `"ASSOCIATION"` ou `"AGREGATION"`. Pour ce type d'attribut, la feuille `fSubChild` est mise à jour avec l'instance référencée et un mappeur `"GetPotentialValues"` est ajouté :

```
          #
          #If field type is a reference entity
          #
          elif fTypeIsEntityReference:
            #Set reference
            ref = self.dictOfProjectedObject[\
              str(fieldElement.getUid())]
            fSubChild.setReference(ref)
            # Assign reference methods to fSubChild
            # SetValue for object reference
```

```
            # (not composition)
            fSubChild.addMethod(\
              DATA_FunctionWrapper.BoundMethodWrapper(\
                "GetPotentialValues",
                getattr(self.reader,\
                  "getPotentialValues_Entity_P"),
                {"field":field}))
            if not fieldIsDerived:
              fSubChild.addMethod(\
                DATA_FunctionWrapper.BoundMethodWrapper(\
                  "SetValue",
                  getattr(self.writer,\
                    "setCField_Entity_P"),
                  {"object":object,
                    "field":field,
                    "index":fieldCount}))
```

Pour un attribut de type `"Entity"` et défini en `"COMPOSITION"`, l'élément en cours `"fieldElement"` est une instance de `spml_class` que l'on reprojette avec la méthode `"self.project"`, et avec `"fSubChild"` comme feuille parente :

```
          #
          #If field type is a composition entity
          #
          elif fTypeIsEntityComposition:
            #Project object and its fields
            self.project(fieldElement,fSubChild)
            #Set last part of feather id
            fSubChild.setId1(str(fieldElement))
```

Une fois toute la liste parcourue, le compteur `"fieldCount"` est incrémenté. Si l'attribut n'est pas un champ dérivé, nous ajoutons un mappeur `"Delete"` sur la feuille `fChild` qui permettra de supprimer la liste :

```
          fieldCount = fieldCount +1
        if not fieldIsDerived:
          # fieldValue is not None:
          # Create and assign method Delete to fChild
          fChild.addMethod(\
            DATA_FunctionWrapper.BoundMethodWrapper(\
              "Delete",
              getattr(self.writer,"deleteCField_P"),
              {"object":object,
                "field":field,
                "index":index}))
```

Ensuite, que la liste existe ou non, nous ajoutons le mappeur `"GetValue"` sur la feuille `fChild` :

```
        #
        # Create and assign method GetValue to fChild
        #
        fChild.addMethod(\
          DATA_FunctionWrapper.BoundMethodWrapper(\
```

```
        "GetValue",
        getattr(self.reader,"getCField_P"),
        {"object":object,"field":field,"index":None}))
```

Un mappeur `"AddElement"` doit aussi être ajouté à la feuille `fChild`. Encore une fois ce mappeur va dépendre du type de l'attribut et de son mode de définition (non dérivé). La variable `index` sera donnée comme argument de la construction de ce mappeur car sa valeur indiquera s'il faut créer une nouvelle liste (`index=None`) ou ajouter un élément à une liste existante (`index=-1`).
Voici le mappeur `"AddElement"` pour un attribut de type `"Intrinsic"` (auquel est ajouté un convertisseur pour les entiers et les réels) :

```
#
# Create and assign method AddElement to fChild
#
if fType.is_a("Intrinsic") and not fieldIsDerived:
  addCValueIntrinsic=\
    DATA_FunctionWrapper.BoundMethodWrapper(\
      "AddElement",
      getattr(self.writer,"setCField_P"),
      {"object":object,"field":field,"index":index})
  if fType.is_a("INTEGER") or fType.is_a("BOOLEAN"):
    # SetValue for integer
    addCValueIntrinsic.setArgsConverter(2,int)
  elif fType.is_a("DOUBLE") or fType.is_a("NUMERIC"):
    # SetValue for integer
    addCValueIntrinsic.setArgsConverter(2,float)
  fChild.addMethod(addCValueIntrinsic)
```

Voici la construction du mappeur `"AddElement"` pour un attribut de type `"Entity"` :

```
elif fType.is_a("Entity"):
  # SetValue for Entity
  setCValueEntity=\
    DATA_FunctionWrapper.BoundMethodWrapper(\
      "AddElement",
      getattr(self.writer,"setCField_Entity_P"),
      {"object":object,"field":field,"index":index})
```

