UNIVERSITE DE NANTES

ECOLE DOCTORALE
SCIENCES POUR L'INGENIEUR
DE NANTES

Année 2000

Thèse de **DOCTORAT**

Discipline : Sciences de l'Ingénieur
Spécialité : Automatique et Informatique Appliquée

présentée et soutenue publiquement par

LEMESLE Richard

le 26 octobre 2000
à l'Ecole Centrale de Nantes

Techniques de Modélisation et de Méta-modélisation

Jury :

Président	J.-F. PERROT	*Professeur, Laboratoire Informatique de Paris*
Rapporteur	J.-M. GEIB	*Professeur, Université de Lille*
Rapporteur	L. FERAUD	*Professeur, Université Paul Sabatier de Toulouse*
	M.OUSSALAH	*Professeur, Université de Nantes*
	P. COINTE	*Professeur, Ecole des Mines de Nantes*
	J.-M. JEZEQUEL	*Professeur, IRISA, Rennes*
Invité	Y. LENNON	*Directeur de la division TNT du Groupe Sodifrance*

Directeur de thèse : Jean Bézivin
Laboratoire : Laboratoire de Recherche en Sciences de Gestion (L.R.S.G.), Université de Nantes.

Remerciements.

Je tiens à exprimer mes plus vifs remerciements à Jean Bézivin, professeur à l'université de Nantes, pour m'avoir proposé de réaliser cette thèse et avoir dirigé mes travaux de recherche. Il m'avait déjà encadré lors de mon D.E.A. et cette thèse, effectuée dans un cadre industriel, m'a permis de continuer à mettre en œuvre ces travaux sur des préoccupations telles que la rétro-ingénierie, l'industrialisation du développement et l'ingénierie des modèles. Tout au long de ces quatre années, il a su me guider et me conseiller de façon pertinente.

Je remercie Yves Lennon, directeur de la division Transitique et Nouvelles Technologie du Groupe Sodifrance au sein de laquelle cette thèse a été réalisée. Il a su me faire confiance en acceptant l'intégration de mes travaux dans les outils que sa société distribue. Je remercie également Jean-Paul Bouchet pour ses nombreux conseils, le suivi constant de mes travaux et les remarques essentielles qu'il a formulé sur ce document.

Je remercie vivement Jean-Marc Geib et Louis Feraud qui ont accepté d'être rapporteurs.

Je remercie Xavier Blanc, Xavier Le Pallec, Juliette Le Delliou, Francois Jocteur-Monrozier, Michelle Sibilla, Mariano Belaunde, Mickaël Peltier et tous les autres membres du groupe de travail "meta" qui ont permis à chacun, au cours de réunions riches en exposés et en discussions de qualité, de valider un certain nombre d'idées.

Je remercie Stéphane Ducasse, de l'Institut de Mathématiques Appliquées de Bern pour ses commentaires et son accompagnement sur la rédaction de l'une de mes publications.

Enfin, je remercie Frédéric, Charles, Sébastien, David, Erwan, Bruno, Jean-Sébastien, Hervé et tous mes autres collègues de la société Soft-maint avec qui j'ai eu le plaisir de travailler tout au long de ces travaux.

Table des Matières

1 Introduction.

Cette thèse a été réalisée dans le cadre d'une convention CIFRE au sein d'une société de services en informatique spécialisée dans la maîtrise et l'évolution des systèmes d'information. La capacité à proposer des outils pour appréhender rapidement les systèmes d'information existants et les faire évoluer est alors indispensable.

En effet, pour réaliser efficacement cette tâche, il faut être en mesure de prendre connaissance du système d'information existant. Cette prise de connaissances passe par l'analyse de tous les éléments composants ce système (des applications aux bases de données en passant par les fichiers de commande) et l'intégration de ceux-ci dans un référentiel. Ce référentiel permet alors de disposer d'une vue globale d'un "site" (le site correspond à l'ensemble des composants logiciels d'un client donné).

Il faut également être à même de prendre en charge l'évolution et la maintenance de cet existant. On se base alors sur les informations récoltées dans le référentiel pour effectuer les modifications liées à cette évolution ou cette maintenance plus rapidement et plus efficacement.

Il faut aussi tenir compte de toutes les évolutions conjoncturelles telles que le passage de l'an 2000 ou encore la prise en compte de la monnaie européenne. Des outils spécifiques peuvent alors être réalisés pour effectuer des transformations systématiques ou encore pour assister les développeurs lors de la recherche des éléments à modifier.

De nos jours, il faut également penser que de nombreuses opérations de migrations de systèmes d'information doivent être réalisés à la suite de fusions de sociétés aux seins desquelles ces systèmes étaient différents. Il est alors indispensable d'intégrer toutes les données du premier site sur le second en assurant l'intégrité de celles-ci avec les contraintes du nouveau système. Les référentiels de chaque site peuvent alors être utilisés pour permettre une mise en correspondance de leurs données respectives.

Enfin, il faut prendre en charge l'évolution de ces systèmes d'information vers les "nouvelles technologies". Il y a quelques années encore, il s'agissait de faire migrer ces systèmes vers des langages et modèles à objets. Maintenant, il est nécessaire de prendre en compte les technologies de composants, le réseau Internet ainsi que les serveurs d'applications.

L'un des soucis majeurs de cette société est par conséquent la structuration de cette connaissance dans les référentiels. Une représentation uniforme de ces informations, bien que très diverses,

apporte de la clarté, de la visibilité et par conséquent des possibilité de réactivité et d'automatisation. De plus, la structure de ces informations pouvant varier d'un site à l'autre, il est nécessaire de rendre les outils le plus indépendant possible de celle-ci de manière à pouvoir les réutiliser sans trop de difficultés. En effet, deux entreprises peuvent utiliser le même système informatique et l'exploiter de façon différente de sorte qu'il est nécessaire de prendre en compte les particularités de chacun de ces deux sites.

Initialement spécialisée dans la transformation et la manipulation de programmes sources, les activités de la société s'orientent de plus en plus vers la manipulation et la transformation de modèles. Les outils génériques basés précédemment sur les grammaires des fichiers sources analysés ont alors du être repensés pour se baser sur la sémantique des modèles manipulés.

De plus, la société travaille depuis quelque temps sur la modélisation de processus et le couplage de cette modélisation avec les moteurs de Workflow de manière à industrialiser également les processus de migration et de maintenance. Le but est alors d'automatiser les enchaînements d'activités et de mécaniser les flux entre les activités et les acteurs à l'aide de formulaires, d'outils de messagerie, et de lancements automatisés d'outils spécifiques. C'est également un travail d'ingénierie sémantique consistant à faire inter-opérer les produits utilisés dans ces différents processus. Cette interopérabilité passe alors par l'échange de modèles via ce que nous appelons des ponts sémantiques (ces ponts sont basés sur la sémantique des modèles échangés). Par sémantique des modèles, nous entendons leurs méta-modèles (nous nous désintéressons ici des aspects notation ou langage de contraintes qui, ajoutés à l'aspect sémantique, composent généralement les méta-modèles).

L'objectif de cette thèse était donc de proposer un formalisme capable de supporter la modélisation d'un ensemble de programmes Cobol de la même façon qu'une modélisation à objets dans un exprimée à l'aide de UML (Unified Modeling Language) ou encore une modélisation de processus. Il fallait que ce formalisme permette la représentation de ces informations de façon structurée de sorte qu'un ensemble d'outils puissent s'appuyer sur cette structure des informations. Il est en effet important que les outils conçus en interne dans la société soient inter-opérable entre-eux mais également avec les outils d'origine externe. La notion de bus sémantique permettant d'effectuer ces ponts sémantiques correspond à cet objectif d'inter-opérabilité.

De plus, une fois le choix de ce formalisme effectué, il fallait déterminer dans quelles mesures des outils génériques basés sur celui-ci pouvait répondre en totalité ou partiellement aux besoins énoncés précédemment (transformations, évolutions conjoncturelles, évolutions vers les nouvelles technologies, etc…).

Nous avions déjà travaillé, au cours d'un DEA en 1994, sur la définition d'un formalisme appelé "sNets" et permettant de représenter toutes les informations manipulées au cours du cycle de développement du logiciel. A l'époque, nous recherchions donc déjà un moyen de supporter la représentation de données très diverses. Les principaux formalismes étudiés alors étaient le formalisme des réseaux sémantiques et ses dérivés tels que les graphes conceptuels. Le résultat du stage de DEA a été de donner une première définition de sNets et c'est bien naturellement sur ce travail que s'est appuyée par la suite cette thèse.

La définition des sNets s'est initialement inspirée du système P.I.E. (Personal Information Environment) réalisé par Dan Bobrow et Ira Goldstein au P.A.R.C. (Palo Alto Research Center). Malheureusement, ce projet P..E implémenté en Smalltalk-76 n'a pas été poursuivi de façon significative, ni à l'intérieur, ni à l'extérieur de la société Xerox. Nous devons à P.I.E. l'idée centrale de ne pas représenter directement une application en langage Smalltalk, mais au contraire de rajouter un niveau d'abstaction supplémentaire s'appuyant sur le mécanisme des réseaux sémantiques (en fait, le projet P.I.E. s'inspirait lui-même de réseaux sémantiques partitionnés de Hendrix).
Par rapport à notre source initiale d'inspiration, nous en avons à la fois réduit et étendu les ambitions. Dans les premières mises en œuvre des sNets, nous avions incorporé les puissants concepts de strates et de versions proposés par P.I.E. Dans le présent travail, nous avons abandonné ces mécanismes car ils relèvent d'une problématique complexe de gestion de version de modèles et des méta-modèles. Cette problématique de gestion de versions n'est pas la notre ici, mais elle correspondra très certainement à l'un des axes de développement dans un futur proche.

Par contre, pour atteindre nos objectifs initiaux de représentation d'un système d'information existant, nous avons étendu le mécanisme des sNets en lui rajoutant un **système de typage**, en lui intégrant une **gestion de la modularité** et en le rendant "**réflexif**" (auto-défini). Ces trois extensions n'étaient pas présentes dans le système P.I.E. et constituent le cœur de notre contribution. Nous avons identifié ces trois propriétés comme nécessaires à nos objectifs industriels de représentation de systèmes d'information. Ce n'est que plus tard que nous avons constaté des similitudes entre notre proposition des sNets et d'autres propositions de recherche ou de développement comme les graphes conceptuels, CDIF, MOF, etc… que nous présentons également ici. Actuellement, les sNets constituent à la fois :

- la base de représentation de nombreux outils propriétaires de la société dans laquelle cette thèse a été réalisée,

- un modèle conceptuel qui nous permet d'analyser et de comparer de approches similaires comme le MOF ou CDIF.

Les trois mécanismes de base introduits dans les sNets répondent à des besoins différents mais corrélés.

- Le typage a permit de canaliser la puissance expressive des réseaux sémantiques. Le système unique de typage est représenté dans notre proposition par la relation meta(x,X) exprimant que x est de type X. Cette relation est globale à notre système de représentation.

- D'autres relations de typage peuvent être définies que l'on peut qualifier de typage "local" ou "régional". Par exemple, on peut définir la relation instanceOf(medor, Chien) indiquant que l'objet Smalltalk medor est une instance de la classe Smalltalk Chien. Cette relation correspond à un typage "régional" car elle est contextuelle à une représentation d'un programme Smalltalk. De façon globale, la relation meta sera utilisée pour indiquer que medor est un objet Smalltalk et Chien est une classe Smalltalk. Bien évidemment, pour que ceci ait du sens, il fallait pouvoir expliciter les régions où ces systèmes spécialisés de typage peuvent s'appliquer. Nous avons donc introduit le concept d'univers pour représenter explicitement les espaces de définition ou d'utilisation des typages régionaux spécifiques. La notion d'univers permet d'établir une partition de l'espace global. Des notions plus ou moins équivalentes ont ensuite été identifiées dans les autres formalismes étudiés telles que les "packages" du MOF, les "subject areas" de CDIF ou encore les "théories" de KIF. L'une des originalités des sNets est l'intégration des trois notions de concept, de relation et d'univers. C'est cette intégration qui différencie les sNets d'un simple schéma entité-association.

- La troisième propriété de base des sNets est son auto-définition. Elle s'appuie sur un espace noyau minimal dont la finalité est de définir les trois concepts de base (concept, relation et univers). Pour des raisons opérationnelles, il a été nécessaire d'étendre ce noyau pour y ajouter d'autres notions représentées sous la forme de nouveaux concepts et de nouvelles relations. Mais notre objectif essentiel a toujours été de conserver l'aspect minimal de ce noyau. Tous les concepts de l'espace noyau sont définis dans cet espace lui-même et c'est pourquoi nous le qualifions d'espace réflexif.

Dans la suite de ce chapitre introductif, nous allons reprendre la présentation de notre contribution de façon plus détaillée. Notre formalisme des sNets sera d'abord présenté comme une extension de réseaux sémantiques. Nous donnerons ensuite une présentation de formalismes CDIF de l'E.I.A. et des formalismes UML, MOF et XMI de l'O.M.G. Ayant ainsi présenté notre propre formalisme et les standards industriels du marché, nous serons alors en mesure de mieux situer la contribution de la thèse.

1.1 Le formalisme des sNets.

Ce formalisme s'était donc largement inspiré des travaux de Bobrow et de Goldstein sur la définition d'un environnement nommé P.I.E. (Personal Information Environment) permettant la manipulations d'informations représentées sous la forme d'un réseau sémantique (c. f. Figure 1).

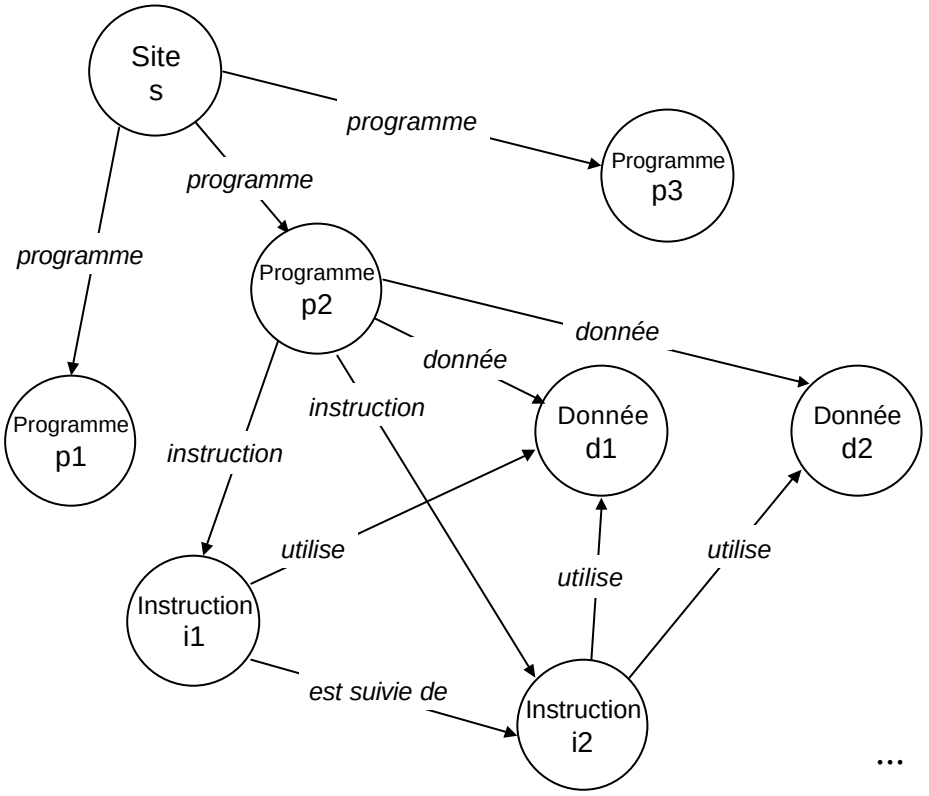


Figure 1 - Description des programmes d'un site représentée sous la forme d'un réseau sémantique.

Dans un réseau sémantique, les informations sont représentées sous la forme d'un graphe dont les nœuds représentent les entités et les liens libellés représentent les relations entre ces entités. L'avantage de ce formalisme est qu'il est à même de représenter tout type d'information. Ainsi, un modèle à objet peut également être représenté sous la forme de réseau sémantique (Figure 2) de la façon suivante :

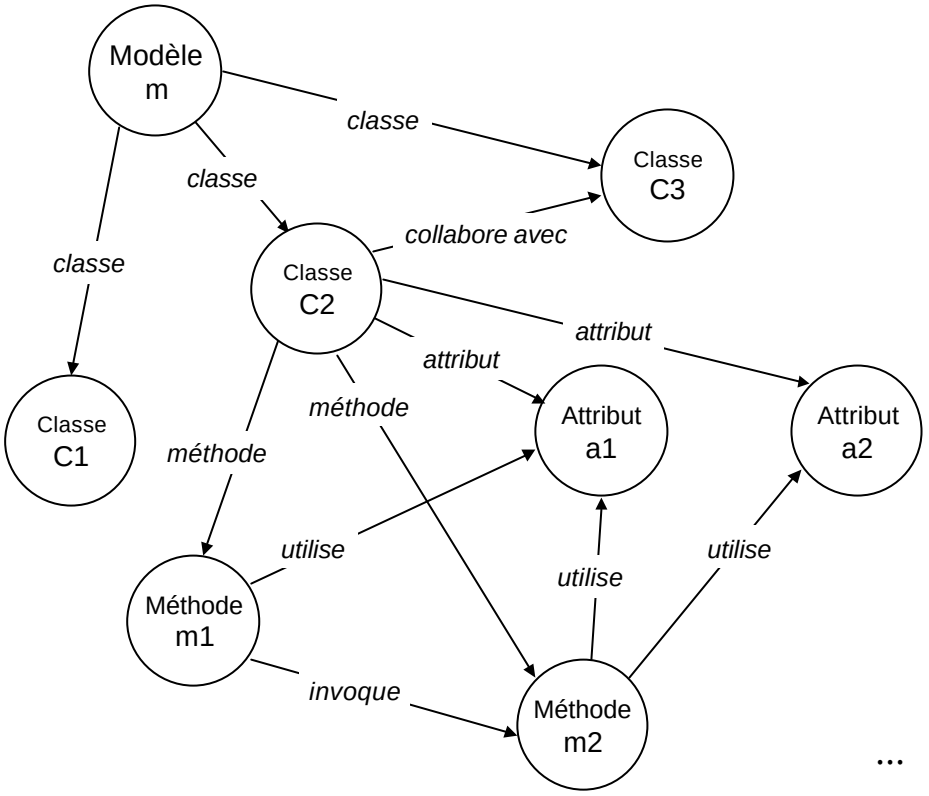


Figure 2 - Modèle à objets représenté sous la forme d'un réseau sémantique.

Il nous manquait alors un moyen de définir la "sémantique" d'un tel réseau. En effet, il fallait être capable d'exprimer le fait qu'un site est rattaché aux programmes qu'il contient via un lien nommé "programme", que les données d'un programme sont rattachées à celui-ci via un lien nommé "donnée", etc… Mais il fallait également être capable d'exprimer le fait que seuls ces liens et ces types étaient valides pour le domaine concerné. Cette notion de domaine fermé est essentielle en modélisation. En effet, dans un réseau sémantique représentant un modèle à objets, il ne doit pas y avoir d'entités représentant par exemple des programmes Cobol. De la même façon, dans un réseau sémantique représentant les programmes présents sur un site, on ne doit pas trouver d'entités représentant de classes. Il fallait de plus que cette sémantique (constituée ici de l'ensemble des types des nœuds et l'ensemble des liens valides entre ces nœuds) soit décrite de façon **structurée** et **modulaire** de manière à ce qu'elle soit clairement identifiée et réutilisable.

Nous avons alors été influencés par Smalltalk, qui était l'un des langages à objets les plus en vogue à l'époque. Ce langage nous avait séduit par sa simplicité, sa minimalité et son extensibilité et nous espérions que notre formalisme dispose de ces mêmes avantages. Nous

sommes donc partis sur les concepts de base des réseaux sémantiques que sont les notions de nœuds et de liens auxquels nous avons ajouté la notion de typage. Nous avions alors un formalisme minimal au sein duquel tout pouvait se ramener à un ensemble nœuds et de liens. Il nous suffisait alors de proposer des mécanismes permettant la définition de nouveaux types de nœud et de nouveaux types de liens de manière à rendre notre formalisme suffisamment extensible et configurable.

Il était ensuite nécessaire de définir les types des nœuds et des liens permettant la représentation de modèles pour chaque domaine d'intérêt : description de programmes Cobol ou Java, mais également description de modèles à objets et de modèles de composants, etc...

Un problème majeur était alors la formalisation de tels ensembles de types de nœuds et de liens. Toujours influencés par Smalltalk, nous nous sommes intéressés à l'aspect réflexivité de ce langage et nous nous sommes demandés pourquoi ne pas utiliser notre formalisme pour représenter ces types de nœuds et de liens. En effet, si le formalisme est à même de représenter n'importe quel type d'information, il doit être à même de représenter les types de nœuds et de liens permettant de décrire des types de nœuds et de liens.

Il nous suffisait donc de définir un type pour la représentation de types de nœuds, un type pour la représentation des types de liens, un lien pour exprimer le fait qu'un nœud d'un type donné peut disposer d'un lien d'un type donné et un lien pour exprimer le fait qu'un lien d'un type donné doit aboutir vers un nœud d'un type donné. Dans les deux figures qui suivent (Figure 3 et Figure 4), ces éléments ont été nommés respectivement "Nœud", "Lien", "peut disposer d'un lien" et "vers un nœud de type".

La Figure 3 représente alors la sémantique utilisée précédemment pour définir la Figure 1. Elle permet effectivement d'exprimer le fait qu'un site peut disposer d'un lien nommé "programme" vers ses programmes et qu'un programme est rattaché à ses instructions par un lien nommé "instruction", etc…

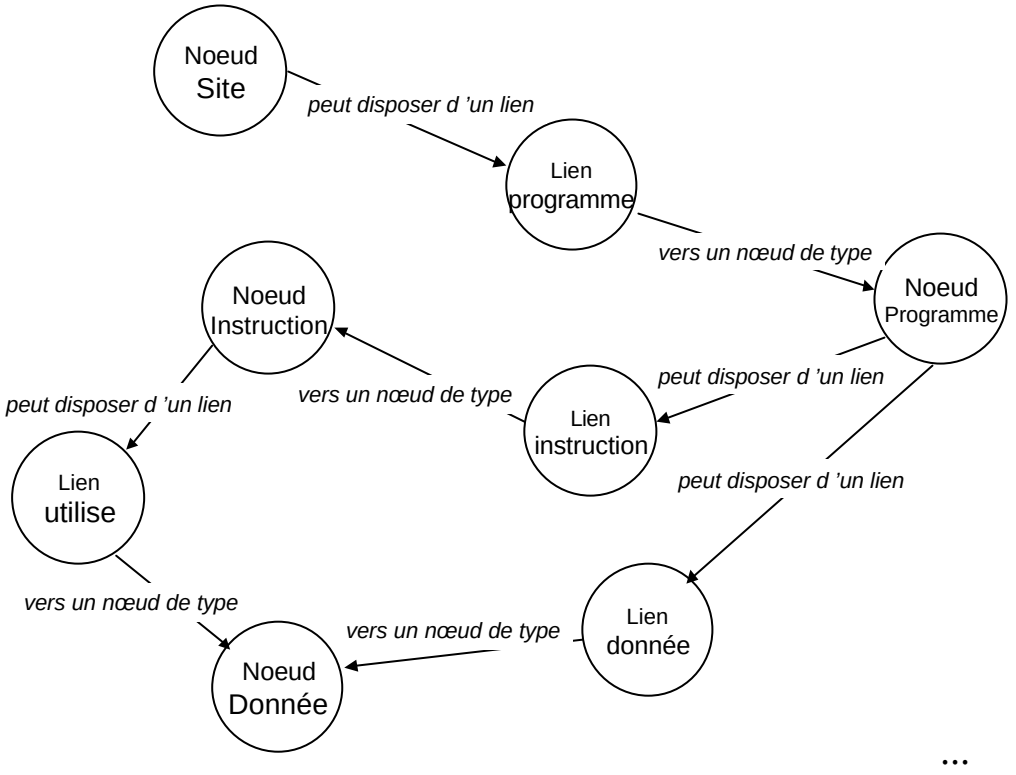


Figure 3 - Sémantique de description des programmes d'un site représentée sous la forme d'un réseau sémantique.

La sémantique d'un modèle à objets peut alors également être exprimée de cette façon ainsi la Figure 4 décrit la sémantique utilisée pour définir le modèle à objets représenté sur la Figure 2. Nous exprimons ici le fait qu'un modèle est constitué de classes et que ces classes sont rattachées à leur modèle via un lien nommé "classe" de même que le fait qu'une méthode peut être liée aux attributs qu'elle manipule via un lien nommé "utilise" ou encore qu'une classe est composée d'attributs et de méthodes respectivement rattachés à celle-ci via des liens nommés "attribut" et "méthode".

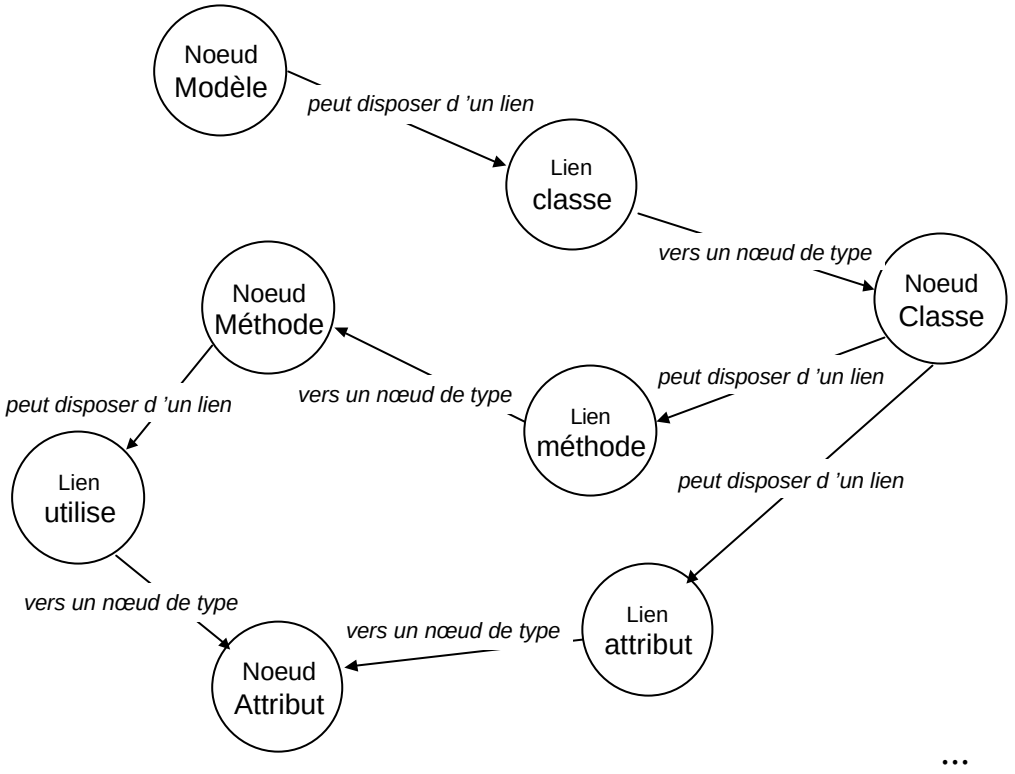


Figure 4 - Sémantique d'un modèle à objets représentée sous la forme d'un réseau sémantique.

Malgré ces descriptions, il est évidemment toujours possible d'aboutir à des représentations d'informations invalides. En effet, il est possible de dire qu'une méthode utilise un attribut alors qu'elle ne l'utilise pas. Par contre il sera impossible de dire qu'une méthode utilise un modèle car cette relation n'est pas définie entre une méthode et un modèle. Les éléments que nous représentons ainsi sous la forme d'un réseau sémantique sont alors nécessairement "sémantiquement" corrects.

Ayant ainsi trouvé un moyen d'exprimer la "sémantique" de nos réseaux, nous nous sommes dits que celle-ci pouvaient, pour nos besoins, être ramenée à un ensemble de types de nœuds et de types de liens. De cette façon, rien ne nous empêchait d'exprimer également celle-ci avec notre formalisme.

Nous avons alors abouti à une expression réflexive de cette sémantique représentée sur la Figure 5 :

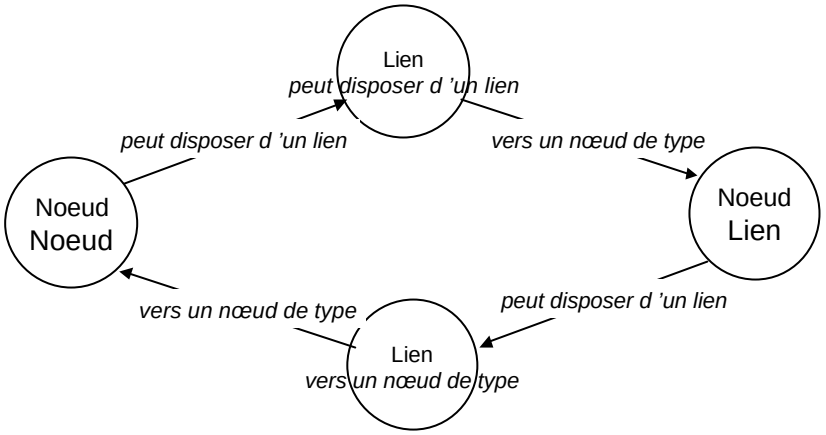


Figure 5 - Une description réflexive de la sémantique d'un réseau sémantique.

Nous avions donc défini un formalisme basé sur les réseaux sémantique qui était minimal, réflexif et extensible. Il était extensible car de nouvelles fonctionnalités pouvaient être ajoutées simplement en définissant de nouveaux types de nœuds ou de liens.

Ainsi, la notion de modularité nécessaire pour la prise en compte de la modélisation de systèmes complexes touchant a différents domaines d'application a pu être facilement intégrée sous la forme d'un nouveau type de nœud que nous appelons univers. Un univers correspond alors à la notion de modèle. Il contient un ensemble de nœuds et de liens spécifiques à un domaine particulier. De plus, dans notre formalisme, la sémantique d'un univers est définie dans un autre univers appelé univers sémantique. Ainsi, un modèle à objets sera représenté dans un univers dont l'univers sémantique définit les notions de classes, de méthodes, d'attributs, etc… Cet univers sémantique sera exprimé comme indiqué sur la Figure 4. Un univers sémantique correspond alors à la notion de méta-modèle. De la même façon, un univers sémantique dispose d'un univers sémantique. Ce dernier est alors appelé méta-méta-modèle. Il définit la sémantique nécessaire à la définition de méta-modèles. Une partie de son contenu est alors exprimé par la Figure 5.

Cette architecture de modélisation est essentielle. Nous allons ainsi présenter dans cette thèse comment de nombreux mécanismes tels que la représentation interne des modèles, leur

manipulation, la recherche d'information dans ces modèles ou encore leur transformation va profiter d'une telle architecture et ce indépendamment de la sémantique de ces modèles. Lorsque cette thèse a débuté en 1997, nous avions donc la possibilité de réutiliser le formalisme des sNets, mais d'autres formalismes avaient vu le jour dans le même temps. En effet, l'EIA (Electronic Industries Association) travaillait depuis 1987 sur la définition d'un format d'échange des informations manipulées dans les ateliers de génie logiciel. Nous avons découvert ce format d'échange après la définition de l'architecture générale des sNets et ce formalisme n'a donc pas eu d'impact direct sur notre modèle de représentation. Cependant, la confrontation des deux schémas et le constat de la similarité des architectures nous a conforté dans les phases ultérieures de notre travail.

1.2 Le formalisme CDIF de l'EIA.

Le format d'échange, appelé CDIF (CASE Data Interchange Format), devait permettre l'échange d'informations entre outils différents. En effet, de plus en plus d'outils sont utilisés au cours du développement d'un logiciel. Des outils de modélisation, des outils d'analyse et de conception, des outils de documentation, de outils de gestion de projet ou encore des outils de programmation sont couramment utilisés sur le même projet. La difficulté est alors de permettre l'échange d'informations entre ces différents outils, et surtout de conserver ainsi la cohérence des données qu'ils représentent.

Ces travaux ont abouti, en 1994, à un ensemble de standards EIA permettant l'échange de modèles. Le point fort de ce formalisme est qu'il est indépendant de l'outil utilisé pour effectuer la modélisation et indépendant de la méthode employée dans cet outil. Pour répondre à ce besoin, le format CDIF est basé sur une architecture formelle qui permet de définir les relations entre les informations représentées, la sémantique de ces informations, les règles d'extension de cette sémantique ainsi que le format syntaxique à utiliser pour représenter ces informations.

Cette architecture formelle se décompose en quatre niveaux représentés sur la figure suivante :

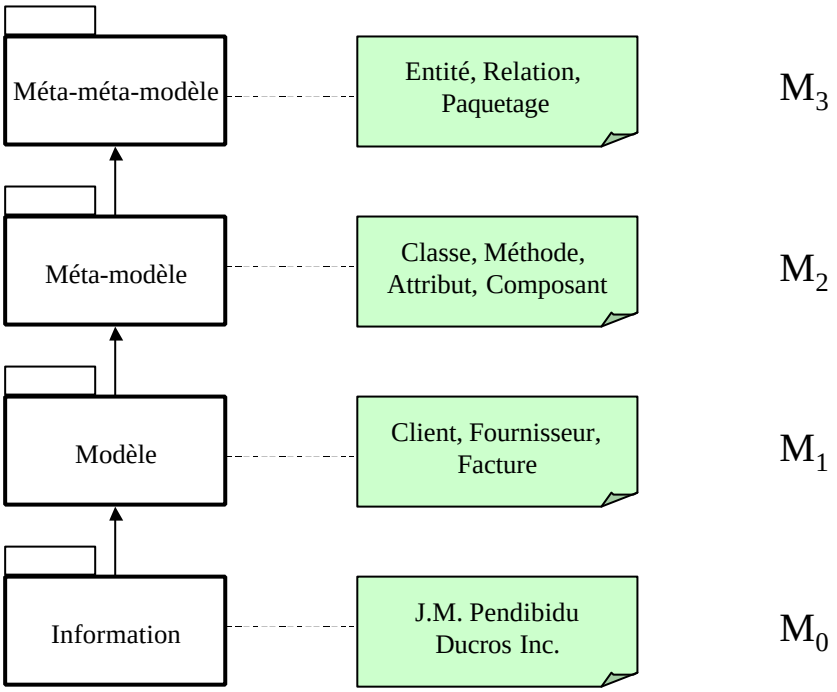


Figure 6 - Une architecture de modélisation à quatre niveaux.

Cette architecture à quatre niveaux est analysée dans le chapitre 6 qui la présente dans un cadre formel de manière à mieux l'appréhender. Le niveau le plus "haut" (s'il on considère que l'on représente toujours un méta-modèle dans un niveau supérieur à un celui d'un modèle) est appelé le méta-méta-modèle. Il regroupe les concepts nécessaires à la définition des méta-modèles. C'est le niveau le plus haut car il est généralement réflexif (il est son propre méta-modèle). Lorsqu'il ne l'est pas, il est tout de même considéré comme niveau le plus haut car on n'attache pas d'importance à un niveau supérieur. Ce niveau est communément appelé le niveau M_3 (le 3 désigne les trois M de Méta-Méta-Modèles). L'aspect réflexivité du méta-méta-modèle est détaillée dans le chapitre 5. Ce chapitre montre le rôle essentiel de cette propriété issu des langages de programmation.

Le niveau suivant contient les méta-modèles. Ces méta-modèles sont définis à partir du langage défini par le méta-méta-modèle (un méta-méta-modèle est presque toujours unique). Ce niveau est communément appelé le niveau M_2 (le 2 désigne les deux M de Méta-Modèles). Chaque

méta-modèle va ainsi définir les concepts nécessaires pour la modélisation d'un domaine particulier.

Le niveau suivant contient les modèles. Ces modèles sont définis à partir du langage exprimé par l'un des méta-modèles du niveau supérieur. Ce niveau est communément appelé le niveau M_1 (le 1 désigne le M de Modèles).

A ce stade, on peut considérer que l'architecture de méta-modélisation est complète, où bien on peut décider d'ajouter un niveau supplémentaire qui sera utilisé pour représenter les information ou données modélisées. Ce niveau est alors appelé le niveau M_0 (en effet, il désigne de l'information ou des données, mais ne correspond en aucun cas à un modèle).

Dans ce contexte, CDIF propose un méta-méta-modèle ainsi qu'un ensemble de méta-modèles standards permettant la manipulation de modèles diverses. L'extension et la définition de nouveaux méta-modèles est alors possible en respectant les règles et le langage défini par le méta-méta-modèle de CDIF. CDIF propose également un format de représentation texte de ces modèles, de leurs méta-modèles ainsi que de l'extension de leurs méta-modèles. Des outils se basant sur ces différents éléments peuvent alors s'échanger de l'information comme indiqué sur la Figure 7.

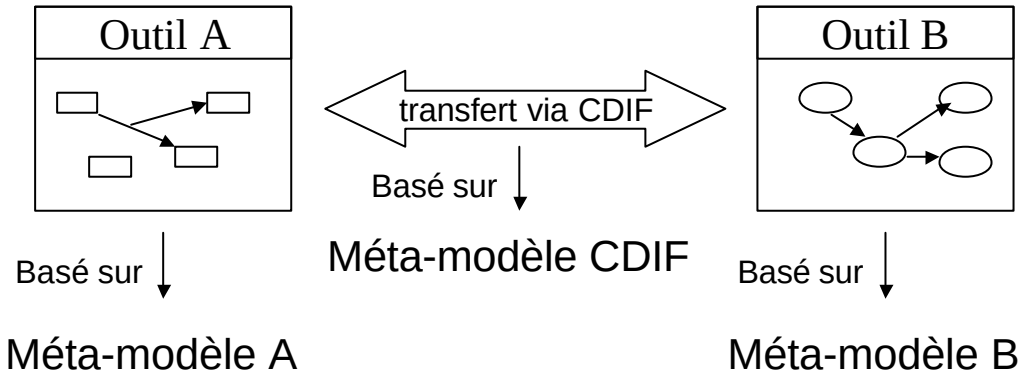


Figure 7 - Transfert de modèles entre outils via CDIF.

Un certain nombre d'outils de modélisation ont alors proposé d'importer et d'exporter les modèles au format CDIF, mais chacun de ces outils se basait sur son propre méta-modèle. En effet, peu de méta-modèles ont été standardisés à l'aide de CDIF. Ainsi, en juin 1997, seuls les méta-modèles permettant la modélisation de données et la modélisation de flots de données était standardisés. Par conséquent, les outils de modélisation à objets et les outils de modélisation de

processus ne disposaient pas de méta-modèles standards CDIF sur lesquels s'appuyer. Chaque outil a alors redéfini son propre méta-modèle en utilisant le méta-méta-modèle proposé par CDIF. Les outils étaient alors capables d'exporter des données au format "syntaxique" CDIF, mais ils n'étaient capables d'importer que les données qu'ils avaient eux-même exporté dans ce format.

En effet, pour permettre un véritable échange entre outils, il est nécessaire de se basé sur un méta-modèle commun à l'aide de convertisseurs (comme indiqué sur la Figure 8). Or les premières implémentations de CDIF se contentaient d'utiliser les encodeurs et décodeurs syntaxiques. Un modèle a exprimé à l'aide d'un méta-modèle A ne pouvait alors pas être importé dans un outil attendant un modèle exprimé à l'aide d'un méta-modèle B.

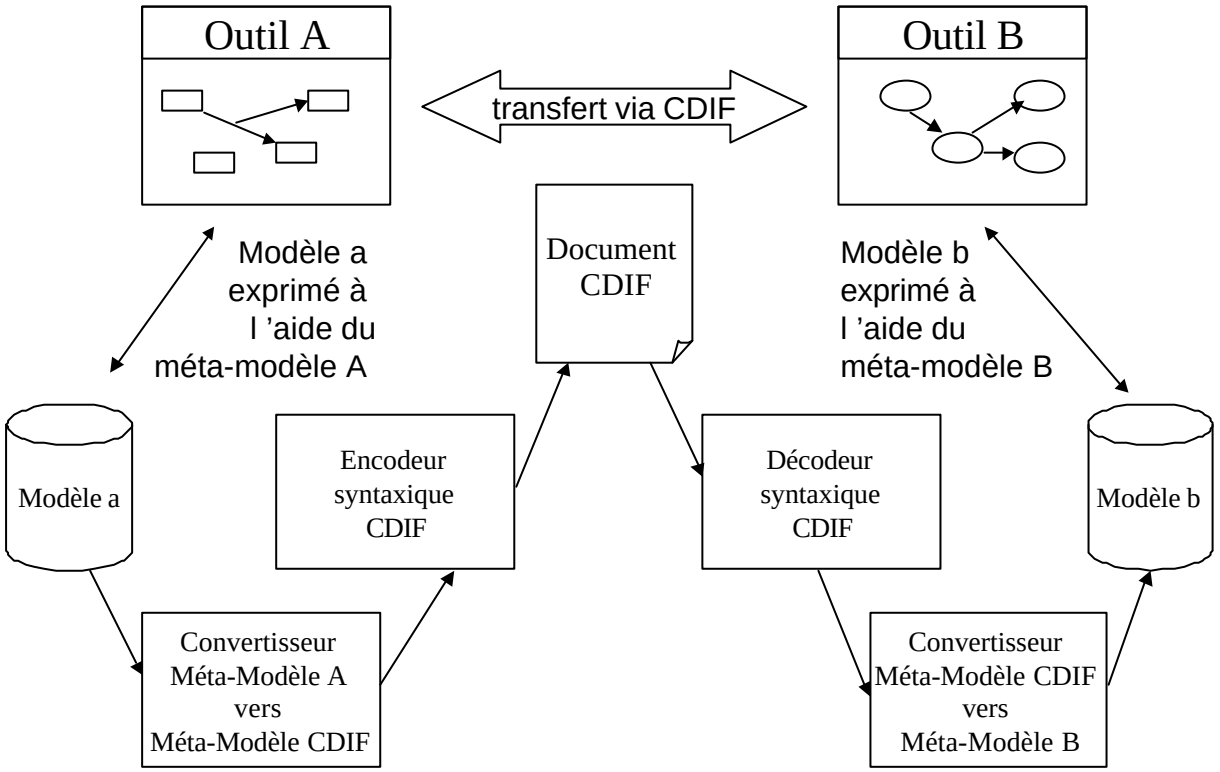


Figure 8 - Implémentation des mécanismes de transfert via CDIF.

La standardisation d'un méta-modèle CDIF supportant la modélisation de l'analyse et de la conception à objets n'a quant à elle pas aboutie car au même moment, un autre groupe de standardisation était sur le point de proposer UML comme notation standard dans ce domaine.

Les industriels ont alors supposé qu'avec un standard tel qu'UML, le problème d'échange de modèles UML entre outils ne se poserait plus.

1.3 Les standards UML, MOF et XMI de l'OMG.

L'OMG (Object Management Group) est un autre organisme de normalisation qui s'est intéressé à la standardisation de langage de modélisation de manière à permettre à différents interlocuteurs de se comprendre en employant ce même langage. L'OMG a été fondée en avril 1989 par onze groupes industriels (3Com Corporation, American Airlines, Canon, Inc., Data General, Hewlett-Packard, Philips Télécommunications N.V., Sun Microsystems et Unisys Corporation). Elle est actuellement composée de plus de 800 sociétés membres. Son rôle est de définir des standards industriels commercialement opérationnels et basés sur des spécifications ouvertes et indépendantes d'un fournisseur.

1.3.1 Les travaux de standardisation de l'OMG.

Les préoccupations de l'OMG sont actuellement en cours d'évolution. Alors qu'il s'agissait initialement de faire communiquer les outils via un mécanisme technique indépendant et qu'à cette fin CORBA et le protocole IIOP ont été définis, elle s'oriente maintenant, en amont du développement, vers l'interopérabilité, l'échange et la distribution de modèles.
Un groupe de travail s'est constitué au sein de l'OMG avec pour préoccupation l'analyse et la conception. Ce groupe de travail est appelé ADTF (Analysis & Design Task Force) et son rôle est le suivant :

- Permettre à des développeurs de mieux comprendre comment développer des applications et autres systèmes répartis de grande envergure.
- Recommander des architectures et les technologies associées en terme de modèles et de méta-modèles pour permettre l'interchangeabilité des produits utilisés ainsi que l'interopérabilité des outils et des référentiels.
- Promouvoir des standards en terme de techniques de modélisation de manière à améliorer la rigueur et la consistance des spécifications.
- Se mettre à niveau et interopérer avec les autres spécifications de l'OMG.
- Mettre en contact les différents organismes ayant des besoins similaires.

Au sein de ce groupe, comme au sein des différents groupes de travail de l'OMG, les besoins sont identifiés dans un premier temps lors de réunions et donnent lieu à des RFI (Request for Information) - demandes d'informations sur un sujet particulier. Lorsque ces informations semblent suffisamment pertinentes pour qu'une standardisation soit proposée dans ce domaine, le RFI donne lieu à un RFP (Request for Proposition) - demande de proposition pour répondre à ce besoin. Ces réunions ont lieu environ cinq fois par an de manière a aboutir rapidement à des standards.

Actuellement, dans ce groupe de travail, trois RFP ont été menés à leur terme et ont donné naissance à des standards de l'OMG :

- Le RFP **OA&D Facility** a été lancé suite à un besoin de standardisation en terme de langage de modélisation de système informatique. Ce RFP, lancé le 6 juin 1996, est terminé et a donné lieu à l'adoption d'UML comme standard de modélisation le 19 novembre 1997.

- Le RFP **Meta Object Facility** a été lancé suite à un besoin de standardisation en terme de méta-modélisation (langage de définition de méta-modèles). Ce RFP, lancé également le 6 juin 1996, est terminé et a donné lieu à l'adoption du MOF comme standard de méta-modélisation le 19 novembre 1997.

- Le RFP **Stream-based Model Interchange Format (SMIF)** a été lancé suite à un besoin de standardisation en terme d'échange de modèles. Ce RFP, lancé le 5 décembre 1997, est terminé et a donné lieu à l'adoption de XMI comme standard d'échange de modèles et de méta-modèles le 23 mars 1999.

Six RFP sont également en cours dans ce groupe de travail :

- Le RFP **Common Warehouse Metadata Interchange (CWMI)** a été lancé le 18 septembre 1998 suite à un besoin de standardisation en terme de langage de modélisation de structure de bases de données (bases de données objets, relationnelle, hiérarchique, ou encore représentation de la partie données des programmes COBOL).

- Le RFP **Action Semantics for the UML** a été lancé le 18 septembre1998 suite à un besoin en terme de modélisation de la sémantique des opération UML. La solution proposée ne

devra pas dépendre d'une architecture ou d'une implémentation spécifique. Cette spécification pourra être utilisée par exemple pour effectuer des simulations et obtenir des preuves formelles sur le modèle en amont du cycle de développement logiciel. Les problèmes étant détectés plus tôt, leur correction entraînera moins d'impact sur le reste du développement. Un autre champ d'utilisation d'une telle spécification est la réutilisation de composants logiciels. En effet, une mise en correspondance de cette spécification vers différents langages d'implémentation pourra être effectuée de manière à implémenter plus facilement ces composants à l'aide de technologies diverses.

- Le RFP **UML Profile for EDOC** a été lancé le 26 mars 1999 suite à un besoin en terme de modélisation spécifique d'un système d'information conséquent basé sur des objets/composants métier distribués (EDOC est l'acronyme de Enterprise Distributing Object Computing).

- Le RFP **UML Profile for CORBA** a été lancé le 26 mars 1999 suite à un besoin en terme de modélisation spécifique d'un système d'information distribué basé sur CORBA. Ce profil doit prendre en compte toutes les spécificités des interfaces IDL.

- Le RFP **Human-usable Textual Notation for UML EDOC Profile** a été lancé le 26 mars 1999 et doit permettre de standardiser une notation textuelle claire pour exprimer les spécificités du profil UML EDOC. Il nécessite donc dans un premier temps la standardisation d'un tel profil et par conséquent l'aboutissement du RFP correspondant.

- Le RFP **UML Profile for Scheduling, Performance and Time** a été lancé le le 26 mars 1999. Etant donné que UML est un langage de modélisation de systèmes informatiques « généraliste », il ne permet pas directement la représentation et la prise en compte de la conception de systèmes temps réel pour lesquels la consommation de ressources et la maîtrise du temps d'exécution sont des point cruciaux..Ce RFP doit aboutir à un profile UML permettant une modélisation précise de la sémantique de tels systèmes. Ce profile doit permettre l'analyse quantitative de leur ordonnancement, de leurs performances et de leurs contraintes en terme d'exécution temps réel.

Sur ces six RFP, quatre nécessitent la définition de la notion de profile pour UML. Un profile est une extension du méta-modèle pour s'adapter à un besoin plus spécifique en terme de modélisation. Par conséquent, cette notion de profile, qui n'existe pas encore, devra très certainement être précisée au niveau de UML et sûrement définie au niveau du MOF.

Alors qu'à la création de ce groupe de travail, le plus important semblait être le RFP **OA&D Facility** avec l'adoption d'UML, c'est le MOF, issu du RFP **Meta Object Facility**, qui se retrouve actuellement au centre des préoccupations.

Le langage UML (Unified Modeling Language) est un langage d'expression de modèles à objets. Il est issu en grande partie de la notation utilisée dans OMT (Object Modeling Techniques). A la différence d'OMT, UML ne défini pas de processus de développement, il se contente de définir précisement une notation et sa sémantique pour la représentation et l'échange de modèles à objets. Il a été adopté par l'OMG (Object Management Group) comme standard de modélisation de systèmes informatiques en novembre 1997.

Ce langage définit les concepts familiers des langages de modélisation et de programmation à objets. Il était initialement représenté de façon réflexive. C'est à dire que la première version de la notation UML était décrite dans le formalisme UML. Le méta-modèle de ce formalisme était donc représenté en UML.

Depuis 1997, d'autres langages de modélisation ont vu le jour au sein de l'OMG. En effet, le formalisme UML, qui se voulait être au départ un formalisme universel de représentation de modèles s'est vite retrouvé cantonné à la modélisation de systèmes informatiques. De nouveaux besoins comme la modélisation de Workflow ou encore la modélisation de méta-modèles (la méta-modélisation) n'était pas suffisamment couverts par UML (dont ce n'était d'ailleurs pas le rôle).

L'OMG s'est alors inspiré des travaux effectués par l'EIA pour proposer une architecture similaire à celle présentée Figure 6 et y intégrer ces différents méta-modèles.

Un langage de méta-modélisation appelé MOF et issu du RFP **Meta Object Facility** a alors été standardisé par l'OMG en novembre 1997. Une fois cette standardisation effectuée, UML, qui était initialement exprimé dans son propre formalisme, est devenu, en théorie, un méta-modèle MOF - c'est à dire un méta-modèle exprimé avec le MOF. En théorie seulement car certains éléments de la version 1.3 du méta-modèle d'UML sont définis à partir de concepts non présents dans le MOF tels que les classes associatives. Un travail d'alignement du MOF et de UML est actuellement en cours.

Les concepts définis par le MOF étant très proches de ceux définis dans le cœur d'UML, la notation graphique d'UML a été retenue pour représenter également les méta-modèles MOF. Ainsi, l'apport principal d'UML dans le domaine de la méta-modélisation est sa notation graphique et ses concepts objets.

1.4 Contribution de la thèse.

Cette thèse CIFRE a permis de mettre en pratique tout un outillage logiciel de manipulation de modèles basés sur une architecture de modélisation et de méta-modélisation formelle. Cette architecture a ainsi pu être validée et nous a permis de mieux comprendre le formalisme MOF. Deux outils basés sur ce formalisme sont actuellement commercialisés par la société dans laquelle cette thèse CIFRE a été réalisée. Ces deux outils sont orientés systèmes d'information, tandis qu'un troisième, orienté modélisation de processus, est en cours de développement.

1.4.1 Applications industrielles.

Le premier outil est un outil de rétro-documentation Cobol. C'est un référentiel des applications Cobol présentes sur un site. Le formalisme nous permet alors d'aller jusqu'à la représentation des diagrammes de flot de données et de flot de contrôle. Une implémentation du langage de requête présenté chapitre 8 est intégrée à cet outil et permet par exemple de rechercher des données susceptibles de contenir des dates avec un siècle codé sur deux chiffres dans les programmes Cobol. Cette application a été réalisée en Smalltalk et se base sur la première implémentation des sNets.

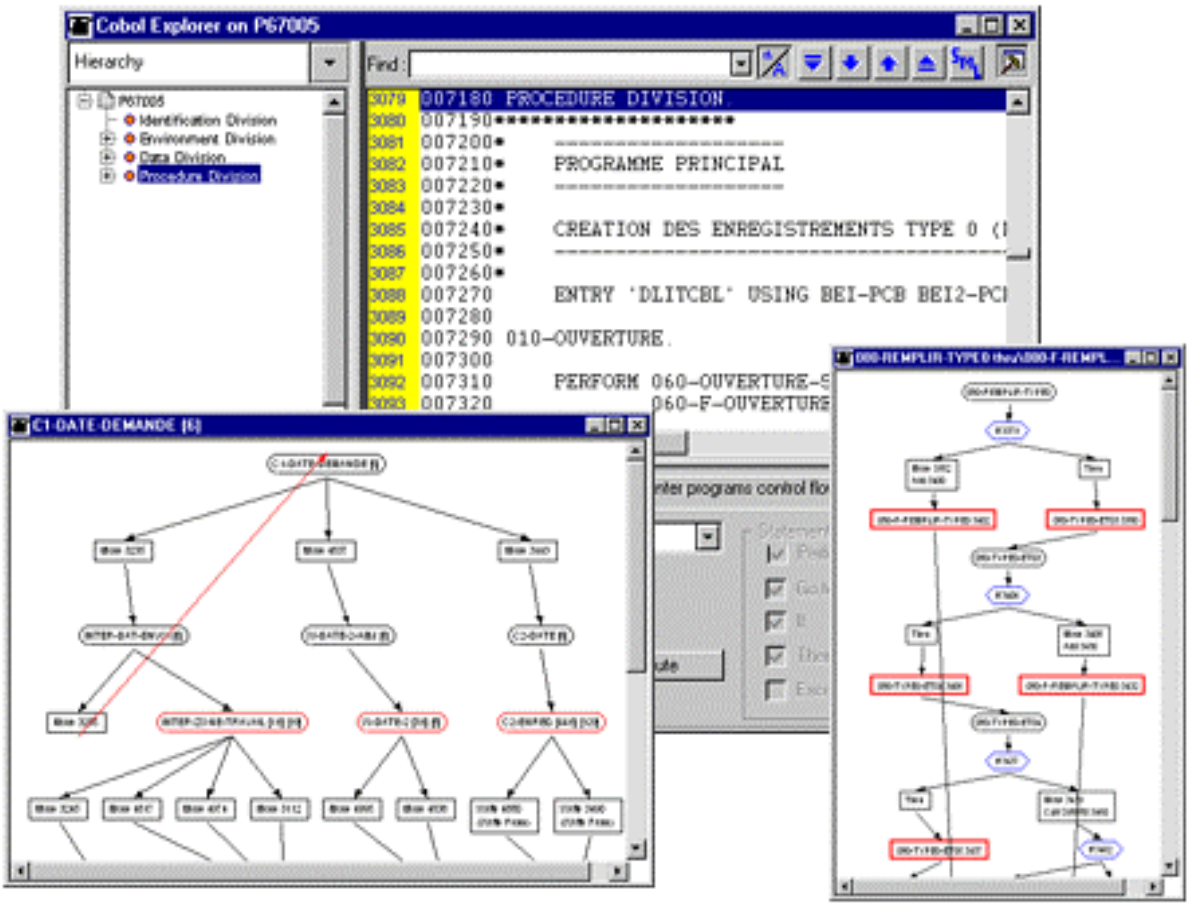


Figure 9 - Semantorã : Une application de rétro-documentation basée sur le formalisme des sNets.

Le second outil est un outil de génération de code à partir d'un modèle UML. Cet outil est en fait un outil de génération de documents textes indépendant d'un méta-modèle précis. La version commercialisée se base sur le méta-modèle de UML de manière à pouvoir être utilisée en aval d'outils de modélisation comme Rose© de la société Rational, Paradigm© de la société Platinum ou encore Mega Development© de la société Mega International. Le code source de cet outil est lui-même obtenu par génération a partir du méta-modèle de UML, il peut donc facilement être modifié pour se baser par exemple sur un méta-modèle de Workflow, un méta-modèle tel que le MOF, ou tout autre méta-modèle spécifique à un client donné. Cet outil est réalisé en Java, il se base sur la dernière implémentation du formalisme décrite en partie au chapitre 4.3.

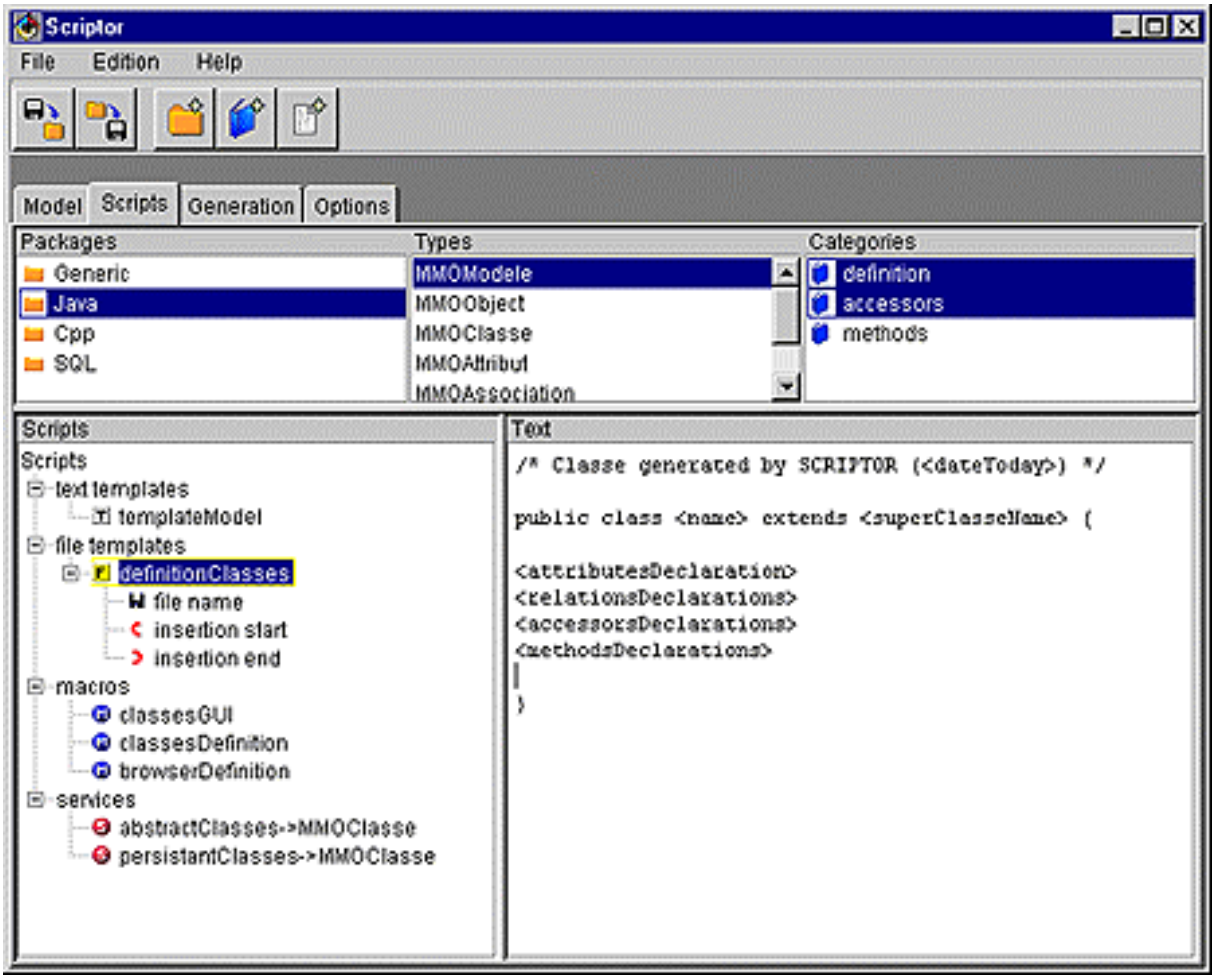


Figure 10 - Scriptorã : Un outil de génération de code basé sur le formalisme des sNets.

Un outil de modélisation de processus est également en cours de réalisation, il se base aussi sur le formalisme des sNets pour la représentation interne des modèles de processus. Comme l'outil de génération de code, cet outil est le plus indépendant possible du méta-modèle de sorte que celui-ci peut évoluer assez facilement sans impacts majeurs dans le code de l'application. Cet outil est réalisé en Java et se base sur la dernière implémentation du formalisme décrite en partie au chapitre 4.3.

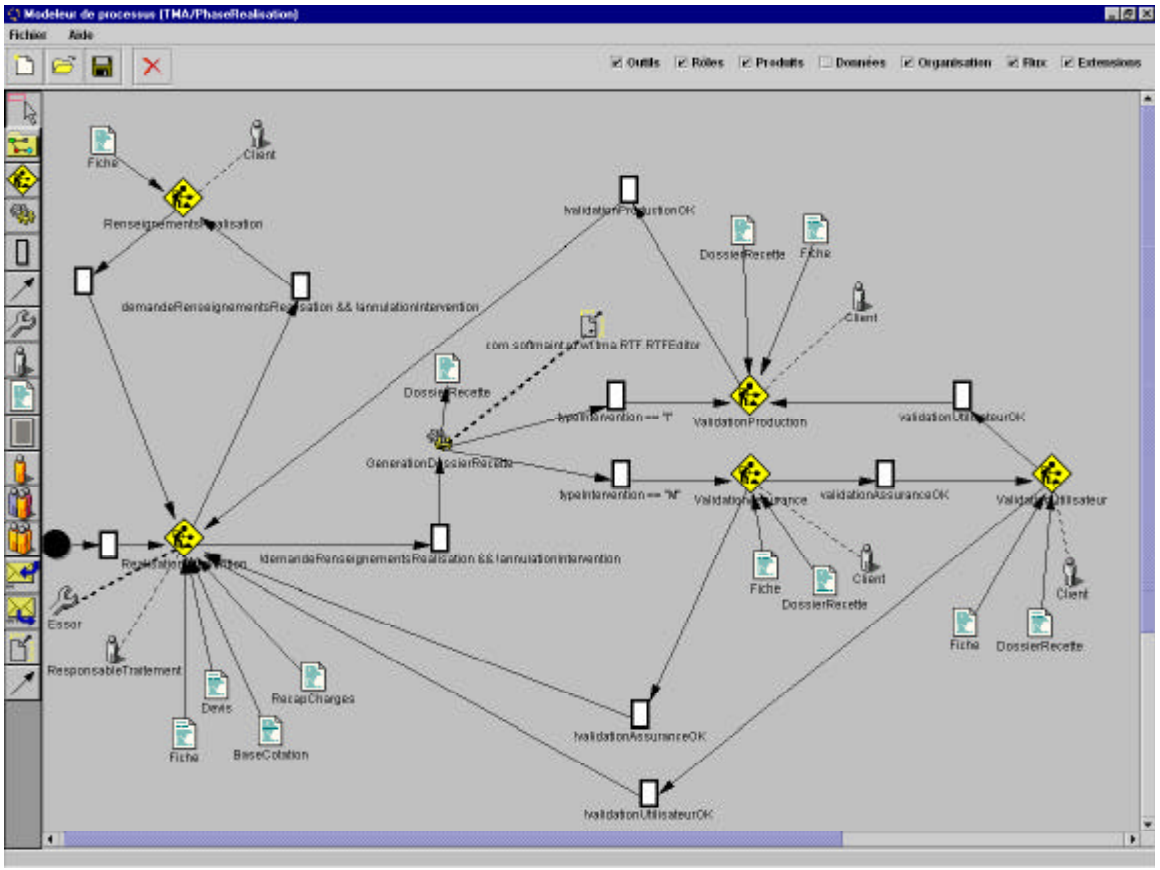


Figure 11 - Modeleur de processus basé sur le formalisme des sNets.

Nous avons également travaillé sur l'interopérabilité des outils par exportation de modèles, transformation selon des règles basées sur les méta-modèles de la source et de la cible, puis importation dans l'outil cible. Ces travaux ont été effectués avec le moteur de transformation réalisé dans le cadre de la thèse et présenté au chapitre 7. Nous avons ainsi été en mesure de réaliser très rapidement un outil d'exportation d'un modèle d'ingénierie de système. Le modèle exporté est ensuite représenté à l'aide de notre formalisme puis transformé à l'aide de règles exprimées sur les méta-modèles en un modèle UML au format sNets. Ce modèle est ensuite exporté au format XMI de manière a être intégrable directement dans un outil de modélisation UML. L'ensemble des composants logiciels utilisés lors de ce processus sont indépendants des méta-modèles des modèles manipulés. Ainsi, les composants d'importation et d'exportation au format XMI, le composant de transformation de modèles, les composants représentant l'implémentation des modèles et de leurs méta-modèles sont complètement génériques et peuvent être réutilisés dans d'autres cadres d'applications.Il existe actuellement une très forte demande

dans le domaine de l'inter-opérabilité et de la continuité sémantique entre les différents outils du marché.

De plus, un atelier de modélisation et de méta-modélisation a été réalisé et permet de prototyper et de valider rapidement des méta-modèles ainsi que leurs modèles. Cet outil permet d'appliquer sur ces modèles les outils précités (i.e. outils de transformation de modèle, exportation et importation), ainsi que des outils génériques de construction tels qu'un grapheur générique et un navigateur générique. Cet outil est réalisé en Java et se base sur la dernière implémentation du formalisme décrite en partie au chapitre 4.3.

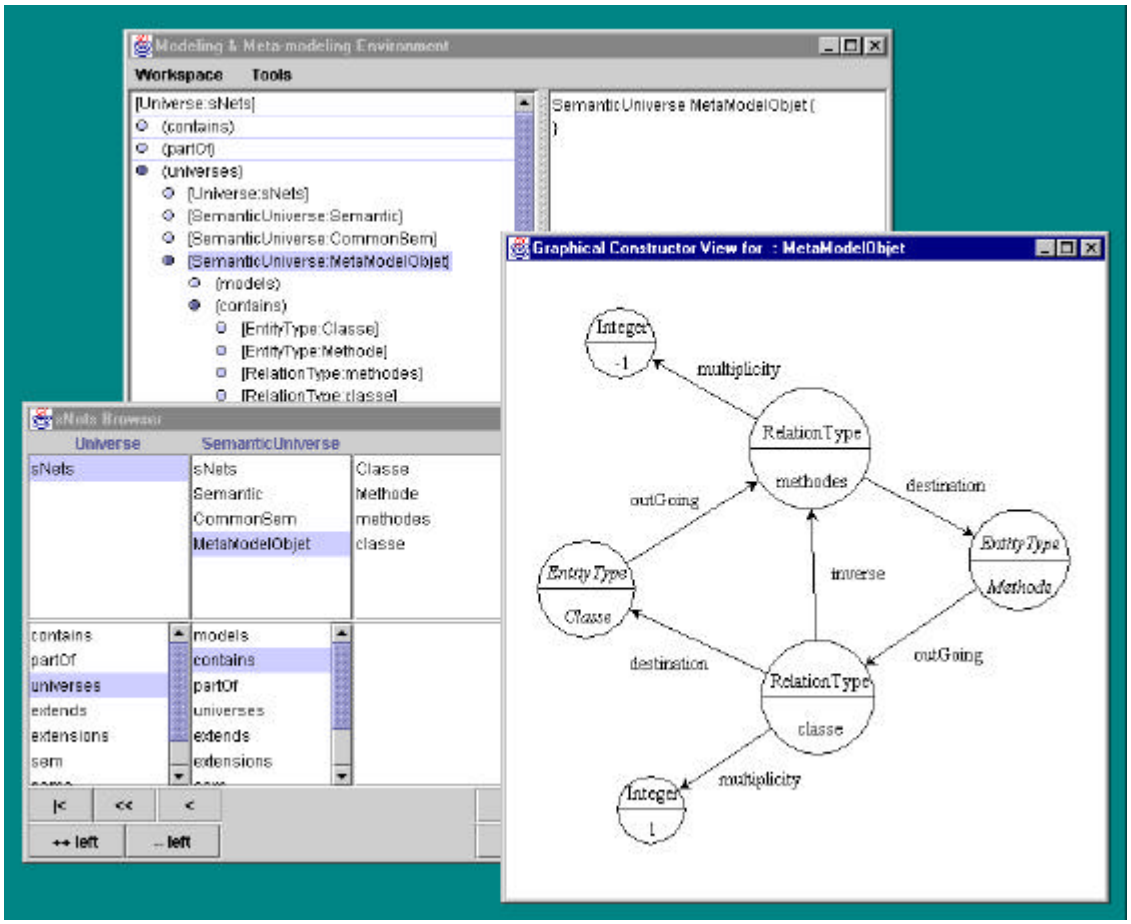


Figure 12 - Un atelier de modélisation et de méta-modélisation basé sur le formalisme des sNets.

Cette thèse présente donc principalement le formalisme des sNets ainsi que l'exploitation de ce formalisme dans le cadre de l'ingénierie et de la rétro-ingénierie logicielle. De plus, notre formalisme constitue un bonne plate-forme de recherche en techniques de modélisation et de modélisation. Il est actuellement utilisé pour représenter des modèles UML, des ensemble de programmes COBOL, ainsi que de modèles de workflow et nous montrons également comment de tels outils peuvent être réalisés en se basant, non plus sur notre formalisme, mais sur des standards tels que le MOF et UML.

1.4.2 Contribution à l'ingénierie des modèles.

Notre travail s'inscrit dans un domaine de recherche en pleine évolution. Dans les années 1980, on a pu assister à la transition de la technologie procédurale à celle des objets puis des composants. Actuellement on s'oriente vers une nouvelle époque où les modèles et les méta-modèles sont traités comme des entités de première classe, tout comme les objets et les classes étaient traitées dans les années 1980.

Les diverses expérimentations et mises en œuvre de produits industriels réalisés dans le cadre de cette thèse nous ont permis de valider et de préciser cette idée centrale.

Un modèle est une entité manipulable avec des propriétés précises. Un modèle peut-être nommé, étendu, transporté, transformé, etc… Un modèle est fermé en ce sens qu'il a une sémantique précise qui sera elle-même définie dans un autre modèle (son méta-modèle).

Nous attachons beaucoup d'importance à cette relation qui existe entre un modèle et son méta-modèle. Cette relation est appelée sem dans notre formalisme. C'est également l'une des principales différences entre le formalisme des sNets et le MOF ou CDIF. En effet, ces formalismes proposent un langage de définition de méta-modèles et un langage de définition de modèles. Mais ces éléments sont distincts de sorte qu'il n'est pas possible de lier explicitement les modèles à leurs méta-modèles. Lorsque l'on définie des méta-modèles, la notion de modèle n'existe pas et lorsque l'on définie des modèles, la notion de méta-modèle n'existe pas. Dans notre formalisme, le concept de modèle (ou univers) et celui de méta-modèle (ou univers sémantique) sont indissociable. Un méta-modèle n'est alors qu'un modèle dont le méta-modèle est notre méta-méta-modèle. De même, ce méta-méta-modèle est un modèle réflexif.

De cette façon, il suffit de définir la notion d'extension de modèle pour autoriser de la même façon l'extension de méta-modèle. On peut alors décrire des frameworks de modélisation complexes comme celui présenté Figure 13.

