Université de Montréal

Modélisation pour la gestion de modèles

par

Thi Lan Anh DINH

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures

en vue de l'obtention du grade de

Philosophiae Doctor (Ph.D.)

en Informatique

Avril, 2007

Université de Montréal

Faculté des études supérieures

Cette thèse intitulée:

Modélisation pour la gestion de modèles

présentée par

Thi Lan Anh DINH

a été évaluée par un jury composé des personnes suivantes:

Stefan Monnier
(président du jury)

Mostapha Aboulhamid
(membre du jury)

Houari Sahraoui
(directeur de recherche)

Olivier Gerbé
(codirecteur de recherche)

Jean Bézivin
(examinateur externe)

Olivier Gerbé
(représentant du doyen)

Thèse acceptée le 28 août 2007

Sommaire

Comme le montrent les nombreux travaux de recherche dans la communauté informatique, la gestion de modèles prend une importance grandissante et intervient dans des domaines aussi divers que la gestion des connaissances, les bases de données, les ontologies, la qualité de service et le génie logiciel. Les chercheurs dans ces domaines travaillent beaucoup sur la modélisation et la métamodélisation. Un défi fondamental pour la gestion de modèles est la représentation des modèles. Par ailleurs, le développement des opérations de manipulations de modèles est un autre défi important.

Nous avons étudiés les formalismes les plus connus pour la représentation des connaissances et la représentation de modèles. Ces formalismes sont utilisés dans les domaines tels que bases de données, intelligence artificielle, génie logiciel et Web sémantique. Face aux besoins pour la définition et la manipulation de modèles, cette étude nous a montré que les formalismes existants ne pouvaient couvrir tous les besoins. Par exemple, les noyaux réflexifs des sNets, MOF, CDIF, de XML Schema, de RDF/RDFS/OWL sont limités en pouvoir d'expression et ne permettent donc pas d'exprimer tous les métamodèles. Les formalismes tels que UML, Entité-Association, les graphes conceptuels, XML Schema, RDF/RDFS, OWL sont aussi limités en pouvoir d'expression et ne permettent pas de modéliser toutes les situations du monde réel.

Ces constats nous ont poussés à développer et proposer un nouveau formalisme en réponse aux besoins de la représentation de modèles et ce à tous les niveaux de modélisation. Simple, puissant et flexible, ce formalisme est basé sur les réseaux sémantiques, et inspiré des GCs et des sNets. À partir de ce formalisme, nous avons développé des opérations de manipulation de modèles. Ce formalisme et ces opérations constituent un premier pas vers la réalisation d'un système de gestion des modèles. Notre formalisme a été validé d'une manière théorique en réponse à nos besoins pour la représentation de modèles. La validation pratique a démontré comment le méta-métamodèle de notre formalisme permettait la mise en œuvre des différents formalismes de modélisation. La validation pratique a également démontré comment notre formalisme permettait la modélisation de situations bien particulières qui peuvent exister dans le monde réel.

Mots clés: Modélisation, Méta-modélisation, Modèle, Métamodèle, Méta-métamodèle, Représentation de connaissances, Représentation de modèles, Opérations de manipulation de modèles.

Summary

Several works of research within the computer science community have shown that model management is becoming increasingly important and is implicated in a wide range of areas such as knowledge management, databases, ontologies, quality of service, and software engineering. Researchers within these areas are focusing particularly on modeling and metamodeling. A fundamental challenge for model management is model representation. Another important challenge is developing model operations.

We studied the most popular formalisms for knowledge representation and model representation. These formalisms are used mostly in the areas of databases, artificial intelligence, software engineering, and semantic web. This study showed that the existing formalisms do not cover all of the requirements for defining and manipulating models. For instance, the reflexive cores of sNets, MOF, CDIF, XML Schema, RDF/RDFS/OWL are limited to expression power and thus do not allow for the definition of all metamodels. Formalisms such as UML, Entity-Association, conceptual graphs, XML Schema, RDF/RDFS, and OWL are also limited by expression power and do not allow for modeling of real life situations.

These findings pushed us to propose and develop a new formalism in response to the requirements for representing models at all modeling levels. Simple, powerful, and flexible, this formalism is based on semantic networks and inspires GCs and sNets. Based on this formalism, we also developed model operations, which, along with the formalism itself, constitute a first step towards the implementation of a model management system. Finally, our formalism has been validated both theoretically, in response to our requirements for model representation, and practically. Practical validation demonstrated not only how meta-metamodel of our formalism enables the implementation of different modeling formalisms, but also the ability of our formalism to model particular situations that may exist in real life.

Keywords: Modeling, Meta-modeling, Model, Metamodel, Meta-metamodel, Knowledge representation, Model representation, Model operations.

Table des matières

Liste des Figures

Liste des Tables

Remerciements

Je tiens à exprimer avant tout ma profonde reconnaissance à monsieur Olivier Gerbé, professeur à HEC Montréal, qui avait accepté d'abord de me diriger dans un stage de fin d'études de maîtrise et m'a soutenue par la suite tout au long de cette thèse. Je tiens à exprimer également toute ma gratitude à monsieur Houari Sahraoui, professeur à l'université de Montréal, qui, conjointement avec monsieur Olivier Gerbé, m'a guidée dans mes travaux de recherche. Tous deux ont su user de leurs connaissances et compétences pour m'encadrer et me donner des renseignements concrets et des conseils précieux. Sans eux, cette thèse n'aurait pas vu le jour.

Je remercie vivement monsieur Stefan Monnier pour avoir accepté de présider le jury de cette thèse ainsi que monsieur Mostapha Aboulhamid pour avoir accepté de faire parti de mon jury. Mes remerciements vont également à monsieur Jean Bézivin pour m'avoir fait l'honneur d'accepter d'être l'examinateur externe de mon jury.

Je voudrais aussi remercier les personnels de HEC Montréal, ceux du département d'informatique et de recherche opérationnelle (DIRO) ainsi que mes collègues dans le laboratoire de génie logiciel (GEODES) de l'université de Montréal, qui se sont toujours montrés très proches. Mes remerciements vont également à mes amis pour m'avoir beaucoup aidée et pour leurs sourires.

Finalement, je remercie de tout mon cœur mes parents, mon petit frère, et mon amour qui sont toujours à mes côtés pour m'apporter du soutien. Merci pour votre compréhension, votre support et tant d'autres choses.

À tous, je vous dis :

Merci !

Chapitre 1. Introduction

La notion de modèles fait l'objet de discussions depuis l'antiquité[1]. Cependant, dans la communauté informatique, l'émergence d'une vision de développement du logiciel appelé *Model Driven Engineering* - Ingénierie Dirigée par les Modèles ou IDM - a relancé l'intérêt en mettant l'accent sur les activités de modélisation et de gestion de modèles. La gestion de modèles traite des mécanismes qui permettent de représenter, créer, stocker et manipuler les modèles. Elle intervient dans les domaines aussi divers que le génie logiciel, les bases de données, la qualité de service, les ontologies, et la gestion des connaissances. Nous allons d'abord faire un survol des domaines mentionnés ci-haut afin de comprendre le contexte et la portée de notre travail de recherche.

Ingénierie Dirigée par les Modèles

Dans le domaine du développement de logiciels, la naissance de *Model-Driven Architecture* (MDA) représente l'approche orientée-modèles plutôt que orientée-objets [104]. Ceci vise le pouvoir d'abstraction, de raffinement et de vues différentes sur des modèles. Et surtout, il donne la possibilité de concevoir des modèles indépendants des plates-formes et de l'environnement d'implémentation. Tout récemment, succédant au MDA, l'approche *Ingénierie Dirigée par les Modèles* ([17], [56], [71]) a pour objectif de définir un cadre pour la génération de code par des transformations successives de modèles. Cette approche a relancé l'intérêt en insistant sur les activités de modélisation et de gestion de modèles.

Bases de données

Lors de la réalisation des applications de bases de données ou des applications Web, plusieurs problèmes tels que l'intégration de schémas, la transformation de schémas, l'évolution d'un schéma, la traduction de données, etc. impliquent le traitement sur la structure des données plutôt que sur les données elles-mêmes. Afin de résoudre les problèmes d'intégration et de traduction de données, l'approche basée sur la gestion des modèles devient une piste de solution [12]. Les aspects structurel et sémantique de certaines fonctions telles que la création, la mise-à-jour, la suppression, la copie, la projection, etc., pour la gestion des modèles sont décrits pour la première fois comme

[1] http://www.muellerscience.com/ENGLISH/Theconceptofmodel.history.htm. The Concept of Model and its Triple History.

une vue globale dans [18]. Les défis sont aussi exposés. Un défi fondamental est le développement d'un mécanisme pour représenter les modèles et pour stocker les représentations des modèles [18]. À propos des problèmes de l'intégration et de la transformation des schémas, une théorie de modèles a été proposée pour la gestion des schémas génériques [3].

Qualité de service

En terme général, la qualité de service désigne la capacité de satisfaire aux exigences des clients. La gestion de la qualité de service pour une demande d'un usager vise à décider quand et comment la réponse est délivrée à cet usager dans un délai raisonnable en respectant le coût ou les exigences donnés. En particulier, dans le domaine de réseaux et systèmes multimédia, la qualité de service implique l'ensemble des paramètres d'un système multimédia réparti; cet ensemble de paramètres influe sur la présentation des données multimédia à l'usager ainsi que sur la satisfaction générale de l'usager envers l'application [163]. Dépendamment des applications, les paramètres sont liés à différentes informations et classifiés en plusieurs catégories telles que la performance, la fiabilité, la sécurité [19] des systèmes ou au niveau du réseau ou encore de l'application [95]. La qualité de service est évaluée non seulement au niveau de la communication mais aussi par l'usager qui décrit ses exigences. La gestion de la qualité de service peut alors être regardée comme une fonctionnalité qui doit être intégrée dans le système multimédia ([5], [102]). Afin d'assurer un fonctionnement adéquat des applications dans un environnement complexe tel que les systèmes multimédias répartis, la modélisation de l'information de gestion devient fondamentale [72].

Ontologies

Dans le domaine des ontologies, le terme «ontologie» récemment utilisé dans l'informatique est emprunté du domaine de la philosophie; sa notion est largement discutée (ex.: [25], [33], [64]). En mettant l'accent sur le développement des ontologies pour les systèmes d'information, nous acceptons la définition de Borist [20], modifiée de celle de Gruber [62], qui précise qu'*une ontologie est une spécification formelle d'une conceptualisation commune*. Une ontologie définit de façon formelle un vocabulaire qui contient des termes pour représenter des concepts dans un domaine et des relations entre ces concepts. Ce vocabulaire est utilisé par les usagers afin de partager des connaissances. Voici un exemple d'une ontologie relative à une organisation. Les

concepts peuvent être «Personne», «Statut», «Organisation», «Département», etc.; la relation «travailler-dans» peut être définie entre «Personne», «Statut», «Organisation» pour indiquer qu'une personne (une instance de «Personne») peut travailler dans une organisation (une instance de «Organisation») avec un ou des statuts (instances de «Statut») comme directeur, ou secrétaire, chercheur, etc.; «Département» est un sous-type de «Organisation»; la relation «faire-partie» peut lier «Département» à «Organisation» pour illustrer qu'un département (une instance de «Département») peut faire partie d'une organisation. Les bénéfices des ontologies sont multiples: la traduction de la sémantique entre différents lexiques, la réutilisation des connaissances d'un domaine à un autre (afin d'éviter la redondance de connaissances), la réduction de l'ambiguïté sémantique, etc. Cette notion est importante pour plusieurs domaines tels que les applications Web sémantique [10], la gestion des connaissances, la modélisation des entreprises ([25], [51]). La représentation et la modélisation des ontologies sont évidemment les questions primordiales dans ce domaine. Notons également qu'une ontologie peut évoluer (exemples: la suppression de termes inutilisés, la modification de la définition d'un terme, le rajout d'un nouveau terme). Il faut donc avoir des opérations pour la gestion des ontologies.

Gestion des connaissances (dans les entreprises)

La gestion des connaissances est depuis le milieu des années 90 devenue une discipline de gestion importante. Un historique nous est présenté dans [7]. En se basant sur différentes définitions existantes de la gestion des connaissances ([7], [73], [100]), et également sur la distinction entre les notions de données, d'information, de connaissance et de sagesse élaborée dans ([9], [100]), nous pouvons définir les termes «connaissance» et «gestion des connaissances» comme suit :

- *La connaissance* est représentée par des objets et modèles qui visent tous ensemble à représenter des objets concrets ou abstraits de l'univers du discours en décrivant divers comportements et propriétés de ces derniers dans un domaine, identifiables et traitables par les systèmes informatiques.
- *La gestion des connaissances* consiste à modéliser, acquérir, entreposer et disséminer la connaissance acquise afin d'exploiter facilement et efficacement la base de connaissances existante et de possiblement produire de nouvelles connaissances.

Ainsi, dans un traitement sémantique, les connaissances nécessitent d'abord d'être modélisées et donc représentées sous forme de modèles. L'intérêt sémantique de la modélisation prime sur l'intérêt d'opérationnalisation. Ce qui est recherché avant tout, c'est la compréhension du monde à modéliser et une représentation ou documentation qui allie précision et souplesse.

La gestion des connaissances est devenue importante dans les entreprises (ex: [66]) et on y parle souvent de notion de mémoire d'entreprise.

Il existe plusieurs notions de mémoire d'entreprise (ex.:[2], [55], [155]). Dans son acceptation courante, le terme «mémoire d'entreprise» (ou «mémoire corporative», «connaissances corporatives», «connaissances de l'entreprise») désigne *l'ensemble des savoirs et savoir-faire de l'entreprise tels que ses processus d'affaires, ses procédures, ses politiques (mission, règlements, normes) et ses données (ventes, achats, informations sur les employés, etc.)* ([2], [55]). «Mémoire d'organisation» est un terme plus générique indiquant que cette notion de mémoire peut s'appliquer à une organisation quelconque (par exemple: une entreprise, un département ou un service au sein de l'entreprise, ou bien un projet, etc.) [2].

Utiles pour déterminer les connaissances essentielles à capitaliser, plusieurs topologies des connaissances dans l'entreprise ont été proposées ([2] , [35], [55], [155]). Ces connaissances peuvent être classifiées en se basant sur divers aspects tels que leur répartition géographique ou administrative, leur usage, leur traitement, ou l'acquisition et la diffusion de ces connaissances. Par exemple, selon la répartition géographique ou administrative, nous avons la mémoire interne et la mémoire externe [2]; la mémoire métier, la mémoire société, la mémoire de projet et la mémoire individuelle; ou la mémoire technique, la mémoire organisationnelle, la mémoire de projet. Et selon l'usage de connaissances, nous avons la mémoire critique et la mémoire stratégique [61]. Dans les traitements possibles des connaissances, [61] et [63] distinguent les éléments tangibles (données, procédures, plans, modèles, algorithmes, documents d'analyse ou de synthèse) et intangibles (capacités, talents professionnels, connaissances privées, connaissances sur l'histoire de l'entreprise et les contextes de décision, etc.). Lors d'une opération de capitalisation, les éléments tangibles peuvent être vérifiés pour la capitalisation (gestion des données techniques, gestion de documents, gestion de configuration), alors que les éléments intangibles requièrent la formalisation de savoir-

faire (acquisition et représentation de savoir-faire et de raisonnement sur un tel savoir-faire). Nous trouvons aussi les connaissances tacites difficiles à formaliser et les connaissances explicites [103]. Récemment, Gerbé différencie au sein des connaissances, la partie statique et la partie dynamique [55]. La partie statique concerne l'aspect structure de la connaisance et définit les concepts de base et les relations pouvant les associer. Quant à l'aspect dynamique, il concerne l'aspect comportemental de la connaisance et définit les concepts utilisés pour représenter la dynamique des objets de connaissances (par exemple la représentation et description des processus d'affaires dans le cas d'une mémoire d'entreprise). Cette classification aide à sortir des types d'éléments principaux utiles à la représentation de connaissances et offre donc, selon nous, une vue conceptuelle claire pour les modélisateurs de connaissances.

La gestion des connaissances dans les entreprises, c'est-à-dire le développement de mémoires corporatives, pose principalement un problème de quantité, de complexité et de diversité ([38], [55]). Ceci implique un défi pour la modélisation de ces mémoires et présente également un défi pour la gestion de modèles s'y appliquant.

Problème et approche

Dans l'aspect de la gestion de modèles dans les domaines présentés précédemment (le génie logiciel, la gestion de données et métadonnées, la qualité de service, les ontologies, et la gestion des connaissances), nous constatons un défi commun: la représentation de modèles et le développement d'opérations pour la manipulation de modèles. Notre recherche s'inscrit dans le cadre de la gestion des modèles et se concentre sur les deux axes de recherche suivants: (i) la représentation de modèles, et (ii) les opérations pour manipuler les modèles. Notre travail repose sur les quatre principaux objectifs présentés ci-dessous:

(1) Trouver une architecture de modélisation dans laquelle s'inscrire;
(2) Étudier les formalismes existants pour la représentation de connaissances afin de les utiliser pour la représentation des modèles (y compris la représentation, la modélisation et la méta-modélisation des connaissances);
(3) Étudier des opérations permettant de manipuler les modèles;
(4) Proposer un formalisme adapté à la gestion de connaissances.

Concernant le premier objectif, la modélisation, la méta-modélisation et l'architecture

de modélisation seront discutées dans la section 2.2 (page 14).

Plusieurs formalismes peuvent être utilisés pour représenter des modèles, par exemple: les graphes conceptuels [148], sNets [80], UML (Unified Modeling Language) (ex.: [109], [110], [113], [114], [141], [142]) et MOF (Meta Object Facility) ([105], [106], [107], [108]), XML (eXtensible Markup Language) ([165], [168]) et XML Schema ([166], [167], [169], [170], [171], [172]), RDF (Resource description Framework) et RDF Schema ([175], [176], [177], [178], [179]), et OWL (Web Ontology Language) ([180], [181], [182], [183]). Cependant, nos études sur les formalismes dans la section 2.4 (page 25) et le Chapitre 4 (page 43) démontrent que ces formalismes ne permettent pas de répondre à tous les besoins essentiels pour la représentation et la gestion de modèles dans le cadre de la gestion de connaissances. De plus, il est à noter que l'utilisation d'un ensemble de modèles avec différents formalismes pose souvent des problèmes de traduction et de pertes d'information lors des échanges entre modèles, bien que ces modèles soient compatibles.

L'objectif le plus important de cette thèse, l'objectif (4), consiste donc à définir un nouveau formalisme qui sera susceptible de répondre à tous les besoins de la gestion de modèles dans le cadre de la gestion de connaissances. Ce formalisme est principalement basé sur les réseaux sémantiques [146], souhaitant maintenir la flexibilité des réseaux sémantiques dans l'expression ainsi que la simplicité dans la représentation afin d'en faciliter la mise en œuvre. Les types de connaissances à traiter, ainsi que leurs caractéristiques, varient énormément selon les problèmes concrets à résoudre. Face à cette situation, l'extensibilité du modèle proposé est primordiale, plus précisément, en ce qui concerne le méta-métamodèle et le métamodèle proposés.

Concernant le troisième objectif, nous avons fait une revue des opérations nécessaires permettant la manipulation des modèles (cf. la section 2.3, page 17) et nous avons traité celles correspondant à nos besoins (cf. le Chapitre 7, page 194).

Les quatre premiers objectifs sont suffisants pour mener à bien notre recherche dans le cadre de cette thèse. Notre formalisme proposé et les opérations spécifiées sur modèles constituent un premier pas vers la réalisation d'un système de gestion des modèles. L'atteinte du cinquième objectif achèverait la réalisation de ce système:

(5) Développer un prototype du système qui implémente le formalisme proposé.

Contributions majeures de la thèse

Les contributions majeures de cette thèse sont:

- Nous avons identifié les besoins essentiels concernant la représentation des métamodèles et ceux concernant la représentation de modèles de connaissances du monde réel (cf. Chapitre 3). Du point de vue modélisation, ceci nous permet de mieux comprendre l'univers à modéliser, d'identifier plus facilement les éléments pour chaque niveau de modélisation et d'aboutir à une représentation la plus précise mais aussi la plus souple possible.
- Nous avons proposé un nouveau formalisme qui remplit tous les besoins recensés pour la gestion de modèles. Nous avons montré que par rapport aux autres noyaux réflexifs étudiés, seul le noyau réflexif de notre formalisme permet de définir tous les formalismes de modélisation. Également, en comparaison avec d'autres formalismes disponibles pour la modélisation du monde réel que nous avons étudiés, notre formalisme permet de représenter d'une manière la plus fidèle les situations du monde réel, et il est le seul permettant de modéliser les situations telles que celles décrites dans le Problème 1 (page 34) et le Problème 2 (page 37). Il est à souligner que notre formalisme répond au critère de l'extensibilité, c'est-à-dire, le méta-métamodèle et le métamodèle proposés sont tous extensibles avec la possibilité de définir et d'ajouter de nouveaux éléments au besoin en se basant sur notre noyau réflexif.
- Nous avons spécifié les opérations nécessaires pour manipuler les modèles dans notre base de modèles. Ces opérations et notre formalisme constituent une base solide pour le développement d'un système de gestion de modèles. Ce système peut fonctionner avec une mise en œuvre relativement simple au sein d'une base de données relationnelles
- Quoique nos études de cas se situent plutôt dans la gestion des connaissances, nos résultats de recherche sont applicables à plusieurs domaines importants tels que les bases de données, les ontologies, la qualité de service et le génie logiciel.

Organisation de la thèse

Le reste du présent document est structuré comme suit. Le deuxième chapitre expose les travaux touchant les problèmes liés à nos intérêts de recherche. Il présente l'état de

l'art concernant les différents formalismes pour la représentation de modèles ainsi que les opérations de manipulation des modèles. Le troisième chapitre identifie nos besoins concernant la représentation de modèles ainsi que les opérations sur modèles. Dans le quatrième chapitre, sont analysés les formalismes de modélisation principaux vis-à-vis des besoins recensés pour la gestion de modèles. Le cœur de notre thèse se trouve au cinquième chapitre portant sur notre proposition. Il définit notre formalisme permettant la représentation de modèles répartis à tous les niveaux de modélisation. Le sixième chapitre présente les évaluations théorique et pratique, concernant le pouvoir d'expression de notre formalisme. Le septième chapitre présente la mise en œuvre de notre cadre de modélisation et les opérations sur les modèles relatives à nos besoins. Finalement, nous concluons sur le travail présenté dans cette thèse en récapitulant les résultats obtenus, les contributions majeures et leurs perspectives.

Chapitre 2. État de l'art

Le présent chapitre expose l'état de l'art pour la gestion de modèles. Notre étude porte sur la modélisation (y compris la représentation de modèles) et les opérations sur les modèles dans le cadre de la gestion des connaissances.

2.1 Terminologies et Notions fondamentales

Cette section présente les terminologies et précise les définitions des notions fondamentales qui seront utilisées tout au long de cette thèse. Ces terminologies et notions sont couramment utilisées par les techniques de modélisation; certaines d'elles peuvent être nommées autrement à travers différents formalismes. Ces définitions permettront de clarifier les notions et terminologies utilisées afin de réduire tout malentendu et ambiguïté. Nous convenons que dans ces définitions, les termes en italique seront ceux utilisés tout au long de la thèse alors que les parenthèses qui les suivent peuvent contenir les synonymes généralement utilisés dans d'autres formalismes.

2.1.1 Modèle

Dans sa thèse [152], Daniel K. Schneider a discuté de la notion de modèle scientifique pour une approche en sciences sociales. Schneider apporte un éclairage pour ceux qui s'intéressent comme nous à la modélisation dans une perspective de gestion de connaissances. Il reprend à son compte les trois fonctions (ou caractéristiques) d'un modèle développé par Stachowiak [153] :

1) Fonction de *représentation*: un modèle représente un original naturel ou artificiel que l'on peut décrire comme un ensemble d'éléments et leurs interrelations.
2) Fonction de *réduction*: un modèle ne représente pas toutes les caractéristiques de l'original mais uniquement celles qui sont pertinentes au but de la modélisation.
3) Fonction *subjectivisante*: un modèle n'a pas de relation «naturelle» avec son original, et son interprétation tient compte du but et de l'usage.

La notion de modèle est l'objet de traitement dans différents domaines, par exemple le génie logiciel (ex.: [17], [78]), les bases de données (ex.: [18], [143]), la représentation de connaissances au sein de l'intelligence artificielle (ex.: [145], [148], [158]),

l'ontologie (ex.: [20], [62]), l'aide à la décision (ex.: [52]), etc. La notion de modèle peut varier dépendamment des domaines. Dans le domaine de l'aide à la décision (la recherche opérationnelle), un modèle peut être vu comme un ensemble d'éléments et de relations entre des éléments, et ces relations expriment plutôt des relations mathématiques entre variables d'entrées et de sorties. Dans le domaine d'ontologies, un modèle peut être vu comme un ensemble de termes et relations entre termes. Un modèle dans le domaine de bases de données peut être un schéma de données. Dans le domaine de la représentation de connaissances, la notion de modèle est proche de celle de contexte, visant la contextualisation des connaissances, bien que cette notion de contexte soit encore discutée ([145] - chapitre 5, [148], [158]). Dans le domaine du génie logiciel, la notion de modèle est représentée par différents termes, par exemple, `space` dans [15], `Universe` dans sNets [80], `SubjectArea` dans CDIF, `Package` dans UML et dans MOF.

Bien que la notion de modèle puisse être concrétisée différemment selon les domaines appliqués, nous constatons qu'un modèle est toujours représenté par un ensemble d'éléments et de relations entre eux. La Définition 1 présente notre définition de modèle qui permet, à notre avis, de couvrir tous les types de modèles dont ceux cités ci-haut, à titre d'exemple.

Définition 1 : Modèle

Un *modèle*, étant un codage d'une représentation de quelques aspects de la réalité ou de l'univers du discours, contient un ensemble d'éléments de cette représentation et de relations entre ces éléments.

2.1.2 Typage

Le mécanisme de typage nous permet de définir et d'organiser d'une manière générique les éléments à modéliser. Les types sont définis à partir des propriétés communes de ces éléments et permettent de regrouper ces derniers en fonction d'une même sémantique. Les types peuvent aussi être organisés en hiérarchie selon leur sémantique. Afin de supporter le mécanisme de typage, nous distinguons: (i) type / type non relationnel / type relationnel, et (ii) instance / occurrence / relation. Ces notions sont présentées dans notre Définition 2. Parmi les types, nous appelons type abstrait un type qui n'a pas directement d'instances (Définition 3). Bien entendu, ces notions

fondamentales peuvent être nommées autrement dans d'autres formalismes de modélisation.

Définition 2 : Type / Instance; Type non relationnel / Occurrence; Type relationnel / Relation

Un *type* représente un ensemble d'éléments ayant les mêmes propriétés. Une *instance d'un type* représente un élément qui se conforme à la définition du type. Parmi les types, il convient de distinguer les *types relationnels* (types de relations; associations) et les *types non relationnels* (type d'entités; classe; catégorie). Parmi les instances, il convient de distinguer les *relations* – instances des types relationnels unaires/binaires/n-aires – et les *occurrences* (entités) – instances des types non relationnels (type d'entités).

Dans ce contexte, une occurrence ou un type non relationnel (dit *élément non relationnel*) ne peut pas relier directement les éléments tandis qu'une relation ou un type relationnel (dit *élément relationnel*) le peut. Une relation n-aire relie *n* éléments (*n* est un entier positif). Nous avons donc les relations unaires (n=1), binaires (n=2), ternaire (n=3), etc. Respectivement, nous avons les types relationnels unaires (n=1), binaires (n=2), ternaire (n=3), etc.

Définition 3 : Type abstrait

Un type dit *abstrait* n'a pas directement d'instances. Il sert plutôt à organiser et à éclaircir sémantiquement une hiérarchie de spécialisation de types.

En particulier, pour le typage entre éléments de différents niveaux de modélisation, nous distinguons: (i) élément / nœud / lien, et (ii) méta-élément / méta-nœud / méta-lien (Définition 4). Ces notions nous aident à clarifier davantage la nature des concepts dans une architecture de modélisation. La notion de niveau de modélisation et le rapport entre les différents niveaux de modélisation sont présentés à la section 2.2 (page 14).

Définition 4 : Élément / Méta-élément; Nœud / Méta-nœud; Lien / Méta-lien; Arc / Méta-arc

Un *élément* (entité, relation, etc.) défini à un niveau de modélisation est vu comme instance d'un type défini au niveau méta, le type est appelé *méta-élément* (méta-entité, métarelation, etc.). Un *élément* se conforme à un et un seul *méta-élément*. Parmi les *éléments* (*instances*) définis à un niveau de modélisation, il convient de distinguer les *élément-nœuds* (*occurrences*) et les *élément-liens* (*relations*).

Un *nœud* (terme abrégé de «*élément-nœud*»), défini à un niveau de modélisation, représente une occurrence dont le type non relationnel est défini au niveau méta; ce

type est dit *méta-nœud* du premier. Un *lien* (terme abrégé de «*élément-lien*») n-aire, défini à un niveau de modélisation, représente une relation n-aire reliant *n* éléments (*n* est un entier positif), son type relationnel est défini au niveau méta pour unir *n* méta-éléments correspondants et dit *méta-lien* n-aire du premier. Nous avons donc les liens / méta-liens unaires (n=1), binaires (n=2), ternaire (n=3), etc. Un *arc* (terme abrégé de «*élément-arc*») représente un lien binaire et unidirectionnel, liant un élément source à un élément destination. Son méta-lien, binaire et unidirectionnel, dit aussi *méta-arc*, est défini pour unir le méta-élément source au méta-élément destination (correspondants à l'élément source et à l'élément destination).

Un *nœud* peut être traité comme *instance* à un niveau mais comme *type* à un niveau inférieur. Par contre, un *lien* / *arc* est traité toujours comme *instance* et n'est jamais vu comme *type*. Ceci signifie que tous les éléments qui peuvent être traités comme méta-éléments (*types*) à un niveau de modélisation sont des nœuds. Le lecteur trouvera dans les pages suivantes des exemples de ces notions.

2.1.3 Hiérarchisation entre types

Concernant la hiérarchisation entre types, il existe deux mécanismes, le sous-typage et l'héritage. Les deux mécanismes permettent à un type d'hériter des propriétés de ses types parents. Cependant, ils ne se comportent pas nécessairement de la même manière [185]. Dans le cas du sous-typage, toute instance d'un type peut se comporter comme instance de n'importe quel type parent. L'héritage est plus général que le sous-typage. Il vise la réutilisation de la mise en œuvre (code) et peut ne pas conduire à la préservation du comportement. Une instance d'un type héritier, prise comme une instance indirecte d'un type parent, peut se comporter différemment d'une instance directe du type parent. Ceci peut s'expliquer par deux raisons que voici. Premièrement, dans le mécanisme d'héritage, une propriété héritée d'un type (classe) parent peut être redéfinie dans un type héritier. Deuxièmement, par l'héritage, un type héritier n'hérite d'un type parent que les propriétés héritables, c'est-à-dire qu'il peut hériter uniquement de certaines propriétés du type parent.

Les notions de sous-typage et d'héritage sont précisées dans les Définition 5 et Définition 6. Le rapport de sous-typage ou d'héritage entre types nous permet de remplacer un type dans un modèle par un sous-type, constituant ainsi un autre modèle.

Définition 5 : Type relationnel de sous-typage

Si un élément `B` est lié à `A` par un lien de sous-typage, alors `B` hérite de toutes les propriétés (attributs, méthodes, relations, comportements, etc.) de `A`. Chaque instance de `B` peut se comporter comme une instance de `A` et compte parmi les instances de `A`. Également, si `B` est un type relationnel, les types unis par `B` peuvent aussi être unis par `A`. Le type relationnel de sous-typage est transitif, antisymétrique, et antiréflexif (cf. la Définition 27, page iv-1, Annexe IV).

Définition 6 : Type relationnel d'héritage

Si un élément `B` est lié à `A` par un lien d'héritage, alors `B` hérite toutes les propriétés (attributs, méthodes, relations, etc.) héritables de `A`, et peut redéfinir une ou des propriétés héritées. Dans le cas où `B` est un type relationnel, les types unis par `B` peuvent aussi être unis par `A`. Une instance directe de `B` est aussi une instance indirecte de `A`. Le type relationnel d'héritage est transitif, antiréflexif, et acyclique (cf. la Définition 27, page iv-1, Annexe IV).

2.1.4 Objet ou valeur

Le monde des valeurs mentionné ici est de nature bien différente de celui des éléments vus pour les objets. Des objets peuvent changer ou être transformés en fonction du temps, mais les valeurs demeurent toujours les mêmes.

Différence entre objets et valeurs

Soient deux personnes `Marie` et `Jean`, et par coïncidence, `Marie` et `Jean` ont le même nom: «Tremblay». Si les noms sont considérés comme objets, alors «Tremblay» est un objet lié à `Marie` et à `Jean` en tant que nom de `Marie` et celui de `Jean` (Figure 1-(a)). Si l'on modifie le nom de `Marie`, «Tremblay», à «Lévesque», alors le nom de `Jean` est automatiquement «Lévesque» (Figure 1-(b)). Par contre, si les noms sont vus comme valeurs, alors le fait que le nom de `Marie` est «Tremblay» est indépendant du fait que le nom de `Jean` est «Tremblay» (Figure 2-(a)). Même si nous modifions le nom de `Marie`, le nom de `Jean` reste inchangé (Figure 2-(b)).

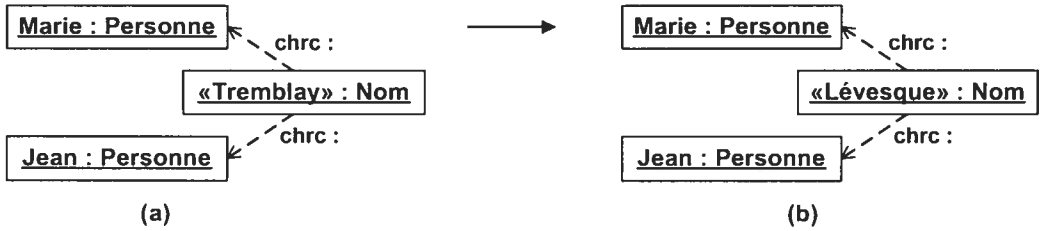


Figure 1 : Noms considérés comme objets

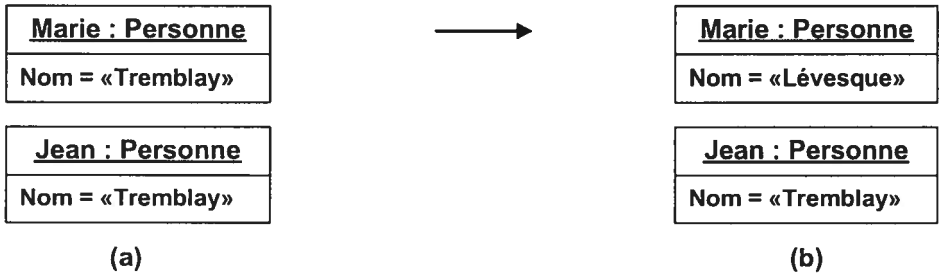


Figure 2 : Noms considérés comme valeurs

2.2 Modèle et Niveaux de modélisation

La modélisation est une activité qui consiste à transformer les descriptions informelles de la réalité en descriptions formelles, appelées modèles, afin de pouvoir opérationnaliser les connaissances [1]. La métamodélisation est une activité qui consiste à définir le vocabulaire et la grammaire, appelés métamodèle, permettant la réalisation de modèles[2]. Pour sa part, la méta-métamodélisation est une activité qui consiste à définir le vocabulaire et la grammaire, appelé méta-métamodèle, permettant la réalisation de métamodèles. Ces trois définitions correspondent aux trois niveaux de modélisation appelés: M1, M2 et M3. Le niveau M0 est ajouté pour indiquer le monde réel que l'on cherche à modéliser.

La Figure 3 illustre l'architecture de modélisation à quatre niveaux (M3, M2, M1, et M0) largement acceptée à ce jour ([16], [40], [56], [80], [140], [143]).

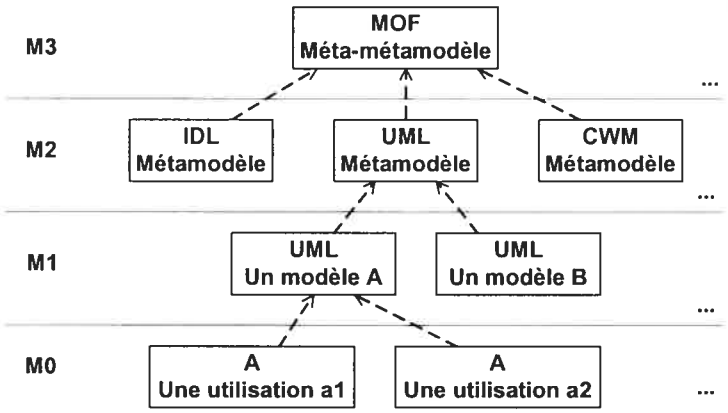


Figure 3 : Architecture à quatre niveaux

[2] http://www.metamodel.com/article.php?story=2002101021353113. Model vs. metamodel vs. meta-metamodel. October 2002
http://www.metamodel.com/staticpages/index.php?page=20021010231056977 *What is metamodeling?*. October 2002.

Chaque niveau présent dans cette architecture est brièvement décrit ci-après :

- M3 (méta-métamodèle) est le niveau le plus abstrait et réflexif dans cette architecture. Il définit les notions de base permettant la représentation des niveaux inférieurs ainsi que son niveau propre.
- M2 est le niveau métamodèle. En utilisant la grammaire spécifiée au niveau M3, ce niveau définit le langage et la grammaire pour représenter les modèles au niveau M1.
- M1 est le niveau modèle. Ce niveau définit les représentations concrètes du monde réel (modèles) ainsi que les descriptions de ces représentations. Chaque modèle au niveau M1 respecte la grammaire spécifiée par son métamodèle au niveau M2.
- M0 est le monde réel décrit au niveau M1. Il s'agit du niveau concret, représentant une situation réelle et unique dans l'espace et dans le temps. Les éléments de M0 sont les éléments existants dans le monde réel que l'on décrit, et ils sont représentés par des objets au niveau M1.

On remarque une évolution dans les architectures de modélisation. Si le nombre de niveaux est inchangé, le rôle des niveaux a été modifié. Pour VODAK [34], IRDS [67], Telos [101], Modèle uniforme [55], UML1.x, etc., le niveau M0 représentait les instances, et le monde réel n'était pas pris en compte. La clarification des activités de modélisation a permis de préciser: (i) seulement les trois derniers niveaux (M3, M2, M1) appartiennent au monde de la modélisation tandis que le niveau M0, présentant le monde réel, n'en fait pas partie; (ii) ce que l'on appelle couramment les types (classes) et les instances sont au même niveau M1 ([16], [80]), contrairement à ce qui est souvent perçu.

De plus, afin d'éviter des problèmes dus à la représentation de la nature des concepts (y compris des modèles), par exemple le problème de l'instanciation double discuté dans ([16], [36], [38]), une architecture de modélisation doit distinguer les notions suivantes [36]: l'instanciation, la conformité entre éléments, et la conformité entre modèles. Ces notions sont en effet de nature très différente, chacune possède son propre comportement vis-à-vis des contextes dans lesquels elle opère. L'*instanciation* indique le rapport «types-instances» entre éléments du niveau M1. La *conformité entre éléments* (de différents niveaux de modélisation) indique le rapport de conformité entre un élément et son méta-élément. Enfin, la *conformité entre modèles* indique le respect sémantique entre un modèle et son métamodèle. La Figure 4 illustre ces notions.

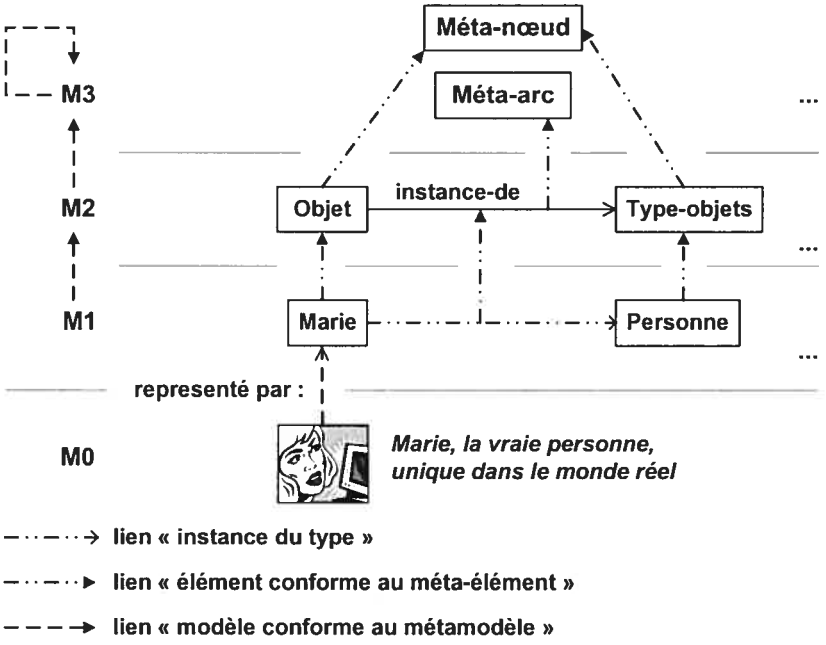


Figure 4 : Distinction entre les notions de conformité

Dans cette figure, `Marie` au niveau M1 représente la vraie personne *Marie* (au niveau M0). `Marie` au niveau M1 est, d'une part, conforme à `Objet` dans le contexte global (ce qui est spécifié par le lien «élément conforme au méta-élément» sur l'axe vertical), et d'autre part, une instance de `Personne` dans le contexte local (ce qui est spécifié par le lien «instance du type» sur l'axe horizontal). La Figure 4 montre aussi sur l'axe vertical par les liens «modèle conforme au métamodèle» que M1 est conforme sémantiquement au métamodèle M2, que M2 est conforme sémantiquement à M3, et que M3 est conforme sémantiquement à lui-même.