Si le stéréotype de l'attribut est `"ASSOCIATION"` ou `"AGREGATION"`, le mappeur `"AddElement"` n'est pas modifié, mais un autre mappeur `"GetPotentialValues"` est ajouté à la feuille `fChild` :

```
  if fTypeIsEntityReference:
    fChild.addMethod(\
      DATA_FunctionWrapper.BoundMethodWrapper(\
        "GetPotentialValues",
        getattr(self.reader,\
          "getPotentialValues_Entity_P"),
        {"field":field}))
```

Par contre si le stéréotype de l'attribut est `"COMPOSITION"`, un convertisseur permettant de créer une nouvelle instance est ajouté au mappeur `"AddElement"`, et un autre mappeur `"GetPotentialValues"` est ajouté à la feuille `fChild` :

```
        elif fTypeIsEntityComposition:
          # This converter creates a new entity
          # and returns its uid
          setCValueEntity.setArgsConverter(2,\
            self.writer.createEntity)
          fChild.addMethod(\
            DATA_FunctionWrapper.BoundMethodWrapper(\
              "GetPotentialValues",
              getattr(self.metaReader,
                "getPotentialTypes_P"),
              {"klass":fType}))
```

Enfin, quel que soit le stéréotype de l'attribut, le mappeur `"AddElement"` est ajouté à la feuille `fChild` :

```
        if not fieldIsDerived:
          fChild.addMethod(setCValueEntity)
```

Pour les deux cas de figure 1a et 1b, la création de la feuille `fChild` se termine en lui ajoutant les mappeurs `"getFieldUiInfo"` et `"getType"` :

```
    # Assign methods to fChild
    fChild.addMethod(getFieldUiInfo)
    fChild.addMethod(getType)
```

2. *Mise à jour de la feuille `fChild` :*
   Nous sommes dans la situation où la feuille `fChild` existe, son id (première partie) est d'abord mis à jour :

```
  if exist:
    #--------------------------
    #update child feather
    #--------------------------
    #Set first part of feather id
    fChild.setId0(field.getId())
```

Ensuite nous distinguons les modes de liste de l'attribut pour lequel la feuille est mise à jour (Simplefield ou Collectionfield).

(a) *Pour un attribut Simplefield :*

```
    if isinstance(field,\
      DATA_SPMLtype.spml_utility_simpleField__):
```

Nous commençons par mettre à jour la valeur et l'id (deuxième partie) de la feuille `fChild` :

```
      #Set the new value
      fChild.setValue(fieldValue)
      #Set last part of feather id
      fChild.setId1(fieldValue)
```

Ensuite, les mappeurs de la feuille `fChild` doivent être mis à jour afin de refléter la nouvelle situation.
Si la valeur est `None`, nous retirons les mappeurs suivants : `"GetName"`, `"SetName"`, `"Delete"` et `"GetClassUiInfo"`.

```
if fieldValue is None:
  fChild.removeMethod("GetName")
  fChild.removeMethod("SetName")
  fChild.removeMethod("Delete")
  fChild.removeMethod("GetClassUiInfo")
```

Toujours avec la valeur à `None`, si le type de l'attribut est `"Entity"` et le stéréotype `"ASSOCIATION"` ou `"AGREGATION"`, nous retirons la référence associée à la feuille `fChild`.

```
  #
  #If field type is a reference entity
  #
  if fTypeIsEntityReference:
      fChild.setReference(None)
```

Toujours avec la valeur à `None`, si le type de l'attribut est `"Entity"`, le stéréotype `"COMPOSITION"` et que le mode de définition n'est pas `"DERIVED"`, alors nous ajoutons le mappeur `"SetValue"` à la feuille `fChild`.

```
  #
  #If field type is a composition entity
  #
  elif fTypeIsEntityComposition and not fieldIsDerived:
    fChild.addMethod(setValueEntity,"SetValue")
```

Si la valeur n'est pas `None` et que le mode de définition de l'attribut n'est pas `"DERIVED"` alors nous ajoutons le mappeur `"Delete"` à la feuille `fChild`.

```
else: # fieldValue is not None
  if not fieldIsDerived:
    fChild.addMethod(deleteField)
```