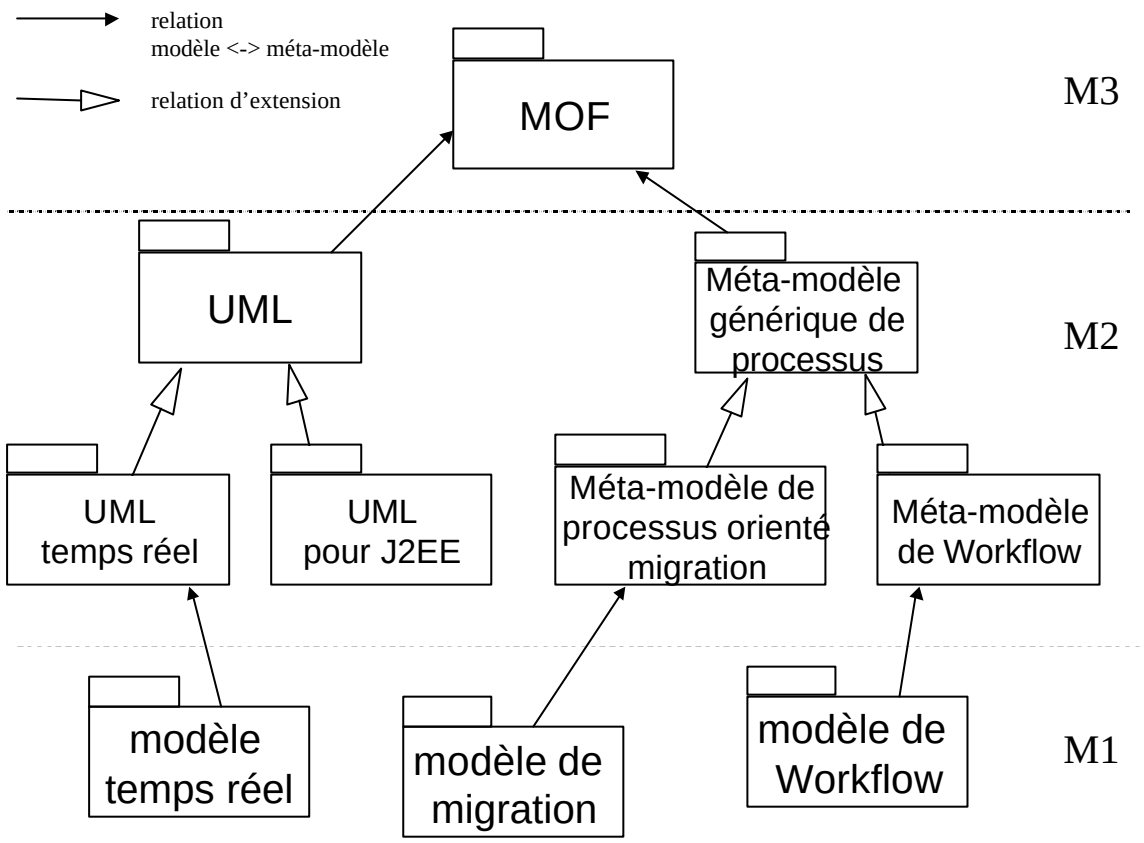


Figure 13 - Un cadre de définition de modèles et de méta-modèles.

Sur cette figure, une relation d'extension est utilisée pour étendre les méta-modèles de manière à répondre à un besoin plus spécifique. Ainsi, on peut définir un méta-modèle "UML temps réel" qui soit une spécialisation du méta-modèle UML et qui intègre tout un ensemble de concepts spécifiques liées à la modélisation temps réel. Cette notion de spécialisation est à rapprocher à la notion de profile définie dans UML 1.3. La notion de profile se base sur les mécanismes standard d'extension de UML (dont on retrouve certains éléments dans le MOF) que sont les stéréotypes (un stéréotype est une sous-classe d'un élément du méta-modèle de UML ayant la même forme que celui-ci mais utilisé dans un but plus spécifique), les étiquettes (que l'on retrouve également dans le MOF - c.f. chapitre 3.1.1.15) et les contraintes (que l'on retrouve également dans le MOF - c.f. chapitre 3.1.1.14). Un profile est alors un ensemble prédéfini de stéréotypes, d'étiquettes et de contraintes dédié à la modélisation d'un domaine spécifique. Les profiles peuvent alors être

comparé à un méta-modèle contenant essentiellement les éléments précités et étendant le méta-modèle sur lequel il s'applique. L'avantage de l'utilisation d'un méta-modèle spécifique par rapport à l'utilisation d'un profile est qu'un méta-modèle se représente facilement à l'aide du MOF alors qu'aucun format de représentation des profiles n'est encore définie.

Cette notion de profile permet de mettre en avant la nécessité de définir le concept de modèle. Le chapitre 5 montre ainsi les éléments essentiels de notre formalisme que sont notamment ce concept de modèle et la relation entre un modèle et son méta-modèle. Nous proposons ainsi un méta-méta-modèle, mis en œuvre dans notre formalisme, contenant l'ensemble des éléments qui nous ont semblé indispensables à la méta-modélisation (c.f Figure 14). Nous montrons ici l'apport de tels concepts et la façon de les intégrer dans une nouvelle version du MOF.

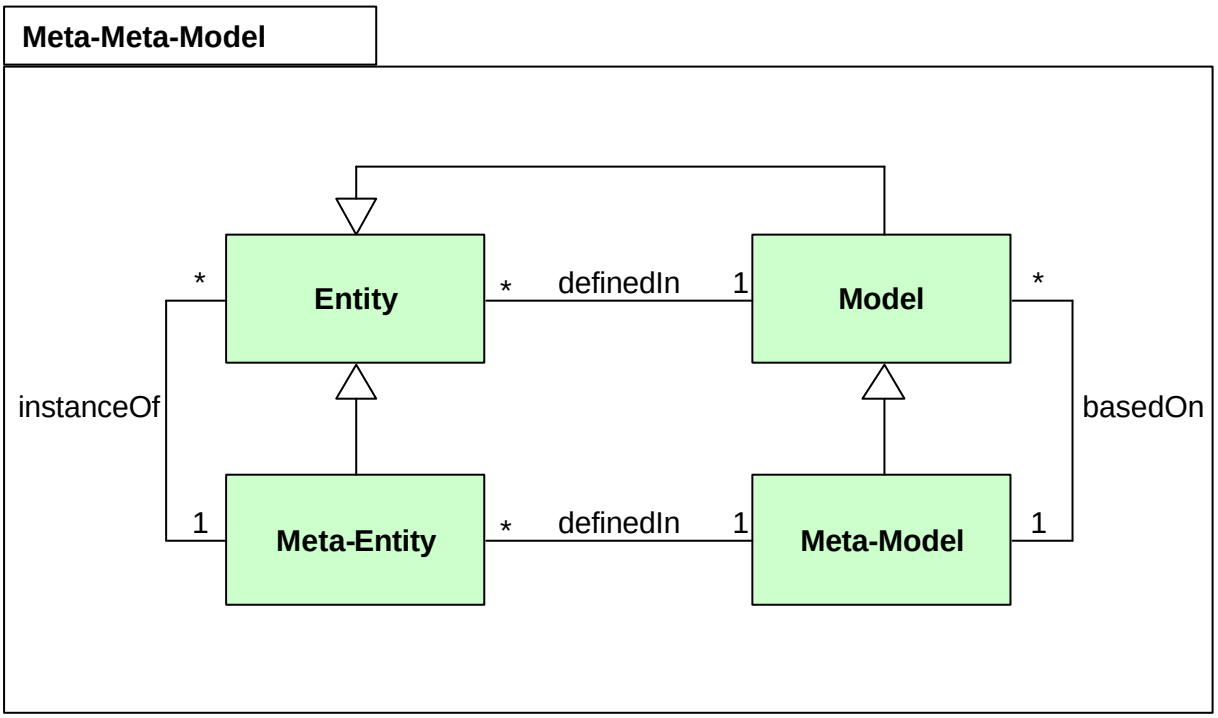


Figure 14 - Notre proposition de méta-méta-modèle.

Avoir défini tous ces concepts et toutes ces relations de façon explicite dans notre formalisme nous a également permis de mieux comprendre ces même éléments implicites dans le MOF et dans CDIF. Comme le présente le chapitre 6 traitant du contenu et de la séparation des différents niveaux de modélisation, beaucoup de confusion est issue de la "définition" implicite de certains concepts et de certaines relations. Ces éléments sont pourtant utilisées abondamment dans le

domaine de la méta-modélisation et le fait de les explicites nous a ainsi permis de mieux les appréhender.

2 L'ingénierie des modèles : état de l'art.

Un modèle est utilisé pour décrire de l'information à l'aide d'une sémantique bien déterminée :

- un modèle à objets décrit de l'information en termes de classes et d'associations,
- un modèle relationnel décrit l'information en termes de tables, de colonnes, de relations,
- un modèle Cobol décrit un programme Cobol en termes d'instructions, de données, etc…

Lorsque l'on s'intéresse à la modélisation en général, l'information modélisée peut être de n'importe quelle nature. En intelligence artificielle par exemple, l'information est constituée d'une base de connaissances dont la sémantique peut être très riche. La diversité des informations entraîne alors une structuration difficile de celles-ci. A l'inverse, dans un contexte industriel ou l'une des préoccupations principales est l'échange d'informations entre outils, les informations doivent être structurées. Lorsque l'on s'intéresse à l'ingénierie des modèles, ces deux aspects doivent être pris en compte. Les modèles manipulés vont être de nature très diverse, mais leur contenu doit être structuré de sorte que des outils différents puissent s'échanger des modèles, les manipuler, les transformer, etc...

2.1 Le domaine de la représentation de connaissances.

Les deux formalismes présentés ici sont le formalisme des réseaux sémantiques, sur lequel s'appuie notre propre formalisme des sNets présenté dans cette thèse, et le formalisme des graphes conceptuels.

2.1.1 Les réseaux sémantiques.

L'utilisation de graphes en représentation de connaissances vient de l'idée de représenter graphiquement des concepts et leurs liens. Le premier formalisme proposé est le formalisme de réseaux sémantiques introduit en 1968 par Quillian. Ce formalisme avait pour but la reconstitution d'un modèle de la mémoire humaine.

Le réseau sémantique est un outil qui simule notre représentation de la mémoire. C'est un modèle qui montre comment :

- l'information pourrait être représentée en mémoire et
- comment on pourrait accéder à ces informations.

En effet, notre mémoire est représentée comme un bassin de données contenant des concepts, des événements, des sensations, ... Tous ces éléments forment un immense réseau en interrelation. Couché sur le papier, un réseau sémantique est composé de *nœuds* dont les interrelations sont établies par des *pointeurs* étiquetés. Les nœuds sont les différents types d'information en mémoire. A ces nœuds peuvent être associées des propositions, énoncés qui caractérisent les propriétés s'appliquant au nœuds du réseau. L'étiquette associée au pointeur indique quel est le type de relation entre deux nœuds.

Un exemple de réseau sémantique est présenté Figure 15. Ce réseau sémantique correspond à l'énoncé suivant :

" Pitou est petit (pour un chien) et Gazou est une femelle."

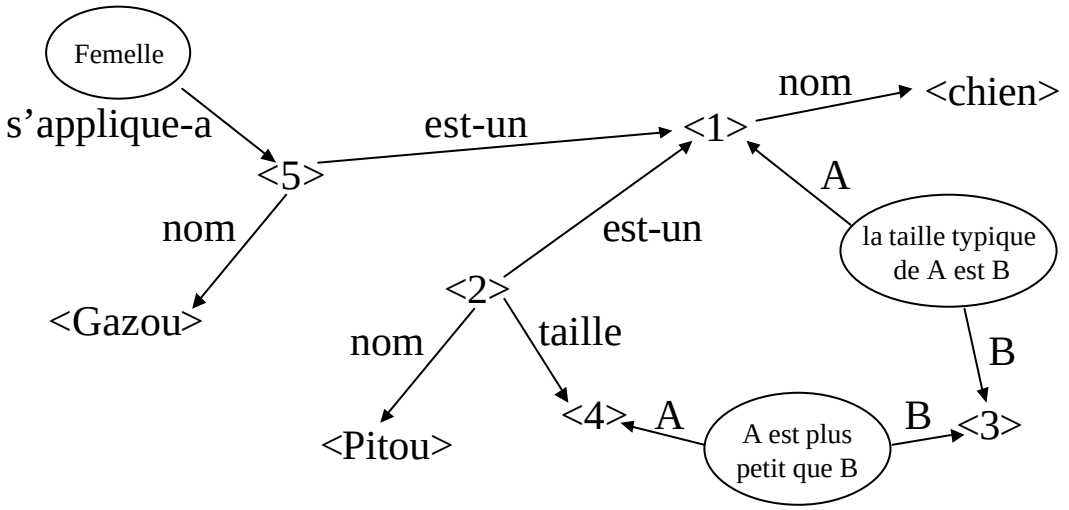


Figure 15 - Un exemple de réseau sémantique représentant l'énoncé "Pitou est petit pour un chien et Gazou est une femelle".

Dans cet exemple, les nœuds sont représentés par des symboles différents pour signifier les différents types d'information. Les termes entre crochets représentent ici les entités ou concepts tandis que les cercles représentent les propositions associées à ces concepts. L'avantage de ce

formalisme est qu'il permet la représentation de tout type d'information tandis que son inconvénient majeur est l'absence de modularité et de structuration de la sémantique de ces informations.

En ce qui nous concerne, les informations représentées sont des programmes COBOL ou des modèles à objets et les Figure 3 et Figure 4 ont déjà montré que ce formalisme se prêtait bien à la représentation de ces informations.

2.1.2 Les graphes conceptuels.

En 1984, Sowa propose le formalisme des Graphes Conceptuels aujourd'hui largement utilisés en Intelligence Artificielle. Sowa prétend que toute forme de représentation pourrait être écrite sous forme de graphe conceptuel. Dans ce formalisme, le sens d'un élément est déduit de sa position relative par rapport aux autres éléments, il ne prend donc un sens que par rapport à un réseau sémantique modélisant les connaissances générales du système.

Les graphes conceptuels sont un système de logique basé sur les graphes existentiels de Peirce et les réseaux sémantiques. Le but du système est de fournir un formalisme d'expression de la sémantique logiquement précis, facilement lisible par les êtres humains et suffisamment simple pour être utilisé par des systèmes informatiques.

Un graphe conceptuel est un graphe dont les nœuds sont de deux sortes. Certains nœuds déffinissent les concepts et se représentent par des rectangles tandis que les autres définissent des relations entre ces concepts et se représentent par des cercles. Une relation est reliée à au moins un concept tandis qu'un concept peut être isolé. Le formalisme des graphes conceptuels est un formalisme typé. Un concept est ainsi constitué d'un type de concept et d'un référent.

Un exemple de graphe conceptuel est présenté Figure 16. Ce graphe conceptuel correspond à l'énoncé suivant :

"Jean pense que Marie veut épouser un marin."

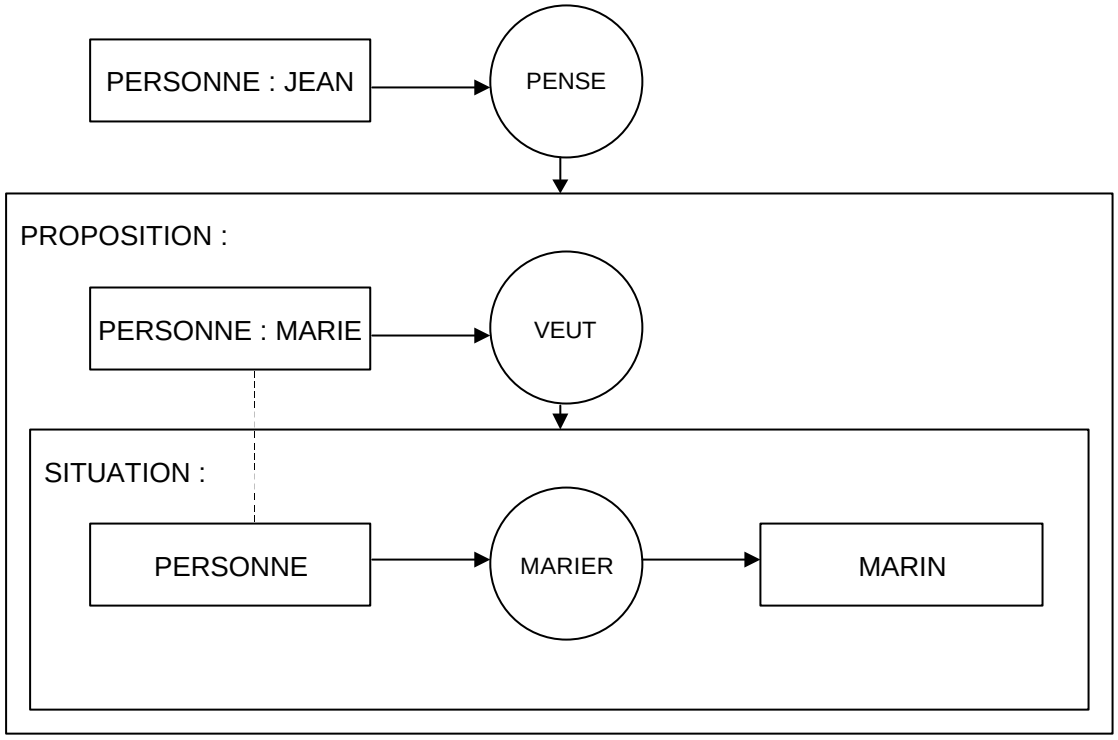


Figure 16 - Un exemple de graphe conceptuel.

Le lien en pointillés est un lien de co-référence. Il permet d'indiquer que la personne référencée dans la Situation et Marie sont une seule et même entité.

Une notation texte appelée forme linéaire est également définie pour les graphes conceptuels et la Figure 16 peut alors se noter de la façon suivante :

```
[PERSONNE: JEAN] -> (PENSE) -> [PROPOSITION:
        [PERSONNE: MARIE?x] -> (VEUT) -> [SITUATION:
                [PERSONNE : *x] -> (MARIER) -> [MARIN] ] ].
```

Le lien de coréférence est ici représenté par une variable x qui sera liée à MARIE.

Ici, nous avons des concepts de quatre types : PROPOSITION, PERSONNE, SITUATION et MARIN. Ces types peuvent être catégorisés. Pour cela, il faut indiquer les relations qu'un concept de ce type entretien avec d'autres concepts.

Ainsi, on pourra dire qu'un marin est une personne qui navigue sur un bateau en définissant le type MARIN de la façon suivante :

```
Type MARIN(x) is
        [PERSONNE: *x] -> (navigue sur) -> [BATEAU]
```

De la même façon, les relations sont typées dans le formalisme des graphes conceptuels. Ainsi, avant de pouvoir utiliser la relation "navigue", il faudra la définir dans ce formalisme. Un type de relation est presque définie de la même façon qu'un type de concept. On utilise les types de relation "AGNT" (pour agent) et "OBJ" (pour objet) pour définir un nouveau type de relation :

```
Relation navigue_sur(x,y) is
        [PERSONNE: *x] <- (AGNT) - [Naviguer sur] - (OBJ) -> [BATEAU: *y]
```

Cette notation indique que la relation "navigue sur" est définie par une expression qui relie une personne identifiée par *x et jouant le rôle d'agent pour l'action "Naviguer sur" qui a pour objet un bateau identifié par *y.

Au sommet de la hiérarchie des types, il existe un type appelé type universel et identifié par le symbole T. A la différence des réseaux sémantiques, ces types nous permettent ainsi de définir formellement la sémantique de nos modèles. Ainsi, la sémantique d'un modèle décrivant les programmes présents sur un site (cette sémantique a déjà été présentée Figure 3) pourra être définie de la façon suivante en utilisant le formalisme des graphes conceptuels :

```
Type SITE(x) is
        [T:*x] -> (est composé du programme) -> [PROGRAMME]

Type PROGRAMME(x) is
        [T:*x] -
                (est composé de l'instruction) -> [INSTRUCTION]
                (est composé de la donnée) -> [DONNEE]

Type INSTRUCTION(x) is
        [T:*x] -
                (utilise la donnee) -> [DONNEE]
                (est suivie de l'instruction) -> [INSTRUCTION]
```

Type DONNEE(x) is
    [T:*x]

Les relations doivent alors être définies de la façon suivante:

Relation est_composé_du_programme(x,y) is
    [SITE:*x] <-(AGNT)- [a pour programme] -(OBJ)-> [PROGRAMME: *y]

Relation est_composé_de_l_instruction(x,y) is
    [PROGRAMME:*x] <-(AGNT) - [a pour instruction] -(OBJ)-> [INSTRUCTION: *y]

Relation est_composé_de_la_donnée(x,y) is
    [PROGRAMME:*x] <-(AGNT) - [a pour donnée] -(OBJ)-> [DONNEE: *y]

Relation utilise_la_donnée(x,y) is
    [INSTRUCTION:*x] <-(AGNT) - [utilise] -(OBJ)-> [DONNEE: *y]

Relation est_suivie_de_l_instruction(x,y) is
    [INSTRUCTION:*x] <-(AGNT) - [est_suivie_de] -(OBJ)-> [INSTRUCTION: *y]

Un site contient des programmes, un programme se décompose en instructions et en données, une instruction utilise des données et une instruction est suivie d'une autre instruction.

Ces types et ces relations peuvent alors être utilisés pour représenter les informations présentes sur un site. Ainsi, les informations de la Figure 1 pourront être représentées en utilisant le formalisme des graphes conceptuels de la façon suivante :

[SITE:s] -
    (est_composé_du_programme) -> [PROGRAMME: p1]
    (est_composé_du_programme) -> [PROGRAMME: p2?x1]
    (est_composé_du_programme) -> [PROGRAMME: p3]

[PROGRAMME: *x1] -
    (est_composé_de_l_instruction) -> [INSTRUCTION: i1?x2]
    (est_composé_de_l_instruction) -> [INSTRUCTION: i2?x3]

(est_composé_de_la_donnée) -> [DONNEE: d1?x4]
(est_composé_de_la_donnée) -> [DONNEE: d2?x5]

[INSTRUCTION: *x2] -
(utilise_la_donnée) -> [DONNEE: *x4]
(est_suivie_de_l_instruction) -> [INSTRUCTION: *x3]

[INSTRUCTION: *x3] -
(utilise_la_donnée) -> [DONNEE: *x4]
(utilise_la_donnée) -> [DONNEE: *x5]

Ce formalisme nous permet donc bien d'exprimer à la fois nos modèles et leur sémantique (les méta-modèles). La difficulté de ce formalisme est qu'il est surdimensionné par rapport à notre besoin. En effet, dans ce formalisme, nous avons vu qu'un concept était identifié par son type et son référent. Maintenant, le type peut-être soit un label (ici INSTRUCTION, PROGRAMME, etc…), soit la lambda expression définissant ce type. On peut ainsi substituer les occurrences du label INSTRUCTION par l'expression suivante :

[T: λ] -
(utilise_la_donnée) -> [DONNEE]
(est_suivie_de_l_instruction) -> [INSTRUCTION]

De plus, le référent lui même peut-être constitué d'un graphe conceptuel. Ce graphe est alors considéré comme décrivant le référent. Le référent peut également être constitué d'un simple quantificateur (le quantificateur existentiel identifié par "∃" ou le quantificateur universel identifié par "∀").

En ce qui concerne la représentation de modèles et de méta-modèles, ces fonctionnalités ne sont pas essentielles. Elle apporte une puissance dans le domaine de l'intelligence artificielle car des opérations de base telles que la copie (copie de graphe conceptuelle), la restriction (remplacement dans un graphe d'un type par un sous-type de celui-ci), la jointure (jointure de deux graphes disposant d'un concept commun) et la simplification (suppression des relations dupliquées dans le graphe) peuvent être utilisées et composées, mais compliquent la manipulation de graphes conceptuels pour une simple représentation de modèles et de méta-modèles.

2.2 Le domaine industriel.

Ce chapitre présente différents formalismes utilisés dans le domaine industriel pour la représention et la manipulation de modèles. Les formalismes étudiés sont CDIF dont nous avons déjà dit quelques mots dans l'introduction, XML qui est un support textuel de représentation structurée d'information de plus en plus utilisé, le MOF dédié à la représentation de méta-modèles et XMI qui est le support textuel de représentation des modèles MOF.

2.2.1 Le formalisme CDIF.

L'architecture sur laquelle se base ce formalisme a déjà été présenté dans l'introduction. Le but de ce formalisme est l'échange d'informations entre outils différents. En effet, de plus en plus d'outils sont utilisés au cours du développement d'un logiciel. Des outils de modélisation, des outils d'analyse et de conception, des outils de documentation, de outils de gestion de projet ou encore des outils de programmation sont couramment utilisés sur le même projet. La difficulté est alors de permettre l'échange d'informations entre ces différents outils, et surtout de conserver ainsi la cohérence des données qu'ils représentent. Nous allons détailler ici les différents éléments qui le composent à savoir son méta-méta-modèle, l'ensemble de ses méta-modèles appelé "CDIF Integrated Meta-model" et le format syntaxique d'échange des modèles.

2.2.1.1 Le méta-méta-modèle de CDIF.

Les méta-modèles de CDIF sont exprimés à l'aide du méta-méta-modèle présenté sur la figure suivante (Figure 17) :

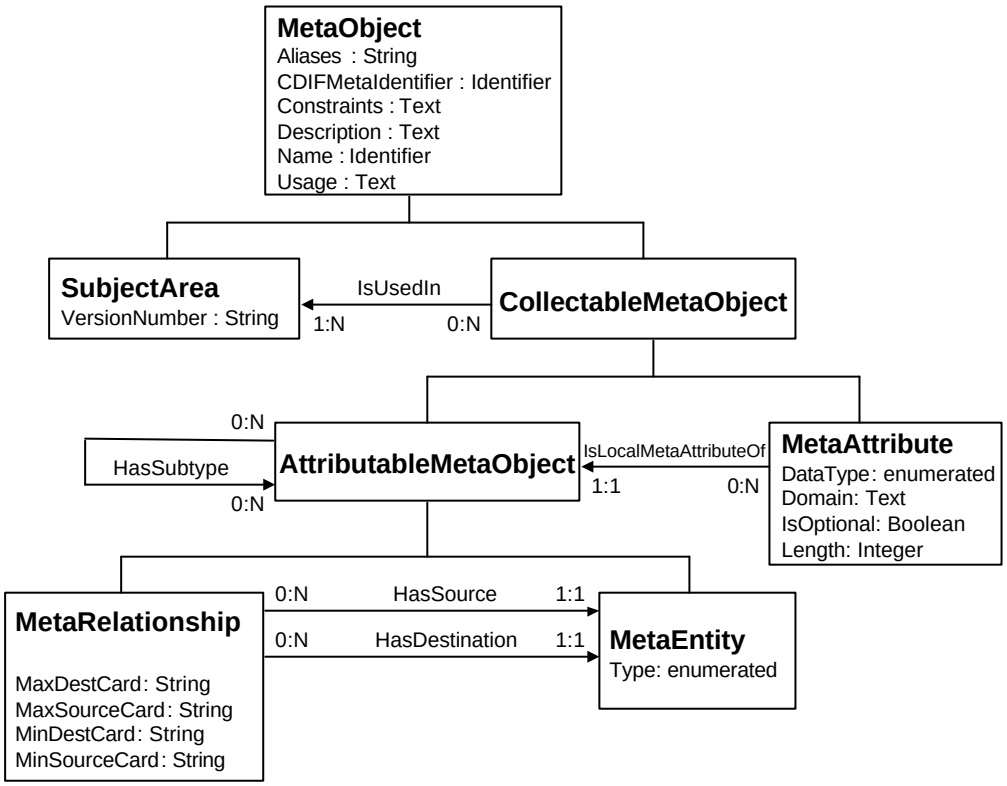


Figure 17 - Méta-Méta-modèle de CDIF.

Les types des entités d'un méta-modèle sont définis par des entités de type MetaEntity et les types des relations sont définis par des entités de type MetaRelationship. Ces entités peuvent disposer d'attributs représentés par des entités de type MetaAttribute. De plus l'héritage multiple est définit pour les méta-entités **et** les méta-relations. C'est l'un des rares formalismes qui propose de l'héritage pour les relations.

Ces ensembles de méta-relations, méta-entités et méta-attributs forment des méta-modèles qui sont organisés ici en "zones d'intérêt" appelées SubjectArea. Un méta-modèle sera donc représenté avec CDIF par une entité de type SubjectArea. Il est a noter qu'une méta-entité peut appartenir à plusieurs SubjectArea.

Nous retrouvons dans ce méta-modèle les points forts que nous avons mis en avant dans le formalisme des sNets. En effet, ce méta-modèle est minimal et réflexif. Il est minimal car il est composé simplement de 7 entités qui permettent tout de même la représentation de tous types de méta-modèles. Et il est réflexif car les rectangles représentés sur cette figure sont des entités de type MetaEntity et les relations entre ces entités sont des entités de type MetaRelationship. De plus, les cardinalités des relations sont portées par les entités de type MetaRelationship sous la forme de méta-attributs.

Ce méta-modèle nous a conforté dans les choix que nous avions fait pour notre formalisme. Les définition de ces deux formalismes (CDIF et le formalisme des sNets) ayant été menés de façon simultanée et indépendante, il est plaisant de constater que leurs méta-méta-modèles sont assez similaires. Le chapitre 5 traitant de la réflexivité dans les méta-méta-modèles montre tout de même l'absence ici de méta-relations et de méta-entités qui nous semblent essentielles.

2.2.1.2 Un méta-modèle intégré.

Nous avons vu que pour permettre l'échange d'informations entre outils, il fallait proposer des méta-modèles standards pour la représentation de ces informations. Un certain nombre de SubjectArea ont alors été proposées pour permettre de représenter ces différents types d'information. Le problème est que CDIF n'a pas été suffisamment suivi par les industriels, et seuls les modèles de données et les modèles de flots de données ont donné lieu à des SubjectArea standardisées. Une SubjectArea a également été proposée pour la représentation de modèles d'analyse et de conception orientés objet, mais elle n'a malheureusement jamais été standardisé. En effet, cette proposition est arrivée en même temps que la standardisation de UML et l'intérêt s'est alors détaché de CDIF pour ce tourner vers ce nouveau langage destiné à normaliser ce type de modélisation.

2.2.1.3 Un format syntaxique standard d'échange de modèles.

Un fichier d'échange de modèles CDIF se décompose en deux parties. La première décrit le méta-modèle dans les termes du méta-méta-modèle CDIF (c.f Figure 17) et la seconde décrit le modèle dans les termes du méta-modèle. Un exemple de contenu de fichier CDIF est présenté ci-dessous :

```
CDIF, SYNTAX "SYNTAX.1" "02.00.00", ENCODING "ENCODING.1" "02.00.00"
(:HEADER
        (:SUMMARY
                (ExporterName       "myTool")
                (ExportVersion      "0.1")
        )
)
(:META-MODEL
        …
        (:SUBJECTAREAREFERENCE DataFlowModel
                (:VERSIONNUMBER         "01.00")
        )
        …
        (MetaEntity ME001
                (Name                   "SecurityClassification")
        )
        …
)
(:MODEL
        …
        (SecurityClassification …
                …
        )
)
```

Le méta-modèle contient généralement une référence à l'un de méta-modèles standards de CDIF (ici DataFlowModel) ainsi qu'un ensemble d'extensions spécifiques à l'outil source de ce

modèle (représenté ici par la définition de la méta-entité SecurityClassification). De cette façon, un outil destination sera capable de relire ce modèle en se basant sur le méta-modèle standard référencé et sera capable de ne pas tenir compte des extensions spécifiques à l'outil source.

Comme indiqué dans l'introduction, le peu de méta-modèles standardisés a poussé les développeurs à redéfinir entièrement leurs propres méta-modèles en ne se basant sur aucun méta-modèle standard. En conséquence, le formalisme CDIF était bien respecté, mais les informations ainsi exportées ne pouvaient être importées dans un outil différent.

2.2.2 Le formalisme XML.

Le formalisme XML (Extended Markup Language) permet la représentation structurée d'informations dans un format texte. Il peut par conséquent être utilisé comme format syntaxique de transport de modèles et de méta-modèles. La structure de l'information est alors définie dans un fichier annexe au format DTD (Document Type Description). L'avantage d'un tel formalisme est de permettre l'échange d'informations dès lors que l'on base celle-ci sur une DTD normalisée.

Voici un exemple de document XML représentant un livre et sa table des matières :

```
<?xml version="1.0" ?>
<!DOCTYPE Livre "Livre.dtd">
<Livre Auteur="Richard Lemesle">
  <Titre>Thèse</ Titre >
  <Chapitre id="1">
     Premier Chapitre. Introduction.
  </Chapitre >
  < Chapitre id="2">
     Second Chapitre. Etat de l'Art.
 </Chapitre >
</Livre>
```

Dans ce formalisme, nous trouvons la notion d'élément. Les éléments sont définis avec un marqueur de début et un marqueur de fin. Le marqueur de début constitué simplement du nom de l'élément tandis que le marqueur de fin est constitué du nom de l'élément préfixé du caractère "/".

Dans l'exemple précédent, nous avons les éléments Livre représentant les livres, Titre représentant le titre des livres et Chapitre représentant l'intitulé d'un chapitre.
Ces éléments peuvent ensuite disposer d'attributs. Les attributs correspondent généralement au niveau le plus fin de décomposition des éléments. Dans l'exemple précédent, nous avons l'attribut Auteur défini sur l'élément Livre et représentant le nom de l'auteur du livre et l'attribut id sur l'élément Chapitre indiquant le numéro du chapitre.

Le rôle du fichier au format DTD est de décrire tous les éléments que l'on est susceptible de rencontrer dans le fichier XML ainsi que tous les attributs que ces éléments sont susceptibles de disposer.

Ainsi, la DTD nommée livre.dtd qui a permis de construire le fichier XML précédent est la suivante :

```
<!DOCTYPE Livre [
    <!ELEMENT Livre (Titre, Chapitre+)>
    <!ATTLIST Livre Auteur CDATA #REQUIRED>
    <!ELEMENT Titre (#PCDATA)>
    <!ELEMENT Chapitre (#PCDATA)>
    <!ATTLIST Chapitre id ID #REQUIRED>
]>
```

Cette DTD décrit les trois éléments (Livre, Titre et Chapitre) ainsi que leurs attributs. Actuellement, l'utilisation de DTD pour décrire le contenu des fichiers XML est en train de disparaître au profit des schémas. Les schémas sont des fichiers au format XML (ce qui n'était pas le cas des fichiers DTD) et proposent un certain nombre de facilités pour la définition de la structure des fichiers XML.

L'un des avantages de XML est sa souplesse et sa lisibilité tandis que ses désavantages sont l'aspect "verbeux" du langage entraînant rapidement des fichiers d'une taille importante et le fait que les données représentées dans un fichier XML le sont sous la forme d'un arbre. En effet, dès que l'on souhaite représenter des graphes dans un document XML, il est nécessaire de définir des références qui vont alors à l'encontre de la souplesse et la lisibilité. Or les modèles et les méta-modèles sont plus généralement constitués de graphes que d'arbres.

Ainsi, si l'on prend le cas de la représentation des programmes présents sur un site. Le méta-modèle de ces informations pourra être exprimé à l'aide d'une DTD de la façon suivante :

```
<!DOCTYPE Site [
    <!ELEMENT Site ( Programme+ )>
    <!ATTLIST Site nom CDATA #REQUIRED>
    <!ELEMENT Programme ( Donnee+, Instruction+)>
    <!ATTLIST Programme nom CDATA #REQUIRED>
    <!ELEMENT Donnee EMPTY>
    <!ATTLIST Donnee id ID #REQUIRED, nom CDATA #Required >
    <!ELEMENT Instruction ( DonneeUtilisee+)>
    <!ATTLIST Instruction id ID #REQUIRED,
                nom CDATA #Required,
                instructionSuivante CDATA #IMPLIED>
    <!ELEMENT DonneeUtilisee EMPTY>
    <!ATTLIST DonneeUtilisee id CDATA #REQUIRED>
]>
```

On remarque ici que le lien d'utilisation entre une instruction et une donnée nécessite un nouveau type d'élément appelé DonneeUtilisee. Ceci est du au fait que les modèles représentés sont des graphes et que ce formalisme ne représente que des données arborescentes. Il est donc nécessaire de passer par des identifiants et des références à ces identifiants pour représenter nos modèles.

Ainsi, les informations de la Figure 1 pourront être représentées en utilisant ce formalisme de la façon suivante :

```
<?xml version="1.0" ?>
<!DOCTYPE Site "Site.dtd">
<Site nom="s">
  <Programme id="#1" nom="p1">
  </Programme>
  <Programme id="#2" nom="p2">
    <Donnee id="#4" nom="d1" >
    </Donnee>
    <Donnee id="#5" nom="d2" >
    </Donnee>
    <Instruction id="#6" nom="i1" instructionSuivante="#6">
      <DonneeUtilisee  id="#4">
      </DonneeUtilisee>
    </Instruction>
    <Instruction id="#7" nom="i2">
      <DonneeUtilisee  id="#4">
      </DonneeUtilisee>
      <DonneeUtilisee  id="#5">
      </DonneeUtilisee>
    </Instruction>
  </Programme>
  <Programme id="#3" nom="p3">
  </Programme>
</Site>
```

La représentation de ces informations devient alors rapidement complexe et la manipulation de celles-ci se complexifie également.

2.2.3 Le formalisme MOF.

Le MOF est le langage d'expression de méta-modèles. Il se trouve au cœur des préoccupations de cette thèse et un chapitre y est donc consacré. Nous vous renvoyons donc au chapitre 3 pour une description détaillée de ce formalisme.

2.2.4 Le formalisme XMI.

XMI (XML Metadata Interchange) est un format d'échange de modèles basé sur la sémantique du MOF et sur la syntaxe de XML. Ce format est issu d'une mise en correspondance des descriptions MOF de méta-modèles et de la façon de définir celles-ci dans une DTD. L'application de cette mise en correspondance sur un méta-modèle permet d'obtenir une DTD standard pour l'échange des modèles décrits à l'aide de ce méta-modèle. Il existe ainsi une DTD standard permettant l'échange de modèles UML (obtenue en appliquant ces règles de mise en correspondance au méta-modèle de UML exprimé à l'aide du MOF) ainsi qu'une DTD standard permettant l'échange de méta-modèles MOF (obtenue en appliquant ces règles de mise en correspondance au méta-modèle du MOF lui-même). XMI est présenté plus en détails dans le chapitre 3.2.2.

Nous allons maintenant présenter plus en détail le MOF car celui-ci va très certainement devenir le standard industriel de représentation de méta-modèles.

3 L'exemple du MOF (Meta Object Facility).

Le MOF est une spécification définie par des industriels pour répondre à un besoin en terme de modélisation d'information de plus en plus diverse et hétérogène. La version 1.3 du MOF, publiée le 2 juillet 1999, est décrite dans ce chapitre. Elle est présentée ici de façon exhaustive et objective dans des termes proches de sa définition officielle de manière à servir de base de discussion pour les chapitres suivants. Le but du MOF est de définir un langage de modélisation unique et utilisé par tous pour représenter des méta-modèles et les modèles qui en découlent. Il est constitué d'un ensemble relativement petit (bien que non minimal) de concepts "objet" permettant de modéliser ce type d'information. Le MOF peut être étendu par héritage ou par composition de manière à représenter des modèles plus évolués. Le MOF est destiné a supporter un grand nombre d'utilisations, de même qu'il est destiné à être supporté par un grand nombre d'applications.

Il peut être utilisé pour définir le méta-modèle d'un domaine d'intérêt particulier. Auquel cas ce méta-modèle pourra entraîner des choix de conception et d'implémentation pour les outils qui lui seront dévolus. Le MOF est alors considéré comme un méta-méta-modèle car il est utilisé pour définir des méta-modèles.

Il peut également être utilisé par un développeur qui désire accéder à des informations stockées dans un référentiel (ou autre application) conforme au MOF. Celui-ci dispose alors d'un certain nombre de mécanismes (via des interfaces CORBA) d'accès à ces informations, ainsi qu'à la structure de ces informations (via des interfaces de réflexion). Il peut alors manipuler ces informations sans connaître au préalable cette structure.

3.1 Le MOF : Un langage de définition de méta-modèles.

Le MOF a donc été défini dans le but de formaliser les langages de modélisation. Il est actuellement utilisé pour sa propre définition (le MOF est en effet réflexif), mais il est également utilisé pour définir le méta-modèle d'UML. Le principal atout du MOF est son ouverture. Son but est de proposer un framework qui pourra supporter tout type de méta-modèles et qui permettra, au besoin, d'enrichir ces méta-modèles avec de nouveaux concepts. Il s'inscrit dans une architecture de méta-modélisation à quatre niveaux. Le point clé d'une telle

architecture est que son méta-méta-modèle va permettre la représentation, via un formalisme commun, de modèles et de méta-modèles.

Le modèle MOF correspond au méta-méta-modèle d'une architecture à quatre niveaux déjà représentée Figure 6. C'est un langage de modélisation à "objets". Il est utilisé pour définir la structure et la sémantique de méta-modèles, qu'ils soient généralistes ou liés à un domaine plus spécifique. Il est particulièrement bien adapté pour définir à la fois des méta-modèles des formalismes à "objets", des méta-modèles plus traditionnels (relationnels, entité-association) ou tout autre méta-modèle encore plus élémentaire.

La notation graphique utilisée pour représenter les méta-modèles MOF est la notation UML "adaptée" au MOF. Par conséquent, les associations et les classes représentées via cette notation ne sont pas des associations et des classes UML, mais des associations et des classes MOF. Le MOF lui-même, de par sa réflectivité, peut donc se représenter via cette notation.

Le MOF, sans être vraiment minimal, peut tout de même se représenter entièrement sur un modèle (c. f. Figure 18).

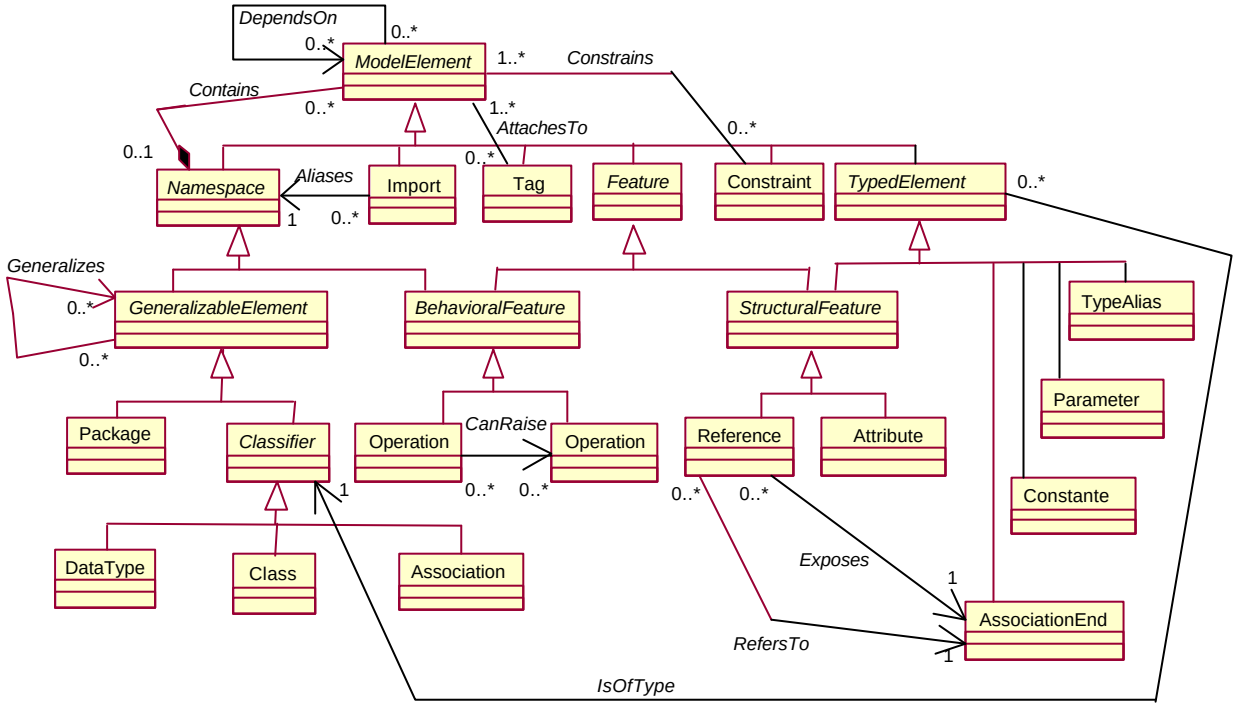


Figure 18 - Le MOF version 1.3.

3.1.1 Description des concepts du MOF.

Le MOF est auto-défini en terme de classes, d'associations, de paquetages, de types de données, d'exceptions, de constantes et de contraintes. Cette auto-définition apporte un certain nombre d'avantages :

Elle permet de valider le modèle MOF. En effet, puisque le MOF peut s'auto-définir, il doit être capable de définir d'autres méta-modèles d'envergure et de complexité similaire.
Elle permet d'appliquer tous les outils et tous les algorithmes applicables aux méta-modèles MOF sur le MOF lui-même. C'est le cas du mapping IDL, mais également du mapping XMI.
Elle permet de traiter de façon similaire les méta-modèles et les modèles dans de tels outils.
Elle peut servir de base pour de futures extensions ou modifications du MOF.

Le MOF est basé sur un framework de modélisation objet qui est simplement un sous-ensemble du cœur d'UML dont les quatre principaux concepts sont les suivants :

- Les classes, qui permettent de définir les types des méta-objets du MOF,
- Les associations, qui permettent de modéliser des relations binaires entre les méta-objets,
- Les types de données, qui permettent de modéliser les autres données (types primitifs, etc…) et
- Les paquetages, qui rendent ces modèles modulaires.

Ainsi, toute entité du niveau M1 aura pour type une classe ou un type de données défini au niveau M2 (i.e méta-entité). De même, toute entité du niveau M2 (i.e. méta-entité) aura pour type une classe ou un type de données défini au niveau M3 (i.e. méta-méta-entité). Une classe MOF définie au niveau M3 trouvera ses instances au niveau M2. Seuls les classes et les types de données définissent des types susceptibles d'avoir des instances.

3.1.1.1 Les classes.

Les classes sont des méta-objets MOF qui définissent les types des entités représentées via le MOF. Des classes définies au niveau M2 (cf. Figure 6) auront leurs instances au niveau M1. Ces instances ont un identifiant, un état et un comportement. L'état et le comportement des instances du niveau M1 est défini par leur classe au niveau M2 dans le contexte du MOF. Les classes sont principalement constituées de trois types d'entités - attributs, opérations et références - mais elles peuvent également contenir des exceptions, des constantes, des contraintes, des types de données, des étiquettes ("tag") ou encore d'autres définitions de classes.

Les attributs et les opérations peuvent être définis au niveau des classes ou au niveau des instances. Un attribut d'instance dispose d'une valeur différente pour chaque instance de la classe tandis qu'un attribut de classe définit une valeur commune pour toutes les instances de la classe.

De la même façon, une opération d'instance peut seulement être invoquée sur une instance de la classe (et en modifiera probablement l'état) tandis qu'une opération de classe peut être invoquée indépendamment d'une instance précise de la classe.

Une classe dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de la classe,
- Un attribut *annotation* qui décrit la classe,
- Un attribut *qualifiedName* qui désigne le nom global unique de la classe, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cette classe,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,
- Un attribut *isAbstract* qui détermine si la classe est abstraite, auquel cas il ne sera pas possible d'en créer des instances,
- Un attribut *isRoot* qui détermine si la classe est une racine d'un arbre d'héritage, auquel cas elle ne peut pas avoir de super-classes,
- Un attribut *isLeaf* qui détermine si la classe est une feuille d'un arbre d'héritage, auquel cas elle ne peut pas avoir de sous-classes,
- Un attribut *isSingleton* qui indique que la classe ne peut disposer que d'une instance.

Elle dispose également des références suivantes :

- Une référence *container* vers le conteneur de cette classe,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cette classe,
- Une référence *constraints* vers les contraintes définies pour cette classe,
- Une référence *contents* vers les éléments définis dans cette classe (attributs, références, opérations, etc…),
- Une référence *supertypes* vers les super-classes de cette classe,

De plus, les contraintes suivantes s'appliquent aux classes MOF :

- Toute classe doit disposer d'un conteneur ,
- Les noms de tous les éléments contenus dans une classe doivent être différents ,
- Une classe ne peut pas être directement ou indirectement sa propre superclasse ,
- Les supertypes d'une classe sont nécessairement des classes ,

- Les noms des éléments contenus dans une classe doivent être différents des noms des éléments contenus dans une superclasse directe ou indirecte ,
- L'héritage multiple doit obéir à la règle dite "du diamant" : Une entité généralisable ne peut hériter par deux voies différentes d'éléments différents mais de même nom (elle peut par contre hériter par deux voies différents du même élément) ,
- Si l'attribut *isRoot* de la classe est à vrai, la classe ne peut avoir de supertypes ,
- Une classe ne peut hériter d'une classe ayant l'attribut *isLeaf* à vrai ,
- Une classe peut seulement contenir des classes, des types de données (datatypes), des attributs, des références, des opérations, des exceptions, des contraintes, des constantes et des étiquettes (tags) ,
- Une classe abstraite ne peut pas être singleton.

3.1.1.2 Les attributs.

Un attribut est un méta-objet MOF qui permet de rattacher des valeurs à une instance d'une classe MOF. La relation entre la valeur cet l'attribut et l'instance aura une sémantique différente suivant que le type de l'attribut est une classe MOF ou un type de données MOF.

Un attribut dont le type est une classe MOF verra sa valeur rattachée de façon "composite" à l'instance pour laquelle il est défini. Une relation composite est une relation liant fortement les objets liés et respectant les propriétés suivantes :

- Une relation "composite" est une relation asymétrique avec un objet appelé le composé d'un côté (en l'occurrence l'instance) et un objet appelé le composant de l'autre (en l'occurrence la valeur de l'attribut),
- Un objet ne peut pas être le composant de plusieurs composés au même moment,
- Un objet ne peut pas être son propre composant ou le composant de l'un de ses composants (et ainsi de suite…),
- Quand on supprime un composé, tous ses composants sont également supprimés.

Un attribut dont le type est un type de données MOF verra sa valeur rattachée de façon "non-agrégée" à l'instance pour laquelle il est défini. Une telle relation est une relation liant faiblement

les objets liés et n'imposant aucune contrainte sur la multiplicité, l'origine ou encore le cycle de vie des objets liés[1].

Dans ce qui suit, il sera identifié par "attribut MOF" pour ne pas le confondre avec ses propres attributs.

Un attribut MOF dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de l'attribut MOF,
- Un attribut *annotation* qui décrit le rôle de l'attribut MOF,
- Un attribut *qualifiedName* qui désigne le nom global unique de l'attribut MOF, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cet attribut MOF,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,
- Un attribut *scope* pouvant prendre pour valeur *instance_level* ou *classifier_level* et déterminant si la valeur de l'attribut MOF est portée par sa classe (et par conséquent commune à toutes les instances de la classe) ou par l'instance,
- Un attribut *multiplicity* qui indique si la valeur de l'attribut MOF est optionnelle, simple ou multiple. Cet attribut détermine également, dans le cas de valeurs multiples, si ces valeurs sont ordonnées et si elle sont uniques (toutes différentes deux à deux).
- Un attribut *isChangeable* qui détermine si la valeur de l'attribut MOF peut être modifiée,
- Un attribut *isDerived* qui indique si la valeur de l'attribut est "déductible" à partir d'autres attributs.

Il dispose également des références suivantes :

- Une référence *container* vers le conteneur de cet attribut MOF (généralement sa classe),
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cet attribut MOF,
- Une référence *constraints* vers les contraintes définies pour cet attribut MOF,
- Une référence *type*, qui peut être une classe ou un type de données (pas une association) et qui définit le type de la valeur de cet attribut MOF.

De plus, les contraintes suivantes s'appliquent aux attributs MOF :

[1] Si l'on veut définir un attribut de type Classe avec un lien non "composite" vers son instance, il faut définir un type de donnée qui correspondra à cette classe et y sera lié via un alias.

- Tout attribut MOF doit disposer d'un conteneur ,
- Le type d'un attribut MOF ne peut pas être une association .

3.1.1.3 Les opérations.

Les opérations permettent d'accéder au comportement défini dans une classe. Les opérations ne spécifient pas le comportement ou les méthodes qui vont implémenter ce comportement, elles spécifient simplement le nom et la signature par lesquels ce comportement peut être invoqué.

Une opération MOF dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de l'opération,
- Un attribut *annotation* qui décrit le rôle de l'opération,
- Un attribut *qualifiedName* qui désigne le nom global unique de l'opération, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cette opération,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,
- Un attribut *scope* qui détermine si l'opération doit être invoquée sur une instance de la classe MOF dans laquelle la méthode est définie ou s'il est peut être invoquée indépendamment d'une instance particulière de cette classe MOF,
- Un attribut *isQuery* qui indique que cette opération ne modifiée pas l'état des instances de la classe MOF sur lesquelles elle est invoquée.

Elle dispose également des références suivantes :

- Une référence *container* vers le conteneur de cette opération,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cette opération,
- Une référence *constraints* vers les contraintes définies pour cette opération,
- Une référence *contents* vers les éléments définis dans cette opération (paramètres, contraintes, "tags"),
- Une référence *exceptions* vers les exceptions susceptibles d'être levée lors de l'invocation de cette opération.

De plus, les contraintes suivantes s'appliquent aux opérations MOF :

- Toute opération doit disposer d'un conteneur ,
- Les noms de tous les éléments contenus dans une opération doivent être différents ,
- Une opération ne peut contenir que des paramètres, des contraintes et des étiquettes ("tags") ,
- Une opération peut avoir au plus un paramètre dont la direction est "*return*" .

3.1.1.4 Les associations.

Les associations sont des méta-objets MOF qui permettent d'exprimer les relations dans un méta-modèle. Une association définie au niveau M2 (cf. Figure 6) va permettre de définir des liens entre des couples d'instances (de classes MOF) définis au niveau M1. Ces liens ne sont pas des objets et n'ont donc pas d'identité. Ils ne peuvent pas avoir d'attributs ou d'opérations.

Une association est composée d'exactement deux extrémités (*association end*). Une association peut être une relation "composite" auquel cas l'une de ses extrémités aura son attribut *aggregation* à *composite* et désignera le composé tandis que l'autre extrémité aura son attribut *aggregation* à *none* et désignera le composant. Une relation composite est une relation liant fortement les objets liés et respectant les propriétés suivantes :

- Une relation "composite" est une relation asymétrique avec un objet appelé le composé d'un côté (en l'occurrence l'instance) et un objet appelé le composant de l'autre (en l'occurrence la valeur de l'attribut),
- Un objet ne peut pas être le composant de plusieurs composés au même moment,
- Un objet ne peut pas être son propre composant ou le composant de l'un de ses composants (et ainsi de suite…),
- Quand on supprime un composé, tous ses composants sont également supprimés.

Une association dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de l'association,
- Un attribut *annotation* qui décrit le rôle de l'association,
- Un attribut *qualifiedName* qui désigne le nom global unique de l'association, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cette association,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,

- Un attribut *isAbstract* nécessairement à faux et sans intérêt pour une association MOF (il est hérité de *GeneralizableElement* mais n'a pas de sens car l'héritage n'est pas applicable aux associations MOF),
- Un attribut *isRoot* nécessairement à vrai pour assurer que l'héritage n'est pas applicable aux associations MOF,
- Un attribut *isLeaf* nécessairement à vrai pour assurer que l'héritage n'est pas applicable aux associations MOF,
- Un attribut *isDerived* qui indique que cette association est dérivée. Une association dérivée définie au niveau M2 (cf. Figure 6) ne permet pas de définir de liens au niveau M1. Ces liens dérivés sont déduits d'autres éléments du modèle (la sémantique de ces liens dérivés est en dehors du champ de cette spécification du MOF).

Elle dispose également des références suivantes :

- Une référence *container* vers le conteneur de cette association,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cette association,
- Une référence *constraints* vers les contraintes définies pour cette association,
- Une référence *contents* vers les éléments définis dans cette association (extrémités, contraintes, etc…),
- Une référence *supertypes* sans intérêt pour une association MOF (elle est héritée de *GeneralizableElement* mais n'a pas de sens car l'héritage n'est pas applicable aux associations MOF).

De plus, les contraintes suivantes s'appliquent aux associations MOF :

- Toute association doit disposer d'un conteneur ,
- Les noms de tous les éléments contenus dans une association doivent être différents ,
- Une association ne peut pas être directement ou indirectement sa propre super-association [2],
- Les supertypes d'une association sont nécessairement des associations [2],
- Les noms des éléments contenus dans une association doivent être différents des nom des éléments contenus dans une super-association directe ou indirecte [2],
- L'héritage multiple doit obéir à la règle du diamant (c.f. contraintes du chapitre 3.1.1.1) [2],

[2] Contrainte sans intérêt étant donné que l'héritage n'est pas applicable pour les associations MOF

- Si l'attribut *isRoot* de l'association est à vrai, l'association ne peut avoir de supertypes ,
- Une association ne peut hériter d'une association ayant l'attribut *isLeaf* à vrai ,
- Une association peut seulement contenir des extrémités, des contraintes et des étiquettes (tags) ,
- L'héritage n'est pas applicable aux associations ,
- Les attributs *isRoot* et *isLeaf* d'une association sont nécessairement à vrai ,
- Une association ne peut pas être abstraite ,
- Une association est nécessairement binaire (elle dispose d'exactement deux extrémités) .

3.1.1.5 Les extrémités d'associations.

Une extrémité est un méta-objet MOF qui décrit l'extrémité d'un lien et qui dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de cette extrémité de l'association,
- Un attribut *annotation* qui décrit cette extrémité de l'association,
- Un attribut *qualifiedName* qui désigne le nom global unique de cette extrémité, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cette extrémité,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,
- Un attribut *multiplicity* qui indique le nombre d'instances de cette extrémité du lien pouvant être liés (via ce lien) à **une** instance de l'autre extrémité du lien. On ne peut avoir deux occurrences d'un même lien (entre deux même instances). Cet attribut indique également si ces instances liées sont ordonnées,
- Un attribut *aggregation* pouvant prendre les valeurs *none* ou *composite*, et indiquant si l'objet connecté à cette extrémité du lien est un composé via cette relation, auquel cas cette relation est dite "composite",
- Un attribut *isNavigable* qui va déterminer s'il est possible de définir une référence pour l'instance de cette extrémité du lien,
- Une attribut *isChangeable* qui indique s'il est possible de modifier l'instance de cette extrémité du lien.

Une extrémité d'association dispose également des références suivantes :

- Une référence *container* vers le conteneur de cette extrémité (l'association),

- Une référence *requiredElements* vers les éléments nécessaires à la définition de cette extrémité,
- Une référence *constraints* vers les contraintes définies pour cet extrémité,
- Une référence *type* vers la classe MOF liée via cette extrémité à l'association.

De plus, les contraintes suivantes s'appliquent aux extrémités d'associations MOF :

- Toute extrémité d'association MOF doit disposer d'un conteneur ,
- Le type d'une extrémité d'association MOF ne peut pas être une association ,
- Le type d'une extrémité d'association MOF est obligatoirement une classe MOF ,
- Une extrémité est nécessairement marquée comme étant unique (pour une valeur de l'extrémité opposée, on ne peut avoir deux fois la même valeur pour cette extrémité ; ce "marquage" s'effectue via l'attribut *multiplicity*) ,
- Une association ne peut avoir deux extrémités marquées comme étant ordonnées (ce "marquage" s'effectue également via l'attribut *multiplicity*) ,
- Une association ne peut pas être marquée comme étant "composite" (valeur *composite* de l'attribut *aggregation*) à ses deux extrémités .

3.1.1.6 Les références.

Le MOF propose deux mécanismes pour construire des relations entre classes. Ces deux mécanismes sont les attributs et les associations[3]. La principale différence entre ces deux mécanismes est la façon dont ces informations seront exploitées.

Les associations apportent un accès orienté "requête" sur le modèle. C'est l'association qui connaît ses extrémités et ce sont ces extrémités qui connaissent les types (Classes MOF) associés. Par conséquent, une instance d'une classe MOF ne connaît pas directement les associations auxquelles elle est liée. Il est possible de faire des opérations sur un ensemble de liens d'un type d'association donné :

- Avantage : ceci permet d'effectuer des requêtes globales sur toutes les relations et pas seulement celles qui "touchent" un objet donné,

[3] Les attributs pourrait également être utilisés pour modéliser les relations entre classes et entre types de données (*DataType*).

- Inconvénient : les opérations pour accéder et mettre à jour ces relations sont plus complexes.

Les attributs apportent un accès orienté "navigation" sur le modèle. C'est l'instance de la classe MOF qui connaît ses attributs et sait y accéder. Il est possible d'accéder simplement en lecture/écriture aux valeurs des attributs d'un objet donné :

- Avantage : les accesseurs en lecteur/écriture sont plus simples,
- Inconvénient : effectuer des requêtes globales sur des relations exprimées à l'aide d'attributs est beaucoup plus complexe.

Définir une référence dans une classe MOF permet d'obtenir un accès orienté "navigation" (une référence est vue comme un attribut MOF) sur les associations. La différence est qu'au lieu de permettre la mise à jour et la consultation d'une valeur d'un attribut de l'instance de la classe MOF, la référence permet la mise à jour et la consultation des associations qui partent ou qui aboutissent à cette instance.

Une référence est un méta-objet MOF qui permet à une instance d'une classe MOF de connaître ses relations avec d'autres instances. Dans ce qui suit, elle sera identifiée "référence MOF" pour ne pas la confondre avec ses propres références.

Une référence MOF dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de la référence MOF,
- Un attribut *annotation* qui décrit le rôle de la référence MOF,
- Un attribut *qualifiedName* qui désigne le nom global unique de la référence MOF, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cet référence MOF,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,
- Un attribut *scope* sans intérêt pour une référence MOF (il est hérité de *Feature* mais n'a pas de sens pour une référence MOF),
- Un attribut *multiplicity* qui doit avoir la même valeur que l'attribut *multiplicity* de l'extrémité de l'association qui est référencée.
- Un attribut *isChangeable* qui doit également avoir la même valeur que l'attribut *isChangeable* de l'extrémité de l'association qui est référencée.

Elle dispose également des références suivantes :

- Une référence *container* vers le conteneur de cette référence MOF (généralement sa classe),
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cette référence MOF,
- Une référence *constraints* vers les contraintes définies pour cette référence MOF,
- Uune référence *type*, vers le type de l'entité référencée (nécessairement une classe MOF)
- Une référence *ReferencedEnd* vers l'extrémité de l'association qui est référencée (l'extrémité opposée par rapport à l'objet qui dispose de cette référence MOF).
- Une référence *ExposedEnd* vers l'extrémité de l'association qui est exposée (l'extrémité directement liée à l'objet qui dispose cette référence MOF).

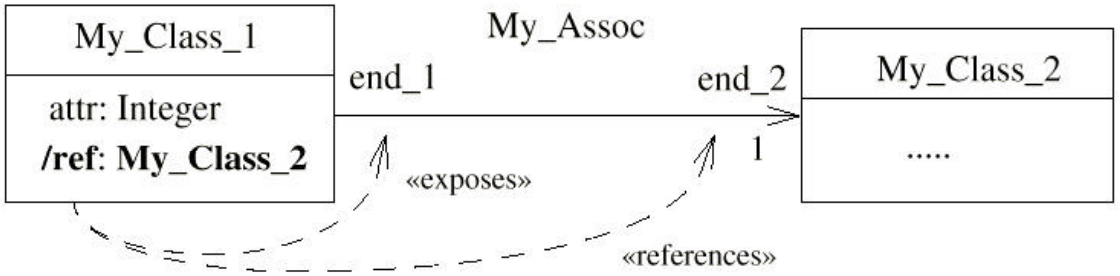


Figure 19 - Un exemple de référence MOF.

Une référence définie dans la classe MOF *My_Class_1* (cf. Figure 19) pour l'association *My_Assoc* devra avoir comme *ReferencedEnd* l'extrémité *end_2* qui est appelée l'extrémité référencée de l'association et comme *ExposedEnd* l'extrémité *end_1* qui est appelée l'extrémité exposée de l'association.

De plus, les contraintes suivantes s'appliquent aux références MOF :

- Toute référence MOF doit disposer d'un conteneur ,
- Le type d'une référence MOF ne peut pas être une association ,
- La multiplicité d'une référence doit être identique à la multiplicité de l'extrémité référencée de l'association ,
- L'attribut *scope* d'une référence a nécessairement pour valeur *instance_level* ,
- Une référence est modifiable seulement si l'extrémité référencée de l'association est également modifiable ,
- Le type de la référence doit être le même que le type de l'extrémité référencée de l'association ,

- Une référence ne peut être définie que pour une extrémité référencée navigable ,
- La classe contenant la référence doit être le type ou un sous-type de l'extrémité exposée de l'association .

3.1.1.7 Les paquetages.

Les paquetages permettent de regrouper les éléments d'un méta-modèle. Au niveau M2 (cf. Figure 6), les paquetages permettent de partitionner l'espace de méta-modélisation. Les paquetages vont pouvoir contenir les classes, les associations, les types de données, les exceptions, les constantes, ainsi que d'autres paquetages. Au niveau M1, une "instance" de paquetage va servir de contexte global pour la représentation d'un modèle. Le MOF fournit quatre mécanismes pour la composition et la réutilisation de méta-modèles. Ces quatre mécanismes que sont la généralisation, l'imbrication, l'importation et la composition (clustering) seront présentés par la suite.

Un paquetage MOF dispose des attributs suivants :

- Un attribut *name* qui désigne le nom du paquetage,
- Un attribut *annotation* qui décrit le paquetage,
- Un attribut *qualifiedName* qui désigne le nom global unique du paquetage, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant ce paquetage,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,
- Un attribut *isAbstract* nécessairement à faux et sans intérêt pour un paquetage MOF (il est hérité de *GeneralizableElement* mais n'a pas de sens car le MOF ne permet pas la déclaration d'un paquetage abstrait),

- Un attribut *isRoot* qui détermine si le paquetage est une racine d'un arbre d'héritage, auquel cas il ne sera pas possible de lui attribuer de super-paquetages,
- Un attribut *isLeaf* qui détermine si le paquetage est une feuille d'un arbre d'héritage, auquel cas il ne sera pas possible de lui attribuer de sous-paquetages.

Il dispose également des références suivantes :

- Une référence *container* vers le conteneur de ce paquetage,

- Une référence *requiredElements* vers les éléments nécessaires à la définition de ce paquetage,
- Une référence *constraints* vers les contraintes définies pour ce paquetage,
- Une référence *contents* vers les éléments définis dans ce paquetage (classes, associations, contraintes, etc…),
- Une référence *supertypes* vers les super-paquetages de ce paquetage.

De plus, les contraintes suivantes s'appliquent aux classes MOF :

- Les noms de tous les éléments contenus dans un paquetage doivent être différents ,
- Un paquetage ne peut pas être directement ou indirectement son propre super-paquetage ,
- Les supertypes d'un paquetage sont nécessairement des paquetages ,
- Les noms des éléments contenus dans un paquetage doivent être différents des nom des éléments contenus dans un super-paquetage directe ou indirecte ,
- L'héritage multiple doit obéir à la règle du diamant (c.f. contraintes du chapitre 3.1.1.1) ,
- Si l'attribut *isRoot* du paquetage est à vrai, il ne peut avoir de supertypes ,
- Un paquetage ne peut hériter d'un paquetage ayant l'attribut *isLeaf* à vrai ,
- Un paquetage peut seulement contenir des paquetages, des classes, des types de données (datatypes), des associations, des exceptions, des contraintes, des imports et des étiquettes (tags) ,
- Un paquetage ne peut pas être déclaré comme étant abstrait .

3.1.1.8 Les imports.

Un import permet à un paquetage de réutiliser des éléments MOF définis dans un autre paquetage. Un import est lié à l'objet importé via la référence *ImportedNamespace*. Quand un paquetage contient un import, il importe l'objet référencé par celui-ci via *ImportedNamespace*. Un objet import peut représenter un import de paquetage ou de classe, mais également une composition, auquel cas l'attribut *isClustered* est positionné à vrai.

Un import dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de l'import,
- Un attribut *annotation* qui décrit l'import,

- Un attribut *qualifiedName* qui désigne le nom global unique de l'import, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cet import,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,
- Un attribut *isClustered* indiquant si l'import définit une composition.

Il dispose également des références suivantes :

- Une référence *container* vers le conteneur de cet import,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cet import,
- Une référence *constraints* sans intérêt pour un import car le MOF ne permet pas de définir de contraintes pour un import,
- Une référence *ImportedNamespace* vers l'élément MOF importé.

De plus, les contraintes suivantes s'appliquent aux imports MOF :

- Tout import MOF doit disposer d'un conteneur ,
- Un paquetage ne peut importer ou être composé (via un import) que des classes ou des paquetages ,
- Un paquetage ne peut s'importer ou être composé (via un import) de lui-même ,
- Un paquetage ne peut importer ou être composé (via un import) d'un paquetage ou d'une classe qu'il contient ,
- Un paquetage imbriqué (contenu par un autre paquetage) ne peut pas contenir d'objets imports ,
- Les contraintes, tags (étiquettes), imports, alias et constantes ne peuvent pas être contraints .

3.1.1.9 Les types de données (DataType).

La définition d'un méta-modèle nécessite souvent l'utilisation de valeurs d'attributs ou de paramètres de type "primitifs". Le MOF propose le concept de type de donnée pour combler ce besoin.

En général, un type de données peut être utilisé pour représenter deux sortes de types :

- Les méta-modèles ont souvent besoin de définir des types dont les valeurs n'ont pas besoin d'identité (des valeurs entières, des chaînes de caractères, des types énumérés, etc…)
- Les méta-modèles ont parfois besoin d'utiliser des types "externes" (des types de données non définis dans un méta-modèle tels que les types image ou son, etc…)

Un type de données dispose des attributs suivants :

- Un attribut *name* qui désigne le nom du type,
- Un attribut *annotation* qui décrit le type,
- Un attribut *qualifiedName* qui désigne le nom global unique du type, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant ce type,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,
- Un attribut *isAbstract* nécessairement à faux et sans intérêt pour un type de données MOF (il est hérité de *GeneralizableElement* mais n'a pas de sens car l'héritage n'est pas applicable aux types de données MOF),
- Un attribut *isRoot* nécessairement à vrai pour assurer que l'héritage n'est pas applicable aux types de données MOF,
- Un attribut *isLeaf* nécessairement à vrai pour assurer que l'héritage n'est pas applicable aux types de données MOF,
- Un attribut *typeCode* qui définit ce type via l'interface CORBA::TypeCode (l'encodage des types de données du MOF doit être conforme à l'encodage des types de données de CORBA 2.2).

Il dispose également des références suivantes :

- Une référence *container* vers le conteneur de ce type de données,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de ce type de données,
- Une référence *constraints* vers les contraintes définies pour ce type de données,
- Une référence *contents* vers les éléments définis dans ce type de données (alias, contraintes, et étiquettes),
- Une référence *supertypes* sans intérêt pour un type de données (elle est héritée de *GeneralizableElement* mais n'a pas de sens car l'héritage n'est pas applicable aux types de données MOF).