Notons le principe suivant : parmi deux niveaux de modélisation consécutifs dans une architecture de modélisation, le niveau inférieur respecte la grammaire spécifiée par son niveau supérieur. Autrement dit, le niveau supérieur spécifie le langage et les règles pour construire les éléments à son niveau directement inférieur. Ceci signifie que l'appellation des niveaux, dans l'architecture de modélisation, dépend en fait du niveau où nous nous plaçons. Si nous sommes au niveau modèle (M1), alors son niveau méta ou bien métamodèle est M2. Mais si nous nous trouvons au niveau M2 ou M3, le niveau méta de ce dernier est M3. À comprendre que le nombre des niveaux dans l'architecture dépend directement de la relation de conformité qui traverse entre les niveaux adjacents et qui relie les éléments d'un niveau aux éléments du niveau immédiatement supérieur. Néanmoins, quel est le nombre de niveaux d'abstraction réellement nécessaires pour la

représentation de connaissances ? Aucune réponse précise n'est possible. L'architecture de modélisation présentée ci-dessus n'est aujourd'hui qu'un consensus. Certaines autres questions du domaine font encore l'objet de discussion. Par exemple, quels sont les critères sémantiques précis permettant de caractériser la frontière entre les niveaux de modélisation ? Quels sont les éléments dans chaque niveau de modélisation ? À remarquer que la réflexivité du niveau le plus haut dans une architecture de modélisation (c'est-à-dire le niveau M3 dans ce cas) possède des avantages multiples. Elle permet de limiter le nombre de niveaux d'abstraction. Le niveau réflexif se valide lui-même; les outils et algorithmes applicables au niveau métamodèle sont donc aussi applicables à ce niveau [80]. Ceci facilite l'adaptation aux extensions ou modifications futures.

Dans notre contexte, nous appellerons généralement *modèles* tous ce que nous pouvons retrouver dans les différents niveaux de modélisation. Parmi les modèles, nous distinguons les métamodèles et méta-métamodèles en utilisant les règles [80 - pages: 195-197] suivantes:

- un modèle appartient au M3 s'il est aussi son propre métamodèle;
- un modèle appartient au M2 si son métamodèle est un autre modèle au niveau M3;
- un modèle appartient à M1 si son métamodèle est un autre modèle au niveau M2.

2.3 Opérations sur modèles

Cette partie présente une revue sur les opérations nécessaires pour manipuler les modèles. Comme l'existence d'une opération dépend du codage et du stockage de modèles, la liste des opérations présentées dans cette revue n'est pas exhaustive. Cette liste ne comporte que les opérations qui nous semblaient les plus pertinentes dans les domaines qui nous concernent: la gestion des connaissances, le génie logiciel, les bases de données, les ontologies, et également la qualité de service. De plus, comme la conception détaillée ainsi que l'implémentation d'une opération sur des modèles dépendent du codage et du stockage des modèles, notre présentation sur les opérations reste au niveau de la description sémantique des opérations.

2.3.1 Opérations de base

Nous nommerons les opérations de base en nous référant aux opérations élémentaires similaires dans la gestion des bases de données ([14], [18], [90]) et ainsi qu'aux

opérations de base dans les GCs ([55], [148]). Ce sont les opérations suivantes: la création; la suppression, la mise-à-jour, l'application de fonction, la copie, l'énumération, la simplification, la restriction de types, la jointure, l'intersection, et la différence entre modèles.

Création de modèles

Cette opération vise à créer un nouveau modèle. Ce modèle peut être vide ou contenir des éléments. Dans [18], on parle de l'opérateur *CreateModel* permettant de créer une structure complète de modèles basée sur un «template». Par exemple, *CreateModel* peut créer un modèle vide d'un programme constitué par une signature, une partie de déclarations et une partie de procédures.

Suppression de modèles

Elle consiste à supprimer un modèle dans la base des connaissances. La suppression d'un modèle implique aussi la suppression d'éléments contenus dans le modèle ([18], [90]).

Mise-à-jour de modèles

Cette opération permet de mettre à jour l'information sur un modèle. Inspirées de la fonctionnalité de mise-à-jour dans un système de gestion de base de données, les activités de mise-à-jour de modèle permettent: d'ajouter de nouveaux éléments au modèle, de supprimer certains éléments du modèle, ou de modifier des valeurs, etc. La mise-à-jour se fait en utilisant des opérations de base pour manipuler les objets, leurs propriétés ainsi que leurs relations [18].

Énumération

Lors du traitement d'un modèle, le parcours des éléments du modèle est parfois demandé. Alors l'opération d'énumération permet de visiter tous les éléments du modèle de manière à ce que chaque élément soit visité uniquement une fois [18].

Application de fonction

Cette opération prend un modèle et une fonction comme entrée et applique la fonction à tous les objets dans le modèle ([14], [18]).

Copie de modèle

Cette opération vise à créer une copie u d'un modèle v. Il s'agit d'une copie profonde dans le sens que tous les éléments du modèle original v sont aussi copiés. Le modèle

copié u et le modèle original v sont sémantiquement équivalents. Voici quelques références: l'opération de copie dans les graphes conceptuels [148], et l'opération de copie proposée dans ([18], [90]).

Simplification de modèle

Cette opération vise à simplifier un modèle en supprimant les éléments redondants sans toutefois perdre d'information, ni modifier la sémantique du modèle. Par exemple, dans les graphes conceptuels, si deux relations conceptuelles sont identiques, une des deux relations peut être supprimée ainsi que ses arcs [55].

Restriction de types

Empruntée de la restriction dans les graphes conceptuels ([55], [148]), cette opération représente le mécanisme de sous-typage entre types. Un type dans un modèle peut être remplacé par un sous-type.

Jointure entre modèles

Empruntée à l'opération de jointure dans les graphes conceptuels ([55], [148]), à celle d'union entre ensembles dans les mathématiques, et similaire à celle de jointure naturelle dans les bases de données, l'opération de jointure entre modèle vise à unir deux modèles sur la base de leurs éléments communs.

Soient m_1, m_2 deux modèles. Soient e_1, e_2 deux éléments identiques respectivement de m_1 et m_2. Si e_1 est identique à e_2, alors la jointure de m_1 et m_2 est le modèle résultant de l'union de m_1 et m_2 en supprimant e_1 et en reliant les relations de e_1 à e_2. Une illustration est présentée dans la Figure 5. Dans cette figure, m_3 est le modèle résultant de la jointure entre m_1 et m_2; b est un élément commun des modèles m_1 et m_2.

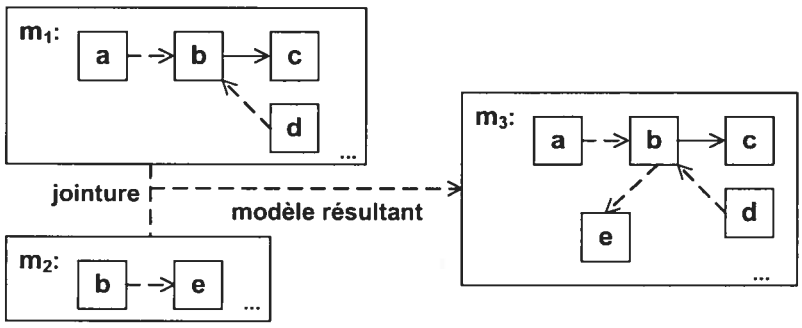


Figure 5 : Exemple de jointure entre modèles

Intersection entre modèles

Cette opération est empruntée de l'opération d'intersection entre ensembles dans les mathématiques [90]. Elle retourne la partie commune entre deux modèles.

Différence entre modèles

Cette opération consiste à retourner les éléments contenus dans un modèle mais pas dans un autre modèle ([18], [90]). Soient m_1, m_2, m_3 trois modèles; m_3 représente la différence d'éléments entre m_1 et m_2 s'il contient uniquement les éléments présents dans m_1 mais non dans m_2.

2.3.2 Autres opérations importantes

Nous présentons ici les autres opérations plus complexes telles que la dérivation de modèles, l'instanciation de modèles, la contraction et l'expansion, la requête, la correspondance entre modèles, la différence complexe entre modèles, la fusion de modèles, la transformation et la migration de modèles.

Dérivation de modèles

À partir d'un modèle, le mécanisme de restriction de type nous permet de construire et de déduire des modèles plus spécifiques tandis que le mécanisme de dérivation de modèles vise à construire des modèles plus généraux.

La dérivation de modèles est similaire à l'opération de projection décrite dans [18]. Dans [53], la dérivation de modèles est définie grâce à l'opération de projection dans les GCs. Un exemple est illustré par les Figure 6 et Figure 7 (cf. [53]). `Model0` (Figure 6) est un modèle de qualité quantitative pour la livraison de vidéo. Il comprend les trois dimensions quantitatives : la langue (`Language`), le débit (`FrameRate`) et la taille (`Size`). `Model2` (Figure 7) est un modèle dérivé de `Model0` et ne comprend que les deux dimensions, `FrameRate` et `Size`.

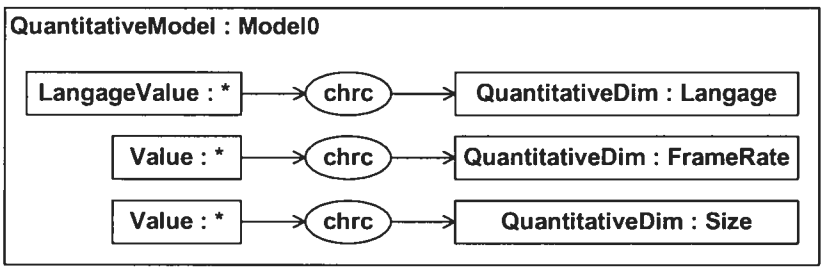


Figure 6 : Modèle de qualité quantitative

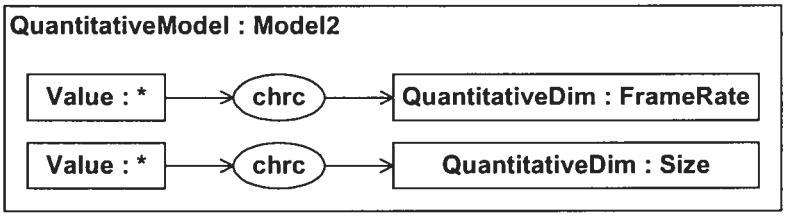


Figure 7 : Dérivation de modèles

Instanciation de modèles

Comme nous le savons, dans le modèle orienté-objet, si un objet est une instance d'une classe, cet objet doit se conformer à la définition de la classe et tous les attributs de l'objet que la classe spécifie doivent avoir des valeurs concrètes. Autrement dit, une fois qu'une classe est instanciée, tous les types attachés (attributs, relations) sont aussi instanciés. Pourtant, l'instanciation de modèles ne nécessite pas que tous les types dans le modèle à instancier soient instanciés. Dans [53], l'instanciation de modèle correspond à la spécialisation dans les GCs, et elle est définie grâce à l'opération de projection dans les GCs. Un exemple est illustré par les Figure 7 et Figure 8 (cf. [53]).

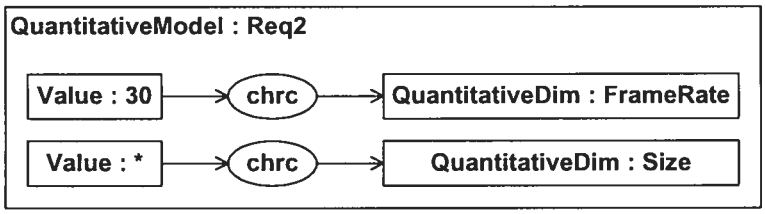


Figure 8 : Instanciation de modèles

La Figure 8 représente que `Req2` est un modèle instancié de `Model2`. Au lieu d'être attachée à une valeur non spécifiée dans le modèle `Model2`, la dimension `FrameRate` dans `Req2` prend la valeur 30. Pour `FrameRate`, le concept générique `[Value:*]` dans `Model2` est remplacé par un concept individu `[Value:30]` dans `Req2`.

Contraction et Expansion

Les deux opérations sont empruntées des GCs. Dans les GCs ([55], [148]), la contraction remplace un graphe correspondant à une définition d'un type par un concept (ou relation) de ce type, et l'expansion remplace un concept par son graphe de définition.

Requête

L'opération de requête permet la recherche ainsi que la sélection d'information dans

la base des modèles. Par exemple: l'opération *Select* décrite dans [18], la recherche dans les GCs présentée dans [55], ou la recherche dans le sNets présenté dans [80]. Et le résultat retourné de cette opération se conforme à la qualification donnée. Les requêtes peuvent être exprimées par un langage qui peut être similaire à certains langages de requête comme, par exemple, celui pour XML (Xquery) ou pour l'interrogation de bases de données relationnelles (SQL) ou objets (OQL).

Correspondance entre modèles (*Matching*)

Cette opération s'intéresse à la partie similaire entre deux modèles. Elle prend deux modèles comme entrées et retourne comme sortie une correspondance sémantique dit «*mapping*» qui est un modèle indiquant comment les éléments dans les deux modèles en entrée sont équivalents (en correspondance). La Figure 9 (cf. figure 4 - [14]) décrit un exemple de ce problème. Map_{ee} est une correspondance représentant la correspondance respective entre les éléments `Emp#` et `Nom` dans `Emp` avec `EmployéID` et `Nom` dans `Employé`. Map_{ee} est le modèle résultat de l'opération `Match(Emp,Employé)`.

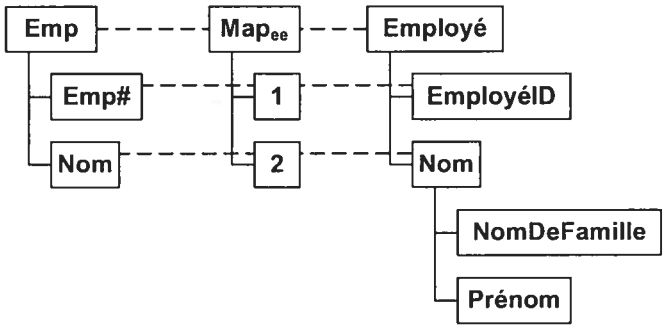


Figure 9 : Match(Emp, Employé) = Map_{ee}

Ce problème existe dans plusieurs domaines tels que la gestion de données et de métadonnées, le «data wearhouse», les ontologies, le génie logiciel, etc. Voici quelques références intéressantes: [13], [18], [42], [43], [44], [76], [86], [87], [90], [91], [93], [104], [138], [188].

Différence complexe entre modèles

Cette opération vise à effectuer la différence entre deux modèles. Si l'opération de différence entre modèles, présentée dans la section 2.3.1 (page 17), ne considère pas la correspondance sémantique entre deux modèles en entrée, l'opération de différence complexe tient compte de cette considération. Les Figure 9 et Figure 10 décrivent un scénario simple. Map_{ee} est une correspondance entre deux modèles `Emp` et `Employé`

(Figure 9). La partie qui existe dans `Employé` mais non dans `Emp` comprend les éléments `NomDeFamille` et `Prénom`, ce qui est indiqué dans le résultat obtenu `Diff(Employé,Map`$_{ee}$`)=<Employé2,Map`$_{ee}$`2>` (Figure 10 (cf. figure 5 – [14])).

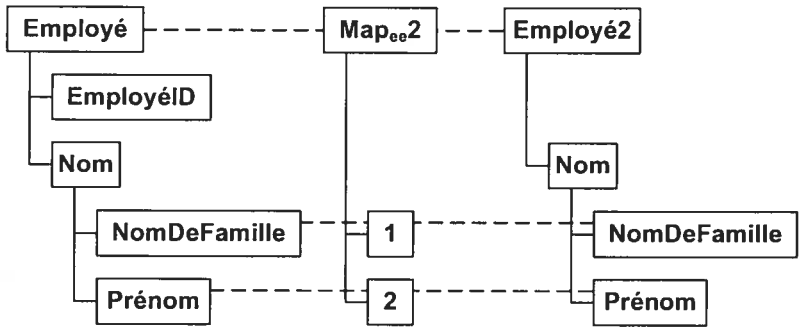


Figure 10 : Diff(Employé, Map$_{ee}$) = < Employé2, Map$_{ee}$2>

Voici quelques références traitant de ce sujet: [14], [18], [90], [132].

Fusion de modèles

L'opération couvre un problème bien plus large que l'opération de jointure présentée dans la section 2.3.1 (page 17). Soient `A`, `B` deux modèles et `Map`$_{AB}$ une correspondance entre `A` et `B`. Alors, l'opération de fusion retourne un modèle résultant de l'union de `A` et `B` selon `Map`$_{AB}$.

Ce problème se présente dans plusieurs domaines d'application comme l'intégration des bases de données, la fusion et l'intégration des ontologies, le développement des logiciels, etc. Certaines références sont: [18], [51], [75], [76], [90], [104], [132], [133], [135], [144], [156].

Transformation de modèles

Cette opération traite de la transformation d'un modèle en un autre modèle, en appliquant des règles de transformation (*mapping*), à chacune de ses phases. Ces deux modèles représentent une même situation mais dans des structures différentes. Si l'on respecte l'architecture de modélisation (Figure 3) présentée dans la section 2.2 (page 14) avec un seul méta-métamodèle au niveau M3, on distingue les deux cas suivants:

- Le modèle source et le modèle destination sont représentés dans un même formalisme. Autrement dit, ils sont basés sur un même métamodèle. Dans ce cas, la transformation ne réside qu'au niveau M1 et ne touche pas le niveau M2 dans l'architecture de modélisation.
- Le modèle source et le modèle destination sont représentés dans des formalismes

(métamodèles) différents. Dans ce cas, la transformation réside au niveau M1 ainsi qu'au niveau M2 dans l'architecture de modélisation. Voir l'exemple 1.

Transformation entre modèles et Liens de transformation entre modèles (exemple 1)

Comme la Figure 11 (page 24) l'illustre, la transformation d'un modèle source (`Modèle1`) en un modèle destination (`Modèle2`) respecte les règles de transformation entre les éléments du métamodèle source (`Métamodèle1`) et les éléments du métamodèle destination (`Métamodèle2`) correspondants. Dans ce cas, concernant la représentation de liens entre modèles, nous pouvons considérer que `Modèle1` est associé à `Modèle2` par un lien «transformation», et celui-ci est associé à une ou des règles de transformation par des liens «applique les règles».

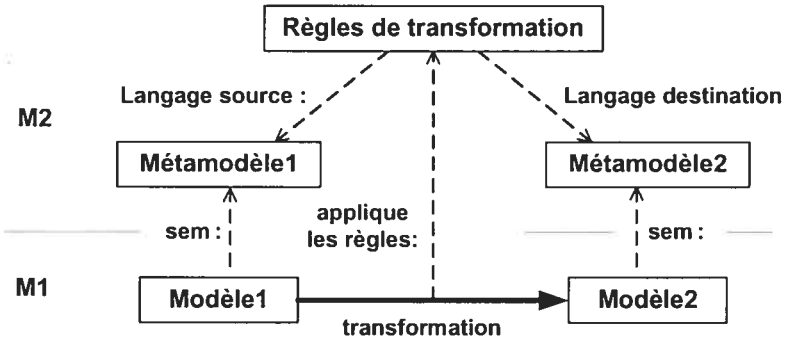


Figure 11 : Mécanisme de transformations

La transformation est un problème important qui se retrouve dans divers domaines tels que la gestion de données et de métadonnées et le génie logiciel. Ces références donnent d'autres informations à propos de cette réalité: [6], [14], [24], [31], [76], [80], [81], [88], [89], [104], [130], [139], [140], [143], [150], [151], [159], [162].

Migration de modèles

La migration de modèles signifie, pour nous, le mécanisme permettant de faire migrer dynamiquement les instances d'un type à un autre type en respectant la définition de ce dernier. Elle est inspirée du mécanisme de définition des types dans les GCs ([55], [148]). Voici un exemple. Soit `PersonneEmployée-Teximus` un type dont la définition exprime que chaque employé (instance de type `PersonneEmployée-Teximus`) est une personne (instance de type `Personne`) travaillant pour la compagnie `Teximus`. Soit *`modèle1`* un modèle représentant que `toto` est une personne travaillant pour la compagnie `Teximus`. `toto` satisfait donc la définition de `PersonneEmployée-Teximus` et devient une instance de `PersonneEmployée-`

Teximus. Soit *modèle2* un modèle représentant que toto est une instance de PersonneEmployée-Teximus. Nous pouvons dire dans ce cas que *modèle2* est le modèle migré du *modèle1* par la migration de toto du type Personne vers le type PersonneEmployée-Teximus. La migration de modèles est une sorte de transformation de modèles basée sur un même métamodèle.

2.4 Formalisme pour la représentation de modèles

Les formalismes de modélisation seront étudiés dans le Chapitre 4.

2.5 Synthèse

La gestion de modèles prend une importance grandissante et notre travail porte sur deux axes de recherche essentiels dans la gestion de modèles, soient: (i) la représentation de modèles, et (ii) les opérations pour manipuler les modèles. Nous avons passé en revue les niveaux de modélisation et également fait une revue des opérations de manipulation des modèles. Nous présentons dans le prochain chapitre les besoins essentiels pour la gestion de modèles et puis dans le Chapitre 4 les formalismes de modélisation existants.

Chapitre 3. Besoins

La représentation de modèles est un axe de recherche essentiel dans la gestion des modèles. Elle est également notre première préoccupation de recherche. Nous souhaitons représenter tous les modèles répartis sur les trois niveaux de modélisations (M1, M2 et M3). Il nous faut identifier les besoins essentiels pour le niveau M2 (afin de représenter le niveau M1), et également identifier ceux pour le niveau M3 (pour représenter les niveaux M2 et M3). Notre besoin d'opérations sur modèles est aussi présenté.

3.1 Besoins pour M2

Nous avons répertorié les besoins fondamentaux (y compris les notions) que les métamodèles de niveau M2 devraient supporter pour représenter des connaissances/modèles au niveau M1. Certains de ces besoins et notions peuvent être nommés autrement dans les différents formalismes.

3.1.1 Typage entre éléments

3.1.1.1 Typage entre éléments du niveau M1

BesoinM2 1 : Typage entre éléments du niveau M1

Pour représenter la structure de connaissances du monde réel sur les deux niveaux *types* et *instances* au niveau M1 (cf. la section 2.2, page 14), un formalisme défini au niveau M2 doit permettre de représenter au niveau M1 les notions de base suivantes:

- Pour le niveau *types*: type, type non relationnel, et type relationnel unaire/binaire/n-aire (cf. la Définition 2, page 11).
- Pour le niveau *instances*: instance, occurrence, et relation unaire/binaire/n-aire (cf. la Définition 2, page 11).

3.1.1.2 Typage entre éléments de deux niveaux de modélisation consécutifs

Étant donné que parmi deux niveaux de modélisation consécutifs dans l'architecture de modélisation, le niveau inférieur respecte la grammaire spécifiée par son niveau supérieur, les éléments du niveau inférieur peuvent donc être vus comme des instances dont leurs types, dits méta-éléments, sont définis au niveau supérieur.

Tout modèle conforme à notre définition de modèles (cf. la Définition 1, page 10)

peut être représenté par des *nœuds* et des *liens binaires* (cf. la Définition 4, page 11). Un modèle est représenté par un ensemble d'éléments et de relations entre ces éléments. Un élément peut être représenté par un nœud ou un lien binaire. Une relation unaire, binaire ou n-aire qui unit un ensemble d'éléments peut toujours être fondamentalement représentée par un nœud et un ensemble de liens binaires de telle sorte que ce nœud puisse via cet ensemble de liens binaires unir l'ensemble d'éléments en question. De cette façon, parmi deux niveaux de modélisation consécutifs dans l'architecture de modélisation, (i) les éléments du niveau inférieur peuvent être seulement des nœuds et des liens binaires et (ii) leurs méta-éléments définis au niveau supérieur peuvent être seulement des méta-nœuds et des méta-liens binaires. Ceci n'influence pas le pouvoir de représenter des modèles et permet en outre de faciliter la mise en œuvre des représentations.

Ainsi, concernant le typage entre éléments de deux niveaux de modélisation consécutifs dans l'architecture de modélisation, un formalisme défini au niveau supérieur (c'est-à-dire le niveau M3 ou M2) doit supporter au moins les notions suivantes: élément / méta-élément, nœud / méta-nœud, et lien / méta-lien binaire (cf. la Définition 4, page 11). Ceci est noté par notre BesoinM2 2. À souligner qu'un lien (ou méta-lien) binaire dans ce contexte associe deux éléments dont un élément peut aussi être un lien (ou méta-lien). L'exemple 1 (page 24) a illustré ces propos.

BesoinM2 2 : Typage entre éléments de deux niveaux de modélisation consécutifs

Pour indiquer les méta-éléments définis à un niveau de modélisation, les notions suivantes sont nécessaires: méta-élément, méta-nœud, et méta-lien binaire; également, pour indiquer les éléments à un niveau de modélisation conformes aux méta-éléments définis au niveau méta, les notions suivantes sont de même nécessaires: élément / nœud / lien binaire (cf. la Définition 4, page 11).

Un lien (ou méta-lien) binaire peut exister sous toutes les formes. Par exemple, un lien (méta-lien) entre deux nœuds (méta-nœuds), ou entre un nœud (méta-nœud) et un lien (méta-lien), ou entre deux liens (méta-liens).

3.1.2 Rapport entre niveaux *types* et *instances*

Afin de pouvoir se déplacer entre les niveaux *types* et *instances* au même niveau M1 ou aux différents niveaux de modélisation, un formalisme doit remplir le besoin de «Passage entre niveaux *types* et *instances*» (BesoinM2 3). Et le besoin «Distinction entre

différentes notions de conformité» (BesoinM2 4) permet, quant à lui, d'éviter des problèmes dus à la représentation de la nature des éléments/modèles (cf. la section 2.2, page 14).

BesoinM2 3 : Passage entre niveaux types et instances

Un élément peut être traité comme *instance* à un niveau mais comme *type* à un autre niveau.

BesoinM2 4 : Distinction entre différentes notions de conformité

Il est nécessaire de distinguer ces notions: l'instanciation «types-instances» entre éléments du niveau M1, la conformité entre éléments et méta-éléments, et la conformité entre modèles et métamodèles.

3.1.3 Classification et Hiérarchisation entre types

La plupart des formalismes de modélisation permettent la hiérarchisation entre types non relationnels mais non entre types relationnels. Cependant, étant donné qu'un type relationnel peut avoir ses attributs propres, ses comportements et types relationnels avec d'autres éléments, alors pour permettre à un type relationnel d'hériter des propriétés (attributs, comportements, etc.) d'un type plus général, la hiérarchisation entre types relationnels est aussi importante. Il en est de même pour la hiérarchisation entre types non relationnels, dans le domaine de la modélisation. Le besoin «Hiérarchisations entre types» (BesoinM2 5) souligne la nécessité de classer des types non relationnels (respectivement des types relationnels) en hiérarchie en fonction de leur sémantique spécialisation/généralisation.

BesoinM2 5 : Hiérarchisations entre types

Les types (respectivement les types non relationnels / les types relationnels) sont organisés en hiérarchie.

Concernant la multi-classification et la connaissance partielle, examinons le problème de classification des automobiles abordé dans ([55], [41]).

Classification des automobiles (exemple 2)

En se basant sur le nombre de portes, nous avons trois types: les véhicules 2 portes (`Véhicule-2portes`), les 4 portes (`Véhicule-4portes`) ou les mini-fourgonnettes (`Véhicule-Van`). En se basant sur le mode de propulsion, nous avons les véhicules à 2 roues motrices (`Véhicule-2RM`) ou à 4 roues motrices (`Véhicule-4RM`). Enfin, selon l'origine

des véhicules, nous pouvons définir trois types: Europe (`Véhicule-Europe`), Amérique (`Véhicule-Amérique`) et Asie (`Véhicule-Asie`).

*Si les véhicules sont classés à partir de ces trois critères (Figure 12), toutes les combinaisons sont possibles (3*2*3=18 possibilités (cf. Figure 13)). Et si ce principe est appliqué pour les cas plus complexes dans le sens où le nombre de critères est plus important, alors le nombre et le volume de toutes les possibilités peuvent rapidement devenir incontrôlables. De plus, il ne faut pas oublier que l'adéquation avec la réalité peut être altérée.*

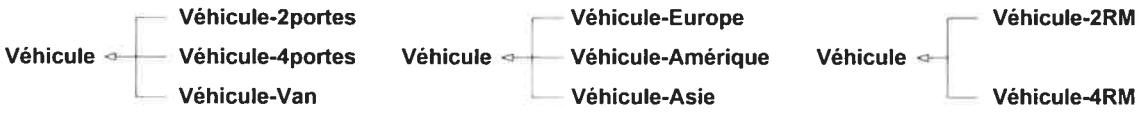


Figure 12 : Différentes hiérarchies basées sur les trois critères

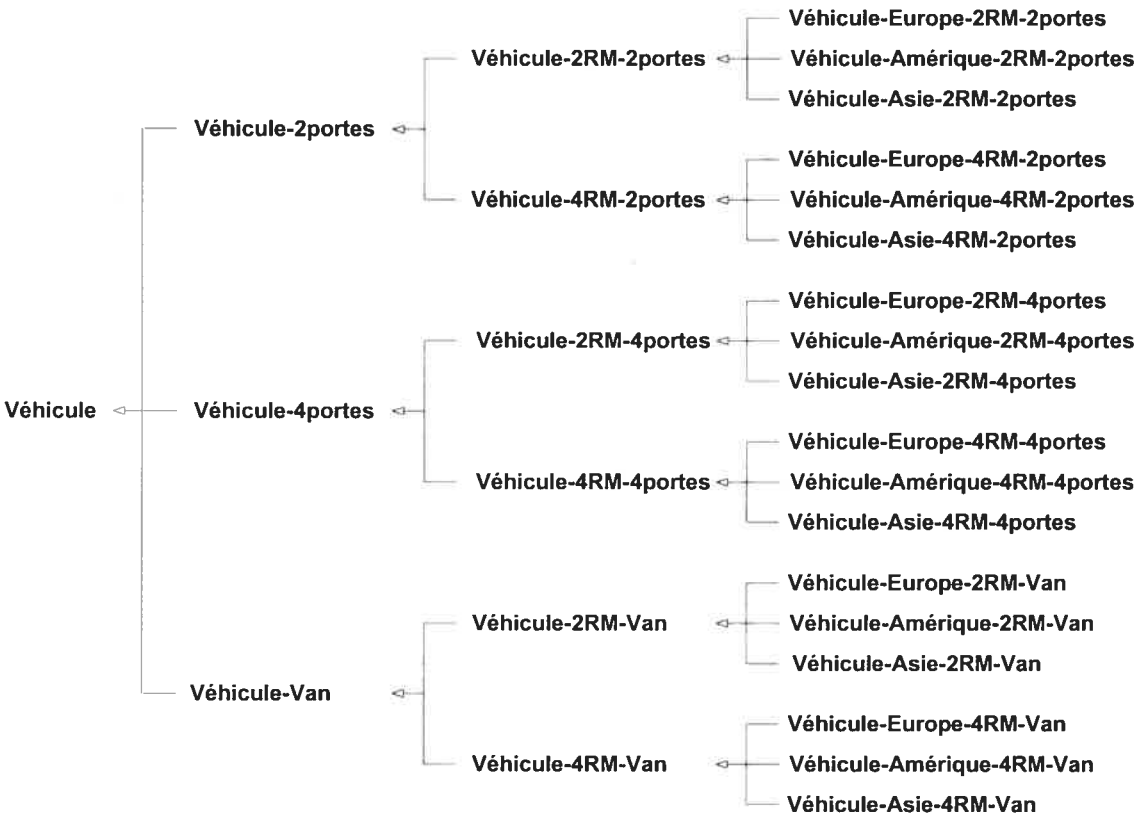


Figure 13 : Une hiérarchie basée sur les trois critères

Un problème concernant la classification lors de connaissances partielles est aussi posé. Si nous avons une voiture 4 portes avec 2 roues motrices dont l'origine est inconnue (car elle a été assemblée en Europe à partir des pièces en provenance de l'Asie), elle est donc classée dans le type `Véhicule-2RM-4portes`. Mais qu'adviendra-t-il lorsque nous connaîtrons son origine?

Une meilleure solution pour ces problèmes est la multi-classification ou un mécanisme

permettant de faire migrer dynamiquement les instances d'un type à un autre tel que la règle «Si l'élément A est une instance de TypeA avec de nouvelles propriétés, alors A devient une instance de TypeB».

Afin d'éviter l'explosion combinatoire en raison du sous-typage (ou de l'héritage) multiple comme les exemples de classifier des automobiles, nous avons le besoin «Multi-classification et Connaissance partielle» suivant:

BesoinM2 6 : Multi-classification et Connaissance partielle

Il permet à une instance d'être classée à plusieurs types (multi-classification) ou il est possible de faire migrer dynamiquement les instances d'un type à un autre type.

3.1.4 Modèle de rôles

La notion de rôles s'attache à la représentation des aspects temporaires et dynamiques des objets du monde réel. Nombreux travaux de recherche (exemples: [148], [147], [131], [77], [184], [60], [29], [145], [154], [34]) traitent cette notion de rôles mais ne partagent pas toujours une même ontologie ni une même approche. Afin de réduire tout malentendu et ambiguïté à propos de rôles, la Définition 7 précise les notions de base utilisées dans notre contexte. L'exemple 3 illustre ces notions.

Définition 7 : Objet / Rôle, Type d'objets / Type de rôles

Le terme *objet* désigne un objet de l'univers du discours ou dans le monde réel. Les *rôles* représentent les aspects temporaires et dynamiques des objets du monde réel ainsi que de l'univers du discours. Un *type d'objets* (un *type de rôles*) représente un ensemble d'objets (de rôles) ayant les mêmes propriétés telles que les attributs, relations et comportements.

Personnes et rôles (exemple 3)

Une personne, durant son existence, passe par différents états correspondant à son âge, suivant l'ordre: enfant, adolescent, adulte, personne âgée. Elle peut aussi passer par d'autres états comme célibataire, marié, salarié, sans-emploi, ou à la retraite. En outre, elle peut jouer divers rôles en ce qui a trait aux études et à l'enseignement comme: étudiant, chargé de cours, ou professeur. Elle peut également être identifiée par des relations relatives au travail, au recrutement comme: employé ou employeur.

Un rôle employé *peut passer par divers états comme employé temporaire, employé permanent. Un employé permanent peut, par ailleurs, s'occuper d'autres rôles comme chef de projet, chef d'équipe.*

Ainsi, les types tels que `Personne` *(représentant l'ensemble de personnes),* `Enfant` *(représentant l'ensemble d'enfants),* `Adolescent` *(représentant l'ensemble d'adolescents), etc. sont des types d'objets alors que les types tels que* `Etudiant` *(représentant l'ensemble de rôles étudiant),* `ChargéDeCours` *(représentant l'ensemble des rôles chargé de cours),* `Professeur` *(représentant l'ensemble des rôles professeur), etc. sont des types de rôles.*

Afin de tenir compte des situations présentées dans l'exemple ci-dessus, le BesoinM2 7 souligne la distinction explicite entre les notions suivantes: objet, rôle, type d'objets, et type de rôles. Aussi, le BesoinM2 8 répertorie toutes les interactions possibles entre objets, rôles.

BesoinM2 7 : Objets, Rôles, Types d'objets, Types de rôles

Distinction explicite entre ces notions: objet, rôle, type d'objets, et type de rôles.

BesoinM2 8 : Rapports entre objets et rôles / types de rôles

(i) Un objet/rôle peut simultanément jouer des rôles d'un même type ou de types différents ([34], [60], [77], [184]).

(ii) Différents objets et/ou rôles ne peuvent pas s'occuper simultanément un même rôle ([29], [34], [60], [77], [131], [145], [147], [148], [154], [184]).

(iii) Des objets de types différents peuvent jouer un même type de rôles ([154]).

(iv) Un objet peut dynamiquement acquérir ou abandonner des rôles / types de rôles ([34], [60], [77], [131], [154]).

(v) Un type de rôles peut être transféré d'un objet à un autre objet ([154]). Nous entendons par là qu'un rôle concret, cédé par un objet, pourra être joué par un autre objet. Il est aussi possible de spécifier un rôle concret sans nommer son joueur. Par exemple, toutes les caractéristiques d'un poste ouvert peuvent être spécifiées indépendamment de la personne qui sera engagée pour ce poste.

(vi) Un rôle (ou type de rôles) peut être acquis ou cédé indépendamment d'autres rôles (ou types de rôles) ([34], [60]). Par exemple, le fait qu'une personne devient étudiante est indépendant du fait qu'elle devient employée.

(vii) Les types de rôles peuvent dépendre des types relationnels ([29], [148], [154]). C'est-à-dire, un type de rôles peut toujours s'attacher à un type relationnel, et un objet doit participer à une relation instanciée de ce type relationnel pour pouvoir jouer un rôle instancié du type de rôles en question.

(viii) Différents types de rôles peuvent partager des structures ou comportements

communs. Par exemple, les types de rôles peuvent être organisés en hiérarchie afin d'hériter des propriétés ([34], [60], [77], [131], [145], [147], [148], [154], [184]).

(ix) La séquence selon laquelle les types de rôles peuvent être acquis et cédés, peut représenter la restriction ([34], [131], [154], [184]). Voici un exemple. Supposons que les personnes peuvent jouer les rôles «*employé permanent*» et que seulement les rôles «*employé permanent*» peuvent jouer les rôles «*chef de projet*». Le séquencement des rôles «*employé permanent*», «*chef de projet*» permet de représenter la restriction suivante: un chef de projet doit être un employé permanent.

(x) L'état d'un objet (il s'agit de la situation de l'objet, représentable par exemple à travers les propriétés de l'objet) peut être spécifié par rôles ou par types de rôles. Autrement dit, l'état d'un objet peut varier dépendamment du rôle ou du type de rôles sous lequel l'objet est examiné ([34], [77], [154], [184]).

(xi) Les propriétés d'un objet peuvent être spécifiées par types de rôles ([34], [60], [131], [154]). Autrement dit, un attribut ou comportement peut caractériser divers types de rôles mais être réalisé différemment par ces types. Par exemple, une personne peut détenir plus d'un numéro de téléphone (exemple: un privé et deux autres pour ses rôles de types *étudiant* et *employé*).

(xii) L'accès dépend du contexte ([34], [60], [77], [131]). Par exemple, si un objet est accédé sous un rôle ou un type de rôles, les propriétés propres à d'autres rôles ou types de rôles de l'objet peuvent être invisibles.

(xiii) Un objet et ses rôles ont des identités distinctes ([34], [184]). Contrairement à l'approche où un objet et ses rôles partagent une même identité d'objet ([60], [77], [154]), l'approche où l'objet et ses rôles ont des identités distinctes possède des avantages multiples. Elle facilite la mise en œuvre des critères suivants: le critère (i) (particulièrement dans le cas où un joueur peut s'occuper de plusieurs rôles d'un même type), le critère (v) (par exemple, le cas où un rôle peut exister sans être attaché à un joueur concret). Elle permet de distinguer facilement le nombre de rôles de celui de leurs joueurs. Par exemple, le dénombrement des passagers, tel qu'abordé dans [184]. Dû au fait qu'une personne peut être comptée plusieurs fois comme passager (si elle prend

l'autobus plusieurs fois par semaine), le nombre de personnes, en tant que passagers, pendant une même semaine, peut être de 1000 alors que le nombre de passagers, selon le rôle passager, peut être de 4000.

3.1.5 Classification dynamique et Classification statique

Examinons les deux exemples ci-après.