Si le type de l'attribut est `"Entity"` et le stéréotype `"ASSOCIATION"` ou `"AGREGATION"`, nous cherchons la feuille référencée dans le dictionnaire `self.dictOfProjectedObject` et nous l'affectons à la feuille `fChild`.

```
  #
  #If field type is a reference entity
  #
  if fTypeIsEntityReference:
    #Set reference
    ref = self.dictOfProjectedObject[\
      str(fieldValue.getUid())]
    fChild.setReference(ref)
```

Si le type de l'attribut est toujours `"Entity"`, mais le stéréotype `"COMPOSITION"`, alors nous reprojetons l'instance à partir de la feuille `fChild` avec la méthode `self.project`. L'id (deuxième partie) de `fChild` est mis à jour et le mappeur `"SetValue"` lui est retiré :

```
          #
          #If field type is a composition entity
          #
          elif fTypeIsEntityComposition:
            #Project object and its fields
            self.project(fieldValue,fChild)
            #Set last part of feather id
            fChild.setId1(str(fieldValue))
            #Remove method SetValue if object exist
            #because the field can only be deleted.
            fChild.removeMethod("SetValue")
```

(b) *Mise à jour d'un attribut CollectionField :*

```
      elif isinstance(field,\
        DATA_SPMLtype.spml_utility_collectionField__):
```

Dans le cas d'une collection, la valeur de l'attribut est une liste d'éléments homogènes. Il y a une feuille SPML pour chaque élément de la liste. Nous définissons une variable `index` qui indiquera au mappeur `"addElement"` s'il faut créer une nouvelle liste (`index = None`), ou ajouter un élément à la liste existante (`index = -1`).

Si la valeur est `None`, `index` est mis à `None` et on retire le mappeur `"Delete"` de la feuille `fChild` :

```
        if fieldValue is None:
          index = None #mean addElement = create new list
          fChild.removeMethod("Delete")
```

Sinon, la liste existe : `index` est mis à `-1` et nous définissons un compteur `fieldCount` qui vas nous permettre de connaître la position de chaque élément dans la liste :

```
        else:
          # If the field is not empty, it is a list
          index = -1 #mean addElement = append existing list
          # Elements counter
          fieldCount = 0
```

Ensuite nous parcourons la liste et pour chaque élément de la liste, nous recherchons sa feuille `fSubChild` dans le dictionnaire `self.dictOfProjectedObject`. La clé à laquelle nous pouvons trouver cette feuille dépend du type de l'attribut :

```
          # Parsing the elements in the field
          for fieldElement in fieldValue:
            exist = 1
            try:
              # update sub child feather
              # dict key is object uid + field name
              #                       + collection number
              if fTypeIsEntityComposition:
                fSubChild = self.dictOfProjectedObject[\
```

```
                str(object.getUid())+\
                field.getId()+\
                str(fieldElement.getUid())]
            else:
              fSubChild = self.dictOfProjectedObject[\
                str(object.getUid())+\
                field.getId()+\
                str(fieldCount)]
            if fSubChild.getParent() is not fChild:
              if verboseMode:
                print "Parent feather",\
                  fSubChild.getParent().getId(),\
                  "of feather",fSubChild.getId(),\
                  "is not good: should be",fChild.getId()
              raise KeyError
            if verboseMode:
              print "fSubChild found:",fSubChild
          except KeyError:
            exist = 0
```

Si la feuille `fSubChild` n'est pas trouvée, ou que sa feuille parent n'est pas `fChild`, nous (re)créons une feuille pour cet élément et nous lui associons des mappeurs `"getFieldUiInfo"` et `"getType"`. Le dictionnaire `self.dictOfProjectedObject` est aussi mis à jour :

```
          if not exist:
            #create sub child feather
            fSubChild = SPMLFeather(fChild)
            if fTypeIsEntityComposition:
              self.dictOfProjectedObject[\
                str(object.getUid())+\
                field.getId()+\
                str(fieldElement.getUid())] = fSubChild
            else:
              self.dictOfProjectedObject[\
                str(object.getUid())+\
                field.getId()+\
                str(fieldCount)] = fSubChild
            # Assign methods to fSubChild
            fSubChild.addMethod(getFieldUiInfo)
            fSubChild.addMethod(getType)
```