De plus, les contraintes suivantes s'appliquent aux types de données :

- Tout type de données doit disposer d'un conteneur ,
- Les noms de tous les éléments contenus dans un type de données doivent être différents ,
- Une classe ne peut pas être directement ou indirectement sa propre superclasse ,
- Les supertypes d'un type de données sont nécessairement des type de données [4],
- Les noms des éléments contenus dans un type de données doivent être différents des nom des éléments contenus dans un super-type direct ou indirect [4],
- L'héritage multiple doit obéir à la règle du diamant (c.f. contraintes du chapitre 3.1.1.1) [4],
- Si l'attribut *isRoot* du type de données est à vrai, il ne peut avoir de supertypes ,
- Un type de données ne peut hériter d'un type ayant l'attribut *isLeaf* à vrai ,
- Un type de données peut seulement contenir des alias, de contraintes et des étiquettes ,
- La valeur de l'attribut *typeCode* doit correspondre à un type de données ou un type d'objet conforme à CORBA 2.2 ,
- L'héritage n'est pas applicable aux types de données ,
- Un type de données ne peut pas être définit comme étant abstrait .

3.1.1.10 Les alias.

Un alias permet d'indiquer q'un type de données est défini par une classe MOF. Comme indiqué au chapitre 3.1.1.2, un attribut dont le type est une classe MOF est nécessairement lié de façon "composite" à l'instance pour lequel il est défini. Les alias permettent de contourner cette règle en définissant l'attribut comme étant d'un type de données MOF disposant d'un alias vers la classe MOF qui définit ce type.

[4] Contrainte sans intérêt étant donné que l'héritage n'est pas applicable pour les types de données MOF.

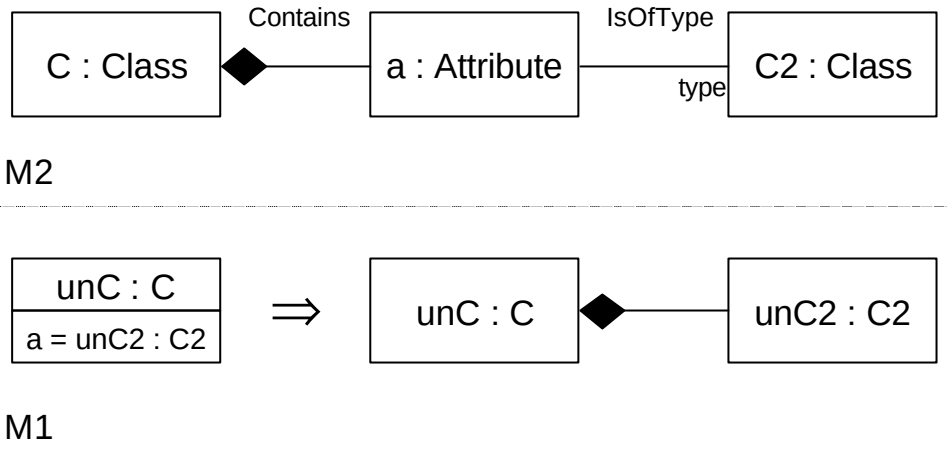


Figure 20 - Définition et utilisation d'un attribut a de type C2 (classe MOF).

Comme présenté sur la Figure 20, un attribut dont le type est une classe MOF au niveau M2 entraîne un lien "composite" entre l'instance de la classe dans laquelle cet attribut (unC) est défini et l'instance affectée comme valeur à cet attribut (unC2). Un lien "composite" est représenté par un losange plein.

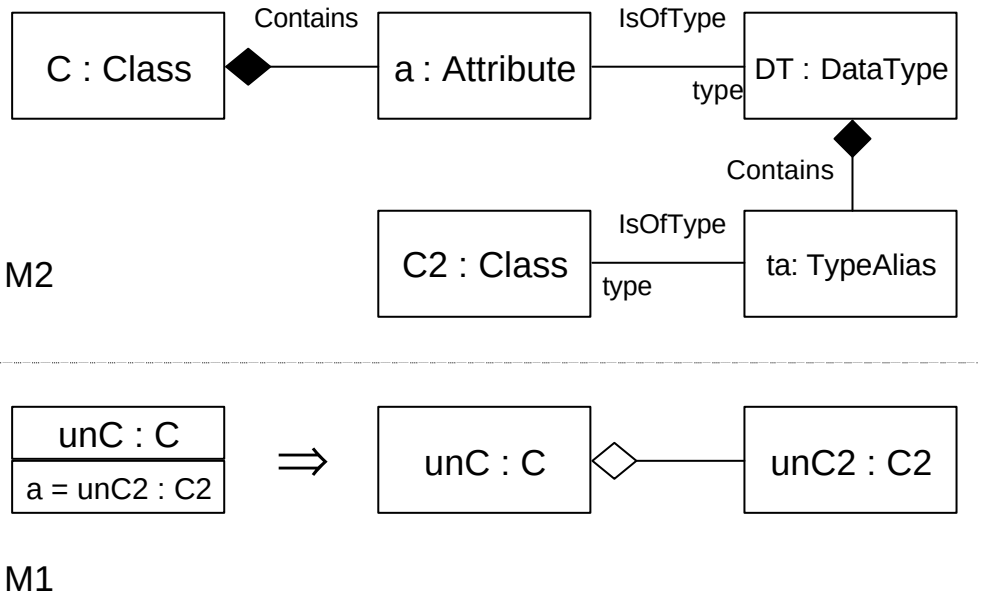


Figure 21 - Définition et utilisation d'un attribut a de type C2 (classe MOF) via un alias.

Comme présenté sur la Figure 21, il est possible d'utiliser un alias de manière à ce que le lien entre une instance et la valeur de l'un de ses attribut de type classe ne soit pas "composite".

Un alias dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de l'alias,
- Un attribut *annotation* qui décrit le rôle de l'alias,
- Un attribut *qualifiedName* qui désigne le nom global unique de l'alias, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cet alias.

Il dispose également des références suivantes :

- Une référence *container* vers le conteneur de cet alias (son type de données),
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cet alias,
- Une référence *constraints* sans intérêt pour un alias MOF car le MOF ne permet pas de définir de contraintes pour un alias MOF,
- Une référence *type* vers la classe MOF qui définit le type de données dans lequel cet alias est contenu.

De plus, les contraintes suivantes s'appliquent aux alias MOF :

- Tout alias MOF doit disposer d'un conteneur ,
- Le type d'un alias MOF ne peut pas être une association ,
- Les contraintes, tags (étiquettes), imports, alias et constantes ne peuvent pas être contraints .

3.1.1.11 Les exceptions.

Une exception MOF est un méta-objet qui permet de définir la signature des exceptions qu'une implémentation d'une opération MOF est susceptible de lever.

Une exception MOF dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de l'exception MOF,
- Un attribut *annotation* qui décrit le rôle de l'exception MOF,

- Un attribut *qualifiedName* qui désigne le nom global unique de l'exception MOF, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cette exception MOF,
- Un attribut *visibility* non utilisé dans la version actuelle du MOF,
- Un attribut *scope*, pouvant prendre pour valeur *instance_level* ou *classifier_level*, dont le sens pour une exception n'est pas précisé dans la spécification du MOF (elle est héritée de *Feature*).

Elle dispose également des références suivantes :

- Une référence *container* vers le conteneur de cette exception MOF,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cette exception MOF,
- Une référence *constraints* vers les contraintes définies pour cette exception MOF,
- Une référence *contents* vers les éléments définis dans cette exception (paramètres, contraintes et étiquettes).

De plus, les contraintes suivantes s'appliquent aux exceptions MOF :

- Toute exception MOF doit disposer d'un conteneur ,
- Les noms de tous les éléments contenus dans une exception doivent être différents ,
- Une exception ne peut contenir que des paramètres ou des étiquettes ,
- Tous les paramètres définis dans une exception doivent avoir pour *direction* la valeur *out* .

3.1.1.12 Les paramètres.

Un paramètre MOF est un méta-objet utilisé pour définir la signature des opérations et des exceptions MOF.

Un paramètre MOF dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de ce paramètre,
- Un attribut *annotation* qui décrit le rôle de ce paramètre,

- Un attribut *qualifiedName* qui désigne le nom global unique de ce paramètre, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant ce paramètre,
- Un attribut *direction* qui indique si c'est un paramètre en entrée (*in_dir*), en sortie (*out_dir*), en entrée/sortie (*inout_dir*) ou le paramètre de retour (*return_dir*) de l'opération dans laquelle il est défini,
- Un attribut *multiplicity* qui indique si la valeur du paramètre MOF est optionnelle, simple ou multiple. Cet attribut détermine également, dans le cas de valeurs multiples, si ces valeurs sont ordonnées et si elle sont uniques (toutes différentes deux à deux).

Il dispose également des références suivantes :

- Une référence *container* vers le conteneur de ce paramètre,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de ce paramètre,
- Une référence *constraints* vers les contraintes définies pour ce paramètre,
- Une référence *type* vers la classe ou le type de données MOF qui définit le type de ce paramètre.

De plus, les contraintes suivantes s'appliquent aux paramètres MOF :

- Tout paramètre MOF doit disposer d'un conteneur ,
- Le type d'un paramètre MOF ne peut pas être une association .

3.1.1.13 Les constantes.

Une constante MOF est un méta-objet utilisé pour définir des constantes ayant pour type un type de données simple et conforme a CORBA 2.3.

Une constante MOF dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de cette constante,
- Un attribut *annotation* qui décrit le rôle de cette constante,
- Un attribut *qualifiedName* qui désigne le nom global unique de cette constante, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cette constante,
- Un attribut *value* qui contient la valeur de cette constante.

Il dispose également des références suivantes :

- Une référence *container* vers le conteneur de cette constante,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cette constante,
- Une référence *constraints* sans intérêt pour une constante MOF car le MOF ne permet pas de définir de contraintes pour une constante MOF,
- Une référence *type* vers le type de données MOF qui définit le type de cette constante.

De plus, les contraintes suivantes s'appliquent aux paramètres MOF :

- Toute constante MOF doit disposer d'un conteneur ,
- Le type d'une constante MOF ne peut pas être une association ,
- Les contraintes, tags (étiquettes), imports, alias et constantes ne peuvent pas être contraints .
- Le type d'une constante MOF (référence *type*) doit être identique au type de sa valeur (attribut *value*) ,
- Le type d'une constante MOF doit être un type de données CORBA légal pour une déclaration de constante CORBA 2.3 .

3.1.1.14 Les contraintes.

Une contrainte MOF est un méta-objet qui permet de rattacher des règles de consistance sur les instances des méta-objets MOF. Une contrainte définit une règle qui restreint l'état ou le comportement de l'instance ou des instances contrainte(s).
Bien que certaines contraintes doivent être considérées comme des invariants du modèle, il est tout de même intéressant de ne pas les imposer tout le temps, ne serait-ce que lors de la construction du modèle.

Une contrainte MOF dispose des attributs suivants :

- Un attribut *name* qui désigne le nom de la contrainte,
- Un attribut *annotation* qui décrit la contrainte,
- Un attribut *qualifiedName* qui désigne le nom global unique de la contrainte, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant cette contrainte,

- Un attribut *expression* qui permet d'exprimer dans une chaîne de caractères la contrainte dans le langage indiqué par l'attribut *language*,
- Un attribut *language* qui indique le langage utilisé pour exprimer la contrainte,
- Un attribut *evaluationPolicy* qui détermine le moment d'évaluation de cette contrainte. Cet attribut peut prendre les valeurs *immediate*, auquel cas cette contrainte engendrera une exception MOF dès qu'elle ne sera plus respectée, ou *deferred* (différé), auquel cas la contrainte ne pourra être évaluée qu'explicitement via l'opération *verify*.

Elle dispose également des références suivantes :

- Une référence *container* vers le conteneur de cette contrainte,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de cette contrainte,
- Une référence *constraints* sans intérêt pour une contrainte MOF car le MOF ne permet pas de définir de contraintes pour une contrainte MOF,
- Une référence *constrainedElements* vers les éléments contraints par cette contrainte.

De plus, les contraintes suivantes s'appliquent aux contraintes MOF :

- Toute contrainte MOF doit disposer d'un conteneur,
- Les contraintes, tags (étiquettes), imports, alias et constantes ne peuvent pas être contraints,
- Une contrainte peut seulement contraindre des éléments de modélisation définis ou hérités par son conteneur immédiat.

3.1.1.15 Les étiquettes (tag).

Un tag MOF est un méta-objet qui permet d'étendre ou de modifier un méta-modèle MOF sans avoir à l'altérer. Il permet de changer le sens d'un élément de modélisation. Ces tags peuvent alors être pris en compte par les outils manipulant ces méta-modèles MOF. Un tag est une sorte de marqueur ou d'étiquette que l'on place sur les méta-objets pour préciser leur sens et en tenir compte par la suite.

Un tag MOF dispose des attributs suivants :

- Un attribut *name* qui désigne le nom du tag,

- Un attribut *annotation* qui décrit le tag,
- Un attribut *qualifiedName* qui désigne le nom global unique du tag, cet attribut est dérivé de l'attribut *name* et des nom successifs des paquetages contenant ce tag,
- Un attribut *tagId* qui identifie le tag,
- Un attribut *values* qui définit les valeurs de ce tag.

Il dispose également des références suivantes :

- Une référence *container* vers le conteneur de ce tag,
- Une référence *requiredElements* vers les éléments nécessaires à la définition de ce tag,
- Une référence *constraints* sans intérêt pour un tag MOF car le MOF ne permet pas de définir de contraintes pour un tag,
- Une référence *elements* vers les éléments MOF auxquels ce tag est rattaché.

De plus, les contraintes suivantes s'appliquent aux tag MOF :

- Tout tag MOF doit disposer d'un conteneur ,
- Les contraintes, tags (étiquettes), imports, alias et constantes ne peuvent pas être contraints .

3.1.2 Description des relations du MOF.

Ce chapitre présente toutes les relations définies entre les classes MOF du chapitre précédent.

3.1.2.1 La relation d'héritage (Generalizes).

Le MOF définit une relation d'héritage (*Generalizes*) applicable pour les éléments de type classes et pour les éléments de type paquetage. Cette relation lie donc un sous-type (le *subtype*) à son ou ses super-types (*supertype*). Bien que les types de données et les associations soient également des sous-types de *GeneralizableElement* (classe MOF définissant les éléments généralisables), le MOF ne permet pas l'héritage pour ces entités.

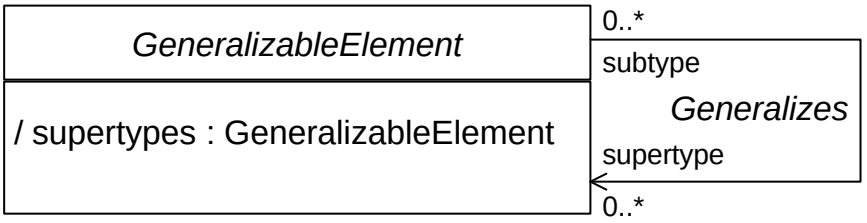


Figure 22 - La relation *Generalizes*.

Cette relation est navigable du sous-type vers le super-type. Ce qui permet la définition d'une référence *supertypes* de manière à accéder directement aux super-types d'un élément généralisable.

3.1.2.1.1 L'héritage des classes.

Le MOF permet aux classes d'hériter d'une ou plusieurs autres classes. La généralisation de classes MOF est similaire à la généralisation de classes UML, ou encore l'héritage d'interfaces IDL en CORBA. La sous-classe hérite de tout le contenu de ses super-classes. C'est à dire tous les attributs, opérations, références de ses super-classes ainsi que toutes les exceptions, constantes, contraintes, types de données, "tag" (étiquette) et autres définitions de classes définis dans ses super-classes.

Les restrictions suivantes sont fixées sur la généralisation pour assurer sa compréhension et son "mapping" vers un certain nombre d'implémentations :

- Une classe ne peut pas se généraliser elle-même, que ce soit directement ou indirectement,
- La surcharge n'est pas autorisée : Une classe ne peut pas contenir un élément de modélisation (attribut, opération, etc…) portant le même nom qu'un élément définit dans une de ses super-classes,
- Lorsqu'une classe a plusieurs super-classes, tous les éléments contenus ou hérités par ces super-classes doivent avoir des noms différents (à l'exception du cas ou plusieurs de ces super-classes auraient un ancêtre commun).

Une classe peut être définie comme étant "*abstraite*". Une classe abstraite est nécessairement sous-classée car seules ses sous-classes pourront être instanciées.
Une classe peut être définie comme étant une "*root*" (racine) ou une "*leaf*" (feuille). Une classe feuille ne pourra pas être sous-classée tandis qu'une classe racine ne pourra pas avoir de super-classe.

3.1.2.1.2 L'héritage des paquetages

Le MOF permet aux paquetages d'hériter d'un ou plusieurs autres paquetages. La généralisation de paquetages MOF est similaire à la généralisation de classes MOF. Le sous-paquetage hérite de tout le contenu de ses super-types. C'est à dire toutes les classes, associations, types de données, imports, etc… contenus dans ses supers-types.

Les restrictions suivantes sont fixées sur la généralisation pour assurer sa compréhension et son "mapping" vers un certain nombre d'implémentations :

- Un paquetage ne peut pas se généraliser lui-même, que ce soit directement ou indirectement,
- La surcharge n'est pas autorisée : Une paquetage ne peut pas contenir un élément de modélisation (classe, association, etc…) portant le même nom qu'un élément définit dans une de ses super-types,
- Lorsqu'un paquetage a plusieurs super-types, tous les éléments contenus ou hérités par ces super-types doivent avoir des noms différents (à l'exception du cas ou plusieurs de ces super-types auraient un ancêtre commun).

Un paquetage ne peut pas être définie comme étant "*abstrait*".
Un paquetage peut être défini comme étant une "*root*" (racine) ou une "*leaf*" (feuille). Un paquetage feuille ne pourra pas être sous-classé tandis qu'un paquetage racine ne pourra pas avoir de super-type.

3.1.2.2 La relation de contenance (Contains).

Cette relation permet de rattacher les classes, associations et autres entités MOF pouvant être définis dans un paquetage à un paquetage. Mais, elle permet également de rattacher les attributs, et opérations à leur classe ou encore les paramètres à leur opération.

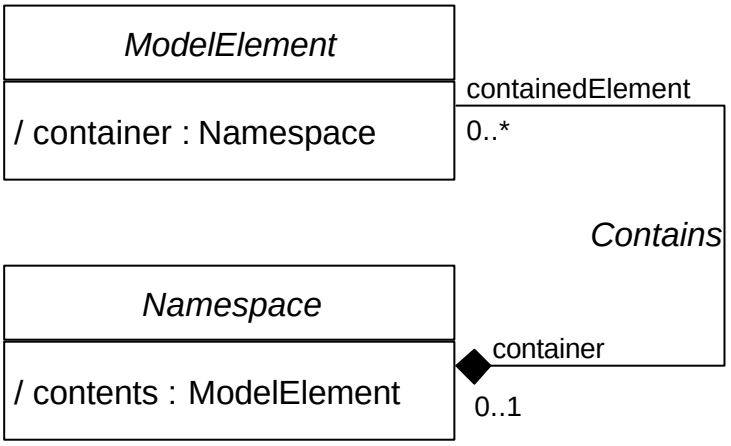


Figure 23 - La relation *Contains*.

Cette relation lie donc un élément (le *containedElement*) à son conteneur (le *container*). Elle est navigable dans les deux sens. Ce qui permet la définition d'une référence *container* sur tout les éléments de modélisation de manière à accéder directement à leur conteneur et d'une référence *contents* sur tous les conteneurs afin d'accéder directement à leur contenu.

Tout les élément MOF, à l'exception des paquetages, doivent être rattachés via cette relation à leur conteneur. De plus, tout élément MOF ne peut avoir qu'un seul conteneur.

Les paquetages peuvent être imbriqués via cette relation, mais sont les seuls éléments qui ne nécessitent pas d'avoir un conteneur.

Comme cette relation est définie à un niveau très abstrait, il est nécessaire de définir des contraintes sur les entités qui pourront effectivement être liées via cette relation. Ces contraintes sont regroupées dans le tableau suivant :

	Paquetage	Classe	Type de données	Association	Attribut	Référence	Opération	Exception	Paramètre	Extrémité d'association	Contrainte	Constante	Alias de Type	Import	Etiquette (Tag)
Paquetage	√	√	√	√				√			√			√	√
Classe		√	√		√	√	√	√			√	√			√
Type de Données											√		√		√
Association										√	√				√
Opération									√		√				√
Exception									√						√

En ligne sont présentés les conteneurs potentiels et en colonne les éléments qu'ils peuvent contenir. Ce tableau est entièrement spécifié dans le MOF via des contraintes. Ainsi, si l'on prends le cas de l'opération, on voit qu'elle ne peut contenir que des paramètres, des contraintes et des tags, ce qui correspond tout simplement à la contrainte C28 définie sur les entités de type Opération (c.f. troisième contrainte présentée au chapitre 3.1.1.3).

3.1.2.3 La relation de dépendance (DependsOn).

Cette relation permet de représenter les dépendance entre les éléments de modélisation. Elle lie un élément de modélisation (le *dependent*) à tous les éléments dont il dépend (*provider*). Elle est navigable du *dependent* vers le *provider*. Ce qui permet la définition d'une référence *requiredElements* sur tous les éléments de modélisation de manière à accéder directement à tous les éléments dont ils dépendent. La relation de dépendance est un association dérivée. C'est à dire qu'il n'est pas possible de définir explicitement un lien de dépendance entre deux éléments de modélisation, mais que cette relation de dépendance est déduite d'autres liens définis entre ces deux éléments.

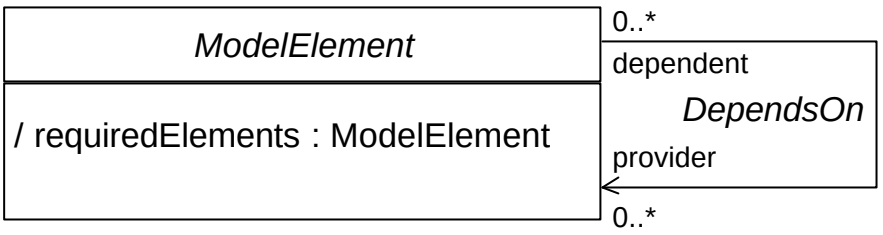


Figure 24 - La relation dérivée *DependsOn*.

Les règles de dérivation de cette association sont définies dans le MOF. Un élément de modélisation *me* sera lié à un autre élément de modélisation *me'* via cette relation *DependsOn* dans les cas suivants :

- *me'* est le conteneur de *me* (*me'* est lié à *me* en tant que *container* via la relation *Contains*),
- *me* est le conteneur de *me'* (*me* est lié à *me'* en tant que *container* via la relation *Contains*),
- *me'* est une contrainte définie pour *me* (*me'* est lié à *me* en tant que *constraint* via la relation *Constrains*),
- *me* est une contrainte définie pour *me'* (*me* est lié à *me'* en tant que *constraint* via la relation *Constrains*),
- *me* est généralisable et a pour supertype *me'* (*me'* est lié à *me* en tant que *supertype* via la relation *Generalizes*),
- *me* est un import et l'élément qu'il importe est *me'* (*me'* est lié à *me* en tant que *imported* via la relation *Aliases*),
- *me* est une opération et *me'* est une exception susceptible d'être levée par cette opération (*me'* est lié à *me* en tant que *except* via la relation *CanRaise*),
- *me* est un élément typé et *me'* est son type (*me'* est lié à *me* en tant que *type* via la relation *IsOfType*),
- *me* est une référence et *me'* est l'une des extrémités de l'association liée à cette référence (*me'* est lié à *me* en tant que *referencedEnd* via la relation *RefersTo*, ou en tant que *ExposedEnd* via la relation *Exposes*),
- *me* est une étiquette et *me'* est un élément auquel cette étiquette est rattachée (*me* est lié à *me'* en tant que *tag* via la relation *AttachesTo*).

3.1.2.4 La relation de typage(IsTypeOf).

Cette relation permet de lier un élément de modélisation typé (un attribut, un paramètre, une constante) à son type (une classe ou un type de données). Elle est également utilisée pour lier les extrémités d'associations aux classes MOF (une extrémité d'association est un élément de modélisation typé). Cette relation lie donc un élément (le *typedElement*) à son type (le *type*). Elle est navigable du *typedElement* vers son *type*. Ce qui permet la définition d'une référence *type* sur tous les éléments de modélisation typés de manière à accéder directement à ce type.

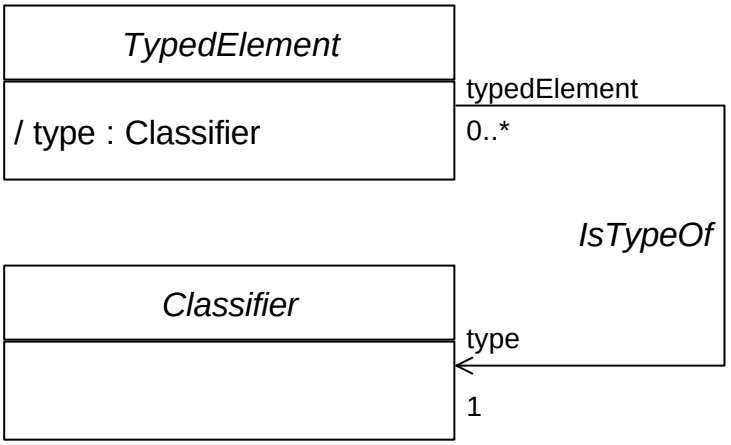


Figure 25 - La relation *IsTypeOf*.

3.1.2.5 Les relations entre références et associations (Exposes & RefersTo).

Ces deux associations permettent de faire le lien entre une référence et les deux extrémités de l'association liées à cette référence (l'extrémité exposée et l'extrémité référencée). L'association *Exposes* est dérivée. En effet, lorsqu'une référence est liée à une extrémité d'association via la relation *RefersTo*, elle est nécessairement liée à l'autre extrémité de l'association via la relation *Exposes*. Ces deux relations sont navigables de la référence vers l'extrémité exposée ou référencée. Ce qui permet la définition d'une référence *referenceEnd* permettant d'accéder directement à l'extrémité référencée de l'association et d'une référence *exposedEnd* permettant d'accéder directement à l'extrémité exposée de cette association.

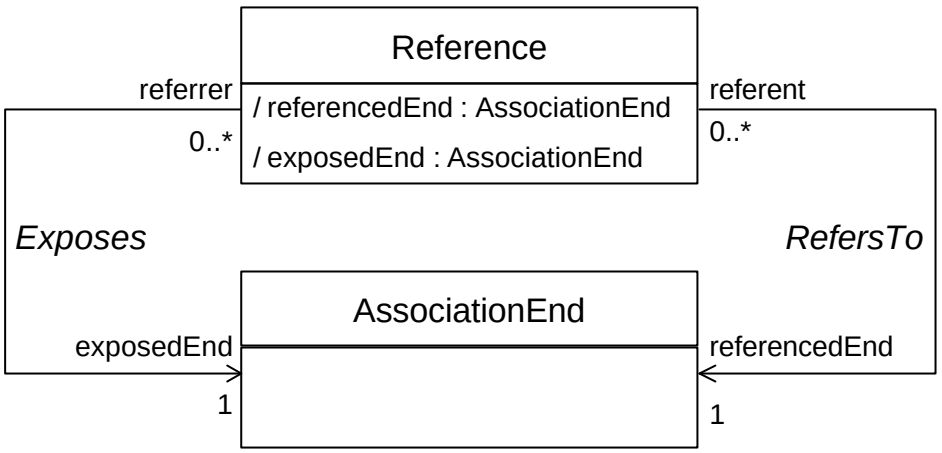


Figure 26 - Les relations *Exposes* & *RefersTo*.

3.1.2.6 La relation entre les opérations et leurs exceptions (CanRaise).

Cette relation permet de lier une opération aux exceptions qu'elle est susceptible de lever. Cette relation est navigable de l'opération vers ses exceptions. Ce qui permet la définition d'une référence *exceptions* sur toutes les opération de manière à accéder directement à ce type.

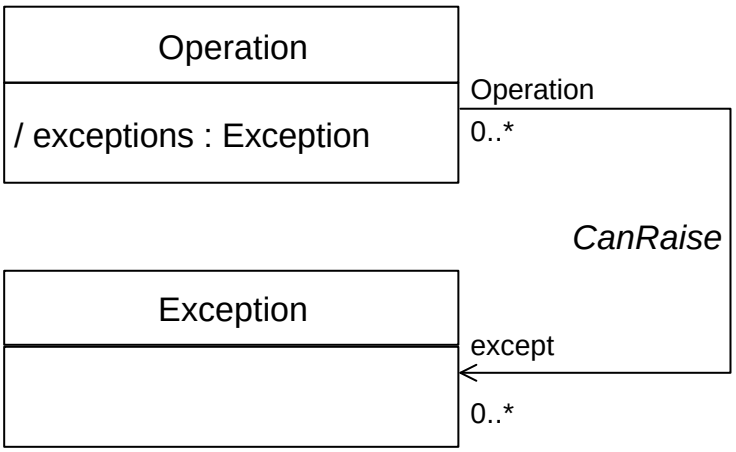


Figure 27 - La relation *CanRaise*.

3.1.2.7 La relation d'importation (Aliases).

Cette relation permet de lier un objet import à l'élément qu'il permet d'importer. Cette relation lie donc un import (l' *importer*) à l'élément importé (l' *imported*). Elle est navigable de l'importer vers l'importé. Ce qui permet la définition d'une référence *importedNamespace* sur tous les imports de manière à accéder directement à l'élément importé.

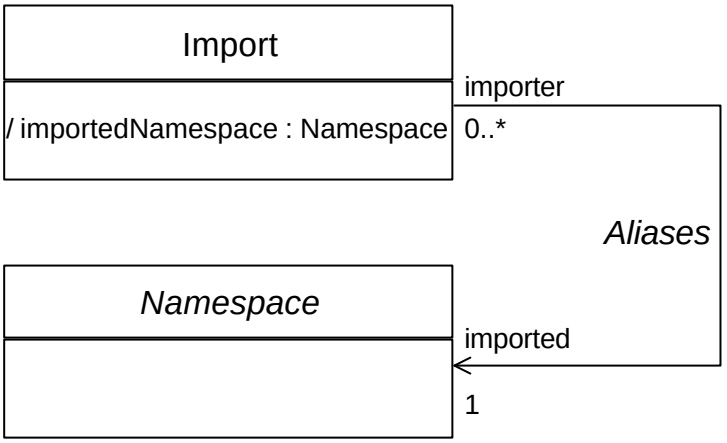


Figure 28 - La relation *Aliases*.

3.1.2.8 La relation entre les entités et leurs contraintes (Constraints).

Cette relation permet de lier un élément de modélisation aux contraintes qui le référence. Cette relation lie donc un élément de modélisation (le *constrainedElement*) a un ensemble de contraintes (les *constraint*). Elle est navigable dans les deux sens. Par conséquent une référence *importedNamespace* est définie sur tous les imports de manière à accéder directement à l'élément importé. Ce qui permet la définition d'une référence *constraints* sur tout les éléments de modélisation de manière à accéder directement à leurs contraintes et une référence *constrainedElements* sur toutes les contraintes afin d'accéder directement aux éléments contraints.

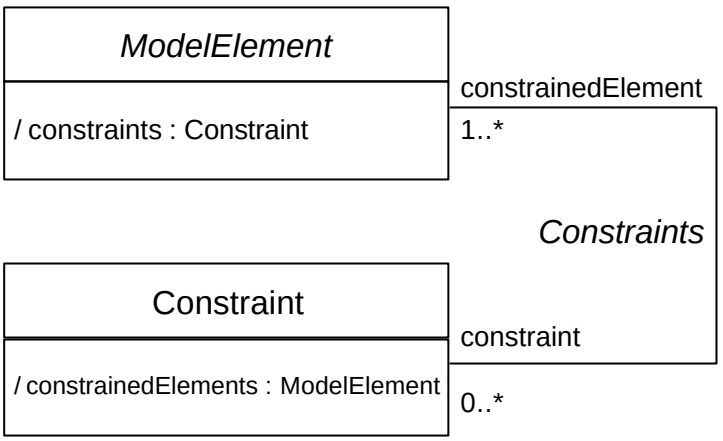


Figure 29 - La relation *Constraints*.

3.1.2.9 La relation entre les éléments et leurs tags (AttachesTo).

Cette relation permet de lier un élément de modélisation aux tags qui le référence. Cette relation lie donc un élément de modélisation (le *modelElement*) a un ensemble de tags (les *tag*). Elle est navigable dans les deux sens. Ce qui permet la définition d'une référence *elements* sur tous les tags de manière à accéder directement à l'élément rattaché.

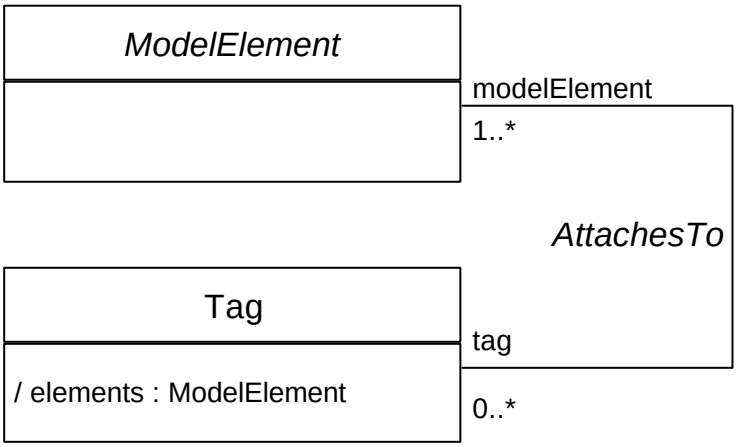


Figure 30 - La relation *AttachesTo*.

3.2 Interopérabilité des outils conformes au MOF.

La standardisation d'un méta-modèle n'est pas suffisante en soi pour permettre l'échange et la communication de modèles entre outils. Pour communiquer, deux outils peuvent s'échanger des modèles de façon "statique" par l'intermédiaire de fichiers ou de façon "dynamique" par l'intermédiaire d'une architecture leur permettant de communiquer (CORBA, COM/DCOM ou encore Java/RMI).

Dans le premier cas, il faut que la syntaxe utilisée pour représenter les modèles soit standardisée de sorte que les deux outils soient à même de produire le fichier pour l'un, et de le lire pour l'autre (c.f. Figure 31).

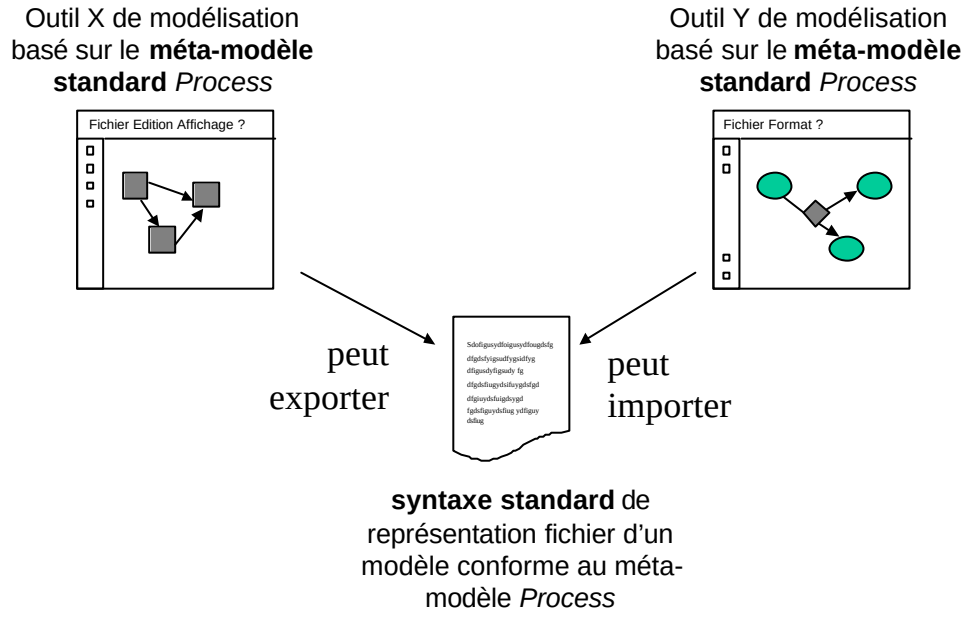


Figure 31 - Interopérabilité via un échange de fichiers.

Tandis que dans le deuxième cas, il faut que les interfaces (ou API : application programming interfaces) de manipulation soient standardisées de manière à ce que l'outil qui accède au modèle via ces interfaces le fasse de la façon la plus indépendante possible de l'outil qui les implémente (c.f. Figure 32).

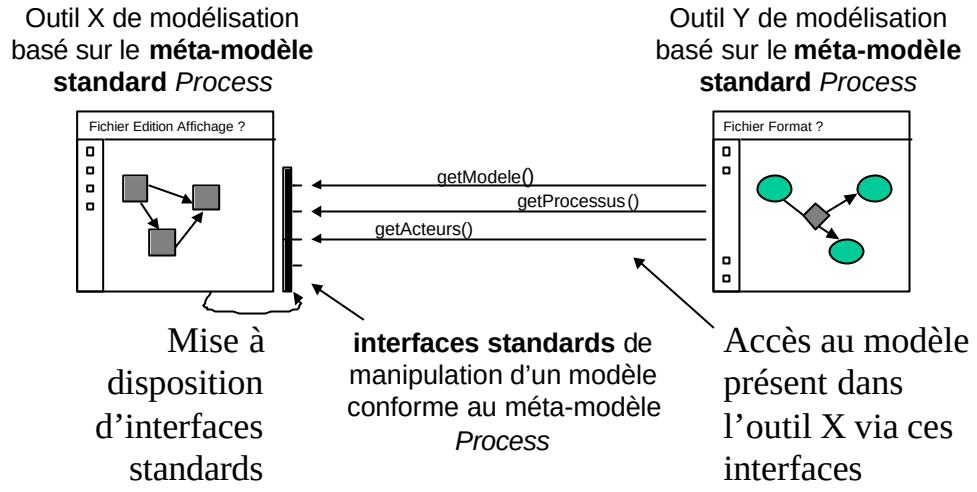


Figure 32 - Interopérabilité par communication via des interfaces standards.

Dans le cas ou les outils que l'on souhaite faire communiquer n'utilisent pas le même méta-modèle ou n'utilisent pas de méta-modèles standardisés, il est toujours possible de se rapprocher de ces standards en utilisant des composants appelés "adapteurs". Ces composants s'intercalent alors entre les applications comme indiqué sur la figure suivante (Figure 33) :

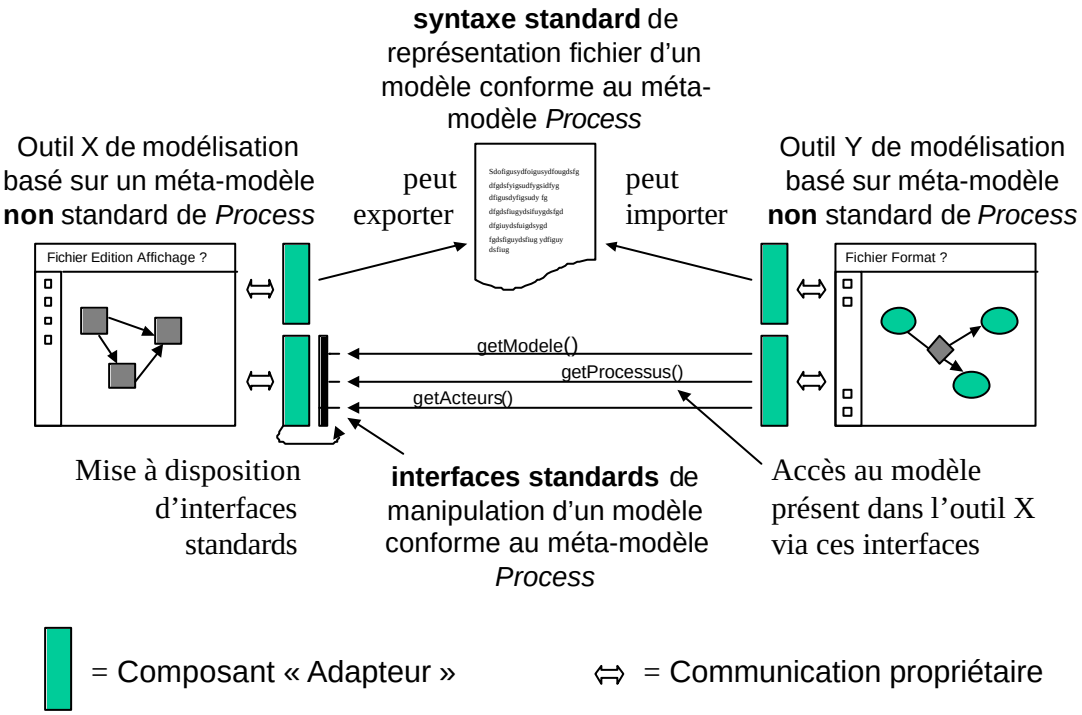


Figure 33 - Interopérabilité via des composants "Adapteurs".

Tous ces mécanismes permettent aux outils de s'échanger des modèles sans dépendre directement les uns des autres.

Il est donc nécessaire de standardiser des interfaces ou une syntaxe pour permettre aux outils de s'échanger ces modèles. Plutôt que de redéfinir ces nouveaux standards pour chaque nouveau méta-modèle, ce sont des règles d'obtention de ces standards qui ont été définies. Ainsi, ils sont obtenus par application, sur le méta-modèle, de règles de mapping (mise en correspondance) basées sur le MOF. Deux ensembles de règles ont ainsi été définies. Le premier ensemble de règles permet, à partir d'un méta-modèle MOF, d'obtenir l'ensemble des interfaces IDL (Interface Definition Language : langage de définition d'interface utilisé notamment par CORBA) nécessaires à la manipulation d'un modèle conforme à ce méta-modèle. Le second ensemble de règles permet, à partir de ce même méta-modèle MOF, d'obtenir le format du fichier qui sera capable de contenir un modèle conforme à ce méta-modèle. Ce format est basé sur le formalisme XML, et l'ensemble de règle se contente alors de produire une description de type DTD à partir du méta-modèle (c.f. chapitre 2.2.2).

La standardisation d'un méta-modèle est alors suffisante pour donner aux outils un moyen d'interopérer. En effet, il suffit d'appliquer les règles de mapping vers IDL ou vers XML pour obtenir le format standard des interfaces ou de la structure fichier d'un modèle conforme à ce méta-modèle. Nous allons présenter succinctement dans ce chapitre ces deux ensembles de règles en les appliquant au méta-modèle MOF suivant (c.f. Figure 34) :

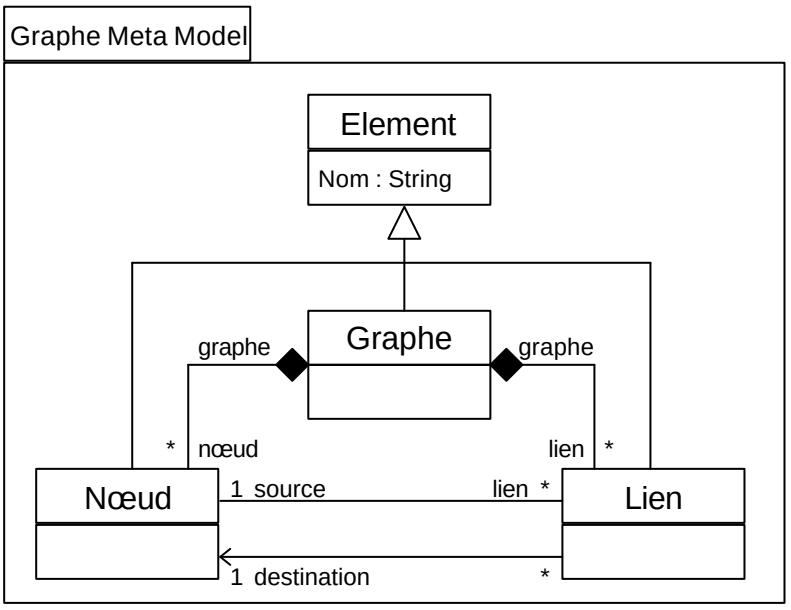


Figure 34 - Un méta-modèle MOF permettant la modélisation de graphes simples.

Ce méta-modèle permet de représenter des graphes constitués de nœuds et de liens nommés. Les liens sont uni-directionnels. L'exemple de graphe présenté ci-dessous (c.f. Figure 35) peut ainsi être modélisé à l'aide de ce méta-modèle.

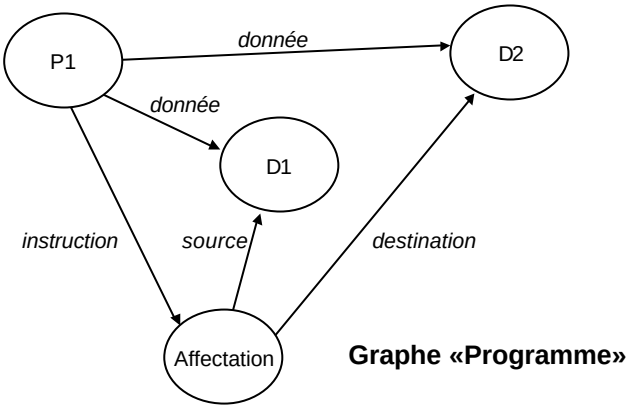


Figure 35 - Un graphe appelé "Programme" et représentant une instruction d'affectation d'un programme.

Un tel graphe peut également être représenté sous la forme d'un diagramme d'instances en utilisant la notation graphique UML associée au méta-modèle MOF précédent (c.f. Figure 36). Sur cette figure, les noms des relations n'ont pas été indiqués pour ne pas surcharger le diagramme :

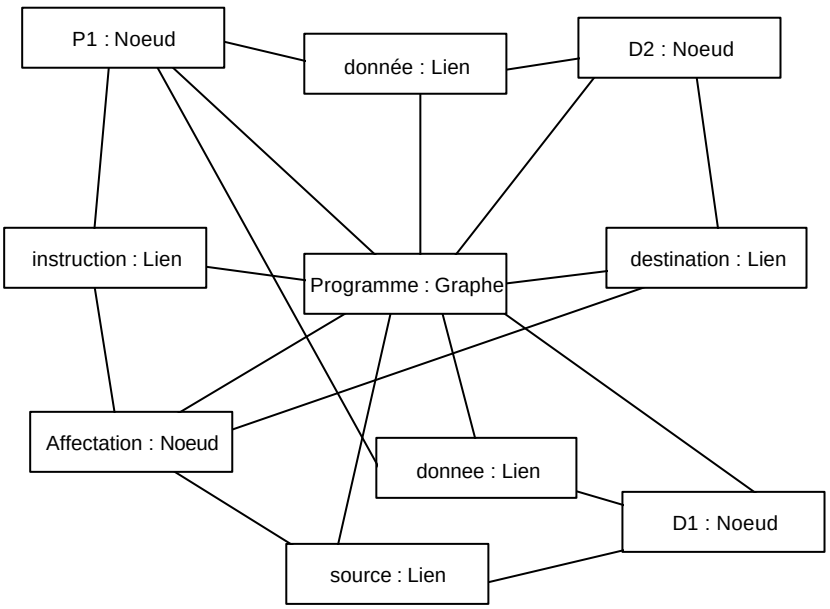


Figure 36 - Graphe précédent représenté avec la notation UML (diagramme d'instances).

A partir de cet exemple, le chapitre 3.2.1 va présenter le mapping MOF/IDL permettant d'obtenir les interfaces de manipulation d'un méta-modèle MOF. Le chapitre 3.2.2 va quant à lui présenter le mapping MOF/XML en décrivant les règles permettant d'obtenir un fichier XML à partir d'un modèle conforme à ce méta-modèle MOF.

3.2.1 Correspondance MOF-IDL : Interopérabilité via des interfaces.

Le rôle des interfaces issues de ce mapping est de permettre la manipulation d'un modèle "instance" d'un méta-modèle MOF. Les entités que nous manipulons alors sont des instances de classes MOF, des instances de relations (ou associations dans les termes du MOF) ainsi que des instances de paquetages.

Afin de permettre une manipulation générique de ces différents éléments, les interfaces obtenues par le mapping se basent sur quatre interfaces définies dans un paquetage appelé "reflective" et décrites sur la figure suivante (c.f. Figure 37) :

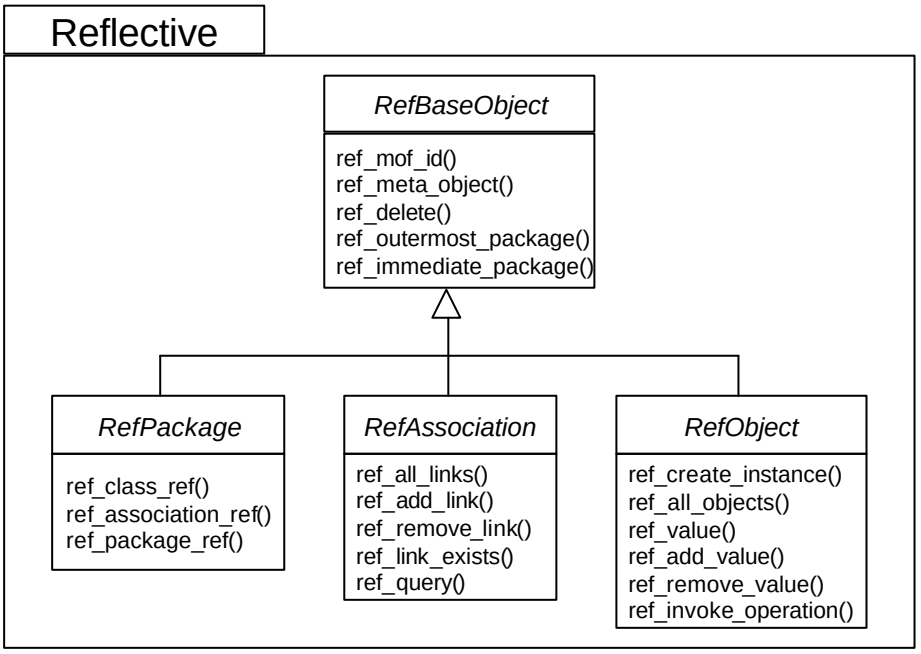


Figure 37 - Le paquetage "reflective" du MOF 1.3.

Ces interfaces sont alors suffisantes pour parcourir, construire ou lancer des opérations sur un modèle "instance" d'un méta-modèle MOF. Il suffit alors que l'outil qui souhaite donner l'accès aux modèles qu'il integre fournisse ces différentes interfaces.

Dans la pratique, ces interfaces ne sont pas suffisantes et il faut également utiliser les interfaces de manipulation d'un méta-modèle MOF (ces interfaces sont l'application de ces règles de mapping sur le MOF lui-même). En effet, si l'on souhaite par exemple obtenir la valeur de l'attribut nom d'une entité x de type Nœud, il faudra passer par le code suivant :

```
RefObject x = …
// …on suppose que c'est l'outil qui nous a renvoyé une référence sur un nœud.
// il faut maintenant passer par la méthode ref_value(…) pour obtenir la valeur de l'attribut "nom",
// mais cette méthode prend en paramètre un RefObject plutôt qu'une chaîne de caractères.
// il faut donc d'abord trouver la méta-entité de type Attribute (sous-type de RefObject et définie dans le paquetage Model du MOF) désignant l'attribut "nom" de la classe Nœud.
// Le schéma classique pour obtenir cette information est le suivant :

Model.Class classeNoeud = Class.narrow(x.meta_object());
// la méthode meta_object invoquée sur x nous renvoie une référence sur la classe MOF Nœud.

Model.Attribute attributNom = Attribute.narrow(classeNoeud.lookup_element("nom"));
// La méthode lookup_element invoquée sur la classe renvoie une référence sur l'entité de type Attribute désignant l'attribut nom de la classe Nœud.
// Attribute est alors une sous-classe de la classe RefObject utilisable pour obtenir la valeur de l'attribut nom du nœud x :

Any nom = x.ref_value(attributNom);
```

Il est probable que dans une prochaine version des spécifications du MOF, ce type de méthode utilise un paramètre de type String. On pourra alors écrire simplement le code suivant pour obtenir la valeur de l'attribut "nom" d'une entité de type Nœud en utilisant simplement cette interface RefObject :

```
Any nom = x.ref_value("nom");
```

Ces mécanismes permettent donc, plus ou moins simplement, de manipuler un modèle quelque soit son méta-modèle via ces interfaces réflexives et les interfaces de manipulation de méta-modèle MOF. Maintenant, lorsqu'un méta-modèle est standardisé, on souhaite disposer

d'interfaces de manipulation d'un modèle plus simple à mettre en œuvre que ces interfaces génériques.

Le mapping MOF vers IDL définit donc les règles d'obtention des interfaces IDL nécessaires à la manipulation d'un modèle. De plus, ces interfaces héritent des interfaces réflexives que l'on vient de présenter. La figure suivante présente l'ensemble des interfaces obtenues par application de ces règles de mapping sur une partie du méta-modèle présenté précédemment :

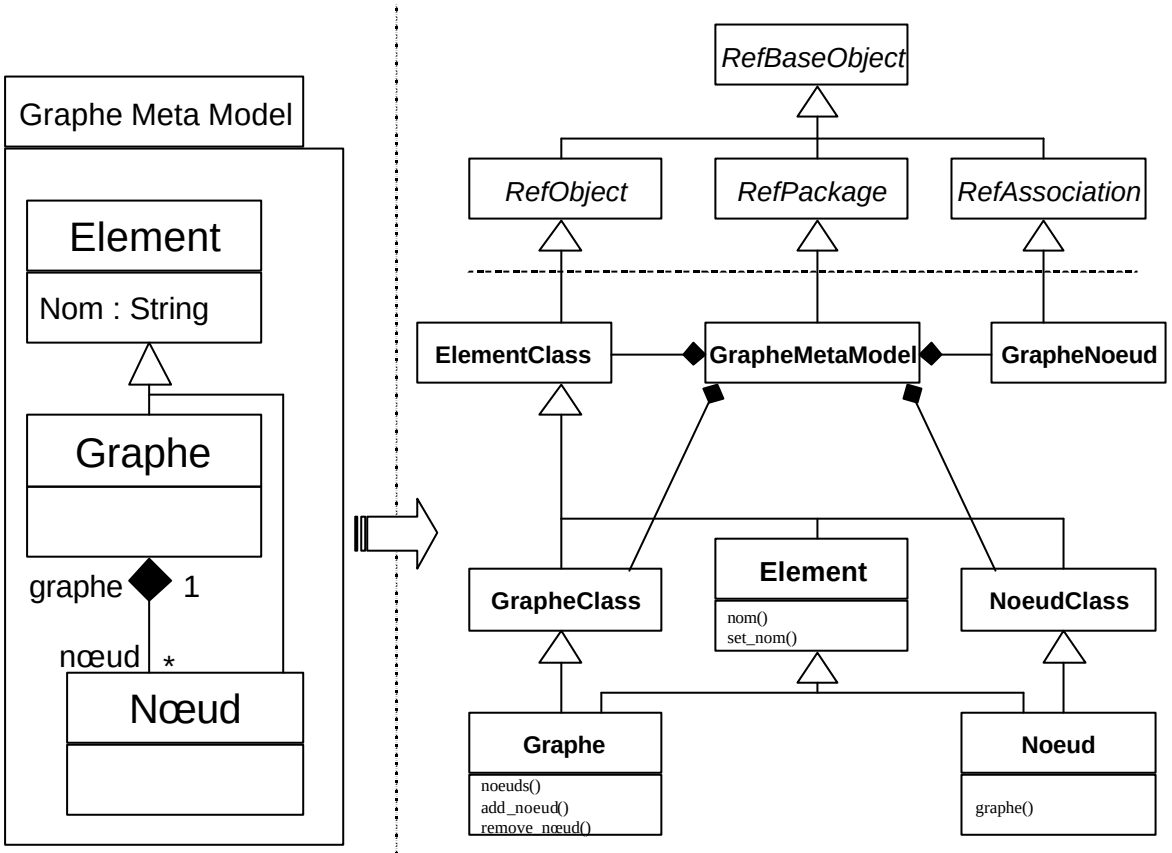


Figure 38 - Obtention des interfaces IDL à partir du méta-modèle décrit Figure 34.

A l'aide de ces interfaces, l'obtention du nom d'un nœud peut alors s'effectuer simplement de la façon suivante :

```
Nœud x = …
// …on suppose toujours que c'est l'outil qui nous a renvoyé une référence sur un nœud.
String nom = x.nom();
```

Les interfaces **Graphe**, **Nœud** et **Element** permettent la manipulation des entités de ces même types. L'interface **GrapheNoeud**, dont chaque instance désigne une relation entre un nœud et son graphe, permet la création et la suppression de ces liens. Les interfaces **GrapheClass**, **NoeudClass** et **ElementClass** peuvent être considérées comme des interfaces regroupant les méthodes et attributs de classe de chacun de ces types. Ce sont ainsi ces classes qui définissent les méthodes createGraphe, createNoeud et createElement permettant respectivement la création d'un nouveau graphe, d'un nouveau nœud ou d'un nouvel élément. L'interface **GrapheMetaModel** permet quant à elle la manipulation d'un modèle conforme au méta-modèle définit par le paquetage qui lui correspond. Elle donne notamment accès, via les méthodes element_ref, graphe_ref et nœud_ref, aux implémentations des interfaces **ElementClass**, **GrapheClass** et **NoeudClass** permettant de créer de nouveaux graphes, nœuds et éléments dans ce modèle.

Ainsi, lorsque l'on sait qu'un outil suit ce mapping, on sait que l'on peut manipuler une entité de type <X> via l'interface <X>, créer une entité de type <Y> via la méthode create<Y> de l'interface <Y>Class, ou encore manipuler un modèle <M> via l'interface <M>.

Le chapitre suivant présenter le mapping MOF/XML permettant d'échanger des modèles conformes au MOF via des fichiers texte au format XML.

3.2.2 Correspondance MOF-XML (XMI) : Interopérabilité par échange de fichier.

De la même façon qu'il est possible de définir des règles génériques d'obtention d'interfaces pour la manipulation d'un modèle, il est possible de définir des règles d'obtention d'un fichier au format DTD (c.f. chapitre 2.2.2) décrivant la structure d'un fichier texte contenant un tel modèle. C'est le rôle du mapping appelé XMI (XML Metadata Interchange). L'application de ce mapping sur les deux méta-modèles conformes au MOF les plus usités (le MOF lui-même permettant la représentation de méta-modèles et UML permettant la représentation de modèles à objets) a donc donné deux formats : XMI UML et XMI MOF. Le premier permet l'échange de modèles UML tandis que le second permet l'échange de méta-modèles MOF.

Pour pouvoir échanger des modèles correspondant au méta-modèle MOF présenté Figure 34, nous lui avons appliqué ces règles XMI (via l'outil XMI Toolkit d'IBM) et nous avons ainsi obtenu la DTD suivante :

```
<!-- METAMODEL PACKAGE:  GrapheMetaModel -->
<!ELEMENT GrapheMetaModel ((GrapheMetaModel.Element | Graphe
                    MetaModel.Noeud | GrapheMetaModel.Graphe |
                    GrapheMetaModel.Lien)*) >

<!-- METAMODEL CLASS:  Element -->
<!ELEMENT GrapheMetaModel.Element.nom (#PCDATA)* >
<!ELEMENT GrapheMetaModel.Element (GrapheMetaModel.Element.nom?)? >

<!-- METAMODEL CLASS:  Noeud -->
<!ELEMENT GrapheMetaModel.Noeud.graphe (GrapheMetaModel.Graphe)? >
<!ELEMENT GrapheMetaModel.Noeud.lien (GrapheMetaModel.Lien)* >
<!ELEMENT GrapheMetaModel.Noeud (GrapheMetaModel.Element.nom?,
                        GrapheMetaModel.Noeud.graphe?,
                        GrapheMetaModel.Noeud.lien*)? >

<!-- METAMODEL CLASS:  Graphe -->
<!ELEMENT GrapheMetaModel.Graphe (GrapheMetaModel.Element.nom?,
                         GrapheMetaModel.Graphe.noeud*,
                         GrapheMetaModel.Graphe.lien*)? >
<!ELEMENT GrapheMetaModel.Graphe.noeud (GrapheMetaModel.Noeud)* >
<!ELEMENT GrapheMetaModel.Graphe.lien (GrapheMetaModel.Lien)* >

<!-- METAMODEL CLASS:  Lien -->
<!ELEMENT GrapheMetaModel.Lien.graphe (GrapheMetaModel.Graphe)? >
<!ELEMENT GrapheMetaModel.Lien.destination (GrapheMetaModel.Noeud)? >
<!ELEMENT GrapheMetaModel.Lien.source (GrapheMetaModel.Noeud)? >
<!ELEMENT GrapheMetaModel.Lien (GrapheMetaModel.Element.nom?,
                        GrapheMetaModel.Lien.graphe?,
                        GrapheMetaModel.Lien.destination?,
                        GrapheMetaModel.Lien.source?)? >
```

Cette DTD permet alors la représentation XML de modèles conformes au méta-modèle de graphe donné en entrée. Cette description n'est pas complète car nous avons oté un certain nombre d'éléments concernant les extensions de DTD XMI et nous avons supprimé l'en-tête contenant une définition d'éléments fixes et présents dans tout fichier décrivant une DTD XMI.

Nous avons simplement laissé les éléments obtenus à partir du méta-modèle que nous avons indiqué en entrée de manière à ce que cette description soit plus claire.

En fait, pour chaque définition d'élément XML, il y a une définition identique de ses attributs (au sens XML) qui est générée de la façon suivante (présentée ici avec l'exemple de l'élément GrapheMetaModel.Graphe.Lien) :

```
<!ATTLIST GrapheMetaModel.Graphe.Lien
        xmi.id          ID #IMPLIED
        xmi.label       CDATA #IMPLIED
        xmi.uuid        CDATA #IMPLIED
        xml:link        CDATA #IMPLIED
        inline          (true | false) #IMPLIED
        actuate         (show | user) #IMPLIED
        href            CDATA #IMPLIED
        role            CDATA #IMPLIED
        title           CDATA #IMPLIED
        show            (embed | replace | new) #IMPLIED
        behavior        CDATA #IMPLIED
        xmi.idref       IDREF  #IMPLIED
        xmi.uuidref     CDATA #IMPLIED
>
```

Les deux attributs principaux sont xmi.id qui correspond à l'identifiant donné à l'élément XML en cours de définition et xmi.idref qui indique l'identifiant de l'élément courant qui est défini dans une autre partie du fichier. Ce second attribut permet de créer des références entre les différents éléments du fichier.

Nous ne nous attarderons pas sur les autres attributs et nous allons présenter maintenant comment le modèle de graphe décrit Figure 35 peut être représenté au format XML en utilisant la DTD ainsi obtenue.

Comme chaque élément doit disposer d'un identifiant, nous allons attribuer respectivement les identifiants e2, e3, e4, e5, e6, e7, e8, e9, e10 et e11 aux éléments Programme, P1, Affectation, D1, D2, instruction, source, destination, donnée (lien de P1 vers D1) et donnée (lien de P1 vers D2).
Le modèle sera quant à lui identifié par e1.

Le contenu du fichier XML décrivant ce modèle sera alors le suivant (la définition complète des nœuds et des liens sera donnée par la suite):

```
<GrapheMetaModel xmi.id='e1'>
  <GrapheMetaModel.Graphe xmi.id = 'e2'>
    <GrapheMetaModel.Element.nom> Programme </GrapheMetaModel.Element.nom>
      <GrapheMetaModel.Nœud xmi.id='e3'>
        <GrapheMetaModel.Element.nom> P1 </GrapheMetaModel.Element.nom>
          …
      </GrapheMetaModel.Nœud>
      <GrapheMetaModel.Nœud xmi.id='e4'>
        <GrapheMetaModel.Element.nom> Affectation </GrapheMetaModel.Element.nom>
          …
      </GrapheMetaModel.Nœud>
      <GrapheMetaModel.Nœud xmi.id='e5'>
        <GrapheMetaModel.Element.nom> D1 </GrapheMetaModel.Element.nom>
      </GrapheMetaModel.Nœud>
      <GrapheMetaModel.Nœud xmi.id='e6'>
        <GrapheMetaModel.Element.nom> D2 </GrapheMetaModel.Element.nom>
      </GrapheMetaModel.Nœud>
      <GrapheMetaModel.Lien xmi.id='e7'>
        <GrapheMetaModel.Element.nom> instruction </GrapheMetaModel.Element.nom>
        …
      </GrapheMetaModel.Lien>
      <GrapheMetaModel.Lien xmi.id='e8'>
        <GrapheMetaModel.Element.nom> source </GrapheMetaModel.Element.nom>
        …
      </GrapheMetaModel.Lien>
      <GrapheMetaModel.Lien xmi.id='e9'>
        <GrapheMetaModel.Element.nom> destination </GrapheMetaModel.Element.nom>
        …
      </GrapheMetaModel.Lien>
      <GrapheMetaModel.Lien xmi.id='e10'>
        <GrapheMetaModel.Element.nom> donnée </GrapheMetaModel.Element.nom>
        …
      </GrapheMetaModel.Lien>
      <GrapheMetaModel.Lien xmi.id='e11'>
        <GrapheMetaModel.Element.nom> donnée </GrapheMetaModel.Element.nom>
        …
      </GrapheMetaModel.Lien>
    </GrapheMetaModel.Graphe>
</GrapheMetaModel>
```

Il reste donc à présenter la définition de chacun des nœuds et des liens ainsi que leurs relations respectives. Dans le méta-modèle, nous avons indiqué que seule la relation "source-lien" était navigable à partir d'un nœud. Par conséquent, nous allons retrouver dans la définition de chaque nœud les références des liens dont ils sont la source.

Ainsi, pour le nœud P1, nous avons la définition suivante (les liens instruction et données, identifiés par e7, e10 et e11, ont pour source le nœud P1) :

```
<GrapheMetaModel.Nœud xmi.id='e3'>
  <GrapheMetaModel.Element.nom> P1 </GrapheMetaModel.Element.nom>
  <GrapheMetaModel.Nœud.lien>
    <GrapheMetaModel.Lien xmi.idref='e7'> </GrapheMetaModel.Lien>
    <GrapheMetaModel.Lien xmi.idref='e10'> </GrapheMetaModel.Lien>
    <GrapheMetaModel.Lien xmi.idref='e11'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Nœud.lien>
</GrapheMetaModel.Nœud>
```

De même, pour le nœud Affectation, nous avons la définition suivante (les liens source et destination identifiés par e8 et e9 ont pour source le nœud Affectation) :

```
<GrapheMetaModel.Nœud xmi.id='e4'>
  <GrapheMetaModel.Element.nom> Affectation </GrapheMetaModel.Element.nom>
  <GrapheMetaModel.Nœud.lien>
    <GrapheMetaModel.Lien xmi.idref='e8'> </GrapheMetaModel.Lien>
    <GrapheMetaModel.Lien xmi.idref='e9'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Nœud.lien>
</GrapheMetaModel.Nœud>
```

En ce qui concerne les liens, les deux relations aboutissant respectivement au nœud source et au nœud destination du lien, sont navigables. Par conséquent, nous allons retrouver dans la définition de chaque lien la référence à son nœud source ainsi que celle de son nœud destination.

Ainsi, pour le lien instruction, nous avons la définition suivante (ce lien a pour source le nœud identifié par e3 et pour destination le nœud identifié par e4) :

```
<GrapheMetaModel.Lien xmi.id='e7'>
  <GrapheMetaModel.Element.nom> instruction </GrapheMetaModel.Element.nom>
  <GrapheMetaModel.Lien.source>
    <GrapheMetaModel.Noeud xmi.idref='e3'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.source>
  <GrapheMetaModel.Lien.destination>
    <GrapheMetaModel.Noeud xmi.idref='e4'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.destination>
</GrapheMetaModel.Lien>
```

Les quatre autres liens sont définis de façon similaire. Le lien source est défini de la façon suivante :

```
<GrapheMetaModel.Lien xmi.id='e8'>
  <GrapheMetaModel.Element.nom> source </GrapheMetaModel.Element.nom>
  <GrapheMetaModel.Lien.source>
    <GrapheMetaModel.Noeud xmi.idref='e4'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.source>
  <GrapheMetaModel.Lien.destination>
    <GrapheMetaModel.Noeud xmi.idref='e5'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.destination>
</GrapheMetaModel.Lien>
```

Le lien destination est défini de la façon suivante :

```
<GrapheMetaModel.Lien xmi.id='e9'>
  <GrapheMetaModel.Element.nom> destination </GrapheMetaModel.Element.nom>
  <GrapheMetaModel.Lien.source>
    <GrapheMetaModel.Noeud xmi.idref='e4'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.source>
  <GrapheMetaModel.Lien.destination>
    <GrapheMetaModel.Noeud xmi.idref='e6'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.destination>
</GrapheMetaModel.Lien>
```

Et les deux liens donnée sont définis de la façon suivante :

```
<GrapheMetaModel.Lien xmi.id='e10'>
  <GrapheMetaModel.Element.nom> donnée </GrapheMetaModel.Element.nom>
  <GrapheMetaModel.Lien.source>
    <GrapheMetaModel.Noeud xmi.idref='e3'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.source>
  <GrapheMetaModel.Lien.destination>
    <GrapheMetaModel.Noeud xmi.idref='e5'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.destination>
</GrapheMetaModel.Lien>

<GrapheMetaModel.Lien xmi.id='e11'>
  <GrapheMetaModel.Element.nom> donnée </GrapheMetaModel.Element.nom>
  <GrapheMetaModel.Lien.source>
    <GrapheMetaModel.Noeud xmi.idref='e3'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.source>
  <GrapheMetaModel.Lien.destination>
    <GrapheMetaModel.Noeud xmi.idref='e6'> </GrapheMetaModel.Lien>
  </GrapheMetaModel.Lien.destination>
</GrapheMetaModel.Lien>
```

Il ne reste plus qu'a présenter l'entête de ce fichier qui est similaire à l'entête suivante :

```
<?xml version="1.0" ?>
<!DOCTYPE GrapheMetaModel "GrapheMetaModel.dtd">
```

Nous avons donc montré ici que l'application des règles XMI sur un méta-modèle conforme au MOF permet d'obtenir une DTD capable de supporter la représentation au format XML de modèles issus de ce méta-modèle.

Ce chapitre a donc présenté l'importance de telles règles de "mapping" pour manipuler dynamiquement des modèles via des interfaces **standardisées** ainsi que pour définir un format **standard** d'échange de modèles. De la même façon, d'autres "mappings" vont être définis. Un mapping vers le langage Java est en cours de définition et on peut supposer que des mapping vers des supports de persistance tels que des bases de données relationnelles ou objets vont également voir le jour de manière à standardiser cette gestion de la persistance des modèles.

4 Le formalisme des sNets.

Ce chapitre présente en détails le formalisme que nous avons défini au cours de cette thèse. Ce formalisme est basé sur les réseaux sémantiques auxquels nous avons rajouter les notions de types, d'univers représentant les modèles et d'univers sémantiques représentant les méta-modèles.

4.1 Description du formalisme.

Le formalisme des sNets est basé sur les réseaux sémantiques. Nous y avons ajouté la modularité et le typage, mais sans pour autant nous éloigner de cette base, de telle sorte qu'un modèle sNets puisse toujours être vu comme un simple réseau sémantique. Dans ce qui suit, nous employons indifféremment le terme entité et le terme nœud. En effet, notre formalisme est le cœur d'un framework de modélisation et de méta-modélisation au sein duquel les éléments représentés sont tous des entités. Toutes les entités sont nommées (ce nom pouvant être arbitraire). En terme de représentation graphique, la plus basique consiste à représenter les entités par des cercles avec le nom de l'entité en son centre. Les liens entre les entités sont alors représentés par des flèches nommées entre ces entités. Un exemple de représentation graphique est proposé Figure 39.