Objets - Type statique/dynamique et Classification statique/dynamique (exemple 4)

L'exemple 3 apporte la distinction suivante au sujet des objets et types d'objets. Durant son existence, une occurrence de `Personne`*, représentant une personne réelle, reste toujours comme une instance de type* `Personne`*. Dépendamment de son état (enfant, adolescent, adulte, personne âgée, célibataire, marié, salarié, sans-emploi, etc.), cette occurrence est aussi classée à un ou des types (*`Enfant`*,* `Adolescent`*,* `Adulte`*,* `PersonneÂgée`*,* `Célibataire`*,* `PersonneMariée`*,* `Salarié`*,* `PersonneSansEmploi`*, etc.) pour une période de son existence. Le type* `Personne` *est dit statique. Les types tels que* `Enfant`*,* `Adolescent`*,* `Adulte`*,* `PersonneÂgée`*,* `Célibataire`*,* `PersonneMariée`*,* `Salarié`*,* `PersonneSansEmploi` *sont dits dynamiques et sous-types dynamiques du type* `Personne`*. Supposons que les personnes sont regroupées en types selon leurs pays de naissance: les personnes nées en France (*`PersonneNéeEnFrance`*), les personnes nées au Canada (*`PersonneNéeAuCanada`*), etc. Étant donné qu'une personne a un seul pays de naissance invariable, les types tels que* `PersonneNéeEnFrance`*,* `PersonneNéeAuCanada` *sont aussi dits statiques et ils sont des sous-types statiques de* `Personne`*.*

Rôles - Type statique/dynamique et Classification statique/dynamique (exemple 5)

Similaire au cas d'objets et types d'objets, nous retirons de l'exemple 3 la remarque suivante sur les rôles et types de rôles. Un rôle employé*, durant son existence, fait partie définitivement du type de rôles* `Employé`*. Tout en restant lié au type* `Employé`*, ce rôle peut aussi faire partie du type de rôles* `TempEmp` *(ou* `PermEmp`*), seulement pour la période où il est dans l'état «employé temporaire» (ou «employé permanent»). Par ailleurs, ce rôle «employé» change de type quand il change d'état. Par exemple, il change de type* `TempEmp` *au type* `PermEmp` *quand il change d'état «employé temporaire» à l'état «employé permanent». Donc, le type de rôles* `Employé` *est statique alors que les types de rôles* `TempEmp`*,* `PermEmp` *sont dynamiques et sous-types dynamiques du type* `Employé`*.*

Alors, le besoin «Classification dynamique et classification statique» tient compte de la distinction entre l'espace d'états et l'espace d'objets pour un type d'objets/rôles.

BesoinM2 9 : Classification dynamique et classification statique

Distinction explicite entre la classification dynamique et celle statique. Pour un type non relationnel (une classe), la *classification dynamique* divise exhaustivement l'espace d'états de ses instances en sous-espaces disjoints et la *classification statique* divise exhaustivement l'espace d'objets de ses instances en sous-espaces disjoints [184].

3.1.6 Contraintes structurelles

Problème 1: Comment peut-on modéliser la situation suivante ?

Au M1, `SignerContrats` *représente un type de relations «*signer-des-contrats*» entre des participants. Ce type implique un nombre de x types de rôles* fournisseur *(*`Fournisseur`*), un nombre de y types de rôles* client *(*`Client`*), et un nombre de z types de rôles* témoin *(*`Témoin`*) pour lesquels: (contrainte 1)* $1 \leq x \leq 3$*; (contrainte 2)* $1 \leq y \leq 2$*; (contrainte 3)* $1 \leq z \leq 2$.

Chaque relation instanciée de `SignerContrats` *va relier les (x+y+z) éléments suivants : x rôles* fournisseur *du type* `Fournisseur`*, y rôles* client *du type* `Client` *et z rôles* témoin *du type* `Témoin`*. L'arité de* `SignerContrats` *est variable.*

De plus, (contrainte 4) dans une relation de type `SignerContrats`*, un rôle* fournisseur *ou* client *doit être joué par une organisation d'état (*`OrganisationGouvernement`*), ou une entreprise privée (*`EntreprisePrivée`*), ou une personne (*`Personne`*). Pour sa part, le rôle* témoin *doit être joué par un rôle* avocat *(*`Avocat`*) ou une organisation (*`Organisation`*). Finalement, un rôle* avocat *doit être joué par une personne (*`Personne`*).*

Toute relation de type `SignerContrats` *décrite ci-dessus peut être illustrée par la Figure 14.*

Nous pouvons ajouter des contraintes sur des participants de `SignerContrats`*. Voici quelques exemples. (contrainte 5) Parmi les joueurs des rôles fournisseur dans chaque instance de* `SignerContrats`*, il y a 1 ou 2 organisations d'état mais une personne au maximum. (contrainte 6) Un rôle fournisseur doit participer à au moins une relation de type* `SignerContrats`*. (contrainte 7) Il y a de 1 à 2 rôles fournisseur différents participant chacun à une ou des relations de type* `SignerContrats` *avec les mêmes participants restants ((x-1) rôles fournisseur, y rôles client et z rôles témoin). (contrainte 8) Une entreprise privée doit participer à au moins une relation de type* `SignerContrats` *sous le rôle fournisseur. (contrainte 9) Il y a au maximum 2 entreprises privées différentes dont chacune participe à une ou des relations de type* `SignerContrats` *sous le rôle fournisseur avec les mêmes rôles restants ((x-1) rôles fournisseur, y rôles client et z rôles témoin) et les mêmes joueurs de ces*

rôles. Similairement, il est possible pour `SignerContrats`*, d'avoir d'autres contraintes de cardinalité sur des types de rôles et de joueurs restants. Si les mêmes participants (x rôles fournisseur, y rôles client et z rôles témoin) peuvent signer des contrats ensemble de 2 à 4 fois au maximum, alors (contrainte 10) le nombre d'itérations de* `SignerContrats` *est de 2 à 4.*

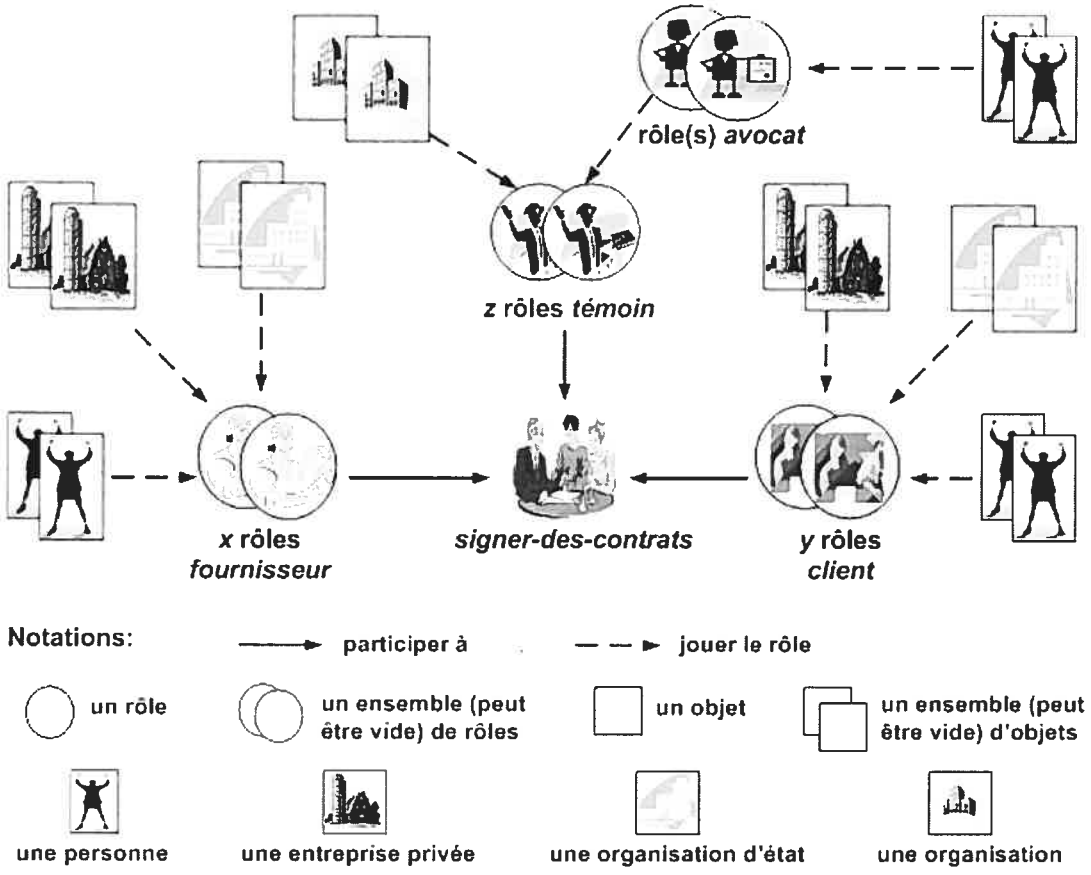


Figure 14 : Une relation *signer-des-contrats* (de type *SignerContrats*)

Évidemment, le pouvoir de représenter des contraintes sur des éléments/modèles est une exigence pour mieux exprimer la sémantique de ces éléments/modèles. Les besoins «Contraintes de l'arité min/max» (BesoinM2 10), «Contraintes de cardinalité totale min/max pour un type relationnel» (BesoinM2 11), «Contraintes de cardinalité locale min/max pour un type relationnel» (BesoinM2 12), «Contraintes de l'itération min/max pour un type relationnel» (BesoinM2 13) et «Contraintes de cardinalité min/max pour un méta-lien» (BesoinM2 14) identifient différents types de contraintes structurelles sur des types d'objets / types de rôles / types relationnels à capturer. Plusieurs parmi ces types de contraintes sont relevés dans le Problème 1 ci-haut. À notre connaissance, aucun des formalismes actuels de niveau M2 ne distingue ni supporte tous ces besoins mentionnés afin de modéliser des situations comme celle présentée dans le Problème 1 (page 34).

BesoinM2 10 : Contraintes de l'arité min/max

D'une façon plus générale, les contraintes de l'arité capturent le nombre minimal/maximal des éléments impliqués conjointement dans un événement. Nous présentons, ci-après, certains types de contraintes de l'arité pouvant exister dans diverses situations à modéliser:

(i) *Contraintes de l'arité min/max sur un attribut (A) pour un type (T).* Ceci capture le nombre minimal/maximal de valeurs pour l'attribut *A* qu'une instance de type *T* peut avoir en même temps. Par exemple, les contraintes suivantes: *une* personne a *un seul* nom, *une seule* date de naissance, mais peut détenir *plusieurs* surnoms.

(ii) *Contraintes de l'arité min/max sur la composition d'un type d'objets (T).* Ceci capture le nombre minimal/maximal des éléments composants d'un objet de type *T*. Par exemple, la contrainte suivante: *une* fenêtre est composée d'*un* châssis et d'*une à cinq* vitres.

(iii) *Contraintes de l'arité min/max sur l'occupation d'un type de rôles (TRôles) pour un type d'objets/rôles (T).* Ceci capture le nombre minimal/maximal de rôles instanciés du type *TRôles* qu'un objet/rôle (instancié du type *T*) peut jouer en même temps. Par exemple, les contraintes suivantes: *une* personne peut s'occuper de *moins de trois* rôles *employé* en même temps.

(iv) *Contraintes de l'arité min/max d'un type relationnel (TR).* Ceci capture le nombre minimal/maximal des éléments pouvant s'unir par une relation (instancié du type relationnel *TR*). Par exemple, les contraintes 1, 2, 3 dans le Problème 1 (page 34).

(v) *Contraintes de l'arité min/max sur l'implication d'un type d'objets/rôles (T) sous un type de rôles (TRôles) dans un type relationnel (TR).* Ceci capture le nombre minimal/maximal d'objets/rôles (instanciés du type *T*) qui, en tant que joueurs de certains rôles instanciés du type *TRôles*, peuvent s'impliquer dans une relation instanciée du type relationnel *TR*. Par exemple, la contrainte 5 dans le Problème 1 (page 34).

BesoinM2 11 : Contraintes de cardinalité totale min/max pour un type relationnel

Pour un type d'objets/rôles impliqué dans un type relationnel, ces contraintes capturent le nombre minimal/maximal des instances de ce type relationnel

auxquelles une instance de ce type d'objets/rôles peut participer. Par exemple, les contraintes 6 et 8 dans le Problème 1 (page 34).

BesoinM2 12 : Contraintes de cardinalité locale min/max pour un type relationnel

Ceci capture le nombre minimal/maximal des relations possibles (instanciées des types relationnels) par rapport aux éléments impliqués dans une relation. Par exemple, les contraintes 7 et 9 dans le Problème 1 (page 34).

BesoinM2 13 : Contraintes de l'itération min/max pour un type relationnel

Ceci capture le nombre minimal/maximal de relations instanciées de ce type relationnel qui existent entre les mêmes participants. Par exemple, la contrainte 10 dans le Problème 1 (page 34).

BesoinM2 14 : Contraintes de cardinalité min/max pour un méta-lien

Ceci capture le nombre minimal/maximal des liens possibles (conformes à ce méta-lien défini au niveau méta) par rapport aux éléments impliqués dans un lien. Par exemple, les contraintes de cardinalité suivantes: *un* élément est conforme à *un et un seul* méta-élément; *un* modèle se conforme à *un et un seul* métamodèle.

3.1.7 Types relationnels entre types et/ou instances

Problème 2: Types relationnels entre types et/ou instances

Reprenons le Problème 1 (page 34). Soient `FournisseurTeximus` *un rôle* fournisseur *joué par l'entreprise privée* `Teximus`; `AvocatPapin` *un rôle* avocat *joué par la personne* `Papin`; `TémoinAvocatPapin` *un rôle* témoin *joué par* `AvocatPapin`. *Examinons par la suite les situations suivantes.*

Situation 1. *Les rôles impliqués directement dans les* `SignerContrats` *peuvent être regroupés selon une ou certaines instances. Prenons, par exemple, le cas des rôles* client. *Soit* `FournisseurTeximus-TémoinAvocatPapin-Client` *un type défini par le modèle suivant. Étant un sous-type de* `Client`, *le type* `FournisseurTeximus-TémoinAvocatPapin-Client` *regroupe tous les rôles* client *dont chacun doit participer à au moins une relation instanciée de* `SignerContrats` *pour laquelle:* `FournisseurTeximus` *est le seul rôle* fournisseur *et* `TémoinAvocatPapin` *est le seul rôle* témoin. *Une telle relation peut être illustrée par la Figure 15 (page 38).*

La question est comment peut-on modéliser le rapport entre le type `FournisseurTeximus-TémoinAvocatPapin-Client` *avec deux instances* `FournisseurTeximus` *et* `TémoinAvocatPapin` *?*

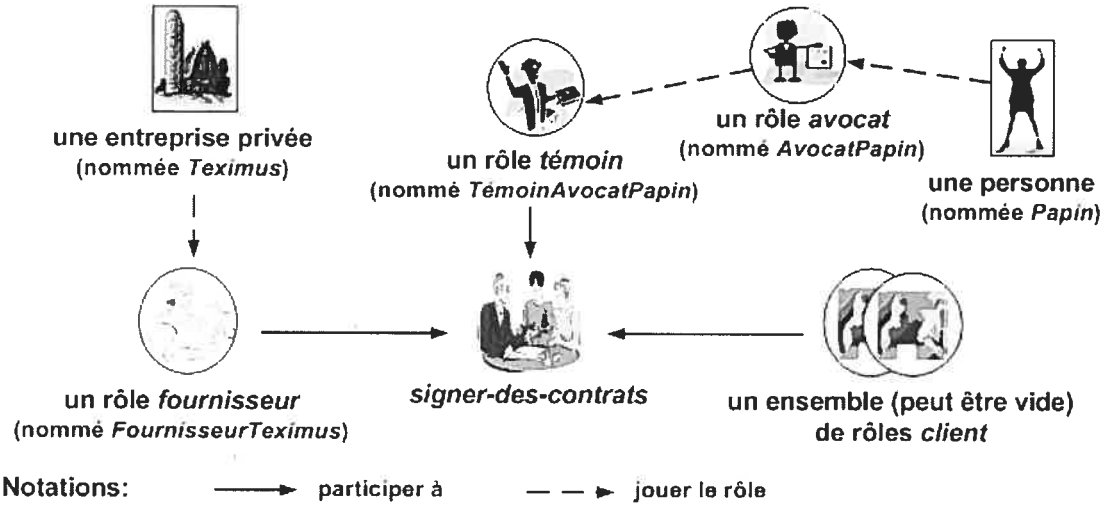


Figure 15 : Une relation *signer-des-contrats* (de type *SignerContrats*) dans la situation 1

Situation 2. *Les joueurs des rôles dans les relations de type* `SignerContrats` *peuvent aussi être regroupés selon une ou certaines instances. Prenons pour exemple le cas des personnes dans le rôle* client. *Soit* `Teximus-Papin-PersonneClient`, *un type défini par le modèle suivant: étant un sous-type de* `Personne`, *le type* `Teximus-Papin-PersonneClient` *regroupe toutes les personnes qui sont les clients de l'entreprise privée* `Teximus` *sous le témoignage de l'avocat* `Papin`. *C'est-à-dire, chacune de ces personnes doit participer dans le rôle* client *à au moins une relation instanciée de* `SignerContrats` *où* `Teximus` *est le seul joueur dans le rôle* fournisseur *et où la personne* `Papin` *est la seule impliquée comme* témoin *et en tant qu'*avocat. *Une telle relation peut être illustrée par la Figure 16 (page 38). La question est comment peut-on modéliser le rapport entre le type* `Teximus-Papin-PersonneClient` *avec deux instances* `Teximus` *et* `Papin` *?*

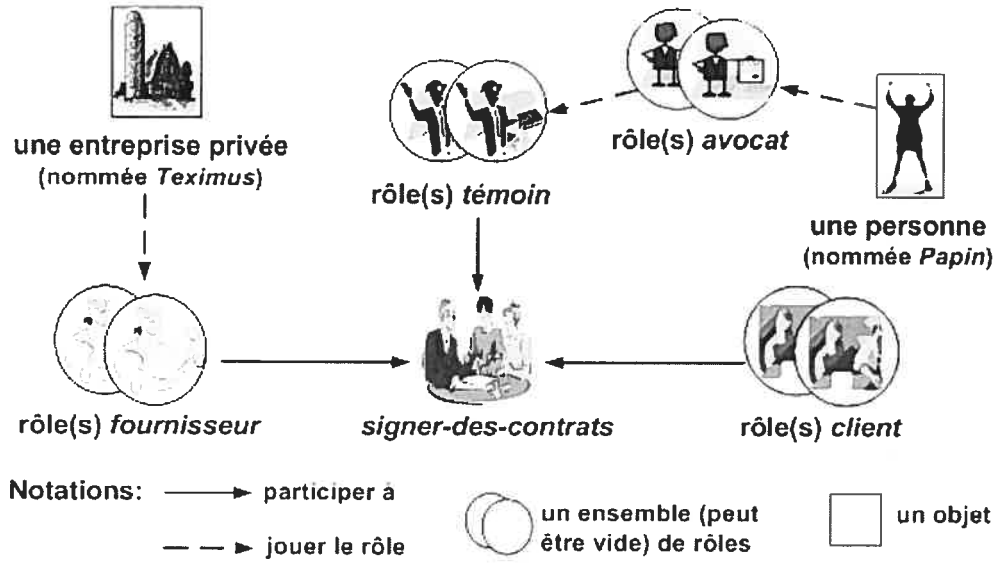


Figure 16 : Une relation *signer-des-contrats* (de type *SignerContrats*) dans la situation 2

Afin de tenir compte des situations réelles à modéliser, telles que celles présentées dans le Problème 2, nous avons le besoin «Types relationnels entre types et/ou instances» (BesoinM2 15). Il s'agit de l'extension du besoin «Relation entre catégories et/ou instances» présenté dans [55].

BesoinM2 15 : Types relationnels entre types et/ou instances

Un type relationnel peut être défini entre types, ou entre type(s) et instance(s), ou entre instances.

3.1.8 Représentation de modèles et de leurs liens

Pour représenter correctement les modèles et mieux les gérer, il est important de classifier les modèles selon leurs natures. Voici certains types essentiels de modèles: *métamodèle* d'un modèle, *modèle structurel* d'un type relationnel, et *modèle de conditions*. La notion de *métamodèle* d'un modèle a été étudiée dans la section 2.2 (page 14). Nous introduisons ici deux autres notions: *modèle structurel* d'un type relationnel, et *modèle de conditions*

Une *structure* ou bien un *modèle structurel* d'un type relationnel spécifie comment ce type relationnel relie ses éléments. Un type relationnel peut avoir plusieurs modèles structurels. Ceci peut être expliqué par l'effet de sous-typage ou d'héritage. Les modèles structurels des types relationnels doivent nécessairement être contextualisés pour mieux se distinguer entre eux-mêmes ainsi qu'avec des modèles de d'autres natures. Parmi les structures d'un type relationnel, nous appelons *structure initiale* une structure attachée à ce type relationnel par un lien de définition pour déclarer officiellement que cette structure est un modèle de définition de ce type relationnel.

Les *modèles de conditions* visent à contextualiser des conditions ou bien hypothèses, par exemple des préconditions / postconditions dans la représentation des règles sémantiques au niveau M1 que les éléments à ce niveau doivent observer.

Modèles de conditions (exemple 6)

Soit une règle au niveau M1. Si une personne (disons `x`*) est admise à un programme de maîtrise en informatique, alors elle a complété un baccalauréat en informatique ou en mathématiques. Nous avons donc les deux expressions suivantes au niveau M1: (a) «*`x`* est admise à un programme de maîtrise en informatique»; et (b) «*`x`* a complété un baccalauréat en informatique ou en mathématique». Le type* `IfThenModel` *(représentant l'ensemble de modèles de conditions au niveau M1) a pour but de faciliter, au niveau M1, l'identification,*

parmi nos expressions dans la base de connaissances, lesquelles sont des hypothèses ou existent sous des hypothèses (cf. Définition 8 – page 40). Étant considérées comme hypothèses au M1, les expressions (a) et (b) se trouvent dans une ou des modèles conformes à `IfThenModel`*. Si nous pouvons trouver une représentation de (a) dans nos modèles qui sont au M1 et non-conformes à* `IfThenModel`*, ou nous pouvons déduire cette représentation en appliquant nos règles d'inférence, alors uniquement sous ces conditions nous pouvons conclure (b).*

La Définition 8 présente notre distinction entre les notions : hypothèse et fait.

Définition 8 : Hypothèse, fait

Une *hypothèse* est la simple formulation d'une proposition, la personne qui l'énonce ne prend pas de position sur le fait de savoir si ce qu'elle dit est vrai ou faux[3]. Nous appelons *fait* une chose qui existe réellement ou qui est déduite à partir d'autres *faits* en appliquant des règles d'inférence.

Le BesoinM2 16 résume les types essentiels de modèles mentionnés qu'un formalisme doit supporter pour classer les modèles.

BesoinM2 16 : Distinction entre modèles de différentes natures

Il est nécessaire d'expliciter différents types de nature pour classer les modèles. Voici quelques types essentiels: *métamodèle* d'un modèle; *modèle structurel* d'un type relationnel (qui spécifie comment ce type relationnel relie ses éléments); *modèle de conditions* (pour contextualiser des conditions ou bien hypothèses).

Concernant la représentation de modèles et de leurs liens, nous avons identifié les deux autres besoins suivants.

BesoinM2 17 : Liens avec les modèles

Il existe différents types de liens entre les modèles et leurs éléments ou entre modèles. Exemples: les liens de *contenant* entre un modèle et ses éléments (pour indiquer clairement le rapport de type «être défini dans», ou «être contenu dans», etc. entre un modèle et ses éléments); les liens d'*accès* entre modèles (pour réutiliser dans un modèle des éléments d'un autre modèle); le lien de *respect sémantique* entre un modèle et son métamodèle; et des liens d'autres types (pour représenter explicitement des opérations entre modèles) comme la *jointure*, l'*intersection*, la *différence*, l'*inférence*, la *restriction* entre modèles.

[3] WikiPédia. http://fr.wikipedia.org/wiki/Hypoth%C3%A8se

BesoinM2 18 : Liens entre modèles au niveau M1

Il existe différents types de liens entre modèles qui sont nécessaires pour la gestion de modèles définis au niveau M1. Exemples: les liens d'*équivalence* entre les modèles (pour indiquer que ces modèles sont équivalents ou bien présentent une même situation); les liens de *transformation* d'un modèle d'une structure à une autre; les liens entre un modèle et ses *vues différentes* (qui sont correspondantes à différents niveaux d'abstraction ou à différents points de vue).

3.2 Besoins pour M3

Cette partie élabore les besoins ainsi que les notions essentiels pour un méta-métamodèle.

3.2.1 Typage entre éléments de deux niveaux de modélisation consécutifs

Comme nous avons expliqué dans la section 3.1.1.2 (page 26), concernant le typage entre éléments de deux niveaux de modélisation consécutifs, un formalisme défini au niveau M3 ou M2 doit supporter au moins les notions suivantes: élément / méta-élément, nœud / méta-nœud, et lien / méta-lien binaire (cf. la Définition 4, page 11). Notre BesoinM3 1 qui note les notions les plus fondamentales qu'un M3 doit supporter est donc identique au BesoinM2 2 (voir page 27) pour un M2.

BesoinM3 1 : Typage entre éléments de deux niveaux de modélisation consécutifs

Le BesoinM3 1 est identique au BesoinM2 2 (voir page 27).

3.2.2 Passage entre niveaux types et instances

BesoinM3 2 : Passage entre niveaux types et instances

Le BesoinM3 2 est identique au BesoinM2 3 (voir page 28).

BesoinM3 3 : Distinction entre différentes notions de conformité au niveau méta

Il est nécessaire de distinguer les notions suivantes: la conformité entre éléments et méta-éléments; et la conformité entre modèles et métamodèles.

3.2.3 Hiérarchisations entre types

Le BesoinM3 4 a pour fin de classer, au niveau méta, les types non relationnels (*méta-nœuds*) et les types relationnels (*méta-liens*) en ordre de spécialisation/généralisation.

BesoinM3 4 : Hiérarchisations entre types

Ce besoin est identique au BesoinM2 5 (voir page 28).

3.2.4 Contraintes structurelles

Le BesoinM3 5 illustre l'exigence de la représentation des contraintes de cardinalités faisant partie de la représentation sémantique des méta-liens. Comme nous l'avons mentionné, les méta-liens peuvent être uniquement binaires. Il n'est pas nécessaire, dans ce cas, de considérer différents types de contraintes, tout comme pour le cas des types relationnels au niveau M1 abordé à la section 3.1.6 (page 34).

BesoinM3 5 : Contraintes de cardinalité min/max pour un méta-lien

Le BesoinM3 5 est identique au BesoinM2 14 (voir page 37).

3.2.5 Représentation de modèles et de leurs liens

Pour un M3, en ce qui concerne la représentation de modèles (répartis aux niveaux M2 et M3) et de leurs liens, nous retrouvons les besoins, «Distinction entre modèles de différentes natures» et «Liens avec les modèles», qui sont également retenues pour M2.

BesoinM3 6 : Distinction entre modèles de différentes natures

Le BesoinM3 6 est identique au BesoinM2 16 (voir page 40).

BesoinM3 7 : Liens avec les modèles

Le BesoinM3 7 est identique au BesoinM2 17 (voir page 40).

3.3 Besoins pour opérations sur modèles

Pour le besoin de manipuler les modèles, il nous faut définir différentes opérations. Les *opérations de base* visent les fonctionnalités élémentaires pour la gestion de modèles, et nous traitons toutes celles introduites dans la section 2.3.1 (page 17): création, suppression, mise-à-jour, énumération, application de fonction, copie, simplification, restriction de types, jointure, intersection, et différence entre modèles. Nous traitons aussi certaines *opérations plus complexes* (instanciation de modèles, dérivation de modèles, requête sur des modèles, et migration de modèles) parmi celles introduites aux sections 2.3.2 (page 20). Nous définissons également d'autres opérations de base (exemple: relaxation de types relationnels, etc.) ainsi que celles plus complexes (exemple: spécialisation entre modèles). Les opérations à traiter sont détaillées dans le Chapitre 7 (page 194).

Chapitre 4. Formalismes de modélisation étudiés

Plusieurs formalismes de modélisation sont aujourd'hui disponibles. Nous nous intéressons, plus particulièrement à ceux qui concernent la représentation de modèles, utilisés dans les domaines suivants: intelligence artificielle, bases de données, génie logiciel et Web sémantique.

Dans le domaine de l'intelligence artificielle, la logique est un moyen traditionnel pour représenter les connaissances et aussi les raisonnements sur les connaissances. Sa fondation est bien solide. Mais le désavantage du formalisme logique est qu'il est non structuré. Par contre, les représentations graphiques structurent l'information. Nous nous sommes intéressés aux représentations graphiques des connaissances. Nous avons les formalismes tels que les réseaux sémantiques ([82], [146], [149]), les graphes conceptuels (GCs) [148], Classic ([21], [128]).

Dans le cadre de la gestion des connaissances, il y a le modèle uniforme des GCs [55] pour la modélisation et la métamodélisation des mémoires d'entreprise.

Dans le domaine des bases de données, pour la description des schémas de données, il y a le modèle relationnel ([32], [161]) et le modèle Entité-Relation Étendu [28] ou bien Entité-Association ([32], [84], [161]).

Le formalisme UML est un formalisme représentatif de l'approche Orientée-Objet. Il est largement utilisé dans le domaine du génie logiciel, et il est aujourd'hui considéré comme un standard de facto.

Pour l'ingénierie des modèles en particulier, nous avons retenu les formalismes suivants : le formalisme sNets [80] basé sur les réseaux sémantiques; CDIF ([68], [69], [70]) inspiré du modèle Entité-Association; MOF reconnu comme le noyau réflexif pour UML et comme le standard de facto pour la représentation de métamodèles; et XMI (XML Metadata Interchange) [126], un format d'échange de modèles basé sur la sémantique du MOF et sur la syntaxe de XML.

En ce qui a trait aux applications Web, particulièrement au Web sémantique et ontologies, les formalismes tels que XML / XML Schema, RDF / RDFS, DAML+OIL, OWL sont en cours de standardisation auprès du W3C. Les documents de spécification les plus récents de XML / XML Schema sont datés de 2006, et ceux de RDF / RDFS, OWL sont datés de 2004. OWL est une révision de DAML+OIL ([8], [30]) incorporant

des expériences apprises des conceptions et applications de DAML+OIL.

D'autres avant nous ont fait des études ou comparaisons de plusieurs formalismes parmi ceux nommés ci-dessus. Nous soulignons deux thèses datées de 2000, l'une de Gerbé [55] et l'autre de Lemesle [80].

En cherchant à modéliser le côté structurel des connaissances, en réponse aux trois besoins «Catégorie ou instance», «Classification et connaissance partielles», et «Relation entre catégories et/ou instances»[4], Gerbé [55] a montré que les graphes conceptuels étaient le meilleur formalisme pour modéliser une mémoire d'entreprise parmi : le modèle relationnel, le modèle Entité-Relation Étendu (ou bien Entité-Association), Classic, UML (version 1.x) ([142], [141]), et les graphes conceptuels (GCs). Il a proposé un modèle uniforme des GCs pour la modélisation et la métamodélisation des mémoires d'entreprise.

Pour l'ingénierie des modèles, Lemesle [80] a analysé les formalismes suivants : le formalisme des réseaux sémantiques, les GCs, XML (version 1.0 datée en 1998), XMI, MOF (version 1.3), CDIF. Il a indiqué les inconvénients ou lacunes de ces formalismes concernant la représentation de modèles et de métamodèles, et il a proposé sNets basé sur les réseaux sémantiques. Nous résumons ci-après les résultats de cette étude.

Le formalisme des réseaux sémantiques «*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*» ([80] – page 37).

Le formalisme des GCs «*nous permet donc bien d'exprimer à la fois nos modèles et leur sémantique (les métamodèles). La difficulté de ce formalisme est qu'il est surdimensionné par rapport à notre besoin*» ([80] – page 41); et des opérations de base dans les GCs [148] «*compliquent la manipulation de graphes conceptuels pour une simple représentation de modèles et de métamodèles*» ([80] – page 41).

Quant au formalisme XML, la souplesse et la lisibilité constituent ses avantages. Cependant, l'aspect «verbeux» du langage entraîne le fait que des fichiers deviennent rapidement volumineux [80]. Un autre désavantage de XML est que les données sont représentées dans un fichier XML sous forme d'un arbre tandis que les modèles et

---

[4] Ces trois besoins correspondent respectivement à nos besoins «Passage entre niveaux *types* et *instances*» (BesoinM2 3), «Multi-classification et Connaissance partielle» (BesoinM2 6) et «*Type relationnel* entre *Types* et/ou *Instances*» (BesoinM2 15) présentés dans le Chapitre 3.

métamodèles sont plus généralement constitués de graphes que d'arbres. Afin de représenter des graphes dans un document XML, il faut définir des références qui vont alors à l'encontre de la souplesse et la lisibilité ([80] – page 47).

XMI (XML Metadata Interchange) [126] est un format d'échange de modèles basé sur la sémantique du MOF et sur la syntaxe de XML, et il est le support textuel de représentation des modèles MOF [80].

Lemesle a fait une comparaison entre les noyaux réflexifs respectivement des trois formalismes CDIF, MOF (version 1.3) et sNets en fonction des concepts principaux suivants qu'il a proposés pour un métamodèle réflexif ([80] – section 5.5): entité; méta-entité; modèle; métamodèle; relation de conformité (*basedOn*) entre un modèle et son métamodèle; relation de conformité (*instanceOf*) entre une entité et sa méta-entité; relation «défini dans» (*definedIn*) entre entités et modèles; et relation «défini dans» (*definedIn*) entre méta-entités et métamodèles. Dans tous ces formalismes, les seuls concepts communs trouvés sont les concepts de méta-entité et de métamodèle, et la seule relation commune trouvée est la relation *definedIn* entre les méta-entités et leur métamodèle [80]. La relation *instanceOf* pour attacher une entité à sa méta-entité «*nous semblant vraiment essentiel pour concevoir un framework de modélisation et de métamodélisation n'est présente que dans les sNets et dans le MOF (avec son paquetage* `Reflective`)» ([80] – page 187).

Très récemment, de nouvelles versions d'UML et MOF (UML 2.0, UML 2.1.1 et MOF 2.0) ont été soumises à l'OMG. Par rapport aux versions précédentes, les versions UML 2.x et MOF 2.0 apportent des points nouveaux et elles suivent MDA.

Ainsi, pour notre analyse de formalismes vis-à-vis des besoins recensés (voir Chapitre 3) pour la gestion de modèles, nous avons choisi des formalismes utilisés dans chacun des domaines mentionnés plus haut :

- Dans le domaine de l'intelligence artificielle, nous avons retenu les réseaux sémantiques (RSs) et les graphes conceptuels (GCs). Le modèle uniforme des GCs [55] et le formalisme sNets [80] basés sur les RSs sont aussi retenus et présentés dans ce groupe afin de faciliter la compréhension de ces formalismes et de clarifier notre ligne de présentation.
- Dans le domaine des bases de données, le formalisme Entité-Association largement utilisé est choisi comme représentant. Comme CDIF est un formalisme

réflexif inspiré du modèle Entité-Association, nous le présentons dans ce groupe aussi pour faciliter la compréhension de ces formalismes.

- Dans le domaine du génie logiciel, nous avons choisi UML et MOF (le noyau réflexif pour UML) comme représentants de l'approche Orienté-Objet.
- Finalement, à propos des applications Web, nous avons retenu XML-XML Schema, RDF-RDFS et OWL.

Il existe de nombreux autres formalismes. Notre choix s'est restreint à ceux qui nous semblaient les plus connus et les plus intéressants dans ces domaines. Le reste du présent chapitre présente notre étude des formalismes choisis vis-à-vis des besoins recensés pour la gestion de modèles dans le chapitre précédent. Chacun de ces formalismes est aussi évalué suivant la liste des besoins essentiels pour le niveau de modélisation correspondant auquel il est défini. Cette évaluation est présentée partiellement dans ce chapitre en mettant l'accent sur les besoins non répondus et elle est présentée au complet dans l'Annexe I. Il est à souligner que, pour chaque formalisme, nous présentons plutôt ses éléments qui nous permettent de voir comment ce formalisme peut traiter nos besoins. Respectivement par rapport aux métamodèles et aux méta-métamodèles étudiés, le métamodèle UML et le méta-métamodèle MOF sont les plus complets (concernant la réponse à nos besoins) et offrent une plus vaste gamme de fonctionnalités. Dans le cadre de cette thèse, nos présentations sur UML et MOF sont donc beaucoup plus développées que celles sur les autres formalismes.

4.1 Réseau Sémantique et sNets

4.1.1 Réseau Sémantique

Un réseau abstrait (un graphe) est une représentation graphique qui se compose de nœuds et de liens directs entre les nœuds. Le réseau sémantique ([149], [146], [82]) est un formalisme utilisé pour représenter des éléments et leurs relations par des graphes en donnant une sémantique aux nœuds et aux liens. Il s'agit d'un outil qui simule notre représentation de la mémoire, la façon de représenter l'information en mémoire et la façon d'accéder à ces informations [80]. Les nœuds sont différents types d'information tels que les propositions, énoncés, propriétés, etc. L'étiquette associée au lien indique le type relationnel entre deux nœuds.

La Figure 17 montre un exemple d'un réseau sémantique qui représente: «La

personne masculine Jean occupe un poste d'employé travaillant pour la compagnie Teximus». Dans cette figure, nous représentons les nœuds par des symboles différents vis-à-vis de différents types d'information. Les rectangles représentent les objets, types ou concepts. Les cercles représentent les propositions associées à ces concepts. Les termes entre guillemets représentent des valeurs telles que des chaînes de caractères.

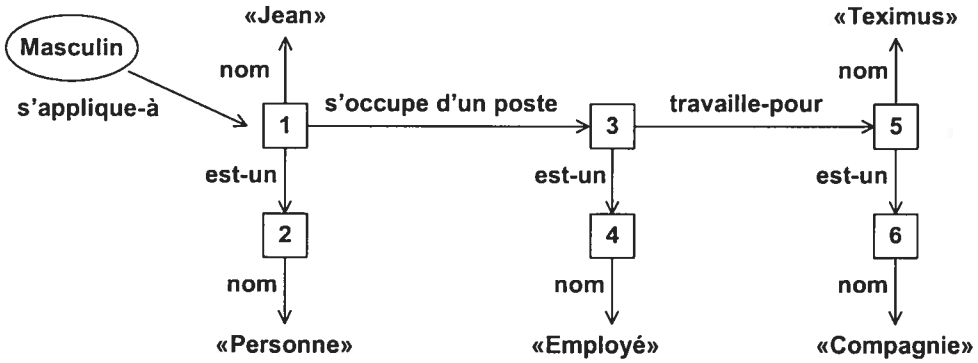


Figure 17 : Exemple d'un réseau sémantique

Évaluation des réseaux sémantiques

Les réseaux sémantiques permettent la représentation de toutes sortes d'information. Toutefois, la modularité et la structuration de la sémantique de ces informations sont absentes [80]. De plus, chaque élément (objet modélisé) est représenté par un nœud. De ce fait, les réseaux deviennent rapidement très volumineux. Il faut donc introduire aux réseaux sémantiques certaines relations pour modéliser la structuration sémantique et la modularité des éléments et standardiser la représentation des nœuds. Il en résulte des formalismes graphiques plus complexes que les réseaux sémantiques mais bien standardisés comme Entité-Association, les graphes conceptuels, sNets, etc.

4.1.2 Formalisme des sNets

Ce formalisme est proposé par Lemesle [80]. L'auteur a ajouté la modularité et le typage au formalisme des réseaux sémantiques mais de telle façon qu'un modèle sNets puisse toujours être vu comme un réseau sémantique.

La Figure 18 (cf. figure 85 - [80]) présente en notation UML le noyau réflexif des sNets. Et la Figure 19 (cf. figure 64 - [80]) présente en notation sNets une partie du méta-métamodèle des sNets. Dans sNets, chaque entité (`Entity`) est représentée par un nœud, identifié par son nom, et lié par un lien `meta` à son méta-entité (`EntityType`) et par un `partOf` à son modèle (`Universe`). Pour la représentation graphique, un nœud se

divise en deux parties: le nom de l'entité codée par ce nœud est écrit dans la partie inférieure, et le nom du type de cette entité est écrit dans la partie supérieure. Une relation (relation directe) entre deux entités est représentée par un lien (lien direct) entre deux nœuds correspondants. Un lien entre deux entités doit se conformer à sa métarelation (`RelationType`) de même nom qui est définie au niveau méta et qui relie deux méta-entités correspondantes via les deux liens `outGoing` et `destination` (cf. Figure 19). sNets dispose de l'héritage multiple pour méta-entités, mais ne dispose d'aucun mécanisme pour la hiérarchisation des métarelations. Si un type `X` est un sous-type de `Y`, alors `X` et `Y` doivent être conformes à `EntityType`, et `X` est relié à `Y` par un lien `inherits` ou `Y` est relié à `X` par un lien `subtypes`. Dans sNets, le type `Universe` a pour but d'organiser l'information, et son sous-type, le type `SemanticUniverse`, a pour but d'organiser des méta-entités et métamodèles. Chaque modèle (`Universe`) est en relation sémantique (`sem`) avec son métamodèle (`SemanticUniverse`) (cf. Figure 18). La métarelation transitive `extends` est définie pour les besoins d'extension des modèles.