Une fois que la feuille `fSubChild` a été trouvée ou créée, elle est mise à jour (id, valeur, validité et indicateur de projection) :

```
          #Set last part of feather id
          fSubChild.setId1(fieldElement)
          fSubChild.setValue(fieldElement)
          if fTypeIsEntityComposition:
            fSubChild.setSPMLValidity(\
              fieldElement.isValid())
          else:
            fSubChild.setSPMLValidity(\
              object.isFieldValid(field.getId()))
          fSubChild.indicator = 1
```

Ensuite nous ajoutons les mappeurs "GetValue" et "Delete" (si le mode de définition de l'attribut n'est pas "DERIVED") à la feuille fSubChild :

```
        fSubChild.addMethod(\
          DATA_FunctionWrapper.BoundMethodWrapper(\
            "GetValue",
            getattr(self.reader,"getCField_P"),
            {"object":object,
              "field":field,
              "index":fieldCount}))
        if not fieldIsDerived:
          fSubChild.addMethod(\
            DATA_FunctionWrapper.BoundMethodWrapper(\
              "Delete",
              getattr(self.writer,"deleteCField_P"),
              {"object":object,
                "field":field,
                "index":fieldCount}))
```

Après cela, les mappeurs associés à la feuille fSubChild dépendent du type, du stéréotype et du mode de définition de l'attribut.
Si le type est "Intrinsic" et le mode de définition de l'attribut n'est pas "DERIVED", voici la création du mappeur "SetValue", avec éventuellement un convertisseur selon le type exact de l'attribut :

```
        #
        #If field type is intrinsic
        #
        if fType.is_a("Intrinsic") and\
          not fieldIsDerived:
          setCValueIntrinsic=\
            DATA_FunctionWrapper.BoundMethodWrapper(\
              "SetValue",
              getattr(self.writer,"setCField_P"),
              {"object":object,
                "field":field,
                "index":fieldCount})
          if fType.is_a("INTEGER") or\
            fType.is_a("BOOLEAN"):
            # SetValue for integer
            setCValueIntrinsic.setArgsConverter(2,int)
          elif fType.is_a("DOUBLE") or\
            fType.is_a("NUMERIC"):
            # SetValue for double
            setCValueIntrinsic.setArgsConverter(2,float)
          fSubChild.addMethod(setCValueIntrinsic)
```

Si le type est "Entity" et le stéréotype "ASSOCIATION" ou "AGREGATION", nous recherchons la feuille référencée pour l'affecter à la feuille fSubChild. Nous ajoutons un mappeur "GetPotentialValues" à la feuille fSubChild et si le mode de définition de l'attribut n'est pas "DERIVED", nous lui ajoutons aussi un mappeur "SetValue" :

```
        #
```

```
            #If field type is a reference entity
            #
            elif fTypeIsEntityReference:
              if not fieldIsDerived:
                # SetValue for object reference
                # (not composition)
                fSubChild.addMethod(\
                  DATA_FunctionWrapper.BoundMethodWrapper(\
                    "SetValue",
                    getattr(self.writer,
                      "setCField_Entity_P"),
                    {"object":object,
                      "field":field,
                      "index":fieldCount}))
              #Set reference
              ref = self.dictOfProjectedObject[\
                str(fieldElement.getUid())]
              fSubChild.setReference(ref)
              #Set last part of feather id
              #fSubChild.setId1(str(ref))
              # Assign reference methods to fSubChild
              fSubChild.addMethod(\
                DATA_FunctionWrapper.BoundMethodWrapper(
                  "GetPotentialValues",
                  getattr(self.reader,\
                    "getPotentialValues_Entity_P"),
                  {"field":field}))
```