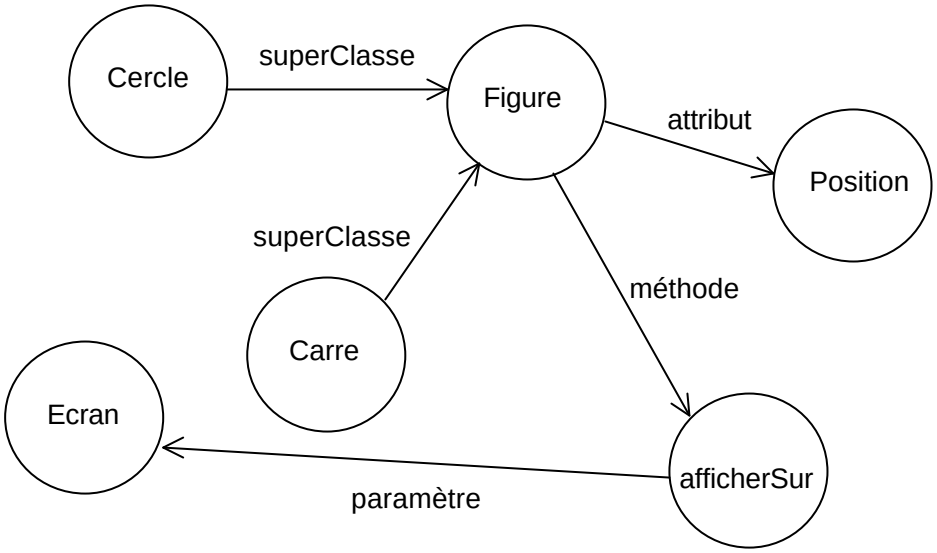


Figure 39 - Un extrait d'une représentation graphique basique d'un sNet.

Afin d'exprimer de façon formelle l'ensemble des règles et propriétés de notre formalisme, nous avons défini un ensemble de formules logiques et de prédicats. Cet ensemble va s'étendre au fur et a mesure de la présentation du formalisme des sNets.

Le premier prédicat que nous pouvons d'or et déjà présenter est celui qui permet de définir une entité sNet. Il se nomme simplement Node et dispose d'un paramètre désignant l'entité (dans ce prédicat, ainsi que dans les prédicats et formules suivants, les variables sont représentées en gras italique) :

- Node(***n***) (p_1)

Ainsi, les entités de la Figure 39 peuvent être représentés formellement par les prédicats suivants :

- Node(Cercle)
- Node(Figure)
- Node(Position)
- Node(Carre)
- Node(Ecran)
- Node(afficherSur)

Le second prédicat que nous pouvons également présenter est celui qui permet de définir un lien sNet. Il se nomme simplement Link et dispose de trois paramètre désignant le lien (les liens sont représentés en italique lorsqu'ils sont représentés dans un prédicat ou une formule logique), l'entité source du lien et l'entité cible du lien :

- Link(***l***, ***source***, ***destination***) (p_2)

Ainsi, les liens de la Figure 39 peuvent être représentés formellement par les prédicats suivants :

- Link(*superClasse* Cercle, Figure)
- Link(*superClasse*, Carre, Figure)
- Link(*attribut*, Figure, Position)
- Link(*méthode*, Figure, afficherSur)
- Link(*paramètre*, afficherSur, Ecran)

Le fait que tout lien lie deux entité peut alors s'exprimer formellement par la formule logique suivante (dans cette formule, ainsi que dans les suivantes, les variables sont représentées en gras italique) :

$$\text{Link}(\boldsymbol{l}, \boldsymbol{s}, \boldsymbol{d}) \Rightarrow (\text{Node}(\boldsymbol{s}) \wedge \text{Node}(\boldsymbol{d})) \qquad (f_1)$$

4.1.1 Mécanisme de nommage des entités.

Pour pouvoir considérer le sNet comme un simple réseau sémantique, il suffit de définir un lien *name* partant de chaque entité et aboutissant sur une entité représentant son nom. Par conséquent, lorsque l'on représente une entité graphiquement par son nom entouré d'un cercle, c'est simplement un raccourci graphique pour ne pas avoir à représenter ce lien *name* et l'entité représentant son nom.(c. f. Figure 40).

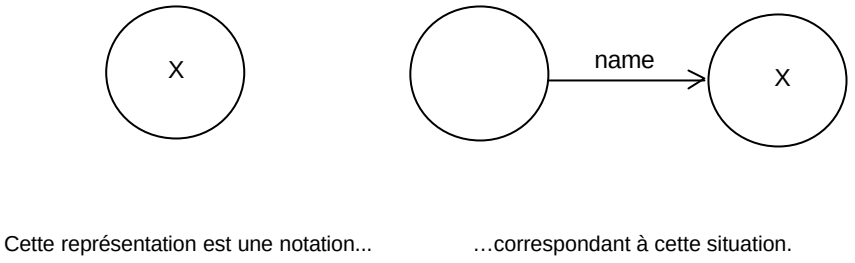


Figure 40 - Le lien *name* permet de définir les nom des entités sNets.

Le fait que toute entité dispose d'un nom peut alors être exprimé formellement par la formule logique suivante :

$$\forall\, x\ \mathrm{Node}(\boldsymbol{x}) \Rightarrow \exists\, \boldsymbol{n}\ \mathrm{Link}(\mathit{name}, \boldsymbol{x}, \boldsymbol{n}) \qquad (f_2)$$

De la même façon, l'unicité de ce lien s'exprime formellement par la formule logique suivante :

$$\forall\, \boldsymbol{x}\ \mathrm{Node}(\boldsymbol{x})\ \mathrm{Link}(\mathit{name}, \boldsymbol{x}, \boldsymbol{n_1})\ \mathrm{Link}(\mathit{name}, \boldsymbol{x}, \boldsymbol{n_2}) \Rightarrow (\boldsymbol{n_1} = \boldsymbol{n_2}) \qquad (f_3)$$

4.1.2 Mécanisme de typage des entités.

Le typage est également introduit dans les sNets sous la forme d'un lien appelé *meta*. Toute entité est liée, via ce lien *meta*, à sa meta-entité (entité représentant son type). D'un point de vue graphique, une entité est représentée par un cercle divisé en deux horizontalement. La partie inférieure contient le nom de l'entité tandis que la partie supérieur contient le nom de son type.

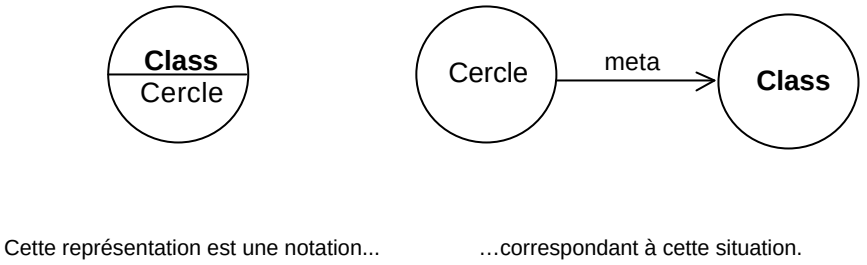


Figure 41 - Représentation graphique du typage des entités.

Les informations représentées par la Figure 39 ont été enrichies avec l'aide notion de typage sur la Figure 42. Les entités sNets présentes sur cette figure sont de quatre type différents, nous avons des classes (Cercle, Figure, Carre), un attribut (Position), une méthode (afficherSur) et un paramètre (Ecran).

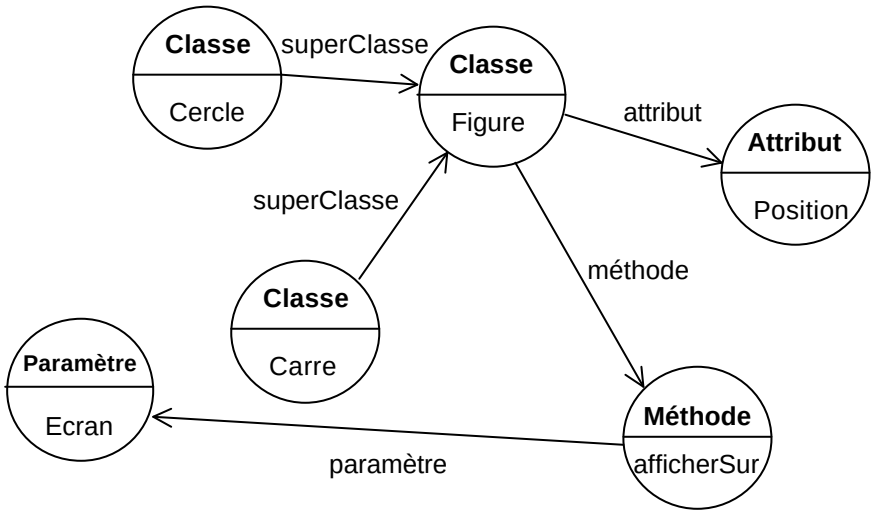


Figure 42 - Un extrait d'une représentation graphique typée d'un sNet.

Nous allons maintenant introduire un prédicat concernant le typage. Ce prédicat s'appelle Type et il prend deux paramètres. Le premier désigne l'entité et le deuxième la méta-entité qui est un type pour cette entité. Nous avons donc le prédicat suivant :

- Type(***x***,***X***) (p_3)

Ainsi, la Figure 41 pourra s'exprimer à l'aide de ce prédicat de la façon suivante :

- Type(Cercle, **Class**)

Et les entités, telles quelles sont représentées sur la Figure 42, nous indiquent que les prédicats suivants sont vérifiés (dans ce qui suit, les types sNets sont représentés en caractères gras):

- Type(Cercle,**Classe**)
- Type(Figure,**Classe**)
- Type(Carre,**Classe**)
- Type(Position,**Attribut**)
- Type(afficherSur,**Méthode**)
- Type(Ecran,**Paramètre**)

Nous pourrions dire que Type(x,y) est équivalent à Link(*meta*, x, y), mais dans un souci d'évolution de notre formalisme, nous préférons dissocier les deux notions et introduire la formule logique suivante :

$$\text{Link}(meta, \boldsymbol{x}, \boldsymbol{X}) \Rightarrow \text{Type}(\boldsymbol{x}, \boldsymbol{X}) \qquad (f_4)$$

En effet, s'il existe un lien *meta*, de ***x*** vers ***X***, c'est que ***X*** désigne le type de ***x***. Par contre, le fait que ***X*** désigne un type pour ***x*** n'implique pas nécessairement qu'il existe un lien *meta* entre ***x*** et ***X*** (nous reviendrons sur ce point par la suite).

Le prédicat Type est également définit avec un seul paramètre. Il indique alors que l'entité sNets désignée en paramètre est une méta-entité. Nous avons donc le prédicat suivant:

- Type(***X***) (p_4)

Ainsi, de la Figure 41, nous pouvons déduire que **Classe** est une méta-entité et l'exprimer à l'aide de ce prédicat de la façon suivante :

- Type(**Classe**)

Nous avons alors la formule logique suivante indiquant que lorsqu'une entité sNets dispose d'un lien meta vers une autre entité sNets, cette deuxième entité est une méta-entité sNets :

$$\text{Link}(meta, \boldsymbol{x}, \boldsymbol{X}) \Rightarrow \text{Type}(\boldsymbol{X}) \qquad (f_5)$$

Et de façon plus générale, nous avons la formule logique suivante :

$$\text{Type}(\boldsymbol{x}, \boldsymbol{X}) \Rightarrow \text{Type}(\boldsymbol{X}) \qquad (f_6)$$

Le fait que toute entité soit typé peut alors être exprimé formellement par la formule logique:

$$\forall\, x\ \text{Node}(\boldsymbol{x}) \Rightarrow \exists\, \boldsymbol{X}\ \text{Type}(\boldsymbol{x}\ , \boldsymbol{X}) \qquad (f_7)$$

Mais cette seule formule ne permet pas d'exprimer que le fait que toute entité sNets dispose d'un lien *meta*. Il est donc nécessaire de définir également la formule logique suivante :

$$\forall\, \boldsymbol{x}\ \text{Node}(\boldsymbol{x}) \Rightarrow \exists\, \boldsymbol{X}\ \ \text{Link}(meta, \boldsymbol{x}, \boldsymbol{X}) \qquad (f_8)$$

De la même façon, l'unicité de ce lien s'exprime formellement par la formule logique suivante :

$$\forall\, \boldsymbol{x}\ \text{Node}(\boldsymbol{x})\ \text{Link}(meta, \boldsymbol{x}, \boldsymbol{n_1})\ \text{Link}(meta, \boldsymbol{x}, \boldsymbol{n_2}) \Rightarrow (\boldsymbol{n_1} = \boldsymbol{n_2}) \qquad (f_9)$$

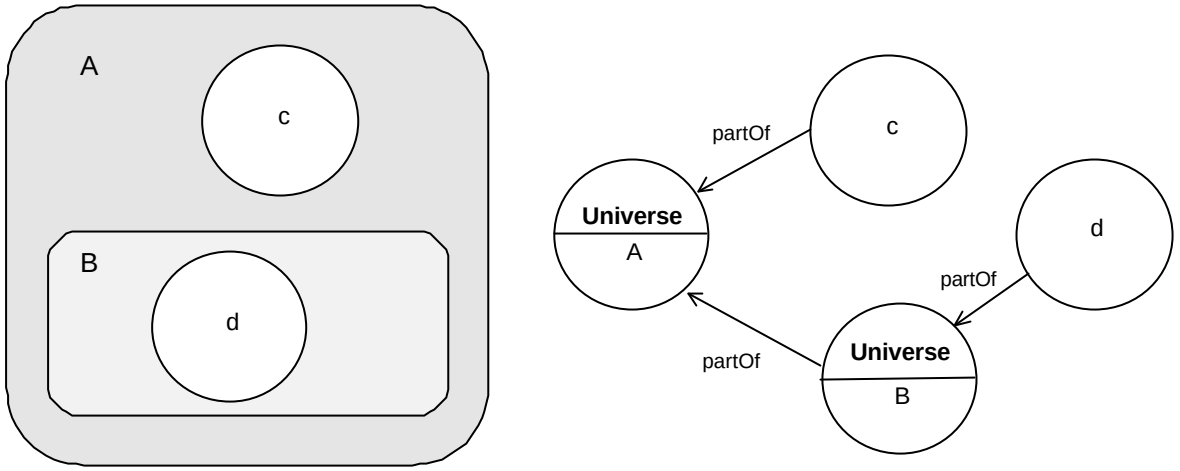
Enfin, les meta-entités qui définissent les types des entités étant également représentés par des entités sNets, elles disposent aussi d'un type, d'un nom et suivent les mêmes règles que celles énoncées précédemment pour les entités.

4.1.3 Mécanisme de modularité.

En plus de la notion de typage, nous avons ajouté la modularité dans le formalisme des sNets par rapports aux simples réseaux sémantiques. Cette notion est définie à partir d'un type d'entité appelé **Universe** et représentant les univers permettant de partitionner le sNets. Une entité est alors rattachée à son univers via un lien appelé *partOf*. Nous avons donc ce nouveau type :

- Type(**Universe**)

La notation graphique utilisée pour représenter les univers est présentée sur la Figure 43.



Cette représentation est une notation...

…correspondant à cette situation.

Figure 43 - Représentation graphique des univers sNets.

Etant donné que toute entité sNets dispose d'un lien *partOf* vers un univers, nous avons défini la formule logique suivante :

$$\forall \boldsymbol{x}\ \mathrm{Node}(\boldsymbol{x}) \Rightarrow \exists \boldsymbol{U}\ \ \mathrm{Type}(\boldsymbol{U}, \mathbf{Universe})\ \mathrm{Link}(\mathit{partOf}, \boldsymbol{x}, \boldsymbol{U}) \qquad (f_{10})$$

De la même façon, l'unicité de ce lien s'exprime formellement par la formule logique suivante :

$$\forall \boldsymbol{x}\ \mathrm{Node}(\boldsymbol{x})\ \mathrm{Link}(\mathit{partOf}, \boldsymbol{x}, \boldsymbol{U_1})\ \mathrm{Link}(\mathit{partOf}, \boldsymbol{x}, \boldsymbol{U_2}) \Rightarrow (\boldsymbol{U_1} = \boldsymbol{U_2}) \qquad (f_{11})$$

Cette définition est valable pour toutes les entités sNets, mais n'est pas valable pour les liens. En effet, un sNets n'étant pas basé sur un hyper-graphe (un hypergraphe permet de lier des liens), nos liens ne peuvent pas disposer de liens *partOf*. Ce mécanisme de modularité n'est donc pas suffisant pour définir l'appartenance d'un lien à un univers. Ce problème est résolu dans le chapitre 4.1.7.

Nous allons maintenant introduire un premier prédicat concernant la modularité. Ce prédicat s'appelle DefinedIn et il prend deux paramètres. Le premier désigne un univers et le deuxième une entité définie dans cet univers. Nous avons donc le prédicat suivant :

- DefinedIn(***entity***, ***universe***) (p_5)

Nous pourrions dire que DefinedIn(x,y) est équivalent à Link(*partOf*, x, y), mais dans un souci d'évolution de notre formalisme, nous préférons dissocier les deux notions et introduire la formule logique suivante :

$$\forall \boldsymbol{x} \; \forall \boldsymbol{U} \; \text{Link}(\textit{partOf}, \boldsymbol{x}, \boldsymbol{U}) \Rightarrow \text{DefinedIn}(\boldsymbol{x}, \boldsymbol{U}) \qquad (f_{12})$$

De sorte que lorsqu'une entité dispose d'un lien *partOf* vers un univers, c'est qu'elle est définie dans cet univers.

Ainsi, de la Figure 43 pourront être déduits les prédicats suivants :

- DefinedIn(c, A)
- DefinedIn(d, B)
- DefinedIn(B, A)

Un deuxième prédicat concernant la modularité va maintenant être présenté. Ce prédicat s'appelle Contains et il prend deux paramètres. Le premier désigne un univers et le deuxième une entité "contenue" par cet univers. Nous avons donc le prédicat suivant :

- Contains(***universe***,***entity***) (p_6)

Nous pourrions dire que Contains(y,x) est équivalent à DefinedIn(x,y), mais ce nouveau prédicat est transitif. De sorte que si a est contenu dans b et b est contenu dans c alors a est contenu dans c. Nous définissons donc la formule logique suivante :

$$\forall \boldsymbol{x} \; \forall \boldsymbol{U} \; \text{Link}(\textit{partOf}, \boldsymbol{x}, \boldsymbol{U}) \Rightarrow \text{Contains}(\boldsymbol{x}, \boldsymbol{U}) \qquad (f_{13})$$

Ainsi lorsqu'une entité dispose d'un lien *partOf* vers un univers, c'est qu'elle est contenue par cet univers.

La formule logique qui suit permet quant à elle d'exprimer la transitivité de ce prédicat :

$$\text{Contains}(\boldsymbol{U_1}, \boldsymbol{x})\ \text{Contains}(\boldsymbol{U_2}, \boldsymbol{U_1}) \Rightarrow \text{Contains}(\boldsymbol{U_2}, \boldsymbol{x}) \qquad (f_{14})$$

Ainsi, de la Figure 43 pourront être déduits les prédicats suivants :

- Contains(A, c)
- Contains(B, d)
- Contains(A, B)

Mais également :

- Contains(A, d)

Ainsi, dire que A contient B (à l'aide du prédicat Contains) indique que B est directement contenu par A (lien *partOf*) ou bien qu'il est directement contenu par un univers contenu par A.
Tandis que dire que B est défini dans A (à l'aide du prédicat DefinedIn) indique que B est directement défini dans A (lien *partOf*) ou bien qu'il est directement défini dans un univers étendu par A (c.f. chapitre 4.1.6.1) .

4.1.4 Définition d'un type d'entité.

Un type d'entité est défini par une méta-entité dans notre formalisme. Pour définir une méta-entité, on utilise le type sNets prédéfini **EntityType**. Un type sNets est une méta-entité sNets par conséquent nous avons le prédicat suivant :

- Type(**EntityType**)

De plus, une entité dont le type est **EntityType** désigne une méta-entité, par conséquent nous avons la formule logique suivante :

$$\text{Type}(\boldsymbol{x}, \textbf{EntityType}) \Rightarrow \text{Type}(\boldsymbol{x}) \qquad (f_{15})$$

4.1.5 Définition d'un type de lien.

Dans notre formalisme, un type d'entité est donc défini par une entité de type **EntityType**, mais également par l'ensemble des relations qu'une entité de ce type est susceptible d'avoir avec d'autres entités. Afin de définir ces relations, nous avons également un type sNets prédéfini nommé **RelationType**. Un type de lien (ou méta-relation) est donc défini par une entité de ce type définissant un type de relation entre deux type d'entités. Par conséquent nous avons le prédicat suivant :

- Type(**RelationType**)

Ainsi, nous avons la formule logique suivante pour dire que tout lien est défini par une entité de ce type :

$$\text{Link}(\boldsymbol{l}, \boldsymbol{x}, \boldsymbol{y}) \Rightarrow \text{Type}(\boldsymbol{l}, \textbf{RelationType}) \qquad (f_{16})$$

Il reste à définir un mécanisme de mise en correspondances des entités de type **EntityType** définissant les meta-entités sNets avec les entités de type **RelationType** représentant méta-relations.

4.1.5.1 Définition d'un type de lien unidirectionnel.

Afin de définir un type de lien unidirectionnel, nous introduisons donc deux nouveaux liens:

- un lien *outGoing* qui part d'une méta-entité et aboutie à une méta-relation,
- un lien *destination* qui part d'une méta-relation et qui aboutie sur une méta-entité.

Un exemple de leur utilisation est présenté sur la figure suivante :

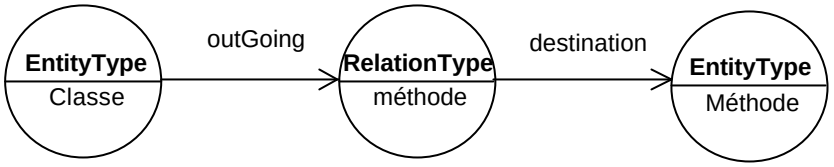


Figure 44 - "Une entité de type Classe est susceptible de disposer de liens *méthode* vers des entités de type Méthode".

Ces deux liens nous permettent de définir un nouveau prédicat nommé LinkType et prenant trois paramètres:

- une méta-relation de type **RelationType** définissant le lien,
- une méta-entité de type **EntityType** définissant le type de la source de ce lien et
- une méta-entité de type **EntityType** définissant le type de sa cible.

Nous avons donc le prédicat suivant :

- LinkType(***Lien***, ***TypeSource***, ***TypeDestination***) (p_7)

Ainsi, la Figure 44 pourra s'exprimer à l'aide de ce prédicat de la façon suivante :

- LinkType(*méthode*, **Classe**, **Méthode**)

Ceci nous permet de définir un lien *méthode* entre une entité de type **Classe** et une entité de type **Méthode** comme représenté sur la Figure 45 :

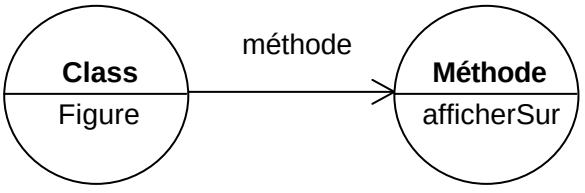


Figure 45 - "La classe Figure dispose d'une méthode afficherSur".

La formule logique suivante est alors définie :

$$\text{Type}(\boldsymbol{x})\ \text{Type}(\boldsymbol{y})\ \text{Link}(\textit{outgoing}, \boldsymbol{x}, \boldsymbol{l})\ \text{Link}(\textit{destination}, \boldsymbol{l}, \boldsymbol{y}) \Rightarrow \text{LinkType}(\boldsymbol{l}, \boldsymbol{x}, \boldsymbol{y}) \quad (f_{17})$$

De plus, un lien entre deux entités sNets ne peut être défini que s'il existe le type de lien correspondant entre les types de ces deux entités. Pour exprimer cette contrainte, nous avons défini un nouveau prédicat nommé LinkTypeDefined. Ce prédicat prends trois paramètres :

- une méta-relation de type **RelationType** indiquant le lien,
- une entité indiquant la source potentielle du lien et
- une entité indiquant la destination potentielle du lien.

Nous avons donc le prédicat suivant :

- $\text{LinkTypeDefined}(\boldsymbol{Lien}, \boldsymbol{source}, \boldsymbol{destination})$ (p_8)

Ce prédicat est alors défini par la formule logique suivante :

$$\text{LinkTypeDefined}(\boldsymbol{l}, \boldsymbol{x}, \boldsymbol{y}) \Leftrightarrow \exists \boldsymbol{T_1} \exists \boldsymbol{T_2}\ \text{Type}(\boldsymbol{x}, \boldsymbol{T_1})\ \text{Type}(\boldsymbol{y}, \boldsymbol{T_2})\ \text{LinkType}(\boldsymbol{l}, \boldsymbol{T_1}, \boldsymbol{T_2})$$

Ainsi, si nous avons un lien l entre une entité x de type T1 et une entité y de type T2, c'est que ce lien est défini entre les types T1 et T2. Cette contrainte est exprimée par la formule logique suivante :

$$\text{Link}(\boldsymbol{l}, \boldsymbol{x}, \boldsymbol{y}) \Rightarrow \text{LinkTypeDefined}(\boldsymbol{l}, \boldsymbol{x}, \boldsymbol{y}) \quad (f_{19})$$

Mais s'il n'existe pas de définition du lien l entre les types de x et de y, alors il ne peut y avoir de lien l entre x et y. Ce qui se traduit par la formule logique suivante :

$$\neg\text{LinkTypeDefined}(\boldsymbol{l}, \boldsymbol{x}, \boldsymbol{y}) \Rightarrow \neg\text{Link}(\boldsymbol{l}, \boldsymbol{x}, \boldsymbol{y}) \quad (f_{20})$$

Maintenant, on veux également être capable de définir la cardinalité maximale d'un lien d'un type donné. Pour ce faire, on a défini un lien *multiplicity* entre la méta-relation définissant le lien

et la valeur de cette multiplicité représentée par un nombre entier (la valeur -1 indiquant une cardinalité maximum infinie).

Ainsi pour spécifier qu'une entité de type **Classe** peut disposer d'un nombre infini de méthodes, on emploiera ce lien *multiplicity* de la façon suivante:

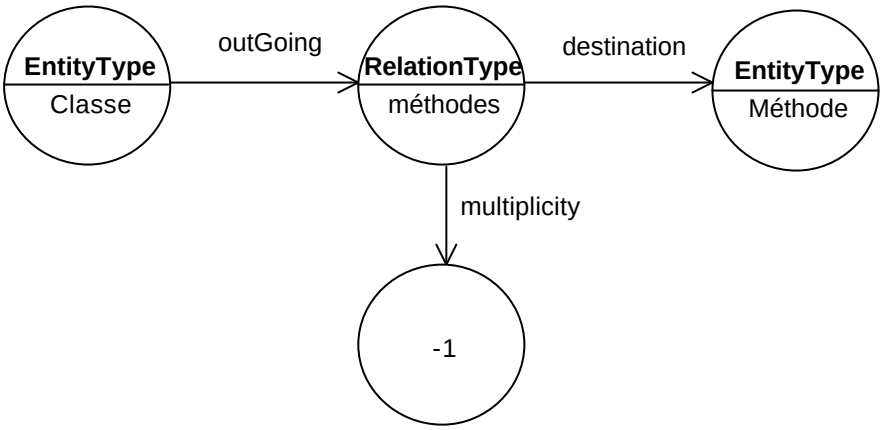


Figure 46 - "Une entité de type Classe est susceptible de disposer d'un nombre infini de liens *méthode* vers des entités de type Méthode".

De même, une relation d'héritage simple entre des entités de type **Classe** se représentera de la façon suivante :

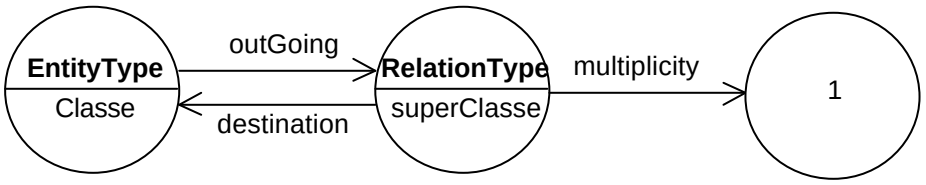


Figure 47 - "Une entité de type Classe est susceptible de disposer d'au plus un lien *superClasse* vers une autre entité de type Classe".

Tandis qu'une relation d'héritage multiple entre des entités de type **Classe** serait représentée de la façon suivante :

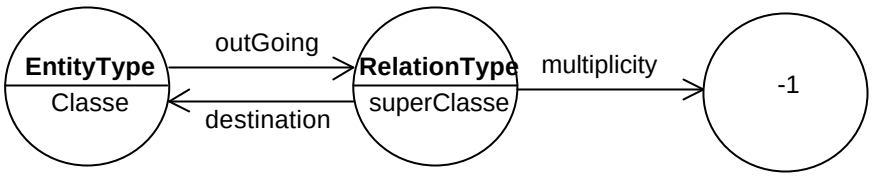


Figure 48 - "Une entité de type Classe est susceptible de disposer d'un nombre infini de liens *superClasse* vers des entités de type Classe".

4.1.5.2 Définition d'un type de lien bidirectionnel.

Etant donné qu'une relation entre deux objets n'est pas nécessairement unidirectionnelle, la seule façon de représenter un relation bidirectionnelle avec les sNets est d'employer deux liens. Ainsi si l'on veux que la relation *superClasse* soit bidirectionnelle, il faut définir un autre lien que l'on appellera par exemple *sousClasse*. On pourra alors avoir le schéma suivant :

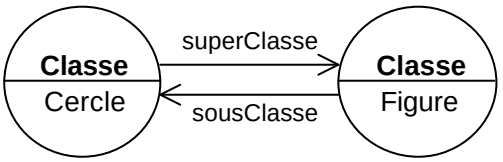


Figure 49 - Utilisation de deux liens pour représenter une relation bidirectionnelle.

Ce lien *sousClasse* doit donc être défini de la même façon que le lien *superClasse* :

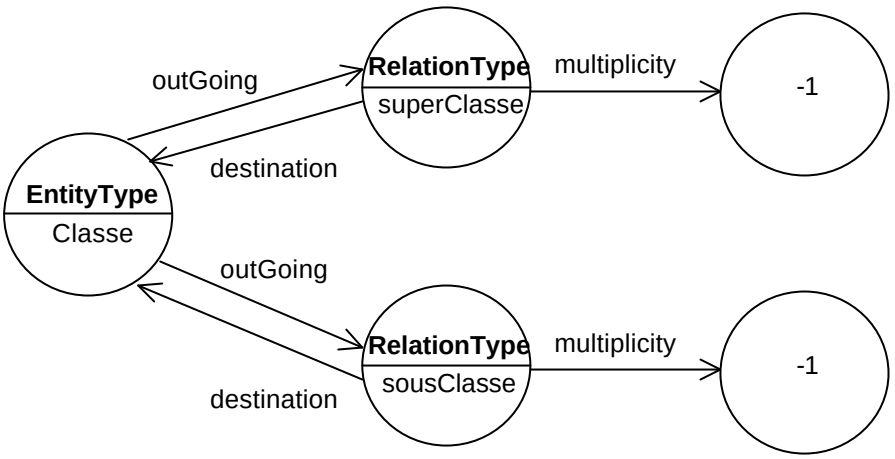


Figure 50 - "Une Classe peut avoir plusieurs sous-classes et plusieurs super-classes".

Il faut maintenant définir un mécanisme qui permette de synchroniser ces deux liens qui constituent une relation bidirectionnelle. En effet, s'il existe un lien superClasse entre une classe A et une classe B, il doit exister également un lien sousClasse entre la classe B et la classe A.

Nous introduisons donc un nouveau lien appelé *inverse* permettant de lier deux méta-relations constituant une relation bidirectionnelle. Ainsi, définition des liens superClasse et sousClasse présentée Figure 50 sera modifiée tel qu'indiqué sur la Figure 51 pour exprimer le fait que ces deux liens forment en fait une relation bidirectionnelle.

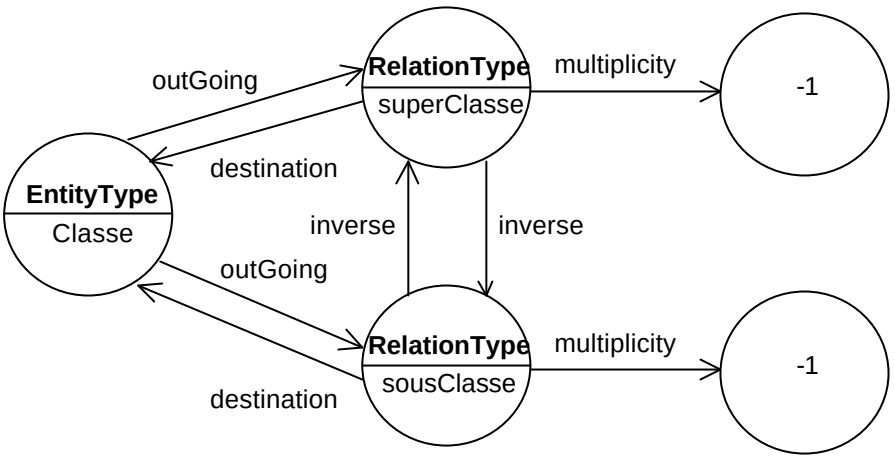


Figure 51 - "Une Classe peut avoir plusieurs sous-classes et plusieurs super-classes & les liens *superClasse* et *sousClasse* définissent une relation bidirectionnelle".

La formule logique suivante est alors définie :

$$\text{Link}(inverse,\ \boldsymbol{r_1},\ \boldsymbol{r_2})\ \text{Link}(\boldsymbol{r_1}, \boldsymbol{n_1}, \boldsymbol{n_2}) \Rightarrow \text{Link}(\boldsymbol{r_2}, \boldsymbol{n_2}, \boldsymbol{n_1}) \qquad (f_{21})$$

De sorte que lorsqu'un lien *inverse* est défini entre deux méta-relations *r1* et *r2* , la création d'un lien *r1* entre deux entités entraîne automatiquement la création d'un lien inverse *r2* entre ces deux entités. De plus, nous avons également la formule logique suivante car si une relation *r1* à pour inverse une relation *r2*, alors cette relation *r2* doit également avoir pour inverse la relation *r1*:

$$\text{Link}(inverse,\ \boldsymbol{r_1},\ \boldsymbol{r_2}) \Rightarrow \text{Link}(inverse,\ \boldsymbol{r_2}, \boldsymbol{r_1}) \qquad (f_{22})$$

4.1.6 Les mécanismes d'extension dans les sNets.

Deux mécanismes d'extension sont définis dans le formalisme des sNets. L'un concerne l'extension des univers et l'autre l'héritage des types.

4.1.6.1 L'extension des univers.

Le mécanisme d'extension des univers permet de factoriser la définition d'un univers. Un univers définissant un ensemble d'entités, il peut être intéressant de factoriser cette définition de sorte que d'autres univers puissent être "vus" comme des extensions de cet univers. Pour pouvoir considérer le sNet comme un simple réseau sémantique, il suffit de définir un lien *extends* partant d'un univers et aboutissant sur le ou les univers qu'il étend.

L'extension d'un univers est donc implémentée de la façon suivante dans le sNet :

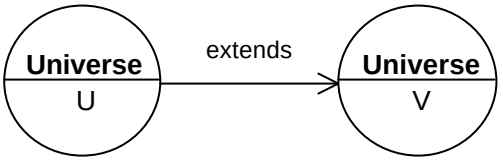


Figure 52 - "L'univers U est une extension de l'univers V".

Nous avons également défini le prédicat suivant :

- Extends(***subUniverse***, ***superUniverse***) (p_9)

Ainsi, la Figure 52 pourra s'exprimer à l'aide de ce prédicat de la façon suivante :

- Extends(U, V)

La formule logique qui suit permet quant à elle d'exprimer la transitivité de ce prédicat :

$$\text{Extends}(\boldsymbol{U_1}, \boldsymbol{U_2})\ \text{Extends}\ (\boldsymbol{U_2}, \boldsymbol{U_3}) \Rightarrow \text{Extends}\ (\boldsymbol{U_1}, \boldsymbol{U_3}) \qquad (f_{23})$$

Toutes les entités définies dans l'univers V sont alors également définis dans l'univers U. La formule logique qui suit permet quant à elle d'exprimer ce dernier postulat :

$$\text{DefinedIn}(\boldsymbol{x}, \boldsymbol{U_1})\ \ \text{Extends}(\boldsymbol{U_2}, \boldsymbol{U_1}) \Rightarrow \text{DefinedIn}(\boldsymbol{x}, \boldsymbol{U_2}) \qquad (f_{24})$$

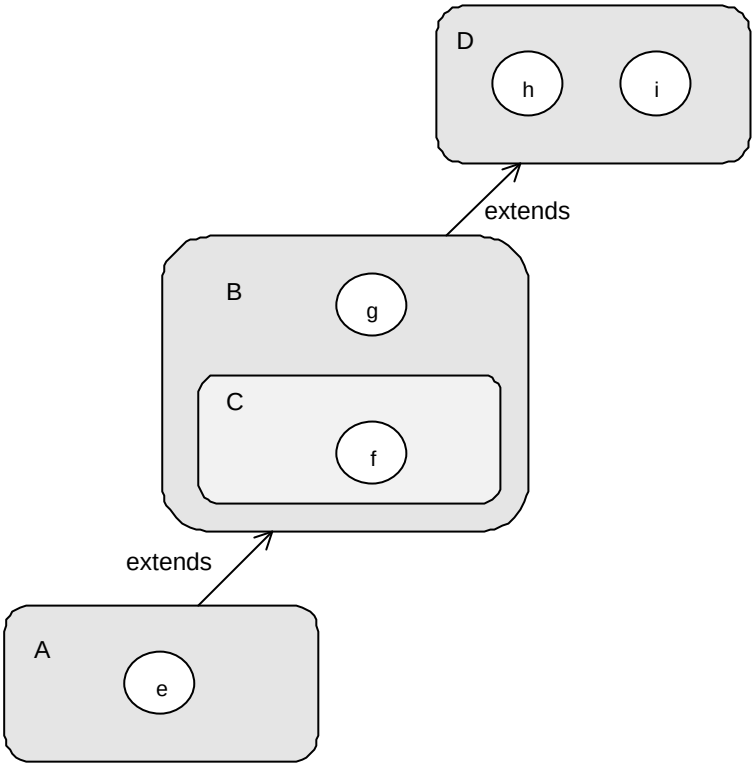


Figure 53 - Exemple d'utilisation de l'extension d'univers.

La Figure 53 pourra s'exprimer à l'aide des prédicats DefinedIn, Contains et Extends de la façon suivante :

- Extends(A, B)
- Extends(B, C)
- Extends(A, C) (par transitivité de l'extension)
- Contains(D, h)
- Contains(D, i)
- Contains(C, f)
- Contains(B, g)
- Contains(B, C)
- Contains(B, f) (par transitivité du lien de contenance)
- Contains(A, e)
- DefinedIn(h, D)
- DefinedIn(h, B) (par extension)
- DefinedIn(h, A) (par extension)

- DefinedIn(i, D)
- DefinedIn(i, B) (par extension)
- DefinedIn(i, A) (par extension)
- DefinedIn(g, B)
- DefinedIn(g, A) (par extension)
- DefinedIn(f, C)
- DefinedIn(e, A)

4.1.6.2 L'héritage des types.

Le mécanisme d'héritage des types mis en place dans notre formalisme est très proche de l'héritage tel qu'on le trouve dans les différents langages de modélisation et de programmation à objets. Une entité d'un type **X** donné peut être manipulée comme une entité de n'importe lequel des super-types de **X**. Le type **X** profite ainsi de toutes les méta-relations rattachées à ses super-types. Les méta-relations ayant pour source un super-type de **X** peuvent également avoir pour source **X**. De même, les méta-relations ayant pour destination un super-type de **X** peuvent également avoir pour destination un **X**. Pour pouvoir considérer le sNet comme un simple réseau sémantique, il suffit de définir un lien *inherits* partant d'un type et aboutissant sur son ou ses super-types.

L'héritage des types est donc implémentée de la façon suivante dans le sNet :

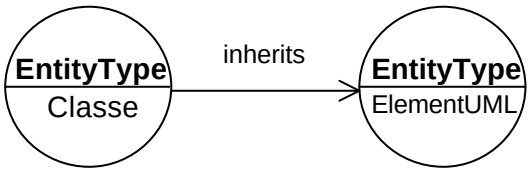


Figure 54 - "Classe est un sous-type de ElementUML".

Nous avons également défini le prédicat suivant :

- Inherits(***subType***, ***superType***) (p_{10})

Ainsi, la Figure 52 pourra s'exprimer à l'aide de ce prédicat de la façon suivante :

- Inherits(**Classe**, **ElementUML**)

La formule logique qui suit permet quant à elle d'exprimer la transitivité de ce prédicat :

$$\text{Inherits}(\boldsymbol{U_1}, \boldsymbol{U_2})\ \text{Inherits}\ (\boldsymbol{U_2}, \boldsymbol{U_3}) \Rightarrow \text{Inherits}\ (\boldsymbol{U_1}, \boldsymbol{U_3}) \qquad (f_{25})$$

Toutes les relations définies pour les éléments de type **ElementUML** sont alors également définies pour les éléments de type **Classe**. Les deux formules logiques suivantes permettent quant à elles d'exprimer ce dernier postulat :

$$\text{LinkType}(\boldsymbol{l}, \boldsymbol{T_2}, \boldsymbol{T_3})\ \ \text{Inherits}(\boldsymbol{T_1}, \boldsymbol{T_2}) \Rightarrow \text{LinkType}(\boldsymbol{l}, \boldsymbol{T_1}, \boldsymbol{T_3}) \qquad (f_{26})$$

Cette formule permet de dire que lorsqu'un type de lien $\boldsymbol{l}$ est défini avec pour source $\boldsymbol{T_2}$ et que $\boldsymbol{T_1}$ est un sous-type de $\boldsymbol{T_2}$, alors ce type de lien $\boldsymbol{l}$ est également défini avec pour source $\boldsymbol{T_1}$. Il en est de même pour les types destination :

$$\text{LinkType}(\boldsymbol{l}, \boldsymbol{T_2}, \boldsymbol{T_3})\ \ \text{Inherits}(\boldsymbol{T_1}, \boldsymbol{T_3}) \Rightarrow \text{LinkType}(\boldsymbol{l}, \boldsymbol{T_2}, \boldsymbol{T_1}) \qquad (f_{27})$$

Cette formule permet de dire que lorsqu'un type de lien $\boldsymbol{l}$ est défini avec pour destination $\boldsymbol{T_3}$ et que $\boldsymbol{T_1}$ est un sous-type de $\boldsymbol{T_3}$, alors ce type de lien $\boldsymbol{l}$ est également défini avec pour destination $\boldsymbol{T_3}$.

La Figure 55 présente une utilisation de la notion d'héritage. L'exemple choisi définit un sous-ensemble des types nécessaires à la représentation d'un modèle UML à l'aide du formalisme des sNets.

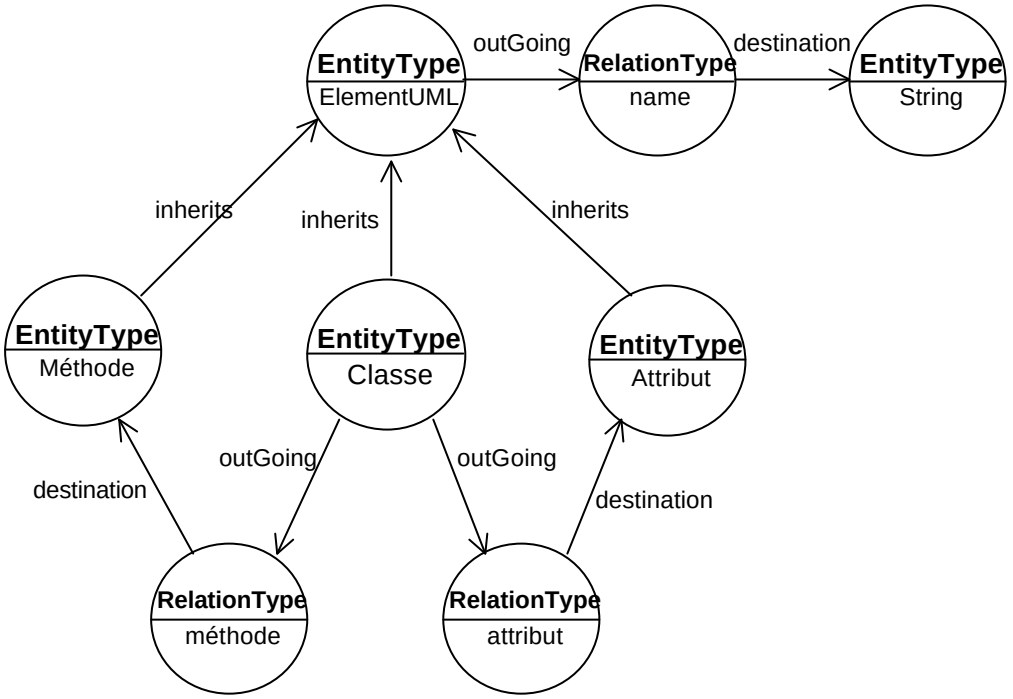


Figure 55 - "Exemple d'utilisation de l'héritage des types sNets".

Cette figure pourra s'exprimer à l'aide des prédicats Type, LinkType et Inherits de la façon suivante :

- Type(**ElementUML**)
- Type(**Classe**)
- Type(**Méthode**)
- Type(**Attribut**)
- Inherits(**Méthode**, **ElementUML**)
- Inherits(**Classe**, **ElementUML**)
- Inherits(**Attribut**, **ElementUML**)
- LinkType(*name*, **ElementUML**, **String**)
- LinkType(*name*, **Classe**, **String**) (par héritage)
- LinkType(*name*, **Méthode**, **String**) (par héritage)

- LinkType(*name*, **Attribut**, **String**) (par héritage)
- LinkType(*méthode*, **Classe**, **Méthode**)
- LinkType(*attribut*, **Classe**, **Attribut**)

4.1.7 Appartenance d'un lien à un univers.

Comme indiqué précédemment au chapitre 4.1.3, l'appartenance d'une entité à un univers se présente sous la forme d'un lien *partOf* entre l'entité et son univers. Pour ce qui est des liens, c'est plus délicat. En effet, comme le formalisme des sNets est basé sur le formalisme des réseaux sémantiques et pas celui des hypergraphes, il n'est pas possible de définir de lien entre autre chose que des entités.

Nous avons donc fait le choix de "rattacher" un lien sNets soit à son entité source, soit à son entité cible. Dans le premier cas, le lien appartient à l'univers de son entité source et dans le second à l'univers de son entité cible. Ainsi un lien, de la même façon qu'une entité sNets, n'appartient bien qu'à un seul univers. Il reste à définir la règle indiquant si un lien doit être rattaché à sa source ou à sa cible.

Un lien rattaché à sa source sera dit "directement valide" tandis qu'un lien rattaché à sa cible sera dit "indirectement valide".

Nous introduisons donc les prédicats suivants :

- Valid(***l***, ***a***, ***b***) (p_{11})
- DirectValid(***l***, ***a***, ***b***) (p_{12})
- IndirectValid(***l***, ***a***, ***b***) (p_{13})

Le premier prédicat indique qu'un lien ***l*** de l'entité ***a*** vers l'entité ***b*** est (ou serai dans le cas ou un tel lien n'existe pas) valide. Le second prédicat indique qu'un lien ***l*** de l'entité ***a*** vers l'entité ***b*** est (ou serai dans le cas ou un tel lien n'existe pas) directement valide. Et le dernier prédicat indique qu'un lien ***l*** de l'entité ***a*** vers l'entité ***b*** est (ou serai dans le cas ou un tel lien n'existe pas) indirectement valide.

4.1.7.1 Les liens dits "directement valides".

Un lien *l* pouvant être défini entre une entité a définie dans un univers U et une entité b définie dans un univers V est dit directement valide de a vers b si U et V désignent le même univers ou si U est une extension de V. Cette contrainte est exprimée par la formule logique suivante :

$$\text{LinkTypeDefined}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b})\ \text{DefinedIn}(\boldsymbol{a}, \boldsymbol{U})\ \text{DefinedIn}(\boldsymbol{b}, \boldsymbol{V}) \wedge (\ \text{Extends}(\boldsymbol{U},\boldsymbol{V}) \vee (\boldsymbol{V}=\boldsymbol{U})\ ) \Rightarrow \text{DirectValid}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b}))$$

(f_{28})

Sur la Figure 56, nous avons présenté deux univers contenant un ensemble d'entités et de liens entre ces entités. Le pré requis pour la définition de ces deux univers est avant tout l'existence des types **Classe**, **Attribut** et **Méthode**, ainsi que des liens *attribut*, *méthode*, *superClasse* et *sousClasse* utilisés sur cette figure. Nous considérons par conséquent que nous avons les prédicats suivants :

- Type(**Classe**)
- Type(**Attribut**)
- Type(**Méthode**)
- LinkType (*attribut*, **Classe**, **Attribut**)
- LinkType (*méthode*, **Classe**, **Méthode**)
- LinkType (*superClasse*, **Classe**, **Classe**)
- LinkType (*sousClasse*, **Classe**, **Classe**)

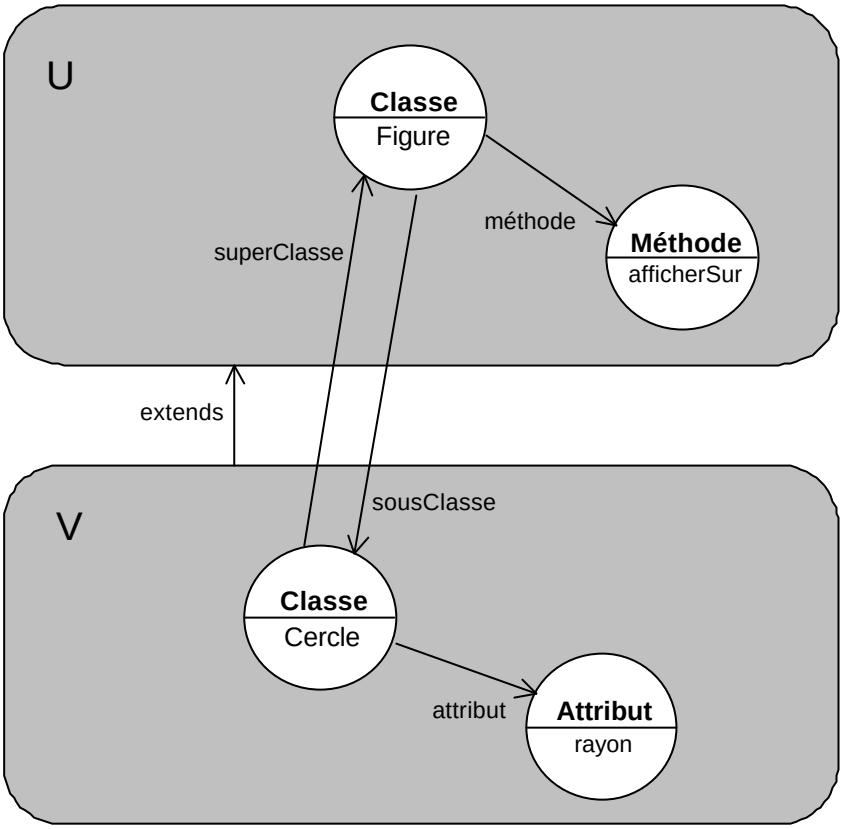


Figure 56 - Exemple de figure contenant des liens "directement valide".

Sur cette figure (Figure 56), nous avons donc les liens :

- Link (*méthode*, Figure, afficherSur)
- Link (*superClasse*, Cercle, Figure)
- Link (*attribut*, Cercle, rayon)
- Link (*sousClasse*, Figure, Cercle)

Pour les trois premiers liens, la formule f_{28} est applicable de sorte que ces trois liens sont directement valides. Il n'en est pas de même pour le lien *sousClasse*. En effet, l'entité Figure et l'entité Cercle étant dans deux univers différents, il faudrait que l'univers de l'entité Figure soit une extension de celui de l'entité Cercle pour que ce lien *sousClasse* soit directement valide.

Par conséquent, comme indiqué précédemment, les liens directement valides sont rattachés à leur entité source et appartiennent à l'univers de cette entité. Cette règle est être exprimée par la formule logique suivante :

$$\text{Link}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b})\ \text{DirectValid}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b})\ \text{Contains}(\boldsymbol{U}, \boldsymbol{a}) \Rightarrow \text{Contains}(\boldsymbol{U}, \boldsymbol{l}) \qquad (f_{29})$$

Cette règle nous permet donc de déterminer que les liens *méthode*, *superClasse* et *attribut* appartiennent respectivement aux univers des entités Figure, Cercle et Cercle. Nous avons donc :

- Contains (U, *méthode*)
- Contains (V, *superClasse*)
- Contains (V, a*ttribut*)

4.1.7.2 Les liens dits "indirectement valides".

Un lien *l* pouvant être défini entre une entité a et une entité b est dit indirectement valide de a vers b si et seulement si ce lien *l* dispose d'un lien inverse l_2 directement valide de b vers a. Cette contrainte est exprimée par la formule logique suivante :

$$\text{DirectValid}(\boldsymbol{l_2}, \boldsymbol{b}, \boldsymbol{a})\ \text{Link}(\textit{inverse}, \boldsymbol{l}, \boldsymbol{l_2}) \Rightarrow \text{IndirectValid}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b}) \qquad (f_{30})$$

Par conséquent, comme indiqué précédemment, les liens indirectement valides sont rattachés à leur entité cible et appartiennent à l'univers de cette entité. Cette règle est être exprimée par la formule logique suivante :

$$\text{Link}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b})\ \text{IndirectValid}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b})\ \text{Contains}(\boldsymbol{U}, \boldsymbol{b}) \Rightarrow \text{Contains}(\boldsymbol{U}, \boldsymbol{l}) \qquad (f_{31})$$

Ces deux règles nous permettent donc de déterminer que le lien *sousClasse* présenté sur la Figure 56 est dit "indirectement valide" et qu'il appartient par conséquent à l'univers de sa cible, c'est à dire V. Nous avons donc :

- Contains (V, *sousClasse*)

4.1.7.3 Validité des liens sNets.

Dans notre formalisme, tout lien est valide. Deux conditions doivent être remplies pour qu'un lien soit valide :

- il doit être défini par une entité de type **RelationType**,
- il doit être soit directement valide, soit indirectement valide.

Cette contrainte est exprimée par la formule logique suivante :

$$\text{Valid}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b}) \Leftrightarrow \text{LinkTypeDefined}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b}) \wedge (\text{DirectValid}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b}) \vee \text{IndirectValid}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b})) \qquad (f_{32})$$

Ainsi, tout lien est directement ou indirectement valide et appartient par conséquent à l'univers de sa source ou l'univers de sa cible. Et la formule logique suivante impose qu'un lien soit valide pour exister :

$$\neg \text{Valid}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b}) \Rightarrow \neg \text{Link}(\boldsymbol{l}, \boldsymbol{a}, \boldsymbol{b}) \qquad (f_{33})$$

4.1.8 Univers et Univers sémantiques.

Les univers permettent d'organiser l'information contenue dans un réseau sNets. Nous avons déjà indiqué que toute entité appartient à un univers. Nous allons maintenant introduire la notion d'univers sémantique.

4.1.8.1 Les univers sémantiques.

Un univers sémantique est un univers dont le contenu définit un ensemble cohérent de types d'entités et de relations entre ces types d'entités. Il est donc composé de méta-entités (entités de type **EntityType** et de type **RelationType**) et va ainsi définir un méta-modèle.

Cette notion d'univers sémantique permet d'imposer une séparation entre les informations représentées dans les univers (les modèles) et leur sémantique représentée dans les univers sémantiques (leurs méta-modèles). L'externalisation de cette sémantique dans un univers permet une réutilisation simplifiée de celle-ci. De plus, les mécanismes d'extension d'univers permettent d'étendre un méta-modèle de la même façon que l'on étend un simple modèle.

Les univers sémantiques sont des représentés par des entités sNets de type **SemanticUniverse**. Ce type est un sous-type du type **Universe**. En effet, un univers sémantique n'est autre qu'un univers dont les entités sont toutes des méta-entités. Nous avons donc les deux prédicats suivants :

- Type(**SemanticUniverse**)
- Inherits(**SemanticUniverse**, **Universe**)

Prenons l'exemple de la Figure 55. Celle-ci définit une partie du méta-modèle de UML. Il serait donc intéressant de l'intégrer dans un univers sémantique qui représentera le méta-modèle d'UML. Cet univers sémantique pourra alors être utilisé dès qu'il sera utile de représenter des modèles UML.

La Figure 57 reprend donc le contenu de la Figure 55 et le place dans un univers sémantique appelé "Univers Sémantique UML".

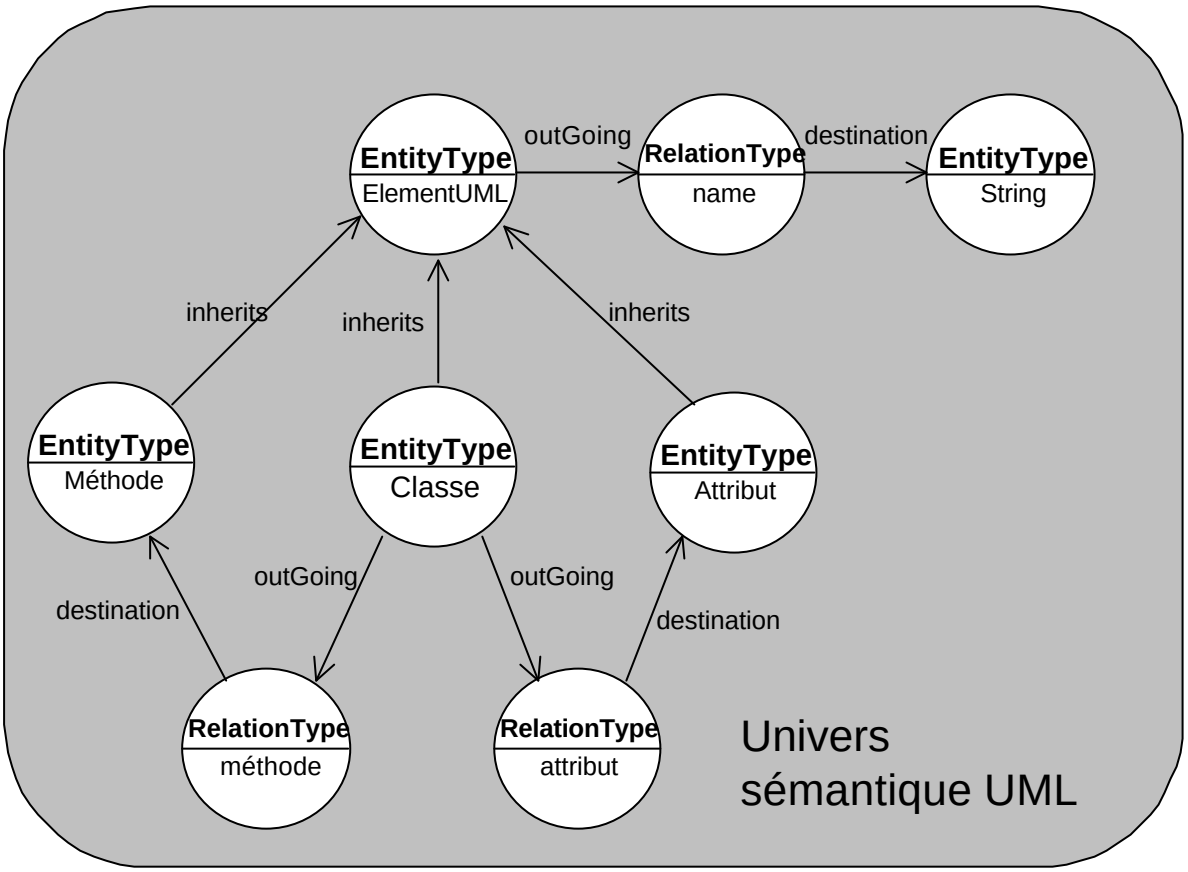


Figure 57 - "Univers sémantique représentant un sous-ensemble du méta-modèle de UML".

Cet univers sémantique peut ensuite être utilisé pour définir des modèles UML dans des univers sNets.

4.1.8.2 Relation entre univers et univers sémantique.

Tout univers dispose d'un univers sémantique. Cette relation entre un univers et son univers sémantique est introduite dans les sNets sous la forme d'un lien appelé *sem* partant d'un univers et aboutissant à l'univers sémantique de celui-ci. Cette contrainte est exprimée par la formule logique suivante :

$$\forall\, U\ \text{Node}(\boldsymbol{U})\ \text{Type}(\boldsymbol{U}, \textbf{Universe}) \Rightarrow \exists\, \boldsymbol{US}\ \text{Node}(\boldsymbol{US})\ \text{Type}(\boldsymbol{US}, \textbf{SemanticUniverse})\ \text{Link}(sem, \boldsymbol{U}, \boldsymbol{US}) \qquad (f_{34})$$

De la même façon, l'unicité de ce lien s'exprime formellement par la formule logique suivante :

$$\forall \boldsymbol{U}\ \text{Node}(\boldsymbol{U})\ \text{Type}(\boldsymbol{U}, \textbf{Universe})\ \text{Link}(\textit{sem}, \boldsymbol{U}, \boldsymbol{US_1})\ \text{Link}(\textit{sem}, \boldsymbol{U}, \boldsymbol{US_2}) \Rightarrow (\boldsymbol{US_1} = \boldsymbol{US_2}) \qquad (f_{35})$$

De plus, lorsqu'un univers ***U*** a pour univers sémantique un univers ***US***, tous les nœuds définis dans l'univers ***U*** doivent avoir leur type de défini dans l'univers ***US***. Cette règle s'exprime formellement par la formule logique suivante :

$$\forall \boldsymbol{x}\ \text{Node}(\boldsymbol{x})\ \text{DefinedIn}(\boldsymbol{x}, \boldsymbol{U})\ \text{Link}(\textit{sem}, \boldsymbol{U}, \boldsymbol{US}) \Rightarrow \exists \boldsymbol{X}\ \text{Link}(\textit{meta}, \boldsymbol{x}, \boldsymbol{X}) \wedge \text{DefinedIn}(\boldsymbol{X}, \boldsymbol{US}) \qquad (f_{36})$$

La Figure 58 contient un univers V représentant un modèle UML associé à son univers sémantique (une représentation sNets du méta-modèle d'UML).

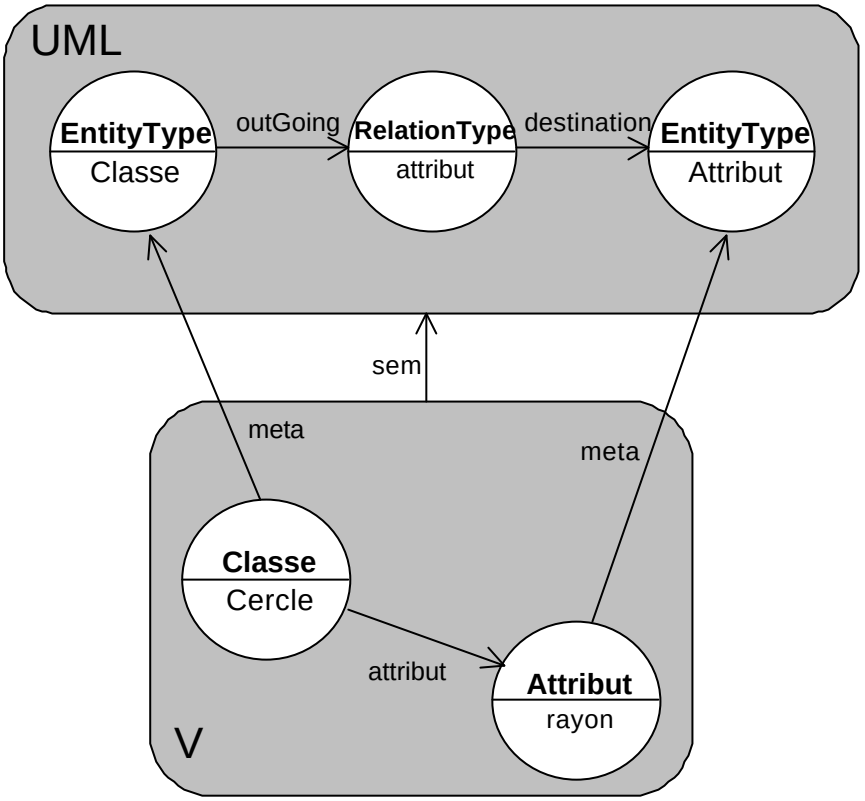


Figure 58 - Relations entre un univers et son univers sémantique.

L'univers V est lié à son univers sémantique par le biais du lien *sem*. De plus, toutes les entités représentées dans l'univers V vérifient bien la formule logique précédente et dispose bien d'un lien *meta* vers des entités définies dans l'univers sémantique de V (en l'occurrence l'univers UML). En effet, la figure 35 définie les prédicats suivants :

- DefinedIn(Cercle, V)
- DefinedIn(Rayon, V)
- Link(*sem*, V, UML)
- Link(*meta*, Cercle, Classe)
- Link(*meta*, Rayon, Attribut)
- DefinedIn(Classe, UML)
- DefinedIn(Attribut, UML)

De plus, lorsqu'un univers en étend un autre, il se doit d'avoir pour univers sémantique l'univers sémantique de l'univers étendu ou une extension de l'univers sémantique de l'univers étendu. Cette contrainte peut s'exprimer de façon formelle avec la formule logique suivante :

$$\forall \boldsymbol{U}\ \text{Node}(\boldsymbol{U})\ \text{Type}(\boldsymbol{U}, \textbf{Universe})\ \forall \boldsymbol{U_1}\ \text{Node}(\boldsymbol{U_1})\ \text{Type}(\boldsymbol{U_1}, \textbf{Universe})$$
$$\text{Link}(sem, \boldsymbol{U}, \boldsymbol{US})\ \text{Link}(sem, \boldsymbol{U_1}, \boldsymbol{US_1})\ \text{Extends}(\boldsymbol{U}, \boldsymbol{U_1}) \Rightarrow$$
$$(\boldsymbol{US} = \boldsymbol{US_1}) \vee \text{Extends}(\boldsymbol{US}, \boldsymbol{US_1})$$

(f_{37})

Dans la formule f_{36}, nous utilisons le prédicat DefinedIn qui tient compte de la relation d'héritage définie pour les univers. Par conséquent, cette formule est également vérifiée lorsque l'univers sémantique utilisé est une extension d'un autre univers sémantique et que l'univers du modèle contient des entités dont les types sont définis indifféremment dans l'un ou l'autre de ces univers sémantiques (cas présenté sur la Figure 59) :

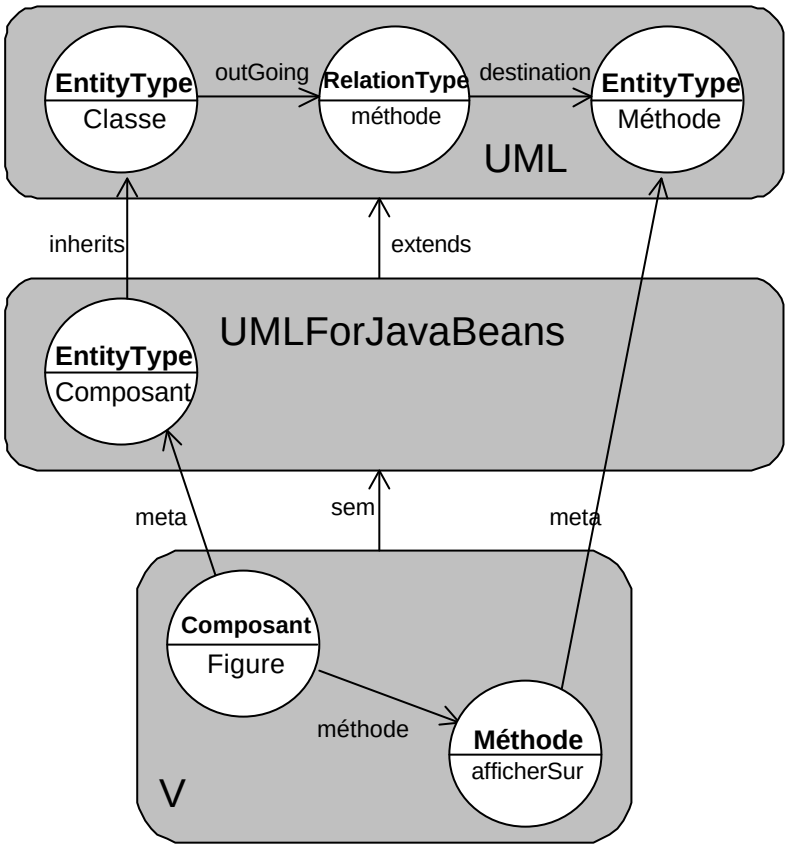


Figure 59 - Relations entre un univers et son univers sémantique (avec utilisation de l'extension des univers sémantiques).

Ainsi sur cette figure, le type **Composant** de l'entité Figure de l'univers V est défini dans l'univers sémantique de V (appelé UMLForJavaBeans) tandis que le type **Méthode** de l'entité afficherSur de ce même univers V est défini dans l'univers sémantique UML qui est un super-type de l'univers sémantique de UMLForJavaBeans (l'univers sémantique de V).

De la même façon, cette formule f_{36} doit être vérifiée lorsque l'univers décrivant le modèle est une extension d'un univers décrivant un modèle existant. Ce cas est représenté sur la Figure 60.

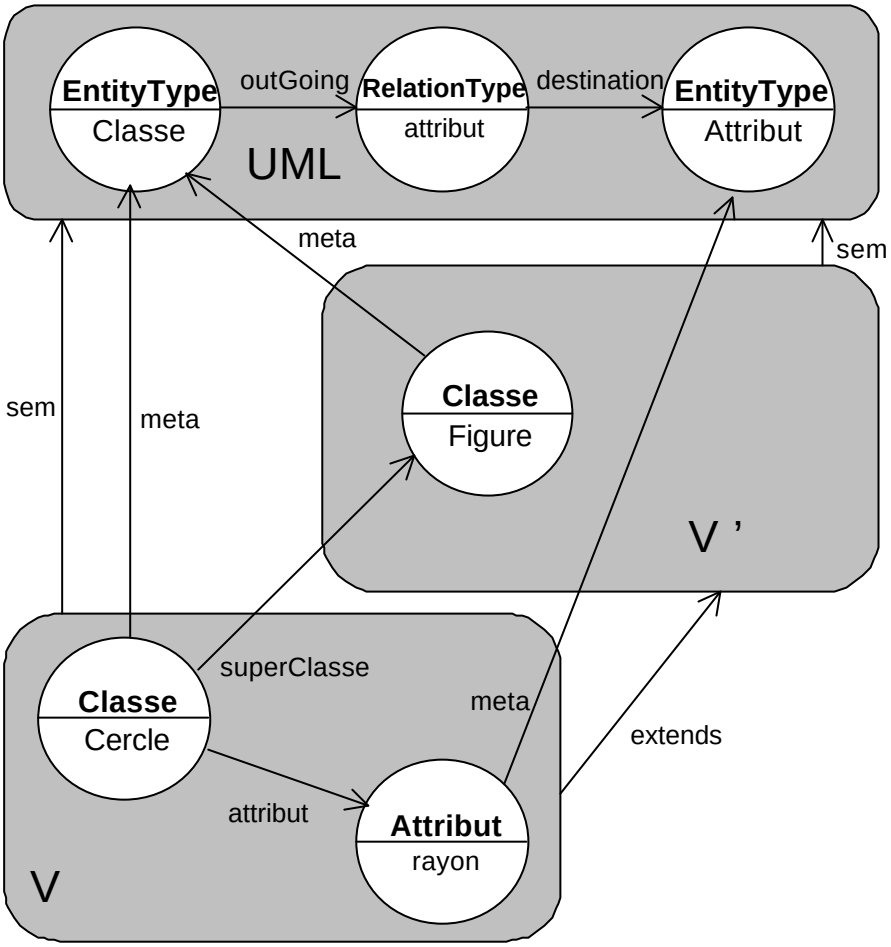


Figure 60- Relations entre un univers et son univers sémantique (avec utilisation de l'extension des univers représentant les modèles).