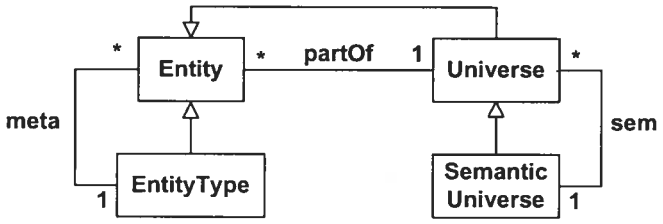


Figure 18 : Noyau réflexif de sNets (notation UML)

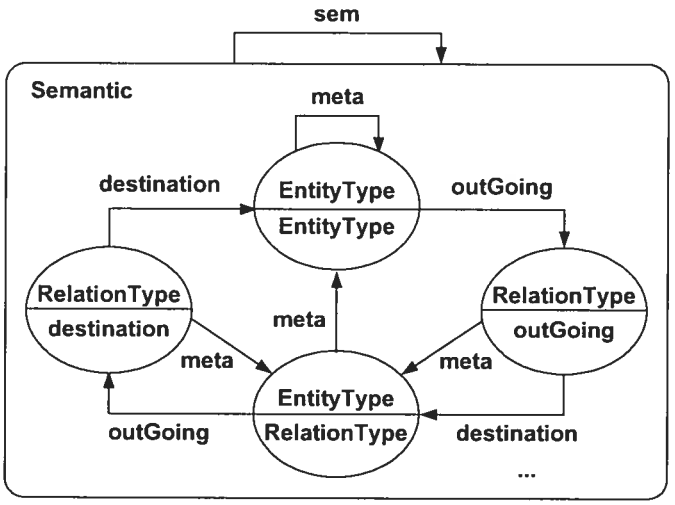


Figure 19 : Réflexivité du méta-métamodèle de sNets

La Figure 20 (cf. figure 58 dans [80]) illustre un métamodèle (M2) défini

conformément au méta-métamodèle (M3) de sNets, et illustre aussi un modèle (M1) défini conformément à ce métamodèle. Le modèle `V` (M1) prend le modèle `UML` pour métamodèle, ce qui est exprimé par un lien sémantique `sem` reliant le modèle `V` à `UML`. Le modèle `V` représente un lien `attribut` reliant l'entité `Cercle` à l'entité `rayon`. `Cercle` et `rayon` prennent respectivement `Classe` et `Attribut` pour leurs méta-entités. Ces deux dernières sont définies dans le modèle `UML` et conformes à `EntityType`. La métarelation `attribut` est aussi définie dans le modèle `UML`. Elle est une entité de type `RelationType` et relie `Classe` à `Attribut` via les deux liens `outGoing` et `destination`.

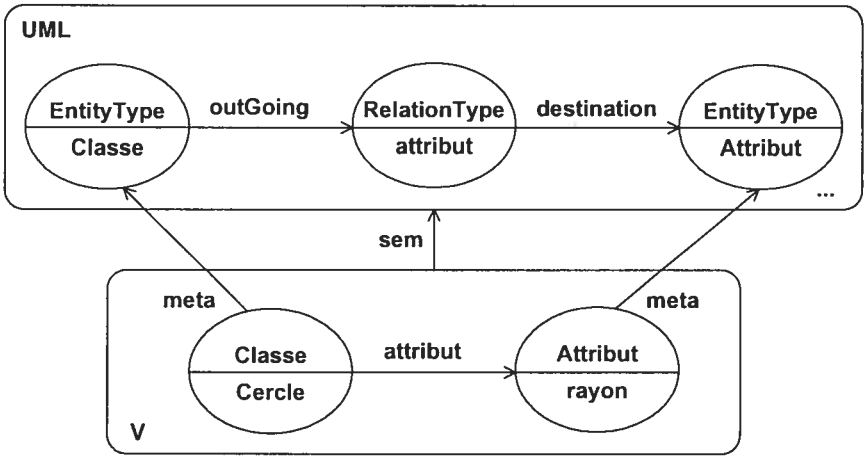


Figure 20 : Exemple d'application des sNets

Évaluation des sNets

Le noyau de sNets proposé dans [80] ne remplit pas tous nos besoins pour M3. Il n'autorise pas la représentation de liens entre liens ou entre entités avec liens (cf. le BesoinM3 1 – «Typage entre éléments de deux niveaux de modélisation consécutifs»). Il ne considère pas la hiérarchisation entre métarelations (cf. le BesoinM3 4 – «Hiérarchisations entre types»). Il ne considère pas non plus les contraintes de cardinalité minimale d'une métarelation (cf. le BesoinM3 5 – «Contraintes de cardinalité min/max pour un méta-lien»), ce qui implique qu'aucun métamodèle conforme à ce noyau ne peut remplir le besoin «Contraintes de cardinalité min/max pour un méta-lien» (BesoinM2 14) pour M2 et ne répond donc pas aux besoins pour M2. Concernant la distinction entre types de modèles, le noyau de sNets ne tient pas compte de notre notion de modèles structurels ni de celle de modèles de conditions (cf. le BesoinM3 6 – «Distinction entre modèles de différentes natures»). Il ne tient pas compte non plus des

différents liens entre modèles pour la gestion de modèles tels que l'intersection, la différence, l'inférence, la restriction, etc. (cf. le BesoinM3 7 – «Liens avec les modèles»). La section I.2 (Annexe I) présente plus en détail cette évaluation.

4.2 Graphes Conceptuels et Modèle uniforme

4.2.1 Graphes Conceptuels

La théorie des graphes conceptuels (GCs) a été développée par John Sowa [148]. Elle constitue une suite aux graphes existentiels de Peirce et aux réseaux sémantiques avec des caractéristiques adoptées des domaines linguistique et de l'intelligence artificielle. Elle a été rapidement reconnue comme un formalisme de représentation de connaissances très puissant et proche de la langue naturelle. Les lecteurs intéressés peuvent lire [148], [4], [55] pour avoir une présentation plus complète sur les GCs.

Concept, Relation, Référent, Type de concepts, Type de relations

Dans un GC, les rectangles représentent des concepts (*occurrences*), les cercles représentent des relations conceptuelles (*relations*), et les arcs relient les relations vers les concepts. Par exemple, la Figure 21 représente les énoncés suivants : *Le pays Canada a pour capitale la ville Ottawa* (représenté par ce que le concept `[Pays:Canada]` est associé au concept `[Ville:Ottawa]` par la relation `(a-Capitale)`), et *Une personne travaille pour une compagnie* (représenté par ce que le concept `[Personne]` est associé au concept `[Compagnie]` par la relation `(travaille-pour)`).

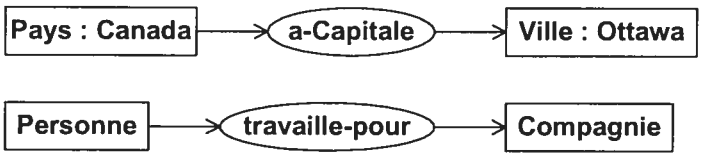


Figure 21 : Exemple de graphes conceptuels

Chaque concept individu représente un objet déterminé; il a un type et un référent. Par exemple, dans le concept `[Pays:Canada]`, `Pays` est le type et `Canada` est un référent. Mais dans un concept générique, le référent est «blank» ou est omis. Par exemple, le concept `[Personne]` est un générique et représente une personne; et dans `[Personne]`, `Personne` est le type. Le référent désigne l'objet de l'univers du discours que ce concept interprète et que le type catégorise. Le référent peut être constitué d'un GC qui le décrit. Le référent peut également être constitué d'un simple quantificateur

existentiel (identifié par «∃») ou universel (identifié par «∀»).

Tous les types de concepts forment une hiérarchie. Cette hiérarchie contient toujours les deux types primitifs suivants : T appelé le type universel, et ⊥ appelé le type absurde. T est le super-type de tous les autres types et se place donc au sommet de la hiérarchie des types, et ⊥ se place à la base de cette hiérarchie. Il n'existe aucun concept de type ⊥. Chaque type de concepts a un graphe de définition auquel toutes ses instances doivent se conformer. La Figure 22 est un exemple de la définition d'un type de concepts dans les GCs. Cette définition exprime que chaque employé (représenté par un concept de type `PersonneEmployée-Teximus`) est une personne qui travaille pour la compagnie `Teximus`.

```
type PersonneEmployée-Teximus (x) is
[Personne:?x]→(travaille-pour)→[Compagnie:Teximus]
```

Figure 22 : Graphes conceptuels – définition de type de concepts

Chaque relation a un type et un nombre entier d'arcs, soit n, pour relier la relation vers les concepts. Cette relation est appelée relation n-aire. S'il y a plusieurs arcs, ils sont numérotés. Chaque relation doit aussi se conformer au graphe de définition de son type. La Figure 23 représente un exemple de définition du type de relation `travaille-pour` mentionné dans les Figure 21 et Figure 22. Les types de relations forment aussi des hiérarchies de l'héritage.

```
relation travaille-pour(*x,*y) is
[Personne:?x]←(Agnt)←[servir]→(Dest)→[Compagnie:?y]
```

Figure 23 : Graphes conceptuels – définition de type de relations

La hiérarchie des types de concept et celle des types de relation forment un treillis des types grâce au multi-héritage.

Contexte

Un contexte est défini comme un concept dont le référent est un GC emboîté. La Figure 24 décrit un GC qui contient deux contextes. Ce GC représente la phrase : *Jean pense que Marie veut se marier avec un marin.* Le premier contexte représente une proposition dans la pensée de Jean, le deuxième contexte est emboîté dans le premier et représente une situation dans le vouloir de Marie. Le lien de coréférence entre `[Personne:Marie]` et `[Personne]` indique que le concept générique `[Personne]` dans le contexte de situation référence au même individu que le concept

[Personne:Marie].

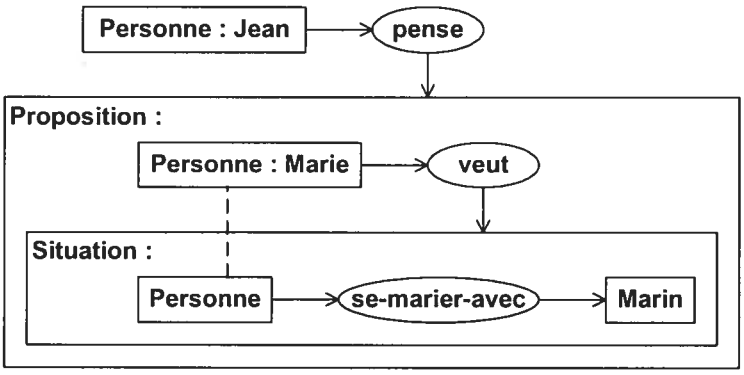


Figure 24 : GCs – Exemple de contextes

Modèle de rôles dans les GCs

Concernant la notion de rôles, Sowa a distingué dans ([145], [148]) les types naturels[5] et les types de rôles. Un type naturel d'*entités* (d'*objets*) est basé sur l'essence ou la nature des entités elles-mêmes. Il existe indépendamment de la façon dont il peut être décrit ou conceptualisé par un modélisateur. Par ailleurs, un type de rôles caractérise une entité par certains rôles qu'elle joue dans des relations avec d'autres entités, et il dépend donc de types relationnels. Sowa a proposé aussi dans [145] les types relationnels pour la représentation des types de rôles dans les GCs. Cependant, comme Steimann l'a remarqué [154], Sowa laisse les types naturels et les types de rôles coexister dans une même hiérarchie de types, ne faisant aucune différence syntaxique entre le côté des types naturels et celui des types de rôles. D'ailleurs, Sowa considère que les types de rôles sont des sous-types des types naturels. Par exemple, les types de rôles «enfant», «animal domestique», «quotient», et «nourriture» sont respectivement les sous-types des types naturels «personne», «animal», «nombre» et «subsistance physique» [147].

Opérations pour manipuler des GCs

Il existe quatre opérations de base pour manipuler les GCs: la copie (copier un GC), la restriction (remplacer un type dans un graphe par un sous-type de ce premier), la jointure (joindre deux graphes disposant d'un concept commun) et la simplification (supprimer des relations dupliquées dans un graphe).

D'autres opérations sont aussi mathématiquement définies pour la recherche des informations et l'inférence dans les GCs comme : la spécialisation entre graphes [148];

[5] Cette notion «type naturel» est équivalente à notre notion «type d'objets»

la projection d'un graphe dans un autre ([55], [137], [148]); l'opérateur de correspondance (pour accéder au éléments dans un graphe) ([55], [148]); la recherche des informations emboîtées dans des contextes [55]; les opérations permettant de transformer un concept en un contexte et à l'inverse ([47], [48]).

4.2.2 Modèle uniforme des GCs

Afin d'appliquer les GCs au domaine de modélisation et de métamodélisation, ce modèle proposé par Gerbé [55] spécifie explicitement le langage utilisé pour modéliser et méta-modéliser les connaissances. Il définit aussi, au moyen de ce langage, le méta-métamodèle et les métamodèles nécessaires pour la modélisation et la métamodélisation des connaissances. La Figure 25 (cf. figure 3.8 dans [55]) représente une partie de l'ontologie permettant la représentation des GCs.

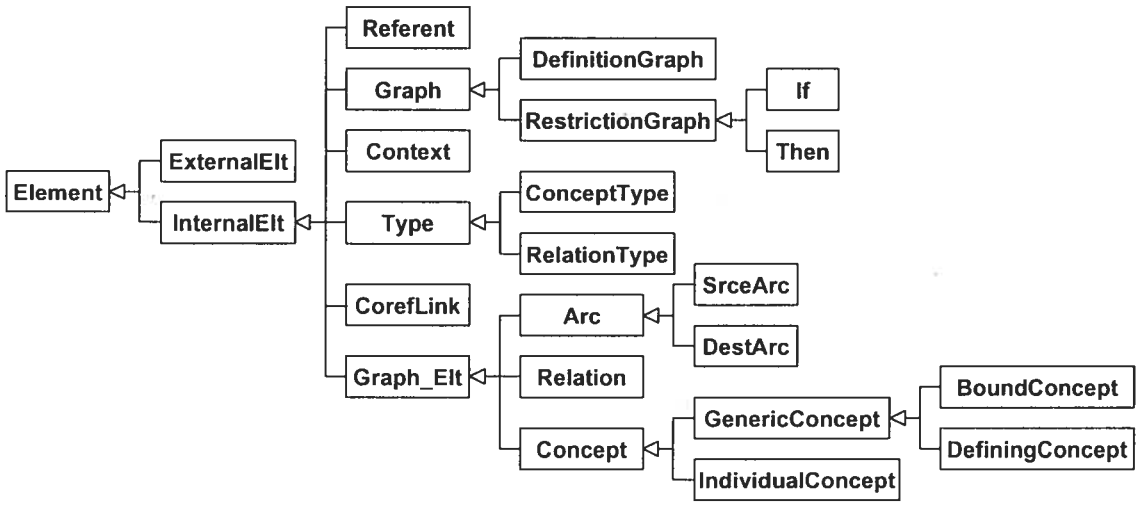


Figure 25 : Hiérarchie des types de concepts du langage

La hiérarchie des types de concepts du langage se divise en deux branches : les éléments externes (`ExternalElt`) et les éléments internes (`InternalElt`). Les éléments externes représentent les objets de l'univers du discours qui peuvent être référés par les éléments internes. Les éléments internes se regroupent en six types de base: `Referent`, `Graph`, `Context`, `Type`, `CoreLink`, `Graph_Elt`. Les référents (`Referent`) sont les représentants des objets. Les graphes (`Graph`) sont les phrases écrites dans le langage. Les contextes (`Context`) visent à regrouper les graphes. Les types (`Type`) catégorisent les références. Les liens de coréférence (`CorefLink`) lient les concepts représentant les mêmes éléments. Les éléments de graphe (`Graph_Elt`) sont les éléments : arcs (source ou destination), relations, concepts (individu ou générique). Il

faut distinguer les graphes de définition (`DefinitionGraph`) et les graphes de restrictions (`RestrictionGraph`). Les graphes de définition servent à définir les types de concepts ou de relations. Étant toujours faux, les graphes de restriction visent à contraindre les définitions des types de concepts.

Avec ce langage, nous pouvons représenter les modèles aux trois niveaux de modélisation et ainsi que d'autres formalismes comme UML, RDF-RDFS [54]. Voir [55], [54] pour les descriptions complètes de ce langage.

Évaluation des GCs et du modèle uniforme des GCs

Non seulement bien reconnus dans le domaine de l'intelligence artificielle, les GCs sont aussi reconnus comme étant le langage le plus puissant dans le groupes des langages de modélisation et de métamodélisation courants ([94], [16], [54]). Les GCs constituent la base du modèle uniforme proposé pour la modélisation et la métamodélisation des connaissances [55]. La théorie des GCs peut être aussi appliquée pour la gestion des modèles [59].

Cependant, en ce qui concerne la gestion de modèles, les GCs ainsi que le modèle uniforme des GCs ne traitent pas explicitement des modèles et de leurs relations. C'est-à-dire qu'ils ne remplissent pas tous les besoins concernés par la représentation de modèles et de leurs liens pour M3 ni ceux pour M2. De plus, la famille des GCs ne remplit pas complètement d'autres besoins pour M3 tels que le besoin «Typage entre éléments de deux niveaux de modélisation consécutifs» (BesoinM3 1), «Distinction entre différentes notions de conformité au niveau méta» (BesoinM3 3). Quant à la représentation des connaissances du monde réel en particulier, la famille des GCs ne permet pas encore de supporter divers besoins pour M2 tels que «Distinction entre différentes notions de conformité» (BesoinM2 4), «Classification dynamique et Classification statique» (BesoinM2 9), «Contraintes de l'arité min/max» (BesoinM2 10), «Contraintes de cardinalité totale min/max pour un type relationnel» (BesoinM2 11), «Contraintes de l'itération min/max pour un type relationnel» (BesoinM2 13), «Liens entre modèles au niveau M1» (BesoinM2 18). Voir les sections I.3 et I.4 (Annexe I) pour plus de détails. En outre, la mise en œuvre des GCs est difficile. Divers problèmes complexes sont encore irrésolus. Par exemple, l'interprétation sémantique ([49], [74]); l'inférence dans les GCs emboîtés complexes contenant des opérations de négation ou marqueurs génériques ([99], [136]); les projections et le «matching» dans les GC sont

connus comme NP-complet ([79], [98], [134], [186]). À notre connaissance, il n'existe pas encore un modèle des GCs répondant à tous nos besoins.

4.3 Entité-Association et CDIF

4.3.1 Entité-Association

Le modèle Entité-Relation Étendu («Extended Entity-Relationship Model») a été développé par Peter Chen en 1976 [28] puis ensuite étendu grâce aux notions de sous-typage ([32], [84], [161]). En français, on l'appelle également le modèle Entité-Association ([26], [97]).

Le métamodèle Entité-Association est défini au niveau M2. Il supporte les notions de base suivantes:

- Une entité représente un objet de l'univers du discours et elle est associée à ses attributs. Les entités ayant les mêmes propriétés forment l'ensemble des entités souvent appelé en pratique *entité* (type non relationnel).
- Une association représente une relation entre plusieurs entités et elle dispose aussi de ses attributs. Les relations ayant les mêmes propriétés forment l'ensemble des relations souvent appelé en pratique *relation* ou *association* (type relationnel).
- La relation de sous-typage entre entités;
- Les contraintes de cardinalité sur des relations entre entités.

Évaluation de Entité-Association suivant les besoins pour M2

Le formalisme Entité-Association, la représentation de connaissances se fait uniquement au niveau des types et non pas au niveau des instances. Donc il ne supporte pas plusieurs besoins concernés tels que ceux liés au passage entre niveaux types et instances. Entité-Association ne supporte pas divers autres besoins. Par exemple, la famille du modèle Entité-Association ([28], [65]) ne supporte pas parfaitement les besoins relatifs aux rôles. Entité-Association ne considère pas différents types de contraintes structurelles retenus par les besoins «Contraintes de l'arité min/max» (BesoinM2 10), «Contraintes de cardinalité totale min/max pour un type relationnel» (BesoinM2 11), «Contraintes de l'itération min/max pour un type relationnel» (BesoinM2 13). Comme Entité-Association ne dispose pas de la notion de modèle, il ne supporte aucun des besoins concernés par la représentation de modèles et de leurs liens. Voir la section I.5 (Annexe I) pour l'évaluation complète de Entité-Association.

4.3.2 CDIF

Le formalisme CDIF (CASE Data Interchange Format) ([22], [50], [46], [23], [58]) permet l'échange d'informations entre différents outils. Il a été standardisé par ISO ([68], [69], [70]).

La Figure 26 représente l'architecture de modélisation [58] et la Figure 27 représente le méta-métamodèle (M3) réflexif de CDIF ([22], [50], [46], [58]). Les méta-attributs (`MetaAttribute`) représentent des attributs et ils peuvent être affectés aux méta-objets attribuables (`AttributableMetaObject`). Les méta-objets attribuables se divisent en deux groupes: les méta-entités (`MetaEntity`) et les métarelations (`MetaRelationship`). Toute métarelation est binaire et associe deux méta-entités. La métarelation `HasSubtype` implémente le sous-typage multiple pour les méta-entités et les métarelations. Les «zones d'intérêt» (`SubjectArea`) visent à organiser les ensembles de métarelations, méta-entités et méta-attributs. Dans CDIF, les métamodèles construits au moyen du M3 de CDIF ainsi que ce M3 lui-même sont exprimés en utilisant des notions du modèle Entité-Association ([28]) développées pour CDIF. Chaque méta-entité est représentée par un rectangle. Chaque métarelation est représentée par une flèche associant deux méta-entités. Et les cardinalités minimale et maximales sur la source (respectivement sur la destination) d'une métarelation sont respectivement de types méta-attributs `MinSourceCard`, `MaxSourceCard` (respectivement `MinDestCard`, `MaxDestCard`).

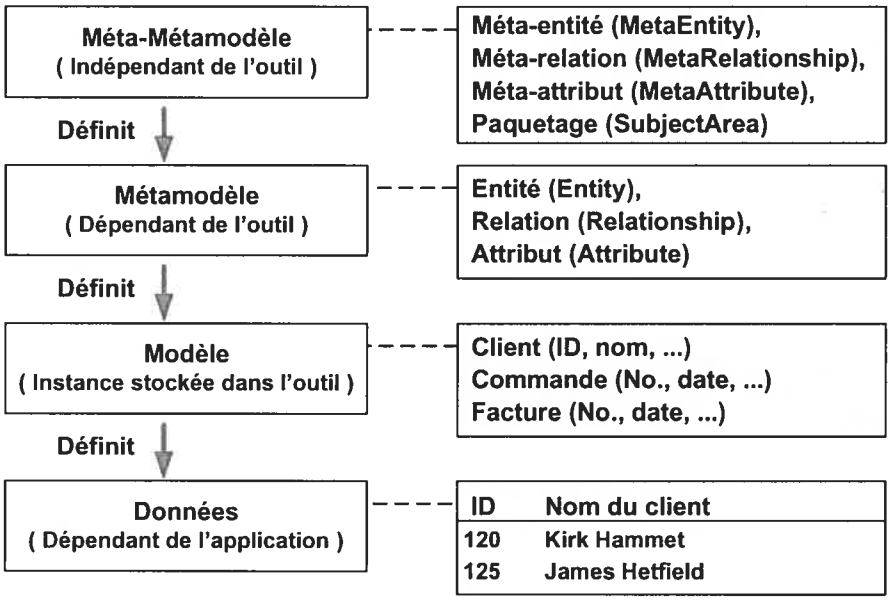


Figure 26 : Architecture de modélisation de CDIF

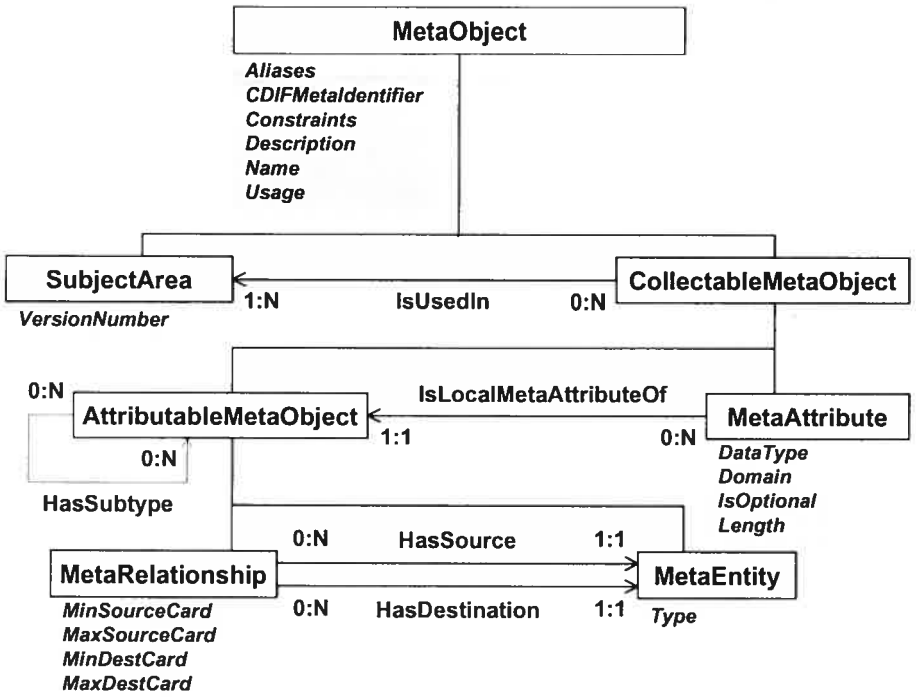


Figure 27 : Méta-métamodèle de CDIF

Évaluation de CDIF suivant les besoins pour M3

Comme la famille des GCs, CDIF ne traite pas explicitement des modèles et de leurs relations. CDIF ne définit pas de liens de conformité entre éléments et méta-éléments. Concernant la représentation des métarelations (*méta-liens*), CDIF autorise des métarelations binaires entre méta-entités mais ne permet pas de métarelations entre métarelations ou entre méta-entités avec métarelations (cf. le BesoinM3 1 - «Typage entre éléments de deux niveaux de modélisation consécutifs»). Voir la section I.6 (Annexe I) pour l'évaluation complète.

4.4 UML et MOF

4.4.1 UML

UML (Unified Modelling Language) est reconnu comme un standard de facto parmi les formalismes Orienté-Objet. Ses versions précédentes (versions 1.x), ne supportaient pas MDA (Model Driven Architecture). De plus, il ne répond pas à certains de nos besoins pour M2. Par exemple, comme elles traitaient le niveau des classes et celui des instances de façon indépendante, elles ne supportaient pas les besoins «Distinction entre différentes notions de conformité» (BesoinM2 4) et «Types relationnels entre types et/ou instances» (BesoinM2 15). Les nouvelles versions, UML 2.x (UML 2.0 et UML 2.1.1)

apporte des points nouveaux et suit l'approche de métamodélisation supportant MDA. Voici quelques documents de spécifications de UML 2.x: [109], [110], [111], [112], [113], [114], [115], [116], [117], [118], [119], [120], [121].

L'architecture du métamodèle UML est conçue selon les principes suivants (cf. page 11 – [113]):

- *La modularité* (forte cohésion et faible couplage) a pour but de regrouper des éléments constructeurs («constructs») dans des paquetages et d'organiser les propriétés dans des méta-classes.
- *L'organisation en couches* («layering») ou bien *la séparation des niveaux d'abstraction* s'effectue sur deux plans. Premièrement, la structure des paquetages est organisée en couches afin de séparer les constructeurs de noyau («core constructs») du métalangage avec les constructeurs de niveau supérieur qui les utilisent. Deuxièmement, l'architecture de modélisation (cf. la Figure 3, page 14) est appliquée afin de séparer clairement les choses (surtout relatives à l'instanciation) traversant des niveaux d'abstraction.
- *La partition* consiste à organiser des éléments dans un même niveau. Dans le métamodèle UML, celle-ci vise à augmenter la cohésion dans les paquetages et à diminuer le couplage entre paquetages.
- *L'extensibilité*. UML peut être étendu de deux façons: (i) on peut définir un nouveau dialecte de UML (en utilisant les `Profiles` pour personnaliser le langage pour des plateformes (exemples: J2EE/EJB, .NET/COM+) ou domaines (exemples: télécommunication, finance) particuliers); (ii) on peut spécifier un nouveau langage relatif à UML (en réutilisant des éléments du paquetage `InfrastructureLibrary` et en ajoutant des méta-classes et métarelations appropriées).
- *La réutilisation*. Une bibliothèque de métamodèles flexible et à grains fins est disponible. Ses éléments sont réutilisés pour définir le métamodèle UML et aussi réutilisés dans MOF (Meta Object Facility), CWM (Common Warehouse Model), etc.

Le métamodèle UML 2.x vise à modéliser les structures et les comportements des connaissances dans le champ du développement de logiciels. Il permet à l'usager de spécifier des modèles au niveau M1. La Figure 28 (cf. figure 7.4 - [113]) présente le

rapport sémantique entre UML et MOF, et la Figure 29 (cf. figure 7.8 - [113]) illustre l'architecture de modélisation dans UML.

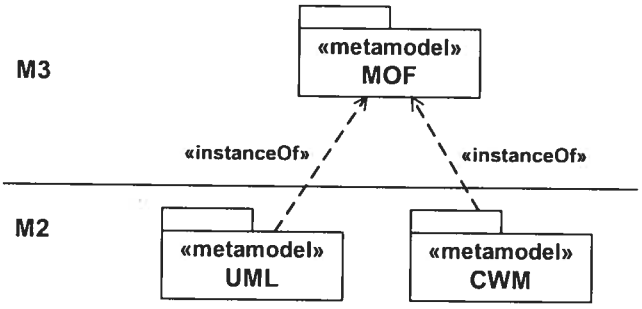


Figure 28 : MOF et UML aux différents niveaux de modélisation

Afin de voir comment UML 2.x peut répondre à nos besoins, nous présentons brièvement uniquement les diagrammes de base tels que les diagrammes de classes (`Classe`), classificateurs (`Classifier`), associations (`Association`), classes d'associations (`AssociationClasses`), instances (`Instance`), dépendances (`Dependency`), power-types (`PowerTypes`), espaces de noms (`Namespace`), paquetages (`Package`), composants (`Component`), modèles (`Model`). Les lecteurs sont invités à lire les documents ([113], [114]) pour une description complète du métamodèle UML 2.0 et les documents ([109], [110]) pour celle du métamodèle UML 2.1.1.

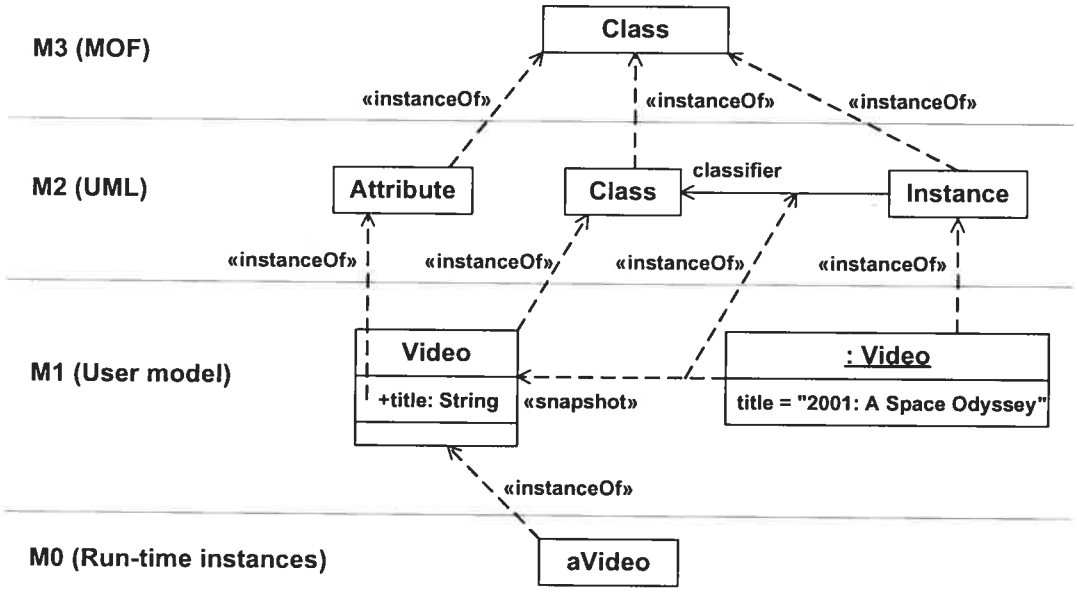


Figure 29 : Architecture de modélisation dans UML

Instances

La Figure 30 (cf. figure 7.8 - [114]) représente le diagramme des instances dans le paquetage `Kernel`.

Une spécification d'instance (`InstanceSpecification`) est un élément représentant une instance dans un système modélisé. Elle spécifie l'existence d'une entité dans le système et elle décrit complètement ou partiellement cette entité. Le type de l'instance dépend du type de son/ses classificateurs : un objet (*object*) pour une classe, un lien (*link*) pour une association, etc. L'entité se conforme à la spécification de chaque classificateur (`Classifier`) de la spécification d'instance, et a des caractéristiques structurelles dont les valeurs sont indiquées par chaque créneau (`Slot`) de la spécification d'instance (Figure 31). Une spécification d'instance peut représenter un instant (*snapshot*) d'une entité (dans le système modélisé). Donc une entité à différents instants peut être modélisée par des spécifications d'instance différentes. Une spécification d'instance peut être liée à plusieurs classificateurs, ce qui montre que la multi-classification est autorisée dans UML.

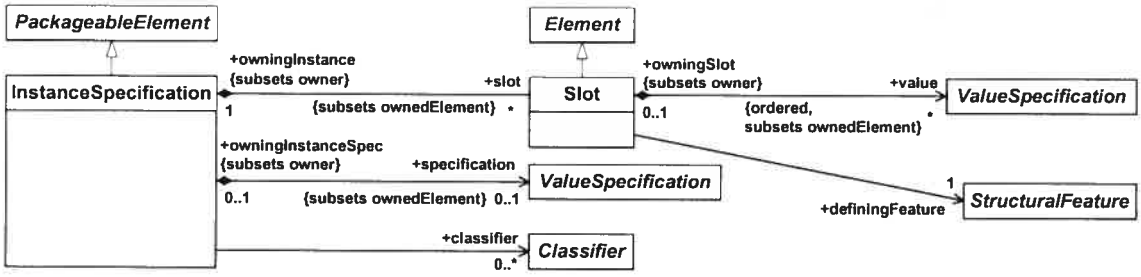


Figure 30 : Paquetage *Kernel* - Diagramme des instances

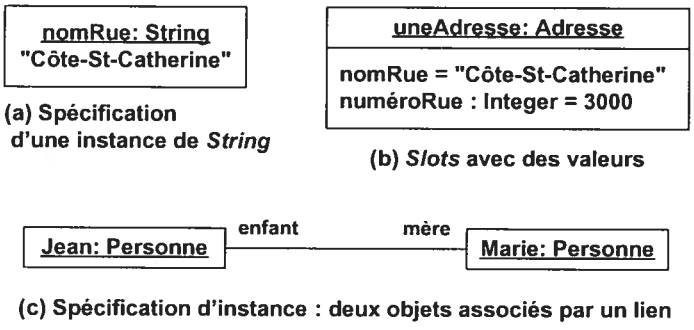


Figure 31 : Spécifications d'instance

Espace de noms

La Figure 32 (cf. figure 7.4 - [114]) représente le diagramme de `Namespace` dans le paquetage `Kernel`. Un `Namespace` représente un espace de nom, un élément nommé pouvant en détenir d'autres. Un élément nommé peut être détenu par, au maximum, un espace de nom. Un élément nommé est identifiable suivant son nom dans un espace de nom. Il est directement dans l'espace de nom ou introduit dans cet espace de nom par

d'autres mécanismes tels que l'importation ou l'héritage. `Namespace` est une méta-classe abstraite.

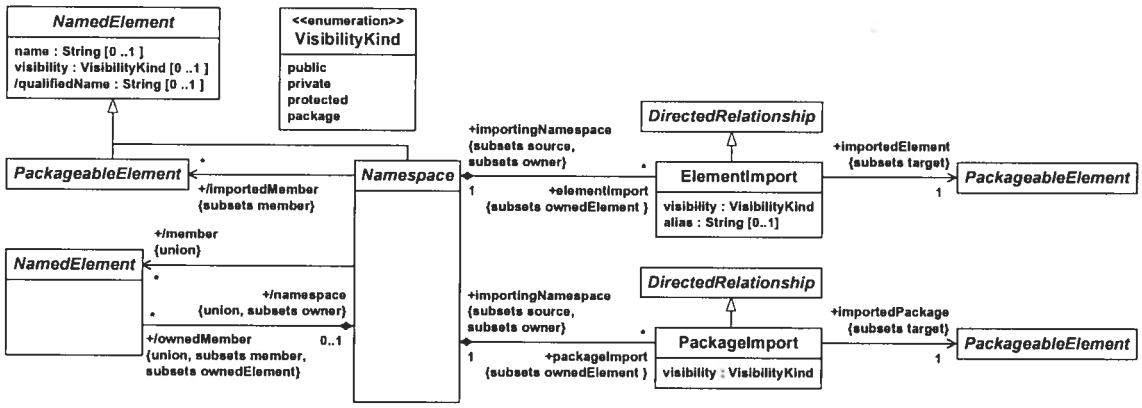


Figure 32 : Paquetage *Kernel* - Diagramme de *Namespace*

Classificateurs, Power-types

La Figure 33 (cf. figure 7.9 - [114]) représente le diagramme des classificateurs dans le paquetage `Kernel`.

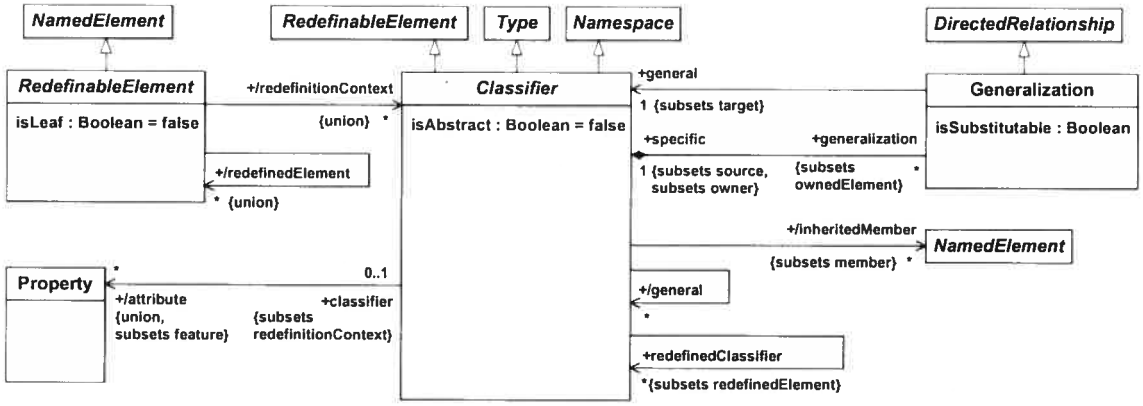


Figure 33 : Paquetage *Kernel* - Diagramme des classificateurs

`Classifier` est une méta-classe abstraite représentant l'ensemble des classificateurs. Un classificateur est une classification des instances, et décrit un ensemble d'instances ayant des propriétés en commun. Un classificateur est un espace de noms (`Namespace`) dont les membres peuvent comprendre des caractéristiques. Par exemple, une classe peut posséder des opérations et des attributs. Quand l'attribut `isAbstract` d'un classificateur prend la valeur *true*, ce classificateur ne fournit pas une déclaration complète, et peut typiquement ne pas être instanciable. Un classificateur abstrait est seulement utilisé par d'autres classificateurs, comme par exemple, la destination dans la relation de généralisation avec ces derniers. La valeur par défaut de

l'attribut `isAbstract` est *false*.

Les classificateurs peuvent s'organiser en hiérarchie de généralisation acyclique en se basant sur la relation de généralisation qui existe entre eux. Une instance d'un classificateur spécifique est également une instance indirecte de chacun des classificateurs supérieurs. Un classificateur est également redéfinissable. Un classificateur est aussi un type. Un type vise à contraindre des valeurs représentées par un élément typé. UML supporte la notion de «power-type» tant une classe dont les instances classifient les instances d'une autre classe.

Le contenu du paquetage des «power-types» (`PowerTypes`) est présenté dans la Figure 34 (cf. figure 7.18 - [114]). Chaque `Generalization` est une relation de généralisation binaire reliant un classificateur (`Classifier`) spécifique à un autre classificateur plus général. Chaque `GeneralizationSet` contient un ensemble de relations de généralisation (`Generalization`) qui décrit collectivement la manière de pouvoir diviser un classificateur (disons *`ClassifieurA`*) selon un autre classificateur (disons, *`ClassifieurB`*) qui est appelé un «power-type». Les instances de *`ClassifieurA`* qui correspondent à une même instance de *`ClassifieurB`* forment une sous-classe de *`ClassifieurA`*. C'est-à-dire les instances d'une classe qui correspondent à une même instance du «power-type» forment une sous-classe de la classe en question.