Enfin si le type est `"Entity"` et le stéréotype `"COMPOSITION"`, nous projetons l'élément (qui est alors une instance de `spml_class`) avec la méthode `self.project` et la feuille `fSubChild` comme feuille parente. L'id (deuxième partie) de `fSubChild` est aussi mis à jour :

```
            #
            #If field type is a composition entity
            #
            elif fTypeIsEntityComposition:
              #Project object and its fields
              self.project(fieldElement,fSubChild)
              #Set last part of feather id
              fSubChild.setId1(str(fieldElement))
```

Une fois l'élément mis à jour, le compteur `fieldCount` est incrémenté :

```
            fieldCount = fieldCount +1
```

Une fois toute la liste parcourue, nous ajoutons un mappeur `"Delete"` à la feuille `fChild`, à la condition que le mode de définition de l'attribut ne soit pas `"DERIVED"` :

```
          if not fieldIsDerived:
              # fieldValue is not None:
              # Create and assign method Delete to fChild
              fChild.addMethod(\
```

```
            DATA_FunctionWrapper.BoundMethodWrapper(\
              "Delete",
              getattr(self.writer,"deleteCField_P"),
              {"object":object,
                "field":field,
                "index":index}))
```

Enfin, que la valeur de l'attribut soit à `None` ou pas, nous ajoutons à la feuille `fChild` un mappeur `"AddElement"` permettant d'ajouter un élément à la liste, voire de créer une liste si l'attribut est vide. Encore une fois la création de ce mappeur dépend de l'attribut.

Si le type est `"Intrinsic"` et le mode de définition n'est pas `"DERIVED"`, voici la création du mappeur `"AddElement"`, avec éventuellement un convertisseur selon le type exact de l'attribut :

```
# Create and assign method AddElement to fChild
if fType.is_a("Intrinsic") and not fieldIsDerived:
  addCValueIntrinsic = \
    DATA_FunctionWrapper.BoundMethodWrapper(\
      "AddElement",
      getattr(self.writer,"setCField_P"),
      {"object":object,
        "field":field,
        "index":index})
  if fType.is_a("INTEGER") or fType.is_a("BOOLEAN"):
    # SetValue for integer
    addCValueIntrinsic.setArgsConverter(2,int)
  elif fType.is_a("DOUBLE") or fType.is_a("NUMERIC"):
    # SetValue for integer
    addCValueIntrinsic.setArgsConverter(2,float)
  fChild.addMethod(addCValueIntrinsic)
```

Voici la création du mappeur `"AddElement"` si le type est `"Entity"` :

```
elif fType.is_a("Entity"):
  # SetValue for Entity
  addCValueEntity = \
    DATA_FunctionWrapper.BoundMethodWrapper(\
      "AddElement",
      getattr(self.writer,"setCField_Entity_P"),
      {"object":object,
        "field":field,
        "index":index})
```

Ce mappeur n'est pas associé tout de suite à la feuille `fChild`. Il faut auparavant tester le stéréotype de l'attribut.

Si le stéréotype est `"ASSOCIATION"` ou `"AGREGATION"`, le mappeur `"AddElement"` n'est pas modifié, et un autre mappeur `"GetPotentialValues"` est associé à la feuille `fChild` :

```
  if fTypeIsEntityReference:
    fChild.addMethod(\
      DATA_FunctionWrapper.BoundMethodWrapper(\
```

```
                "GetPotentialValues",
                getattr(self.reader,\
                  "getPotentialValues_Entity_P"),
                {"field":field}))
```

Si le stéréotype est `"COMPOSITION"`, nous ajoutons un convertisseur au mappeur `"AddElement"` pour que celui-ci créé une nouvelle instance, et un autre mappeur `"GetPotentialValues"` est associé à la feuille `fChild` :

```
            elif fTypeIsEntityComposition:
              # This converter creates a new entity
              # and returns its uid
              addCValueEntity.setArgsConverter(2,\
                self.writer.createEntity)
              fChild.addMethod(\
                DATA_FunctionWrapper.BoundMethodWrapper(\
                  "GetPotentialValues",
                  getattr(self.metaReader,\
                    "getPotentialTypes_P"),
                  {"klass":fType}))
```

Finalement, le mappeur `"AddElement"` est associé à la feuille `fChild`, à la condition que le mode de définition de l'attribut ne soit pas `"DERIVED"` :

```
            if not fieldIsDerived:
              fChild.addMethod(addCValueEntity)
```