Sur cette Figure 60, comme l'univers V est une extension de l'univers V', il se doit de disposer pour univers sémantique du même univers sémantique que V' ou d'une extension de celui-ci. De sorte que toutes les entités définies dans V (sachant que V hérite de toutes les entités définies dans V') trouvent leur type dans l'univers sémantique de V.

Ces deux cas combinés nous conduisent à l'exemple représenté Figure 61. Ici, nous avons un modèle représenté par univers U ayant pour univers sémantique l'univers UML et un modèle représenté par l'univers V ayant pour sémantique l'univers UMLForJavaBeans. De plus, l'univers V est une extension de l'univers U.

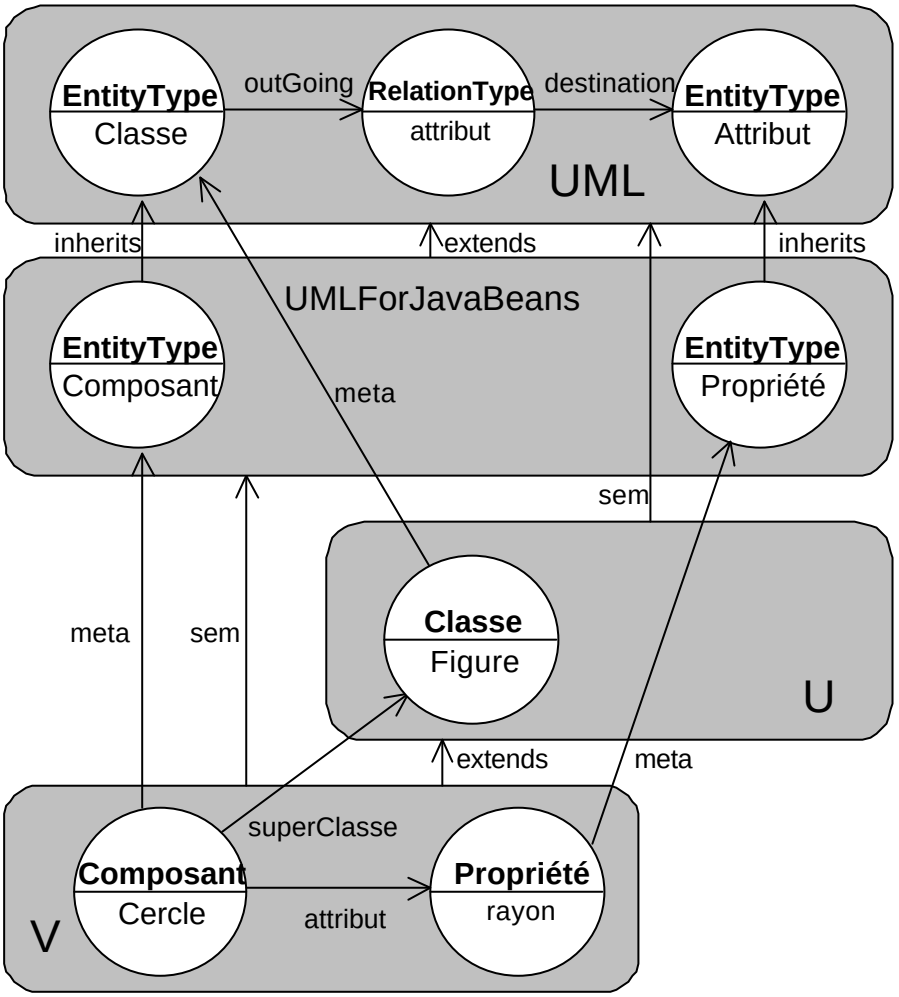


Figure 61 - Relations entre univers et univers sémantiques (avec utilisation de l'extension des univers représentant les modèles et leurs méta-modèles).

Pour que la formule f_{37} soit vérifiée sur cette figure, il est nécessaire que l'univers sémantique de V soit une extension de l'univers sémantique de U. Ainsi les entités Cercle, rayon et Figure (par héritage) de l'univers V trouvent bien tous leurs types respectifs **Composant**, **Propriété** et **Classe** (par héritage) dans l'univers UMLForJavaBeans.

Le formalisme des sNets ayant été présenté, nous allons maintenant passer à son utilisation. La première utilisation est son auto-définition. En effet, le cœur de ce formalisme est réflexif et par conséquent, nous allons présenter dans le chapitre suivant la façon dont ce cœur est défini.

4.2 La réflexivité dans le formalisme des sNets.

Nous avons vu dans le chapitre précédent que tout univers (ou modèle) doit disposer d'un univers sémantique (ou méta-modèle) décrivant tous les types d'entités et les types de relation qu'il sera susceptible de contenir. De plus, nous avons vu qu'un univers sémantique n'est ni plus ni moins qu'un univers dont le rôle est de définir une sémantique. Par conséquent, un univers sémantique doit également, au même titre que tout univers, disposer d'un univers sémantique et l'univers sémantique de nos univers sémantiques va constituer le méta-méta-modèle de notre formalisme.

4.2.1 Le noyau réflexif de notre méta-méta-modèle.

Ainsi, dans le chapitre précédent, nous avons vu que nous avions le schéma suivant :

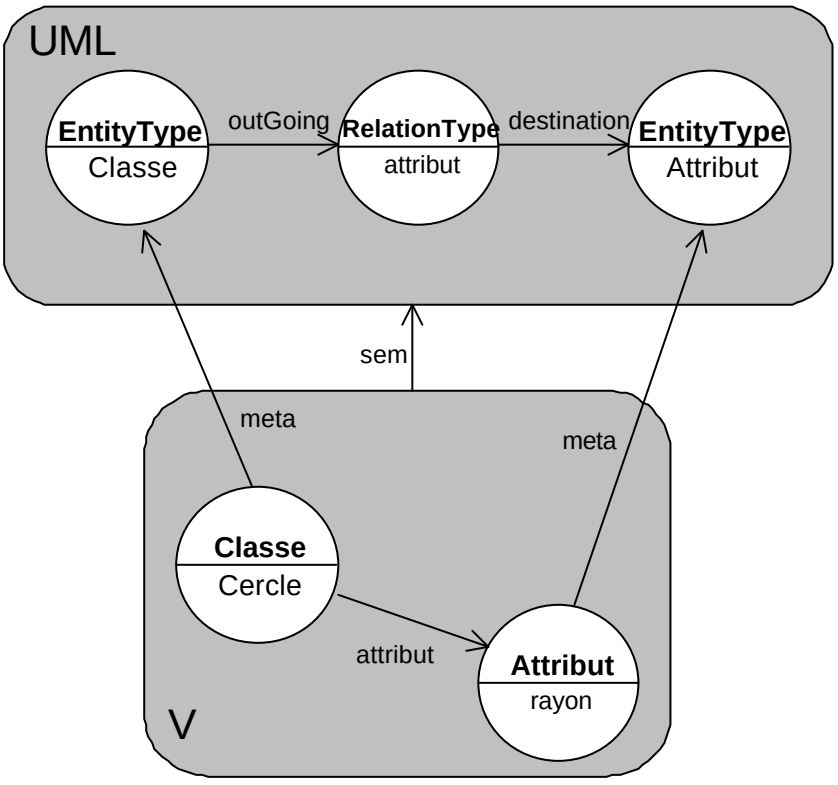


Figure 62 - Relations entre un univers et son univers sémantique.

Nous allons maintenant montrer qu'il doit en être de même l'univers sémantique UML présenté ici (Figure 62). En effet, UML étant un univers sémantique, c'est également un univers et il se doit de disposer de son propre univers sémantique. Cet univers sémantique qui représente la sémantique utilisée pour définir un univers sémantique va donc constitué le méta-modèle de nos méta-modèles. C'est en fait ce que l'on appelle méta-méta-modèle.

Ce méta-méta-modèle que l'on appelle Semantic dans notre formalisme doit donc définir les types et les relations nécessaires à la définition d'un univers sémantique. Ainsi, tous les types et toutes les relations présentées dans le chapitre précédent comme étant "prédéfinis" trouvent leur définition dans ce méta-méta-modèle. Un sous-ensemble de cet univers est présenté Figure 63.

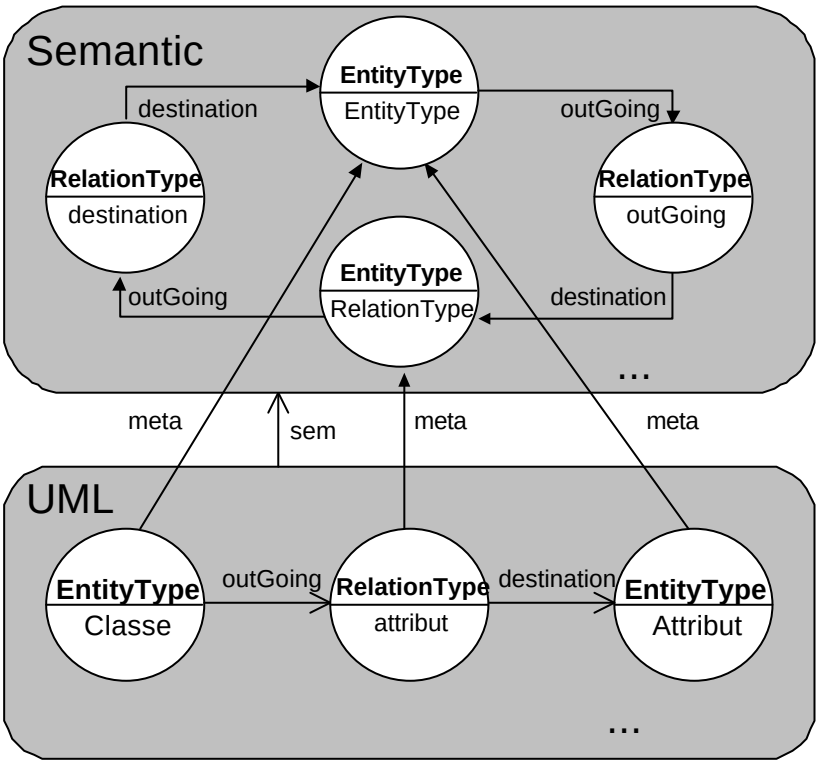


Figure 63 - Une partie de notre méta-méta-modèle représenté par l'univers sémantique Semantic.

Nous avons vu dans le chapitre précédent que pour définir un type d'entité dans le formalisme des sNets on définissait une entité de type **EntityType** (c.f. chapitre 4.1.4) et que pour définir un type de relation entre deux types d'entités on définissait une entité de type **RelationType** (c.f.

chapitre 4.1.5). Le type de l'entité source est alors liée au type de la relation par un lien *outGoing* et le type de la relation est lié au type de l'entité destination par un lien *destination*.

Ces quatres types (**EntityType**, **RelationType**, *outGoing* et *destination*) sont donc définis dans cet univers sémantique Semantic constituant notre méta-méta-modèle de sorte qu'il soit possible de créer des entités et des liens de ces types dans nos univers sémantiques.

Nous avons donc un univers sémantique décrivant la sémantique de tous les univers sémantiques. Mais cet univers doit également avoir son propre univers sémantique. Il se trouve que la sémantique qu'il utilise est celle qu'il définit. Par conséquent, la définition de l'univers Semantic est réflexive et les règles et contraintes établies au chapitre précédent vont être valable également pour lui.

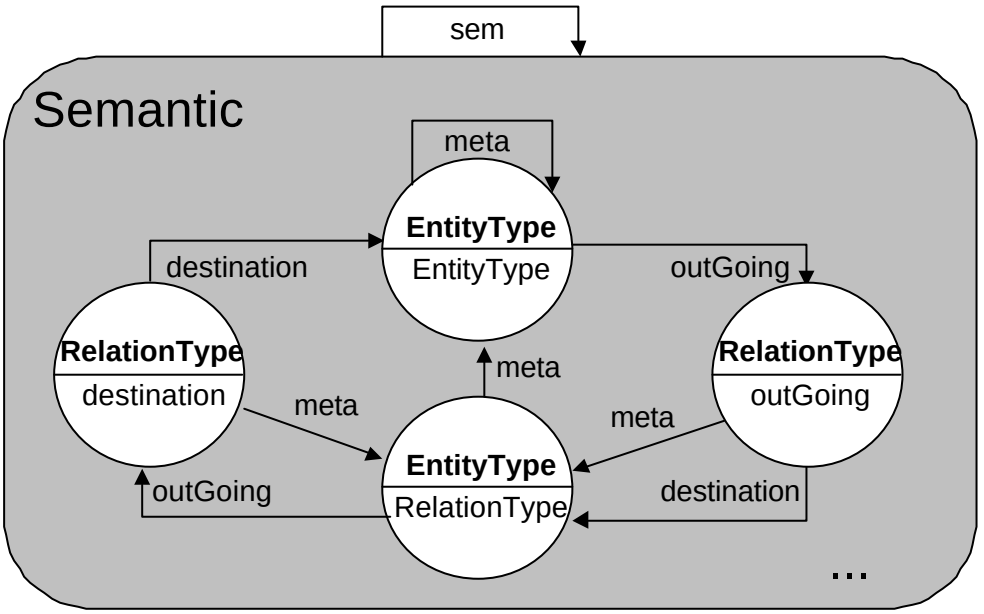


Figure 64 - Réflexivité du méta-méta-modèle des sNets.

La Figure 64 présente donc notre univers sémantique Semantic disposant d'un lien *sem* vers lui-même indiquant que les types de toutes les entités présentes dans Semantic sont également définis dans Semantic.

Cette figure nous permet de définir les prédicats suivants concernant les entités de Semantic :

- DefinedIn(EntityType, Semantic)
- DefinedIn(RelationType, Semantic)
- DefinedIn(destination, Semantic)
- DefinedIn(outGoing, Semantic)

Et les types de ces entités sont définis par les prédicats suivants :

- Type(EntityType, **EntityType**)
- Type(RelationType, **EntityType**)
- Type(destination, **RelationType**)
- Type(outGoing, **RelationType**)

Et les relations qu'ils définissent ici sont les suivantes :

- LinkType(*outGoing*, **EntityType**, **RelationType**)
- LinkType(*destination*, **RelationType**, **EntityType**)

Par conséquent, nous trouvons effectivement la définition de tous les types utilisés pour définir Semantic (en l'occurrence **EntityType** et **RelationType**) dans l'univers Semantic.

4.2.2 Nommage des entités.

Toute entité doit disposer d'un nom matérialisé par une entité de type **String** liée à l'entité via un lien *name*. Ce nommage est défini de la façon suivante dans notre méta-méta-modèle :

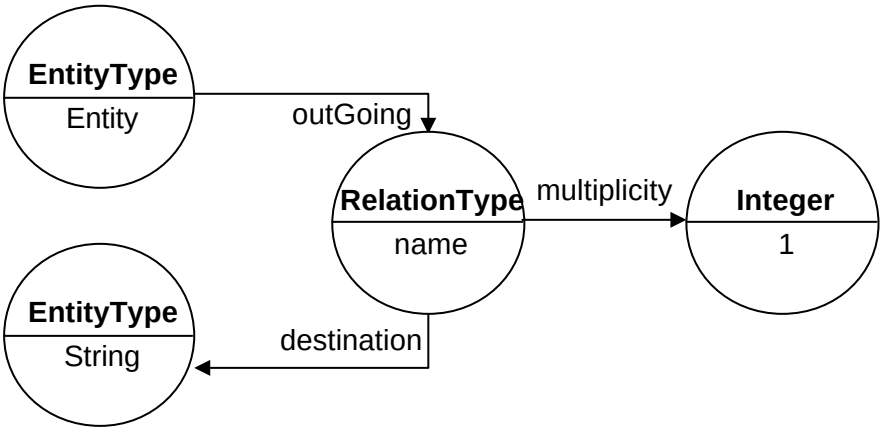


Figure 65 - Définition de la relation *name* entre une entité et son nom.

Les types des entités pésentes sur cette figure sont définis par les prédicats suivants :

- Type(Entity, **EntityType**)
- Type(name, **RelationType**)
- Type(String, **EntityType**)

Et la relation qu'ils définissent ici est la suivante :

- LinkType(*name*, **Entity**, **String**)

Tous les types d'entité (entités de type **EntityType**) devront alors **nécessairement** hériter de l'entité Entity de manière à ce que toutes les entités puissent être effectivement manipulées comme des entités de type **Entity** et dispose ainsi d'un lien *name*. Cette définition devrait donc être présente dans le méta-méta-modèle, mais également dans tous les méta-modèles.

4.2.3 Un univers de base commun à tous les univers sémantiques.

Nous avons donc défini un univers CommonSem contenant tous les mécanismes de base nécessaires à la définition d'un méta-modèle (sachant que notre méta-méta-modèle est également un méta-modèle). Tout méta-modèle doit alors nécessairement étendre (via la relation *extends*) cet univers (y compris Semantic). La définition de la relation *name* entre une entité et son nom est donc placée dans cet univers CommonSem de la façon suivante :

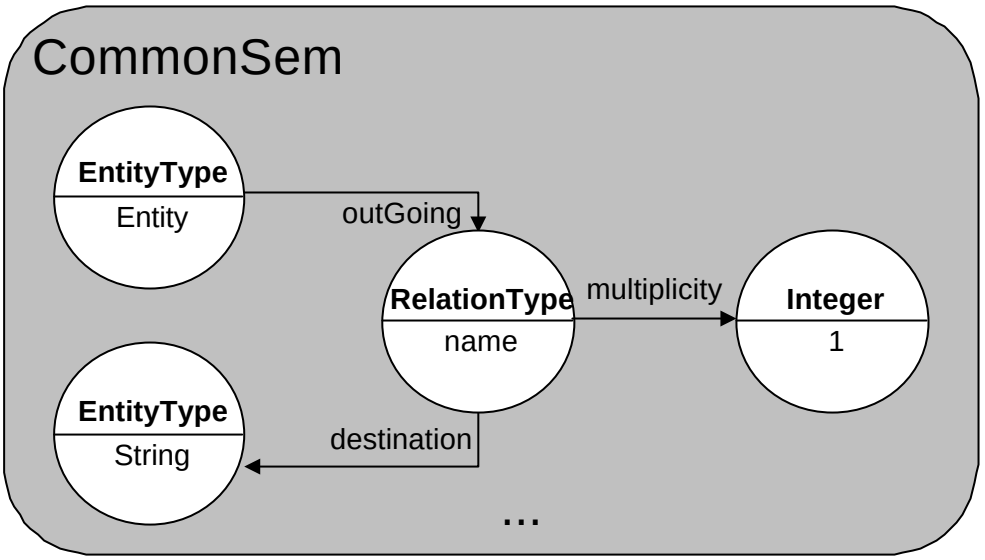


Figure 66 - Définition de la relation de nommage dans l'univers CommonSem.

De plus, comme l'univers CommonSem est un univers sémantique (il définit des entités de type **EntityType** et des entités de type **RelationType**), il doit avoir pour univers sémantique l'univers Semantic qui définit ces types.

Nous avons donc les relations suivantes entre l'univers Semantic désignant notre méta-méta-modèle et l'univers CommonSem contenant les pré-requis de tous méta-modèles :

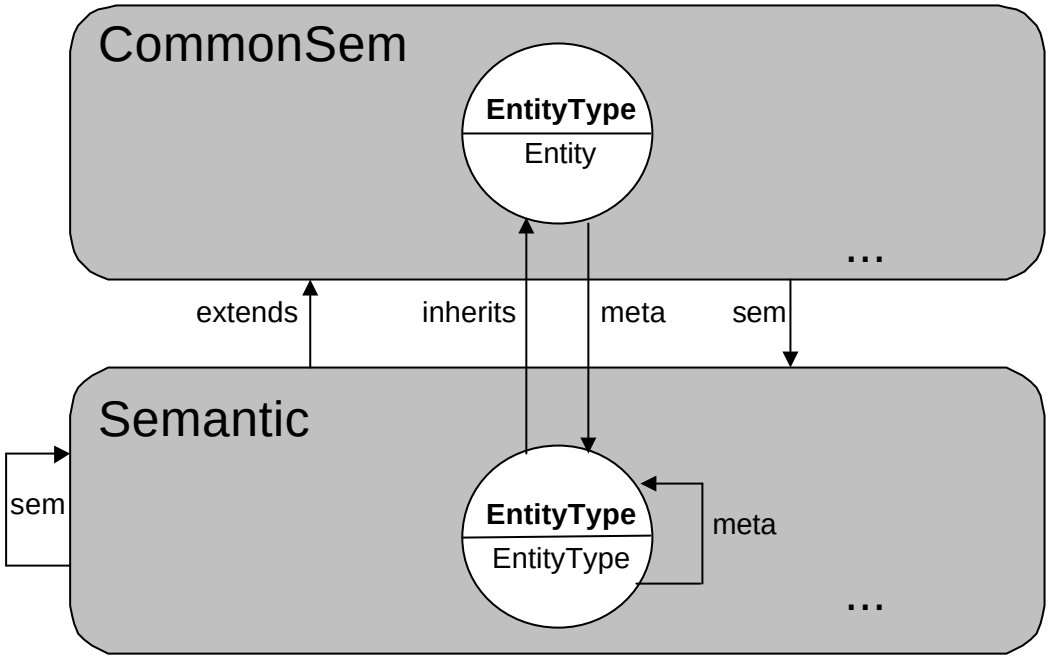


Figure 67 - Relations entre l'univers Semantic et l'univers CommonSem.

Par la suite, tout ce qui est utilisé essentiellement pour définir un méta-modèle est placé dans notre méta-méta-modèle réflexif Semantic tandis que tout ce qui est nécessaire à la définition d'un modèle est placé dans l'univers CommonSem. La Figure 67 montre ainsi que la notion d'entité définie par le type **Entity** doit être présente dans tous les méta-modèles de manière à pouvoir être utilisée dans tous les modèles tandis que la notion de méta-entité définie par le type **EntityType** doit être présente seulement dans notre méta-méta-modèle car seuls les méta-modèles vont définir des méta-entités.

4.2.4 Typage des entités.

Toute entité doit disposer d'un type matérialisé par une entité de type **EntityType** liée à l'entité via un lien *meta*. Par conséquent, le typage des entités doit être défini de la façon suivante dans notre méta-méta-modèle (l'univers Semantic) :

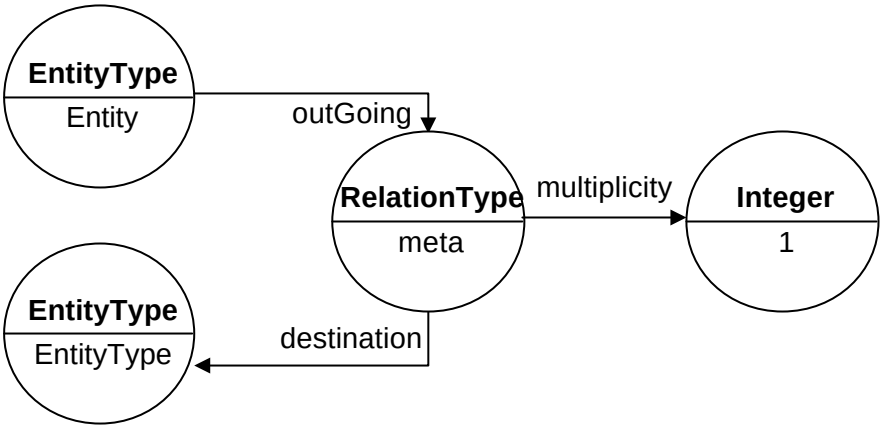


Figure 68 - Définition de la relation *meta* entre une entité et sa méta-entité.

Nous avons vu dans le chapitre précédent que Entity était défini dans l'univers CommonSem tandis que EntityType était défini dans l'univers Semantic. La relation *meta* est quant à elle placée dans l'univers CommonSem et nous avons donc le schéma suivant concernant la définition de cette relation :

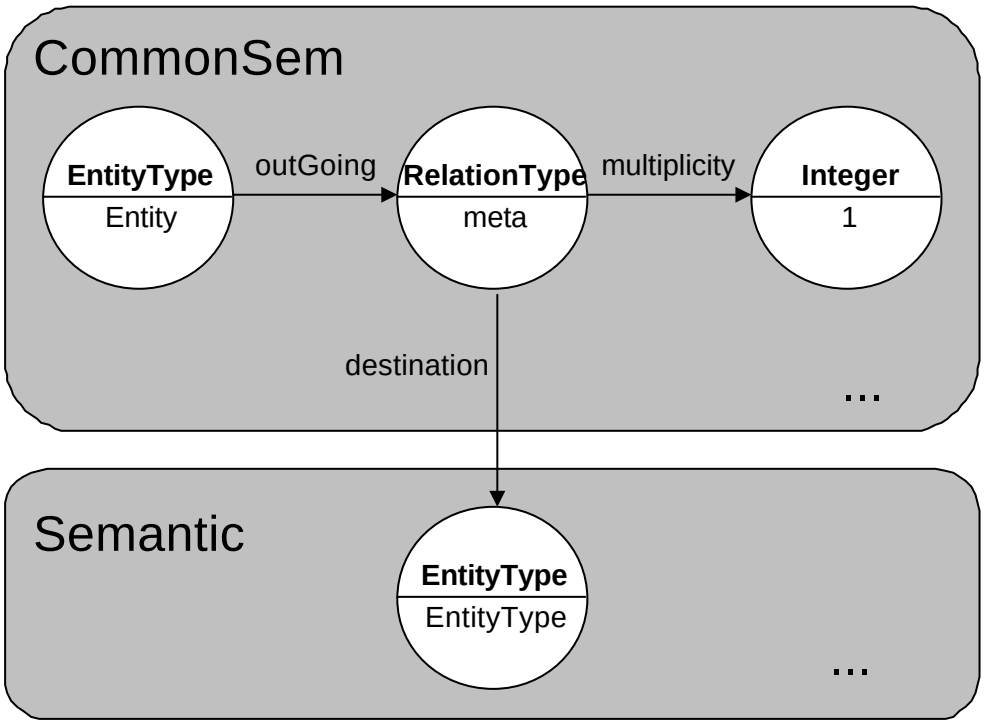


Figure 69 - Définition de la relation *meta* dans l'univers CommonSem.

Cette figure nous permet de définir les prédicats suivants concernant le typage des entités :

- DefinedIn(Entity, CommonSem)
- DefinedIn(meta, CommonSem)
- DefinedIn(EntityType, Semantic)

Les types de ces entités sont définis par les prédicats suivants :

- Type(Entity, **EntityType**)
- Type(meta, **RelationType**)
- Type(EntityType, **EntityType**)

Et la relation qu'ils définissent ici est la suivante :

- LinkType(*meta*, **Entity**, **EntityType**)

Toute les types d'entité (entités de type **EntityType**) devront alors hériter de l'entité Entity de manière à ce que toutes les entités puissent être effectivement manipulées comme des entités de type **Entity** et dispose ainsi d'un lien *meta*.

Dans l'univers CommonSem sont également définis tous les types "simples" (**String**, **Integer**, etc…) car des entités, "instances" de ces types, peuvent être définies dans tous les modèles et ces méta-entités peuvent alors être référencées dans tous les méta-modèles. Nous avons donc :

- DefinedIn(String, CommonSem)
- DefinedIn(Integer, CommonSem)

etc…

Et toutes ces entités sont de type **EntityType** :

- Type(String, **EntityType**)
- Type(Integer, **EntityType**)

etc…

4.2.5 Définition de la notion d'univers.

Les univers sont représentés par des entités de type **Universe**, il faut donc que ce type soit défini dans notre méta-méta-modèle. Ce type n'est pas spécifique à la définition d'un méta-modèle et sera par conséquent placé dans l'univers CommonSem. Il faut également que toute entité puisse disposer d'un lien *partOf* vers une entité de ce type.

Nous avons donc défini les éléments suivant dans l'univers CommonSem :

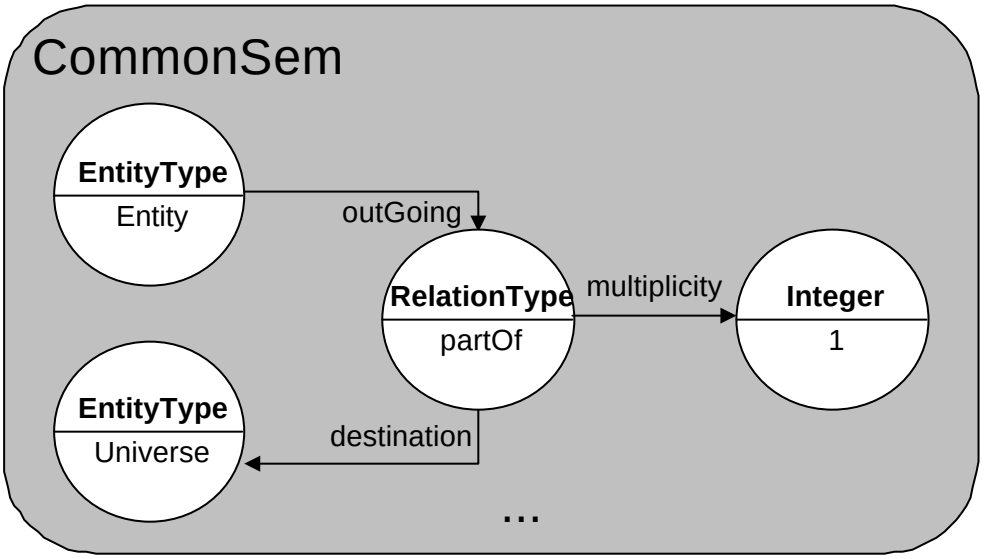


Figure 70 - Définition du type Universe et de la relation *partOf* dans l'univers CommonSem.

De plus, pour faciliter la navigation dans un modèle sNets, nous avons également définit une relation contains qui se trouve être la relation inverse de la relation *partOf*.

Ces deux relations forment donc une relation bidirectionnelle définie de la façon suivante dans CommonSem :

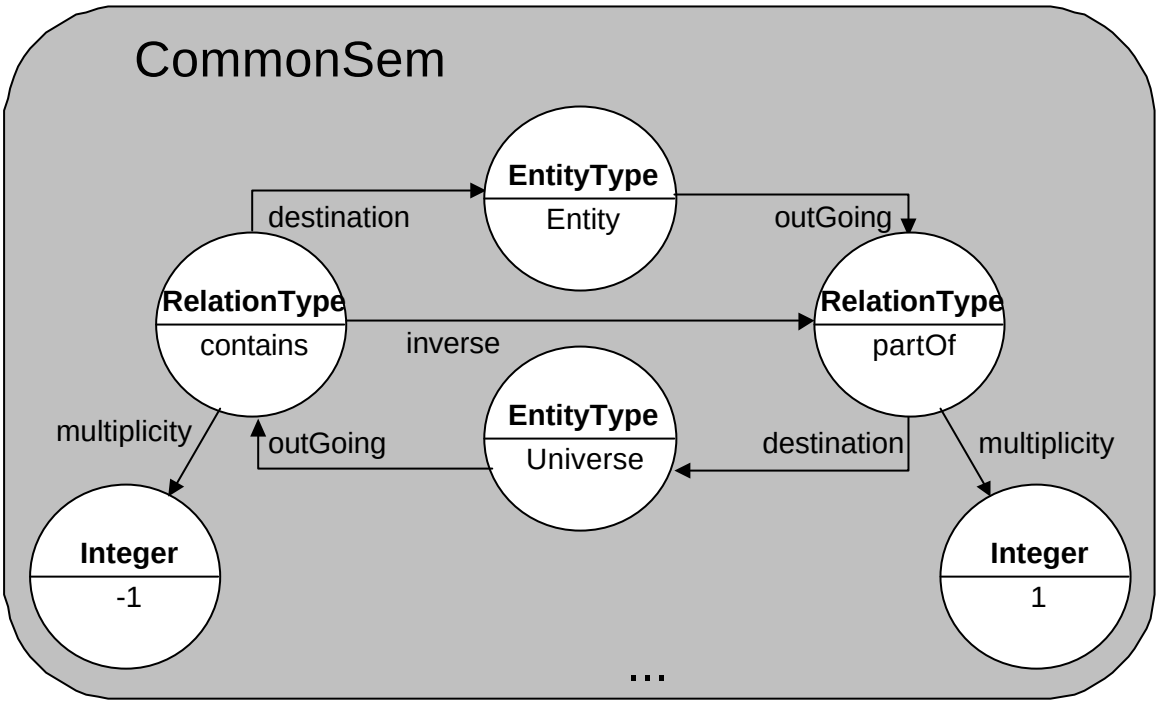


Figure 71 - Définition de la relation bidirectionnelle entre entités et univers dans l'univers CommonSem.

Cette figure nous permet de définir les prédicats suivants concernant l'appartenance des entités aux univers :

- DefinedIn(Entity, CommonSem)
- DefinedIn(partOf, CommonSem)
- DefinedIn(contains, CommonSem)
- DefinedIn(Universe, CommonSem)

Les types de ces entités sont définis par les prédicats suivants :

- Type(Entity, **EntityType**)
- Type(partOf, **RelationType**)
- Type(contains, **RelationType**)

- Type(Universe, **EntityType**)

Et les relations qu'ils définissent ici sont les suivantes :

- LinkType(*partOf*, **Entity**, **Universe**)
- LinkType(*contains*, **Universe**, **Entity**)

De plus, le liens inverse suivant est défini :

- Link(*inverse*, partOf, contains)

4.2.6 Définition d'un type de lien.

Les méta-entités permettant la définition d'un type d'entité (**EntityType**) et d'un type de lien (**RelationType**) ont déjà été présentée Figure 64 car elles se trouvent au cœur de notre méta-méta-modèle.

Pour faciliter la navigation, nous avons également définit des liens inverse pour les liens *outGoing* et *destination*. Ces liens sont appelés respectivement *inComing* et *source*. De plus, les cardinalités n'étaient pas représentées Figure 64.

Tous ces éléments sont donc repris sur la figure suivante :

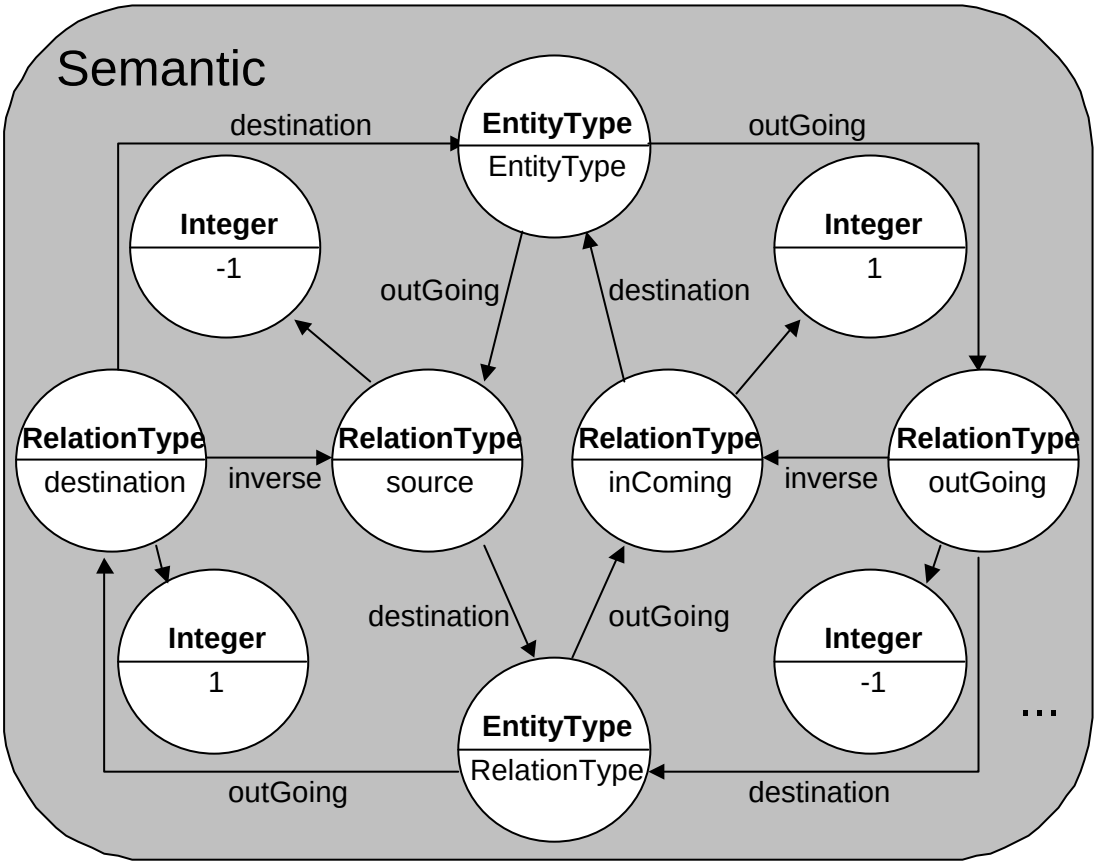


Figure 72 - Définition des types EntityType et RelationType ainsi que des méta-relations auxquelles ils participent.

Cette figure nous permet de définir les prédicats suivants concernant ces entités :

- DefinedIn(EntityType, Semantic)
- DefinedIn(RelationType, Semantic)
- DefinedIn(destination, Semantic)
- DefinedIn(outGoing, Semantic)
- DefinedIn(source, Semantic)
- DefinedIn(inComing, Semantic)

Et les types de ces entités sont définis par les prédicats suivants :

- Type(EntityType, **EntityType**)
- Type(RelationType, **EntityType**)
- Type(destination, **RelationType**)
- Type(outGoing, **RelationType**)
- Type(source, **RelationType**)
- Type(inComing, **RelationType**)

Et les relations qu'ils définissent ici sont les suivantes :

- LinkType(*outGoing*, **EntityType**, **RelationType**)
- LinkType(*destination*, **RelationType**, **EntityType**)
- LinkType(*source*, **EntityType**, **RelationType**)
- LinkType(*inComing*, **RelationType**, **EntityType**)

De plus, les liens inverse suivant sont définis :

- Link(*inverse*, outGoing, inComing)
- Link(*inverse*, destination, source)

La notion de relation inverse doit également être définie dans notre méta-méta-modèle. Cette relation se matérialise par un lien nommé inverse partant d'une entité de type **RelationType** et aboutissant à une entité de type **RelationType**. Cette relation est également bi-directionnelle de sorte que lorsque l'on définit indique qu'une relation a est l'inverse d'une relation b, cette relation b est nécessairement l'inverse de la relation a.

Ces différents éléments sont donc définis sur la figure suivante :

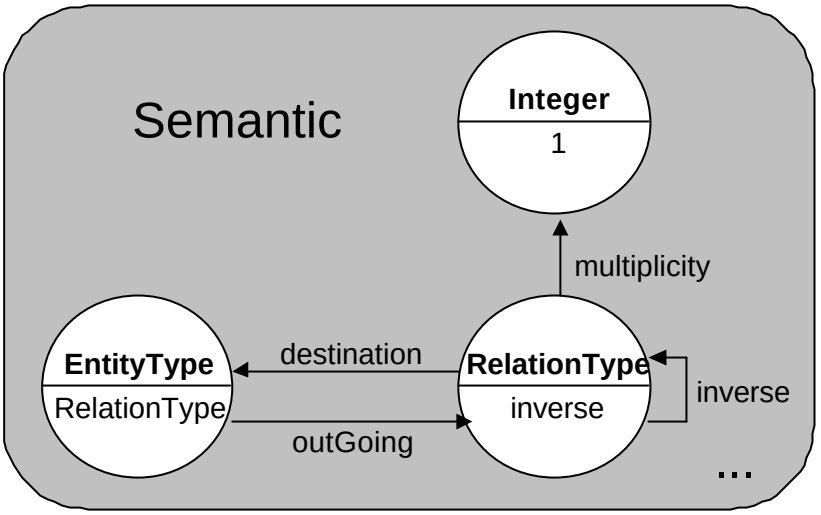


Figure 73 - Définition de la relation bidirectionnelle inverse permettant de définir des relations bidirectionnelles.

Cette figure nous permet de définir les prédicats suivants concernant la définition de relations bi-directionnelles :

- DefinedIn(inverse, Semantic)
- Type(inverse, **RelationType**)
- LinkType(*inverse*, **RelationType**, **RelationType**)
- Link(*inverse*, inverse, inverse)

4.2.7 Définition des mécanismes d'extension.

Nous décrivons ici comment ces mécanismes sont définis dans notre formalisme.

4.2.7.1 Définition du mécanisme d'extension des univers.

La relation d'extension entre univers se matérialise par un lien *extends* entre un univers et les univers qu'il étend. Dans un souci de navigation, nous avons rendu cette relation bi-directionnelle en y ajoutant un lien inverse appelé *extensions*.

Ces mécanismes sont définis de la façon suivante dans l'univers CommonSem :

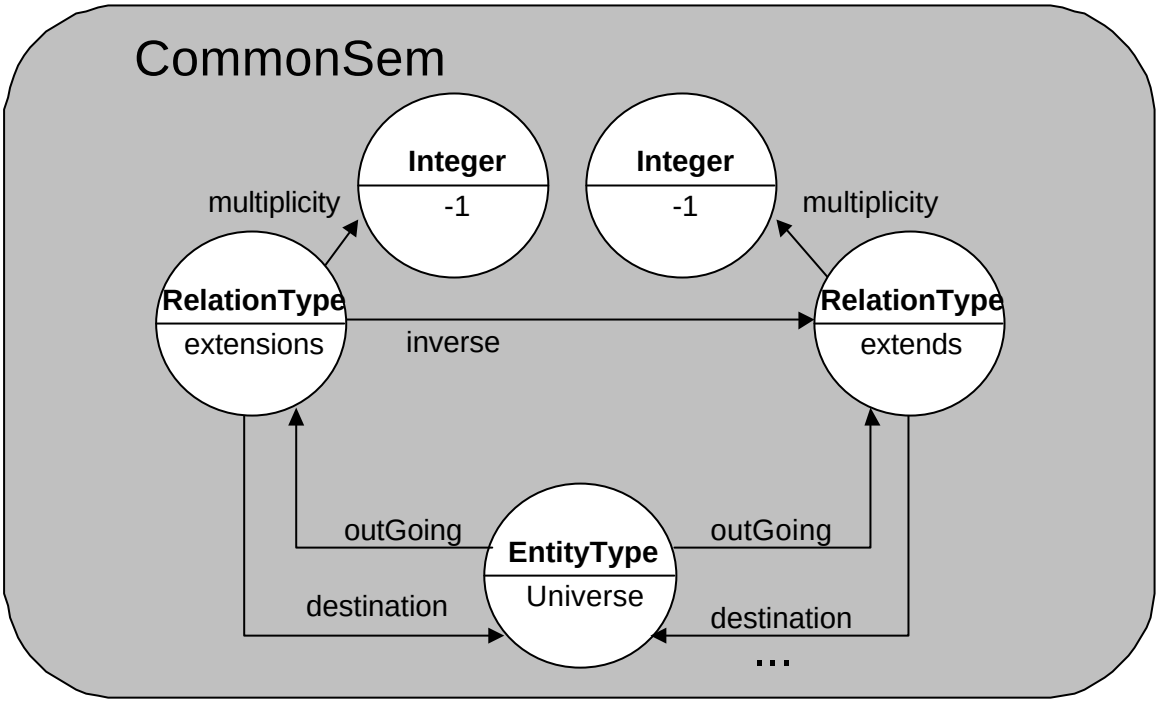


Figure 74 - Définition des relations d'extension entre univers dans CommonSem.

Cette figure nous permet de définir les prédicats suivants concernant ces entités :

- DefinedIn(Universe, CommonSem)
- DefinedIn(extensions, CommonSem)
- DefinedIn(extends, CommonSem)

Les types de ces entités sont définis par les prédicats suivants :

- Type(Universe, **EntityType**)
- Type(extensions, **RelationType**)
- Type(extends, **RelationType**)

Et les relations qu'ils définissent ici sont les suivantes :

- LinkType(*extensions*, **Universe**, **Universe**)
- LinkType(*extends*, **Universe**, **Universe**)

De plus, le lien inverse suivant est défini :

- Link(*inverse*, extensions, extends)

4.2.7.2 Définition du mécanisme d'héritage des types.

La relation d'héritage entre méta-entités se matérialise par un lien *inherits* entre une entité de type **EntityType** et son super-type représenté également par une entité de type **EntityType**. Dans un souci de navigation, nous avons rendu cette relation bi-directionnelle en y ajoutant un lien inverse appelé *subtypes*.

Ces mécanismes sont définis de la façon suivante dans l'univers Semantic :

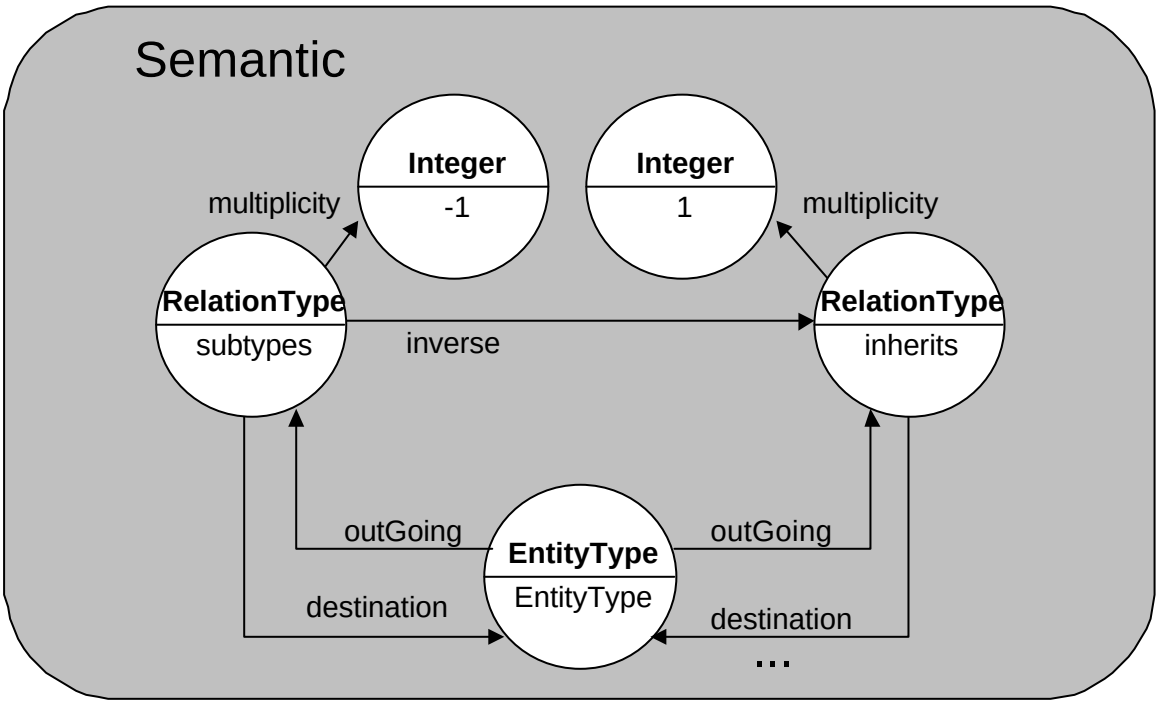


Figure 75 - Définition de la relation d'héritage entre entités de type EntityType dans Semantic.

Cette figure nous permet de définir les prédicats suivants concernant ces entités :

- DefinedIn(EntityType, Semantic)
- DefinedIn(inherits, Semantic)
- DefinedIn(subtypes, Semantic)

Les types de ces entités sont définis par les prédicats suivants :

- Type(EntityType, **EntityType**)
- Type(inherits, **RelationType**)
- Type(subtypes, **RelationType**)

Et les relations qu'ils définissent ici sont les suivantes :

- LinkType(*inherits*, **EntityType**, **EntityType**)
- LinkType(*subtypes*, **EntityType**, **EntityType**)

De plus, le lien inverse suivant est défini :

- Link(*inverse*, inherits, subtypes)

4.2.8 Définition des relations entre univers et univers sémantique.

Tout univers doit disposer d'un univers sémantique. Cette relation se matérialise par un lien *sem* entre un univers et son univers sémantique. Dans un souci de navigation, nous avons rendu cette relation bi-directionnelle en y ajoutant un lien inverse appelé *models*. De plus, un univers sémantique est un sous-type d'univers.

Ces mécanismes sont définis de la façon suivante dans l'univers CommonSem :

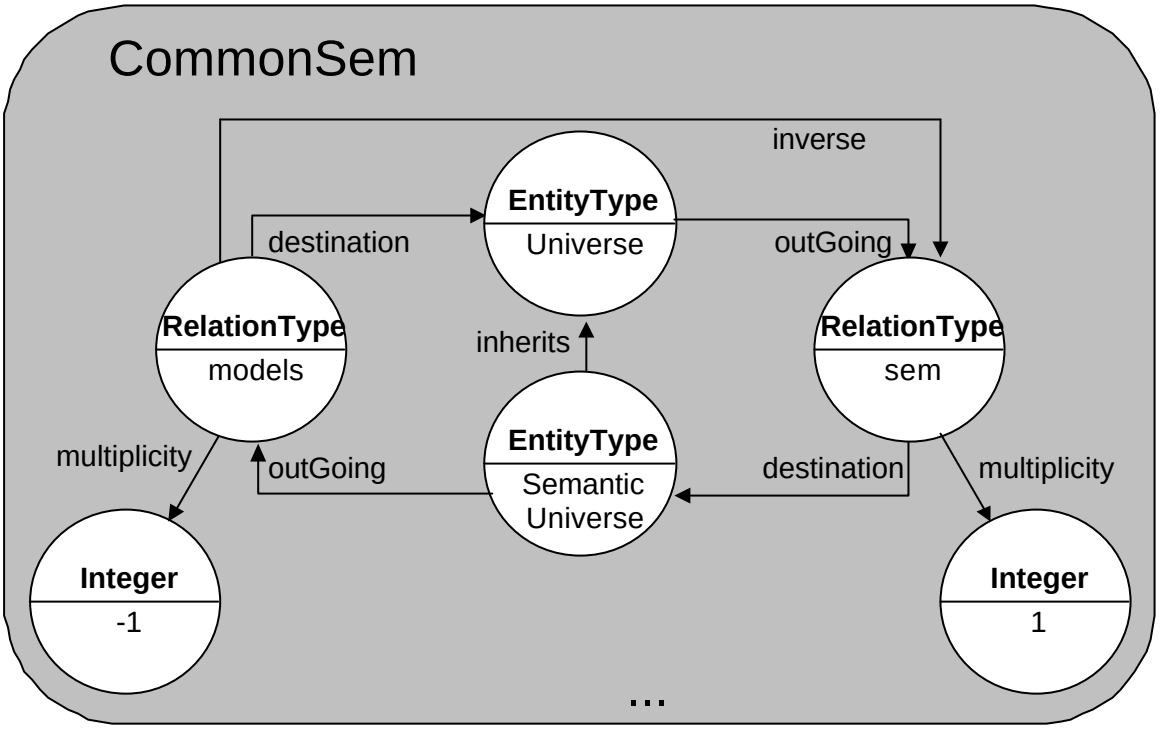


Figure 76 - Définition de la notion d'univers sémantique et de leur relations avec les univers.

Cette figure nous permet de définir les prédicats suivants concernant les univers :

- DefinedIn(Universe, CommonSem)
- DefinedIn(models, CommonSem)
- DefinedIn(sem, CommonSem)
- DefinedIn(SemanticUniverse, CommonSem)
- Inherits(SemanticUniverse, Universe)

Les types de ces entités sont définis par les prédicats suivants :

- Type(Universe, **EntityType**)
- Type(models, **RelationType**)
- Type(sem, **RelationType**)
- Type(SemanticUniverse, **EntityType**)

Et les relations qu'ils définissent ici sont les suivantes :

- LinkType(*sem*, **Universe**, **SemanticUniverse**)
- LinkType(*models*, **SemanticUniverse**, **Universe**)

De plus, le liens inverse suivant est défini :

- Link(*inverse*, sem, models)

Tous les éléments qui composent le noyau réflexif de notre formalisme ont maintenant été présentés. Nous allons donc étudier, dans les chapitres suivants, l'exploitation d'un tel formalisme et les apports d'un tel noyau réflexif.

4.3 Une implémentation du formalisme des sNets.

Une première implémentation du formalisme des sNets avait été tout d'abord réalisée en Smalltalk Cette implémentation est au cœur de l'outil Semantor de la société Softmaint. Semantor est un outil de rétrodocumentation de programmes COBOL. Le langage COBOL y est représenté par un méta-modèle au format des sNets enrichi par un ensemble de types supplémentaires permettant entre autres choses la représentation de flots de données et de flots de contrôle de ces programmes. Les programmes COBOL sont alors représentés par des modèles suivant la sémantique décrite dans ce méta-modèle et des outils spécifiques ont été réalisés de manière à manipuler ce type d'information.

Une seconde implémentation de ce formalisme a été réalisée en Java. Cette implémentation est au cœur d'un outil de génération de code. Cet outil dispose du méta-modèle de UML représenté en utilisant le formalisme UML de sorte que les modèles UML peuvent alors être manipulés dans cet outil comme de simples modèles sNets.

De manière a dissocier l'implémentation du formalisme de son utilisation, nous avons défini un ensemble d'interfaces. Une implémentation du formalisme des sNets se doit alors d'implémenter chacune de ces interfaces.

4.3.1 Implémentation d'une entité sNets.

Toute entité sNets se doit d'implémenter l'interface suivante :

```
public interface SemNode {

    public String getName();
    public void setName(String name);
    public String getGlobalName();
    public SemNodeNode getMeta();
    public SemUniverseNode getPartOf();
    public Iterator getLinkNames();
    public void setLinkToNode(String linkName, SemNode node);
    public SemNode getNodeForLink(String linkName);
    public Iterator getNodesForLink(String linkName);
```

```
        public boolean hasLinkToNode(String linkName, SemNode node);
        public void removeAnyLink(String linkName);
        public void removeLinkToNode(String linkName, SemNode node);
}
```

Les méthodes getName et setName permettent d'accéder et de modifier le nom de l'entité.

La méthode getGlobalName renvoie le nom global de l'entité. Ce nom global est constitué du nom de l'entité préfixé du nom global de son univers (les deux étant séparés par un caractère ".").

La méthode getMeta renvoie la méta-entité représentant le type de cette entité. L'entité renvoyée doit alors être de type **EntityType** et doit implémenter l'interface SemNodeNode.

La méthode getPartOf renvoie l'entité représentant l'univers de cette entité. L'entité renvoyée doit alors être de type **Universe** et doit implémenter l'interface SemUniverseNode.

La méthode getLinkNames renvoie les noms de tous les liens qui partent de cette entité.

La méthode setLinkToNode permet de créer un lien linkName entre l'entité courante et l'entité représentée par le paramètre node**. Attention, le lien n'est effectivement créé que s'il est défini entre les types de ces deux entités de sorte que lorsque l'on construit un modèle sNets via ces interfaces, le modèle construit respecte toujours les règles définies dans le chapitre 4.1**.

Les méthodes getNodeForLink et getNodesForLink renvoient le ou les entités rattachée à l'entité via un lien linkName.

La méthode hasLinkToNode renvoie vrai si l'entité dispose d'un lien linkName vers l'entité représentée par le paramètre node.

La méthode removeAnyLink supprime tous les liens nommés linkName et quittant l'entité.

La méthode removeLinkToNode supprime le lien nommés linkName reliant l'entité courante à celle représentée par le paramètre node.

Cette interface permet de manipuler indifféremment des entités sNets, des méta-entités sNets, des univers sNets et de nombreux outils génériques vont pouvoir profiter de celle-ci. Nous avons ainsi développé un grapheur générique, un browser générique et un outil de transformation de modèles générique.

Afin de manipuler simplement un certain nombre d'autres entités sNets de base (en l'occurrence, les méta-entités, les méta-relations et les univers) nous avons défini trois autres interfaces.

4.3.2 Implémentation d'une méta-entité sNets.

Toute méta-entité sNets (entité sNets de type **EntityType**) se doit d'implémenter l'interface suivante :

```
public interface SemNodeNode extends SemNode {

        public Iterator getOutgoingLinks();
        public Iterator getAllOutgoingLinks();
        public Iterator getOutgoingLinksNamed(String linkName);
        public Iterator getAllOutgoingLinksNamed(String linkName);
        public Iterator getOutgoingLinksToNodesOfType(sNets.SemNodeNode node);
        public Iterator getAllOutgoingLinksToNodesOfType(sNets.SemNodeNode node);
        public SemLinkNode getOutgoingLinkToNodesOfType(String linkName,
        SemNodeNode node);
        public Iterator getAllPossibleTypesForLink(String name);
        public Iterator getPossibleTypesForLink(String name);
        public Iterator getSourceLinks();
        public Iterator getAllSourceLinks();
        public Iterator getSubtypes();
        public Iterator getSupertypes();
        public boolean inherits(String nodeName);
        public boolean inherits(SemNodeNode node);
        public SemNode newNodeIn(SemUniverseNode universe);
}
```

Cette interface hérite de l'interface SemNode car toute méta-entité sNets est également une entité sNets. Les méthodes qu'elle définit facilitent la manipulation d'une méta-entité sNets mais toutes les fonctionnalités définies dans cette interfaces pourraient être obtenues en se contentant de manipuler une méta-entité sNets via l'interface SemNode. En effet, si l'on prend l'exemple de la méthode getSuperTypes() qui renvoie les super-types de cette entité, ils pourraient être obtenus de la même façon en invoquant la méthode getNodesForLink("inherits") car une méta-entité sNets est liée à ses super-entités via un lien présenté précédemment et appelé *inherits* (c.f. chapitre 4.1.6.2*)*.

La méthode getOutgoingLinks renvoie les méta-relations dont la source est la méta-entité courante.

La méthode getAllOutgoingLinks renvoie les méta-relations dont la source est la méta-entité courante en tenant compte de l'héritage. Par conséquent elle renvoie également les méta-relations dont la source est un super-type de l'entité courante.

La méthode getOutgoingLinksNamed renvoie les méta-relations dont la source est la méta-entité courante et dont le nom est linkName.

La méthode getAllOutgoingLinksNamed renvoie les méta-relations dont la source est la méta-entité courante et dont le nom est linkName en tenant compte de l'héritage.

La méthode getOutgoingLinksToNodesOfType renvoie les méta-relations dont la source est la méta-entité courante et dont la destination est la méta-entité référencée par Node.

La méthode getAllOutgoingLinksToNodesOfType renvoie les méta-relations dont la source est la méta-entité courante et dont la destination est la méta-entité référencée par Node en tenant compte de l'héritage.

La méthode getOutgoingLinkToNodesOfType renvoie **la** méta-relation dont la source est la méta-entité courante, dont le nom est linkName et dont la destination est la méta-entité référencée par Node.

La méthode getPossibleTypesForLink renvoie les types possibles (des entités de type **EntityType** implémentant par conséquent SemNodeNode) des entités sNets qui seraient liées à l'entité courante via un lien name.

La méthode getAllPossibleTypesForLink renvoie les types possibles (des entités de type **EntityType** implémentant par conséquent SemNodeNode) des entités sNets qui seraient liées à l'entité courante via un lien name en tenant compte de l'héritage.

La méthode getSourceLinks renvoie toutes les méta-entités dont le type destination est l'entité courante.

La méthode getAllSourceLinks renvoie toutes les méta-entités dont le type destination est l'entité courante (ou l'un de ses super-types).

La méthode getSubtypes renvoie toutes les méta-entités représentant des sous-types de l'entité courante.

La méthode getSupertypes renvoie toutes les méta-entités représentant des super-types de l'entité courante.

La méthode inherits prenant une chaine de caractères en paramètre renvoie vrai si l'entité courante est sous-type d'une entité nommée nodeName. La méthode inherits prenant une méta-entité en paramètre renvoie vrai si l'entité courante est sous-type de l'entité référencée par node.

La méthode newNodeIn crée une nouvelle entité sNets dont le type sera la méta-entité courante dans l'univers désigné par universe. Le nom attribué a cette nouvelle entité est a<TypeName> ou TypeName désigne le nom de la méta-entité courante.

De la même façon, nous avons défini une interface SemLinkNode permettant de manipuler simplement une entité sNets de type **RelationType**.

4.3.3 Implémentation d'une méta-relation sNets.

Toute méta-relation sNets (entité sNets de type **RelationType**) se doit d'implémenter l'interface suivante :

public interface SemLinkNode **extends** SemNode {

```
        public SemNodeNode getIncomingType();
        public SemNodeNode getDestinationType();
        public int getMultiplicity();
        public SemLinkNode getInverseLink();
}
```

Cette interface hérite de l'interface SemNode car toute méta-relation sNets est également une entité sNets. Les méthodes qu'elle définit facilitent la manipulation d'une méta-relation sNets mais toutes les fonctionnalités définies dans cette interfaces pourraient être obtenues en se contentant de manipuler une méta-relation sNets via l'interface SemNode. En effet, si l'on prend l'exemple de la méthode getMultiplicity() qui renvoie la cardinalité maximale de la relation, elle pourrait être obtenue de la même façon en invoquant la méthode getNodeForLink ("multiplicity") car une méta-relation sNets est liée à sa cardinalité via un lien présenté précédemment et appelé *multiplicity* (c.f. chapitre 4.1.5*)*.

La méthode getIncomingType renvoie la méta-entité source de cette méta-relation.

La méthode getDestinationType renvoie la méta-entité cible de cette méta-relation.

La méthode getMultiplicity renvoie la cardinalité maximale de la méta-relation.

La méthode getInverseLink renvoie la méta-relation représentant l'inverse de la méta-relation courante si elle existe.

La dernière interface que nous avons défini est l'interface SemUniverseNode permettant de manipuler simplement une entité sNets de type **Universe**.

4.3.4 Implémentation d'un univers sNets.

Tout univers sNets (entité sNets de type **Universe**) se doit d'implémenter l'interface suivante :

```
public interface SemUniverseNode extends SemNode {

        public Iterator getExtendedUniverses();
        public Iterator getExtensions();
```

```
        public SemNode getNodeNamed(String nodeName);
        public Iterator getNodes();
        public SemUniverseNode getSem();
        public void setSem(SemUniverseNode semUniverse);
        public Iterator getUniverses();
        public SemNode newNodeOfType(SemNodeNode node);
}
```

Cette interface hérite de l'interface SemNode car tout univers sNets est également une entité sNets. Les méthodes qu'elle définit facilitent la manipulation d'un univers sNets mais toutes les fonctionnalités définies dans cette interfaces pourraient être obtenues en se contentant de manipuler un univers sNets via l'interface SemNode. En effet, si l'on prend l'exemple de la méthode getSem() qui renvoie l'univers sémantique de l'univers, il pourrait être obtenu de la même façon en invoquant la méthode getNodeForLink ("sem") car un univers sNets est lié à son univers sémantique via un lien présenté précédemment et appelé *sem* (c.f. chapitre 4.1.8.1*)*.

La méthode getExtendedUniverses renvoie les univers étendus par l'univers courant.

La méthode getExtensions renvoie les univers qui étendent l'univers courant.

La méthode getNodeNamed renvoie l'entité appartenant à l'univers courant et nommée nodeName.

La méthode getNodes renvoie toutes les entités de l'univers courant.

Les méthodes getSem et setSem permettent d'accéder et de modifier l'univers sémantique de l'univers courant. **Attention** car aucun vérification n'est effectuée lors de la modification de cette propriété et il est fortement déconseillé d'utiliser la méthode setSem à un autre moment qu'à la création d'un univers.

La méthode getUniverses renvoie les univers définis dans l'univers courant.

La méthode newNodeOfType crée une nouvelle entité sNets dans l'univers courant et dont le type est désigné par node. Le nom attribué a cette nouvelle entité est a<TypeName> ou TypeName désigne le nom de la méta-entité node.

4.3.5 Un exemple d'utilisation de ces interfaces.

Nous allons montrer ici comment il est possible d'utiliser ces différentes interfaces de manière à :

- construire un méta-modèle à objets puis,
- y définir les concepts de classe, d'attribut et de relation classe-attribut,
- construire un modèle à objets suivant ce méta-modèle puis,
- y définir une classe Rectangle avec un attribut largeur et un attribut hauteur.

4.3.5.1 Création d'un univers sémantique (ou méta-modèle) "MetaModeleObjet".

Nous devons déjà disposons d'une référence sur un sNets que l'on appelera sNets et qui va correspondre à l'univers racine de notre formalisme. Cet univers contient tous les autres. L'utilisation des interfaces pour créer le méta-modèle MetaModeleObjet est alors la suivante :

```
// obtention d'une référence sur le méta-méta-modèle de notre formalisme (appelé
Semantic) :

SemUniverseNode semantic = (SemUniverseNode) sNets.getNodeNamed("Semantic");

// Obtention de la méta-entité Universe défini dans notre méta-méta-modèle et
permettant la définition d'un nouvel univers :

SemUniverseNode universe =
(SemUniverseNode)semantic.getNodeNamed("Universe");

// création d'un nouvel univers (ou modèle) nommé "MetaModeleObjet"

SemUniverseNode objectSem = (SemUniverseNode) sNets.newNodeOfType(universe);
objectSem.setName("MetaModeleObjet");

// Attribution de la sémantique définie par notre méta-méta-modèle à cet  univers de
sorte qu'il définisse bien un univers sémantique (ou méta-modèle) :

objectSem.setSem(semantic);
```

Voilà, nous avons ainsi créé un nouvel univers sémantique appelé "MetaModeleObjet" et suivant lui même la sémantique définie par notre méta-méta-modèle.

4.3.5.2 Définition des types Classe et Attribut dans ce méta-modèle.

L'utilisation des interfaces pour créer les méta-entités Classe et Attribut est alors la suivante :

```
// obtention d'une référence sur la méta-entité EntityType définie dans notre méta-méta-modèle et permettant la définition d'un nouveau type d'entité :

SemNodeNode entityType = (SemNodeNode) semantic.getNodeNamed("EntityType");

// Définition du type Classe dans le méta-modèle MetaModeleObjet :

SemNodeNode classType = (SemNodeNode)objectSem.newNodeOfType(entityType);
classType.setName("Classe");

// Définition du type Attribut dans le méta-modèle MetaModeleObjet :

SemNodeNode attributeType =
(SemNodeNode)objectSem.newNodeOfType(entityType);

attributeType.setName("Attribut");
```

Voilà, nous avons ainsi créé deux nouvelles méta-entités de type EntityType dans le méta-modèle désigné par objectSem.

4.3.5.3 Définition d'une relation bidirectionelle entre Classe et Attribut.

Nous allons créer une méta-relation attributs de Classe vers Attribut, une méta-relation classe de Attribut vers Classe, puis nous rattacherons ces méta-relations par un lien inverse de manière à préciser qu'elles ne forment qu'une seule relation bi-directionnelle. L'utilisation des interfaces est alors la suivante :

// obtention d'une référence sur la méta-entité RelationType définie dans notre méta-méta-modèle et permettant la définition d'un nouveau type de relation :

```
SemNodeNode relationType = (SemNodeNode)
semantic.getNodeNamed("RelationType");

// Définition de la méta-relation attributs entre Classe et Attribut dans le méta-modèle
MetaModelObjet :

SemLinkNode attributesRelation =
(SemLinkNode)objectSem.newNodeOfType(relationType);
attributesRelation.setName("attributs");

// Cardinalité maximum de la relation : n (représentée par -1 dans notre implémentation
:
attributesRelation.setLinkToNode("multiplicity", -1);

// Mise en relation des méta-entités Classe et Attribut et de la méta-relation attributs :

classType.setLinkToNode("outGoing", attributesRelation);
attributesRelation.setLinkToNode("destination", attributeType);

// Définition de la méta-relation classe entre Attribut et Classe dans le méta-modèle
MetaModelObjet :

SemLinkNode classRelation =
(SemLinkNode)objectSem.newNodeOfType(relationType);
attributesRelation.setName("classe");

// Cardinalité maximum de la relation : 1 (un attribut appartient à une classe) :

attributesRelation.setLinkToNode("multiplicity",1);

// Mise en relation des méta-entités Attribut et Classe et de la méta-relation classe :
```

```
attributeType.setLinkToNode("outGoing", classRelation);
classeRelation.setLinkToNode("destination", classType);

// Mise en relation des deux méta-relation créées de manière à indiquer qu'elles ne
forment qu'un seule relation :

attributesRelation.setLinkToNode("inverse", classRelation);
```

Voilà, nous avons ainsi créé une relation bidirectionnelle attributs/classe entre les méta-entités Classe et Attribut dans le méta-modèle désigné par objectSem.

4.3.5.4 Création d'un univers suivant cette sémantique.

La création de ce modèle s'effectue de façon similaire à la création du méta-modèle. La seule différence est que l'univers sémantique du méta-modèle est notre méta-méta-modèle (l'univers semantic) tandis que l'univers sémantique de notre modèle à objets est le méta-modèle que l'on vient de créer. L'utilisation des interfaces pour créer le modèle ModeleObjet est la suivante :

```
// obtention d'une référence sur le méta-modèle objet que l'on vient de créer :

SemUniverseNode objectSem = (SemUniverseNode)
sNets.getNodeNamed(MetaModeleObjet");

// Obtention de la méta-entité Universe défini dans notre méta-méta-modèle et
permettant la définition d'un nouvel univers :

SemUniverseNode universe =
(SemUniverseNode)semantic.getNodeNamed("Universe");

// création d'un nouvel univers (ou modèle) nommé "ModeleObjet"

SemUniverseNode objectModel =
(SemUniverseNode) sNets.newNodeOfType(universe);

objectSem.setName("ModeleObjet");
```

```
// Attribution de la sémantique définie par notre méta-modèle à objets à cet  univers de sorte qu'il définisse bien un modèle à objets :

objectModel.setSem(objectSem);
```

Voilà, nous avons ainsi créé un nouvel univers appelé "ModeleObjet" et suivant la sémantique définie par notre méta-modèle à objets.