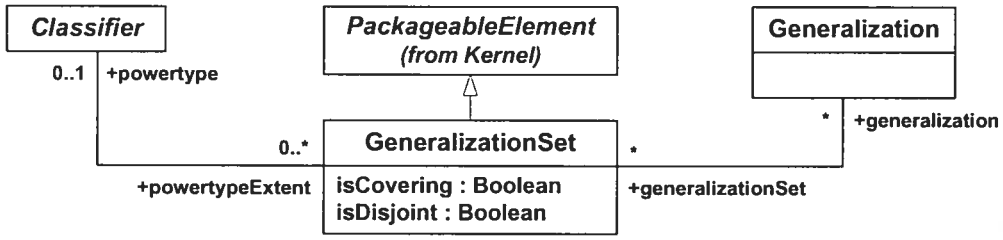


Figure 34 : Paquetage *PowerTypes*

En reprenant l'exemple de «power-types» dans la partition des politiques d'assurance présenté dans (section 7.3.22 – [114]), la Figure 35 illustre les partitions des politiques d'assurance (`PolitiqueAssurance`) selon les deux «power-types»: types de couverture (`TypePolitiqueCouverture`), et lignes d'assurance (`LigneAssurance`). Selon les types de couverture, il y a une partition de `PolitiqueAssurance` en deux sous-classes : les assurances collectives (`AssuranceCollective`) et les assurances individuelles (`AssuranceIndividuelle`). Pour chaque sous-classe, ses instances

correspondent à un même type de couverture qui est l'instance du «power-type» `TypePolitiqueCouverture`; ces instances sont considérées comme une instance de ce «power-type». De façon similaire, suivant le «power-type» `LigneAssurance`, l'on obtient une autre partition de `PolitiqueAssurance` en sous-classes : les assurances-vie (`AssuranceVie`), les assurances-santé (`AssuranceSanté`), les assurances-accidente (`AssuranceAccidente`), ou d'autres lignes d'assurance. De plus, les assurances-accidente (`AssuranceAccidente`) sont aussi divisibles en sous-classes telles que les assurances-automobile (`AssuranceAutomobile`), les assurances d'équipements (`AssuranceEquipment`), etc.

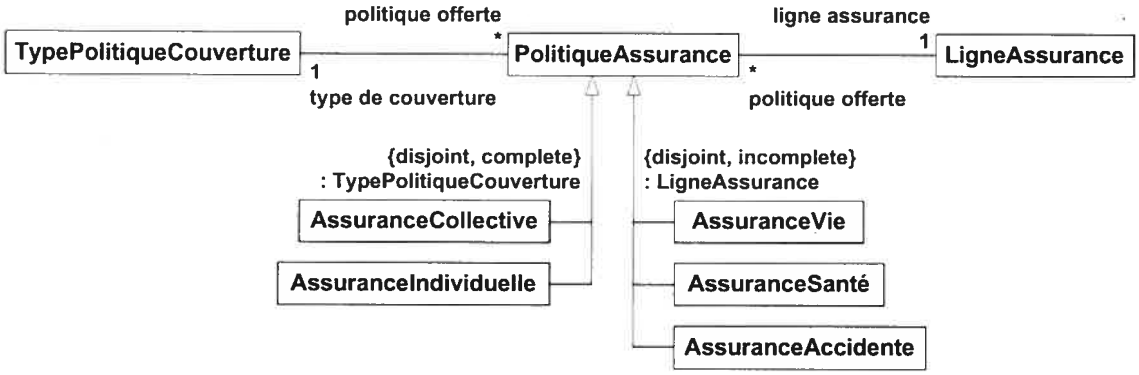


Figure 35 : Exemple de «power-types»

Classe, Association, Classe d'associations

La Figure 36 (cf. figure 7.12 - [114]) représente le diagramme des classes dans le paquetage `Kernel`.

Une classe (`Class`) décrit un ensemble d'objets qui partagent les mêmes caractéristiques et contraintes ainsi que la même sémantique. Une classe est un classificateur dont les caractéristiques sont des *attributs* (`Property`) ou des *opérations* (`Operation`). Un attribut d'une classe représente soit un attribut «classique» (possédé par la classe), soit une extrémité navigable d'une association binaire. Un objet d'une classe doit se conformer à cette classe. Ainsi, l'objet doit avoir des valeurs pour chaque attribut attaché à la classe en respectant les caractéristiques de cet attribut, par exemple, le type et la multiplicité pour l'attribut en question. Un attribut d'une classe peut être restreint à avoir une valeur particulière ou une valeur par défaut. Des opérations d'une classe permettent d'accéder au comportement défini dans cette classe et peuvent être invoquées sur un objet de la classe. Une invocation d'une opération peut faire changer

les valeurs des attributs de l'objet en question et/ou celles d'autres objets impliqués; elle peut aussi créer ou supprimer un ou des objets, etc.

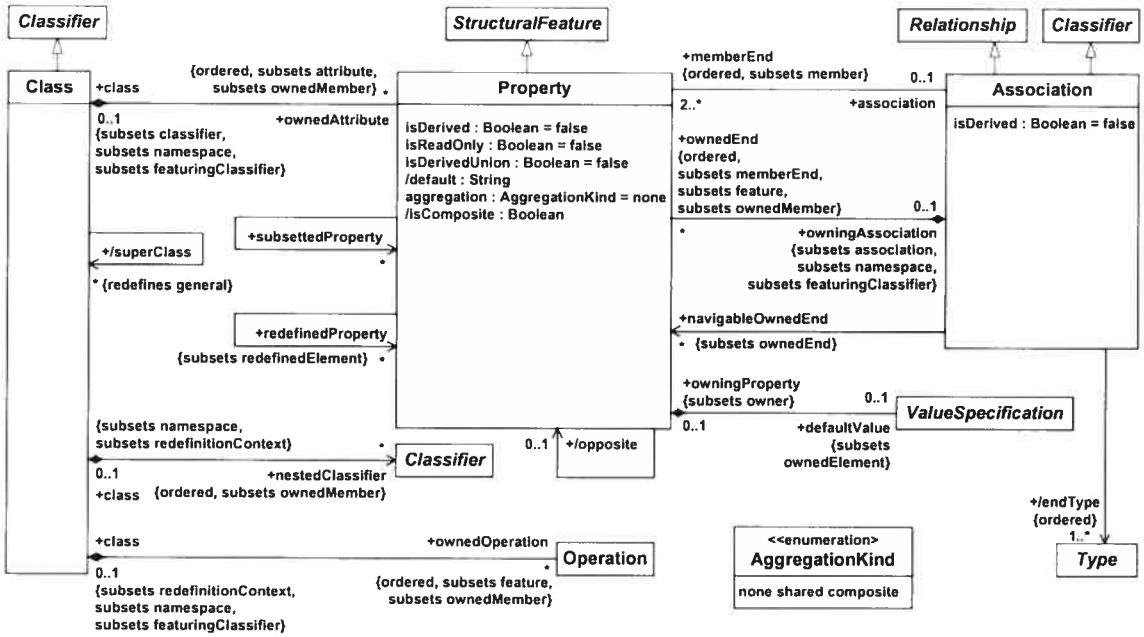


Figure 36 : Paquetage *Kernel* - Diagramme des classes

Une association (`Association`) spécifie une relation sémantique qui peut exister entre les instances des types associés. Une association a au moins deux extrémités. Toutes les extrémités sont représentées par des *attributs* (`Property`) dont chacun est relié au type de l'extrémité en question. Plusieurs extrémités de l'association peuvent avoir un même type. Pour une association binaire en particulier, l'attribut attaché à une extrémité navigable de l'association peut être représenté comme étant la propriété de la classe attachée à l'extrémité opposée. Par exemple, dans la Figure 37 (cf. les figures 7.59 et 7.60 - [114]), les attributs `achat` et `compte` attachés vis-à-vis aux extrémités navigables opposées de l'extrémité `Client` dans les associations de `Client` à `Achat` et de `Client` à `Compte` sont représentables comme attributs dont `Client` est propriétaire. On appelle un lien une instance d'une association qui est un n-uplet (*tuple*) des valeurs dont chacune a pour chaque extrémité et est une instance du type de cette extrémité. Comme pour les classes, la relation de généralisation entre des associations est autorisée.

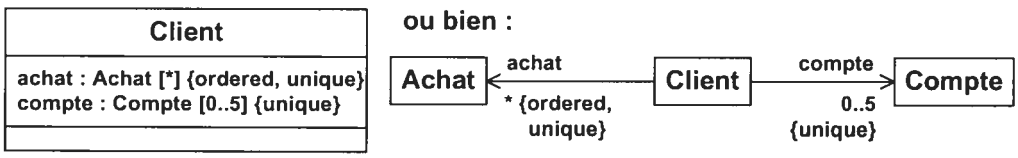


Figure 37 : Associations binaires représentées comme attributs

Les classes d'associations (`AssociationClass`) combinent des caractéristiques des classes (`Class` dans le paquetage `Kernel`) avec celles des associations (`Association` dans le paquetage `Kernel`). Une classe d'associations joue, d'une part, le rôle d'une association qui relie un ensemble de classes, et d'autre part, le rôle d'une classe ayant ses propriétés propres, dissociées de toute autre classe. La Figure 38 (cf. figure 7.25 - [114]) illustre que `Emploi` est une classe d'associations entre `Personne` et `Compagnie` et possède `salaire` comme attribut.

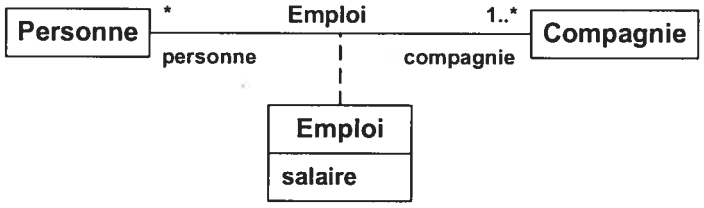


Figure 38 : Classe d'associations

Dépendances

La Figure 39 (cf. figure 7.15 - [114]) présente le contenu du paquetage des dépendances. Une dépendance (`Dependency`) est une relation indiquant que pour la spécification ou l'implémentation, un élément ou un ensemble d'éléments sources (client - `client`), requiert un autre élément ou un autre ensemble d'éléments destinations (fournisseur - `supplier`).

Un usage (`Usage`) est une relation de dépendance où le client nécessite de la présence du fournisseur pour son implémentation complète ou son opération.

Une abstraction (`Abstraction`) est une relation d'usage qui établit un rapport entre deux éléments ou ensembles d'éléments qui représentent le même concept aux différents niveaux d'abstraction ou selon différents points de vue. Les stéréotypes spécifiques pour `Abstraction`, prédéfinis dans les profils sont : «derive» (à spécifier que le ou les éléments clients peuvent être calculés à partir des éléments fournisseurs); «refine» (à relier des éléments entre différents niveaux sémantiques (par exemple le niveau d'analyse et le niveau de conception)); et «trace» (à relier des éléments dans différents modèles mais représentant le même concept).

Une réalisation (`Realization`) est une relation d'abstraction particulière dans laquelle le fournisseur représente la spécification et le client représente une implémentation.

Une substitution (`Substitution`) est une relation de réalisation entre deux

classificateurs (`Classifier`) qui spécifie que le classificateur source (`substitutingClassifier`) est conforme à la spécification du classificateur destination. Autrement dit, les instances du classificateur source peuvent remplacer, au moment de l'exécution, les instances du classificateur cible lorsque les instances du classificateur destination sont attendues.

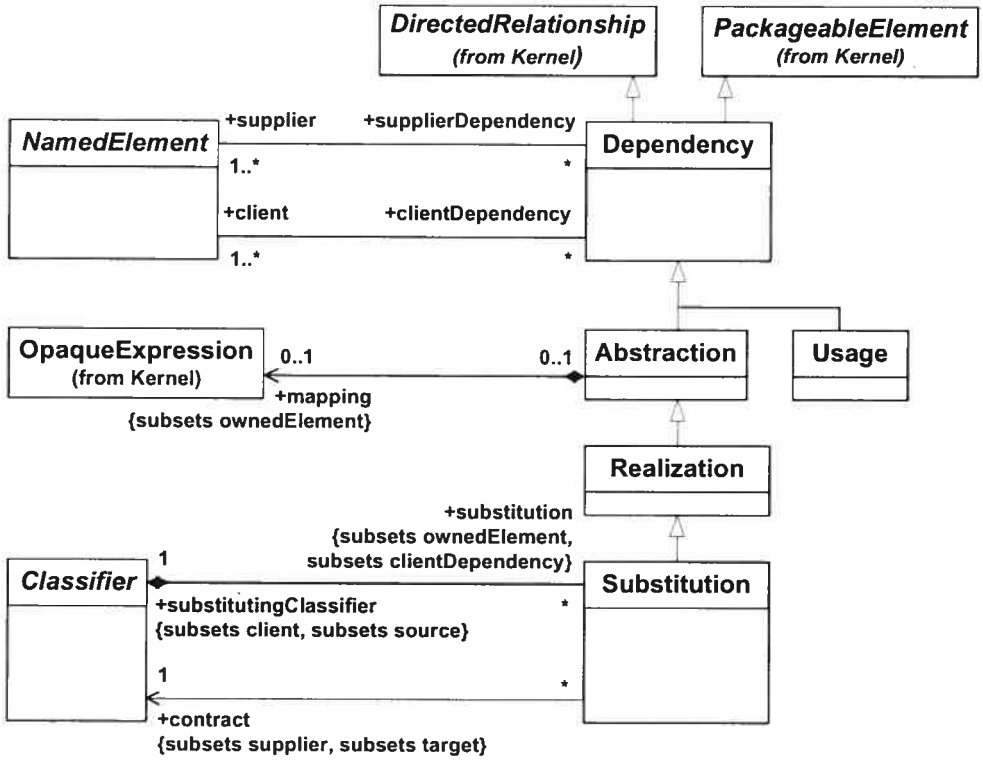


Figure 39 : Paquetage *Kernel* - Diagramme des dépendances

Modèle de rôles dans UML

UML distingue les rôles statiques et les rôles dynamiques. Les rôles statiques («*role names*») spécifient les fonctions des participants pour une relation donnée. Ils ressemblent aux rôles dans les modèles Entité-Association. Les rôles dynamiques spécifient les fonctions des participants dans une collaboration. Une collaboration décrit les interactions pertinentes entre les instances en identifiant les rôles spécifiques que ces instances joueront afin d'achever collectivement une fonctionnalité (comportement) d'un système. D'ailleurs, une collaboration dans UML2.0 est une sorte de classificateur, et peut avoir tout type de descriptions comportementales associées à un classificateur. Une instance (objet) donnée peut simultanément jouer des rôles dans une ou plusieurs collaborations différentes. Par ailleurs, une collaboration dans laquelle une instance joue un ou des rôles peut représenter uniquement les aspects (rôles) de l'instance qui sont

appropriés à la collaboration en question. Les rôles dans une collaboration seront souvent typés par des interfaces. Une interface peut être vue comme une projection des propriétés observables extérieurement d'un classificateur qui réalise cette interface. Les instances de différents classificateurs peuvent jouer un même rôle défini par une interface donnée, et plusieurs interfaces peuvent être réalisées par un même classificateur. Voir [114] pour plus de détails à ce sujet dans UML.

Paquetage, Composant, Modèle

Le diagramme de `Package` dans le paquetage `Kernel` est illustrée dans la Figure 40 (cf. figure 7.14 - [114]).

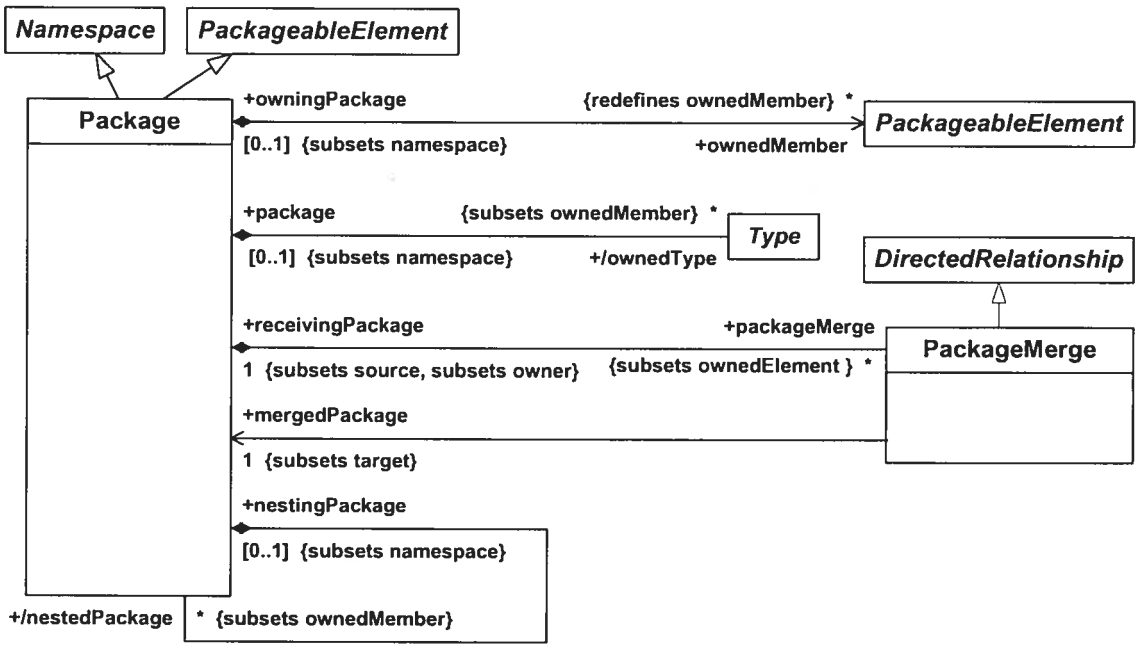


Figure 40 : Paquetage *Kernel* - Diagramme de *Package*

Un `Package` représente un paquetage qui est un espace de nom et aussi un élément empaquetable (`PackageableElement`). Un paquetage peut contenir d'autres paquetages. Seulement des éléments empaquetables peuvent être membres d'un paquetage. Un paquetage peut importer quelques ou tous les membres d'autres paquetages. Il peut aussi être fusionné avec d'autres paquetages. À chaque membre d'un paquetage peut être affecté un type de visibilité pour déterminer si ce membre est accessible ou non de l'extérieur du paquetage.

Une fusion de paquetages (`PackageMerge`) est une relation entre deux paquetages. Le contenu du paquetage destination est fusionné avec le contenu du paquetage source en appliquant les règles de spécialisation et de redéfinition indiquées. Tout contenu du

paquetage résultant de la fusion est celui du paquetage source.

Une importation de paquetage (`PackageImport`) est une relation entre deux paquetages. Elle indique que les éléments importés appartenant au paquetage destination, sont réutilisables dans le paquetage source mais qu'ils ne sont pas fusionnés avec les éléments de mêmes noms dans ce dernier.

Pour modéliser les systèmes, UML supporte aussi la notion de composants (`Component`) et celle de modèles (`Model`). La Figure 41 (cf. figures 8.2, 8.4, et 17.8 - [114]) illustre les diagrammes de composants et modèles.

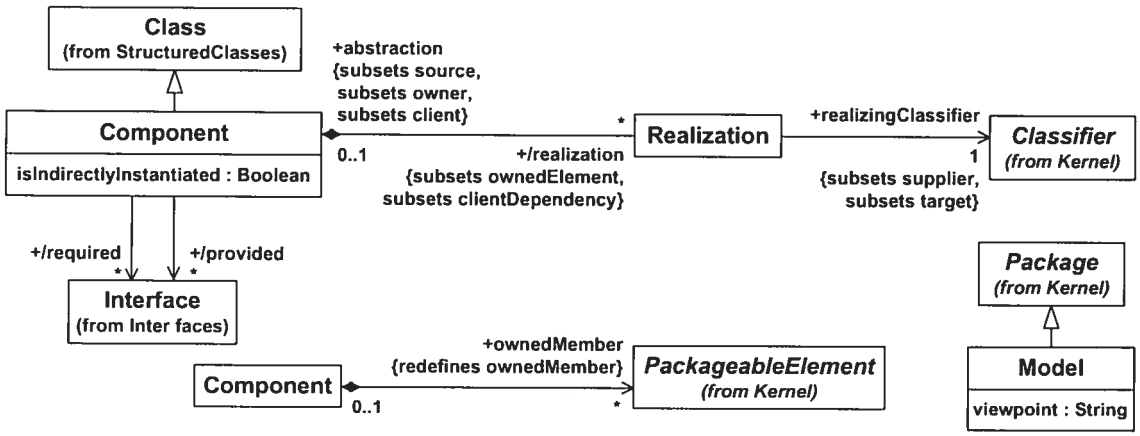


Figure 41 : Diagramme de *Component*, et Paquetage de *Model*

`Component` est un sous-type de `Class` et représente l'ensemble des composants. Un composant représente une unité modulaire d'un système. Un composant encapsule son contenu. Toute communication avec l'extérieur se fait grâce aux interfaces que le composant fournit et requiert. Un composant peut être utilisé comme un type (une classe). Il peut être réutilisé ou redéfini.

`Model` est un sous-type de `Package` et représente l'ensemble des modèles. Un modèle (`Model`) est une abstraction d'un système physique. Il est défini comme un paquetage et il contient un ensemble d'éléments (possiblement hiérarchisés) qui, ensemble, décrivent le système physique à modéliser. Un modèle peut aussi contenir des éléments représentant l'environnement du système, y compris les acteurs, et leurs interrelations comme les associations et les dépendances. Un système peut être défini par plusieurs modèles sous différentes perspectives (concepteur, analyste, utilisateur, etc.) et aux différents niveaux d'abstraction (analyse, conception, implémentation). Les relations entre les éléments de différents modèles sont utiles pour tracer le raffinement et les dépendances entre ces modèles.

Évaluation de UML suivant les besoins pour M2

Au niveau M1, l'usager peut définir des éléments qui sont instanciés par des méta-classes non abstraites que le métamodèle UML fournit, telles que `Class`, `Association`, `AssociationClass`, `Property`, `InstanceSpecification`, `Package`, etc. Donc, l'usager peut représenter des classes, des types relationnels ainsi que des instances; par exemple, le type relationnel entre les classes `Personne` et `Voiture`, la classe `Personne` et son instance `Jean:Personne` (Figure 42).

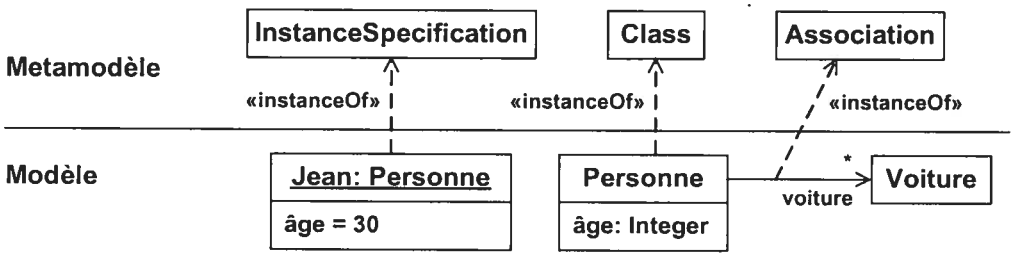


Figure 42 : Définition de classes, relations, instances au niveau M1

Cependant, UML2.0 ne remplit pas divers besoins pour M2 dont en voici quelques uns. Dans UML2.0, le rapport de conformité entre éléments et méta-éléments et celui entre modèles et métamodèles ne sont pas distingués (cf. BesoinM2 4 - «Distinction entre différentes notions de conformité») et ils sont représentés par les liens `instanceOf`. Comme divers autres formalismes, UML ne distingue pas la classification statique de la classification dynamique (cf. BesoinM2 9 - «Classification dynamique et Classification statique»); il ne distingue pas non plus les différents types de contraintes structurelles tels que ceux présentés dans les besoins «Contraintes de cardinalité totale min/max pour un type relationnel» (BesoinM2 11), «Contraintes de l'itération min/max pour un type relationnel» (BesoinM2 13). Il existe deux moyens dans UML2.0 pouvant supporter certains cas correspondant au besoin «Types relationnels entre types et/ou instances» (BesoinM2 15): (i) des valeurs par défaut pour des attributs d'une classe; (ii) le «power-type». Cependant, ces deux moyens sont inapplicables pour certains cas comme celui dans le Problème 2 (page 37) où la partition d'un classificateur se fait en fonction de plusieurs «power-types» combinés, conformément à un type relationnel n-aires entre le classificateur avec ces «power-types». Concernant la représentation des modèles et de leurs liens, UML n'explicite pas notre notion de modèles de conditions (cf. le BesoinM2 16 - «Distinction entre modèles de différentes natures»); UML ne permet pas de représenter différents types de liens entre modèles comme l'intersection,

la différence, l'inférence, la restriction (cf. le BesoinM2 17 - «Liens avec les modèles»). La section I.7 (Annexe I) présente l'évaluation complète de UML.

4.4.2 MOF

MOF (Meta Object Facility) a pour but de définir un langage de modélisation unique permettant de représenter des modèles et des métamodèles. MOF est utilisé pour sa propre définition ainsi que dans plusieurs technologies standardisées par OMG comme UML, CWM, XMI (cf. la Figure 28, page 59). Contrairement aux versions précédentes, MOF2.0 ([107], [106], [105]) suit MDA. La collaboration entre «UML2.0 Infrastructure» et MOF2.0 vise une meilleure réutilisation et l'intégration des concepts de modélisation afin de fournir une fondation pour MDA. La Figure 43 (cf. figure 7.1 ou bien 14.1 - [105]) montre que MOF importe le paquetage *Core* de UML et qu'il étend celui-ci avec des paquetages additionnels. Cette figure montre aussi les rapports entre les paquetages dans MOF.

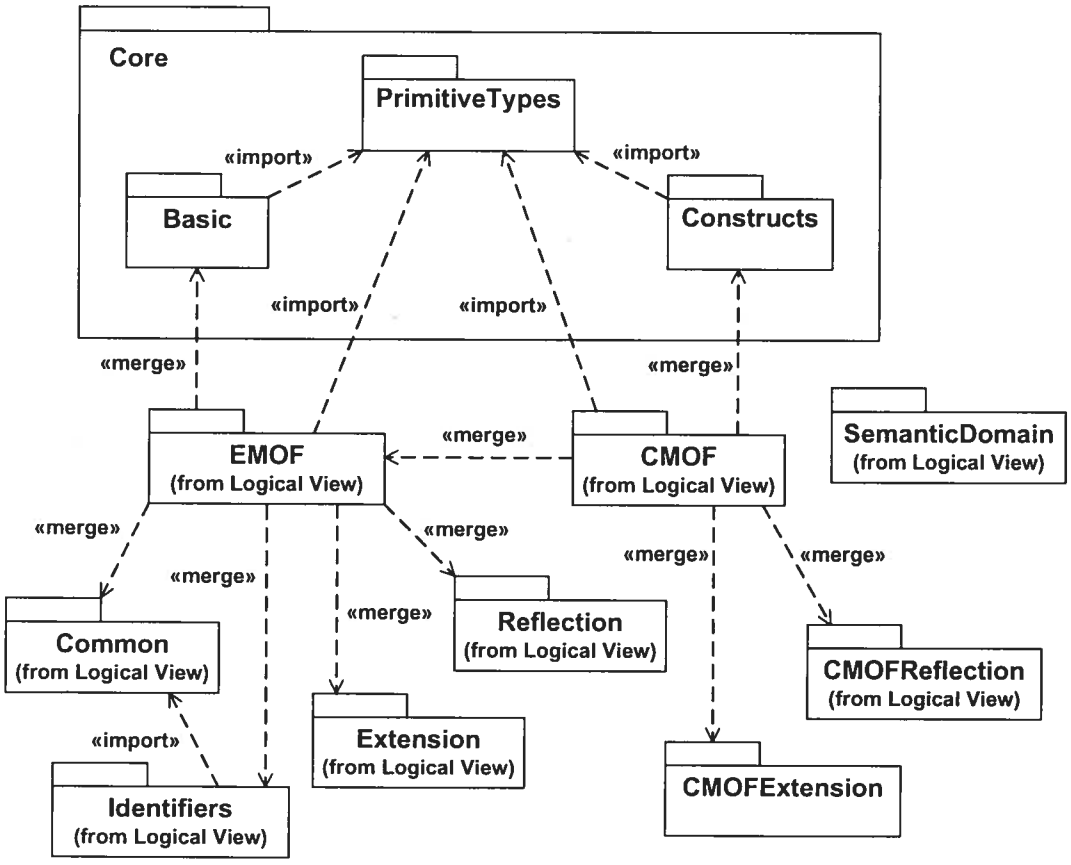


Figure 43 : Réutilisation du noyau de UML2.0 dans MOF

MOF fait usage de certains concepts de base («building-block concepts») pouvant

être réutilisés dans d'autres modèles et métamodèles. Ces concepts sont répartis dans les trois paquetages concernant la réflexivité (`Reflection`), les identificateurs (`Identifiers`) et l'extension (`Extension`).

La Figure 44 (cf. figure 9.1 – [105]) représente le contenu du paquetage `Reflection` concernant la réflexivité. La réflexivité vise à étendre un modèle avec la capacité de s'auto-définir. Ceci permet de manipuler des objets (`Object`) sans connaître au préalable leurs caractéristiques. `Object` représente l'ensemble de tous les éléments et valeurs de données. `Object` est équivalent à `java.lang.Object` dans Java. La classe `Element` est la superclasse de tous les éléments dans MOF et aussi la superclasse de toutes les instances des éléments de MOF. Tout élément (`Element`) a une classe (`Class`) à son niveau méta, dite *méta-classe* qui décrit ses propriétés et ses opérations. L'opération `getMetaClass()` de l'élément retourne la méta-classe (`Class`) de celui-ci. `Element` fusionne et étend `Basic::Element`. Tous les éléments qui spécifient `Reflection::Element` (exemples: tous les éléments dans «UML2.0 Infrastructure») héritent la réflexivité de celui-ci. Chaque élément étant l'instance de la classe `Factory` dans MOF crée des instances des types dans un paquetage (`Package`) auquel il est attaché. Un `Element` peut être créé d'un `Factory`.

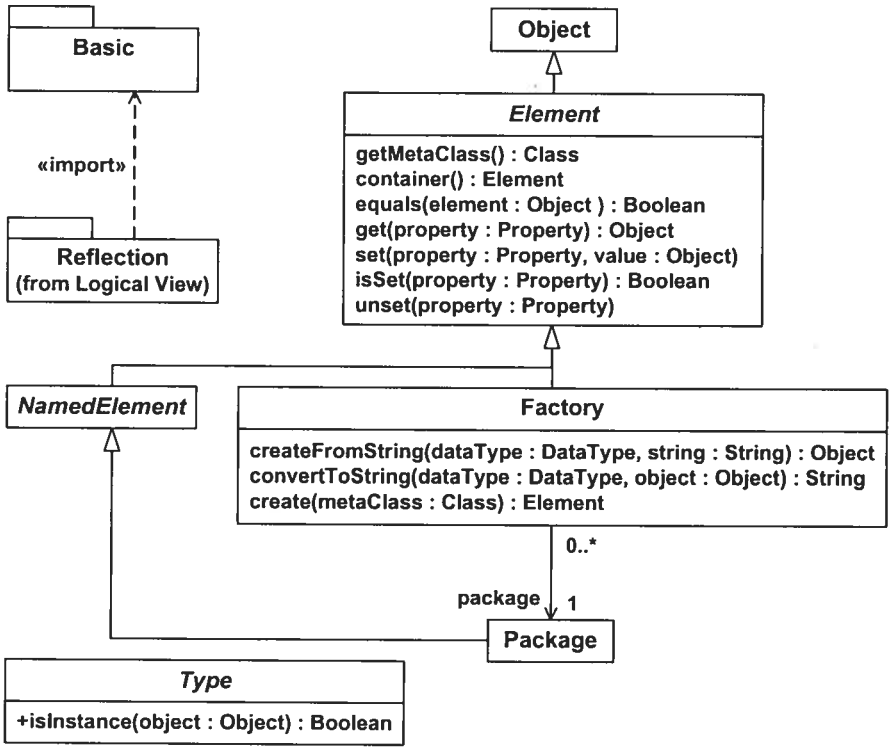


Figure 44 : Paquetage *Reflection*

MOF2.0 définit aussi `ReflectiveCollection` et `ReflectiveSequence` pour représenter respectivement les collections et les séquences réflexives (Figure 45 (cf. figure 10.2 [105])).

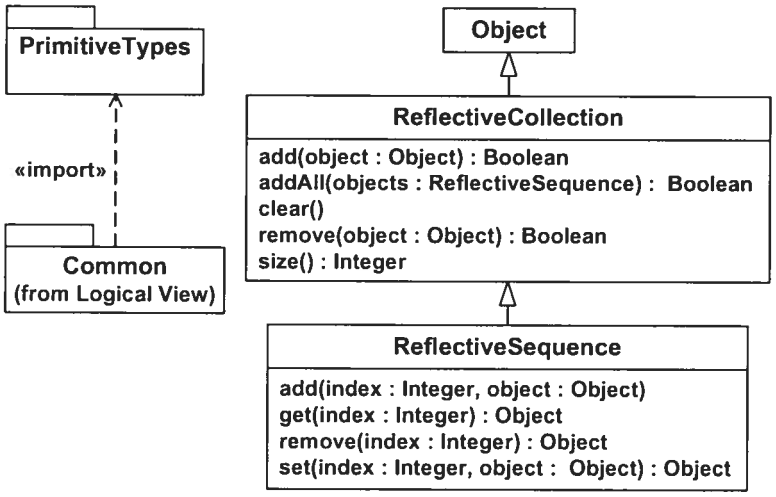


Figure 45 : Paquetage *Common*

Une collection réflexive (`ReflectiveCollection`) est une classe réflexive utilisée pour accéder aux propriétés qui peuvent avoir plusieurs valeurs possibles. Aussi, une séquence réflexive (`ReflectiveSequence`) est une collection réflexive utilisée pour accéder aux propriétés ordonnées. Les classes `ReflectiveCollection` et `ReflectiveSequence` sont définies dans le paquetage `MOF::Common` en vue de faciliter la réutilisation dans la représentation d'autres capacités de MOF.

Les identificateurs (`Identifiers`) servent à identifier uniquement les objets de métamodèles indépendamment des données de modèles qui peuvent être changées. La Figure 46 (cf. figure 10.1 [105]) représente le paquetage `Identifiers`. Une étendue (`Extent`) est un contexte dans lequel un élément (`Element`), au sein d'un ensemble d'éléments, dispose d'un identificateur lui permettant, sans ambigüité, de se distinguer des autres éléments, tout en lui permettant de s'identifier de manière indépendante de ses données. Un élément peut n'être membre d'aucune étendue, ou membre d'une ou de plusieurs étendues. `Identifiers::Package` étend `Basic::Package` avec l'attribut `uri` qui peut être utilisé comme un identificateur externe pour un paquetage. `Identifiers::Property` étend `Basic::Property` avec la capacité de désigner une propriété lui permettant d'être un identificateur pour l'élément qui le contient.

MOF dispose de la capacité de définir les éléments de métamodèles comme classes

ayant des propriétés et des opérations. Pourtant, il est parfois nécessaire d'annoter dynamiquement des éléments de modèles avec l'information additionnelle et peut-être imprévue. Cette information peut inclure l'information manquante d'un modèle ou des données exigées par un outil particulier. En vue de répondre à cette nécessité, le paquetage `Extension` de MOF propose un mécanisme simple pour associer une collection de paires nom-valeur aux éléments de modèles. Le contenu de ce paquetage est illustré dans la Figure 47 (cf. figure 11.1 - [105]). Une paire nom-valeur est codée par une étiquette (`Tag`). Une étiquette peut être associée à plusieurs éléments et un élément peut s'associer à plusieurs étiquettes. Cependant un élément ne peut pas avoir plus d'une étiquette du même nom. D'ailleurs, MOF ne précise pas le mode d'établissement des étiquettes pour un élément.

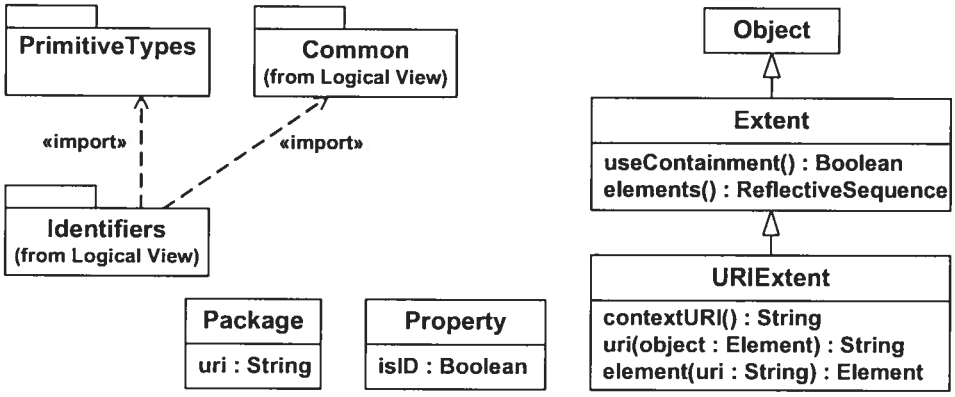


Figure 46 : Paquetage *Identifiers*

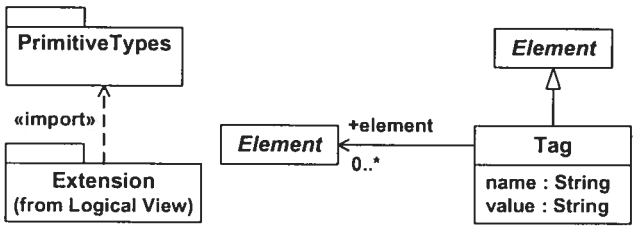


Figure 47 : Paquetage *Extension*

Ces trois paquetages, `Reflection`, `Identifiers` et `Extension`, sont réutilisés (par le mécanisme de fusion) dans EMOF (Essential MOF), CMOF (Complete MOF) ou autres métamodèles. EMOF vise à fournir un cadre (*framework*) pour mettre en correspondance les modèles MOF et les implémentations telles que XMI (XML Metadata Interchange) et JMI (Java Metadata Interchange) pour les métamodèles simples. EMOF dispose d'un ensemble minimum d'éléments nécessaires pour modéliser des systèmes orienté-objets. EMOF fusionne le paquetage `Core::Basic` dans UML2.0

avec les paquetages `Common`, `Reflection`, `Identifiers`, et `Extension` concernant les capacités de MOF (cf. la Figure 43, page 70). EMOF réutilise sans extension le paquetage `Basic` de «UML2.0 Infrastructure» [113] mais y ajoute quelques contraintes. Quant à CMOF (cf. la Figure 43, page 70), il est construit à partir de EMOF et du paquetage `Core::Constructs` dans «UML2.0 Infrastructure» [113]. EMOF est utilisé pour définir les métamodèles plutôt simples alors que CMOF est utilisé pour spécifier les métamodèles plus sophistiqués tels que UML 2.0. Ainsi, nous retrouvons dans MOF les éléments de UML 2.0 tels que les espaces de noms (`Namespasces`), classificateurs (`Classifier`), classes (`Class`), types relationnels (`Relationship`), association (`Association`), attributs (`Property`), opérations (`Operation`), commentaires (`Comments`), contraintes (`Contraints`), etc. La sémantique de ces derniers ne change pas. Par exemple, la Figure 48 (cf. figure 14.2 - [105]) décrit le diagramme de classes réutilisées de `Core::Constructs` dans UML 2.0.

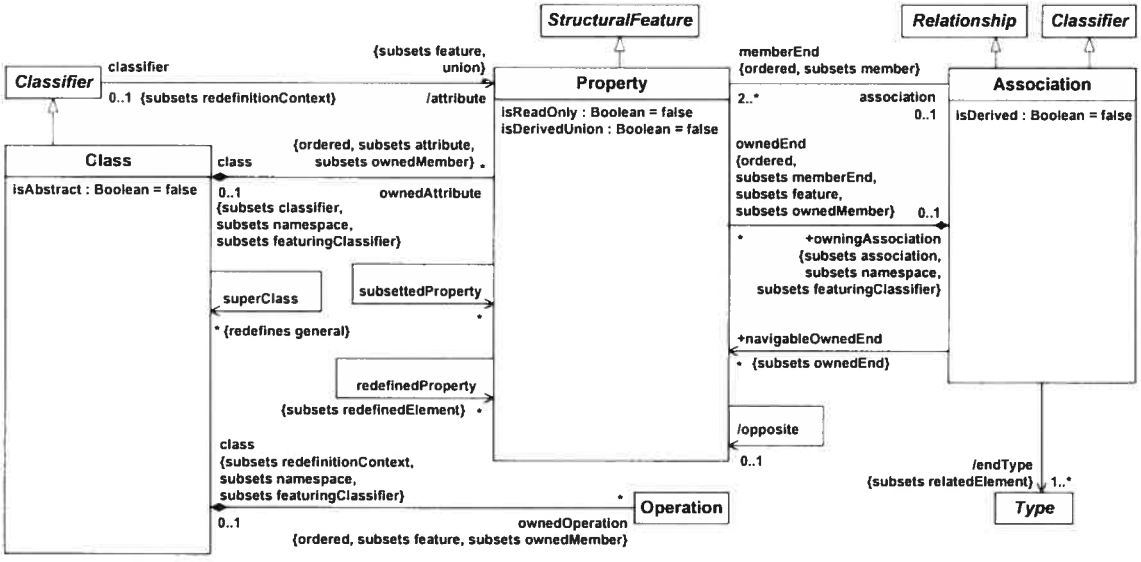


Figure 48 : CMOF - Classes concrètes réutilisées de *Core::Constructs* dans UML 2.0

La Figure 49 (cf. figures 13.1 et 13.2 – [105]) décrit le contenu du paquetage `CMOFReflection` représentant la capacité de réflexivité de CMOF. `CMOFReflection` fusionne les nouvelles opérations dans les classes existantes `Object`, `Extent`, et `Factory`. De plus il ajoute la classe `Link` et le type de données `Argument`. `Link` est une nouvelle classe représentant l'ensemble des instances des `Association`. Autrement dit, un lien (`Link`) représente une instance d'une association (`Association`), tout comme un élément (`Element`) représente une instance d'une

classe (`Class`). Une association (`Association`) dans MOF peut avoir deux extrémités ou plus, c'est-à-dire qu'elle peut être une association n-aire (cf. Figure 48). Pourtant, au niveau de l'implémentation, MOF ne traite que des associations binaires. Effectivement, selon [107], un lien (`Link`) ne relie que deux objets (`Object`), mais la classe `Link` n'est pas un sous-type de la classe `Object`, c'est-à-dire qu'un lien n'est pas vu comme un objet. Comme illustré dans la Figure 49, selon ([106], [105]), un lien (`Link`) ne relie que deux éléments (`Element`). Cependant, la classe `Link` n'est pas un sous-type de la classe `Element`, ou bien un lien n'est pas vu comme un élément. Le nouveau type de données `Argument` sert à représenter les arguments nommés pour les opérations réflexives et ouvertes/modifiables (*open-ended reflective operations*). La valeur d'un argument (`Argument`) est un objet (`Object`), c'est-à-dire qu'elle peut être un élément (`Element`) ou une valeur de données.