Annexe E

Organisation générale du moteur de description physique

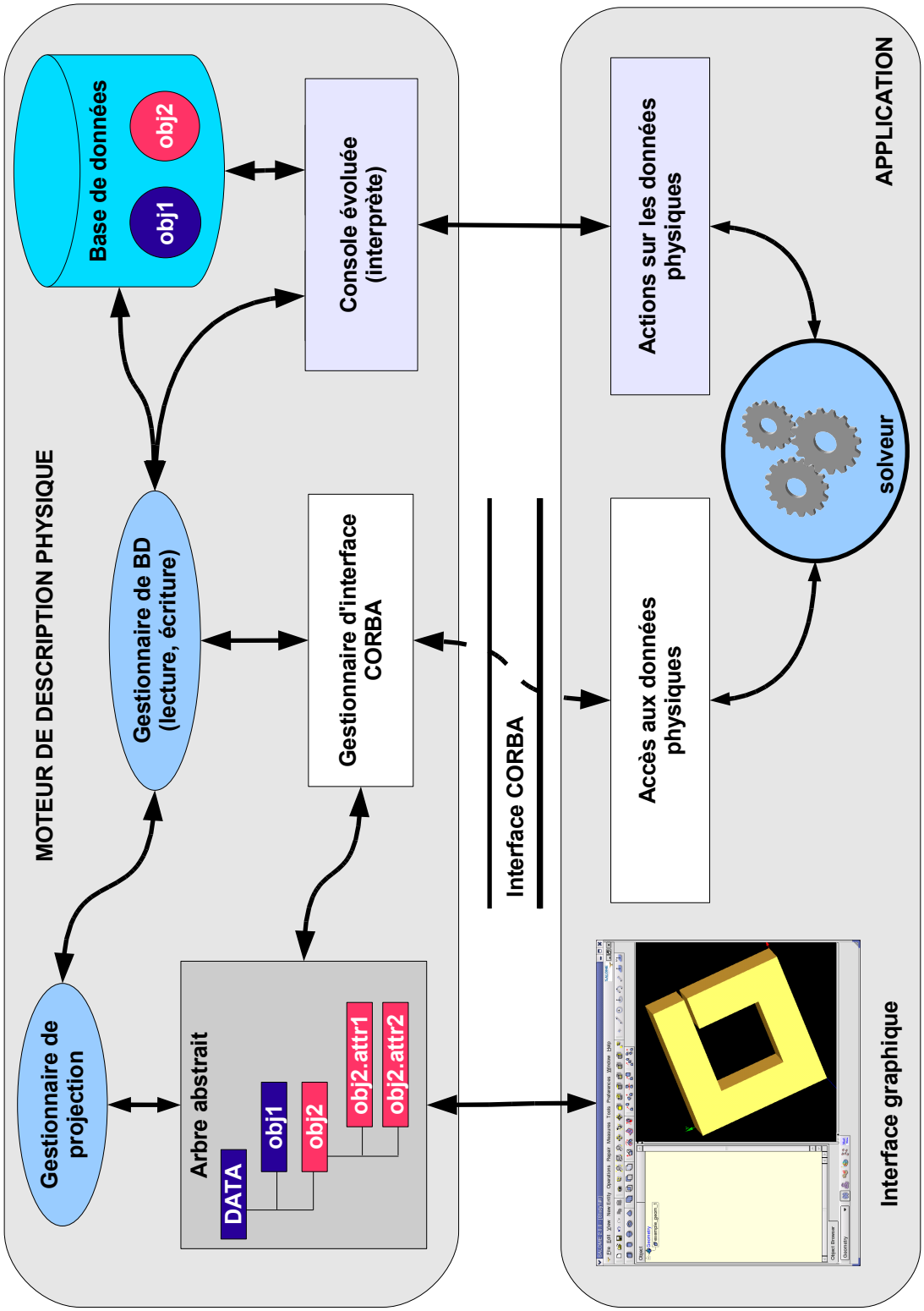


FIG. E.1 – Organisation générale du moteur de description physique

Annexe F

Modèle de données du solveur du CSTB