4.3.5.5 Création d'entités de type Classe et Attribut.

L'utilisation de ces interfaces pour créer une entités de type Classe et deux Attribut dans le modèle que l'on vient de créer est alors la suivante :

```
// obtention d'une référence sur la méta-entité Classe définie dans le méta-modèle à objets :

SemNodeNode classType = (SemNodeNode) objectSem.getNodeNamed("Classe");

// Définition de l'entité Rectangle de type Classe dans le modèle ModelObjet :

SemNode rectangle = (SemNode) objectModel.newNodeOfType(classType);
rectangle.setName("Rectangle");

// obtention d'une référence sur la méta-entité Attribut définie dans le méta-modèle à objets :

SemNodeNode attributeType = (SemNodeNode) objectSem.getNodeNamed("Attribut");

// Définition de deux attributs dans le modèle ModelObjet :

SemNode largeur = (SemNode) objectModel.newNodeOfType(attributeType);
largeur.setName("Largeur");

SemNode hauteur = (SemNode) objectModel.newNodeOfType(attributeType);
```

```
hauteur.setName("Hauteur");

// Mise en correspondance de ces attributs avec la classe Rectangle via les méta-
relations définies dans le méta-modèle à objets :

rectangle.setLinkToNode("attributes", largeur);
rectangle.setLinkToNode("attributes", hauteur);
```

Voilà, nous avons ainsi créé trois entités dans notre modèle à objets et celui-ci est conforme au méta-modèle que nous avions défini. Nous avons, via ces quelques lignes de programmation, construit le modèle et le méta-modèle sNets représentés sur la figure suivante (Figure 77) :

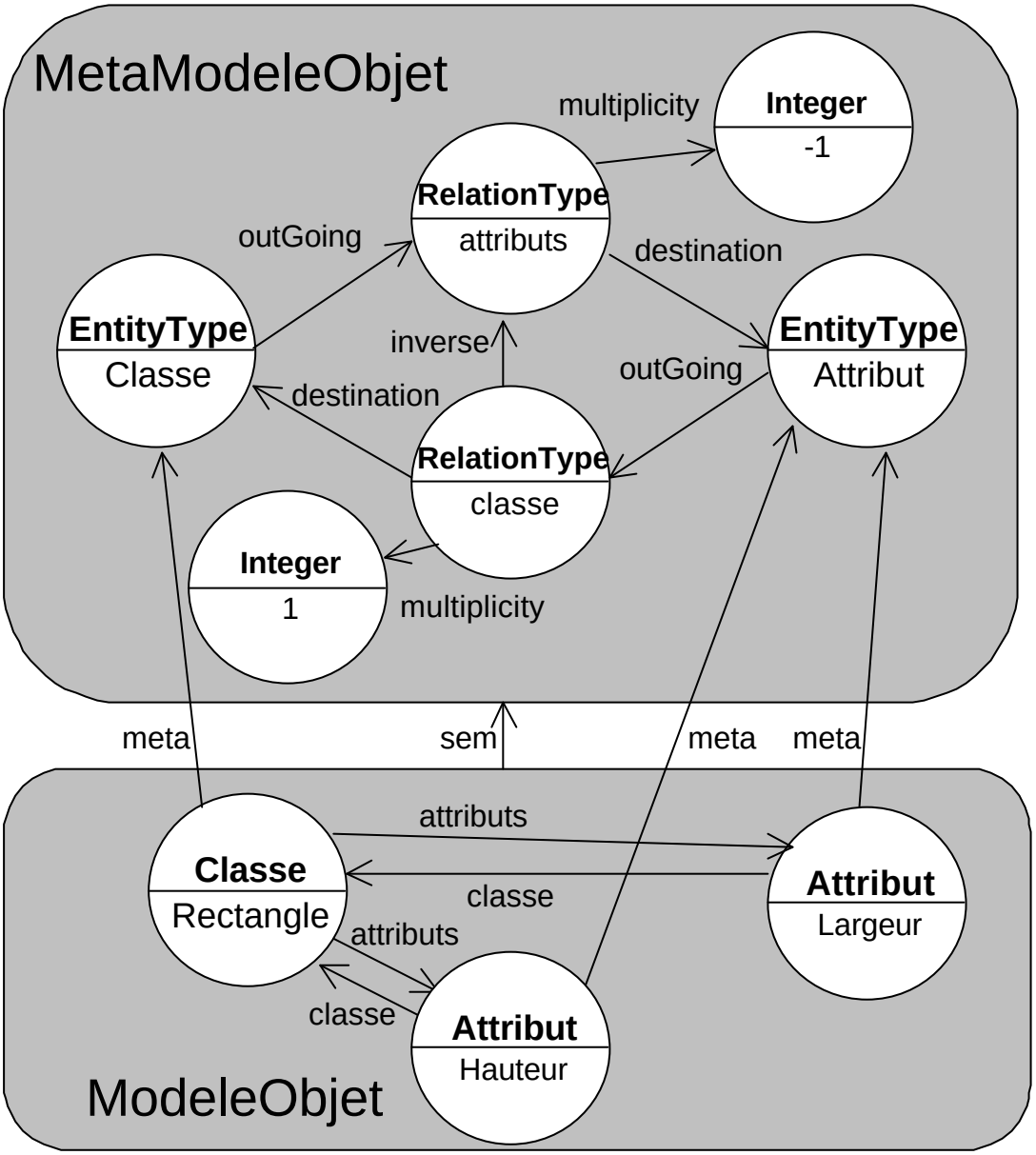


Figure 77 - Univers sNets obtenus par programmation.

5 Vers un méta-modèle réflexif.

La réflexivité est le nom donné au principe permettant à un système de contenir une représentation de lui-même de sorte qu'il puisse facilement s'adapter aux modifications de son environnement. La réflexivité et les techniques de méta-niveaux sont maintenant arrivés à maturité et ont été largement utilisés pour répondre à des problèmes réels dans les domaines des langages de programmation, des systèmes d'exploitation, des bases de données ou encore des environnements distribués. Ce chapitre présente comment ces même techniques peuvent être appliquées au domaine de la modélisation et montre que dans ce domaine également, la réflexivité peut jouer un rôle central. Il présente aussi les problèmes pratiques résultant de la multiplicité des modèles et des différentes ontologies dans les environnements de développement actuels ainsi que la nécessité de proposer des mécanismes de réflexivité dans les architecture de méta-modèles.

5.1 De la réflectivité des langages à la réflectivité des modèles.

La Figure 78 présente un browser (un navigateur) Smalltalk. En plus des cinq panneaux contenant les catégories, les classes, les protocoles, les méthodes et le corp des méthodes, l'une des principales caractéristiques de ce navigateur est la présence d'un bouton radio permettant de passer du niveau des objets à celui des classes. Dans le premier cas, le navigateur affiche les méthodes d'instance de la classe sélectionnée tandis que dans le second cas, ce sont les méthodes de classe de cette classe qui sont affichées.

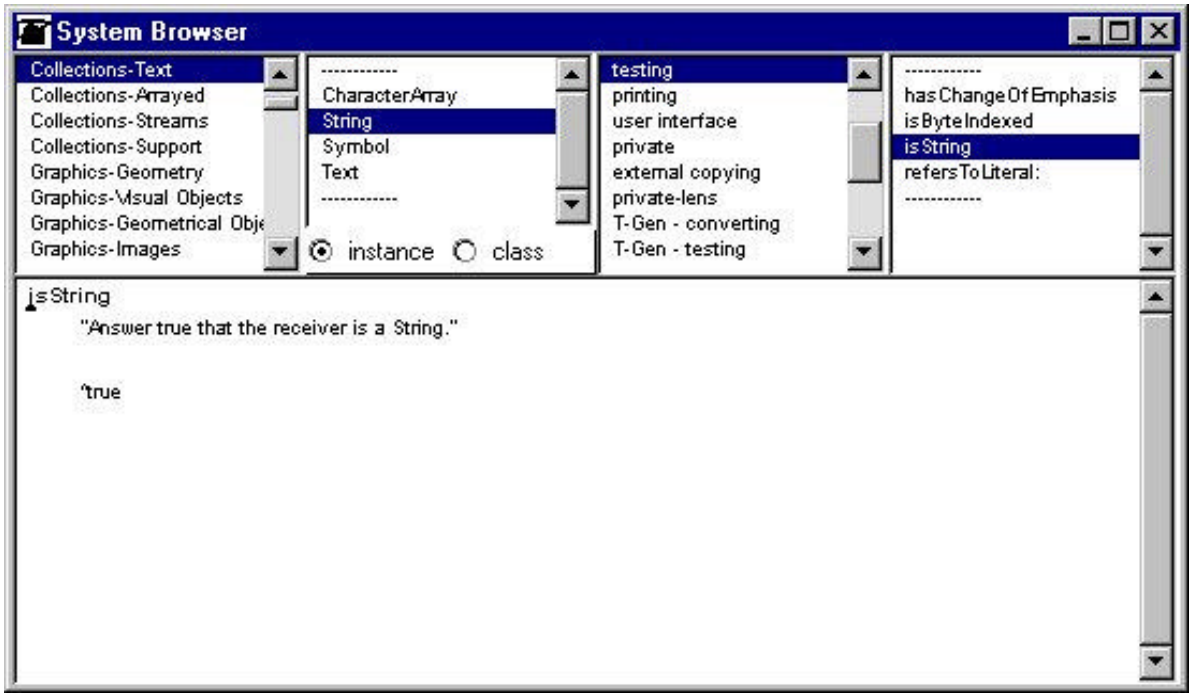


Figure 78 - Un browser Smalltalk avec son bouton radio permettant de passer du niveau des objets au niveau des classes.

On observe maintenant le même type de bouton radio dans les outils CASE les plus récents. Il permet alors d'indiquer si le diagramme UML présenté correspond à un modèle UML ou bien à un méta-modèle MOF. C'est à dire qu'il permet de passer du niveau de la modélisation (niveau M_1; c.f. Figure 6) à celui de la méta-modélisation (niveau M_2; c.f. Figure 6). Ces différents niveaux de modélisation présentant des similitudes, il était normal que des outils similaires et de procédures semblables puissent être employés à la fois pour manipuler des modèle et pour manipuler des méta-modèles. Prenons l'example de l'outil de représentation graphique des modèles : le fait d'être à même de représenter les méta-modèles MOF avec la notation UML permet de réutiliser naturellement les outils de modélisation graphique UML afin de définir des méta-modèles MOF.

Après avoir présenté les aspects réflexifs du MOF, nous en aborderons les limitations :

- absence de méta-entité représentant les méta-modèles,
- absence de méta-entité représentant les modèles,
- absence de relation entre un modèle et son méta-modèle,

- absence de relation entre une entité et sa méta-entité.

Nous reviendrons ensuite notre propre noyau réflexif de méta-modélisation puis nous tenterons d'effectuer une mise en correspondance de ce noyau avec ceux des formalismes MOF et CDIF.

5.2 Pourquoi le MOF devrait-il être réflexif ?

Après CORBA et UML, nous observons une troisième vague de propositions à l'OMG avec l'émergence du MOF. Une interpretation possible de ce phénomène peut être que l'ingénierie des modèles est progressivement en passe de devenir une technologie majeure. Les modèles et les méta-modèles sont maintenant reconnues comme des entités de première classe dans le processus de développement logiciel. L'émergence d'UML puis la prise de conscience qu'UML n'était que l'un des méta-modèles possibles du paysage du développement de logiciel ont ammené à la définition du MOF. En effet, face au danger de disposer d'un grand nombre de méta-modèles évoluant indépendamment les uns de autres et définis différemment (modèles objets, modèles de processus, modèles de workflow, modèles de données, etc…), il y avait un besoin urgent de définir un formalisme commun de manipulation de ces différents méta-modèles. Cette standardisation permet d'apporter des fonctionnalités communes de manipulation des différents modèles issus de ces méta-modèles. Le MOF à donc été défini comme un langage d'expression de méta-modèles. C'est sa principale fonction, mais certains outils utilise le MOF de manière à être indépendants d'un méta-modèles particulier. Ces outils peuvent alors manipuler des modèles UML de la même façon qu'ils manipules des modèles de workflow ou de modèles de données. Mais pour que cette opération soit applicable, il est nécessaire que le MOF propose des mécanismes de réflexivité afin qu'il soit possible pour un outil de prendre connaissance d'un méta-modèle au moment de son exécution.

5.3 MOF = UML + CDIF.

Le MOF a su s'inspirer du format de représentation de données CDIF. La communauté CDIF à travaillé pendant près de dix ans à l'élaboration d'un formalisme similaire basé sur cette même architecture à quatre niveaux (c.f. Figure 6) qui a été acceptée comme un schéma d'organisation pour les modèles, les méta-modèles ainsi que les méta-méta-modèles. CDIF est un standard EIA (Electronic Industry Associates) qui structure les domaines de connaissances en domaines

appelés "Subject areas". Le MOF peut être vu comme une fusion des principaux concepts de UML dans un schéma similaire à celui de CDIF.

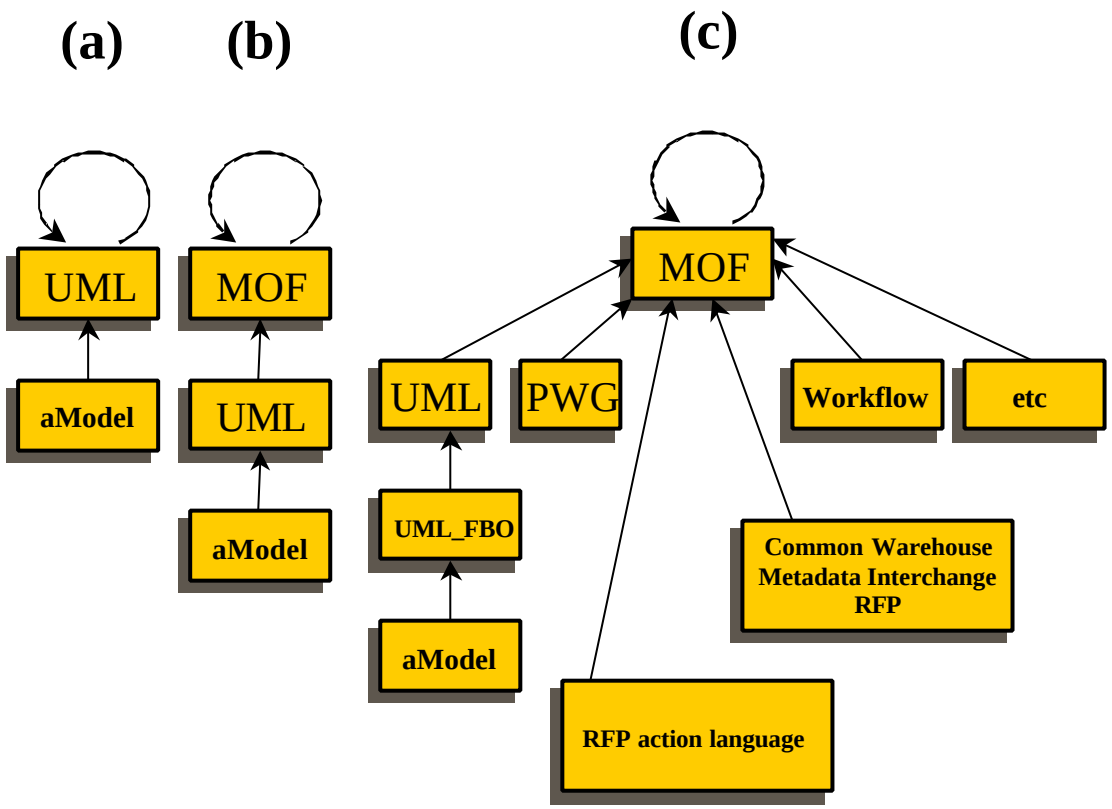


Figure 79 - D'une définition réflexive d'UML utilisé pour la modélisation objet à un ensemble de méta-modèles incluant UML et basé sur un MOF réflexif.

Intéressons nous maintenant plus précisément au sens de chacunes de relations décrites sur la Figure 79. La relation entre l'espace ontologique *UML_FBO* et *UML* est définitivement différente de la relation entre *UML* et *MOF*. Dans le premier cas, c'est une relation qui veut dire que UML_FBO est une extension (un sur-ensemble) de UML tandis que dans le second cas, la relation exprime le fait que UML est défini avec le MOF (le MOF est le méta-modèle de UML).

Par la suite, nous tentons d'expliciter toutes les relations que nous employons. La première relation sera donc nommée *extends* et exprimera le fait qu'un modèle peut être une extension d'un autre modèle et la seconde sera nommée *basedOn* afin d'exprimer le fait qu'un modèle est défini

à partir d'un méta-modèle. La situation décrite par la Figure 79-(b) correspond alors aux relations suivantes :

- *basedOn*(MOF, MOF)
- *basedOn*(UML, MOF)
- *extends*(UML_FBO, UML)

Comme on peut le constater, il est essentiel de définir précisément les relations potentielles entre modèles et méta-modèles ainsi que leur sémantique. La même opération doit également être effectuée entre les entités et leurs méta-entités. Or dans le contexte du MOF, les questions suivantes restent sans réponse :

- Qu'est-ce qu'un modèle ?
- Qu'est-ce qu'un méta-modèle ?
- Qu'est-ce qui lie un modèle à son méta-modèle (cette relation est appelée *basedOn* par la suite) ?
- Qu'est-ce qu'une entité ?
- Comment une entité est-elle rattachée à sa méta-entité (cette relation est appelée *instanceOf* par la suite) ?

5.4 Réflexivité & Lacunes du MOF.

Le MOF contient des mécanismes basiques pour définir de méta-entités (appelées Classe dans le MOF), des méta-relations (appelées Association dans le MOF) et des paquetages qui vont jouer le rôle de conteneur pour ces méta-entités et méta-relations. Ces mécanismes sont présentés en détails dans le chapitre 3.1.1. Un méta-modèle décrit avec le MOF sera généralement composé d'un paquetage principal qui contiendra toute la définition MOF de ce méta-modèle. Par conséquent, bien qu'il n'y ait pas de concept représentant les méta-modèles dans le MOF, un paquetage peut être utilisé dans ce but.

Avec le MOF, la seule relation définie entre les paquetages est la relation *extends* : un paquetage est un élément généralisable et on peut considérer que la relation de généralisation des paquetages MOF peut être utilisée dans le même but que la relation d'extension que nous avons introduite précédemment. Un paquetage peut donc être utilisé pour représenter un méta-modèle,

mais rien ne peut être utilisé pour représenter un modèle. Par conséquent, la relation *basedOn* ou son équivalent MOF, permettant de lier un modèle à son méta-modèle, n'existe pas.

De plus, bien que le concept de classe du MOF corresponde à la notion de méta-entité, aucun concept MOF ne peut être utilisé pour représenter une entité. Par conséquent, aucune relation ne peut être définie entre une entité et sa méta-entité.

Pour conclure sur les propriétés réflexives du MOF, on remarque donc que deux des relations les plus importantes dans ce domaine sont absente de sa définition. Ces deux relations sont la relation *basedOn* permettant de lier un modèle à son méta-modèle (cette relation est souvent appelée *instanceOf* par abus de langage, mais nous avons choisi de ne pas utiliser ce terme afin de ne pas générer de confusion avec la seconde relation) et la relation *instanceOf* permettant de lier une entité à sa type représenté par une méta-entité. La première relation indique qu'un modèle est défini par un méta-modèle (i.e. que toutes les entités d'un modèle doivent trouver leur type dans le méta-modèle de celui-ci. Tandis que la seconde relation indique qu'une entité dispose d'un type représenté par une méta-entité. Le sens de ces deux relations est différent et le fait de ne pas les expliciter génère énormément de confusion. entraine la confusion. De nombreux formalismes ne font pas de différence entre ces deux relations. Mais prenons l'exemple des deux phrases suivantes :

- "Le MOF est un méta-modèle réflexif"
- "UML est un méta-modèle MOF compliant"

Dans la première phrase, nous avons "Le MOF est un méta-modèle" qui se traduit par la relation suivante :

- *instanceOf* (MOF, MetaModel)

Et nous avons "Le MOF est réflexif", ce qui veux dire que son méta-modèle est lui-même et qui se traduit par la relation suivante :

- *basedOn* (MOF, MOF)

Si le sens de ces deux relation était identique, nous n'aboutirions pas sur deux relation différentes. De même, la deuxième phrase se traduit par les deux relation suivantes :

- *instanceOf* (UML, MetaModel)
- *basedOn* (UML, MOF)

Nous avons donc bien deux relations distinctes et il nous semble par conséquent essentiel de clairement définir tous les concepts ou méta-entités nécessaires en méta-modélisation ainsi que toutes les relations potentielles entre ces différents concepts. Cette opération devrait permettre de supprimer toute confusion sur le sens que l'on attribue à ces relations.

5.5 Un noyau réflexif dédié à la méta-modélisation.

Nous présentons ici les éléments qui nous semblent essentiels pour définir précisément un méta-modèle réflexif dédié à la méta-modélisation. Ce méta-modèle sera ensuite comparé au MOF, à CDIF, ainsi qu'a notre propre formalisme (les sNets). Pour nous, un tel méta-modèle se doit de définir les concepts et les relations suivantes :

- le concept d'entité,
- le concept de méta-entité,
- la relation instanceOf entre une entité et sa méta-entité,
- le concept de modèle,
- le concept de méta-modèle,
- la relation basedOn entre un modèle et son méta-modèle,
- la relation definedIn entre une entité et le modèle dans lequel elle est définie,
- et la relation definedIn entre une méta-entité et le méta-modèle dans lequel elle est définie.

La Figure 80 regroupe tous ces éléments dans un méta-méta-modèle. La notation utilisée pour représenter ce méta-modèle est UML.

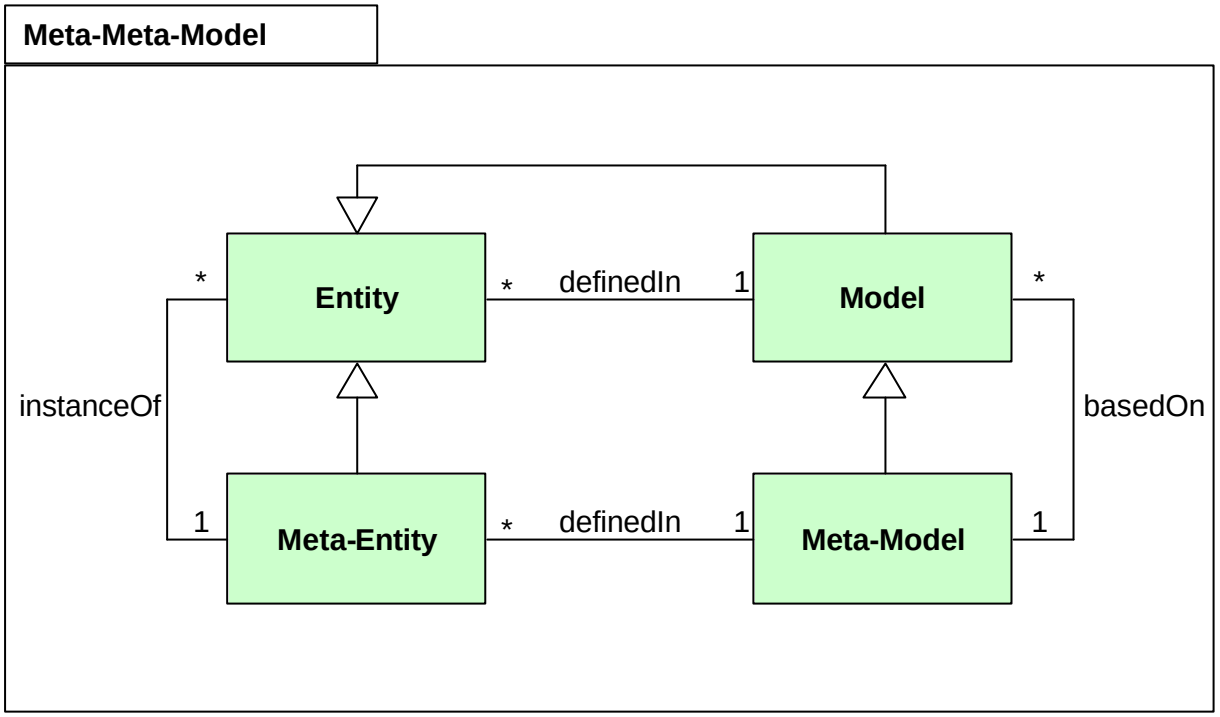


Figure 80 - L'essentiel d'un méta-méta-modèle réflexif.

Dans ce méta-méta-modèle, nous considérons que tout est entité. Ainsi, un méta-modèle est un modèle (un sous-type de modèle : un méta-modèle n'est autre qu'un modèle de modèle) qui lui-même est une entité. De même, une méta-entité est une entité (un sous-type d'entité : une méta-entité n'est autre qu'une entité qui va permettre de décrire un type d'entité). C'est la relation d'héritage qui va nous permettre de formaliser cette propriété.

Pour plus de lisibilité, la méta-entité représentant une méta-relation (appelée **Meta-Relationship**) n'a pas été représentée sur cette figure. Mais sur celle-ci, les rectangles représentent des méta-entités tandis que les relations entre les rectangles représentent des méta-relations. Les méta-relations on une relation *hasSource* vers la méta-entité source de la relation et une relation *hasDestination* vers la méta-entité cible de la relation. De même la méta-relation définissant l'héritage n'a pas été représentée sur la Figure 80. Cette méta-relation a pour source et pour destination une méta-entité. Ces différents éléments n'étant pas essentiels pour traiter de l'aspect réflectivité du modèle, ils ne seront pas abordés dans la suite de ce chapitre. Par contre, la Figure 81 présente, pour information, ce même méta-modèle complété de ces différents éléments.

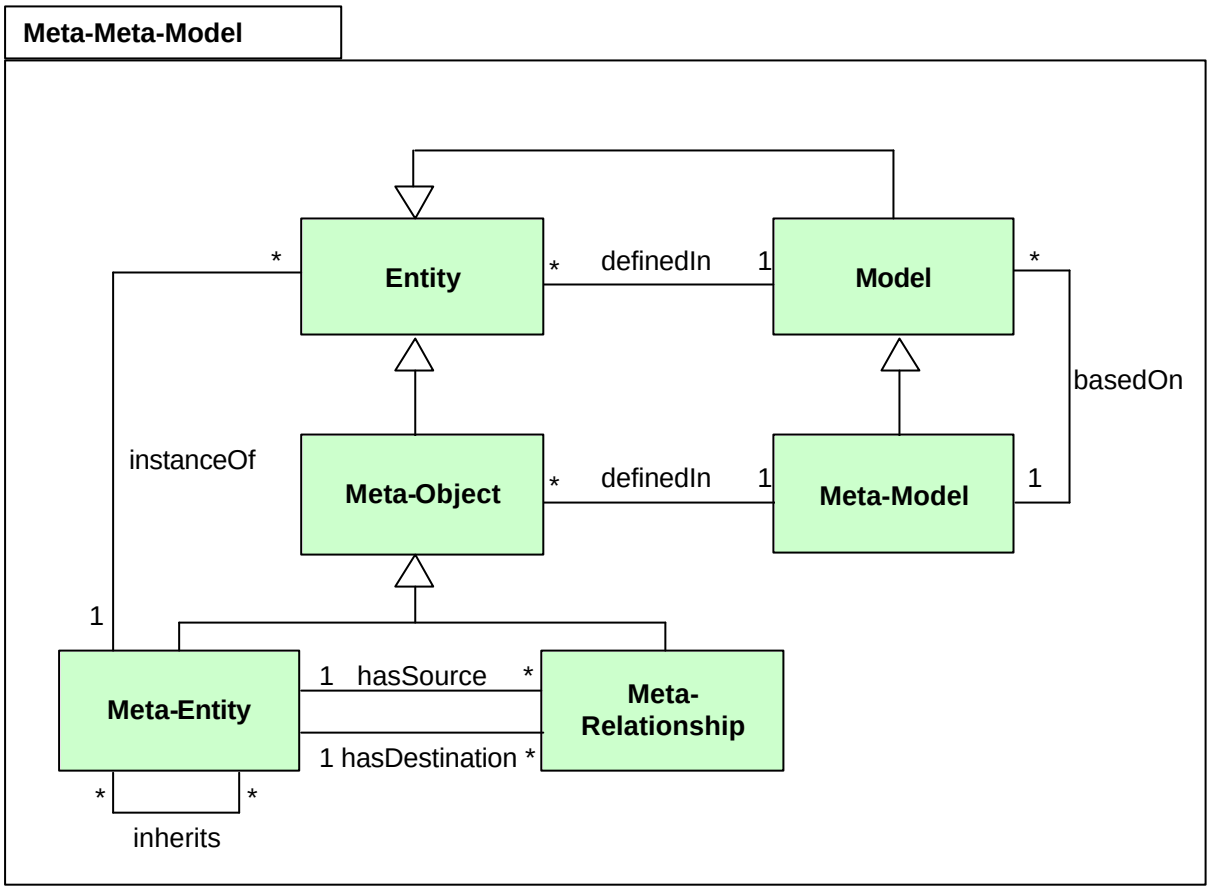


Figure 81 - L'essentiel d'un méta-méta-modèle réflexif (incluant méta-relation et relation d'héritage).

En fait, un tel méta-modèle doit définir précisément les concept et les relations suivantes (respectivement via des méta-entités et des méta-relations) :

- les concepts : Entity, Model, Meta-Entity, Meta-Model et Meta-Relationship,
- et les relations : *instanceOf* entre Entity et Meta-Entity et *basedOn* entre Model et Meta-Model

Les relations *instanceOf* suivantes sont alors présentent dans ce méta-modèle (de sorte qu'il puisse effectivement être réflexif) :

- *instanceOf* (Entity, Meta-Entity)
- *instanceOf* (Meta-Entity, Meta-Entity)
- *instanceOf* (Model, Meta-Entity)

- *instanceOf* (Meta-Model, Meta-Entity)
- *instanceOf* (instanceOf, Meta-Relationship)
- *instanceOf* (basedOn, Meta-Relationship)
- *instanceOf* (definedIn, Meta-Relationship)
- *instanceOf* (Meta-Meta-Model, Meta-Model)

De plus, la relation *basedOn* est également utilisée de la façon suivante pour exprimer le fait que ce méta-méta-modèle est réflexif :

- *basedOn* (Meta-Meta-Model, Meta-Meta-Model)

Toute entité dispose d'un lien *instanceOf* vers sa méta-entité. Cela montre clairement la différence entre la relation *instanceOf* et la relation *basedOn*. Par conséquent, lorsque l'on dit, par abus de langage, qu'un modèle est une instance d'un méta-modèle, on veux parler de la relation *basedOn*, pas de la relation *instanceOf*. Un modèle dispose d'un lien *instanceOf* vers la méta-entité Model (un modèle "est un" modèle) tandis qu'il dispose d'un lien *basedOn* vers son méta-modèle (un modèle "est défini par un" meta-modèle). Ceci est primordial, car les architectures de modélisation à quatre niveaux sont toutes basées sur de telles relations non explicites.

5.6 Les noyaux réflexifs de différents formalismes de modélisation.

Nous allons maintenant présenter comment les entités des méta-méta-modèles des principaux formalismes de méta-modélisation peuvent être mises en correspondance avec les entités du noyau réflexif que nous venons de proposer. Ces formalismes sont le bien évidemment le MOF, son précurseur CDIF, et notre propre formalisme les sNets.

5.6.1 Le noyau réflexif du MOF.

Le MOF dispose d'un méta-méta-modèle réflexif qui a déjà été décrit dans le chapitre 3.1. Vous pouvez vous référer à la Figure 18 pour avoir une vue générale du contenu de son méta-méta-modèle.

Une méta-entité MOF est définie par une entité de type Class et un méta-modèle est généralement constitué d'une entité de type Package. La relation entre une classe et son

paquetage (l'équivalent de notre relation *definedIn* entre une méta-entité et son méta-modèle) est définie par l' association MOF appelée contains (c.f. chapitre 3.1.2.2). En effet, cette relation est définie entre un espace de noms (un paquetage MOF est un espace de noms) et un élément de modélisation (une classe MOF est un élément de modélisation).

Ces trois éléments (la méta-entité Class, la méta-entité Package et l'association Contains) sont les seuls éléments du MOF que l'on peut mettre en correspondance avec des entités du noyau réflexif que nous avons proposé en Figure 80. Le MOF ne propose pas de concepts pouvant correspondre aux notions de modèle et d'entité ainsi qu'aux relations entre entités et méta-entités (notre relation *instanceOf*), entre entité et modèle (notre relation *definedIn*) et entre modèles et méta-modèles (notre relation *basedOn*).

Mais le MOF définit, en plus de ce méta-méta-modèle, un paquetage appelé Reflective. L'utilisation de ce paquetage est présentée Figure 82. Ce paquetage doit être importé par tous les méta-modèles MOF compliant. Par conséquent, le méta-méta-modèle du MOF lui-même importe ce paquetage Reflective (le méta-méta-modèle du MOF étant lui-même MOF compliant).

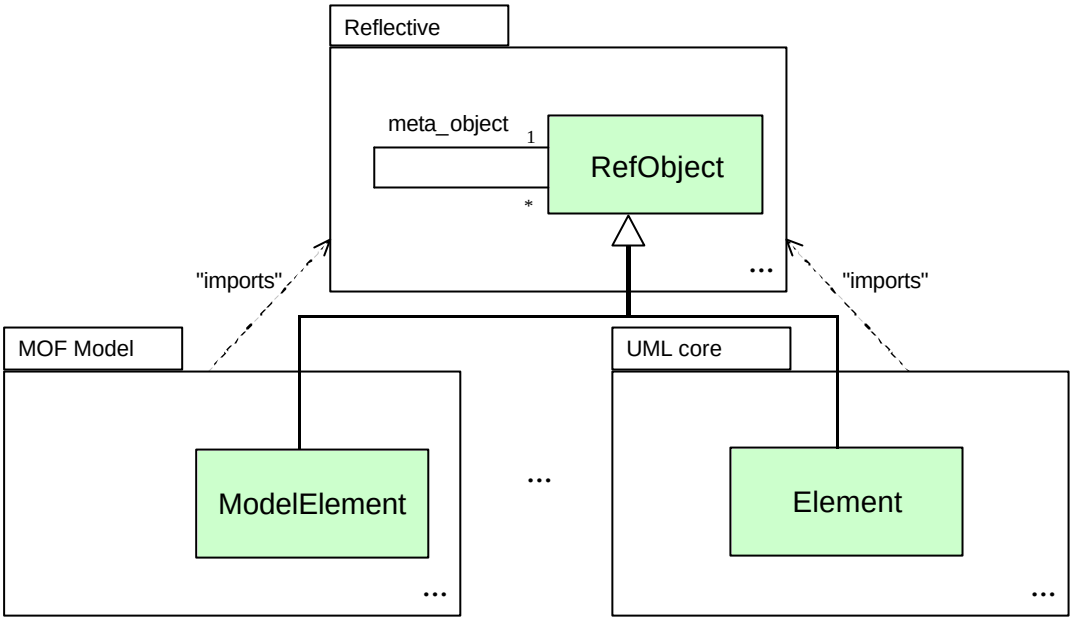


Figure 82 - Le paquetage Reflective du MOF sur lequel doivent s'appuyer tous les méta-modèles MOF compliant.

Ce paquetage contient une classe appelée RefObject et devant nécessairement être un super-type de toute les classes définies dans tous les modèles MOF compliant. De sorte que toute entité MOF puisse être au moins considérée comme une instance de RefObject. Par conséquent, cette classe RefObject peut être considérée comme notre méta-entité Entity (c.f. Figure 80).

Ainsi, sur la Figure 82, nous avons représenté ce paquetage Reflective ainsi que le paquetage MOF Model constituant le cœur du méta-méta-modèle du MOF et le paquetage UML core constituant le cœur du méta-modèle UML (méta-modèle MOF compliant). De plus, nous représentons le fait que la méta-entité racine "locale" du MOF appelée ModelElement, ainsi que la méta-entité racine "locale" de UML appelée Element sont toutes deux sous-type de RefObject.

Une autre particularité importante de ce paquetage est qu'il définit une méthode appelée meta_object sur la classe RefObject. . Cette méthode peut être considérée comme une association partant d'une entité de type RefObject et aboutissant à une autre entité de type

RefObject représentant le type de la première entité. C'est tout simplement l'équivalent de notre relation *meta* entre une entité et sa méta-entité.

Par contre, même en tenant compte de ce paquetage Reflective, le MOF ne définit pas de concept équivalent à notre concept de modèle.

5.6.2 Le noyau réflexif de CDIF.

Le second exemple est celui du format CDIF (Common Data Interchange Format) qui est basé sur le méta-méta-modèle décrit par la Figure 83.

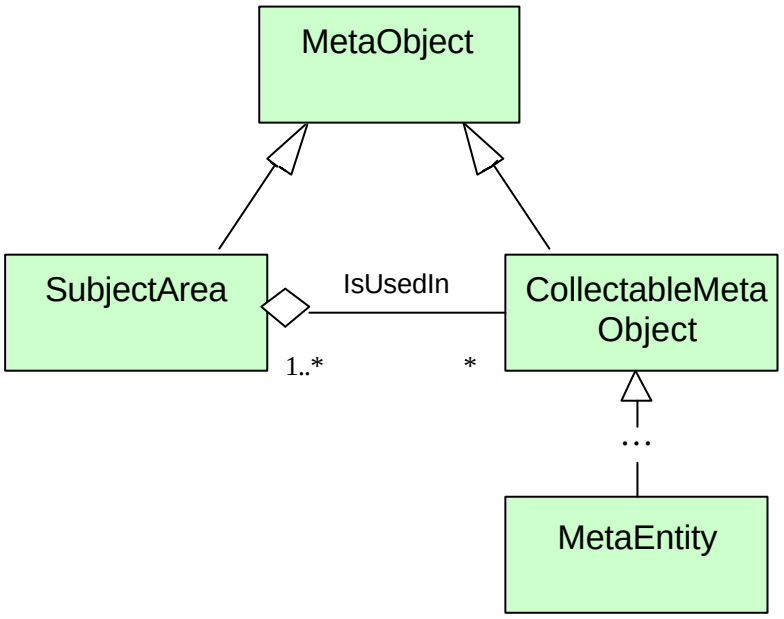


Figure 83 - Le cœur du méta-méta-modèle de CDIF.

A partir de cette figure, on peut voir qu'une méta-entité CDIF est définie en utilisant le type MetaEntity. De plus, un méta-modèle est défini par une entité de type SubjectArea, et la relation entre une méta-entité et le méta-modèle dans lequel elle est définie (notre relation *definedIn* entre Meta-Entity et Meta-Model) se nomme ici IsUsedIn. Une particularité de CDIF que l'on peut voir ici est de permettre la définition d'un type dans *n* méta-modèles (ou

SubjectArea). Par contre, lorsque l'on exploite cette particularité, il devient alors impossible de savoir dans quel méta-modèle une méta-entité à été initialement définie.

Si l'on se contente de ce méta-méta-modèle, aucun concept supplémentaire ne peut être mis en correspondance avec un concept défini dans le méta-méta-modèle que nous proposions Figure 80.

Par contre, lorsque l'on s'intéresse à l'utilisation qui est faite de ce méta-méta-modèle, on s'aperçoit qu'il existe des prérequis similaires au paquetage Reflective du MOF nécessaires à la définition de tout nouveau méta-modèle CDIF. En effet, dans l'ensemble des méta-modèles CDIF, appelé CDIF Integrated Meta-Model (c.f. Figure 84) et contenant toutes les SubjectAreas définies avec le méta-méta-modèle CDIF, un certain nombre de règles sont imposées.Un méta-modèle particulier appelé Foundation Subject Area y est défini et se doit d'être importé par tous les autres méta-modèles du CDIF Integrated Meta-Model (CDIF Framework for Modeling and Extensibility, EIA/IS-107) :

> "*The Foundation Subject Area provides the basic definitions which underpin the remainder of the CDIF Integrated Meta-model. It consists of an AttributableMetaObject called RootObject, which is purely abstract, and acts as the root of the Attributable-MetaObject Hierarchy*."

Ce méta-modèle définit donc une méta-entité appelé RootObject et devant être le super-type de toutes les méta-entités (de la même façon que RefObject dans le paquetage Reflective du MOF). Par conséquent, cette méta-entité peut être mise en correspondance avec notre méta-entité Entity définie sur la Figure 80.

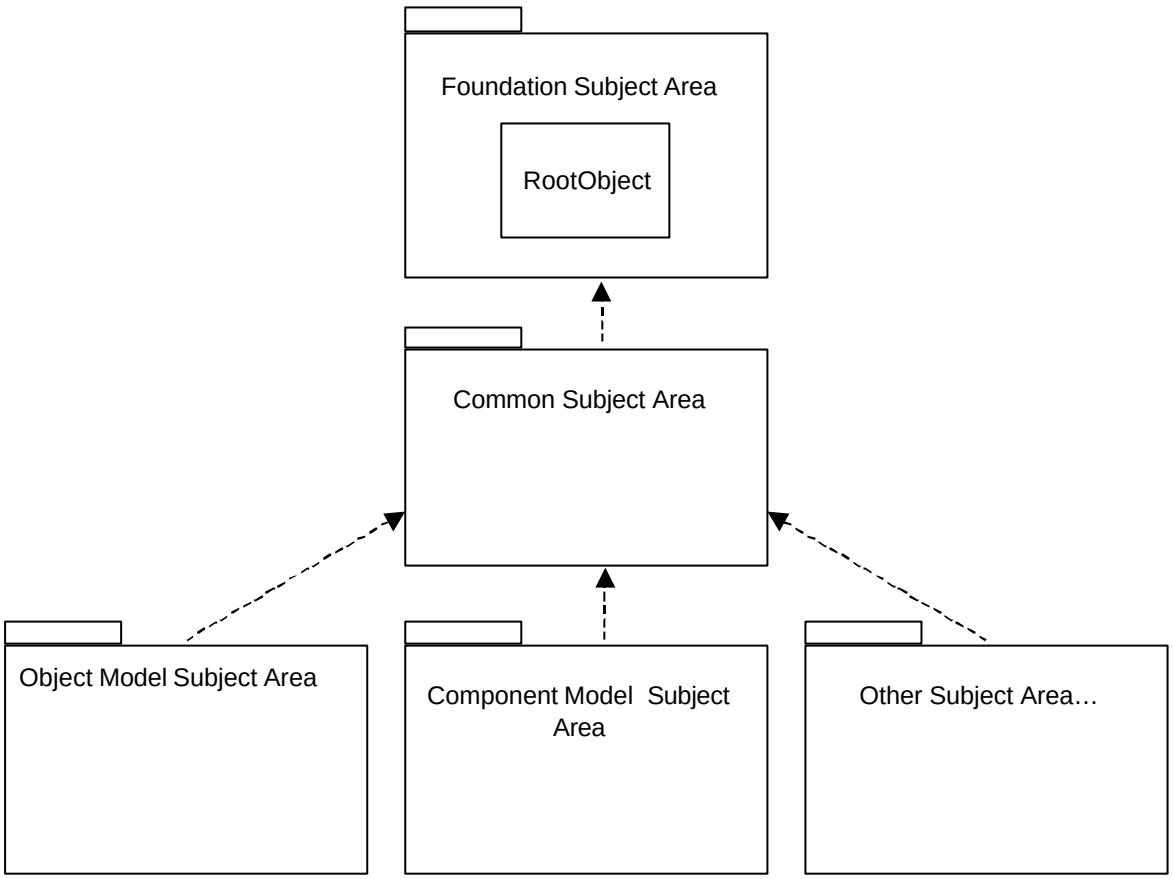


Figure 84 - CDIF Integrated Meta-Model.

Par conséquent, CDIF tout comme le MOF ne définissent pas la notion de modèle l'on ne trouve donc pas dans ces formalismes le moyen de lier une entité à son modèle ou encore de lier un modèle à son méta-modèle.

5.6.3 Le noyau réflexif des sNets.

Le formalisme des sNets est basé sur celui des réseaux sémantiques (c.f. chapitre 0). Le noyau réflexif de ce formalisme est représenté Figure 85.

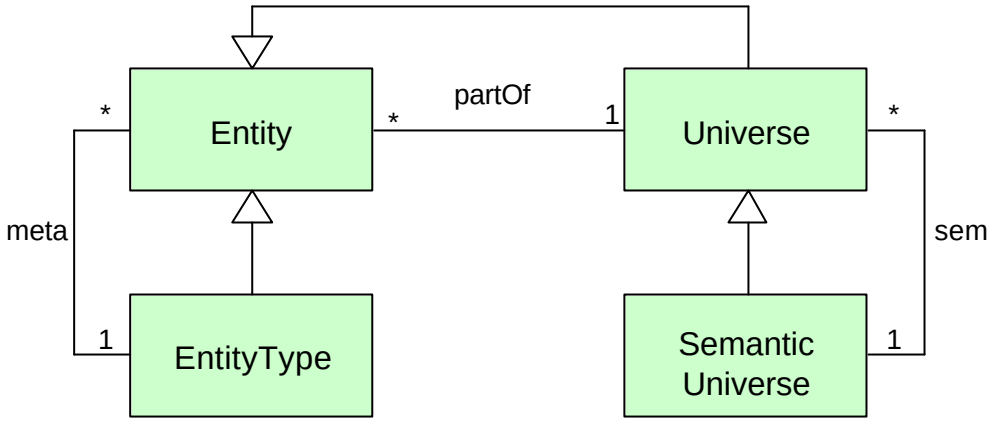


Figure 85 - Le cœur du méta-méta-modèle des sNets (notation UML).

Dans le formalisme des sNets, tout est entité. Toute entité dispose d'un type représenté lui-même par une méta-entité (une entité de type EntityType). Toute méta-entité est sous-type de la méta-entité Entity représentée sur la Figure 85. Par conséquent, toute entité dispose d'un lien *meta* vrs l'entité de type EntityType représentant son type et d'un lien *partOf* vers l'entité de type Universe représentant son modèle. Par conséquent, le concept d'entité existe dans les sNets et s'appelle Entity. De plus, les méta-entités sont des entités de type EntityType, la notion d'Univers correspond directement à la notion de modèle et la relation *meta* correspond à la relation *instanceOf* présentés sur la Figure 80. La relation *partOf* entre une entité sNet et son univers va quant à elle correspondre à la relation *definedIn* entre une entité et son modèle.

La notion de méta-modèle est également présente et s'appelle ici "univers sémantique". Un univers sémantique est un sous-type d'univers de la même façon qu'un méta-modèle est un modèle particulier. De plus tout univers sNets dispose d'un univers sémantique et la relation entre un univers et son univers sémantique est matérialisée par une relation nommée *sem*. Cette relation est l'équivalent de la relation basedOn que nous proposions sur la Figure 80 pour exprimer la relation entre un modèle et son méta-modèle.

Dans le formalisme des sNets, les seules entités qu'un univers (ou modèle) peut contenir sont celles dont les types sont définies dans son univers sémantique (ou méta-modèle). De la même façon, les seules méta-entités qu'un univers sémantique peut contenir sont celles dont les types sont définis dans son propre univers sémantique (ou méta-méta-modèle). Par conséquent, seules des méta-entités pourront être définies dans les univers sémantiques et la relation *partOf* entre

entité et univers pourra également jouer le rôle de la relation *definedIn* entre méta-entités et méta-modèles. Le formalisme des sNets est alors le seul des trois formalismes présentés à pouvoir proposer une représentation de chacuns des concepts et relation proposés par la Figure 80.

5.6.4 Comparaison des concepts principaux de ces différents formalismes.

La Figure 86 présente un résumé des principaux concepts définis dans ces différents formalismes.

	MOF seul	MOF & Reflective	CDIF	sNets
Méta-entités:				
Entity	N/A	Reflective::RefObject	MetaEntity "RootObject"	Méta-Entité "Entity"
Meta-Entity	Class "Class"	Class "Class"	MetaEntity "MetaEntity"	Méta-Entité "EntityType"
Model	N/A	N/A	N/A	Méta-Entité "Universe"
Meta-Model	Class "Package"	Class "Package"	MetaEntity "SubjectArea"	Méta-Entité "SemanticUniverse"
Méta-relations:				
instanceof	N/A	meta_object() définie dans l 'interface Reflective:: RefBaseObject	N/A	Méta-Relation "meta"
definedIn (entre méta-entités & méta-modèles)	Association "contains"	Association "Contains"	Meta-Relationship "IsUsedIn"	Méta-Relation "partOf"
definedIn (entre entités & modèles)	N/A	N/A	N/A	
basedOn	N/A	N/A	N/A	Méta-Relation "sem"

Figure 86 - Comparaison des concepts principaux de ces différents formalismes (MOF, CDIF, sNets).

Les seuls concepts communs à chacun de ces formalismes sont les concepts de méta-entité et de méta-modèle. Tandis que la seule relation que l'on retrouve dans tous ces formalismes est la relation definedIn entre les méta-entités et leur méta-modèle.

La relation *instanceOf* permettant de lier une entité à sa méta-entité et nous semblant vraiment essentiel pour concevoir un framework de modélisation et de méta-modélisation n'est présente que dans les sNets et dans le MOF (avec son paquetage Reflective).

En comparant ces trois architecture (CDIF : un standard qui n'évoluera plus; les sNets : un prototype de recherche et MOF : une recommandation industrielle), nous avons été plus à même de comprendre l'impact des techniques de réflectivité dans l'ingénierie des modèles. L'arrivée à maturité de techniques de modélisation précises et inter opérables va nous la possibilité de réaliser un framework régulier de modèles et de méta-modèles diverses et interconnectés dans lequel tout modèle sera basé sur un méta-modèle explicite et tout méta-modèle sera basé sur un seul et même méta-méta-modèle.

6 Modélisation, Méta-modélisation et niveaux d'abstraction.

Un problème récurrent en méta-modélisation consiste à identifier précisémment le nombre et le contenu des différents niveaux d'abstraction. Les niveaux M_3, M_2 et M_1 sont généralement bien acceptés. Le niveau M_0 pourra quant à lui ne pas être représenté lorsque l'on ne s'intéresse qu'aux modèles et méta-modèles. Enfin, pour certains, il existe ensuite un nombre infini de niveaux permettant de représenter les informations. Mais il faut garder à l'esprit que le nombre de ces niveaux dépend directement de la relation d'instanciation qui les traverse et qui lie les entités d'un niveau aux entité du niveau directement supérieur. Afin d'expliquer comment ces niveaux sont identifiés, nous présentons l'exemple de l'architecture classique à quatre niveaux référencée notamment par le MOF, UML et CDIF et déjà présentée Figure 6.

6.1 L'architecture classique à quatre niveaux.

Cette architecture est donc utilisée par le MOF, UML et CDIF. Les quatre niveaux qu'elle les définit sont représentés sur la Figure 87. A cette architecture est associée une relation d'instanciation. Celle-ci doit être clairement définie de manière à ce qu'elle permette de déterminer à quel niveau une entité appartient.

Etant donnée une telle relation *instanceOf*, nous avons alors les axiomes suivants :

- un niveau contient des entités,
- le fait qu'une entité appartienne à un niveau implique qu'elle dispose d'une relation *instanceOf* vers une entité du niveau supérieur (on considère alors que le niveau supérieur au niveau M_3 est le niveau M_3 lui-même car il est réflexif).

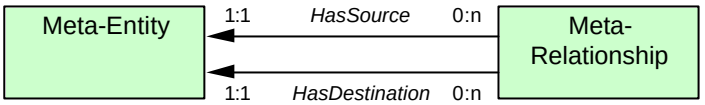
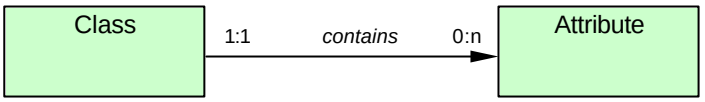
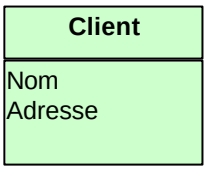
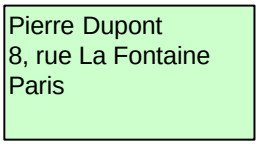
Niveau	Contenu	Exemple
Méta-Méta-Modèles (M_3)	Langage d'expression de méta-modèles	
Méta-Modèles (M_2)	Langage d'expression de modèles	
Modèles (M_1)	Langage d'expression d'information	
Information (M_0)	Information décrite dans les termes d'un modèle	

Figure 87 - L'architecture classique à quatre niveaux et la description de son contenu.

Dans les exemples présentés sur la Figure 87, nous avons un certain nombre d'entités et chacune d'elles dispose d'un lien *instanceOf* vers sa méta-entité (bien que ces relations ne soit pas explicites sur cette figure). Par conséquent, nous avons décidé d'expliciter ces relations d'instanciation sur la Figure 88 afin de nous permettre de déterminer précisément les niveaux dans lesquels ces entités doivent être définis.

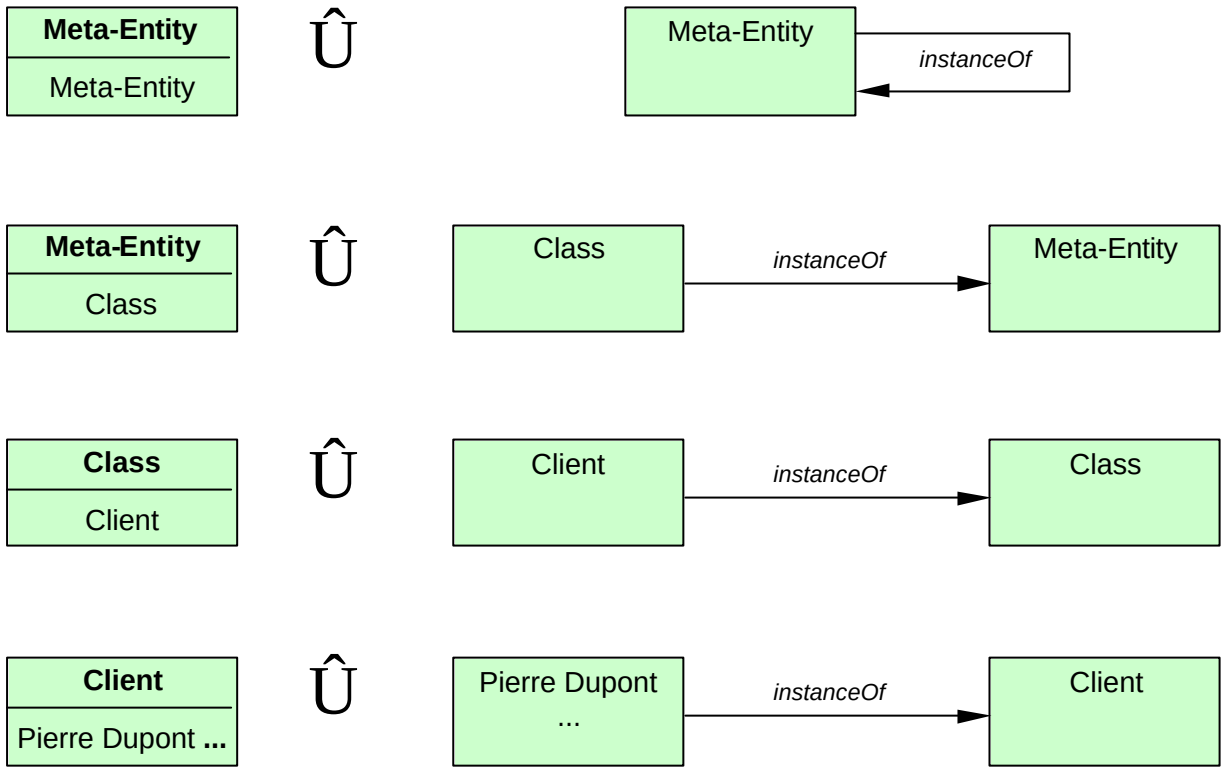


Figure 88 - Relations d'instanciation matérialisant la séparation des niveaux de modélisation.

La première relation *instanceOf* indique que l'entité Meta-Entity appartient au niveau M_3 car sa méta-entité est elle-même et appartient par conséquent également à M_3.

La seconde relation *instanceOf* indique que l'entité Class appartient au niveau M_2 car sa méta-entité Meta-Entity appartient à M_3.

La troisième relation *instanceOf* indique que l'entité Client appartient au niveau M_1 car sa méta-entité Class appartient à M_2.

Tandis que la dernière relation *instanceOf* indique que l'entité « Pierre Dupont » appartient au niveau M_0 car sa méta-entité Client appartient à M_1.

A première vue, cette relation *instanceOf* et son utilisation semble cohérente avec les axiomes que l'on à défini précédemment. Mais intéressons nous maintenant plus précisément au sens de cette relation.

6.2 Le sens contextuel de la relation d'instanciation.

Nous avons dit précédemment que la relation d'instanciation lie les entités d'un niveau aux entités du niveau supérieur. Cette relation doit être définie une et une seule fois dans notre contexte de méta-modélisation et nous dirons par conséquent qu'elle doit être **globale**. Si cette relation est définie dans deux endroits différents, alors rien ne garantie plus que nous sommes effectivement en présence d'une seule et unique relation.

Prenons maintenant la dernière des relation présentées sur la Figure 88. Cette relation est entre une entité "Pierre Dupont" de type Client et une entité Client de type Class. La relation *instanceOf* qui lie "Pierre Dupont" à sa classe Client ne peut pas être considérée comme identique à notre relation globale d'instanciation parce qu'elle ne définie qu'un type **local** dans une contexte de modélisation à objets.

Dans ce cas précis, cette relation *instanceOf* est une relation définie entre un objet et sa classe dans un modèle à objets. Une telle relation n'existe et n'a de sens que dans un modèle à objets et la confusion vient du fait que cette relation **locale** porte le même nom et joue le même rôle (un rôle de typage) que la relation **globale** *instanceOf* qui permet d'effectuer la séparation des niveaux de modélisation.

En fait, dans un contexte de modélisation global, l'entité "Pierre Dupont" dispose d'un lien *instanceOf* vers la méta-entité **Object** définie dans un méta-modèle correspondant à un paradigme objet. Tandis que la relation locale *instanceOf* entre l'entité "Pierre Dupont" et sa classe va être renommée $instanceOf_2$ et est quant à elle contextuelle au modèle objet. Une telle relation est alors définie entre la méta-entité **Object** et la méta-entité **Class** du méta-modèle correspondant à un paradigme objet. La définition de cette relation est présentée sur la Figure 89.

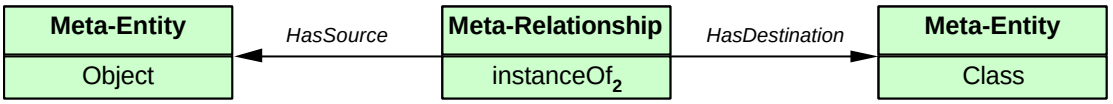


Figure 89 - Relation d'instanciation définie entre la méta-entité Object et la méta-entité Class dans un méta-modèle représentant un paradigme à objets.

Cette relation est donc renommée *instanceOf₂* afin de la différencier de la relation globale d'instanciation utilisée sur la Figure 88. Par conséquent, le quatrième et dernier schéma présenté sur cette Figure 88 sera remplacé par celui présenté par la Figure 90 .

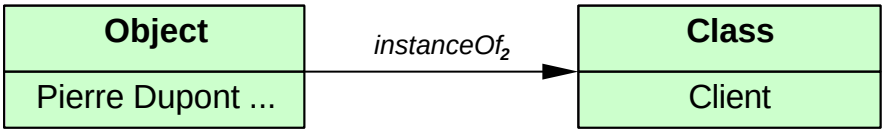


Figure 90 - relation *instanceOf₂* utilisée dans le niveau modèles (M_1).

Par conséquent, la définition explicite de cette relation d'instanciation globale permettant de séparer les différents niveaux de modélisation, nous conduit à modifier la Figure 87 de la façon suivante (Figure 91 dans laquelle cette relation globale d'instanciation est présentée) :

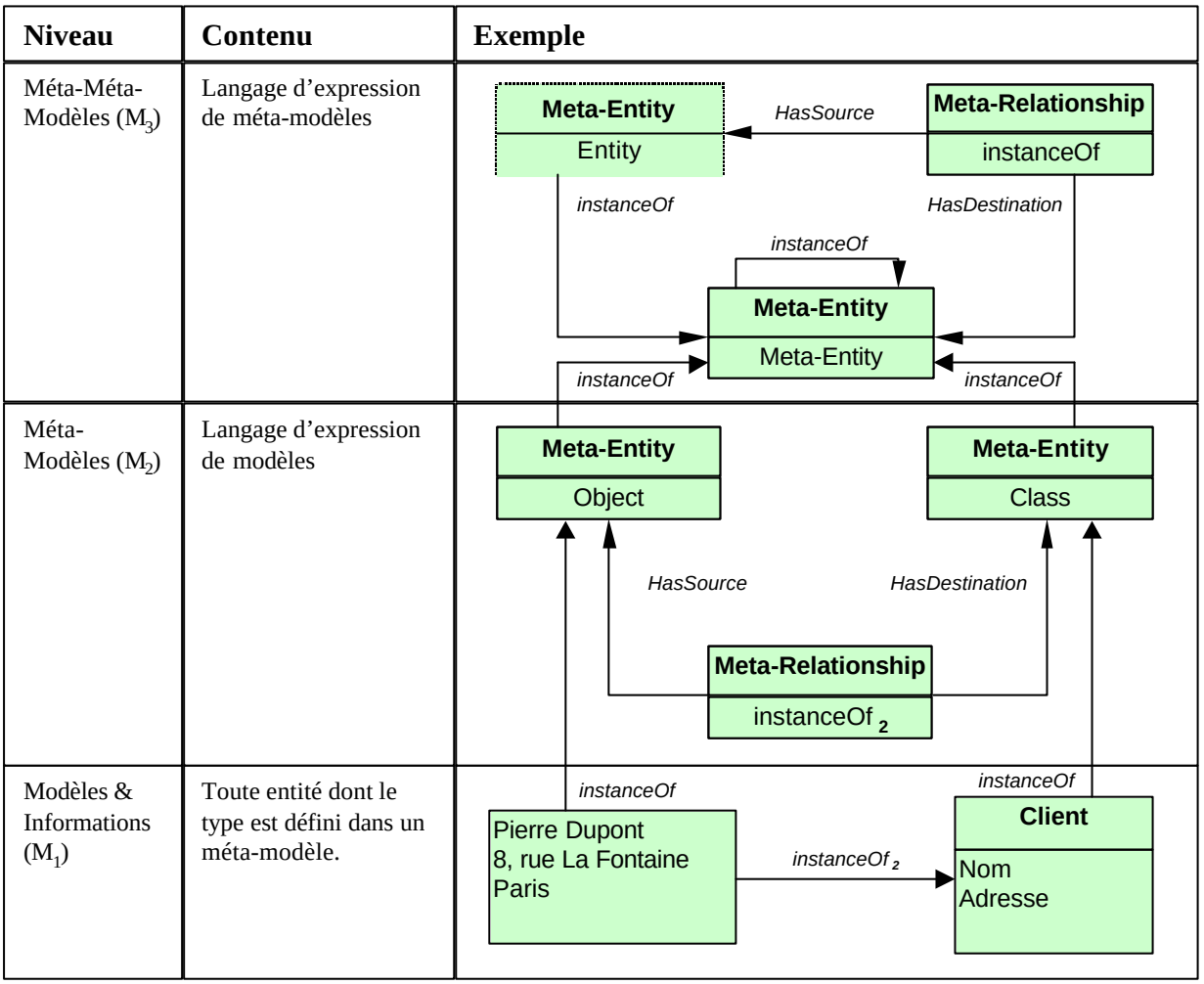


Figure 91 - Une architecture à trois niveaux basée sur une relation d'instanciation définie précisément et explicitement.

En tentant de définir précisément la relation d'instanciation globale permettant d'effectuer la séparation entre les différents niveaux de modélisation, on s'aperçoit que deux relations distinctes sont employées lorsque l'on représente une architecture à quatre niveaux. L'une de ces deux relation est bien une relation globale d'instanciation tandis que l'autre est locale et contextuelle au méta-modèle choisi pour représenter le niveau M_2 (en l'occurrence un méta-modèle représentant un paradigme à objets).

Supposons que l'on ait choisi de représenter un méta-modèle de workflow dans ce niveau M_2 pour "définir" cette architecture à quatre niveaux plutôt qu'un méta-modèle (plus courant peut-être) représentant un paradigme à objets. Nous aurions alors pu représenter un modèle de workflow au niveau M_1, mais qu'aurions nous mis au niveau M_0 ?

Nous voulions mettre en avant dans ce chapitre le fait que si l'on ne définit pas précisément les relations que l'on utilise, alors on peut leur faire dire n'importe quoi. De plus, deux relations semblant jouer le même rôle et portant le même nom dans deux contextes différents peuvent alors être injustement considérées comme identiques. Seulement de plus en plus de choses dans le domaine de la méta-modélisation vont s'appuyer sur de telles architectures et il est préférable que ces fondations soit suffisamment solides pour supporter le poids de tous les méta-modèles qui viendront s'y ajouter.

Nous avons donc montré que le nombre de niveaux est basé sur une relation qui n'est pas clairement identifiée. Ce qui explique pourquoi certains sont amenés à considérer que dans une telle architecture il existe trois niveaux, alors que d'autres en "identifierons" quatre ou même un nombre infini.

6.3 Apport d'une relation d'instanciation explicite.

Le fait d'avoir introduit une relation explicite d'instanciation nous permet de proposer une définition explicite de la notion de niveaux de modélisation. La Figure 92 s'appuie sur cette relation *instanceOf* ainsi que sur les relations *basedOn* et *definedIn* qui ont été introduites précédemment dans le chapitre 5. Cette figure définie trois modèles liés entre eux via la relation *basedOn* et chacun de ces modèle contient une entité rattachée à son modèle via la relation *definedIn*.

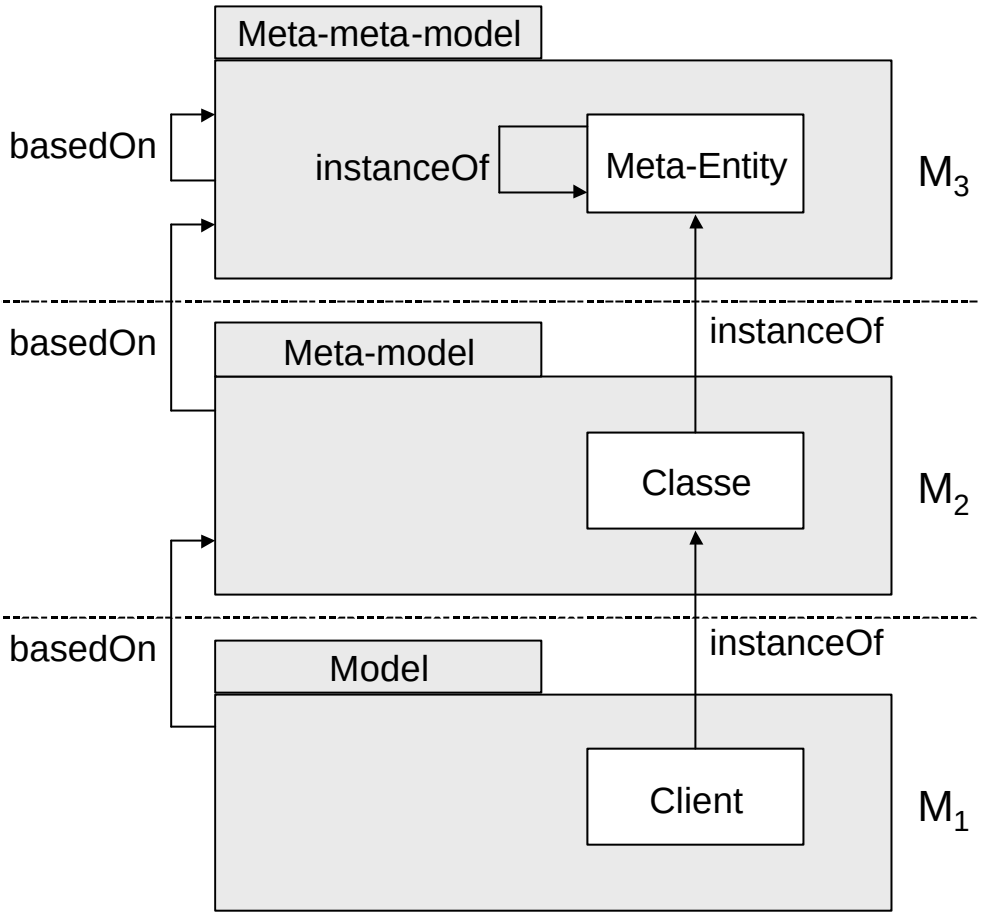


Figure 92 - Une architecture de méta-modélisation à trois niveaux.

Sur cette figure, nous n'avons donc que trois niveaux. On considère que le niveau M_3 est réflexif. Par conséquent la formule logique suivante nous permet d'identifier les modèle constituant ce niveau :

$$\forall m : Model \ \ basedOn(m,m) \Rightarrow belongsToM_3 (m)$$

Cette formule consiste à dire que tout modèle réflexif appartient au niveau M_3. Maintenant les entités de ce niveau peuvent être identifiées simplement car ce sont toutes les entités qui appartiennent à des modèles du niveau M_3. Nous avons donc la formule logique suivante permettant d'identifier les entités de ce niveau M_3:

$$\forall e : Entity \ \ definedIn(e,m) \wedge belongsToM_3 (m) \ \Rightarrow belongsToM_3 (e)$$

Dans le niveau M_2, nous allons trouver tous les modèle définis par un modèle du niveau M_3. De plus, les modèles du niveau M_2 ne peuvent pas être réflexifs (sinon ils appartiendraient au niveau M_3). Par conséquent, la formule logique suivante permet d'identifier les modèles du niveau M_2:

$$\forall m : Model \ basedOn(m,mm) \wedge belongsToM_3(mm) \wedge (m \neq mm) \Rightarrow belongsToM_2(m)$$

Cette formule consiste à dire qu'un modèle dont le méta-modèle appartient au niveau M_3 et qui n'est par réflexif appartient au niveau M_2. Maintenant les entités de ce niveau peuvent être identifiées simplement car ce sont toutes les entités qui appartiennent à des modèles du niveau M_2. Nous avons donc la formule logique suivante permettant d'identifier les entités de ce niveau M_2:

$$\forall e : Entity \ definedIn(e,m) \wedge belongsToM_2(m) \Rightarrow belongsToM_2(e)$$

La même démarche est appliquée pour le niveau M_1. Ainsi, dans ce niveau, nous allons trouver tous les modèle définis par un modèle du niveau M_2 (il n'est pas nécessaire ici de s'assurer que le modèle n'est pas réflexif car on impose que sont méta-modèle soit du niveau M_2 et la contrainte de non réflexivité est déjà vérifiée sur celui-ci). Par conséquent, la formule logique suivante permet d'identifier les modèles du niveau M_1:

$$\forall m : Model \ basedOn(m,mm) \wedge belongsToM_2(mm) \Rightarrow belongsToM_1(m)$$

Cette formule consiste à dire qu'un modèle dont le méta-modèle appartient au niveau M_2 appartient au niveau M_1. Maintenant les entités de ce niveau peuvent être identifiées simplement car ce sont toutes les entités qui appartiennent à des modèles du niveau M_1. Nous avons donc la formule logique suivante permettant d'identifier les entités de ce niveau M_1:

$$\forall e : Entity \ definedIn(e,m) \wedge belongsToM_1(m) \Rightarrow belongsToM_1(e)$$

Nous avons donc ainsi défini précisément ce que nous entendions par niveau de modélisation. Cette architecture étant issue de notre interprétation, elle n'est pas nécessairement identique à ce que d'autres pourraient définir, mais elle a au moins le mérite d'être explicite.

7 Méta-modélisation et transformation de modèles.

Dans ce chapitre, nous montrons comment les techniques de méta-modélisation peuvent être utilisées pour définir des règles de transformations de modèles. En effet, un méta-modèle décrivant la sémantique d'un modèle, les transformations entre deux modèles peuvent être décrites dans les termes définis dans chacun de leurs méta-modèles. Ce travail s'est inspiré des travaux effectués dans le domaine de la réécriture de graphes utilisée notamment dans PROGRES (Programmed Graph Replacement Systems) et de la notion d'hyper-généricité.