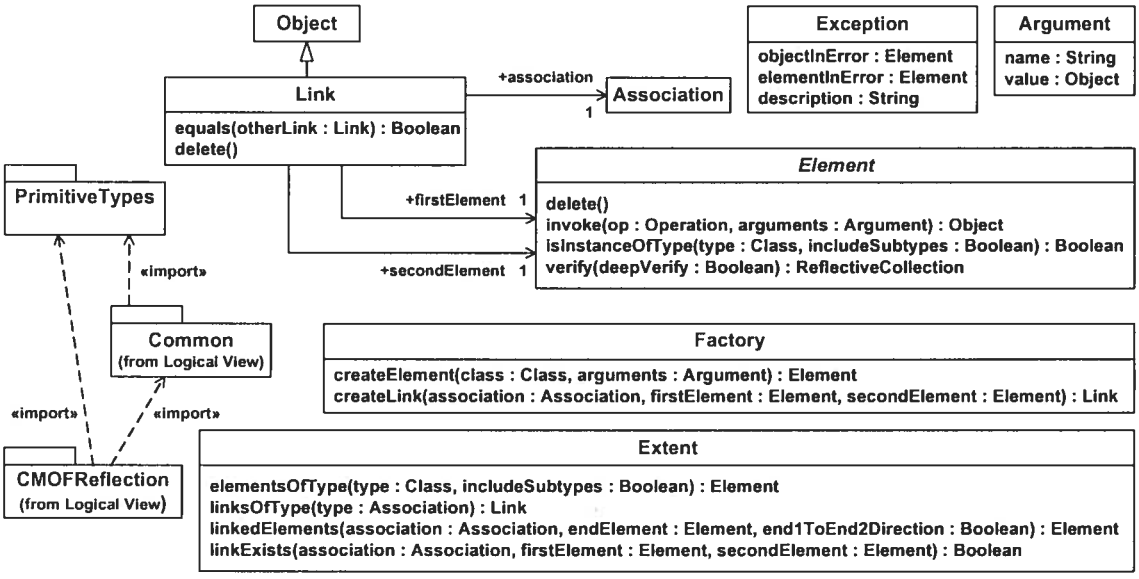


Figure 49 : Paquetage *CMOFReflection*

Et CMOF étend le paquetage `Extension` en spécifiant le nom de rôles `tag` pour l'extrémité non navigable de l'association entre `Tag` et `Element` (Figure 50 (cf. figure 14.3 – [105])).

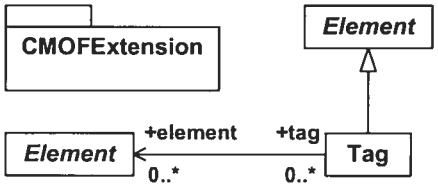


Figure 50 : Paquetage *CMOFExtension*

Évaluation suivant les besoins pour M3

Par rapport aux versions précédentes de MOF (exemple: MOF 1.4 [108]), la version 2.0 apporte plusieurs changements en tenant compte du guide MDA. MOF est le plus complet parmi les noyaux réflexibles. Pourtant, il ne remplit pas tous les besoins importants pour M3. Effectivement, MOF supporte différentes notions de base mais il ne couvre pas toutes les structures possibles pour un lien binaire (cf. le BesoinM3 1 - «Typage entre éléments de deux niveaux de modélisation consécutifs»). MOF n'explicite pas notre notion de modèles de conditions (cf. le BesoinM3 6 - «Distinction entre modèles de différentes natures»). Il ne fait pas de distinction entre la relation de conformité d'un élément à un méta-élément et la relation de conformité d'un modèle à un métamodèle. Finalement, il ne permet pas de représenter différents types de liens entre modèles comme l'intersection, la différence, l'inférence, la restriction (cf. le BesoinM3 7 - «Liens avec les modèles») pour la gestion de modèles.

4.5 Formalismes utilisés dans le Web

Comme nous l'avons expliqué à propos des applications Web (page 43), nous avons retenu XML-XML Schema, RDF-RDFS et OWL.

4.5.1 XML-XML Schema

XML (Extensible Markup Language) ([165], [168]) est un langage de balisage («markup») largement utilisé sur le Web. Il permet la représentation structurée d'informations dans un format texte où l'information représentée est encadrée par des balises. En XML, les balises sont redéfinissables. Il est donc possible d'adapter le vocabulaire de description au domaine spécifique du document. La définition des balises et leurs relations sont spécifiées dans une DTD (Document Type Description) ou dans un XML Schema. Concernant la représentation de modèles (schémas), XML Schema est plus expressif et utile que DTD [173]. Pour ces raisons, nous parlerons de XML Schema plutôt que de DTDs.

La Figure 51 présente un document XML représentant qu'une personne travaille pour une compagnie. Les informations sur la personne sont l'identité (id), le nom, le prénom et l'adresse. Les informations sur la compagnie sont l'identité (id), le nom et l'adresse. Le schéma définissant ce document XML est représenté dans la Figure 52 et nommé

exXMLSchema.xsd sur le site http://www.exemple.org/test/exXMLSchema.

```
<?xml version="1.0"?>
<cible:Personne
        xmlns:cible="http://www.exemple.org/test/exXMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        id="toto">
  <nom>Tremplay</nom>
  <prenom>Jean</prenom>
  <Adresse pays="Canada">
          <numero>4671</numero>
          <!– et ainsi de suite ... -->
  </Adresse>
  <travaille-pour id="Teximus">
          <nom>Teximus Inc.</nom>
          <Adresse xsi:type="cible:TypeAdresse-Canada">
                  <numero>33</numero>
                  <!-- and so on ... -->
          </Adresse>
  </travaille-pour>
</cible:Personne>
```

Figure 51 : XML – Exemple

```
<xs:schema
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.exemple.org/test/exXMLSchema">
  <xs:element   name="Adresse"  type="TypeAdresse"/>
  <xs:element   name="Personne"  type="Personne"/>
  <xs:element   name="nom"  type="xs:string"/>
  <xs:attribute   name="id"  type="xs:NMTOKEN"/>
  <xs:complexType   name="TypeAdresse">
    <xs:sequence>
      <xs:element   name="numero"  type="xs:string"/>
      <!– et ainsi de suite ... -->
    </xs:sequence>
    <xs:attribute   name="pays"  type="xs:NMTOKEN"/>
  </xs:complexType>
  <xs:complexType   name="TypeAdresse-Canada">
    <xs:complexContent>
```

```
14.       <xs:restriction   base="TypeAdresse">
15.         <xs:attribute   name="pays"  type="xs:NMTOKEN"
                            fixed="Canada"/>
16.       </xs:restriction>
17.     </xs:complexContent>
18.   </xs:complexType>

19.   <xs:complexType   name="Personne">
20.     <xs:sequence>
21.       <xs:element   ref="nom"/>
22.       <xs:element   name="prenom"  type="xs:string"/>
23.       <xs:element   ref="Adresse"/>
24.       <xs:element   name="travaille-pour"  type="Compagnie"
                        minOccurs="0"/>
25.     </xs:sequence>
26.     <xs:attribute   ref="id"/>
27.   </xs:complexType>

28.   <xs:complexType   name="Compagnie">
29.     <xs:sequence>
30.       <xs:element   ref="nom"/>
31.       <xs:element   ref="Adresse"/>
32.     </xs:sequence>
33.     <xs:attribute   ref="id"/>
34.   </xs:complexType>

  </xs:schema>
```

Figure 52 : XML Schema - Exemple d'un schéma de la Figure 51

XML Schema ([166], [167], [169], [170], [171], [172]) définit le vocabulaire et la grammaire de XML. Il dispose des notions : schéma (`schema`), type complexe (`complexType`), type simple (`simpleType`), élément (`element`), attribut (`attribute`), contenu simple (`simpleContent`) ou complexe (`complexContent`), etc. Un type complexe peut avoir des attributs et/ou contenir d'autres éléments. Pour sa part, un type simple ne peut pas être composé d'autres éléments, ni avoir d'attribut. Les valeurs des attributs `minOccurs` et `maxOccurs` d'un élément (attribut) contraignent le nombre d'occurrences de cet élément (attribut) dans un type complexe. Un élément ou attribut peut être typé ou non. Dans le cas où il est typé, il appartient à un type au maximum. Le type d'un élément peut être de type simple ou complexe mais celui d'un

attribut doit être de type simple. Une déclaration d'un élément est une association d'un nom avec une définition de type (simple ou complexe), (optionnellement) une valeur par défaut, et un ensemble (possiblement vide) de définitions de contraintes d'identité (en rapport avec l'unicité et le référencement relativement au contenu de plusieurs éléments et attributs). La déclaration d'un attribut est une association entre un nom et une définition d'un type simple, complétée d'une information d'occurrences, et optionnellement d'une valeur par défaut. La portée d'un élément ou attribut déclaré peut être soit globale soit locale à la définition du type complexe qui le contient. L'attribut `type` est utilisé pour identifier le type d'un élément ou attribut. L'attribut `fixed` est utilisé dans la déclaration d'éléments et d'attributs pour affecter les valeurs particulières (chaînes de caractères) à ces éléments et ces attributs. Par exemple, l'attribut `pays` du type `TypeAdresse-Canada` est toujours `Canada` comme une valeur fixe (cf. la Figure 52, page 78). L'élément `simpleContent` indique que le modèle de contenu d'un nouveau type contient uniquement des chaînes de caractères et aucun élément. L'élément `complexContent` signale la restriction ou l'extension du modèle de contenu d'un type complexe.

XML Schema supporte aussi les mécanismes pour l'extension (`extension`) et la restriction (`restriction`) des types. Soient *A* et *B* deux types. Si *A* étend (`extension`) *B*, *A* hérite des éléments et des attributs de *B*. Si *A* est plus spécifique (`restriction`) que *B*, *A* hérite aussi des éléments et des attributs de *B* mais les valeurs possibles de ces éléments et de ces attributs pour *A* sont plus spécifiques que celles pour *B*. XML Schema définit les éléments `include` et `import` vis-à-vis pour l'extension entre schémas d'un même «espace de nom cible» (`targetNamespace`) et celle entre schémas de différents espaces de nom cibles. Dans ces cas, les éléments définis dans le schéma initial sont réutilisés sans modification dans le schéma étendu. En outre, le mécanisme `redefine` permet à un schéma de redéfinir la structure de certains types simples/complexes, groupes, groupes d'attributs définis auparavant dans un autre schéma ayant un même espace de nom cible que le schéma en question ou n'ayant pas d'espace de nom.

Il est à noter que dans XML Schema, deux choses considérées comme distinctes peuvent avoir un même nom sans entraîner aucun conflit ([169], [170]). Par exemple, dans le schéma décrit dans la Figure 52 (page 78), l'élément `Personne` (déclaré à la ligne 2 de la figure) prend pour son type le type complexe `Personne` (déclaré à la ligne

19 de la figure).

Il y a plusieurs façons dans XML – XML Schema pour représenter une même chose. Par exemple, le rapport de `travaille-pour` entre une personne (`Personne`) et une compagnie (`Compagnie`) peut être codé au moins de trois façons différentes : (i) `travaille-pour` est un élément dans le contenu du type `Personne` et prend `Compagnie` pour son type (cf. la Figure 51, page 77 et la Figure 52, page 78); ou (ii) `Personne` et `Compagnie` font partie du contenu de `travaille-pour`; ou (iii) `travaille-pour` est un élément de `Personne`, et `Compagnie` est un élément de `travaille-pour`. Ceci montre d'un part la souplesse de XML – XML Schema mais signale, d'autre part, l'ambiguïté sémantique dans la représentation des choses, ce qui rend difficile le traitement sémantique des informations.

Noyau réflexif de XML Schema

Nous trouvons dans ([166], [170], [171]) comment les éléments de XML Schema sont définis par les éléments de XML Schema eux-mêmes. C'est-à-dire le schéma XML Schema du vocabulaire XML Schema est le métamodèle de lui-même. Dans ce cas, XML Schema est pratiquement placé au niveau M2 dans l'architecture de modélisation et par ailleurs au niveau M3 dans cette architecture. Ceci montre que la séparation des niveaux de modélisation dans XML Schema manque de clarté.

La Figure 53 décrit une partie du schéma définissant le vocabulaire XML Schema ([166], [167], [170], [171]). Elle illustre la façon dont sont déclarés les éléments de niveau supérieur, dits les éléments *top-level*. Il s'agit des éléments suivants: `element` (`xs:element`), `attribute` (`xs:attribute`), `complexType` (`xs:complexType`), et `simpleType` (`xs:simpleType`). Les déclarations de ces éléments sont signalées par l'élément *top-level* `element` (`xs:element`). Les occurrences des éléments *top-level* seront définies conformément aux types indiqués par les attributs `type` dans leurs déclarations. Il s'agit des types: `topLevelElement` (`xs:topLevelElement`), `topLevelAttribute` (`xs:topLevelAttribute`), `topLevelComplexType` (`xs:topLevelComplexType`),`topLevelSimpleType` (`xs:topLevelSimpleType`). Ce sont des types complexes et leurs déclarations sont signalées par l'élément *top-level* `complexType` (`xs:complexType`) mentionné.

```
...
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
```

```
  blockDefault="#all"
   ...
>
  ...
  <xs:element name="element" type="xs:topLevelElement" id="element">
    <xs:annotation>
      <xs:documentation
        source="http://www.w3.org/TR/...#element-element"/>
    </xs:annotation>
  </xs:element>
  ...
  <xs:element name="attribute" type="xs:topLevelAttribute"
  id="attribute"> ... </xs:element>
  ...
  <xs:element name="complexType" type="xs:topLevelComplexType"
  id="complexType"> ... </xs:element>
  ...
  <xs:element name="simpleType" type="xs:topLevelSimpleType"
  id="simpleType"> ... </xs:element>
  ...
  <xs:complexType name="topLevelElement"> ... </xs:complexType>
  ...
  <xs:complexType name="topLevelAttribute"> ... </xs:complexType>
  ...
  <xs:complexType name="topLevelComplexType"> ... </xs:complexType>
  ...
  <xs:complexType name="topLevelSimpleType"> ... </xs:complexType>
  ...
</xs:schema>
```

Figure 53 : Schéma du vocabulaire XML Schema

4.5.2 RDF-RDFS

RDF (Resource Description Framework) et RDFS (RDF Schema) ([175], [176], [177], [178], [179]) sont des langages pour la représentation sémantique d'informations sur le Web. Ils sont proposés par le W3C comme un cadre général pour la définition de toutes sortes de métadonnées.

Le modèle RDF définit trois types d'objets : ressources, propriétés, et valeurs. Une ressource (`rdfs:Resource`) est un objet décrit par RDF et identifié par son URI

(Uniform Resource Identifier). Une propriété (`rdf:Property`) est un attribut, un aspect, une caractéristique qui s'applique à une classe de ressources. Une instance d'une propriété peut être un lien (relation) binaire d'une ressource avec une autre. Les valeurs sont les valeurs particulières affectées aux propriétés. Une valeur peut être une ressource ou un littéral. Un littéral est d'un type primitif tel qu'un entier ou une chaîne de caractères. Une ressource est identifiée par un URI tandis qu'un littéral ne l'est pas.

Une assertion RDF sur une ressource est représentée comme un triplet (*ressource*, *propriété*, *valeur*) ou bien (*sujet*, *prédicat*, *objet*). Une description RDF est une suite d'assertions, pouvant être représentée sous forme de graphes ou sous forme de documents RDF/XML. Dans la représentation graphique, la *ressource* (*sujet*) et la *valeur* (*objet*) correspondent aux nœuds du graphe, et la *propriété* (*prédicat*) entre la *ressource* (*sujet*) et la *valeur* (*objet*) correspond à l'arc reliant le nœud *ressource* (*sujet*) au nœud *valeur* (*objet*).

La Figure 54 montre un exemple d'un graphe représentant l'énoncé «*Jean est une personne et son nom est Jean Tremblay*». La Figure 55 décrit un document RDF/XML correspondant à ce graphe. La personne Jean est représentée par la ressource de URI : `http://www.exemple.org/test/exRDF#Jean`. Cette ressource est associée par `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` (`rdf:type`) à la ressource `http://www.exemple.org/test/exRDFS#Personne` (`"&exRDFS;Personne"`) comme son type. Elle est aussi associée par `http://www.exemple.org/test/exRDFS#nom` (`exRDFS:nom`) au littéral «Jean Tremblay» comme son nom.

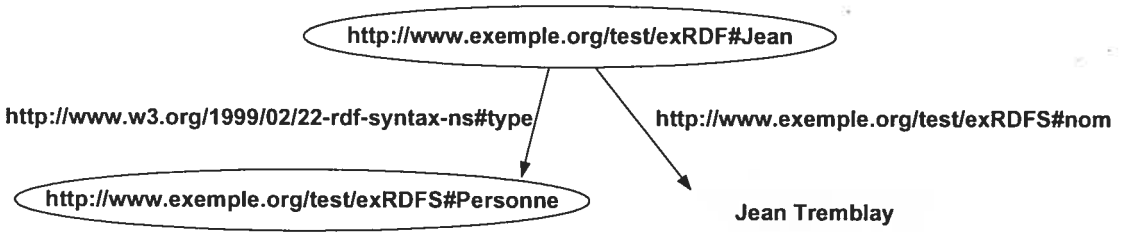


Figure 54 : RDF/RDFS - Représentation graphique

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF
          [<!ENTITY exRDFS "http://www.exemple.org/test/exRDFS#">]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:exRDFS="http://www.exemple.org/test/exRDFS#"
```

```
        xmlns:base="http://www.exemple.org/test/exRDF">
  <rdf:Description rdf:ID="Jean">
      <rdf:type rdf:resource="&exRDFS;Personne"/>
      <exRDFS:nom>Jean Tremblay</exRDFS:nom>
  </rdf:Description>
</rdf:RDF>
```

Figure 55 : Document RDF/XML correspondant à la Figure 54

RDFS spécifie le vocabulaire et la grammaire de RDF. Les schémas RDF sont eux-mêmes écrits en RDF en utilisant les balises de l'espace de nom du langage RDFS.

RDFS supporte deux notions primitives suivantes: classes et propriétés. Les ressources sont divisibles en groupes/classes (`rdfs:Class`). Une classe est aussi une ressource identifiée d'une manière unique et elle décrite par des propriétés. La propriété `rdf:type` est utilisée pour indiquer qu'une ressource est une instance d'une classe. Et la propriété `rdfs:isDefinedBy` est utilisée pour indiquer qu'une ressource est définie par une ressource. Une propriété est déclarée comme un type relationnel binaire entre classes de ressources. Une instance d'une propriété est décrite comme un lien binaire entre une ressource *sujet* avec une ressource *objet*. Les propriétés `rdfs:domain` et `rdfs:range` restreignent respectivement les domaines d'application et de variation pour une propriété donnée. Soit P une propriété qui a un domaine d'application D et un domaine de variation R, et soit (d, P, r) une assertion. Alors P est une instance de `rdf:Property`; D et R sont des instances de `rdfs:Class`; et d, r sont respectivement les instances de D et R. Les rapports de sous-typage entre classes et celui entre propriétés sont définis respectivement par les propriétés `rdfs:subClassOf` et `rdfs:subPropertyOf`.

RDF/RDFS laissent aux applications l'interprétation de certaines déclarations dans RDFS, par exemple, la déclaration des propriétés pour une classe (partie 5.3 – [175]). Ceci implique que l'interprétation peut varier d'une application à une autre application et rend donc plus difficile la construction d'un mécanisme d'inférence pour RDF.

Noyau réflexif de RDF/RDFS

Les éléments de RDF/RDFS sont définis par les éléments de RDF/RDFS eux-mêmes [179]. Ceci montre donc que, comme dans le cas de XML Schema, le schéma des vocabulaires RDF/RDFS est le métamodèle de lui-même, et la séparation des niveaux de

modélisation dans RDF/RDFS n'est pas si évidente.

La Figure 56 présente une partie du schéma RDFS des vocabulaires RDF et RDFS. Cette figure décrit le noyau réflexif de RDF/RDFS avec la définition de ces éléments de base de RDF/RDFS: `rdfs:Resource`, `rdfs:Class`, `rdf:Property`, `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, et `rdfs:subPropertyOf`. Toutes les classes (y compris `rdfs:Resource`, `rdfs:Class`, `rdf:Property`) sont conformes à `rdfs:Class` (ce qui est indiqué par des liens `rdf:type`). `rdfs:Class`, `rdf:Property` sont sous-types de `rdfs:Resource` (ce qui est indiqué par des liens `rdfs:subClassOf`). Toutes les propriétés (y compris `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, et `rdfs:subPropertyOf`) sont conformes à `rdf:Property` (ce qui est indiqué par des liens `rdf:type`). La signification et l'interaction entre ces éléments de base ont été présentées plus haut (dans la présentation sur RDFS).

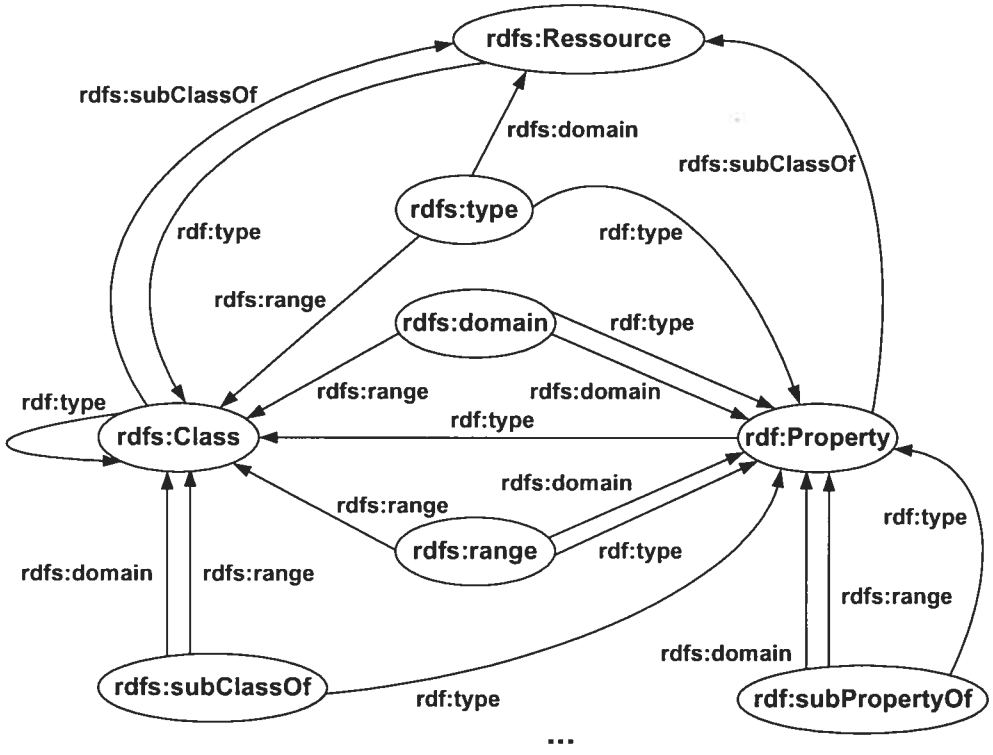


Figure 56 : Noyau réflexif de RDF/RDFS

4.5.3 OWL

OWL (Ontology Web Language) ([180], [181], [182], [183]) standardisé par W3C est un langage d'ontologie pour le Web. Il est une extension de RDF, et est inspiré de

DAML+OIL (présenté dans [8], [30]).

Comme les langages précédemment évoqués, OWL est construit au-dessus de RDF. Un document OWL est donc composé de triplets RDF, pouvant être écrits dans la syntaxe RDF de son choix. Au sein d'un document, les triplets RDF non définis dans la spécification OWL sont ignorés. Contrairement à XML Schema, OWL permet de faire des raisonnements et des inférences sur l'ontologie. Ainsi, OWL permet de conclure sur des faits implicites, alors que XML Schema ne traite que des faits explicites. Un exemple de déduction de type d'une instance est donné dans la Figure 57. Comme la propriété `travaille-pour` relie la classe `Personne` à `Organization`, et l'individu `Jean` est relié par `travaille-pour` à `Teximus`, l'on peut déduire que `Jean` est une instance de `Personne`.