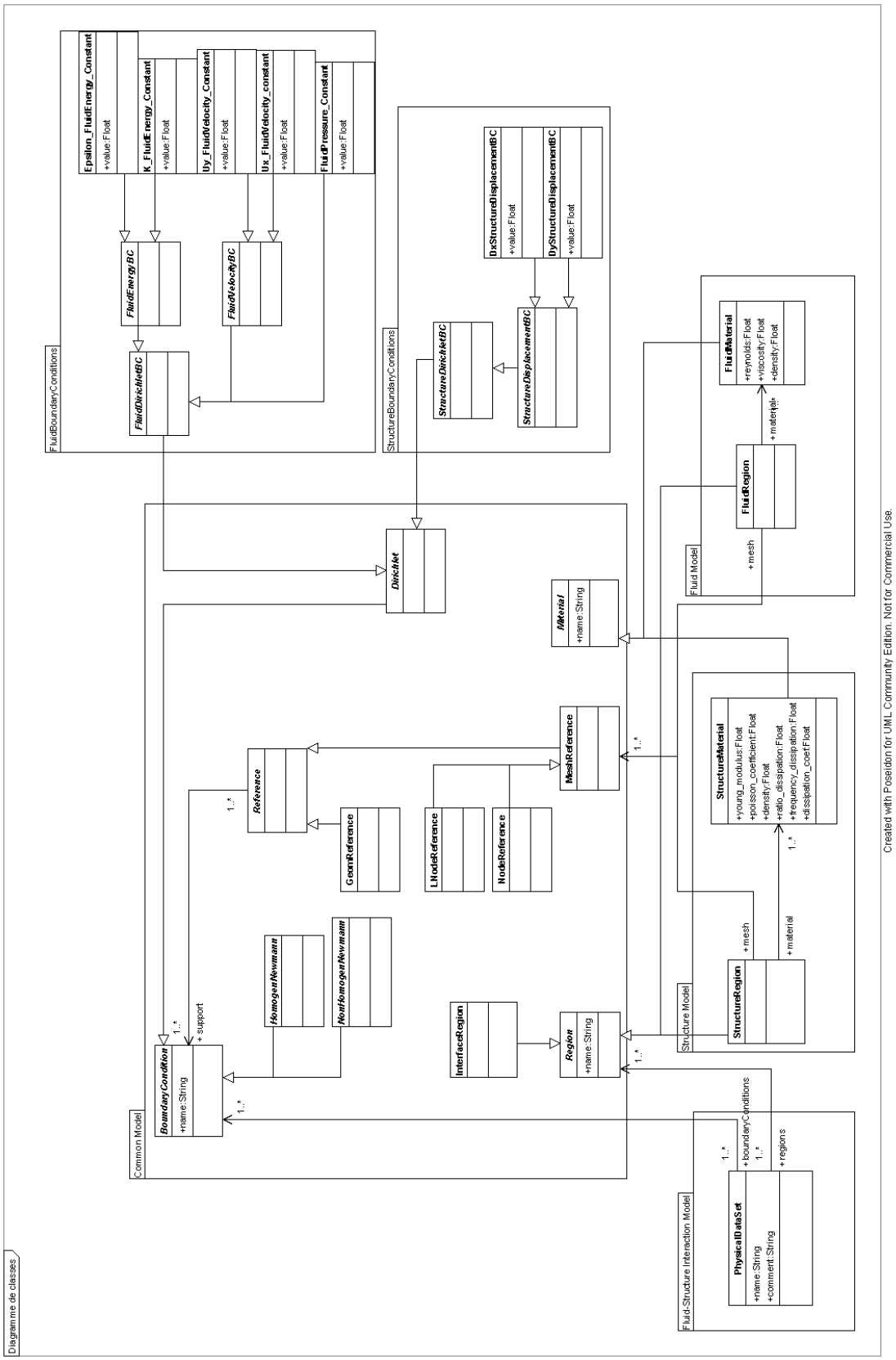


FIG. F.1 – Plaque excitée par des tourbillons – Diagramme de classes du modèle de données

Bibliographie

[1] Réseau National des Technologies Logicielles : http://www.rntl.org.