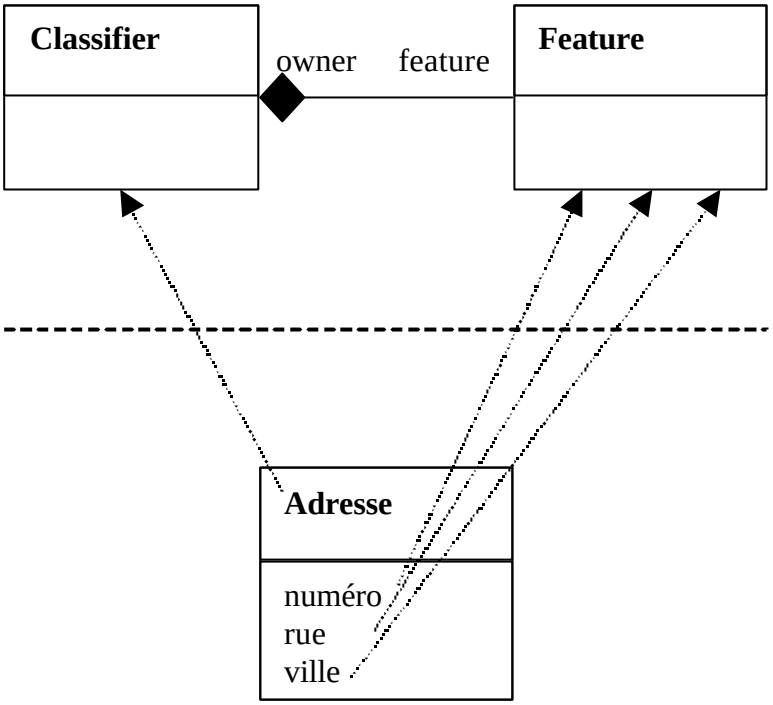
Nous allons donc montrer ici que lorsque deux modèles sont exprimés dans deux langages définis par des méta-modèles exprimés eux-mêmes dans un langage identique (i.e. un méta-méta-modèle identique), il est possible de définir précisément des règles de transformation de l'un des modèles vers l'autre. L'exemple présenté dans ce chapitre consiste à décrire précisément des règles transformations d'un modèle à objets (utilisant la sémantique d'UML) vers un modèle relationnel. Notre but n'est pas de montrer que cette transformation particulière est évidente parce ce n'est pas le cas et qu'il existe n façons de passer d'un modèle à objet vers un modèle relationnel. En effet, certains désirent obtenir autant de tables relationnelles que de classes dans le modèle à objets tandis que d'autres ne voudrons des tables relationnelles que pour les classes concrètes du modèle. Nous voulons simplement montrer que la formalisation des méta-modèles à objets et relationnels nous permettra de formaliser également de telles règles de transformations. Le formalisme des sNets sera utilisé pour représenter ces deux modèles .

L'outil d'application des règles de transformations a été développé dans le cadre de cette thèse.

7.1 La représentation des modèles et des méta-modèles.

La Figure 93 présente un exemple de modèle objet et son méta-modèle.

Partie d'un méta-modèle à objets (UML)



Partie d'un modèle à objets

Figure 93 - Une partie d'un modèle à objets et son méta-modèle.

Sur cette Figure 93, nous avons employé le formalisme UML pour décrire à la fois le modèle et le méta-modèle. Nous avons fait apparaître les liens entre les entités du modèle et les entités du méta-modèle via des flèches pointillées représentant les relations *meta* (également appelée *instanceOf*). En modélisation, ainsi qu'en méta-modélisation, toute entité dispose d'un tel lien vers son type. Lorsque ce lien n'est pas explicite (comme c'est le cas en UML par exemple), il est existe tout de même implicitement.

De la même façon, la notation UML a été utilisée pour représenter une partie d'un méta-modèle relationnel sur la Figure 94. Ce méta-modèle définit les concepts de tables et de colonnes ainsi que la relation de composition qui existe entre ces concepts (une table relationnelle est constituée d'un ensemble de colonnes).

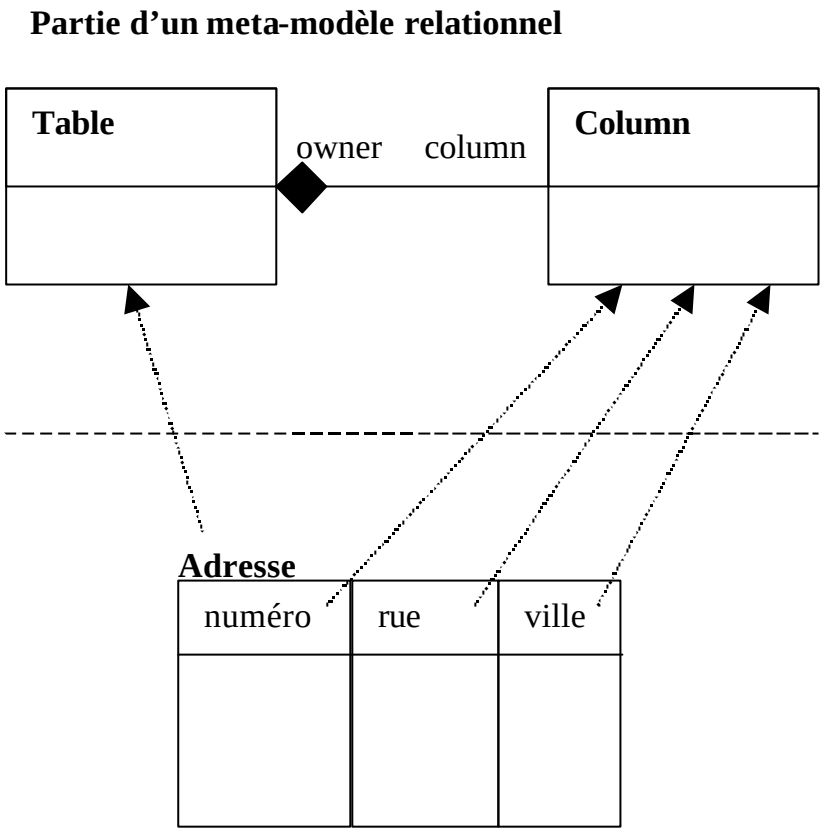


Figure 94 - Une partie d'un modèle relationnel et son méta-modèle.

Nous avons également représenté ici par des flèches pointillées le lien entre les entités du modèle relationnel et leurs types représentés par des entités du méta-modèle relationnel.

Afin de pouvoir modéliser les transformations, nous allons utiliser un formalisme commun pour représenter ces deux modèles et leurs méta-modèles. Nous utilisons le formalisme des sNets car il est défini de façon formelle, tout y est explicite, et il permet de"voir" toutes ces informations comme un simple réseau sémantique de sorte que les techniques de réecritures de graphes peuvent alors être utilisées.

7.1.1 Utilisation du formalisme des sNets.

Dans ce formalisme, tout est entité. De plus, une entité sNets dispose d'au moins trois liens vers trois autres entités sNets :

- un lien appelé *name* vers l'entité représentant son nom,
- un lien appelé *meta* vers l'entité décrivant son type et
- un lien appelé *partOf* vers l'entité représentant son univers (un univers sNets correspond à un modèle)

La classe Adresse représentée Figure 93 peut alors être représentée de la façon suivante dans le formalisme des sNets :

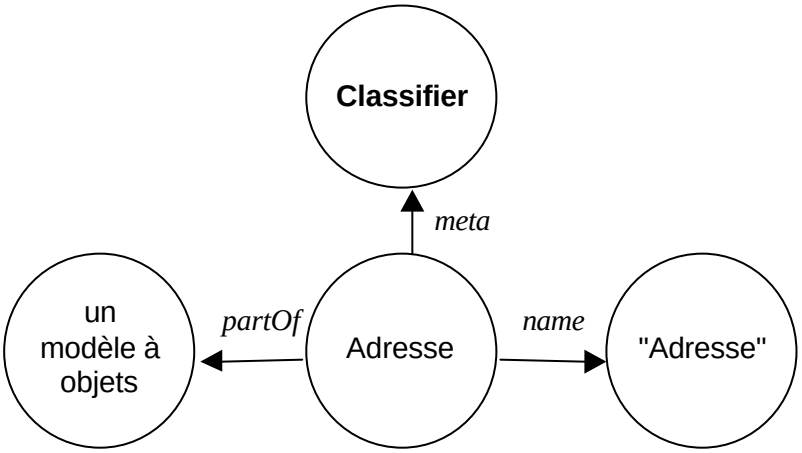


Figure 95 - L'entité Adresse représentée avec le formalisme des sNets.

Sur cette Figure 95, nous retrouvons les quatre entités suivantes :

- l'entité Adresse représentant la classe Adresse dans le formalisme des sNets,
- l'entité "Adresse" représentant son nom,
- l'entité **Classifier** représentant son type et
- l'entité "un modèle à objets" représentant son univers.

La notation graphique simplifiée suivante peut également être utilisée pour représenter l'entité "Adresse" de type **Classifier** définie dans l'univers "un modèle à objets" :

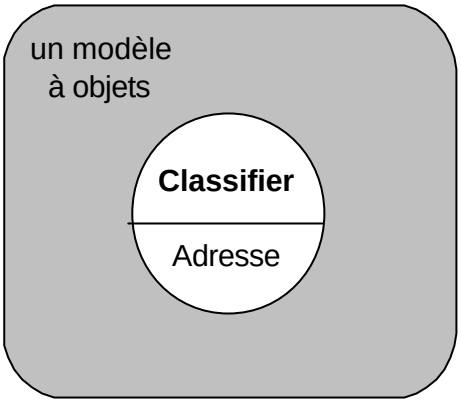


Figure 96 - L'entité Adresse représentée avec le formalisme des sNets (notation simplifiée).

Dans le formalisme des sNets, les entités représentant les types sont des entités de type EntityType tandis que les entités représentant les types de relation sont des entités de type RelationType. La figure suivante représente alors à la fois le modèle à objets et son méta-modèle :

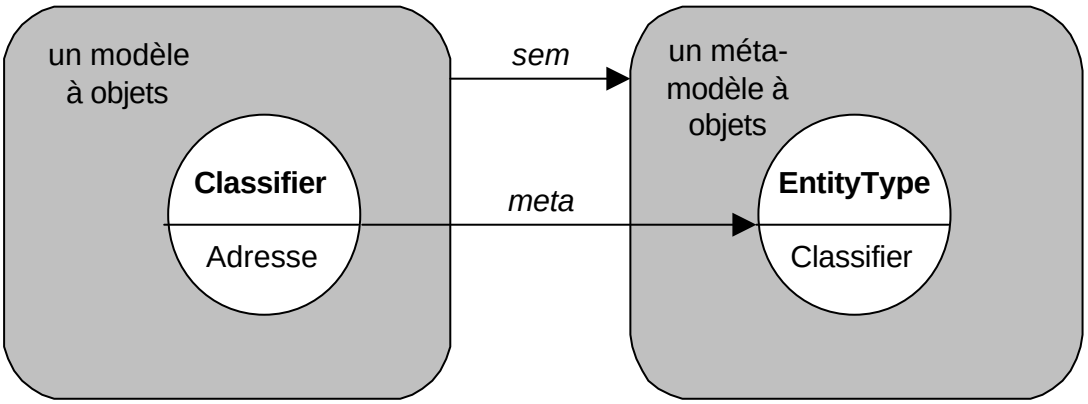


Figure 97 - Le modèle à objets et son méta-modèle (formalisme sNets).

Les entités du modèle sont liées aux entités définissant leur type dans le méta-modèle via les liens *meta* tandis que le modèle est lié à son méta-modèle via le lien *sem*.

Une représentation plus complète au formalisme sNets du méta-modèle à objets présenté Figure 93 peut alors être la suivante :

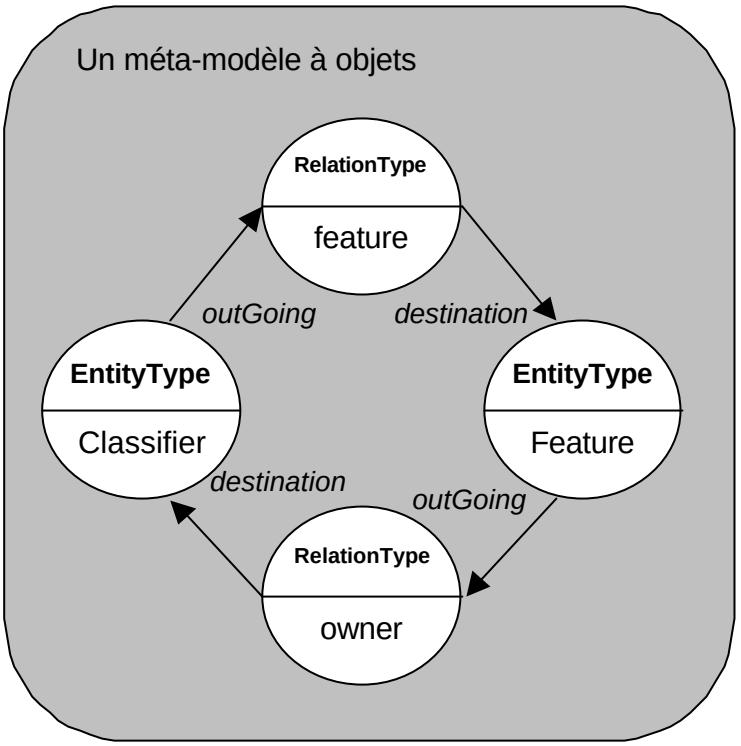


Figure 98 - Une partie d'un méta-modèle à objets au format sNets.

De cette figure, nous pouvons déduire que :

- Un modèle à objets contient des entités de type **Classifier**,
- Un modèle à objets contient des entités de type **Feature**,
- Une entité de type **Classifier** peut disposer d'un lien *feature* vers des entité de type **Feature**,
- Une entité de type **Feature** est rattachée à une ou plusieurs entités de type **Classifier** via un lien *owner*.

Le modèle à objets et son méta-modèle peuvent alors être représentés de la façon suivante en utilisant le formalisme des sNets :

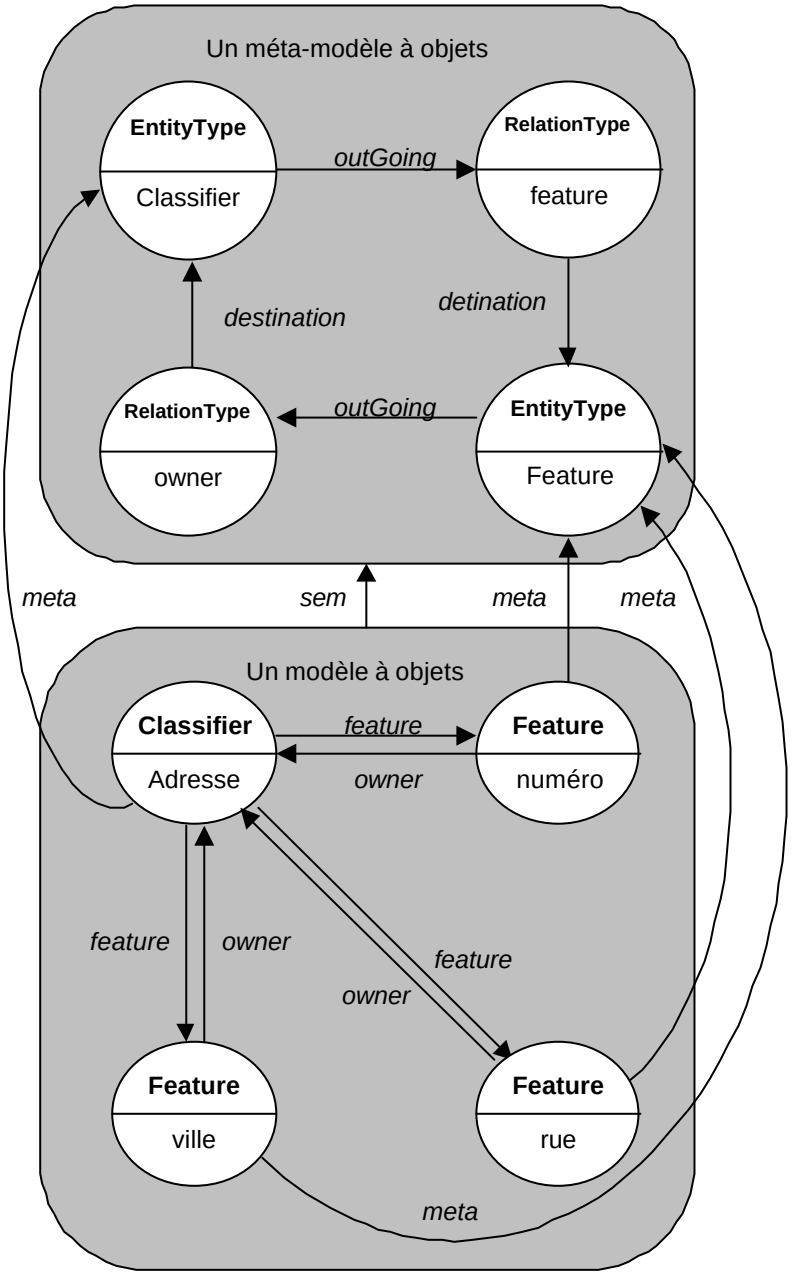


Figure 99 - Une partie d'un modèle à objets et son méta-modèle (exprimé en utilisant le formalisme des sNets).

Dans le formalisme des sNets, tous les méta-modèles sont représentés par des univers sémantiques qui ont eux-mêmes pour unives sémantique l'univers Semantic décrit dans le

chapitre 4.2. Ainsi, de la même façon que nous avons représenté le modèle à objets et son méta-modèle en utilisant le formalisme des sNets, il nous est possible de représenter un modèle relationnel ainsi que son méta-modèle dans ce même formalisme. Tous ces éléments sont alors réunis dans un seul réseau sémantique sur lequel nous pourrons appliquer nos transformations.

Comme le décrit la figure suivante, le formalisme des sNets permet la représentation, dans un même réseau, d'un ensemble de modèles et d'un ensemble de méta-modèles :

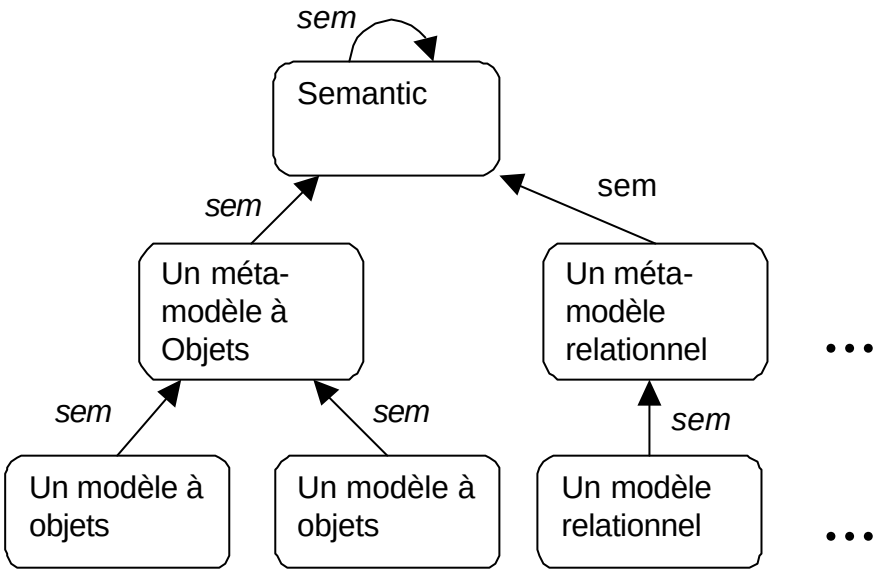


Figure 100 - Le formalisme des sNets représentant, dans le même réseau, un ensemble de modèles et de méta-modèles.

Ainsi, tout ensemble de modèles et méta-modèles peut être représenté sous forme de graphe orienté composé de nœuds (les entités sNets) et de liens nommés (les relations sNets) de sorte que la transformation de modèles puisse être vue comme un simple processus de réécriture de graphes.

7.2 Les règles de transformation.

Avant de décrire précisément les règles de transformation, nous allons présenter ce qui peut être assimilé à une transformation de modèles.

7.2.1 Qu'est-ce qu'une transformation de modèles ?

Une transformation est décrite en utilisant la sémantique des méta-modèles. En effet, qu'est-ce qu'une transformation de modèles ? Une expression de transformation de modèle se présente généralement sous la forme d'une phrase informelle dont voici quelques exemples :

- A chaque classe d'un modèle à objets correspondra une table dans le modèle relationnel,
- a chaque attribut de classe dans un modèle à objets correspondra une colonne dans une table du modèle relationnel.

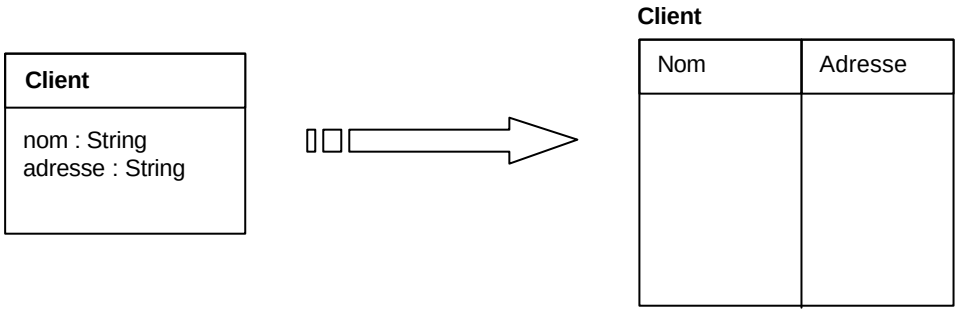


Figure 101 - Transformation d'un modèle à objets en un modèle relationnel.

L'application d'un design pattern (patron de conception) peut également être vue comme une transformation de modèle :

- Tout objet observable doit disposer d'une méthode "register(obj)" tandis que tout objet observeur doit disposer d'une méthode "update()".

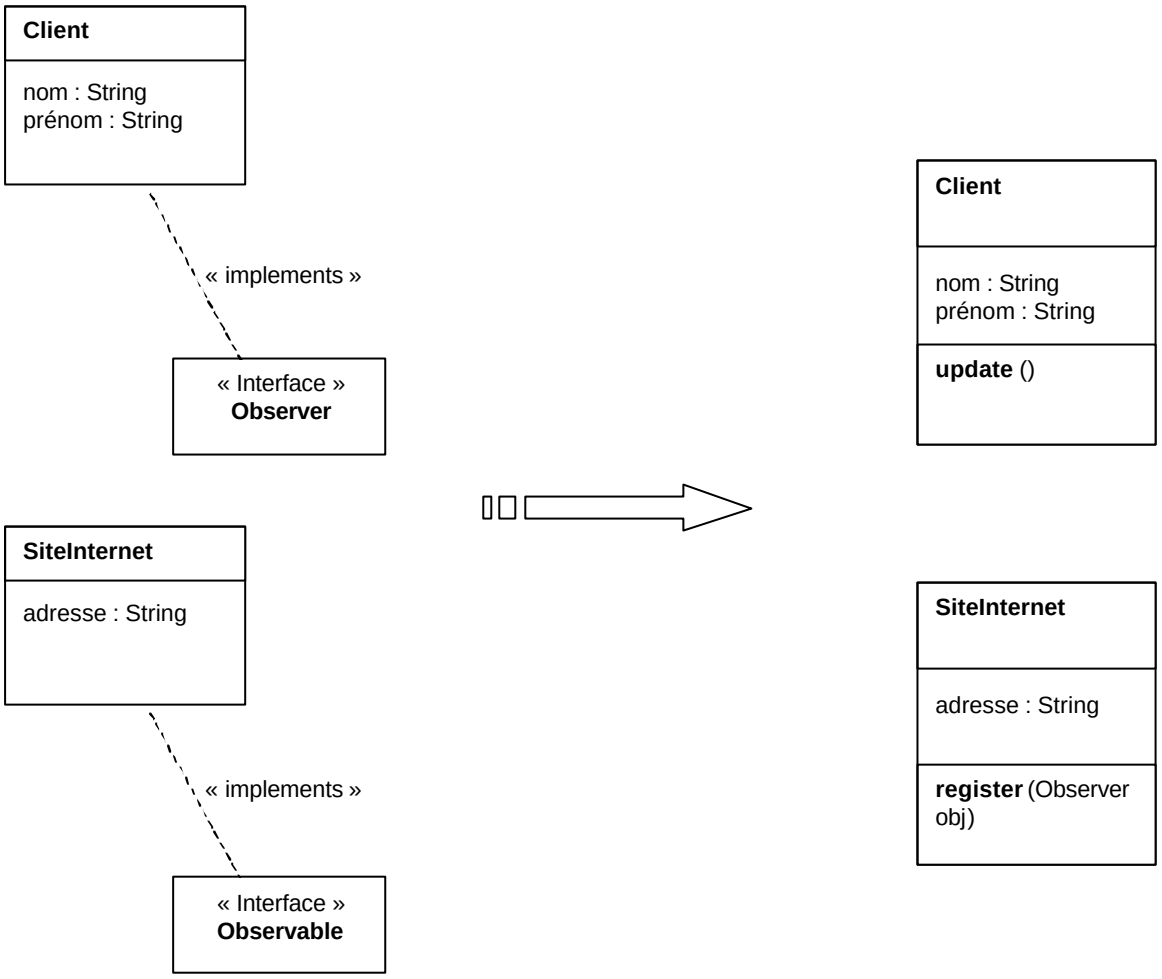


Figure 102 - Application d'un design pattern "Observeur/Observable".

Dans ces deux cas, la transformation peut s'exprimer en utilisant les termes définis dans les méta-modèles des modèles à transformer :

- Une **classe** devient une **table**,
- Un **attribut** *d'une* **classe** devient une **colonne** *d'une* **table**,
- Une **classe** *implémentant l'interface* "Observer" dispose *d'une* **méthode** "update",
- Une **classe** *implémentant l'interface* "Observable" dispose *d'une* **méthode** "register", etc…

Dans ces différentes phrases décrivant des transformations, nous avons fait apparaître en gras les concepts définis dans les méta-modèles utilisés et en italique nous avons fait apparaître les relations définies entre ces concepts.

Les transformations peuvent ainsi être définies sous formes de contraintes exprimées sur entités et les relations d'un modèle et entrainant la création d'entités et de relations dans le modèle cible :

- Une entité dont le type est un concept du méta-modèle de la source va pouvoir entrainer la création d'une entité dont le type est un concept du méta-modèle de la cible,
- Une relation dans le modèle source va pouvoir entrainer la création d'une relation dans le modèle cible,

Et plus généralement, un ensemble d'entités typées et de relations entre ces entités dans un modèle source va entrainer la création d'un ensemble de nouvelles entités typées et de relations dans le modèle de la cible.

Ainsi, dans les trois transformations informelles présentées précédemment, les noms en gras vont donner lieu à des conditions sur les types des entités tandis que les noms en italique vont donner lieu à des contraintes sur les relations entre ces entités. Ces transformations peuvent alors être formalisées de la façon suivante :

- ∀ x Type(x,**Classifier**) Contains(x, modèleSource) ⇒ ∃ y Type(y, **Table**) DefinedIn(y,modèleCible)

- ∀ x Type(x,**Attribut**) Contains(x, modèleSource) ⇒ ∃ y Type(y, **Column**) DefinedIn(y,modèleCible)

- ∀ x Type(x,**Classifier**) Link(*implements*, x, y) Link(*name*, y, "Observer") Contains(x, modèleSource) ⇒ ∃ z Type(z, **Method**) Link(*features*, x, z) Link(*name* z, "update")

- ∀ x Type(x,**Classifier**) Link(*implements*, x, y) Link(*name*, y, "Observable") Contains(x, modèleSource) ⇒ ∃ z Type(z, **Method**) Link(*features*, x, z) Link(*name* z, "register")

Toutes ces transformations peuvent donc être définies en utilisant la sémantique des méta-modèles de modèles source et cible de la transformation. La Figure 103 réunit les éléments permettant la transformation d'un modèle à objets vers un modèle relationnel.

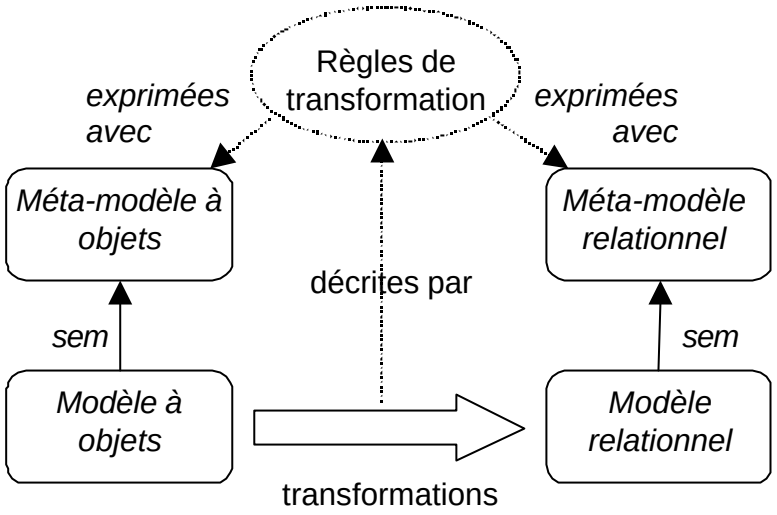


Figure 103 - Mécanisme de transformations exprimées dans les termes des méta-modèles.

Nous avons donc développé un outil de transformation de modèle prenant en entrée deux modèles au format sNets et un ensemble de règles définies dans un fichier texte. La transformation consiste à appliquer chaque règle définie dans ce fichier sur les deux modèles en entrée jusqu'à ce qu'aucune règle ne soit plus applicable. Nous considérons alors que la transformation est terminée. Chaque règle est composée d'un ensemble de conditions et d'un ensemble de conclusions. Une condition peut être une condition sur le type d'une variable (représentant une entité) ou une condition sur un lien entre deux variables (représentant deux entités). Une conclusion peut être la création d'une entité dans le modèle cible ou la création d'une relation entre deux entités du modèle cible.

Nous avons défini la grammaire suivante (au format BNF) pour l'expression de ces règles :

<RulesDesc> ::= <Header> **:** <Rules>
<Header> ::= <SourceSem> **->** <TargetSem>
<Rules> ::= <Rule> *
<Rule> ::= <Conditions> **->** <Conclusions> **;**
<Conditions> ::= <Condition> *
<Conclusions> ::= <Conclusion> *

L'en-tête permet d'indiquer la sémantique du modèle source et la sémantique du modèle cible de manière à pouvoir effectuer de vérifications de type lors du processus de transformation.

7.2.2 La partie "conditions" d'une règle de transformation.

La partie "conditions" d'une règle de transformation est constituée d'un ensemble de conditions simples. Une condition peut être une contrainte sur le type d'une variable ou sur l'existence d'un lien entre deux variables.

7.2.2.1 Condition sur le type d'une variable.

Une telle condition se présente sous la forme suivante dans notre format de représentation des transformations :

<TypeName> **(** v **)**

Cette condition sera vérifiée si et seulement si la variable v est liée à une entité sNets appartenant à l'univers représentant le modèle source ou celui représentant le modèle cible et dont le type est une entité de type **EntityType** nommée <TypeName>. Cette condition peut être formalisée à l'aide des prédicats définis dans le chapitre 4.1 de la façon suivante :

$$\forall v \ \text{Node}(v)\ \text{Type}(v, vt)\ \text{name}(vt, \langle\text{TypeName}\rangle) \wedge (\text{DefinedIn}(v, sm) \vee \text{DefinedIn}(v, tm))$$

Dans cette expression :

- v désigne une variable qui sera liée à une entité sNets au cours du processus de transformation,
- vt désigne également une variable qui sera liée à une entité sNets de type **EntityType** au cours du processus de transformation,
- <TypeName> représente une constante chaine de caractères,
- sm est une variable liée à l'entité sNets représentant le modèle source avant le démarrage du processus de transformation et

- tm est une variable liée à l'entité sNets représentant le modèle cible avant le démarrage du processus de transformation.

7.2.2.2 Condition sur le lien entre deux variables.

Une telle condition se présente sous la forme suivante dans notre format de représentation des transformations :

<linkName> **(** sv**,** tv **)**

Cette condition sera vérifiée si et seulement si les variables sv et tv sont liées à deux entité sNets appartenant à l'univers représentant le modèle source ou celui représentant le modèle cible et pour lesquelles il existe un lien sNets appelé <LinkName> partant de sv et aboutissant à tv. Cette condition peut être formalisée à l'aide des prédicats définis dans le chapitre 4.1 de la façon suivante :

$$\forall\ sv,tv\ \ Node(sv)\ Node(tv)\ Link(lnk, sv, tv)\ Link(\textit{name}, lnk, <linkName>) \wedge$$
$$(\ DefinedIn(\ sv, sm) \vee DefinedIn(\ sv, tm)\ ) \wedge$$
$$(\ DefinedIn(\ tv, sm) \vee DefinedIn(\ tv, tm)\ )$$

Dans cette expression :

- sv et tv désignent des variables qui seront liées aux entités sNets au cours du processus de transformation,
- lnk désigne également une variable qui sera liée à une entité sNets de type **RelationType** au cours du processus de transformation,
- <linkName> représente une constante chaine de caractères,
- sm est une variable liée à l'entité sNets représentant le modèle source avant le démarrage du processus de transformation et
- tm est une variable liée à l'entité sNets représentant le modèle cible avant le démarrage du processus de transformation.

tv peut également représenter une constante auquel cas la variable tv est directement liée à cette constante et l'expression formelle de cette condition est contractée de la façon suivante :

$$\forall\ sv\ Node(sv)\ Link(lnk, sv, tv)\ Link(\textit{name}, lnk, <linkName>) \wedge (\ DefinedIn(\ sv, sm) \vee DefinedIn(\ sv, tm)\ )$$

7.2.2.3 Quelques exemples de conditions de règles de transformation.

Voici quelques exemple de conditions de règles de transformation ainsi que l'expression formelle qui leur correspond :

- Classifier(class) permettant de "selectionner" toutes les entités de type **Classifier** et dont le sens formel est le suivant :

  $\forall$ class Node(class) Type(class, vt) name(vt, "Classifier") $\wedge$ (DefinedIn(class, sm) $\vee$ DefinedIn(class, tm))

- Table(t) permettant de "selectionner" toutes les entités de type **Table** et dont le sens formel est le suivant :

  $\forall$ t Node(t) Type(t, vt) name(vt, "Table") $\wedge$ (DefinedIn(t, sm) $\vee$ DefinedIn(t, tm))

- owner(attribut, classe) permettant de "selectionner" tous les attributs associés à une classe (et toutes ces classes) et dont le sens formel est le suivant :

  $\forall$ attribut, classe Node(attribut) Node(classe) Link(lnk, attribut, classe) Link(*name*, lnk, "owner") $\wedge$
  (DefinedIn(attribut, sm) $\vee$ DefinedIn(attribut, tm)) $\wedge$
  (DefinedIn(classe, sm) $\vee$ DefinedIn(classe, tm))

- name(c, "Adresse") permettant de "selectionner" toutes les entités nommées "Adresse" et dont le sens formel est le suivant :

  $\forall$ c Node(c) Link(lnk, c, "Adresse") Link(*name*, lnk, "name") $\wedge$
  (DefinedIn(c, sm) $\vee$ DefinedIn(c, tm))

7.2.3 La partie "conclusions" d'une règle de transformation.

La partie "conclusions" d'une règle est constituée d'un ensemble de conclusions simples. Une conclusion dans une règle de transformation va consister à créer une nouvelle entité sNet dans le modèle cible et la lier à une variable, ou bien à créer une nouveau lien entre deux entité désignées par des variables liées.

7.2.3.1 Conclusion entraînant la création d'une entité.

Une telle conclusion se présente sous la forme suivante dans notre format de représentation des transformations :

<TypeName> **(** v **)**

Cette conclusion indique qu'il existe une entité dans le modèle cible dont le type est une entité de type **EntityType** nommée <TypeName>. Il faut par conséquent créer cette entité et la lier à la variable v. Cette conclusion peut être formalisée à l'aide des prédicats définis dans le chapitre 4.1 de la façon suivante :

$$\exists\, v \;\; Node(v)\; Type(v, vt)\; name(vt, \langle TypeName \rangle) \wedge DefinedIn(v, tm)$$

Dans cette expression :

- v désigne une variable qui sera liée à une entité sNets **crée** au cours du processus de transformation,
- vt désigne également une variable qui sera liée à une entité sNets de type **EntityType** au cours du processus de transformation,
- <TypeName> représente une constante chaine de caractères et
- tm est une variable liée à l'entité sNets représentant le modèle cible avant le démarrage du processus de transformation.

7.2.3.2 Conclusion entraînant la création d'un lien.

Une telle conclusion se présente sous la forme suivante dans notre format de représentation des transformations :

<linkName> **(** sv**,** tv **)**

Cette conclusion indique qu'il existe un lien entre l'entité liée à la variable sv et l'entité liée à la variable tv. Cette deux entités peuvent être définie indifféremment dans le modèle source ou dans le modèle cible. Il faut par conséquent créer ce lien entre ces entités. Cette conclusion peut être formalisée à l'aide des prédicats définis dans le chapitre 4.1 de la façon suivante :

$$\forall\ sv,tv\ \ Node(sv)\ Node(tv)\ \exists\ lnk\ \ Link(lnk, sv, tv)\ Link(\textit{name}, lnk, \text{<linkName>}) \wedge$$
$$(\ DefinedIn(\ sv, sm) \vee DefinedIn(\ sv, tm)\ ) \wedge$$
$$(\ DefinedIn(\ tv, sm) \vee DefinedIn(\ tv, tm)\ )$$

Dans cette expression :

- sv et tv désignent des variables qui seront liées aux entités sNets au cours du processus de transformation,
- lnk désigne également une variable qui sera liée à une entité sNets de type **RelationType** au cours du processus de transformation,
- sm est une variable liée à l'entité sNets représentant le modèle source avant le démarrage du processus de transformation,
- tm est une variable liée à l'entité sNets représentant le modèle cible avant le démarrage du processus de transformation. et
- <linkName> représente une constante chaine de caractères.

tv peut également représenter une constante auquel cas la variable tv est directement liée à cette constante et l'expression formelle de cette condition est contractée de la façon suivante :

$$\forall\ sv\ Node(sv)\ \exists\ lnk\ Link(lnk, sv, tv)\ Link(\textit{name}, lnk, \text{<linkName>})$$

7.2.3.3 Quelques exemples de conclusions de règles de transformation.

Voici quelques exemple de conclusions de règles de transformation ainsi que l'expression formelle qui leur correspond :

- Classifier(class) permettant de créer une entité de type **Classifier** dans le modèle cible et de la lier à la variable class. Le sens formel de cette expression est le suivant :

  ∃ class Node(class) Type(class, vt) name(vt, "Classifier") ∧ DefinedIn(class, tm)

- Table(t) permettant de crée une entité de type **Table** dans le modèle cible et de la lier à la variable t. Le sens formel de cette expression est le suivant :

  ∃ t Node(t) Type(t, vt) name(vt, "Table") ∧ DefinedIn(t, tm)

- owner(attribut, classe) permettant de créer un lien "owner" entre l'entité sNets liée à la variable attribut et l'entité sNets liée à la variable classe. Le sens formel de cette expression est le suivant:

  ∀ attribut, classe Node(attribut) Node(classe) ∃ lnk Link(lnk, attribut, classe)
  Link(*name*, lnk, "owner") ∧
  (DefinedIn(attribut, sm) ∨ DefinedIn(attribut, tm)) ∧
  (DefinedIn(classe, sm) ∨ DefinedIn(classe, tm))

- name(c, "Adresse") permettant de créer un lien "name" entre l'entité sNets liée à la variable c et la constante "Adresse". Le sens formel de cette expression est le suivant :

  ∀ c Node(c) ∃ lnk Link(lnk, c, "Adresse") Link(*name*, lnk, "name") ∧
  (DefinedIn(c, sm) ∨ DefinedIn(c, tm))

7.2.4 Constitution d'une règle de transformation.

Une règle de transformation est composée d'un ensemble de conditions et d'un ensemble de conclusion et se présente sous la forme suivante dans notre format de représentation des transformations:

<Condition> * **->** <Conclusion > * **;**

Une telle règle indique que si toutes les conditions sont vérifiées pour un ensemble donné de variables liées, alors toutes les conclusions doivent également être vérifiées pour ce même ensemble de variables liées. Pour que chacune des conclusions soit vérifiée, les entités et les liens nécessaires sont créés. Cette règle se formalise de la façon suivante :

<conditions> $\Rightarrow$ <conclusions>

Ainsi, si l'on prend par comme exemple la règle suivante :

Classifier(c) -> Table (t);

Elle sera formalisée de la façon suivante :

$$\forall\ c\ \text{Node}(c)\ \text{Type}(c, vt)\ \text{name}(vt, \text{"Classifier"}) \wedge (\text{DefinedIn}(c, sm) \vee \text{DefinedIn}(c, tm)) \Rightarrow (\exists\ t\ \text{Node}(t)\ \text{Type}(t, vt')\ \text{name}(vt', \text{"Table"}) \wedge \text{DefinedIn}(t, tm))$$

Dans cette expression :

- c, t, vt, vt' désignent des variables qui seront liées aux entités sNets au cours du processus de transformation,
- sm est une variable liée à l'entité sNets représentant le modèle source avant le démarrage du processus de transformation et
- tm est une variable liée à l'entité sNets représentant le modèle cible avant le démarrage du processus de transformation.

Cette règle de transformation va ainsi générer dans le modèle cible autant d'entités de type **Table** qu'il y a d'entités de type **Classifier** dans le modèle source.

7.2.5 Le processus de transformation.

Le processus de transformation va consister à appliquer l'ensemble des règles de manière à obtenir le modèle cible. L'application de cet ensemble de règles consiste à appliquer les règles une à une sans se préoccuper de l'ordre et de boucler jusqu'à ce qu'il n'y ait plus de règle applicable.

Dans le format de représentation des règles que nous venons de présenter, le fichier décrivant les transformations dispose d'une en-tête indiquant la sémantique du modèle source et celle du modèle cible de la façon suivante :

<source sem> **->** <target sem> **:**

Ainsi, l'ensemble des règles permettant de transformer un modèle à objets en un modèle relationnel disposera d'une entête similaire à l'entête suivante :

ObjectMetaModel -> RelationMetaModel :

De sorte que le transformateur puisse s'assurer que les entités et les relations créées ont leurs types effectivement définis dans ces méta-modèles. Ainsi, la transformation permettant de créer un table dans un modèle relationnel à partir de chacune des classes d'un modèle à objets pourra s'exprimer de la façon suivante dans un fichier utilisant notre format d'expression des transformations :

```
ObjectMetaModel -> RelationalMetaModel :

Classifier(c) -> Table(t);
```

Cette transformation va donc créer autant de tables dans le modèle relationnel qu'il y à de classes dans le modèle à objets. Maintenant, ces tables ne sont pas nommées explicitement. En effet, rien dans cette transformation n'indique par exemple que la table créée porte le même nom que la classe dont elle est issue. Les entités créées portent alors un nom arbitraire. La création d'une entité de type <X> dont le nom n'est pas précisé donne une entité nommée a<X> par défaut ainsi les tables créées par la transformation précédente s'appellent toutes aTable. Si l'on souhaite que les tables créées portent le même nom que les classes dont elles sont issues, il suffit de modifier la règle de création des tables de la façon suivante :

```
Classifier(c) name(c, n) -> Table(t) name(t, n);
```

7.2.5.1 Les relations temporaires.

Nous avons indiqué que les types des entités et des relations créées durant le processus de transformation doivent être définis dans les méta-modèles indiqués dans l'en-tête des règles. Par conséquent, les modèle obtenus après transformation sont nécessairement conformes au méta-modèle cible.

L'implémentation de notre moteur de transformation nous à montré qu'il pouvait parfois être utile, voire nécessaire, de créer des relations temporaires au cours de la transformation. En effet, supposons que nous ayont une règle pour créer une table correspondant à chaque classe de notre modèle à objets, puis une autre règle pour créer une colonne correspondant à chacun des attributs de ces classes. Il nous faut ensuite une troisième règle permettant de rattacher les colonnes aux tables auxquelles elles doivent être rattachées.

En effet , supposons que nous avons défini les règles suivantes :

```
Classifier(c) name(c,n) -> Table(t) name(t,n);
Feature(f) name(f,n) -> Column(c) name(c,n);
```

Celles-ci créent les tables et leurs colonnes, mais il faut maintenant rattacher ces tables à leurs colonnes. Nous avons alors envie d'écrire une règle du style :

```
Classifier(c) Feature(f) feature(c,f) -> column(t,col);
```

Pour chaque lien *feature* entre un attribut et sa classe nous voulons créer un lien *column* entre la colonne et sa table. Le problème qui se pose alors est que la création du lien ne peut s'effectuer car les variables `t` et `col` ne sont pas liées (ce sont des variables libres). En effet, il nous faut un moyen de dire que `t` est la table qui correspond à `c` et que `col` est la colonne qui correspond à `f`. La solution que nous avons mis en place consiste à autoriser la création d'un lien temporaire à la transformation permettant de lier une classe à sa table et une colonne à son attribut. Pour signaler que ces liens sont temporaires et peuvent par conséquent ne pas être définis au niveau sémantique, nous les identifions en les préfixant du caractère "_".

Les deux règles précédentes sont alors modifiées de la façon suivante afin de lier les tables à leurs classes et les attributs à leurs colonnes avec ces liens temporaires :

```
Classifier(c) name(c,n) ->
        Table(t) name(t,n) _coref(t,c) ;

Feature(f) name(f,n) ->
        Column(c) name(c,n) _coref(c,f);
```

Le lien _coref désigné ici n'a pas de définition dans l'un ou l'autre des méta-modèle, il est juste "créé" au cours de la transformation, puis supprimé dès la fin de celle-ci.

Une règle permettant de lier les colonnes à leurs tables respective peut alors être la suivante :

```
Classifier(c) Table(t) _coref(t,c) Feature(f) Column(col) _coref(col,f) feature(c,f) ->

        column(t,col);
```

En clair, pour toute classe c associée à une table t et pour tout attribut f associé à une colonne col pour lesquels l'attribut c est un attribut de la classe f, nous devons définir un lien *column* entre t et col de sorte que la colonne col doit alors être une colonne de la table t.

Ces liens *_coref* n'ont alors un sens qu'au cours de la transformation. Une fois le processus de transformation terminé, les liens temporaires sont supprimés de sorte que les seuls liens restant sont effectivement définis dans les méta-modèles des modèles transformés.

Ainsi, l'exemple suivant regroupe et enrichi tous les éléments d'une telle transformation :

```
ObjectMetaModel -> RelationalMetaModel :

Classifier(c) name(c,n)  ->

        Table(t) name(t,n) _coref(c,t) Column(id)
        name(id,"id") column(t,id);

Classifier(c) generalization(c,g) supertype(g,sc) ->

        _super(c,sc);
```

```
Classifier(c) Classifier(sc1) Classifier(sc2) _super(c,sc1) _super(sc1,sc2) ->
        _super(c,sc2);
Classifier(c) feature(c,a) ->
        _attribute(c,a);
Classifier(c) _super(c,sc) _attribute(sc,a) ->
        _attribute(c,a);
Classifier(c) _coref(c,t) _attribute(c,a) name(a,n) ->
        Column(col) name(col,n) column(t,col);
```

Ces six règles ont alors les rôles suivants :

- la première règle entraine la création d'une table avec une colonne nommé "Id" pour chacune des classes du modèle à objets,
- la seconde règle et la troisième règle créent un lien temporaire *_super* permettant d'accéder directement à toutes les super-classes d'une classe du modèle à objets. Ce lien temporaire est utilisé dans la cinquième règle,
- la quatrième règle et la cinquième régle créent un lien temporaire *_attribute* entre les classes et tous leurs attributs (directs et hérités) et
- la dernière règle utilise ces liens temporaires de manière à créer une colonne pour chaque attribut (direct ou hérité) de chaque classe et l'associer à la table correspondante.

Maintenant, si l'on applique cet ensemble de règle tel quel dans notre moteur de transformations, il ne va jamais terminer la transformation ca il y aura toujours une règle d'applicable…

En effet, prenons la première règle : "si j'ai une classe c nommée n alors je crée une table t". Le moteur va appliquer cette règle tant qu'elle est applicable, et va par conséquent boucler car le fait de l'appliquer une fois lui permet toujours d'être réappliquée.

7.2.5.2 La pseudo variable None et l'opérateur "!".

Afin de résoudre le problème précédent, il y avait deux solutions. La première consistait à modifier le moteur de transformation et la seconde à modifier les règles.

La modification du moteur consistait à faire en sorte qu'une **même** règle ne soit pas applicable deux fois sur le **même** ensemble de variables liées au cours d'une transformation. Cette contrainte nous a semblée trop restrictive et nous avons préféré modifier les règles.

Pour modifier les règles, il nous a fallut introduire deux nouveaux artefacts :

- Une variable liée appelée None et pouvant être utilisée comme deuxième paramètre de l'expression d'une contrainte sur un lien dans la partie condition d'une règle. Le sens de cette contrainte est alors de dire qu'il n'existe pas de lien de ce type entre une variable liée à une entité et une autre entité.
- L'opérateur de négation symbolisé par le caractère "!" et pouvant être utilisé dans la partie condition d'une règle devant l'expression d'une contrainte sur un lien. Le sens de cette notation est de dire qu'il ne doit pas existe de lien entre les deux entités représentées par les deux variables liées utilisées dans cette contrainte sur un lien.

La première des règles décrites précédemment est alors modifiée de la façon suivante pour ne pas être réappliquée une seconde fois sur les mêmes entités (les parties modifiées sont en gras) :

```
Classifier(c) name(c,n)  _coref(c, None) ->

        Table(t) name(t,n) _coref(c,t) Column(id)
        name(id,"id") column(t,id);
```

La condition ajoutée ici permet de dire qu'il faut que la classe c ne dispose pas d'un lien *_coref* vers une autre entité pour que la règle puisse être appliquée. Comme ce lien _coref est créé dans la partie conclusion de la règle, celle-ci ne pourra être réappliquée une seconde fois sur la même classe c.

Attention, nous aurions pu vouloir corriger le problème en écrivant simplement :

Classifier(c) name(c,n) **! _coref (c, t)** ->

Mais, dans cette condition, la variable t va pouvoir être liée à **toutes** les entités qui ne sont pas liées à c via un lien *_coref* ! Il faut bien faire la différence entre l'utilisation de ces deux artefacts.

La seconde règle pourra quant à elle être modifiée de la façon suivante :

```
Classifier(c) generalization(c,g) supertype(g,sc) ! _super(c, sc) ->

        _super(c,sc);
```

En effet, dans cette règle ainsi modifiée, on ne crée le lien temporaire *_super* entre c et sc que s'il n'existe pas déjà.

L'ensemble des règles décrivant la transformation d'un modèle à objets vers un modèle relationnel et interpreté correctement par notre moteur de transformation est alors le suivant :

```
ObjectMetaModel -> RelationalMetaModel :

Classifier(c) name(c,n) _coref(c, None) ->

        Table(t) name(t,n) _coref(c,t) Column(id)
        name(id,"id") column(t,id);

Classifier(c) generalization(c,g) supertype(g,sc) ! _super(c, sc) ->

        _super(c,sc);

Classifier(c) Classifier(sc1) Classifier(sc2) _super(c,sc1) _super(sc1,sc2)
! _super(c, sc2) ->
        _super(c,sc2);

Classifier(c) feature(c,a) ! _attribute(c,a) ->

        _attribute(c,a);
```

Classifier(c) _super(c,sc) _attribute(sc,a) **!** _**attribute(c,a)** ->

_attribute(c,a);

Classifier(c) _coref(c,t) _attribute(c,a) name(a,n) **!** _**hasColumn(a,a)** ->

Column(col) name(col,n) column(t,col) **!** _**hasColumn(a,a)**;

Dans la dernière règle, on crée un lien temporaire *_hasColumn* de a vers lui-même qui va juste servir de marqueur pour indiquer que cette règle à été appliquée pour l'attribut référencé par a.

De cette façon, cet ensemble de règle est correctement interprété par notre moteur de transformation et la transformation de n'importe quel modèle à objets peut être effectuée avec celles-ci.

7.2.6 Prérequis pour effectuer une telle transformation.

Un moteur de transformation capable d'appliquer des règles telles qu'elles ont été formulées précédemment ne peut se faire que si le formalisme de représentation des modèles et des méta-modèles est suffisamment générique et propose des mécanismes basiques de création d'entités et de liens entre entités.

7.2.6.1 Transformation basée sur le formalisme des sNets.

En ce qui concerne les sNets, les interfaces présentées au chapitre 4.3 sont suffisantes pour permettre une telle transformation.

En effet la création d'une nouvelle entité de type "Table" par exemple dans l'univers cible peut s'effectuer simplement de la façon suivante via les interfaces définies dans le chapitre 4.3 :

// Recherche du type de l'entité à créer
// tm désigne l'univers de la cible

```
SemNodeNode type =  tm.getSem().getNodeNamed("Table");

// Création de l'entité de ce type dans l'univers cible

SemNode nouvelleEntité = tm.newNodeOfType(type);
```

De la même façon, la création d'un lien "*column*" entre une entité liée à une variable t et une entité liée à une variable c s'effectue simplement de la façon suivante en utilisant ces même interfaces :

```
// Obtention des entités liées aux variables t et c

SemNode sourceNode = variables.getValue("t");
SemNode destinationNode = variables.getValue("c");

// Création d'un lien "column" entre une entité liée à la variable t et une entité liée à la variable c

sourceNode.setLinkToNode("column",destinationNode)
```

Le moteur de transformation peut ainsi se baser sur de telles interfaces et devenir indépendant des méta-modèles des modèles à transformer.

7.2.6.2 Transformation basée sur le MOF.

Le moyen que propose le MOF pour manipuler de façon dynamique des modèles est le mapping vers des interfaces IDL (c.f. chapitre 3.2.1). Il est alors possible, sous réserve que l'invocation dynamique de méthode soit disponible, de baser le moteur de transformation sur de telles interfaces. En effet, nous allons voir qu'a la différence du formalisme des sNets, le nom des méthodes à invoquer est à déterminer lors de l'exécution.

Par exemple, si l'on désire créer une nouvelle entité de type "Table" dans le modèle cible, il faudra procéder de la façon suivante :

// obtention d'un objet de type Tableclass capable de créer de nouvelles entités de type Table dans le modèle cible représenté par la variable model.

Tableclass metaObject = model.Table_class_ref();

// Invocation de la méthode create_Table sur cet objet et obtention du nouvel objet de type Table

Table newEntity = metaObject.create_Table();

En fait, de façon générique, pour créer une nouvelle entité de type <X> dans le modèle cible, il faudra invoquer dynamiquement les méthodes suivantes :

// obtention d'un objet de type <X>class capable de créer de nouvelles entités de type <X> dans le modèle cible représenté par la variable model.

<X>class metaObject = model.<X>_class_ref();

// Invocation de la méthode create_<X> sur cet objet et obtention du nouvel objet de type <X>

<X> newEntity = metaObject.create_<X>();

Le même type d'invocation dynamique devra être effectué de manière à créer un lien entre deux entités.

Le transformateur à été réalisé à partir de modèles basés sur le formalisme des sNets, mais il ne semble donc pas impossible de baser un tel transformateur sur le MOF.

Nous avons donc montré que les techniques de méta-modélisation pouvaient nous permettre d'exprimer de manière formelle des règles de transformation de modèle et que les formalismes basés sur un méta-méta-modèle proposaient des mécanismes permettant de construire un transformateur générique capable d'appliquer de telles règles.

7.3 Comparaison avec PROGRES et hypergenericity

Dans Progres, les transformations sont décrites par des règles de production. Ce type de règles est assez proche du formalisme que nous venons de définir. En ce qui concerne hypergenericity, les transformations sont décrites dans un langage à objets appelé H. Ce langage permet de manipuler le modèle et de créer, supprimer et modifier les éléments qu'il contient. Alors que Progres, tout comme ce que l'on vient de définir, s'appuie sur la réécriture de graphes, le langage H est quant à lui associé à un méta-modèle spécifique (le méta-modèle d'UML).

L'exemple suivant présente les règles, définies en utilisant chacun de ces trois formalismes, permettant d'ajouter une méthode "print" à un ensemble de classes d'un modèle à objets.

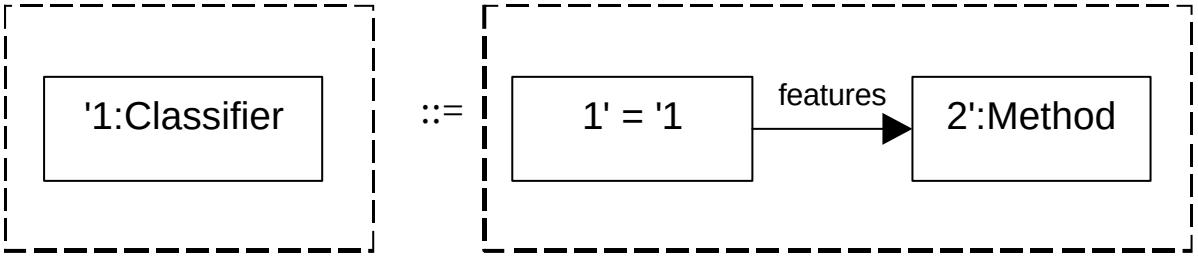
En utilisant le formalisme que nous venons de définir, une telle règle s'écrira de la façon suivante :

```
ObjectMetaModel -> ObjectMetaModel :

Classifier(c) ! _printMethod(c,m) ->

        Method(m) name(m,"print") features(c,m) _printMethod(c,m) ;
```

Pour chaque classe c du modèle à objets n'ayant pas déjà de méthode print, nous définissons une nouvelle méthode que nous appelons print et que nous ratachons à c via le lien *features*.

Avec le formalisme de Progres, une telle règle sera représentée par une règle de production de la façon suivante :

production createPrintMethod() =



```
transfer 2'.name := "print";
end;
```

Tandis qu'avec le langage H, il sera nécessaire de programmer toute l'opération de la façon suivante :

```
C : Class;
m : Method;
m := Method.create();
m.setName("print");
C.addComponent(m);
```

7.4 Conclusion.

Actuellement, les techniques de méta-méta-modélisation sont largement utilisées dans les outils de modélisation. De plus en plus de méta-modèles sont standardisés, mais peu d'outils sont capables de permettre la manipulation de plusieurs modèles, basés sur des méta-modèles différents, en même temps. Pour faciliter le développement de tels outils, une architecture de méta-modélisation doit être clairement définie et nous avons déjà montré que ce n'était pas encore le cas (c.f. chapitre 6). De plus en plus de traitements pourraient être réalisés dans un tel environnement et la définition d'un outil de transformation de modèles tel que nous venons de présenter dans ce chapitre est un exemple d'un tel traitement. Notre but, en définissant précisément et explicitement le formalisme des sNets, est de proposer un tel environnement.

8 Méta-modélisation et langage de requête.

Après avoir montré qu'il était possible de construire des outils et des formalismes génériques de transformation et d'expression de règles de transformation, nous allons montrer qu'il est également possible de définir un langage de requête générique ainsi qu'un outil d'application de ces requêtes permettant d'effectuer des recherches dans un modèle ou un méta-modèle quelconque. Le formalisme que nous avons défini est très inspiré des formalismes d'interrogation de bases de données relationnelles (SQL) ou objets (OQL). Nous y avons intégré un certain nombre de spécificités liées à notre formalisme.

Ce langage de requêtes a été développé en début de thèse et le langage d'expression de contrainte OCL n'existait pas encore. Si nous devions redévelopper ce type d'outil maintenant, nous nous rapprocherions certainement plus du formalisme OCL pour exprimer nos requêtes.

L'outil qui a été développé dans le cadre de cette thèse et qui est capable d'appliquer les requêtes que nous allons vous présenter dans ce chapitre se nomme sNets Request Executor et fonctionne sur n'importe quel type de modèles.

8.1 Les requêtes de base.

Ce langage est un langage de requête associé aux sNets. Sa syntaxe est proche de celle de SQL. Il permet d'effectuer tout type de recherche dans le sNet. Sa forme la plus simple est la suivante :

```
select node from universe
```

Cette requête va renvoyer une collection de toutes les entités du sNet appartenant à l'univers appelé universe. node désigne ici les entités renvoyées par la requête.

Pour obtenir la liste des entités d'un type T de l'univers universe, on écrira la requête suivante :

```
select node
from universe
where node isA T
```

Si l'on veux toutes les entités de type T ou d'un type T', sous-type de T, on écrira :

```
select node
from universe
where node isKindOf T
```

De la même manière que **isA** et **isKindOf**, il est possible d'utiliser les mot-clés **name** et **partOf**. Ainsi si l'on veux toutes les entités nommés 'A', on écrira la requête suivante :

```
select node
from universe
where node name 'A'
```

Il est également possible de composer l'expression conditionnelle de la clause **where** avec les opérateurs logiques **and**, **or** et **not**. Ainsi, si l'on veux obtenir, dans un universe représentant du code Cobol, toutes les entités de type CobolData, excepté celles de type LiteralData, on écrira la requête suivante :

```
select variable
from cobolProgram
where variable isKindOf CobolData
and not variable isA LiteralData
```

De la même façon, si l'on souhaite, dans un modèle à objets conforme au méta-modèle UML, obtenir toutes les entités de type **Class** et **Interface**, mais pas celles de type **DataType**, il suffira d'écrire la requête suivante :

```
select type
from objectModel
where type isKindOf Classifier
and not type isA DataType
```

Le moteur d'exécution de ces requêtes a été entièrement développé autour des interfaces génériques définies dans le chapitre 4.3 de sorte qu'il s'appliquera (tout comme notre outil de transformation) aussi bien à un modèle objet, à un modèle représentant un programme Cobol ou à tout autre type de modèle ou de méta-modèle.

8.2 Les expressions de liens.

Le langage de requête gagne en puissance dès que l'on utilise la structure même du sNet, c'est à dire, dès que l'on exprime des conditions sur les relations entre les différentes entités du réseau. Cette fonctionnalité peut s'apparenter aux jointure de tables dans les langages de requête relationnels. Ainsi, si l'on veux l'ensemble des couples composés des variables obtenues dans la précédente requête sur le programme Cobol et de leur "picture" (format) respective. Il suffit de savoir que dans le sNet, une variable est reliée à sa picture par un lien *picture*. La requête est alors la suivante :

```
select aVariable, aPicture
from cobolProgram
where aVariable isKindOf CobolData
and not aVariable isA LiteralData
and aVariable <picture> aPicture
```

La condition « aVariable <picture> aPicture » ne sera vérifiée que si l'entité représentée par aVariable dispose d'un lien *picture* vers l'entité représentée par aPicture. On obtient donc une collection de couples { aVariable, aPicture } pour lesquels aVariable est de type **CobolData**, n'est pas un **LiteralData**, et a pour format aPicture.

En utilisant le Semantic Request Executor, la requête est la suivante (le programme cobol se trouve dans un univers appelé KP7200) :

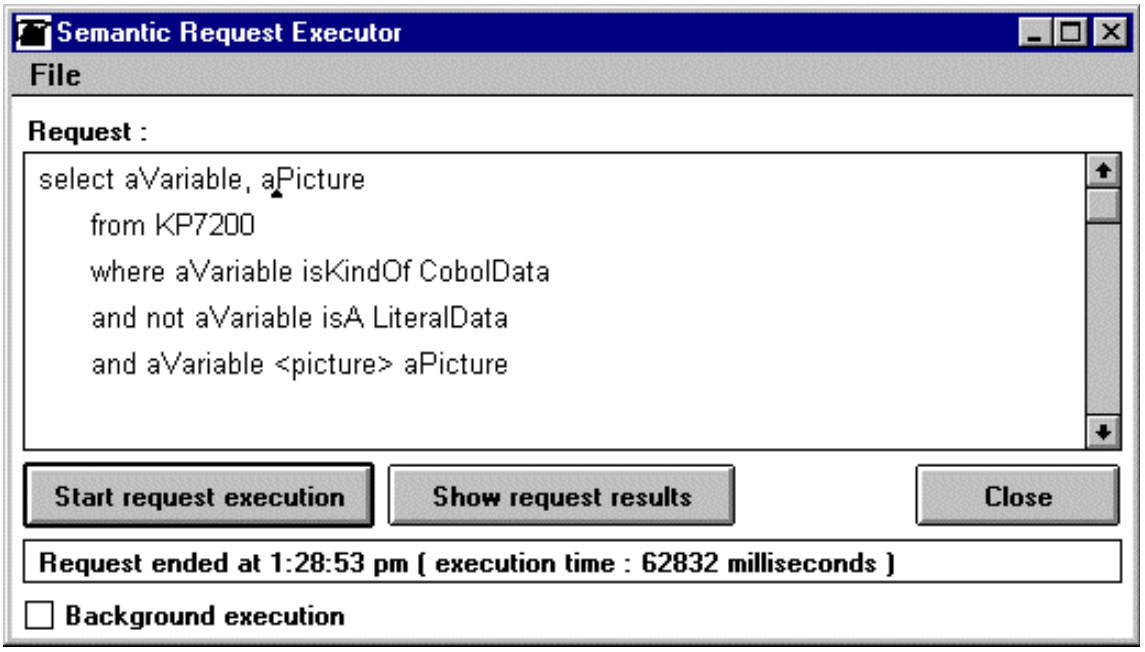


Figure 104 - L'outil de requête utilisé sur un modèle représentant un programme Cobol.

Le résultat de cette requête est obtenu dans une fenêtre appelée Semantic Request Result. Il se présente alors de la façon suivante dans cette fenêtre :

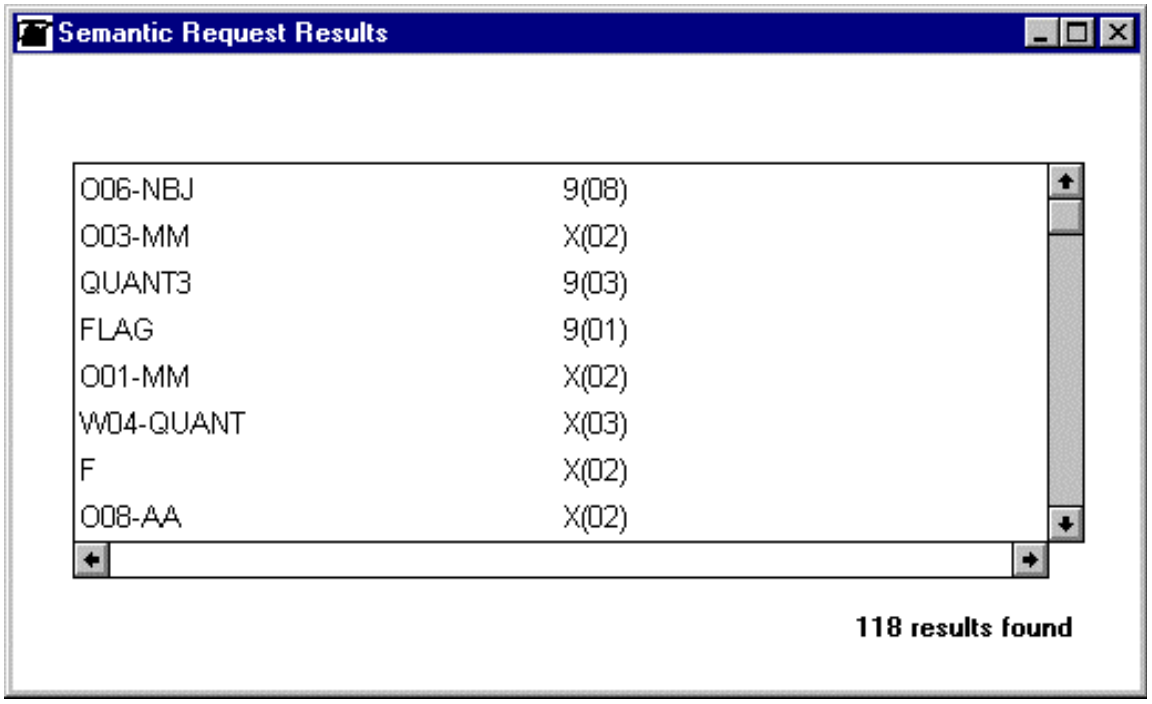


Figure 105 - Résultat d'une requête effectuée sur un modèle représentant un programme Cobol.

Chaque ligne correspond alors à un résultat de la requête précédente. Ainsi, nous pouvons déduire que OO6-NBJ est une variable du programme Cobol KP7200 dont le format est défini par la picture '9(08)''.

8.3 La clause Order by.

Le résultat de la requête précédente est constitué de toutes les variables non littérales du programme, ainsi que de leur format. Comme on le constate sur cette figure, l'ordre des variables est totalement indéterminé. Il est alors intéressant d'ordonner le résultat de la requête. Pour déterminer cet ordre, on utilise la clause **order by**. On modifie donc la requête précédente de la façon suivante :

```
select aVariable, aPicture
from cobolProgram
where aVariable isKindOf CobolData
```

and not aVariable **isA** LiteralData
and aVariable <picture> aPicture
order by aVariable

De même, il est intéressant de pouvoir libeller les colonnes du résultat de la requête de manière à savoir ce que chacune de ces colonnes représente. Il est possible de rajouter un libellé sous forme d'une chaîne de caractères placée derrière le nom de la variable dans la liste des variables de la requête. On obtient donc la requête suivante :

select aVariable 'une variable de KP7200', aPicture 'et sa picture'
from cobolProgram
where aVariable **isKindOf** CobolData
and not aVariable **isA** LiteralData
and aVariable <picture> aPicture
order by aVariable

Et le résultat de cette nouvelle requête se présente de la façon suivante :

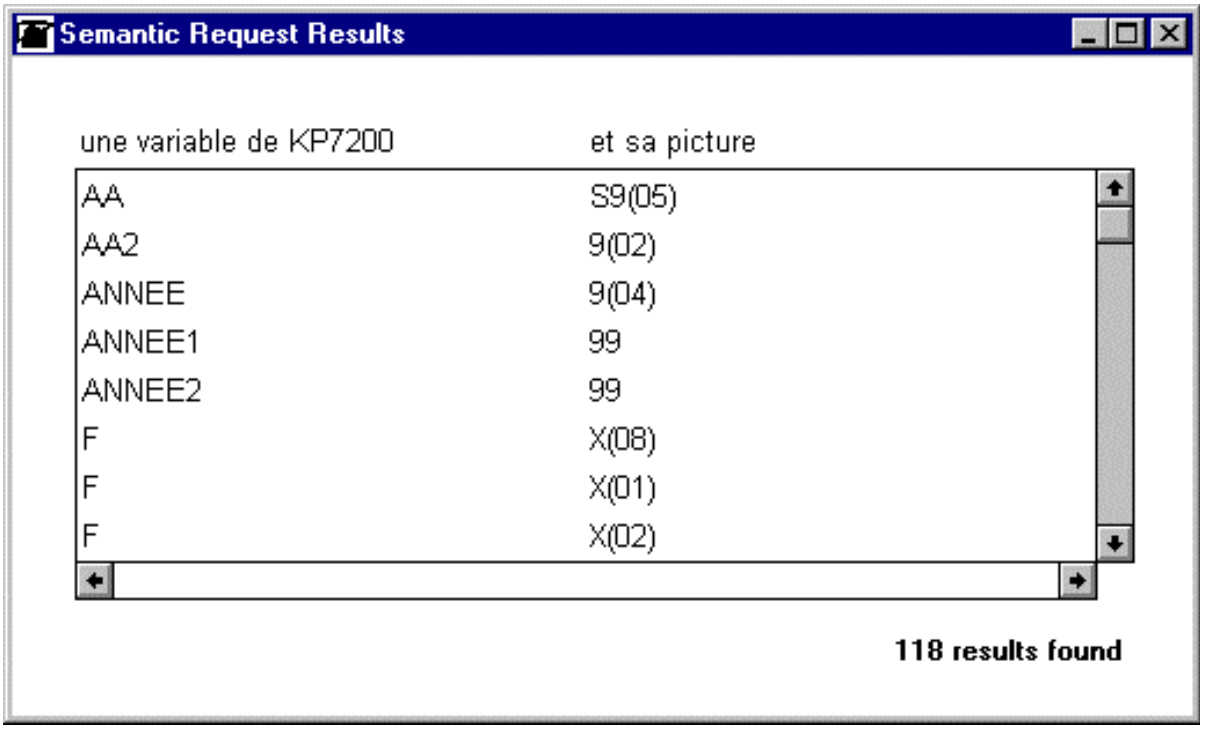


Figure 106 - Résultat d'une requête effectuée sur un modèle représentant un programme Cobol (avec tri et libellés).