```
...
<owl:ObjectProperty rdf:ID="travaille-pour">
   <rdfs:domain rdf:resource="#Personne"/>
   <rdfs:range rdf:resource="#Organization"/>
</owl:ObjectProperty>
<owl:Thing rdf:ID="Jean">
  <travaille-pour rdf:resource="#Teximus"/>
</owl:Thing>
```

Figure 57 : OWL – Exemple de déduction de type d'une instance

Il existe trois sortes de OWL: *OWL Lite*, *OWL DL*, *OWL Full*. OWL Lite supporte les besoins primitifs comme la constitution de taxonomies ou de thesaurus. Il supporte également les contraintes de cardinalité simples où la valeur de cardinalité doit être 0 ou 1 afin de simplifier l'implémentation, de faciliter, et d'accélérer la migration des taxonomies et thesaurus. OWL DL (OWL Description Logic) correspond exactement aux logiques de description [85]. Ce langage est expressif, et les procédures d'inférences sont complètes (complétude computationnelle), et réalisables en un temps fini (décidabilité). Dans OWL DL, les trois types *classes*, *propriétés* et *individus* doivent être disjoints mutuellement. OWL Full offre une expressivité maximale, mais sans garantie quant à la complétude et à la terminaison des procédures d'inférence. Comme dans RDF/RDFS, une classe dans OWL Full peut être traitée comme une collection d'individus et également comme un individu. Les relations de ces trois sortes OWL sont les suivantes:

– Les ontologies qui sont légales dans OWL Lite sont légales dans OWL DL.

– Les ontologies qui sont légales dans OWL DL sont aussi légales dans OWL Full.

– Les conclusions qui sont valides dans OWL Lite sont valides dans OWL DL.

– Les conclusions qui sont valides dans OWL DL sont aussi valides dans OWL Full.

Une ontologie OWL (`owl:Ontology`) est composée de triplets RDF. Elle peut contenir les éléments suivants: en-têtes optionnels (commentaire, version, importation d'ontologie), éléments de classe, éléments de propriétés et instances.

Une classe OWL (`owl:Class`) est soit anonyme, soit désignée par son URI. `owl:Thing` et `owl:Nothing` sont deux classes (de type `owl:Class`) prédéfinies dans OWL. Tout individu est une instance de la classe `owl:Thing` alors que `owl:Nothing` n'a aucune instance. Dans OWL Full, la classe `owl:Thing` est équivalente à `rdf:Ressource`, et `owl:Class` équivalente à `rdf:Class`. Pour définir une classe, OWL supporte entre autres : des opérateurs de relation et de subsomption entre classes (le sous-typage (`rdfs:subClassOf`); la disjonction (`owl:disjointWith`), l'équivalence (`owl:equivalentClass`)); les opérateurs booléens entre classes (l'intersection (`owl:intersectionOf`), l'union (`owl:unionOf`), le complément (`owl:complementOf`)); l'énumération exhaustive des instances de la classe; et la restriction (`owl:Restriction`) sur une propriété d'une classe préexistante (`owl:allValuesFrom`, `owl:someValuesFrom`, `owl:hasValue`, `owl:cardinality`, `owl:maxCardinality`, `owl:minCardinality`); etc.

OWL distingue deux sortes de propriétés: les propriétés reliant les individus (`owl:ObjectProperty`) et les propriétés reliant les individus aux types de données (`owl:DatatypeProperty`). Les types de données sont ceux de XML Schema tels que les entiers (`integer`), les chaînes de caractères (`string`), etc. Ces deux sortes de propriétés, `owl:ObjectProperty` et `owl:DatatypeProperty`, sont bien disjointes dans OWL Lite et OWL DL. Alors que dans OWL Full, `owl:DatatypeProperty` est une sous-classe de `owl:ObjectProperty` (c'est-à-dire les valeurs de données font partie du domaine des individus), et `owl:ObjectProperty` est équivalente à `rdf:Property`. Pour définir des propriétés, OWL réutilise les éléments de RDFS comme `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range` et en définit d'autres. En voici quelques-uns. `owl:equivalentProperty` indique les propriétés

équivalentes. owl:inverseOf exprime que deux propriétés sont inversées : si *P1* est inversée de *P2*, alors il existe *P1*(x,y) si et seulement s'il existe *P2*(y,x). Une propriété peut être transitive (owl:TransitiveProperty), symétrique (owl:SymmetricProperty), fonctionnelle (owl:FunctionalProperty), ou une propriété fonctionnelle inversée (owl:InverseFunctionalProperty). Soit P une propriété fonctionnelle, si *P*(x,y) et *P*(x,z), alors y est identique à z. Soit *P* une propriété fonctionnelle inversée, et si *P*(x,z) et *P*(y,z), alors x est identique à y.

OWL définit aussi les propriétés et les classes pour indiquer la comparaison entre les individus : individus d'une même identité (owl:sameAs), individus différents d'autres individus (owl:differentFrom), individus différents deux à deux (owl:AllDifferent). OWL définit également des propriétés (owl:OntologyProperty) et des classes pour la gestion des versions («versionning») d'ontologies. Pour une ontologie, owl:versionInfo indique les informations sur la version en cours; owl:priorVersion pointe à la version précédente; la version en cours est compatible (ou non) avec la version pointée par owl:backwardCompatibleWith (ou owl:inCompatibleWith). owl:DeprecatedClass (owl:DeprecatedProperty) sert à indiquer qu'une classe (propriété) pourra ne plus être utilisée dans les versions futures et devra désormais être remplacée par une autre classe (propriété). Une ontologie peut importer (owl:imports) une autre ontologie, c'est-à-dire les éléments définis dans l'ontologie importée sont considérés comme présents et utilisables dans l'ontologie qui l'importe.

Noyau réflexif de OWL

Le schéma du vocabulaire OWL importe le schéma RDFS des vocabulaires RDF/RDFS [182]. Autrement dit, les éléments du vocabulaire OWL sont définis en utilisant le noyau de RDF/RDFS. Le noyau réflexif de OWL est donc étendu à partir de celui de RDF/RDFS que nous avons analysé dans la section 4.5.2 (page 81). La définition des éléments de base de ce noyau a été illustrée dans la Figure 56 (page 84). Les éléments de OWL qui jouent le rôle de métaclasses sont conformes à rdfs:Class; exemples: owl:Ontology, owl:Class, owl:Restriction, owl:ObjectProperty, etc. Les éléments de OWL qui jouent le rôle de méta-propriétés sont conformes à rdf:Property; par exemple: owl:disjointWith,

`owl:equivalentClass`, `owl:intersectionOf`, `owl:allValuesFrom`, `owl:sameAs`, `owl:imports`, `owl:cardinality`, etc.

Évaluation des formalismes XML-XML Schema, RDF-RDFS et OWL

Évaluation suivant les besoins pour M2

Les formalismes XML-XML Schema, RDF-RDFS et OWL sont utilisés dans le contexte du Web sémantique. XML et XML Schema sont reconnus pour leur flexibilité d'expression mais aussi pour l'ambiguïté sémantique dans la représentation qui rend le traitement sémantique des informations difficile. RDF/RDFS et OWL sont des langages développés explicitement pour la représentation sémantique de l'information sur le Web. Cependant, ils ne supportent pas tous nos besoins pour M2. Par exemple, ils ne supportent pas des types relationnels ou des relations autres que binaires (BesoinM2 1 - «Typage entre éléments du niveau M1»). Ils ne font pas de distinctions entre les notions : la conformité entre éléments de différents niveaux de modélisation, la conformité sémantique entre modèles, et l'instanciation locale «types-instances» (BesoinM2 4 - «Distinction entre différentes notions de conformité»). Ils ne supportent aucun modèle de rôles. Ils ne tiennent pas compte du besoin «Classification dynamique et Classification statique» (BesoinM2 9). Ils ne tiennent pas compte non plus des différents types de contraintes structurelles. Ils ne répondent pas complètement au besoin «Types relationnels entre types et/ou instances» (BesoinM2 15). Étant donné que ces formalismes ne visent pas la représentation ni la gestion de modèles, ils ne précisent pas les différents types de modèles cités dans le besoin «Distinction entre modèles de différentes natures» (BesoinM2 16) tels que les notions de métamodèles, modèles structurels, modèles de conditions; ils ne considèrent pas non plus les différents liens entre modèles tels que la conformité sémantique entre un modèle et son métamodèle, l'intersection, la différence, l'inférence, la restriction (cf. le BesoinM2 17 - «Liens avec les modèles»), l'équivalence entre modèles, les transformations entre modèles, etc. (cf. le BesoinM2 18 - «Liens entre modèles au niveau M1»). Voir les sections I.9.1, I.10.1 et I.11.1 (Annexe I) pour l'évaluation plus détaillée.

Évaluation des noyaux de XML-XML Schema, RDF-RDFS et OWL

Ces formalismes XML-XML Schema, RDF-RDFS et OWL possèdent chacun son noyau réflexif. Pourtant, ces noyaux ne répondent pas non plus à tous nos besoins pour

M3. Concernant les notions de base présentées dans le besoin «Typage entre éléments de deux niveaux de modélisation consécutifs» (BesoinM3 1), le noyau de XML Schema ne fait pas explicitement la distinction entre la notion de «*méta-nœuds*» et celle de «*méta-lien*» ni entre «*nœud*» et «*lien*»; les noyaux de RDF/RDFS/OWL n'autorisent aucune propriété (c'est-à-dire *méta-lien* binaire) entre propriétés ou entre propriété avec classe (*méta-nœud*). Évidemment, en se basant sur les manques cités ci-dessus au sujet de la représentation des modèles et de leurs liens dans les formalismes XML-XML Schema, RDF-RDFS et OWL, aucun des besoins «Distinction entre modèles de différentes natures» (BesoinM3 6) et «Liens avec les modèles» (BesoinM3 7) n'est satisfait entièrement par les noyaux de ces formalismes. Voir les sections I.9.2, I.10.2 et I.11.2 (Annexe I) pour plus d'explications.

4.6 Synthèse

Nous avons fait une revue des principaux formalismes de l'intelligence artificielle et de l'industrie pour la représentation de connaissances ainsi que pour la représentation des modèles. Les formalismes choisis pour l'étude sont: les réseaux sémantiques, les graphes conceptuels (GCs), le modèle uniforme des GCs, sNets, CDIF, Entité-Association, MOF2.0, UML2.0, XML-XML Schema, RDF/RDFS et OWL. Les points forts et faibles de chacun des formalismes ont été analysés en tenant compte de nos besoins (cf. le Chapitre 3, page 26) et aussi de la facilité de l'implémentation. La Table 1 (page 90) présente une synthèse de la comparaison des différents noyaux réflexifs en réponse aux besoins essentiels pour M3. Et la Table 2 (page 91) présente une synthèse de la comparaison des différents formalismes permettant de modéliser le monde réel, en réponse aux besoins essentiels pour M2 concernant la représentation de modèles de connaissances du monde réel.

BesoinM3 1 : Typage entre éléments de deux niveaux de modélisation consécutifs

BesoinM3 2 : Passage entre niveaux types et instances

BesoinM3 3 : Distinction entre différentes notions de conformité au niveau méta

BesoinM3 4 : Hiérarchisations entre types

BesoinM3 5 : Contraintes de cardinalité min/max pour un méta-lien

BesoinM3 6 : Distinction entre modèles de différentes natures

BesoinM3 7 : Liens avec les modèles

Noyaux réflexifs (au M3)	Besoins pour M3 (*BesoinM3*)						
	1	2	3	4	5	6	7
Réseaux sémantiques (RSs)	-	-	-	-	-	-	-
sNets	+-	+	+	+-	+-	+-	+-
Graphes conceptuels (GCs)	+-	+	-	+	-	+-	+-
Modèle uniforme des GCs	+-	+	-	+	+	+-	+-
CDIF	+-	+	-	+	+	-	+-
MOF	+-	+	-	+	+	-	+-
XML-XML Schema	+-	+	-	+	-	+-	+-
RDF-RDFS	+-	+	-	+	-	+-	+-
OWL	+-	+	-	+	+	+-	+-

Notations:

«+» : remplir le besoin correspondant;

«+-»: remplir partiellement le besoin correspondant et à développer pour le remplir;

«-» : ne pas remplir et à développer pour remplir le besoin correspondant

Table 1: Synthèse des formalismes à représenter et à gérer les métamodèles

BesoinM2 1 : Typage entre éléments du niveau M1

BesoinM2 2 : Typage entre éléments de deux niveaux de modélisation consécutifs

BesoinM2 3 : Passage entre niveaux types et instances

BesoinM2 4 : Distinction entre différentes notions de conformité

BesoinM2 5 : Hiérarchisations entre types

BesoinM2 6 : Multi-classification et Connaissance partielle

BesoinM2 7 : Objets, Rôles, Types d'objets, Types de rôles

BesoinM2 8 : Rapports entre objets et rôles / types de rôles

BesoinM2 9 : Classification dynamique et classification statique

BesoinM2 10 : Contraintes de l'arité min/max

BesoinM2 11 : Contraintes de cardinalité totale min/max pour un type relationnel

BesoinM2 12 : Contraintes de cardinalité locale min/max pour un type relationnel

BesoinM2 13 : Contraintes de l'itération min/max pour un type relationnel

BesoinM2 14 : Contraintes de cardinalité min/max pour un méta-lien

BesoinM2 15 : Types relationnels entre types et/ou instances

BesoinM2 16 : Distinction entre modèles de différentes natures

BesoinM2 17 : Liens avec les modèles

BesoinM2 18 : Liens entre modèles au niveau M1

Métamodèle au M2	**Besoins pour M2 (*BesoinM2*)**																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
RSs	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
sNets	-	+-	+	-	-	-	-	-	-	-	-	-	-	+-	-	-	-	-
GCs	+	+-	+	-	+	+	+-	+-	-	-	-	-	-	-	+-	+-	+-	-
Modèle uniforme des GCs	+	+-	+	-	+	+	-	-	-	-	-	+	-	+	+-	+-	+-	-
Entité-Association	+-	+-	-	-	+-	-	+-	+-	-	-	-	+	-	+	-	-	-	-
UML	+	+-	+	-	+	+	+	+	-	+-	-	+	-	+	+-	+-	+-	+
XML-XML Schema	+-	+-	+	-	+	-	-	-	-	+-	-	-	-	-	+-	+-	+-	+-
RDF-RDFS	+-	+-	+	-	+	+	-	-	-	-	-	-	-	-	+-	+-	+-	+-
OWL	+-	+-	+	-	+	+	-	-	-	-	-	+	-	+	+-	+-	+-	+-

Notations:

«+» : remplir le besoin correspondant;

«+-» : remplir partiellement le besoin correspondant et à développer pour le remplir;

«-» : ne pas remplir et à développer pour remplir le besoin correspondant

Table 2: Synthèse des formalismes à représenter et gérer les modèles au niveau M1

Cette analyse a montré qu'aucun de ces formalismes ne répond à tous les besoins correspondants à son ou ses niveaux de modélisation et également qu'il existe des besoins dont chacun n'est pas complètement rempli par aucun de ces formalismes. Comme aucun des formalismes pour la modélisation du monde réel étudiés ne remplit pas tous les besoins concernant les contraintes structurelles (cf. les BesoinM2 10, BesoinM2 11, BesoinM2 12, BesoinM2 13 et BesoinM2 14), aucun de ces formalismes ne permet de modéliser les situations telles que celle décrite dans le Problème 1 (page 34). Aucun de ces formalismes ne permet non plus de modéliser les situations telles que celle décrite dans le Problème 2 (page 37). Ces constats nous ont poussés à développer et proposer un nouveau formalisme permettant de répondre à nos besoins de représentation des modèles répartis à tous les trois niveaux de modélisation (M1, M2, et M3). Les chapitres qui suivent présentent notre proposition.

Chapitre 5. Méta-métamodèle et Métamodèle pour la gestion de modèles

Comme nous l'avons mentionné, notre architecture de modélisation est conforme à celle à quatre niveaux (M0, M1, M2 et M3) étudiée à la section 2.2 (page 14). Notre architecture permet de distinguer clairement les différentes notions de conformité selon les niveaux de modélisation et également de bien séparer les éléments de différents niveaux de modélisation. Ceci est illustré par la Figure 58. Les liens de type `instOf` représentent l'instanciation entre les types et les instances au niveau M1. Les liens de type `Mmeta` représentent le rapport de conformité entre les éléments au niveau M2/M3 et les méta-éléments au niveau M3, alors que les liens de type `meta` représentent celui entre les éléments au niveau M1 et les méta-éléments au niveau M2. Les liens de type `Msem` représentent le rapport de conformité entre les modèles au niveau M2 et le niveau M3 ainsi que celui entre le niveau M3 et lui-même, alors que les liens de type `meta` représentent le rapport de conformité entre les modèles de niveau M1 et le niveau M2.

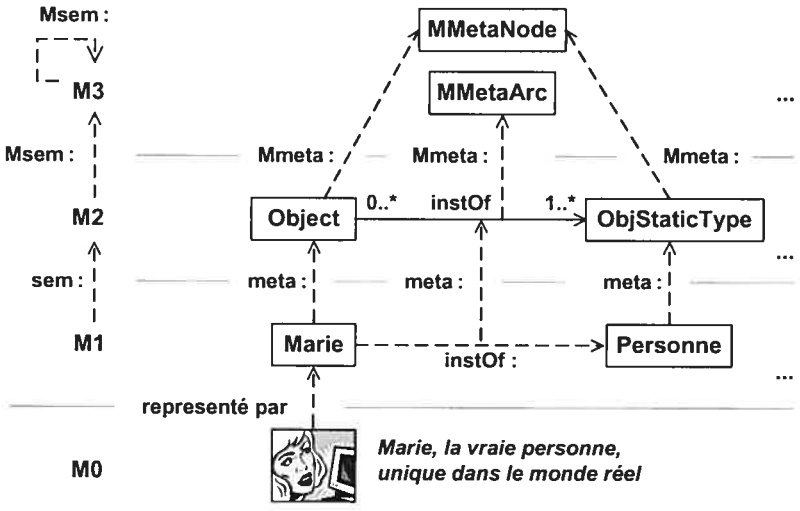


Figure 58 : Notre architecture de modélisation

La Figure 58 présente également en partie notre «framework» proposé pour la gestion des modèles. Ce «framework» se base sur celui dans notre travail de spécification d'un métamodèle pour la gestion des connaissances ([36], [38], [39], [41]). Nous définissons dans notre «framework», aux niveaux méta-métamodèle (M3) et métamodèle (M2), tous les éléments nécessaires ainsi que leurs interactions. Ces éléments représentent le vocabulaire et la grammaire au moyen desquels les utilisateurs pourront spécifier leurs

modèles au niveau M2 et/ou au M1.

Par la suite, nous présentons nos méta-métamodèle et métamodèle pour la gestion de modèles. Nous commençons par le principe de la représentation dans notre formalisme.

5.1 Représentation basée sur les nœuds et les arcs

Comme pour les réseaux sémantiques, la représentation de notre formalisme est basée sur les *nœuds* et les *arcs* dont les notions ont été précisées par la Définition 4 (page 11). La Notation 1 présente nos notations graphiques pour les nœuds et les arcs.

Notation 1 *: Notations pour les nœuds et les arcs*

Un nœud est représenté graphiquement par un rectangle alors qu'un arc est représenté graphiquement par une flèche pointillée, allant de l'élément source à l'élément destination (Figure 59).

Figure 59 : Notations graphiques pour les nœuds et les arcs

Soient `A` un élément, `MetaA` le méta-élément de `A`. `A` s'écrit parfois sous la forme «`MetaA : A`» qui indique que l'élément `A` est conforme au méta-élément `MetaA`.

Un *arc* est un *lien binaire et unidirectionnel* (cf. la Définition 4, page 11). Nous pouvons donc représenter tout modèle conforme à notre définition de modèles (cf. la Définition 1, page 10) par des nœuds et des arcs, tout comme expliqué dans la section 3.1.1.2 (page 26), par des *nœuds* et des *liens binaires* (cf. la Définition 4, page 11). Un modèle est représenté par un ensemble d'éléments et de relations entre ces éléments. Un élément est représenté par un nœud ou un arc. Une relation unaire, binaire ou n-aire est fondamentalement représentée par un nœud et un ensemble d'arcs. Évidemment, le fait de choisir des éléments pour notre formalisme qui soient des nœuds et des arcs n'influence pas le pouvoir d'expression (concernant la représentation de modèles) de ce formalisme. Ce choix simplifie en outre la mise en œuvre des représentations.

5.2 Méta-métamodèle

Le méta-métamodèle (M3) fournit le langage et la grammaire pour décrire les formalismes de modélisation. Notre M3 est inspiré de celui des GCs et des sNets.

5.2.1 Éléments de base

Les éléments de base du niveau M3 sont: `MElement`, `MNode`, `MArc`, `MMetaElement`, `MMetaNode`, `MMetaArc`, `MsubType`, `Mmeta`, `Msrce`, et `Mdest`. La signification de ces éléments ainsi que leurs interactions sont détaillées ci-dessous.

MElement, MNode, MArc

`MElement`, `MNode`, `MArc` sont des types abstraits (cf. la Définition 3, page 11). `MElement` est à la racine de la hiérarchie de tous les méta-éléments définis au niveau M3. Cette hiérarchie se divise en deux branches: l'une pour tous les méta-nœuds de M3 dont `MNode` est à la racine, et l'autre pour tous les méta-arcs de M3 dont `MArc` est à la racine. Chaque élément défini au niveau méta (c'est-à-dire au niveau M2 ou M3) est vu comme nœud ou un arc qui est conforme à un et un seul méta-élément de M3. Ce premier est attaché par un arc de type `Mmeta` (représentant le lien de conformité) à ce méta-élément.

MMetaElement, MMetaNode, MMetaArc

Comme un méta-élément est aussi un élément au niveau méta, il doit être conforme à un méta-élément de M3. Pour cette raison, `MMetaElement` représente l'ensemble de tous les méta-éléments. Ces méta-éléments, exceptés ceux conformes à `MMetaElement`, sont classifiés en deux groupes : les méta-nœuds (`MMetaNode`), et les méta-arcs (`MMetaArc`). Chaque élément considéré comme un méta-nœud (ou un méta-arc) doit être conforme à `MMetaNode` (ou à `MMetaArc`).

Puisque les méta-éléments sont aussi des nœuds, les éléments `MMetaElement`, `MMetaNode`, `MMetaArc` sont des méta-nœuds et sont donc sous-types de `MMetaNode`.

Msrce, Mdest

Les méta-arcs `Msrce` et `Mdest` permettent de spécifier les sources et les destinations des méta-arcs. Un méta-arc unit deux méta-éléments via un arc de type `Msrce` et un arc de type `Mdest`. Quelques exemples sont illustrés dans les Figure 60 et Figure 61.

Puisqu'il est le super-type de tous les méta-arcs de M3, la structure du méta-arc `MArc` devrait être la plus générale que possible. Comme l'illustre la Figure 60, le méta-arc `MArc` (rectangle) lie le méta-élément `MElement` à `MElement` lui-même via un arc de type `Msrce` et un arc de type `Mdest` (flèches pointillées). Cette structure de `MArc` exprime par ailleurs qu'un méta-arc peut être défini pour relier deux méta-éléments où

un méta-élément peut être un méta-nœud ou un méta-arc. De cette façon, un arc peut exister sous toutes les formes: un arc entre deux éléments où un élément peut être un nœud ou un arc.

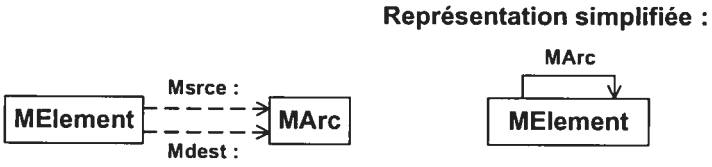


Figure 60 : Structure initiale de *MArc*

La Figure 61 illustre une structure de `instOf`, un méta-arc qui est défini au niveau M2 pour représenter le type d'instanciation entre les objets (`Objet`) et les types statiques d'objets (`ObjStaticType`). Dans cette figure, le méta-arc `instOf` (rectangle) lie `Objet` à `ObjStaticType` via un arc de type `Msrce` et un arc de type `Mdest` (flèches pointillées).

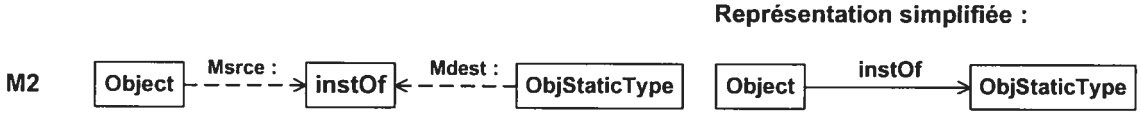


Figure 61 : Structure d'un méta-arc de M2

MsubType

`MsubType` sert à classifier les méta-éléments. Il implémente la relation de sous-typage entre méta-éléments où le sous-typage multiple est permis. Cependant, un méta-nœud (ou un méta-arc) ne peut être un sous-type d'un méta-arc (ou d'un méta-nœud) (cf. la Règle 2, Annexe IV). L'effet de la relation de sous-typage (cf. la Définition 5, page 13, et la Règle 1, Annexe IV) nous permet, en particulier, de déduire de nouvelles structures pour un méta-arc à partir de sa structure initiale, en remplaçant l'élément source ou destination dans cette structure par un de ses inférieurs (sous-types)

Mmeta

Les arcs de type `Mmeta` représentent les relations de conformité qui associent les éléments au niveau méta à leurs méta-éléments, soit du même niveau M3 ou du niveau M2 au M3 dans l'architecture de modélisation. Dans le deuxième cas, ces liens jouent le rôle de transition entre les niveaux M2 et M3.

Un élément (nœud ou arc) au niveau M3 ou M2 a un et un seul méta-élément

(respectivement méta-nœud ou méta-arc) auquel il est rattaché, une fois défini, par un arc de type `Mmeta`. Les arcs de type `Mmeta` permettent alors d'indiquer la nature de chaque élément représenté au niveau M3 ou M2.

De plus, la règle existentielle entre un nœud (ou un arc entre deux éléments) et son méta-nœud (ou son méta-arc entre deux méta-éléments correspondants) doit être respectée (cf. la Règle 3, Annexe IV). La Figure 62 illustre un exemple du cas.

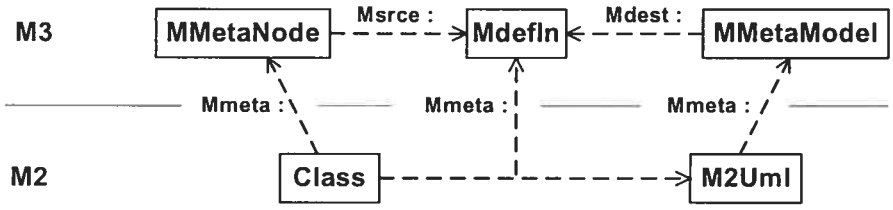


Figure 62 : Rapport existentiel entre un arc et son méta-arc

Dans cette figure, au niveau M3, le méta-arc `MdefIn` (rectangle) lie `MMetaNode` à `MMetamodel` via un arc de type `Msrce` et un arc de type `Mdest` (flèches pointillées). Ceci spécifie que les méta-nœuds (`MMetaNode`) sont définis dans les métamodèles (`MMetamodel`). Afin de représenter que le méta-nœud `Class` est défini dans le métamodèle `M2Uml`, l'arc qui associe `Class` à `M2Uml` est de type `MdefIn` et est conforme au méta-arc `MdefIn` défini entre deux méta-éléments correspondants `MMetaNode` et `MMetamodel`. L'arc de type `MdefIn` de `Class` à `M2Uml` que la Figure 62 spécifie peut être représenté plus simplement comme à la Figure 63.

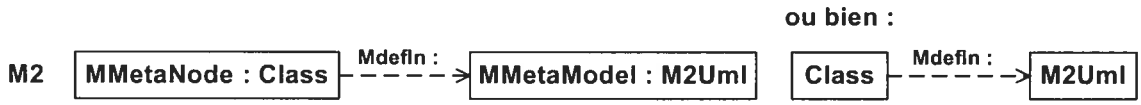


Figure 63 : Version simplifiée de la Figure 62

5.2.2 Autres éléments

Nous présentons maintenant les autres éléments du méta-métamodèle qui sont nécessaires pour définir les éléments de M2. Ces éléments sont regroupés en fonction de leur utilité.

5.2.2.1 Nommage d'éléments

Les éléments `MLabel`, `Mname` permettent le nommage d'éléments (Figure 64). Un libellé (`MLabel`) peut nommer (`Mname`) au maximum un `MElement`, et un `MElement`

peut être nommé par au maximum un `MLabel`. Les `MElement` nommés peuvent être identifiés par leurs noms. La Figure 65 illustre comment nommer un élément. Le libellé «Object» est attaché par un arc de type `Mname` à un méta-nœud sans nom; ce dernier est donc appelé le méta-nœud `Object`.

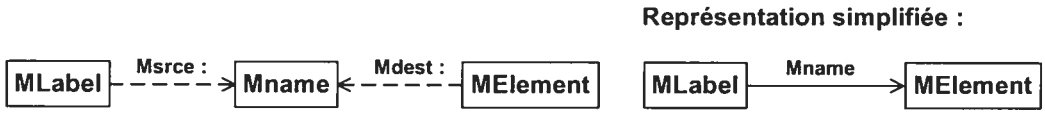


Figure 64 : Structure initiale de *Mname* (sans contraintes de cardinalité)

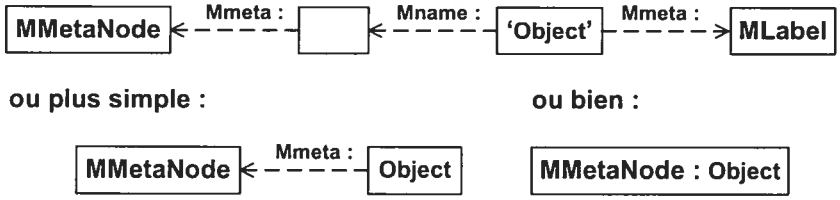


Figure 65 : Nommage d'un élément

`MNamedNode` est un méta-élément abstrait représentant l'ensemble des `MNode` nommés. Il est subdivisé en: `MLabelledInteger`, `MBoolean`, `MMetaElement`, `MModel`, `MRule`, `MUser`, `MList` et `MRef`.

5.2.2.2 Interprétation d'éléments

`MValue`, `MLanguage`, `Mdepend`, `MvalueOf` permettent le multilinguisme dans la représentation des interprétations des `MNamedNode`.

Au niveau méta, `MLanguage` et `MValue` sont utilisés pour représenter respectivement les langues et les valeurs de l'univers du discours qui servent à représenter les interprétations des éléments. Ces langues et ces valeurs sont spécifiées et attachées par les arcs de type `Mdepend/MvalueOf`.

Nous représentons le langage universel par «u», le français par «fr», l'anglais par «en», etc. Un élément peut être interprété différemment, dépendamment des langues. L'interprétation d'un élément en langage universel exprime que cette interprétation est identique pour toutes les langues. La Figure 66 illustre un exemple d'interprétations d'éléments. `Object` (de M2) est interprété en anglais comme «Object» et en français comme «Objet» (ce qui est indiqué par des arcs de type `Mdepend/MvalueOf`).

Les structures de `Mdepend`, `MvalueOf` sont illustrées dans la Figure 67.

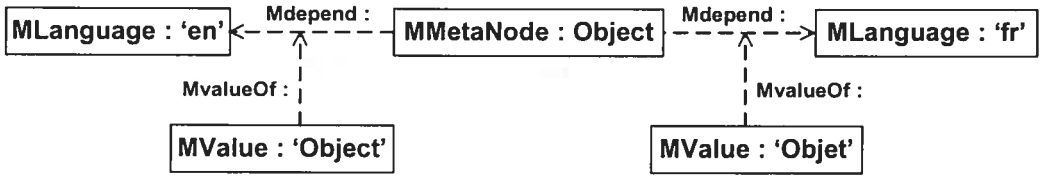


Figure 66 : Interprétation d'éléments

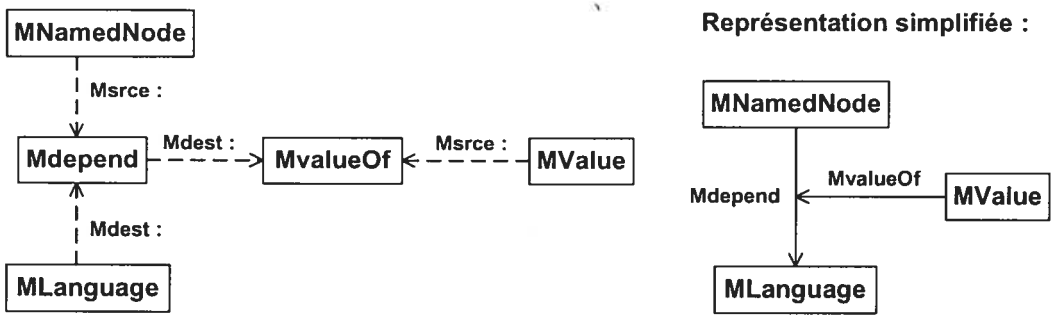


Figure 67 : Structures initiales de *Mdepend*, *MvalueOf* (sans contraintes de cardinalités)

5.2.2.3 Contraintes de cardinalité sur méta-arcs

MLabelledInteger, Minterval

`MLabelledInteger` permet de représenter les entiers étiquetés. Un entier étiqueté est interprété comme un entier. Par exemple, étant conforme à `MLabelledInteger`, l'élément `Mname-cardMin` représente un entier étiqueté qui vaut une valeur de 0. Le méta-arc `Minterval` (Figure 68) permet de former des intervalles fermés dont les bornes inférieures et celles supérieures sont des entiers étiquetés (`MLabelledInteger`). Ceci est utile à la représentation des contraintes de cardinalités sur les méta-arcs.

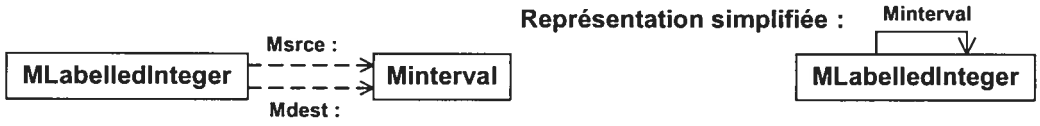


Figure 68 : Structure initiale de *Minterval* (sans contraintes de cardinalités)

Mcard

Le méta-arc `Mcard` sert à spécifier les contraintes de cardinalité sur les sources et destinations des méta-arcs. Une structure d'un méta-arc spécifie comment ce méta-arc relie ces éléments. Pour cette structure, il existe fréquemment, sur la source ou bien sur la destination du méta-arc, des contraintes de cardinalité minimale/maximale qui précisent le nombre d'instances de ce méta-arc dans lesquelles un élément peut être impliqué (cf. la Règle 4, Annexe IV). Ce nombre d'instances doit appartenir à un

intervalle, soit [*a*..*b*]; c'est-à-dire la cardinalité minimale (respectivement maximale) correspondante est de valeur *a* (respectivement *b*). Les contraintes de cardinalité dans ce cas peuvent être exprimées comme l'intervalle [*a*..*b*]. Cet intervalle est attaché à l'arc source ou destination correspondant du méta-arc en question par un arc de type `Mcard`.

L'exemple ci-après illustre comment spécifier les contraintes de cardinalités pour un méta-arc.

Exemple de spécification de contraintes de cardinalité pour un méta-arc

Le méta-arc à l'étude est `Mname`. *Un libellé (*`MLabel`*) peut nommer (*`Mname`*) au maximum un* `MElement`, *et un* `MElement` *peut être nommé par au maximum un* `MLabel`. *Les contraintes de cardinalités sur la source et celles sur la destination de* `Mname` *sont identiques: [0..1]. Dans ce cas, nous pouvons considérer que l'arc source et l'arc destination de* `Mname` *sont attachés au même intervalle. Comme l'illustre la Figure 69, sur la source et la destination de* `Mname`, *les cardinalités minimale et maximale de valeurs 0 et 1 sont représentées respectivement par deux entiers étiquetés,* `Mname-cardMin` *et* `Mname-cardMax`. *Ces deux cardinalités s'attachent par un arc de type* `Minterval` *pour former l'intervalle [*`Mname-cardMin`*..*`Mname-cardMax`*]. En remplaçant* `Mname-cardMin` *et* `Mname-cardMax` *par leurs valeurs effectives, étant respectivement 0 et 1 (en langue universel «u»), cet intervalle devient [0..1].*

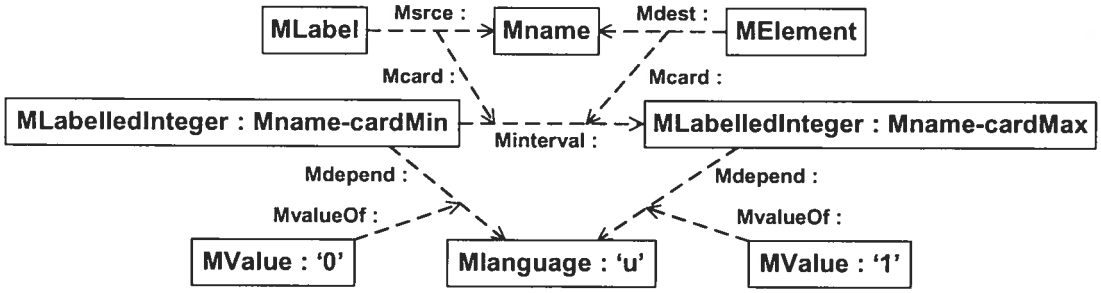


Figure 69 : Contraintes de cardinalité pour *Mname* (dans la structure initiale de *Mname*)

La structure plus restrictive de `Mname` *permet de spécifier qu'un* `MNamedNode` *doit être nommé. Ainsi, dans cette structure, les cardinalités minimale et maximale sur la source de* `Mname` *sont de la même valeur 1. Si nous considérons que ces cardinalités ont toujours la même valeur que* `Mname-cardMax`, *elles peuvent être représentées par* `Mname-cardMax`, *comme l'illustre la Figure 70. Dans ce cas, si la valeur de* `Mname-cardMax` *est modifiée à une nouvelle valeur, soit 2, les cardinalités minimale et maximale en question auront automatiquement la valeur 2. Mais si nous considérons qu'elles sont indépendantes l'une de l'autre et que chacune détient son propre libellé, alors la mise-à-jour d'une de ces cardinalités n'affectera en rien les autres.*

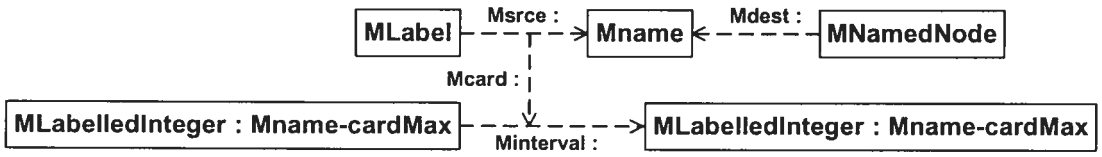


Figure 70 : Contraintes de cardinalité sur la source de *Mname* (dans une autre structure)

Nous énonçons ci-dessous les règles de notations pour les méta-arcs.

Notation 2 *: Notations pour les méta-arcs avec leurs contraintes de cardinalité*

Dans le cas où la cardinalité minimale (ou maximale) sur la source ou destination pour un méta-arc (c'est-à-dire sur un arc de type `Msrce/Mdest` pour un méta-arc) n'est pas indiquée, la valeur pour cette cardinalité minimale (ou maximale) est respectivement 0 (ou *). Le symbole «*» désigne un nombre entier de valeur infinie. Dans les illustrations, les contraintes de cardinalités pour un méta-arc s'écrivent sur l'arc de type `Msrce/Mdest` auquel elles sont attachées.

Afin de simplifier la représentation graphique d'un méta-arc, un méta-arc qui sert à unir un méta-élément source à un méta-élément destination peut être représenté par une flèche, en ligne solide, du méta-élément source au méta-élément destination. Si le méta-arc est représenté d'une manière simplifiée par une flèche en ligne solide, les contraintes (sur la source ou destination) s'écriront sur l'extrémité correspondante (source ou destination) de la flèche. La notation retenue est semblable à celle de UML pour en faciliter la lecture. Comme l'illustre la Figure 71, une instance de *X* peut être associée par des arcs de type *R* avec de *c* à *d* instances de *Y*, et une instance de *Y* peut être associée par des arcs de type *R* avec de *a* à *b* instances de *X*.

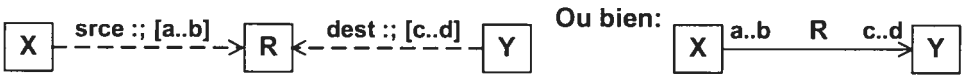


Figure 71 : Notations graphiques pour les méta-arcs

Ci-après nous présentons quelques exemples des structures de méta-arcs avec des contraintes de cardinalités. Les structures de `Mcard` illustrées dans la Figure 72 spécifient qu'un arc de type `Msrce` (respectivent un `Mdest`) peut être associé à un intervalle au maximum. La structure de `Mname` illustrée dans la Figure 73-(a) est une représentation simplifiée de celle dans la Figure 69 (page 99). La structure plus restrictive de `Mname` illustrée dans la Figure 73-(b) permet de spécifier qu'un

MNamedNode doit être nommé. Les structures de Mdepend, MvalueOf et leurs contraintes, illlustrées par la Figure 74 permettent de spécifier, pour une langue donnée, qu'un élément nommé a une seule interprétation.

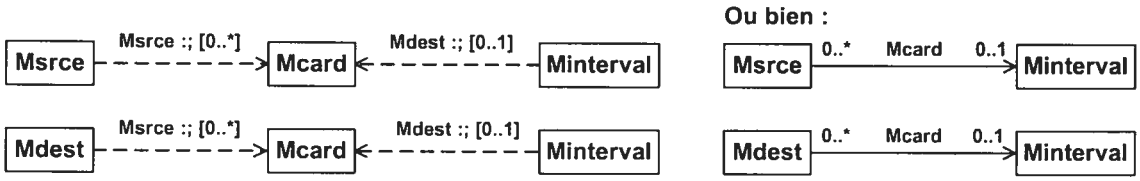


Figure 72 : Structures de *Mcard*

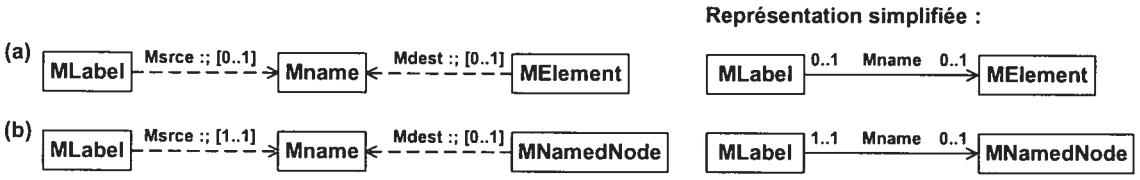


Figure 73 : Méta-arc *Mname* - (a) Structure initiale ; (b) Structure plus restrictive

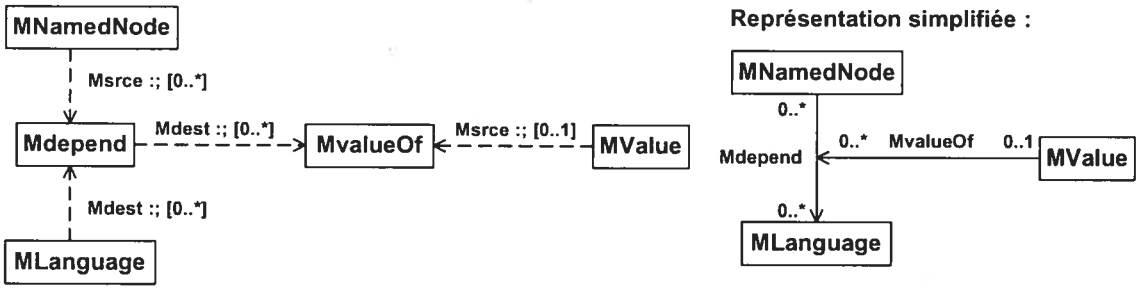


Figure 74 : Structures initiales de *Mdepend*, *MvalueOf*

5.2.2.4 Type relationnel binaire et bidirectionnel

Comme les méta-arcs sont des types relationnels binaires et unidirectionnels, le méta-arc Minverse offre un moyen permettant de représenter les types relationnels binaires et bidirectionnels entre méta-éléments. Le méta-arc Minverse est défini pour lier deux méta-arcs constituant un type relationnel binaire et bidirectionnel (Figure 75).

Figure 75 : Structure initiale de *Minverse*

Par exemple, étant binaire et bidirectionnel, le type relationnel «spécialisation-généralisation» entre des méta-éléments peut être formé par deux méta-arcs MsubType et MsuperType qui sont l'inverse l'un de l'autre (c'est-à-dire que le méta-arc

MsuperType est l'inverse du méta-arc MsubType, et le méta-arc MsubType est l'inverse du méta-arc MsuperType). Un arc de type MsubType entre deux méta-éléments, soit de *A* à *B*, spécifie que *A* est un sous-type de *B*. Cet arc implique un arc de type MsuperType de *B* à *A* (ce qui spécifie que *B* est un super-type de *A*). Vice-versa, un arc de type MsuperType de *B* à *A* implique également un arc de type MsubType de *A* à *B*. Ce type de comportement de Minverse est régi par la Règle 22 (Annexe IV).

5.2.2.5 Représentation de modèles et de leurs liens

MModel, MMetaModel, MStructure, MIfThenModel, Msem, MdefAs

MModel permet la contextualisation des éléments au niveau méta. Parmi les modèles (MModel) aux niveaux M2 et M3, nous distinguons les métamodèles (MMetaModel), les structures (MStructure), et les modèles de conditions (MIfThenModel).

MMetaModel représente l'ensemble des métamodèles. Les éléments dans un modèle ont leurs méta-éléments spécifiés dans un métamodèle auquel ce modèle est conforme (cf. la Règle 18, Annexe IV). Un modèle se conforme à un et un seul métamodèle. Msem (Figure 76) met en œuvre la relation de conformité entre les modèles au niveau méta et leurs métamodèles.

Figure 76 : Structure initiale de *Msem*

Le type MStructure a pour but de contextualiser les structures des méta-arcs. Le méta-arc MdefAs permet de représenter les *liens de définition* entre les méta-arcs et leurs structures initiales (Figure 77). Un méta-arc peut être défini par (MdefAs) une ou plusieurs structures (MStructure) dites *structures initiales* et une structure peut représenter la définition d'un seul méta-arc. Dans le cas où un méta-arc n'a aucune structure initiale propre à lui, sa définition est déduite de la ou les structures initiales de son ou ses super-types. Les Règle 16 et Règle 17 (Annexe IV) définissent ce comportement.