[2] Emmanuel Vinot, Guillaume Donnier-Valentin, Pascal Tixador, and Gérard Meunier. *AC losses in superconducting solenoids. IEEE Trans. Appl. Superconduct.*, 12(2) :1790–1794, Juin 2002.

[3] Iszabela Klutsch. Modélisation des supraconducteurs et mesures. Thèse de Doctorat, INPG, LEG, 2003.

[4] Le logiciel Flux est développé par la société Cedrat : http://www.cedrat.com.

[5] Quoc Hung Huynh. Gestion de la complexité dans un logiciel dédié à la simulation numérique multi-physique. Thèse de Doctorat, INPG, LEG, 2006.

[6] SALOME platform : http://www.salome-platform.org.

[7] Site internet du langage Python : http://www.python.org.

[8] Shalabh Chaturvedi. *Python Types and Objects*, 2005. http://www.cafepy.com/article/python_types_and_objects/.

[9] OMG. *UML semantics version 1.1*, 1997.

[10] The Lex & Yacc Page : http://dinosaur.compilertools.net/.

[11] ANTLR parser generator : http://www.antlr.org/.

[12] Easter-eggs : http://www.easter-eggs.com.

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

[14] Stephan Giurgea. Traitement unifié des propriétés physiques dans un environnement d'analyse intégré. Thèse de Doctorat, INPG, LEG, 2003.

[15] Site web de CORBA : http://www.corba.org/.

[16] CORBA FAQ sur le site de l'OMG : http://www.omg.org/gettingstarted/corbafaq.htm.

[17] Centre Scientifique et Technique du Bâtiment : http://www.cstb.fr/.

[18] Bibliothèque MED-fichier V2.2 : http://www.code-aster.org/outils/med/.

[19] Extending and Embedding the Python Interpreter : http://www.python.org/doc/2.4.3/ext/ext.html.

[20] Boost C++ libraries : http://www.boost.org.

[21] Simplified Wrapper and Interface Generator (SWIG) : http://www.swig.org.

Résumé

L'objectif de cette étude est de développer des outils informatiques permettant de faciliter la modélisation de phénomènes physiques et de leurs couplages éventuels. Nous partons d'un exemple de résolution de problème multiphysique (le couplage magnéto-thermique dans un ruban supraconducteur) pour aborder la question de la démarche de modélisation. En particulier nous mettons en avant le besoin systématique de description des propriétés physiques du problème traité. Pour répondre à ce besoin, nous proposons d'utiliser un formalisme générique permettant de décrire les propriétés physiques de tout problème numérique. Pour cela, ce formalisme permet de décrire la structure des propriétés physiques, c'est-à-dire leur modèle de données. Ce formalisme se comporte alors comme un modèle de modèles : c'est un métamodèle. Nous présentons ensuite la structure du métamodèle ainsi que des outils et services qui ont développés autour et qui permettent de gérer les modèles de données et les propriétés physiques. Le métamodèle a été réalisé sous la forme d'un langage informatique orienté objet : le SPML. Nous justifions ce choix et nous détaillons la réalisation des principales fonctionnalités du SPML. Enfin nous présentons l'intégration du métamodèle dans la plate-forme de simulation numérique SALOME et son utilisation pour la résolution d'un problème de magnétostatique simple et un d'un problème d'intéractions fluide-structure.

Abstract

The aim of this study is to develop software tools facilitating the modelisation of physical phenomena and of their possible couplings. Starting from the example of the resolution of a multi-physics problem (the magneto-thermal coupling in a superconductor rubber), we tackle the question of the modeling steps. More precisely we emphasize the systematic need for the description of the physical properties of the problem. In order to satisfy this need, we propose to use a generic formalism which allows to describe the physical properties of any numerical problem. For that, this formalism allows the description of the structure of the physical properties, in other words their data model. The formalism hence behaves like a model's model : it is a metamodel. Then we present the structure of the metamodel and of the tools and services which were developped along and are used for the management of the data models and of the physical properties. The metamodel was realised in the form of a object-oriented data-processing language : the SPML. This choice is justified and the main caracteristics of the SPML language are detailled. Finally we present how we integrated the metamodel into the platform for numerical simulations SALOME and how it was successfully used for the resolution of a simple magnetostatic problem and of a fluid-structure interaction problem.