De la même façon, on peut rechercher toutes les méthodes de toutes les classes d'un modèle à objet avec une requête qui aurait la forme suivante :

```
select type 'classe du modèle', method 'méthode de cette classe'
from objectModel
where type isKindOf Class
and method isKindOf Operation
and type <feature> method
```

8.4 Les opérateurs de comparaison.

Des opérateurs de comparaison sont également définis. On peut ainsi vérifier que deux entités sont identiques (opérateur =), vérifier que deux entités sont différentes (opérateur **!=**), mais également vérifier que le nom d'une entité concorde avec un masque donné (opérateur **match**).

Ainsi, pour obtenir la liste de toutes les variables précédemment obtenue dont le nom contient la chaîne 'ANNEE', il suffit de complèter la requête précédente de la façon suivante :

```
select aVariable 'une variable de KP7200', aPicture 'et sa picture'
from cobolProgram
where aVariable isKindOf CobolData
and not aVariable isA LiteralData
and aVariable <picture> aPicture
and aVariable match '*ANNEE*'
order by aVariable
```

8.5 La clause using.

Dans le méta-modèle Cobol utilisé pour représenter les programmes Cobol, les entités de type CobolVariable peuvent disposer d'un lien isUsedBy vers les entités de type CobolStatement (les instructions Cobol) qui les utilise dans le programme. On peut donc obtenir la liste de toutes les variables effectivement utilisées par un programme à l'aide de la requête suivante :

```
select aVariable 'une variable de KP7200', aStatement 'utilisée par'
```

from cobolProgram
where aVariable **isKindOf** CobolData
and aVariable <isUsedBy> aStatement

Maintenant, il se peut que l'on s'interesse aux variables effectivement utilisées, sans pour autant s'intéresser aux instructions qui les utilisent. Dans ce cas, on place la variable aStatement dans une clause appelée clause **using**. Elle indique les variables utilisées pour exprimer la condition, mais non présentes dans le résultat de la requête. La requête devient alors :

select aVariable 'une variable utilisée par KP7200'
using aStatement
from cobolProgram
where aVariable **isKindOf** CobolData
and aVariable <isUsedBy> aStatement

Cette requête renvoie alors toutes les entités de type **CobolData** disposant d'un lien *isUsedBy* vers une entité de type quelconque.

8.6 L'imbrication des requêtes.

La requête précédente renvoie donc la liste des variables effectivement utilisées par le programme. Nous pouvons utiliser le résultat de cette requête pour obtenir la liste des variables non utilisées par le programme. Cette nouvelle requête s'écrit alors de la façon suivante :

select aVariable 'une variable non utilisée par KP7200'
from cobolProgram
where aVariable **isKindOf** CobolData
and aVariable **not in**
    (**select** aUsedVariable
    using aStatement
    from cobolProgram
    where aUsedVariable **isKindOf** CobolData
    and aUsedVariable <isUsedBy> aStatement)

Cette méthode fonctionne, mais peux sembler quelque peu laborieuse. Il arrive souvent que l'on veuille exprimer le fait qu'une entité ne dispose pas d'un lien particulier (ici, le lien *isUsedBy*). Nous avons donc défini une valeur appelée **none** et utilisée dans les expressions de chemin pour indiquer qu'un lien n'existe pas a partir d'une entité donné. La requête précédente peut alors s'écrire de la façon suivante en utilisant ce terme **none** :

```
select aVariable 'une variable non utilisée par KP7200'
from cobolProgram
where aVariable isKindOf CobolData
and aVariable <isUsedBy> none
```

On exprime donc ici que l'on désire l'ensemble des entités de type **CobolData** de l'univers cobolProgram ne disposant pas de lien *isUsedBy*.

8.7 Les expressions de chemins.

Lorsque l'on exprime une relation entre deux entités, il est possible de l'exprimer non seulement sous la forme d'un lien simple, mais également sous la forme d'une expression de chemin.

Par exemple, lorsque l'on écrit :

```
aVariable isKindOf CobolData
```

C'est comme si l'on écrivait l'expression de chemin suivante :

```
aVariable <meta inherits * name> 'CobolData'
```

En effet, cette expression sera vrai si et seulement si il est possible, a partir de l'entité aVariable, d'aboutir à l'entité 'CobolData' en empruntant un chemin constitué, dans un premier temps, d'un lien meta, puis d'un certain nombre de liens inherits (peut-être aucun), et enfin un lien name. Cette expression de chemin correspond effectivement a la dause «aVariable **isKindOf** CobolData » du fait de la structure même du formalisme de sNets. En effet, toute entité du sNets dispose d'un lien meta vers un noeud de type Node décrivant son type. De même, tout noeud de type Node dispose d'un lien inherits vers son super type et d'un lien name vers son nom.

8.8 Les actions sémantiques.

Les actions sémantiques sont des actions liées à un type d'entité. Nous ne nous étendons pas sur cette fonctionnalité, mais elle est à rapprocher aux travaux effectués sur les langages d'action associés aux langages de modélisation. Dans notre formalisme, nous n'avons pas redéfini de langage particulier pour coder ce type d'actions, nous nous sommes contenté de reprendre le langage d'implémentation du formalisme. Ainsi, dans l'implémentation Smalltalk du formalisme des sNets, les actions sont écrites en Smalltalk. Tandis que dans l'implémentation Java des sNets, elles doivent être écrites en Java. Le principe de fonctionnement est similaire à la définition de méthodes sur des classes d'un langage à objets. Ici, on définie des actions sémantiques sur des types d'entités.

Il est possible de placer dans la clause where des appels à ces actions sémantiques. Ces actions sont alors sensées renvoyer une valeur logique. Ainsi, de la même façon que l'on pouvait remplacer l'expression « aVariable **isKindOf** CobolData » par une expression de chemin, il est possible de la remplacer par une action sémantique.

Il suffit de définir une action sémantique appelée par exemple isCobolData sur le type d'entité **CobolData**. Cette action doit alors renvoyer vrai sachant qu'elle doit être exprimée dans le langage d'implémentation du formalisme en cours d'utilisation. Il faut ensuite définir une autre action sémantique de même nom sur la méta-entité racine de l'univers sémantique CobolSem avec pour valeur renvoyée faux de sorte que l'appel de cette action sur tout autre élément qu'un élément de type **CobolData** renverra faux.

La clause :

```
aVariable isKindOf CobolData
```

Pourra alors être remplacée par la clause suivante :

```
aVariable isCobolData()
```

Cette nouvelle expression a pour effet de faire appel a l'action sémantique indiquée et définie pour l'entité aVariable. De plus, elle est beaucoup plus performante que l'expression « aVariable **isKindOf** CobolData » qui utilise une expression de chemin.

8.9 La source de la requête.

La requête s'effectue à partir des entités contenus dans la liste des univers indiquée dans la clause FROM. Ce qui veux dire que les variables indiquées dans la clause SELECT et dans la clause USING appartiendront obligatoirement à l'un des univers indiqué dans cette liste. Lorsqu'un nom d'univers est indiqué, cela veux dire que toutes les entités appartenant à cet univers seront "candidates". Mais si cet univers contient d'autres univers, les entités de ces univers ne seront pas elles même "candidates". Pour indiquer que la requête doit s'effectuer sur un univers donné, ainsi que sur tous les univers qu'il contient, il faut rajouter le mot-clé **recursive** derrière le nom de l'univers. Ainsi pour lancer une requête sur un ensemble de programmes cobol présents dans un univers cobolPrograms, par exemple, il suffira d'indiquer comme source de la requête : cobolPrograms **recursive**.

Ainsi pour obtenir par exemple les tous les types d'entité définis dans un ensemble d'univers sémantiques regroupés dans un univers semantics, on écrira la requête suivante :

```
select aType
from semantics recursive
where aType isA EntityType
```

8.10 Un exemple de requête complexe.

Une requête complexe peut être la recherche, dans tous les programmes cobol contenus dans cobolPrograms de toutes les données susceptibles de représenter des dates non "conformes". Par date non "conforme", on entend date sans siecle. Ce type de requête était bien évidemment très utile pour s'assurer qu'un programme cobol est à même de "passer" l'an 2000.

Pour effectuer cette requête, on utilise une action sémantique qui indique, pour un noeud de type **CobolData**, la taille exacte en octets utilisée par la variable. Cette action appelée byteLengthIs prend un entier en paramètre et renvoie vrai si la longueur de la donnée est égale à cette valeur.

La requête est alors la suivante :

```
select      dateVariable 'Donnee ressemblant a une date sans siecle',
            lineNumber 'declaree a la ligne',
            prog 'du programme'

      from cobolPrograms recursive
      where
      (dateVariable in (
      select dateVariable
         from cobolPrograms recursive
         where    dateVariable isCobolData()
         and      ( dateVariable match '*DAT*'
         or    dateVariable match '*AAMMJJ*'
         or    dateVariable match '*JJMMAA*' )
         and dateVariable byteLengthIs (6) )

      or dateVariable in (

      select dateVariable
         using field1, field2, field3
         from cobolPrograms recursive
         where    dateVariable isCobolData()
         and      dateVariable <containsAFirst> field1
         and      field1 <isFollowedBy> field2
         and      field2 <isFollowedBy> field3
         and      field3 <isFollowedBy> None
         and      (field1 <containsAFirst> None)
         and      (field2 <containsAFirst> None)
         and      (field3 <containsAFirst> None)
         and      ( (( field1 match '*AA*' or field1 match '*AN*') and field1
                  byteLengthIs (2) )
         or    (( field2 match '*AA*' or field2 match '*AN*') and field2 byteLengthIs (2) )
         or    (( field3 match '*AA*' or field3 match '*AN*') and field3 byteLengthIs (2) )
         ) )
      )
```

```
    and dateVariable <lineNumber> lineNumber
    and dateVariable <partOf> prog
    order by prog, dateVariable
```

Cette requète est composée de deux sous-requêtes. La première va rechercher toutes les données Cobol dont le nom est '*DAT*', '*AAMMJJ*' ou '*JJMMAA*' et dont la taille est de 6 octets. Tandis que la seconde va rechercher toutes les données qui se décomposent en trois sous champs non décomposables et dont l'un des trois fait une taille de deux octets et se nomme '*AA*' ou '*AN*'.

Ce type de recherche, qui s'exprime "simplement" dans un formalisme comme celui-ci, est très complexe à réaliser sans ce type d'outils.

8.11 Grammaire du langage de requêtes.

Cette grammaire est présentée en utilisant la norme BNF. La notation utilisée dans la grammaire est la suivante :

- [symbol] indique que symbol est facultatif.
- **keyword** représente un mot clé terminal.
- xxx_name représente un nom d'identificateur.
- xxx_literal représente un littéral.

L'axiome principal est le suivant :

```
sNetsRequest    ::=    select variableList
                       [using tempVariableList]
                       from universeList
                       [where condition]
                       [order by orderList]
```

La liste des variables est décrite de la façon suivante :

```
variableList   ::=    variableList , variable [label]
variableList   ::=    variable [label]
```

Il en est de même pour la liste des variables temporaires :

```
tempVariableList   ::=   tempVariableList , variable
tempVariableList   ::=   variable
```

La liste des univers est décrite de la façon suivante :

```
universeList  ::=   universeList , universe [recursive]
universeList  ::=   universe [recursive]
```

Les conditions sont décrites de la façon suivante :

// operateurs logiques

```
condition     ::=   condition or condition
condition     ::=   condition and condition
condition     ::=   not condition
condition     ::=   (condition)
```

// operateurs de comparaison

```
condition     ::=   basic != basic
condition     ::=   basic = basic
condition     ::=   variable match string
```

// operateurs liés aux liens prédéfinis

```
condition     ::=   variable name string
condition     ::=   variable name node_name
condition     ::=   variable partOf string
condition     ::=   variable partOf node_name
condition     ::=   variable isA string
condition     ::=   variable isA node_name
condition     ::=   variable isKindOf string
condition     ::=   variable isKindOf node_name
```

// operateurs liés aux expressions de chemin

condition ::= variable **<** pathExpression **>** basic
condition ::= variable **<** pathExpression **> none**
pathExpression ::= pathExpression link
pathExpression ::= **(** orExpression **)**
pathExpression ::= link
orExpression ::= orExpression **|** link
orExpression ::= link **|** link
link ::= link_name [**+**]
link ::= link_name [*****]
link ::= string [**+**]
link ::= string [*****]

// operateur d'imbrication de requêtes

condition ::= variable [**not**] **in (** SNetRequest **)**

// operateur d'appel à une action sémantique

condition ::= variable action **(** parameters **)**
condition ::= variable action **()**
parameters ::= parameters **,** basic
parameters ::= basic

Les variables de la clause order sont décrites de la façon suivante :

orderList ::= orderList **,** variable
orderList ::= variable

Et les terminaux sont les suivants :

variable ::= variable_name
action ::= action_name
label ::= string_literal

string	::=	string_literal
basic	::=	variable_name
basic	::=	string_literal
basic	::=	integer_literal
basic	::=	**true**
basic	::=	**false**

8.12 Conclusion.

Les outils de modélisation proposent généralement des mécanismes plus ou moins avancés pour effectuer des recherches sur les éléments qu'ils permettent de modéliser. L'exemple présenté ici montre qu'il a tout à fait sa place dans un logiciel de rétro-documentation pour lequel les modèles manipulés ne sont pas des modèles à objets, mais des programmes cobols. La standardisation d'un formalisme de représentation de modèles et de méta-modèles (tel que le MOF, le formalisme des sNets, ou CDIF) permet donc le développement d'un outil générique pouvant être utilisé quelles que soient les données modélisées.

9 Application des sNets.

Ce chapitre présente rapidement les différentes applications du formalisme des sNets. Il montre que ce formalisme est à même de permettre la modélisation de données ou plus généralement d'univers très divers. En effet, cela va d'un ensemble de programmes COBOL avec l'outil Semantor© à un modèle UML avec l'outil Scriptor© en passant par un outil générique de modélisation et de méta-modélisation regroupant tous les outils que nous avons développé au cours de cette thèse sur le formalisme des sNets :

- un navigateur permettant de naviguer indifféremment sur les modèles, les méta-modèles ou encore le méta-méta-modèle de notre formalisme,
- un grapheur permettant de construire graphiquement un nouveau méta-modèle, un nouveau méta-modèle, mais également d'enrichir le méta-méta-modèle de notre formalisme,
- des outils d'importation et d'exportation de nos modèles et méta-modèles au format XML,
- un outil de transformation de modèles présenté chapitre 7.

L'outil de navigation est également présent, pour les utilisateurs expérimentés, dans les outils Scriptor© et Semantor©.

Quant au langage de requêtes, une implémentation adaptée au formalisme des sNets est présente dans l'outil Semantor©. Il permet par exemple de rechercher dans un ensemble de programmes Cobol les données de type date disposant d'un siècle codé sur deux caractères ainsi que les instructions qui les manipulent.

9.1 Semantor©.

Basé sur le formalisme des sNets et une analyse détaillée de la sémantique COBOL et de ses dérivés, Semantor© créé une véritable base de connaissances des applications informatiques.

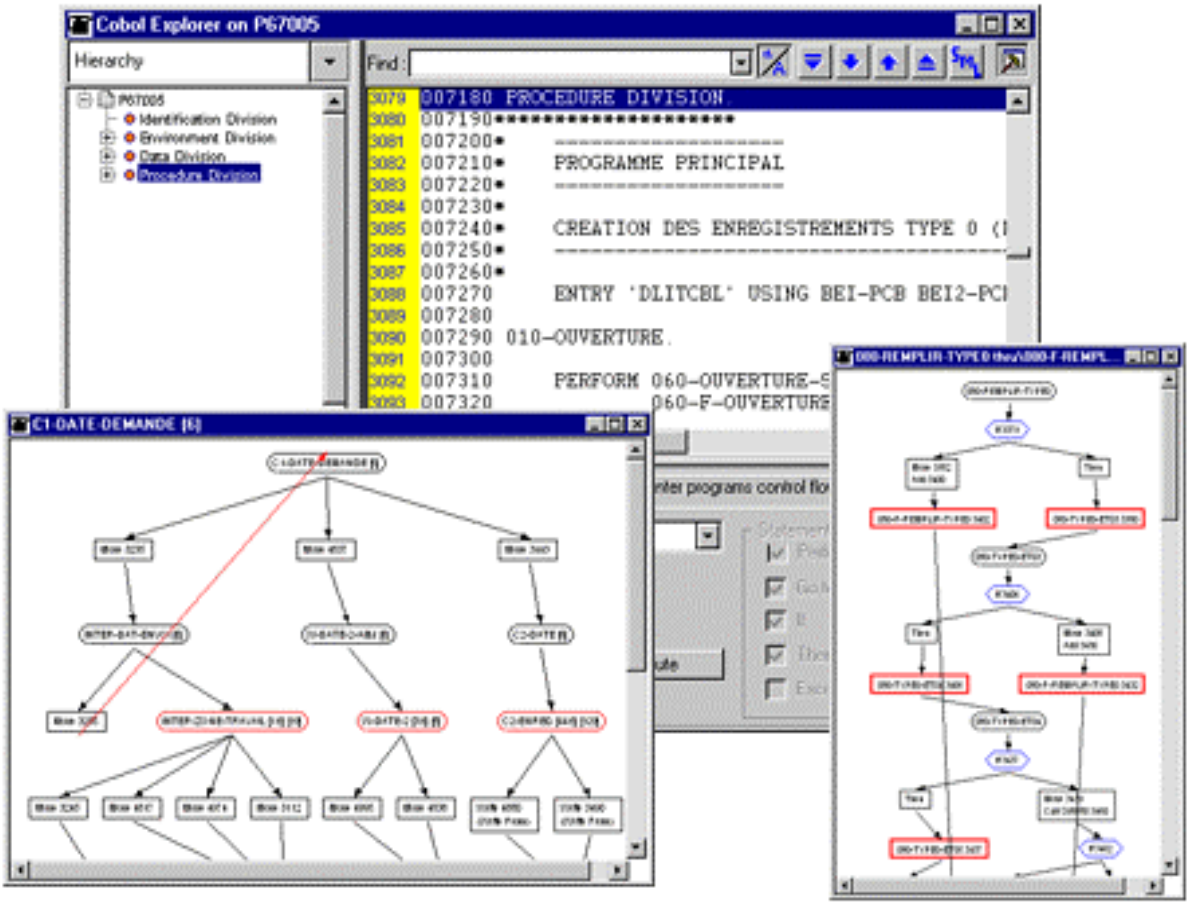


Figure 107 - L'application de rétro-ingénierie Semantor ©, basée sur le formalisme des sNets.

Les éléments suivants peuvent ainsi être extraits d'un ensemble de programmes COBOL :

- graphes de flot de contrôle et flot de données,
- extraits de sources,
- tableaux de métriques, …

De plus, les analyses de Semantor© ne se limitent pas aux programmes. Elles prennent en compte l'ensemble des supports d'information de l'application (fichiers physiques, enregistrements IDSII, segments DL1, copy Cobol, ...).

En synchronisant tous ses résultats d'analyses avec la visualisation du code source correspondant, Semantor© permet de naviguer très simplement au sein des applications car toutes ces données sont représentées dans un seul et même réseau sémantique. Tous les liens implicites qui étaient présents entre ces différents éléments peuvent alors être explicités et rendre la navigation plus simple.

Réaliser une analyse de flot de données véritablement exhaustive, seul gage d'une analyse d'impact fiable, est une opération excessivement complexe. Par une approche originale de cette question, Semantor© permet de prendre en compte de manière exhaustive toutes les propagations, quelles soient implicites (par les structures parentes, les redéfinitions, ...), ou liées à des problèmes de cadrage - troncature lors d'affectation de données de formats différents.

9.2 Scriptor©.

Scriptor© est un outil de génération de code à base de scripts de génération wisiwig (What you see is what you get - Ce que vous voyez est ce que vous obtiendrez). Dans Scriptor© UML, ces scripts sont répartis sur les différents éléments du méta-modèle de UML. Né d'un besoin de productivité et de réactivité sur un grand projet de développement objet, Scriptor© assure la continuité entre les différentes phases de conception, de réalisation et de maintenance.

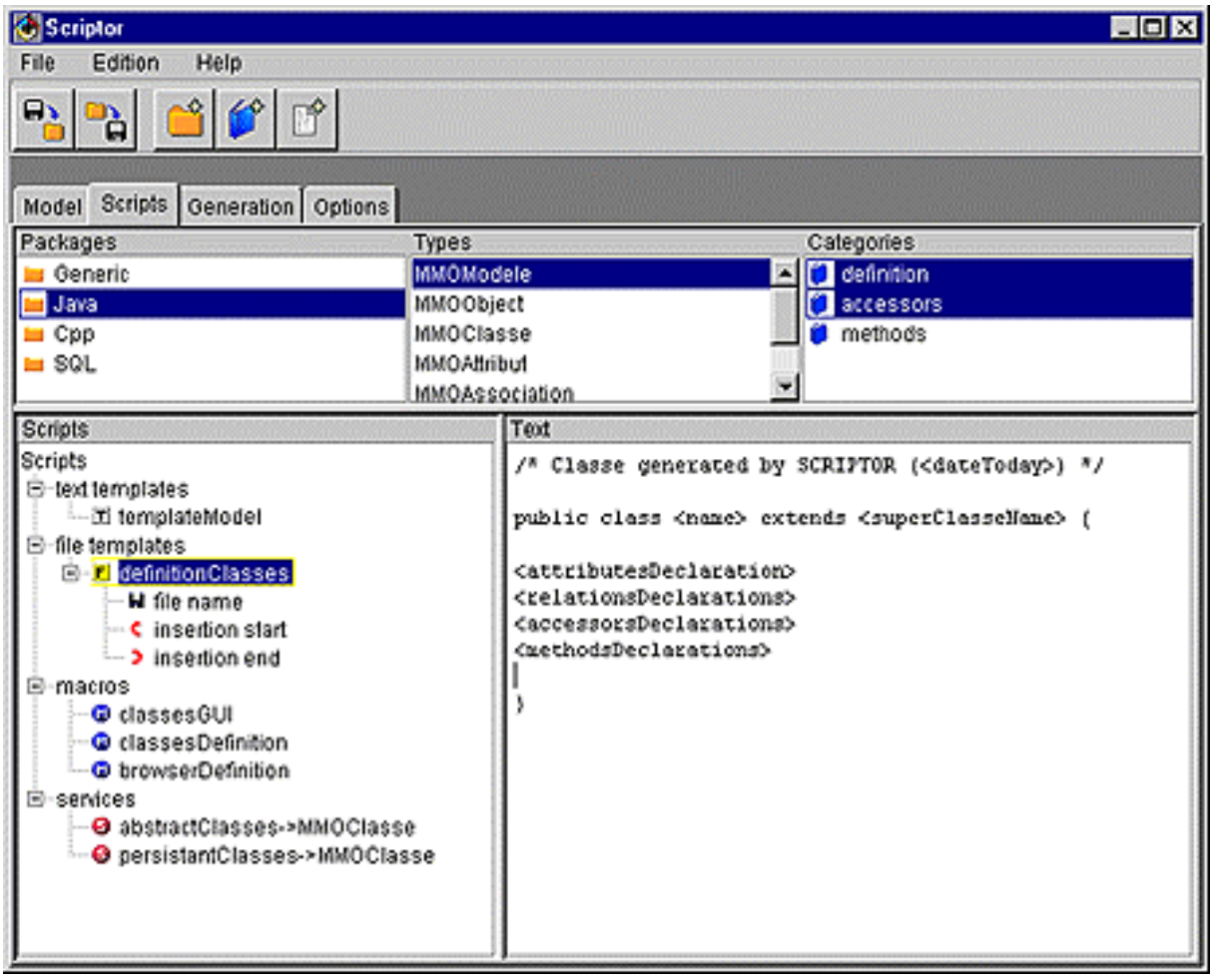


Figure 108 - L'outil de génération de code Scriptor ©, basé sur le formalisme des sNets.

Scriptor© est un outil de génération de code à base de scripts de génération répartis sur les différents éléments du méta-modèle. Ces scripts de génération sont à la fois très simples à

construire, puisque basés sur un principe wisiwig, et d'un grand pouvoir d'expression et de généricité, du fait de l'incorporation de macro-instructions écrites en langage Java.

Au sein de l'outil Scriptor, le méta-modèle UML est exprimé avec le formalisme de sNets. Il en est de même pour les modèles UML importés dans Scriptor à partir des différents outils de modélisation. Ce formalisme permet à l'outil d'être complètement indépendant du méta-modèle sous-jacent et de profiter de tous les composants développés autour de ce formalisme comme l'import et l'export au format XMI par exemple.

De plus, les différents scripts de l'outil sont rattachés aux entités de type EntityType du méta-modèle et le mécanisme de gestion des scripts est totalement indépendant du méta-modèle tandis que les scripts peuvent utiliser la notion d'héritage définie pour les entités de ce type et profiter du polymorphisme.

Acceptant des modèles issus de différents AGL (Mega, Rose, Paradigm Plus, Power AMC, …) et facilement adaptable à tout type d'outil propriétaire, il permet aussi bien de :

- générer du code (génération de composants EJB, de code JSP, d'implémentations Java, de structure de tables, de code smalltalk, C++, etc..), ou de la documentation
- transformer des modèles dans le cadre d'un changement d'AGL
- établir des ponts entre logiciels

De plus, toute la partie du code source de Scriptor© spécifique au méta-modèle de UML est générée à partir de Scriptor© en appliquant des scripts de génération sur ce méta-modèle. Une version de Scriptor© dédiée à la représentation de processus a ainsi été obtenue en appliquant ces mêmes scripts sur le méta-modèle de processus que nous avons défini. Cette version de Scriptor est notamment utilisée pour faire le lien entre l'outil de modélisation de processus et les moteurs de Workflow. Elle permet de générer la totalité du code nécessaire au moteur de Workflow pour exécuter le modèle et s'affranchir ainsi de travaux de paramètrage parfois lourds et complexes sur ces moteurs.

9.3 Un atelier de modélisation et de méta-modélisation.

Cet atelier regroupe tous les outils que nous avons défini autour du formalisme des sNets. C'est une plateforme idéale pour la validation de modèles et de méta-modèles. Il permet la confection interactive de méta-modèles. La Figure 109 présente le navigateur générique qui est ici utilisé pour naviguer sur le méta-modèle en cours de construction et elle présente également le grapheur générique qui a été utilisé pour construire graphiquement ce méta-modèle.

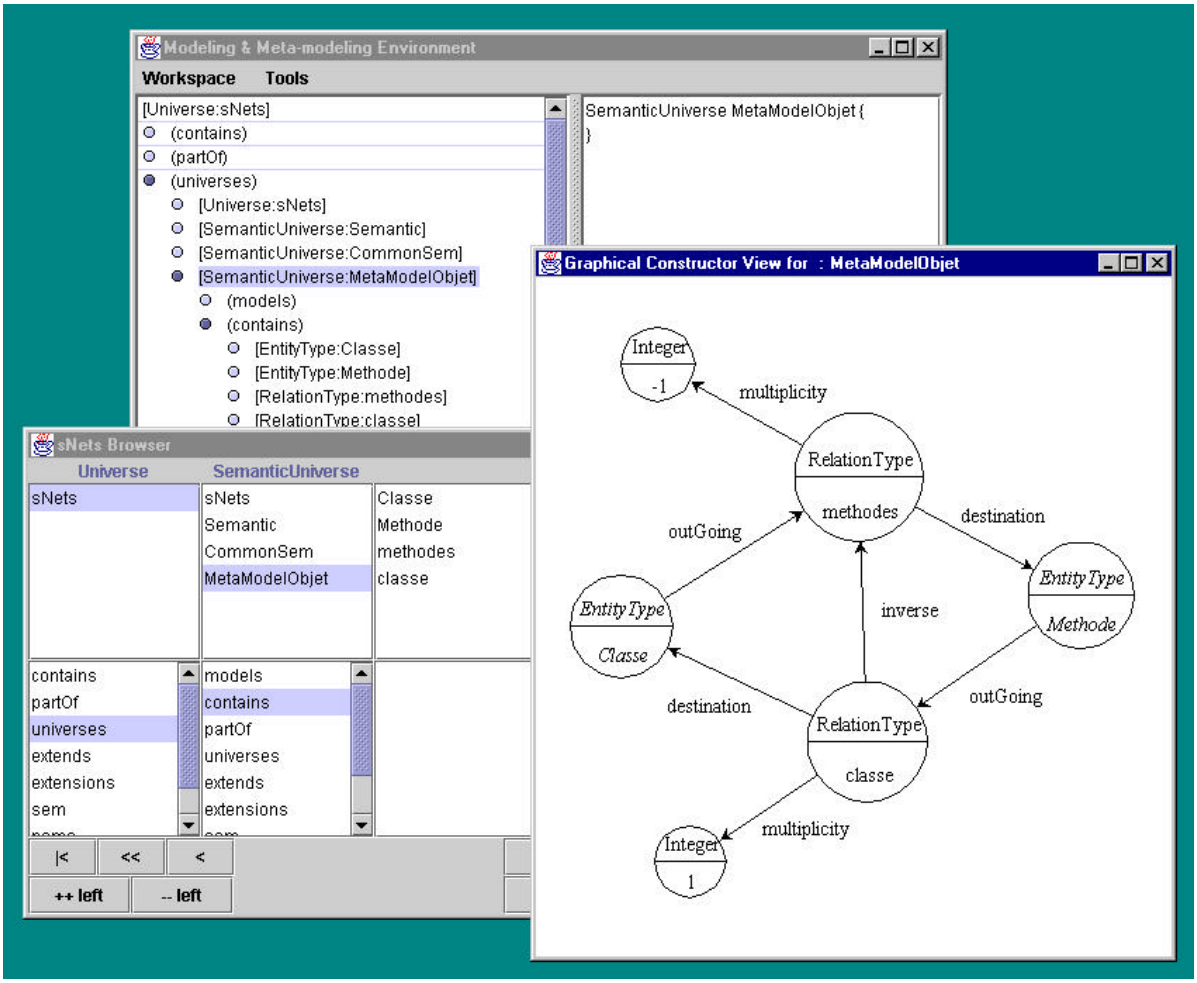


Figure 109 - L'Atelier de modélisation et de méta-modélisation utilisé pour la confection d'un méta-modèle à objets.

Ce méta-modèle peut alors être testé sans quitter l'atelier. Il suffit de créer un nouvel univers et de lui affecter comme sémantique l'univers MetaModeleObjet que l'on vient de créer. Les mêmes outils peuvent alors être utilisés pour concevoir un modèle à objets conforme à ce méta-modèle. La Figure 110 présente ainsi le navigateur utilisé pour naviguer sur le modèle à objets créé par le grapheur également représenté sur cette figure. Ce grapheur est contraint par la sémantique de l'univers en cours de conception. Ainsi, alors que dans le cas précédent, il n'était possible de créer que des entités de type EntityType et des entités de type RelationType, il n'est ici possible que de créer des entités de type Classe et de type Methode. De la même façon, seules les relations définies précédemment par des entités de type RelationType peuvent être créées sur ce modèle.

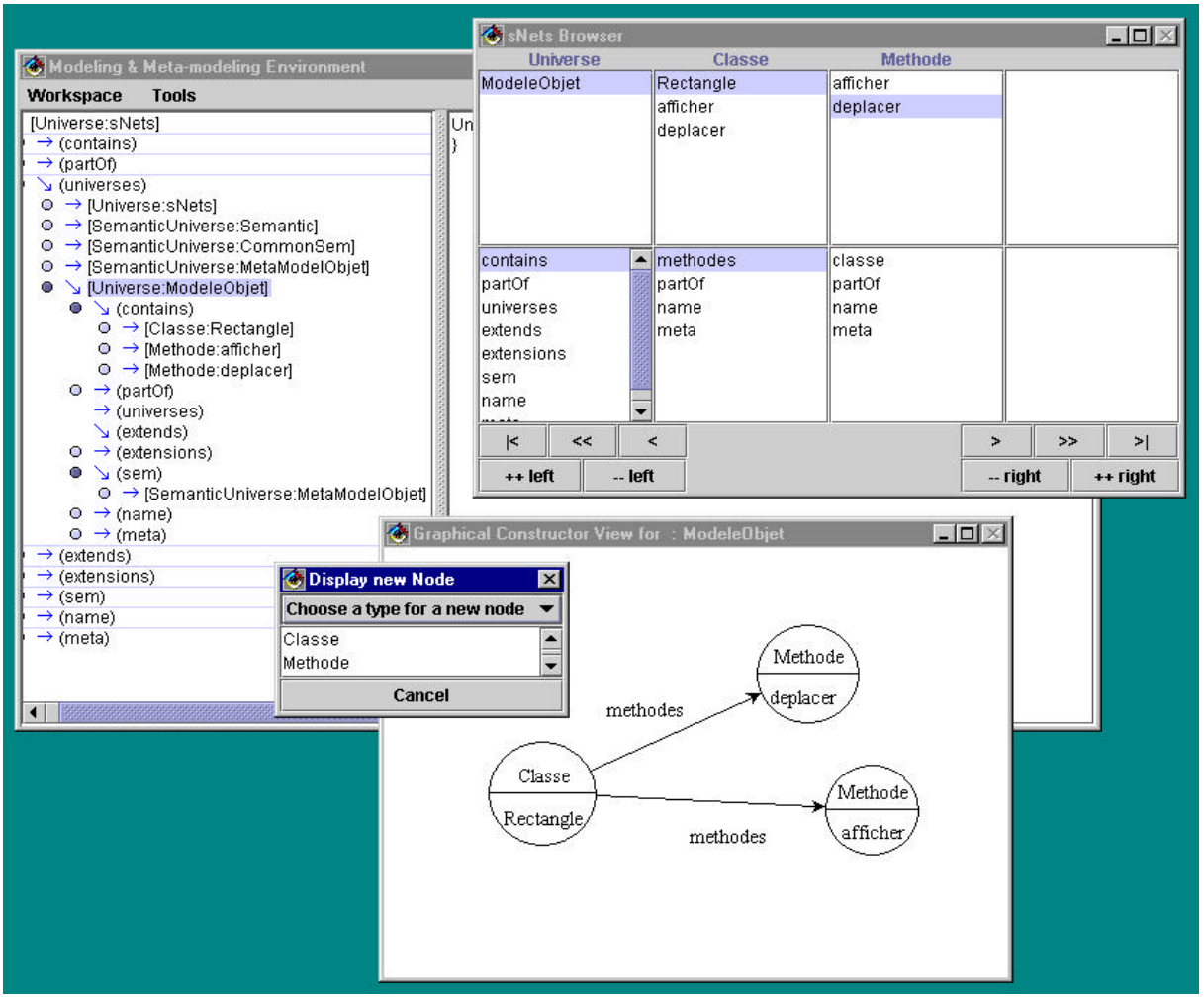


Figure 110 - L'Atelier de modélisation et de méta-modélisation utilisé pour la confection d'un modèle à objets.

Les outils d'importation et d'exportation au format XMI peuvent également être utilisés dans cet outil. Les composants de transformation de modèle et de langage de requêtes n'ont par contre pas encore été intégrés dans cet atelier.

10 Conclusions.

Face à la complexité croissante des systèmes d'information, le constat est le suivant :

- il est de plus en plus difficile d'appréhender l'univers du réel et de structurer la connaissance,
- le risque de fossé sémantique entre l'univers réel et ses modèles est de plus en plus important,
- la variété des acteurs participant à la réalisation des tâches d'une entreprise et la variété des postes de travail et des outils utilisés provoquent nécessairement une rupture sémantique tout au long de la réalisation de ces tâches.

Un formalisme tel que le formalisme des sNets, apporte un certain nombre de réponses ou tout au moins d'éléments de réponse :

- il joue le rôle de solution unifiée de représentation de modèles,
- il permet de structurer ces modèles et d'en réduire ainsi la complexité,
- il apporte une structuration modulaire de la sémantique des modèles et permet la spécialisation de ces univers sémantiques,
- il propose des facilités de navigation, de manipulation et par conséquent de réactivité pour les usagers,
- il propose également des facilités de transformation de modèles et d'élaboration de ponts pour assurer la continuité sémantique et ainsi rapprocher les différents acteurs de ces processus.

Ce travail a permis d'apporter une contribution pertinente et efficace aux problèmes rencontrés dans les métiers de l'ingénierie des systèmes d'information, de la rétro-documentation, de l'industrialisation et de la modélisation. Ces différents métiers sont au cœur des activités de la société de service dans laquelle cette thèse a été réalisée. Nous avons ainsi pu mettre en œuvre un framework de modélisation et de méta-modélisation explicites. Cet aspect explicite est

essentiel et nous a permis de lever un certain nombre d'ambiguïtés et d'incompréhensions liées à l'aspect implicite de certains éléments des formalismes tels que le MOF et CDIF.

Nous avons ainsi mis l'accent sur le rôle central de la relation globale de typage que nous appelons meta et sur le fait qu'il existe des typages locaux contextuels à des domaines de représentation spécifiques. Nous avons également montré qu'il était nécessaire de bien faire apparaître la notion de modèle au même titre que la notion de méta-modèle et que la entre un modèle et son méta-modèle (cette relation est appelée sem dans les sNets). Ces deux relations globales de typage d'entité (pour la relation meta) et de typage de modèle (pour la relation sem) étant identifiées, il est alors plus facile de les différencier de manière à ne pas les confondre et les assimiler à une seule et même relation. En effet, cette confusion était souvent réalisée avec les formalismes CDIF et MOF. Nous avons également montré qu'une fois les modèles et les méta-modèles considérés comme des entités de première classe, il était possible de définir explicitement la notion d'extension de modèle et de méta-modèle et de bien différencier cette relation de la relation sem.

10.1 Le rôle central de la relation meta.

Cette relation nous semble essentielle et n'a pourtant pas d'équivalent dans les formalismes MOF et CDIF. Elle est pourtant à l'origine de l'architecture à base de niveaux de modélisation dans laquelle ces deux formalismes s'inscrivent (c.f. Figure 111).

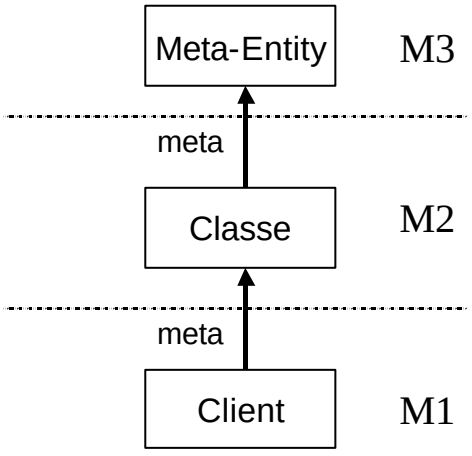


Figure 111 - La relation meta permet de séparer les niveaux de modélisation.

Sa définition explicite nous permet de lever l'ambiguïté entre son rôle de typage global et les relations de typage contextuelles telles que la relation instanceOf entre une instance d'une classe et sa classe dans un modèle à objets (c.f. Figure 112).

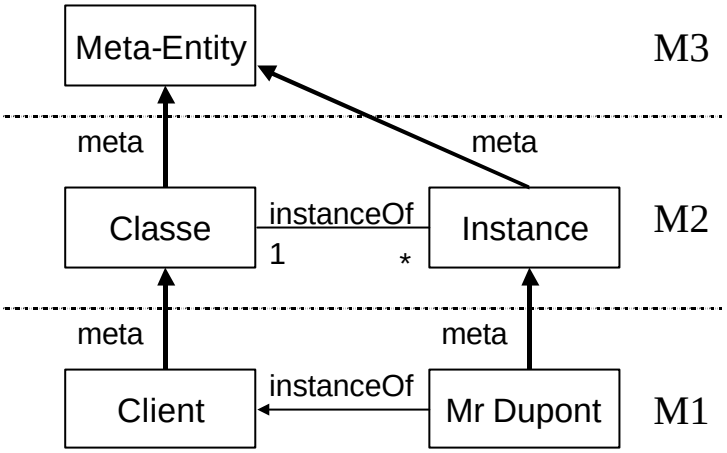


Figure 112 - La relation de typage globale meta et une relation de typage contextuel instanceOf.

Cette confusion entre une relation globale de typage et une relation contextuelle est également à l'origine de la confusion qui existe autour du nombre de niveaux de modélisation. En effet, l'assimilation de ces deux relations à une seule et même relation est à l'origine du schéma suivant dans lequel la séparation des niveaux repose sur deux relations de typage différentes : l'une globale et l'autre locale ou contextuelle (c.f. Figure 113) :

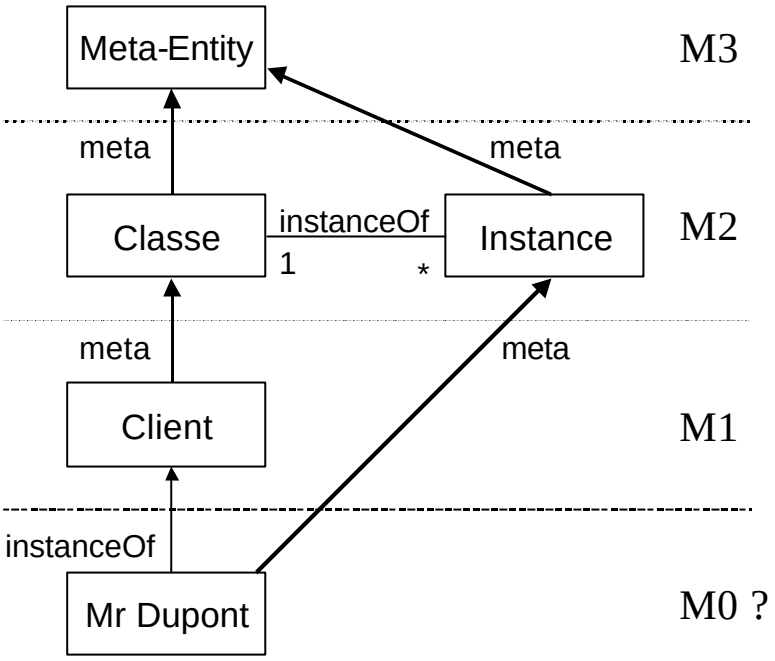


Figure 113 - Une architecture à trois ou quatre niveaux…

Il est alors important de définir la notion de modèle pour structurer correctement ces informations.

10.2 Nécessité de définir la notion de modèle.

La notion de modèle est une notion essentielle qui n'apparaît pas systématiquement dans les formalismes CDIF et MOF. C'est l'une des contributions majeurs des sNets. Dans les sNets, les modèles sont représentés par des univers. De même, les méta-modèles et le méta-méta-modèle sont représentés par des univers (c.f. Figure 114). Dans notre formalisme, nous utilisons indifféremment le terme univers à la place de celui de modèle.

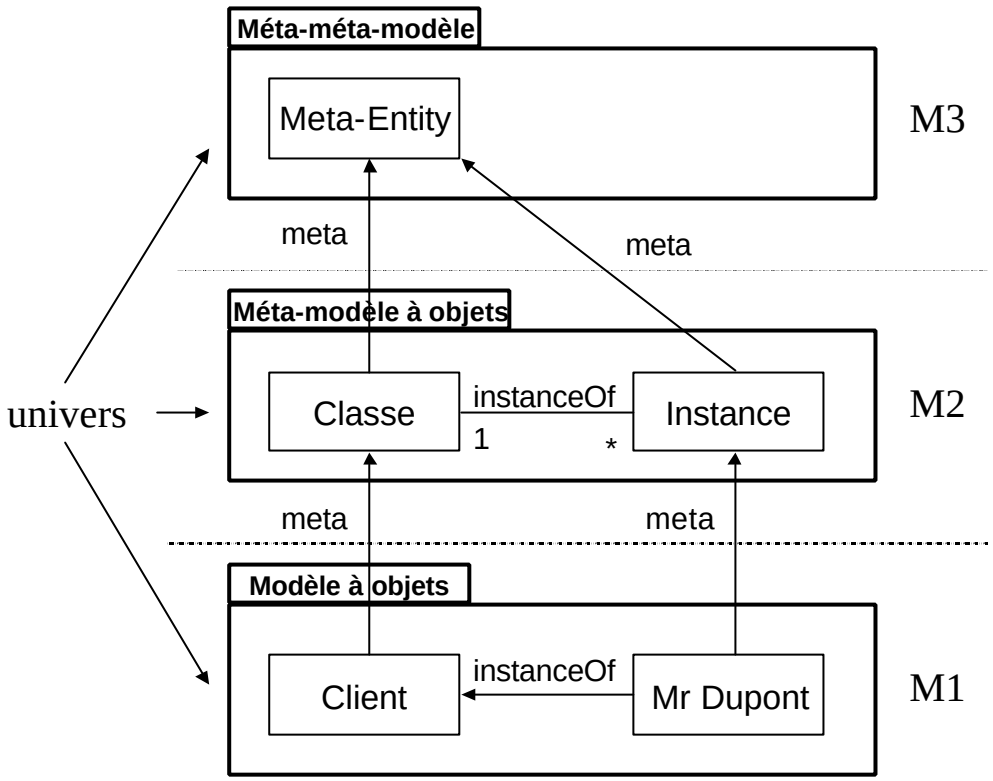


Figure 114 - Une organisation via la notion de modèles (ou d'univers).

Toute entité est alors explicitement définie dans un modèle. De plus, toute entité doit être liée explicitement à la méta-entité représentant son type et définie dans un méta-modèle. Il est alors plus facile de structurer correctement les informations dans des modèles et des méta-modèles et d'identifier le niveau de modélisation correspondant à chacun de ces modèles. De plus, la définition explicite de ces notions de modèles et de méta-modèles respectivement appelées univers et univers sémantique dans notre formalisme, permet d'imposer à tout modèle d'être défini à partir d'un méta-modèle. Il suffit pour cela de définir une relation explicite que nous avons appelé sem entre un modèle et son méta-modèle.

10.3 Apport d'une relation sem explicite entre un modèle et son méta-modèle.

La relation sem permet donc de lier explicitement un modèle à son méta-modèle (c.f. Figure 115).

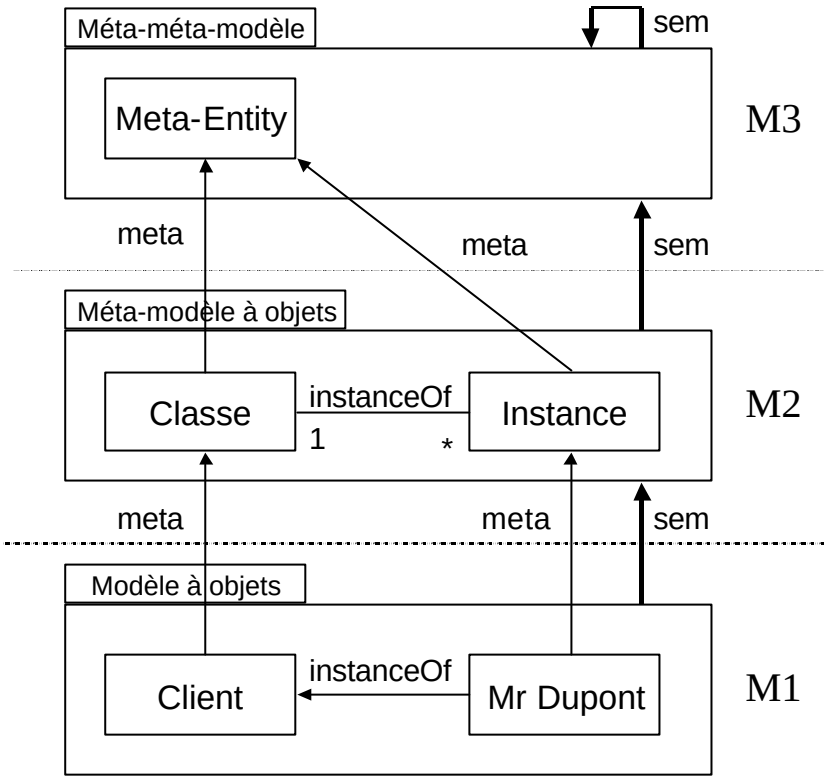


Figure 115 - La relation sem traverse les niveaux et lie explicitement les modèles à leurs méta-modèles.

Cette relation nous apporte la notion de modularité sémantique. En effet, tout modèle se doit de disposer d'un méta-modèle définissant **tous** les types et **toutes** relations qu'il utilise. Nous pouvons ainsi exprimer explicitement le fait que lorsque x, de type X est défini dans un modèle m dont le méta-modèle est M, alors X doit nécessairement être défini dans M. Dans l'exemple présenté ici, Client est une classe d'un modèle à objet dont le méta-modèle contient effectivement la définition du type Classe. De même, "Mr Dupont" est une instance de ce modèle dont le méta-modèle contient la définition du type Instance.
Mais cette règle se doit d'être valable à tous les niveaux de modélisation. Ainsi Classe est une méta-entité définie dans un modèle ("méta-modèle à objets") dont le méta-modèle (notre méta-méta-modèle) contient effectivement la définition du type Meta-Entity.

10.4 Différence entre les relations meta et sem.

La définition explicite de ces deux relations sem et meta nous permet de ne pas les confondre. En effet, par abus de langage, on dit généralement qu'un objet est instance d'une classe, qu'une entité est instance d'une méta-entité ou encore qu'un modèle est instance d'un méta-modèle. Mais dès que l'on cherche à expliciter ces relations, on se rend compte que leurs sens diffèrent. En faisant le choix de les différencier et de les nommer différemment, nous sommes à même de mieux les comprendre. Ainsi, la première relation citée ici est une relation de typage contextuelle au paradigme objet appelée instanceOf sur les figures précédentes, la seconde est globale au framework de modélisation et se nomme meta dans notre formalisme tandis que la troisième, que nous nommons sem, lie un modèle à son méta-modèle.

Ces trois relations ont donc bien un sens différent et leur définition explicite est faite dans différents niveaux de modélisation. La relation meta et la relation sem sont définies dans notre méta-meta-modèle (niveau M3). La relation meta y est définie entre le type Entité et le type Meta-Entité tandis que la relation sem y est définie entre le type Univers (notre notion de modèle) et le type Univers Semantique (notre notion de méta-modèle). La relation instanceOf est quant à elle définie dans un méta-modèle décrivant le paradigme objet et placé dans le niveau M2 . Ainsi, une fois les notions de modèle et de méta-modèle explicitement définies dans le formalisme, il est facile de définir la relation sem qui les associent.

10.5 Apport de la relation d'extension des modèles.

De la même façon que l'on a défini une relation sem entre modèles et méta-modèles, il est possible de définir une relation que nous avons appelé extends entre modèles et qui va nous permettre d'étendre de façon similaire les modèles et les méta-modèles. Cette relation est définie de la façon suivante dans notre méta-méta-modèle (c.f. Figure 116) :

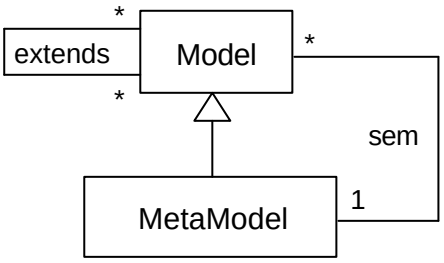


Figure 116 - Définition de la relation sem et de la relation extends dans notre formalisme.

Ces deux relations que sont donc sem et extends vont nous permettre de structurer nos modèles, de définir et de structurer leurs méta-modèles, mais également de factoriser dans des modèles et des méta-modèles plus abstraits les éléments de modélisation "réutilisables". Ces méta-modèles réutilisables tels que le méta-modèle UML ou le méta-modèle générique de processus décrits sur la figure suivante (c.f. Figure 117) vont pouvoir servir de base à des outils de manipulation des modèles qu'ils décrivent. Les spécialisations de ces méta-modèles n'engendreront alors pas de modifications dans ces outils.

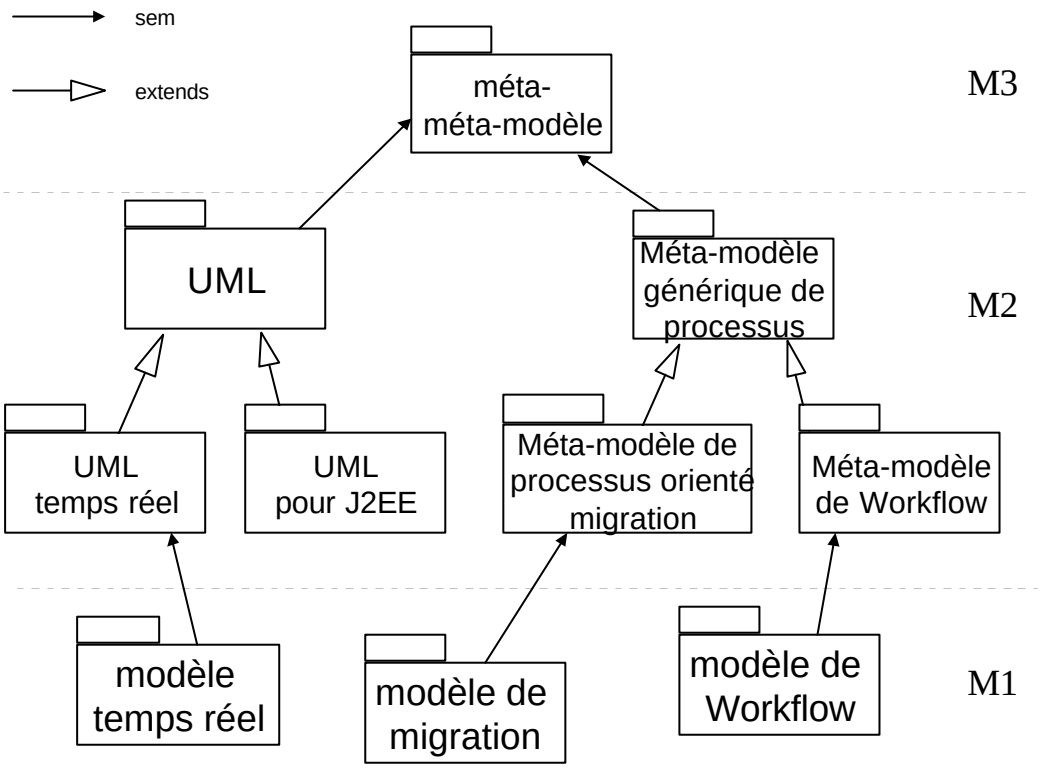


Figure 117 - Utilisation des relation sem et extends.

10.6 Evolutions et les travaux futurs.

Il y a actuellement un engouement pour la méta-modélisation, mais une pénurie de méta-modèles normalisés et par conséquent une nécessité de définir ses propres méta-modèles. Il serait alors intéressant de disposer d'outils génériques de modélisation dans lesquels ont puisse injecter le méta-modèle que l'on a ainsi défini. On peut noter qu'aucun méta-modèle n'a encore été standardisé pour supporter la modélisation des interfaces utilisateurs. L'apport d'un tel méta-

modèle permettrait la réutilisation de ce type d'information. Sur de nombreux projets, on cherche à représenter ce type de modèle dans les outils de modélisation dont on dispose (généralement des outils de modélisation ne supportant que UML) or ils ne sont pas adaptés. L'atelier de modélisation et de méta-modélisation que nous avons réalisé est un premier pas vers ce type d'outil.

Nous avons proposé un framework de modélisation et de méta-modélisation explicites qui diffère du standard actuel représenté par le MOF. Nous avons proposé un ensemble de modification du méta-modèle du MOF qui permettraient de prendre en compte ces différences. Mais nous travaillons surtout sur la manipulation, dans notre formalisme, des méta-modèles et des modèles conformes aux spécifications du MOF. Nous sommes alors à même de profiter à la fois de l'ouverture du MOF, de la puissance expressive de notre formalisme et de tous les outils que nous avons développés autour de celui-ci. Ainsi, les outils de transformation de modèles ont déjà pu être appliqués sur des modèles MOF représentés dans notre formalisme. Pour cela, il est nécessaire de définir précisément des règles de correspondance entre ces deux formalismes.

De la même façon, nous n'avons pas traité de la mise en correspondance des modèles. Celle-ci est pourtant indispensable dès qu'il s'agit de représenter un système d'information complexe constitué d'un ensemble de modèles d'analyse, de modèles de données, de modèles de processus, etc… Cette notion peut-être rapprochée des relations de co-référence que l'on retrouve dans les graphes conceptuels. Un produit décrit dans un modèle de données peut-être référencé dans un modèle de processus et l'on doit pouvoir naviguer facilement entre ces différents modèles. Ces relations vont notamment permettre de gérer la traçabilité au sein des modèles. On peut ainsi imaginer des modèles d'implémentation représentant le code source des applications et des liens de tracabilité entre les éléments de ces modèles (c.f. Figure 118).

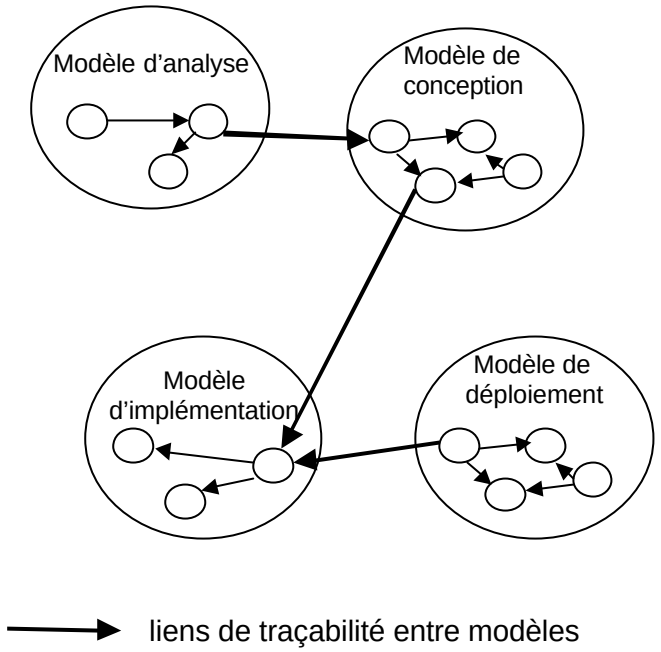


Figure 118 - Mise en correspondance de modèles au travers de liens de traçabilité.

Une telle mise en correspondance est également intéressante dans les outils de rétro-ingénierie de manière à faire le lien entre un modèle d'implémentation obtenu par analyse du code source des application et une rétro-modélisation de cette application basée sur le modèle d'implémentation obtenu.

Tous ces modèles ne sont alors plus seulement des éléments de documentation, mais vont également co-exister pendant toute la durée de vie des applications qu'ils décrivent. Il faut alors qu'ils soient en phase avec les implémentations de ces applications. Il est même probable que ces modèles deviennent alors accessible par ces applications au cours même de leur exécution. On pourra pour cela représenter tous ces modèles dans un même référentiel accessible lors de l'execution de l'application (c.f. Figure 119). Une autre possibilité est également de définir un protocole de communication standardisé entre ces différents éléments.

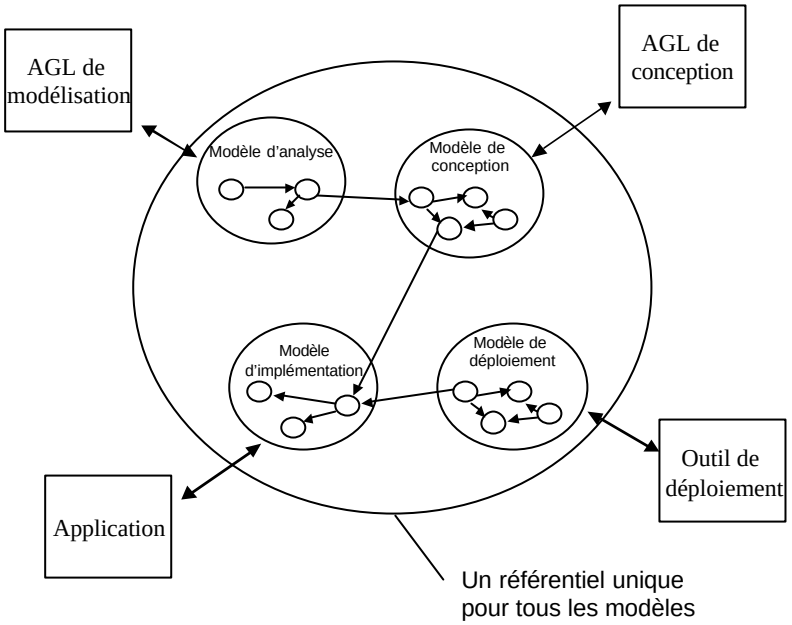


Figure 119 - Un référentiel unique pour tous les modèles.

La notion de référentiel unique pour tous les modèles permettrai également de supporter leur nombre croissant et la navigation entre les éléments qu'ils contiennent.

11 Table des Figures.

12 Bibliographie.

[1] Atkinson, C. **Metamodeling for Distributed Object Environments** in First International Enterprise Distributed Object Computing Workshop (EDOC'97), Brisbane, Australia, (1997)

[2] Atkinson, C. **Supporting and Applying the UML Conceptual Framework** in J. Bézivin & P.A. Muller (eds.), <<UML>>'98: Beyond the Notation, LNCS #1618, (1999)

[3] Baddley, A. **La mémoire humaine**. Théorie et pratique. Presses universitaires de Grenoble, (1993)

[4] Banach, R. & Papadopoulos; G.A**. A study of two graph rewriting formalisms: Interaction Nets and MONSTR**, Journal of Programming Languages, vol 05, issue 01, p. 201-231

[5] Bauderon, M. & Courcelle B. **Graph expressions and graph rewritings**, Mathematical Systems Theory, vol 20, p. 83-127

[6] Bézivin, J & Lemesle, R. **sNets : The Core Formalism for an Object-Oriented CASE Tool** COODBSE'94 Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering, World Scientific Publishers, ISBN 981-02-2170-3, pp 224-239

[7] Bézivin, J. & Lanneluc, J. & Lemesle, R. **A Kernel Representation System for OSM**. Rapport de recherche ERTO, Université de Nantes, (1994)

[8] Bézivin, J. & Lemesle, R. **Ontology-Based Layered Semantics for Precise OAD&D Modeling** ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques, Jyväskylä (Finland). LNCS Vol. 1357, Springer, 1998, ISBN 3-540-64039-8, pp 151-154

[9] Bézivin, J. & Lemesle, R. **Reflective Modeling Schemes**, OOPSLA'99 workshop on Object-Oriented Reflection and Software Engineering, Denver. pp 107-122 (proceedings)

[10] Bézivin, J. & Lemesle, R. **sBrowser : a prototype Meta-Browser for Model Engineering**, OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, Vancouver

[11] Bézivin, J. & Lemesle, R. **Towards A True Reflective Scheme** Reflection and Software Engineering, LNCS Vol. 1826, Springer, 2000, pp 21-38

[12] Bézivin, J., Ernst, J. & Pidcock, W. **Model Engineering with CDIF** OOPSLA'98, Vancouver, post-proceedings, Summary of the workshop, (October 1998)

[13] Bézivin, J., Lanneluc, J., Lemesle, R. **Representing Knowledge in the Object-Oriented Lifecycle** TOOLS PACIFIC'94, Melbourne, (December 1994), Prentice Hall, pp. 13-24

[14] Blostein, D. & Schürr, A. **Computing with Graphs and Graph Rewriting**. Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany

[15] Bobrow, D.G. & Goldstein, I.P. **Representing Design Alternatives** Proc. Conference on Artificial Intelligence and the Simulation of Behaviour, Amsterdam, July 1980

[16] Bobrow, D.G., Kiczales, G. & de Rivières, J. **The Art of the Metaobject Protocol**, The MIT Press, Cambridge, Massachusetts, (1991)

[17] Booch, G. & Rumbaugh, J. **Unified Method for Object-Oriented Development Documentation Set** Version 0.8, Rational Software Corporation, (Oct. 1995)

[18] Coad, R. & Yourdon, E. **Object Oriented Analysis** Englewood Cliffs, N.J., Prentice-Hall, (1990).

[19] Courcelle, B. **Graph rewriting : An algebraic and logic approach**, Handbook of Theorical Computer Science, vol B, p. 193-242

[20] Cox, B. **Object-Oriented Programming : An Evolutionary Approach**, Addison-Wesley, (1986)

[21] Crawley, S., Davis, S., Indulska, J., McBride, S., Raymond, K. **Meta Information Management,** Proc. Second IFIP International conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'97), Canterbury, United Kingdom, 21st-23rd July, 1997

[22] Desfray, P. **Automation of design pattern : concepts, tools and practices**, UML'98 International Workshop, (June 1998)

[23] Electronic Industries Association/CDIF Technical Committee Electronic Industries Association/CDIF Technical Committee, **Transfer Format - General Rules for Syntaxes**, Interim Standard EIA/IS-108 (1994),

[24] Electronic Industries Association/CDIF Technical Committee, **Framework for Modeling and Extensibility**, Interim Standard EIA/IS-107 (1994),

[25] Electronic Industries Association/CDIF Technical Committee, **Integrated Meta-model - Foundation Subject Area**, Interim Standard EIA/IS-111 (1994),

[26] Electronic Industries Association/CDIF Technical Committee, **Integrated Meta-model - Common Subject Area**, Interim Standard EIA/IS-112 (1996),

[27] Electronic Industries Association/CDIF Technical Committee, **Transfer Format - Transfer Format Syntax - SYNTAX.1**, Interim Standard EIA/IS-109 (1994),

[28] Electronic Industries Association/CDIF Technical Committee, **Transfer Format - Transfer Format Encoding - ENCODING.1**, Interim Standard EIA/IS-110 (1994),

[29] Ernst, J. **Introduction to CDIF**, Integrated Systems, Inc., (Jan. 1997),

[30] Ernst, J. **CDIF - Technical Introduction**, Integrated Systems, Inc., (Juin 1997),

[31] Evans A.S. (moderator), Cook, S., Mellor, S., Warmer,J., Wills, A. **Panel Paper - Advanced Methods and Tools for a Precise UML.** 2nd International Conference on the

Unified Modeling Language. editors: B.Rumpe and R.B.France, Colorado, LNCS, 1999. http://www.cs.york.ac.uk/puml/

[32] Genesereth, M.R. & Fikes, R.E. **KIF: Knowledge Interchange Format** Version 3.0 Reference Manual TR Logic-92-1, Computer Science Department, Stanford University, (1992)

[33] Genesereth, M.R. & Fikes, R.E**. Knowledge Intechange Format version 3.0 Reference Manual** T.R. Logic-92-1, Computer Sciences Department, Stanford University, (1992)

[34] Goldberg, A. & Robson, D. **Smalltalk-80 : The language and its Implementation**, Addison-Wesley, (1983)

[35] Greenspan, S. & Mylopoulos, J. & Borgida, A. **On Formal Requirements Modeling Languages: RML Revisited** Invited Plenary Talk, ICSE 94, (1994)

[36] Gruber, T.G. **A Translation Approach to Portable Ontology Specifications** Knowledge Acquisition, V.5, N.2, (1993)

[37] Habel, A. **Hyperedge replacement : grammars and languages**. *LNCS*, vol 643

[38] Henderson-Sellers, B. & Edwards, J.M. **The Object-Oriented Life Cycle** CACM, V.33, N.9, (September 1990), pp. 142-159

[39] Jackson, M.A., **System Development** Prentice Hall International, (1983)

[40] Kayser, D. **Ontologically Yours** Invited Talk, ICCS'98, Montpellier, France, LNAI #1453, M.L. Mugnier & M. Chein eds, (August 1998)

[41] Keene, S**. Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS**, Addison-Wesley, (1989)

[42] Kikzales, G. Reflexion'96, San Francisco, (April 21-23, 1996)

[43] Kilov, H. James Ross. **Information Modeling: an Object-oriented Approach**. Prentice-Hall, Englewood Cliffs, NJ, (1994)

[44] Kobryn, C. **UML 2001: A Standardization Odyssey** CACM, October 1999, V.42, N.10, pp.29-37

[45] Lamb, D.A. Editor, **Studies of Software Design**, Lecture Notes in Computer Science, Volume 1078, Springer Verlag, (1996)

[46] Lemesle, R **Meta-modeling and Modularity : Comparison between MOF, CDIF and sNets formalisms**, OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, Vancouver

[47] Lemesle, R. **Transformation Rules based on Meta-modeling**, EDOC'98 (Enterprise Distributed Object Computing), San Diego

[48] Lemesle, R. **Un réseau sémantique au cœur d'un AGL**, mémoire de DEA. Université de Nantes.

[49] Lindsay & Norman, **Traitement de l'information et comportement humain**, Ed. Etudes Vivantes, (1980)

[50] Maes, P., Nardi, D. (eds), **Meta-Architectures and Reflection**, North-Holland, 1988

[51] Meyer, B. **EIFFEL: The Language**, Prentice Hall, (1991)

[52] Microsoft, Microsoft Repository Product Information, **Open Information Model Overview**, 1999

[53] Odell, J. **Meta-modeling**, (1995). OOPSLA'95 Workshop, (Oct.1995)

[54] OMG/Action **Action Semantics for the UML**, Request for Proposal, OMG Document ad/98-11-01, (November 1998)

[55] OMG/BOM **Workflow Management Facility Specification** OMG Document bom/98-01-11, [1998]

[56] OMG/Corba **The Common Object Request Broker: Architecture and Specification**, Revision 2.0, Object Management Group, April 1995

[57] OMG/CWMI **Common Warehouse Metadata Interchange Request For Proposal**, OMG Document ad/98-09-02, [September 18, 1998]

[58] OMG/MOF **Meta Object Facility (MOF) Specification**. AD/97-08-14, Object Management Group, Framingham, Mass., (September 1997)

[59] OMG/SPE Analysis and Design PTF**, Software Process Engineering Request for Information**, Version 1.0, OMG Document ad/98-10-08, (13 November 1998)

[60] OMG/**UML Specification. Version 1.1**, Rational Software (September 1997)

[61] OMG/**UML Specification. Version 1.3R9**, Rational Software (January 1999)

[62] OMG/**UML Unified Modeling Language UML Notation Guide**. AD/97-08-05, Object Management Group, Framingham, Mass., (November 1997)

[63] OMG/XMI **XML MetaData Interchange (XMI) Proposal to the OMG OA&DTF RFP3 : Stream Based Model Interchange Format (SMIF)** Document ad/98-10-05, (October 20, 1998), Adopted at the Washington Meeting, (January 1999)

[64] Raymond, K. **Meta-Meta is Better-Better**. IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (September/October 1997)

[65] Ross, D. **Structured Analysis (SA) : A Language for Communicating Ideas** in Tutorial on Design Techniques Freeman & Wasserman Eds. IEEE Computer Society Press, Long Beach, CA, (1980), pp. 107-125

[66] Rumbaugh, J. & Blaha, M. & Premerlani, W. & Eddy, F. & Lorensen, W. **Object-Oriented Modeling and Design** Prentice Hall, (1991)

[67] Schneider, P.F. **Contexts in PSN** in Topics in PSN-II, TR-CSRG-112, (April 1980), University of Toronto

[68] Schûrr, **A. Programmed Graph Replacement Systems**, G. Rozenberg, Handbook on Graph Grammars: Foundations, Vol. 1, Singapure: World Scientific (1997), p 479-546

[69] Sowa, J. **Conceptual Structures : Information Processing in Mind and Machine**, Addison-Wesley, (1984)

[70] Sowa, J. **Knowledge Representation : Logical, Philosophical and Computational Foundations**, Brooks Cole, (2000)

[71] Stephen Crawley **Type Management using Type Graphs**, in Proc. DSTC Symposium '96, Brisbane. 11-12 July 1996

[72] Szyperski, C. **Component Software : Beyond Object-Oriented Programming** Addison Wesley, (1998)

[73] Tardiveau, M. **The Meta Object Facility: The Final Frontier of Modeling**, JOOP, June 1999, pp. 8-11

[74] UML Specification. Version 1.3R9, Rational Software (January 1999)

[75] Warmer, J., & Kleppe, A. **The Object Constraint Language : Precise Modeling with UML** Addison Wesley, (October 1998)

[76] World Wide Web Consortium (W3C), **Extensible Markup Language (XML) 1.0**, (1998)

[77] Yiengar, S. **The Universal Repository UREP** (web pages), UNISYS. http://www.unisys.com/marketplace/products/urep/