Figure 77 : Structure initiale de *MdefAs*

La Figure 78 illustre la façon dont un méta-arc est défini. Elle décrit la définition du

type d'instanciation (`instOf`) entre les objets (`Objet`) et les types statiques d'objets (`ObjStaticType`). Le méta-arc `instOf` de M2 se conforme à `MMetaArc` et est rattaché à sa structure initiale `Structure-instOf`. Cette structure permet à un objet d'être vu comme une instance d'au moins un type statique d'objets.

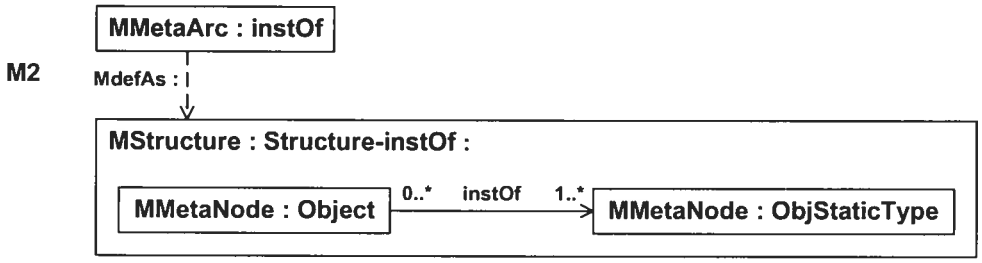


Figure 78 : Définition d'un méta-arc de M2

Inspiré du rôle des modèles de conditions que l'exemple 6 (page 39) a illustré, `MIfThenModel` sert à contextualiser des préconditions et postconditions dans la représentation des règles sémantiques des métamodèles. L'élément `MIfThenModel` est présenté plus en détail à la section 5.2.2.6 (page 105).

Nous présentons maintenant les autres méta-arcs qui permettent de gérer les modèles (au niveau méta) et leur contenu.

MdefIn, Mcontain, Mextend

Un élément est défini (`MdefIn`) dans un et un seul modèle (Figure 79-(a)). Et un modèle peut contenir (`Mcontain`) plusieurs éléments (Figure 79-(b)). Si un élément est défini dans un modèle, ce dernier contient le premier. Autrement dit, le méta-arc `Mcontain` inverse le méta-arc `MdefIn`, ce qui est indiqué par un arc de type `Minverse` de `Mcontain` à `MdefIn` (Figure 80).

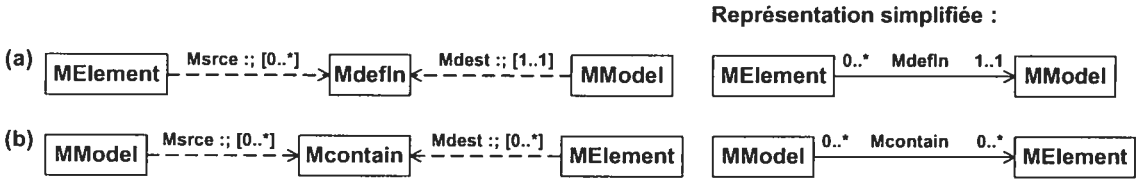


Figure 79 : (a) - Structure initiale de *MdefIn*; (b) - Structure initiale de *Mcontain*

Figure 80 : *Mcontain*, *MdefIn*

Concernant l'extension entre modèles, un modèle peut étendre (`Mextend`) d'autres

modèles et plusieurs modèles peuvent étendre un même modèle (Figure 81). Si un modèle (m_1) étend un autre modèle (m_2), tous les éléments de m_2 sont considérés comme faisant partie de m_1 (cf. la Règle 14, Annexe IV). C'est pour cette raison qu'un élément est défini dans un et un seul modèle (cf. la Figure 79-(a)) mais qu'il peut être contenu dans plusieurs modèles (cf. la Figure 79-(b)).

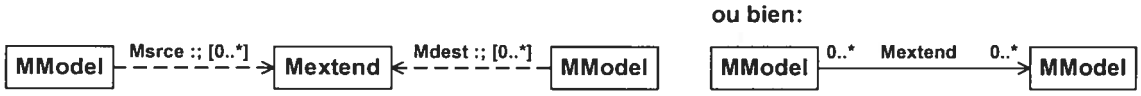


Figure 81 : Structure initiale de *Mextend*

Les méta-arcs `MdefIn`, `Mcontain`, et `Mextend` sont antiréflexifs et acycliques (cf. les Règle 6 et Règle 7, Annexe IV). Également, le méta-arc `MdefIn` est restreint par les Règle 9, Règle 11 et Règle 12 (Annexe IV), et `Mcontain` est restreint par les Règle 10, Règle 11 et Règle 12 (Annexe IV).

Mjoin, Mdiff, Mintersection, Mresult, Minfer, Mrestrict

Ces méta-arcs permettent de représenter d'une manière explicite, au niveau méta, des opérations entre modèles telles que celles mentionnées dans le besoin «Liens avec les modèles» (BesoinM3 7).

Les méta-arcs `Mjoin`, `Mdiff`, `Mintersection` visent respectivement à représenter les opérations *jointure*, *différence*, et *intersection* entre modèles. Ces méta-arcs sont les sous-types de `MmodelOp`, le type abstrait utilisé pour regrouper les méta-arcs représentant les opérations binaires entre les modèles et une telle opération retourne un et un seul modèle comme résultat. Les modèles résultants de ces opérations sont indiqués par des arcs de type `Mresult`. Par le sous-typage, les structures de `Mjoin`, `Mdiff`, `Mintersection`, `Mresult` sont donc déduites des structures de `MmodelOp` et `Mresult` illustrées dans la Figure 82.

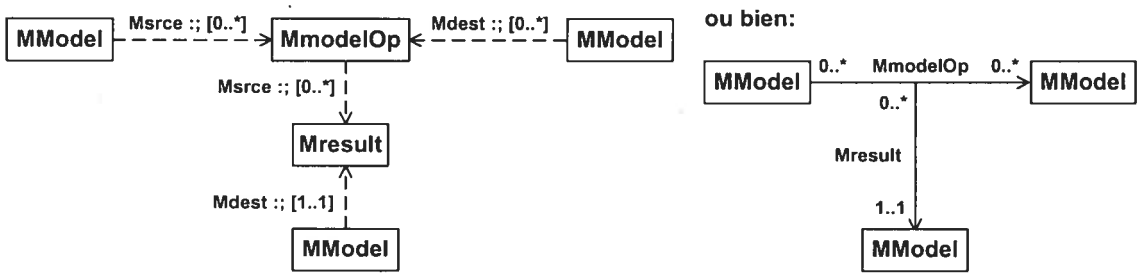


Figure 82 : Structures initiales de *MmodelOp*, *Mresult*

Les opérations *jointure*, *différence*, et *intersection* entre modèles sont spécifiées en

détail à la section 7.2.1 (page 200). À noter que les méta-arcs `Mjoin` et `Mintersection` qui implémentent les opérations *jointure* et *intersection* sont symétriques (cf. la Règle 21, Annexe IV).

Les méta-arcs `Minfer` (cf. la Figure 83-(a)) et `Mrestrict` (cf. la Figure 83-(b)) visent respectivement les opérations l'*inférence* et la *restriction* entre modèles.

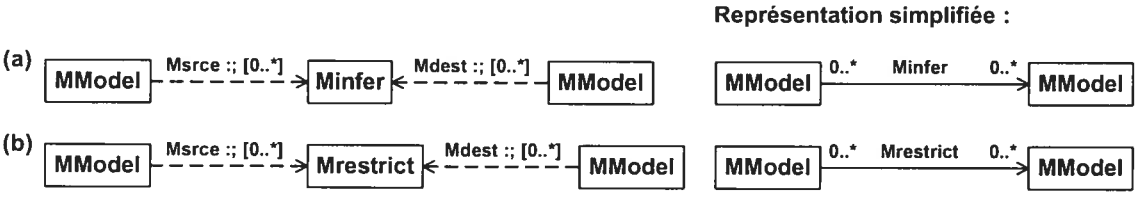


Figure 83 : Structures initiales de *Minfer*, *Mrestrict*

Le fait qu'un modèle soit relié par un arc de type `Minfer` (cf. la Figure 83-(a)) à un autre indique que le dernier est inféré du premier par les règles d'inférence, y compris l'effet de sous-typage. Par exemple, à partir de la structure de `Minfer` (`MModel` – `Minfer` – `MModel`) décrite dans la Figure 83-(a), nous pouvons appliquer la règle de restriction de type (la Règle 1, Annexe IV) pour inférer d'autres structures dont (`MStructure` – `Minfer` – `MModel`), (`MModel` – `Minfer` – `MStructure`), (`MStructure` – `Minfer` – `MStructure`), etc. La règle de respect de contrainte de cardinalité (la Règle 8, Annexe IV) est présente même si les contraintes de cardinalité relatives à une structure (inférée d'une autre structure par la règle de restriction de type) ne sont pas captées.

Le fait qu'un modèle soit relié par un arc de type `Mrestrict` à un autre modèle spécifie que le premier est plus restreint/spécifique que le dernier. Par exemple, la structure du méta-arc `Mname` illustrée dans la Figure 73-(b) (page 101) est plus restreinte/spécifique que celle dans la Figure 73-(a) (page 101). La spécialisation entre les modèles au niveau méta est présentée en détail à la section 7.2.2.1 (page 208). Le méta-arc `Mrestrict` est transitif et acyclique (cf. les Règle 5 et Règle 7, Annexe IV).

5.2.2.6 Représentation de règles

MRule, MIfThenModel, MRef, MEveryRef, Mif, Mthen

Ces types visent la représentation des règles sémantiques de métamodèles. Une règle (`MRule`) est attachée à sa partie de préconditions (`MIfThenModel`) et à sa partie de postconditions (`MIfThenModel`) respectivement par des arcs de type `Mif`/`Mthen`

(Figure 84). Pour le traitement des variables dans les règles, nous avons défini les méta-éléments `MRef` et `MEveryRef`. `MRef` correspond à l'interprétation du quantificateur existentiel (∃) alors que `MEveryRef` correspond à celle du quantificateur universel (∀). Une variable peut être remplacée par un élément concret dépendamment des cas d'application. La Règle 19 (Annexe IV) sur l'attribution d'éléments à une variable doit être respectée.

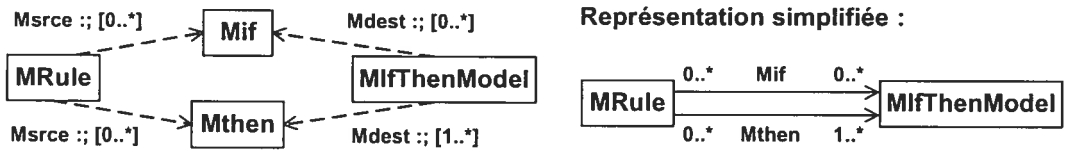


Figure 84 : Structures initiales de *Mif, Mthen*

Exemple de représentation de règles sémantiques

Afin de spécifier que pour tout méta-nœud, tous ses sous-types ne sont que des méta-nœuds, nous avons établi la règle suivante: si (i) `x` *est un méta-nœud, et* `y` *est un sous-type de* `x`*, alors (ii)* `y` *est un méta-nœud. Cette règle peut être modélisée telle qu'illustrée par la Figure 85.*

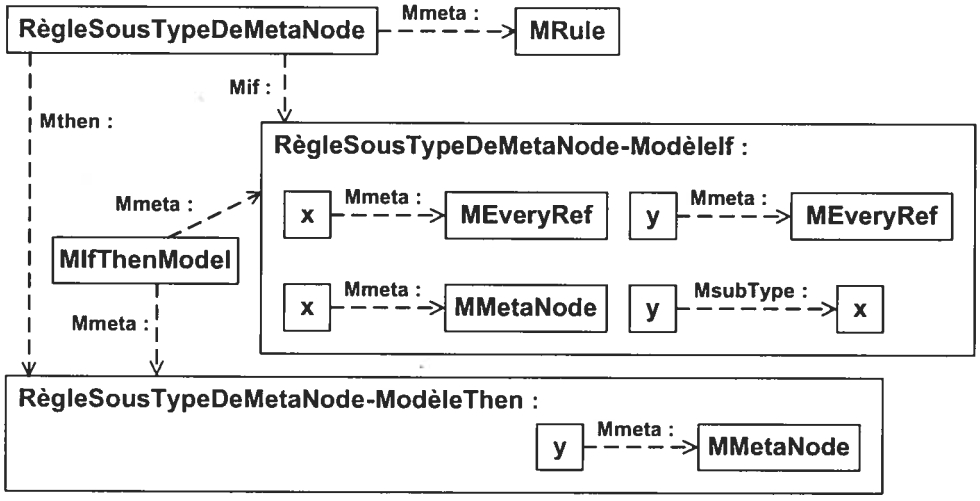


Figure 85 : Règle sur les sous-types des méta-nœuds

Dans cette figure, l'élément `RègleSousTypeDeMetaNode` *(représentant la règle) est associé par un arc de type* `Mif` *à* `RègleSousTypeDeMetaNode-ModèleIf`*, un modèle de conditions contextualisant la partie de pré-conditions: l'expression (i). Cet élément est aussi associé par un arc de type* `Mthen` *à* `RègleSousTypeDeMetaNode-ModèleThen`*, un modèle de conditions contextualisant la partie de post-conditions: l'expression (ii).*

Le modèle `RègleSousTypeDeMetaNode-ModèleIf` *exprime l'expression (i). L'élément* `x` *représente une variable désignant n'importe quel méta-nœud. L'élément* `x` *est déclaré comme conforme à* `EveryRef`*, ce qui est indiqué par un arc de type* `Mmeta` *de* `x` *à*

EveryRef. Le fait que x *désigne un méta-nœud est indiqué par un arc de type* Mmeta *de* x *à* MMetaNode. *L'élément* y *représente une variable désignant n'importe quel sous-type de* x. y *est donc déclaré comme conforme à* EveryRef, *ce qui est indiqué par un arc de type* Mmeta *de* y *à* EveryRef. *Le fait que* y *désigne un sous-type de* x *est indiqué par un arc de type* MsubType *de* y *à* x.

Le modèle RègleSousTypeDeMetaNode- ModèleThen *exprime l'expression (ii). Le fait que* y *désigne un méta-nœud est indiqué par un arc de type* Mmeta *de* y *à* MMetaNode.

La Règle 20 (Annexe IV) illustre la combinaison des modèles de conditions présents dans la partie de préconditions/postconditions d'une règle. À titre d'exemple, soit *RègleA* une règle qui n'a aucun arc de type Mif et qui est attachée par des arcs de type Mthen aux deux modèles de conditions *ModelIfThen*$_1$ et *ModelIfThen*$_2$ (Figure 86). Alors, la *RègleA* est interprétée comme (*ModelIfThen*$_1$ ou *ModelIfThen*$_2$). On peut voir ainsi que la combinaison (*ModelIfThen*$_1$ ou *ModelIfThen*$_2$) est toujours vrai.

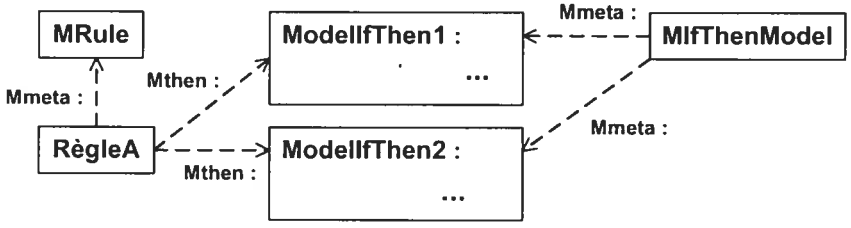


Figure 86 : Règle avec plusieurs modèles de post-conditions alternatifs

Mfulfil, Mwhere

Mfulfil s'utilise pour indiquer des règles que les méta-éléments doivent vérifier (Figure 87). Par exemple, comme l'illustre la Figure 88, le méta-arc MsubType doit vérifier la règle RègleSousTypeDeMetaNode. Il est donc associé à cette règle par un arc de type Mfulfil.

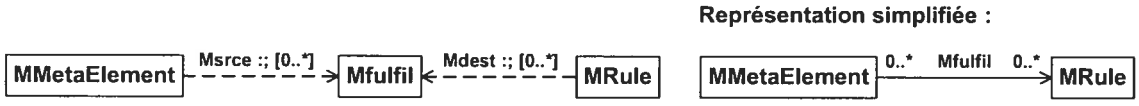


Figure 87 : Structure initiale de *Mfulfil*

Figure 88 : Exemple de *Mfulfil*

De plus, à l'aide du méta-arc Mwhere, il est possible de représenter les cas où une règle peut aussi être personnalisée en remplaçant d'abord, une ou des variables dans cette

règle par des éléments concrets, puis être appliquée à une situation. Par exemple, comme le montre la Figure 89, l'élément `MNode` doit vérifier la règle spécifiant que tous les sous-types de `MNode` sont des méta-nœuds. Remplacer x par `MNode`, la règle `RègleSousTypeDeMetaNode` (cf. la Figure 85, page 106) exprime que tous les sous-types de `MNode` sont des méta-nœuds. Donc, dans la Figure 89, `MNode` est associé par un arc de type `Mfulfil` à la règle `RègleSousTypeDeMetaNode` où (ce qui est indiqué par un arc de type `Mwhere`) `MNode` est attribué à la variable x (ce qui est indiqué par un arc de type `MassignedTo`).

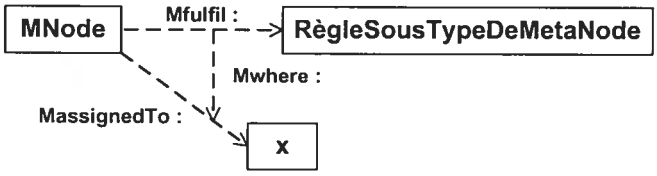


Figure 89 : Exemple de *Mfulfil*, *MassignedTo*, *Mwhere*

La Figure 90 montre la structure de `Mwhere`. En remplaçant le méta-arc `Massign` dans cette structure par ses sous-types (c'est-à-dire par `MassignedTo`, `MvalueIn`, `MnoValueIn`), nous obtenons d'autres structures de `Mwhere`.

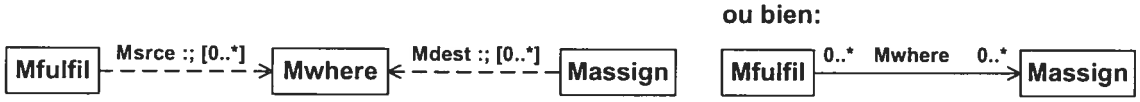


Figure 90 : Structure initiale de *Mwhere*

La sémantique de `Mwhere` est spécifiée comme suit. Soit r_1 un arc de type `Mfulfil` qui attache un élément (soit *eA*) à une règle (soit *RègleB*); soit r_2 un `Massign` (précisément, un arc de type `MassignedTo` / `MvalueIn` / `MnoValueIn`) qui contraint des éléments attribuables à une variable (soit *variableX*) contenue dans *RègleB*. Un arc de type `Mwhere` de r_1 à r_2 spécifie que l'élément *eA* doit remplir la règle *RègleB* où la variable *variableX* est contrainte par les éléments indiqués par r_2. La Règle 19 (Annexe IV) concernant l'attribution d'éléments à une variable s'applique.

Massign, MassignedTo, MvalueIn, MnoValueIn, MList, MListOfDiffEle, MmemberOf

Le méta-arc `Massign` est abstrait. Il regroupe les méta-arcs `MassignedTo`, `MvalueIn` et `MnoValueIn` qui, relativement au traitement des variables, représentent des fonctions d'affectation d'éléments concrets à des variables. Un arc de type `MassignedTo` associant un élément (`MElement`) à une variable spécifie que cette

variable est remplacée par cet élément. Un arc de type `MvalueIn` associant une variable à une liste (`MList`) spécifie que dans ce contexte, seulement les membres de cette liste sont les éléments attribuables à cette variable. Un arc de type `MnoValueIn` d'une variable à une liste spécifie qu'aucun membre de cette liste n'est attribuable à cette variable.

Concernant la spécification des listes, un `MList` représente au niveau méta une liste d'éléments. Plus spécifique, un `MListOfDiffEle` représente une liste dont les membres sont distincts deux à deux. Le méta-arc `MmemberOf` permet de représenter qu'un élément identifié (`MNamedNode`) est membre d'une liste (`MList`).

5.2.2.7 Négation logique

Le méta-arc `Mnot` implémente la négation logique, symétrique (cf. la Règle 21, Annexe IV) mais antiréflexive (cf. la Règle 6, Annexe IV), entre les méta-arcs (Figure 91). Si deux méta-arcs sont reliés par un arc de type `Mnot`, alors l'un est la négation de l'autre et vice-versa. Par exemple, `MnoValueIn` est la négation logique de `MvalueIn` et inversement, `MvalueIn` est la négation logique de `MnoValueIn`.

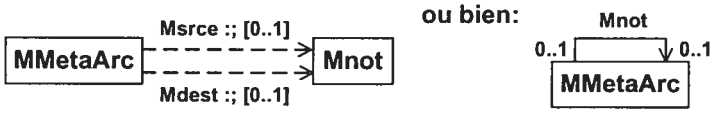


Figure 91 : Structure initiale de *Mnot*

5.2.2.8 Représentation d'usagers

Pour la gestion administrative de la base de modèles, nous avons introduit la notion d'utilisateur et d'interactions avec les objets de connaissances. Pour cette fin, le méta-nœud `MUser` représente les utilisateurs (aux niveaux M3 et/ou M2) qui créent, dans le système, des objets de connaissances. Le méta-arc `Mcreate` permet d'indiquer qu'un élément (`MElement`) est créé par un utilisateur (`MUser`) dans la base de connaissance (Figure 92). Par exemple, la Figure 93 montre qu'au niveau M2, c'est l'utilisateur *admin* qui crée l'élément `Object`. Un utilisateur peut créer plusieurs éléments mais un élément ne peut être créé que par un seul utilisateur.

Figure 92 : Structure initiale de *Mcreate*

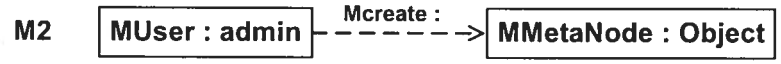


Figure 93 : Création d'éléments

5.2.2.9 Autres utilités

Nous définissons d'autres éléments dans le but de supporter d'autres fonctionnalités telles que la comparaison entre les éléments, la représentation de termes équivalents et la représentation de types abstraits.

Comparaison entre éléments

Le méta-arc abstrait `Mcompare` est à la racine de la hiérarchie des méta-arcs qui implémentent les opérateurs des opérations de comparaison entre deux éléments au niveau méta. Il est subdivisé en: `M=` (égal), `M!=` (inégal, ou différent) et `MforValues`. Le méta-arc `MforValues` est abstrait et regroupe: `M>` (supérieur), `M<` (inférieur), `M<=` (inférieur ou égal) et `M>=` (supérieur ou égal). `M==` est un sous-type de `M=` et vise à représenter l'identité. Ainsi, un arc de type `M==` entre un élément e_1 et un élément e_2 indique que e_1 et e_2 désignent le même élément.

Nous pouvons faire toutes les comparaisons (c'est-à-dire les comparaisons: `M=`, `M!=`, `M>`, `M<`, `M<=`, `M>=`) entre deux entiers (`MLabelledInteger`). Cependant, seulement la comparaison d'égalité (`M=`) et celle de différence (`M!=`) sont autorisées pour tous les éléments (`MElement`). Il est à noter que les méta-arcs `M=` et `M!=` sont symétriques (cf. la Règle 21, Annexe IV).

Représentation de termes équivalents

Un arc de type `Meqv` entre deux `MNode` exprime que ces derniers représentent des termes équivalents. Ceci permet de faire la correspondance entre les termes équivalents de différents modèles, aidant ainsi à l'intégration de ces modèles.

Représentation de types abstraits

`MBoolean` a seulement deux instances prédéfinies, `MTrue` et `MFalse`, représentant respectivement les valeurs logiques étiquetées : `MTrue` pour vrai, et `MFalse` pour faux. `MTrue` et `MFalse` sont interprétées respectivement comme «1» et «0» en langage universel, comme «true» et «false» en anglais et comme «vrai» et «faux» en français.

Le méta-arc `MisAbst` (Figure 94) permet d'indiquer qu'un méta-élément est abstrait ou non (cf. la Définition 3, page 11). Par défaut, un méta-élément est non abstrait. Un méta-élément lié à `MTrue` par un arc de type `MisAbst` sera un méta-élément abstrait.

Figure 94 : Structure initiale de *MisAbst*

5.2.3 Réflexivité

Cette section vise à montrer que notre méta-métamodèle se définit lui-même.

5.2.3.1 Le cœur réflexif de notre méta-métamodèle

Nous avons décrit dans la section 5.2.1 (page 94) la signification ainsi que le rôle des éléments de base du niveau M3 : `MElement`, `MNode`, `MArc`, `MMetaElement`, `MMetaNode`, `MMetaArc`, `MsubType`, `Mmeta`, `Msrce`, et `Mdest`. Comme l'a montré la section 5.2.2 (page 96), ces éléments de base permettent de définir les autres éléments du niveau M3 ainsi que les éléments du niveau M2. Ces éléments de base forment le cœur réflexif de notre méta-métamodèle. Nous expliquons ci-après comment ces éléments de base se définissent eux-mêmes.

Liens de conformité entre les éléments de base

Étant donné que chaque méta-nœud ou méta-arc doit être conforme à `MMetaNode` ou `MMetaArc` respectivement, `MNode` et ses sous-types (y compris `MMetaElement`, `MMetaNode`, et `MMetaArc`) se conforment à `MMetaNode`, alors que `MArc` et ses sous-types (y compris `Msrce`, `Mdest`, `Mmeta`, `MsubType`) se conforment à `MMetaArc`. Si le méta-élément `MElement` n'est ni un méta-nœud ni un méta-arc, il se conforme donc à `MMetaElement`. Les liens de conformité entre les éléments de base de M3 sont représentés dans la Figure 95.

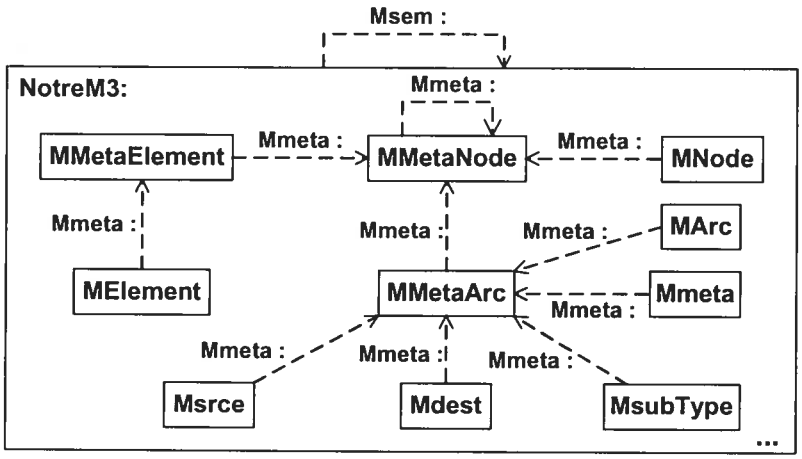


Figure 95 : Les éléments de base - le cœur de notre méta-métamodèle

Structures des méta-arcs de base: MArc, Msrce, Mdest, MsubType, Mmeta

MArc

Comme nous l'avons présenté dans la section 5.2.1 (page 94), le méta-arc `MArc` lie le méta-élément `MElement` à `MElement` lui-même via un arc de type `Msrce` et un arc de type `Mdest` (cf. la Figure 60, page 95).

Msrce, Mdest

Comme nous l'avons décrit, les éléments `Msrce` et `Mdest` permettent de spécifier les sources et les destinations des méta-arcs. Suivant la règle d'existence entre un arc et son méta-arc, un arc d'un élément source à un élément destination peut exister si et seulement si son méta-arc est prédéfini pour relier le méta-élément source au méta-élément destination correspondant. Similairement à la situation décrite dans la Figure 62 (page 96), dans la structure du méta-arc `MArc` (cf. la Figure 60, page 95), l'arc de type `Msrce` ou `Mdest` de `MElement` à `MArc` peut exister si et seulement si le méta-arc `Msrce` ou `Mdest` est prédéfini pour relier `MMetaElement` à `MMetaArc`. Ceci est illustré par la Figure 96.

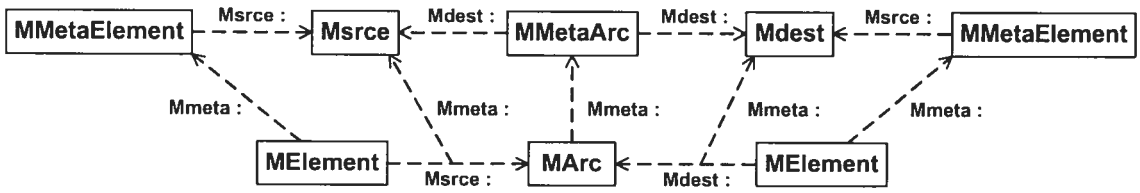


Figure 96 : Rapport existentiel entre un arc de type *Msrce* / *Mdest* et son méta-arc

Puisque `MMetaElement` représente l'ensemble de tous les méta-éléments, les structures de `Msrce/Mdest` dans lesquelles les méta-arcs `Msrce/Mdest` sont définis pour relier `MMetaElement` à `MMetaArc` (cf. la Figure 96) nous permettent de définir tous les types de méta-arcs: un méta-arc entre deux méta-éléments où un méta-élément peut être un méta-nœud ou méta-arc. Ceci signifie également que nous pouvons définir tous les types d'arcs : un arc entre deux nœuds, entre deux arcs ou entre un nœud et un arc. Le pouvoir d'expression du formalisme augmente alors considérablement. C'est pour cette raison que les structures de `Msrce/Mdest` sont celles indiquées à la Figure 97. Ces structures spécifient la définition de tous les méta-arcs: un `MMetaArc` unit deux `MMetaElement` via un arc de type `Msrce` et un arc de type `Mdest`. Cette définition permet d'expliquer aussi comment la direction d'un arc/méta-arc est établie. Tous les éléments conformes à `MMetaArc` se conforment à cette définition.

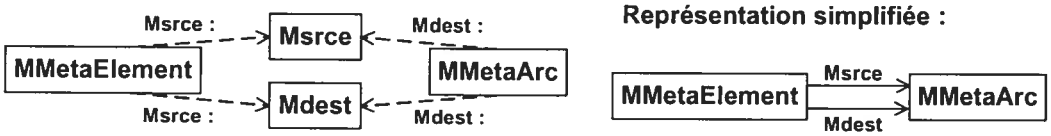


Figure 97 : Structures initiales de *Msrce*, *Mdest* (sans contraintes de cardinalités)

MsubType

`MsubType` implémente la relation de sous-typage entre les méta-éléments. Afin que les arcs de type `MsubType` puissent exister entre les méta-éléments, le méta-arc `MsubType` doit être défini pour relier l'élément `MMetaElement` à `MMetaElement` lui-même via un arc de type `Msrce` et un arc de type `Mdest` (Figure 98).

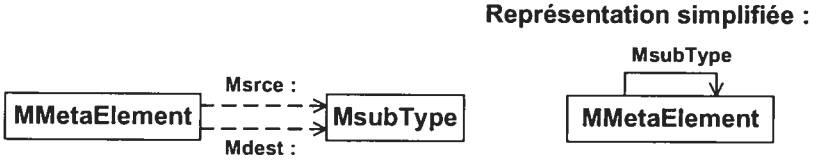


Figure 98 : Structure initiale de *MsubType* (sans contraintes de cardinalités)

Comme `MMetaNode` et `MMetaArc` sont des sous-types de `MMetaElement`, la structure initiale de `MsubType` (Figure 98) autorise l'existence d'arcs de type `MsubType` entre les `MMetaElement`, y compris les `MMetaNode` et les `MMetaArc`. En principe, un `MMetaNode` (respectivement un `MMetaArc`) ne peut cependant pas être associé à un `MMetaArc` (respectivement un `MMetaNode`) par un arc de type `MsubType` (cf. la Règle 2, Annexe IV). Puisque le type relationnel de sous-typage entre est transitif, antiréflexif, et acyclique (cf. la Définition 27, page iv-1, Annexe IV), le méta-arc `MsubType` remplit les Règle 5, Règle 6 et Règle 7 (Annexe IV).

Mmeta

Comme les arcs de type `Mmeta` associent les éléments au niveau méta à leurs méta-éléments, le méta-arc `Mmeta` associe l'élément `MElement` à `MMetaElement` via un arc de type `Msrce` et un arc de type `Mdest` (Figure 99-(a)). Puisqu'un nœud ou un arc doit être conforme respectivement à un méta-nœud ou à un méta-arc (cf. la Règle 3, Annexe IV), un `MNode` ou un `MArc` doit être associé respectivement à un `MMetaNode` ou à un `MMetaArc` par un arc de type `Mmeta`. Ceci peut être spécifié par les structures plus restrictives de `Mmeta` (illustrées dans les Figure 99-(b) et Figure 99-(c)) qui sont déduites de sa structure initiale (illustrée dans la Figure 99-(a)).

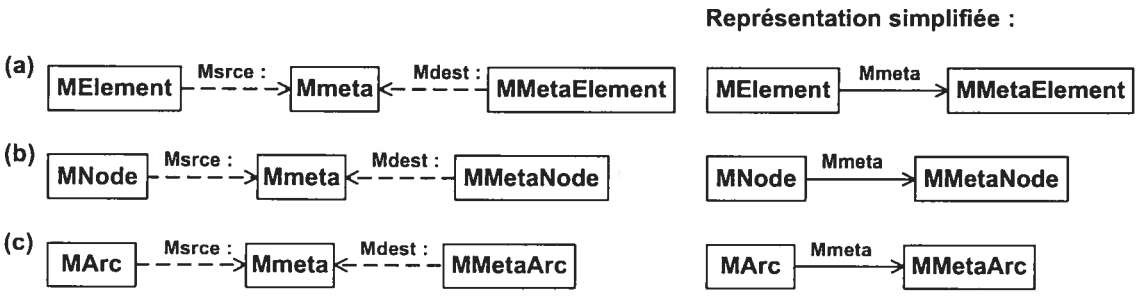


Figure 99 : Structures de *Mmeta* (sans contraintes de cardinalités) - (a) Structure initiale; - (b),(c) Structures plus restrictives

Réflexivité du noyau constitué des éléments de base de notre méta-métamodèle

Les liens de conformité entre les éléments de base et les structures des méta-arcs de base, que nous venons de décrire, mettent en évidence que chacun de ces éléments de base est défini en se basant sur ces éléments eux-mêmes, conformément à la sémantique et à la définition de ces éléments. Ceci signifie que le noyau constitué de ces éléments de base, dit aussi le *cœur* de notre méta-métamodèle, est réflexif.

5.2.3.2 Réflexivité de notre méta-métamodèle

Le noyau constitué des éléments de base abordés ci-dessus permet d'expliquer la façon dont sont définis les éléments (nœuds, arcs, méta-nœuds, méta-arcs) au niveau méta au sein de notre formalisme. Une illustration se trouve dans l'exemple 7.

Définition d'un méta-arc (exemple 7)

La Figure 100 montre la structure de `Msem` *que la Figure 76 (page 102) a illustrée.*

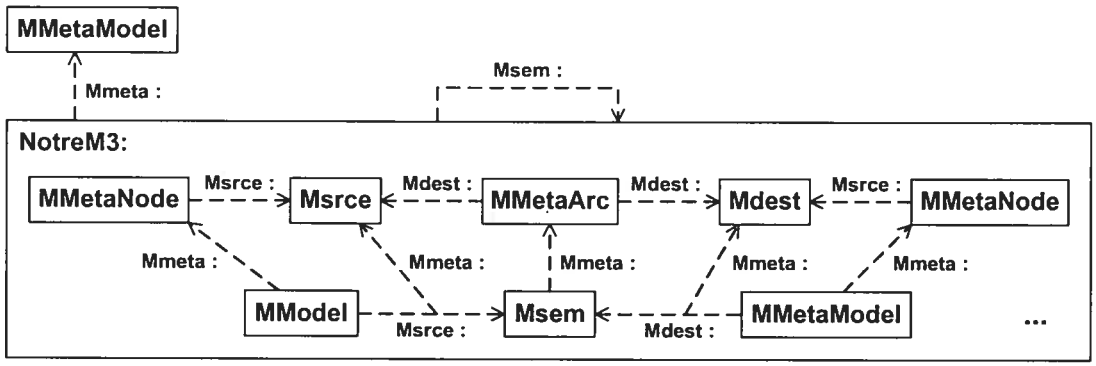


Figure 100 : Définition d'un méta-arc

Dans cette figure, le méta-arc `Msem` *lie le méta-nœud* `MModel` *au méta-nœud* `MMetaModel` *via un arc de type* `Msrce` *et un arc de type* `Mdest`. `MModel` *et* `MMetaModel` *sont conformes à* `MMetaNode`. `Msem` *est conforme à* `MMetaArc`. *L'arc de type* `Msrce` *entre* `MModel` *et*

Msem est conforme sémantiquement au méta-arc Msrce entre deux méta-éléments correspondants, MMetaNode et MMetaArc. De même, l'arc de type Mdest entre MMetaModel et Msem est conforme sémantiquement au méta-arc Mdest entre deux méta-éléments correspondants, MMetaNode et MMetaArc.

Notre méta-métamodèle est composé du noyau mentionné précédemment auquel s'ajoutent les éléments du niveau M3 présentés dans la section 5.2.2 (page 96). Chacun de ces éléments est défini en se basant sur ce noyau, tel que démontré par l'exemple 7. Les éléments ajoutés permettent d'enrichir le pouvoir d'expression du méta-métamodèle, et d'exprimer ainsi davantage la sémantique des éléments au niveau méta au sein de notre formalisme (y compris ceux du méta-métamodèle). À titre d'exemple, le méta-arc Msem est défini pour représenter les liens de conformité entre les modèles au niveau méta et leurs métamodèles. Et, comme illustré à la Figure 100, la structure de Msem nous permet de relier NotreM3 (l'élément conforme à MMetaModel et contextualisant notre méta-métamodèle) à lui-même par un arc de type Msem, pour indiquer que notre méta-métamodèle (NotreM3) est le métamodèle de lui-même.

Nous pouvons conclure ainsi que notre méta-métamodèle se définit lui-même.

5.2.4 Diagramme d'associations complet de M3

Les Figure II-1 (Annexe II) et Figure II-2 (Annexe II) illustrent respectivement la hiérarchie de spécialisation des méta-nœuds et celle des méta-arc de notre méta-métamodèle. Le diagramme d'associations entre tous les éléments de notre M3 est illustré dans la Figure II-3 (Annexe II). Les contraintes de cardinalité y sont également indiquées. Les règles sémantiques applicables à M3 sont présentées dans l'Annexe IV.

5.3 Métamodèle

Le métamodèle du niveau M2 se conforme au méta-métamodèle présenté dans le chapitre précédent. Le métamodèle (M2) représente le vocabulaire et la grammaire au moyen desquels les usagers pourront spécifier leurs modèles (de connaissances) au niveau M1.

Les éléments définis au niveau M1 seront des instances se conformant aux méta-éléments définis dans ce métamodèle. En fonction de leurs sémantiques, les types (c'est-à-dire les méta-éléments) de M2 forment une hiérarchie de spécialisation où Element

est au sommet. `Element` est similaire au type `MElement` de M3. `Element` est conforme à `MMetaElement`. Il est le super-type de tous les types définis dans M2 et représente donc l'ensemble de tous les éléments appartenant au niveau M1. Cet ensemble est subdivisé en deux groupes: les nœuds (`Node`), et les arcs (`Arc`). `Node` et `Arc` sont les deux types à la racine respectivement de la hiérarchie de tous les méta-nœuds de M2 (Figure III-1, page iii-1, Annexe III) et celle de tous les méta-arcs de M2 (Figure III-2, page iii-6, Annexe III). `Element`, `Node`, `Arc` sont des types abstraits. Les méta-nœuds se conforment à `MMetaNode`. Les méta-arcs se conforment à `MMetaArc`.

Nous aborderons maintenant les éléments (les méta-nœuds et les méta-arcs) principaux de M2 ainsi que leurs interactions. Ces éléments sont présentés en fonction de leur utilité, par exemple pour: représenter la conformité entre éléments et méta-éléments, nommer des éléments, interpréter des éléments, représenter les types et les instances de M1, représenter des modèles et leurs liens, ou représenter des règles, etc. Cette présentation-ci met plutôt l'accent sur les raisons d'être de ces éléments et de leurs interactions. Les spécifications supplémentaires du métamodèle se trouvent dans l'Annexe III.

5.3.1 Liens de conformités entre les éléments de M1 et les méta-éléments de M2

Si le méta-arc `Mmeta` représente l'ensemble des liens de conformité entre les éléments de niveau M2/M3 et leurs méta-éléments de M3, le méta-arc `meta` représente l'ensemble des liens de conformité entre les éléments de M1 et leurs méta-éléments de M2 (Figure 101).

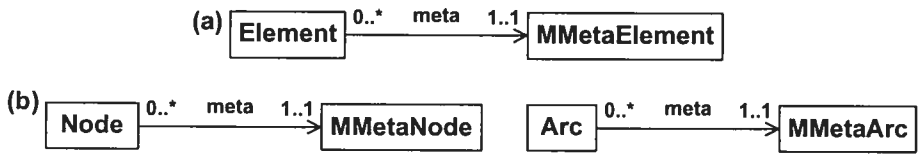


Figure 101 : Méta-arc *meta* - (a) Structure initiale ; (b) Structures plus restrictives

Chaque élément (nœud, arc) de M1 est conforme à un et un seul méta-élément (méta-nœud, méta-arc) de M2. Il est attaché à ce méta-élément par un lien de conformité de type `meta`.

5.3.2 Nommage d'éléments

D'une manière similaire à M2/M3, les éléments `Label` et `name` permettent le

nommage d'éléments de niveau M1. Un libellé (`Label`) peut nommer (`name`) au maximum un élément (`Element`) et un élément peut être nommé par au maximum un libellé (Figure 102-(a)). De façon plus restrictive, chaque nœud nommé (`NamedNode`) doit être lié à son libellé (Figure 102-(b)).

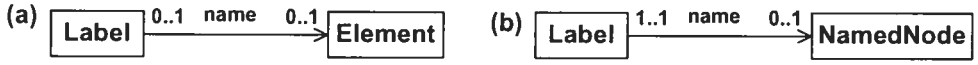


Figure 102 : Méta-arc *name* – (a) Structure initiale ; (b) Structure plus restrictive

5.3.3 Interprétation d'éléments

Similaires aux éléments `MValue`, `MLanguage`, `Mdepend`, `MvalueOf` de M3 appliqués aux niveaux M3 et M2, les éléments `Value`, `Language`, `depend`, `valueOf` de M2 (Figure 103) permettent le multilinguisme dans la représentation des interprétations des éléments de niveau M1.

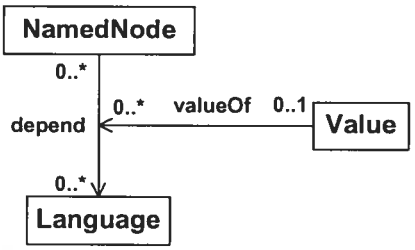


Figure 103 : Structures initiales de *depend*, *valueOf*

5.3.4 Représentation de types et instances de M1

Les méta-nœuds abstraits `Type` et `TypeInstance` ont pour but de catégoriser les éléments au niveau M1 suivant deux niveaux: les types (`Type`), et les instances de types (`TypeInstance`). `Type` est subdivisé en deux sous-types: `Attribute`, `RolePlayerType`. Et `TypeInstance` est subdivisé en: `AttrInst`, `RolePlayer`.

Les `Attribute` représentent les attributs servant à caractériser des `Type`. Les `AttrInst` représentent les instances d'attributs, ou bien les valeurs d'attributs, servant à caractériser des `TypeInstance`/`Type`. Les méta-nœuds abstraits `RolePlayer` et `RolePlayerType` représentent respectivement l'ensemble de joueurs de rôles et celui de types de joueurs de rôles. Les joueurs de rôles (`RolePlayer`) sont divisés en groupes: les objets (`Object`), les rôles (`Role`), les relations (`Relation`), et les contextes de relations (`Context`). Respectivement, les types de joueurs de rôles

(RolePlayerType) sont divisés en groupes: les types statiques/dynamiques d'objets (ObjType / ObjStaticType / ObjDynType), les types statiques/dynamiques de rôles (RoleType / RoleStaticType / RoleDynType), les types de relations (RelationType) et les structures de types de relations (Structure).

Dans le reste de la présente section, nous présentons plus en détail la signification de ces éléments ainsi que leurs comportements.

5.3.4.1 Objets et types d'objets

Object, ObjType/ObjStaticType/ObjDynType

Un Object représente un objet de l'univers du discours ou dans le monde réel; exemples: la personne Anna, un pays, un dragon, etc.

Un type d'objet désigne un ensemble d'objets possédant les mêmes propriétés (attributs, relations et comportement). Donc, le type abstrait ObjType représente l'ensemble des types d'objets. Celui-ci est subdivisé en deux: (a) les types statiques d'objets et (b) les types dynamiques d'objets. Sous-types de ObjType, les méta-nœuds ObjStaticType et ObjDynType distinguent ces ensembles (a) et (b). La Définition 9 permet d'expliciter la nature *statique* ou *dynamique* d'un type.

Définition 9 : Type statique/dynamique

Un type est dit *statique* si chaque élément déclaré de ce type demeure, tout au long de son existence, comme une instance de ce type. Par contre, un type est dit *dynamique* si: un élément n'étant pas de ce type peut migrer vers ce type et/ou si un élément de ce type peut ne plus rester comme une instance de ce type.

instOf

instOf (Figure 104) permet de représenter les liens d'instanciation entre les objets et les types d'objets. Un objet peut être une instance de plusieurs types d'objets.

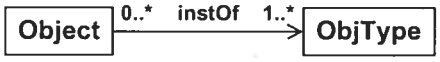


Figure 104 : Structure initiale de *instOf* pour les objets et les types d'objets

Exemple d'objets et leurs types d'objets

Comme l'illustre la Figure 105, l'élément Jean (de M1) désignant la personne Jean prend d'un côté Object (de M2) pour son méta-élément (ce qui est indiqué par un arc de type meta de Jean à Object et peut être représenté simplement sous forme «Object : Jean») et d'un autre côté Personne (de M1) pour son type d'objets (ce qui est indiqué par un arc de type

instOf de Jean à Personne). Personne est un type statique d'objets (ce qui est indiqué par un arc de type meta de Personne à ObjStaticType, et peut être représenté simplement sous forme «ObjStaticType : Personne»). Également, le fait que l'objet Jean soit une instance de Personne peut être représenté d'une manière simplifiée sous la forme «Object : Jean : Personne».

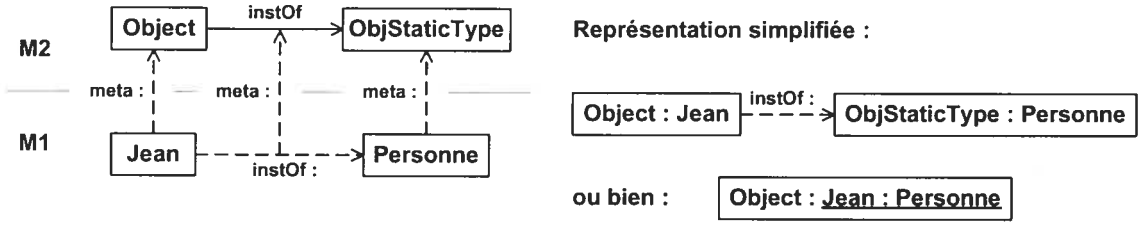


Figure 105 : Un objet et son type d'objets

D'une même façon, comme l'illustre la Figure 106, afin de représenter que Jean est adulte et marié, l'élément Jean est vu comme une instance de deux types dynamiques d'objets Adulte et PersonneMariée auxquels il est relié par des arcs de type instOf. Ceci peut être exprimé simplement sous forme «Object : Jean : Adulte» et «Object : Jean : PersonneMariée».

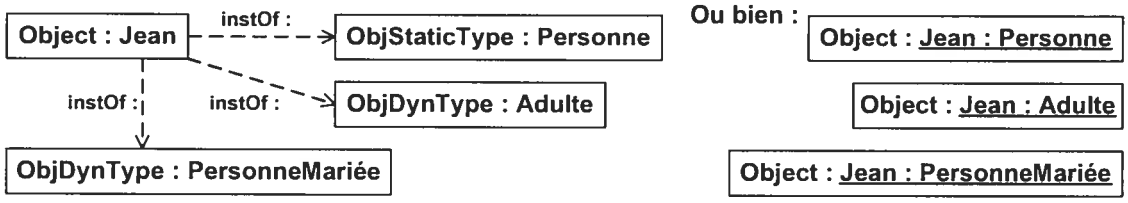


Figure 106 : Multi-classification

À noter la Notation 3 pour la représentation simplifiée des liens d'instanciation (`instOf`) entre les instances et les types de M1.

***Notation 3** : Notation pour les liens d'instanciation entre les instances et les types de M1*

Soient *TypeA* un type défini au niveau M1, *A* une instance de type *TypeA*. *A* s'écrit parfois sous la forme «*A : TypeA*» qui indique que l'élément *A* (de M1) est une instance conforme au type *TypeA* (de M1).

compose, composeType

Un `Object` (respectivement `ObjType`) peut être composé de d'autres `Object` (respectivement `ObjType`), ce qui est indiqué par des arcs de type `compose` (respectivement `composeType`) (Figure 107). À souligner qu'à chaque instant, un objet peut avoir plusieurs composants mais qu'il peut être composant d'au plus un objet. Le

méta-arc compose est antiréflexif et acyclique (cf. les Règle 6 et Règle 7, Annexe IV).

Figure 107 : Structures initiales de *composeType*, *compose*

5.3.4.2 Rôles et types de rôles

Role, RoleType/RoleStaticType/RoleDynType

Ces types ainsi que ObjDynType permettent de représenter les aspects temporaires, dynamiques des objets du monde réel. Voir l'exemple 3 (page 30).

Un Role représente un rôle qu'un RolePlayer peut jouer. Un RoleType représente un type de rôles. Similairement au cas de types d'objets, un type statique (dynamique) de rôles est conforme à RoleStaticType (RoleDynType). RoleStaticType et RoleDynType sont des sous-types de RoleType.

instOf

Ce méta-arc (Figure 108) permet aussi de représenter les liens d'instanciation entre les rôles et les types de rôles. Un rôle peut être une instance de plusieurs types de rôles.

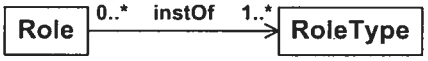


Figure 108 : Structure initiale de *instOf* pour les rôles et les types de rôles

playedByType, playedBy

Le méta-arc playedByType (Figure 109) vise à indiquer des types de joueurs de rôles (RolePlayerType) pour un type de rôles (RoleType).

Figure 109 : Structure initiale de *playedByType*

Le méta-arc playedBy sert à indiquer le joueur d'un rôle.

La première structure de playedBy (Figure 110-(1)) permet à des arcs de type playedBy d'associer des rôles (Role) à des joueurs de rôles (RolePlayer). Un joueur de rôles peut jouer plusieurs rôles en même temps. Par contre, un rôle peut être joué à chaque moment par un joueur de rôles au maximum. Puisqu'un rôle est vu aussi comme un joueur de rôle et qu'un rôle ne peut être le joueur de lui-même, le méta-arc playedBy est donc antiréflexif et acyclique (cf. les Règle 6 et Règle 7, Annexe IV).

Figure 110 : Structures initiales de *playedBy*

La deuxième structure de `playedBy` (Figure 110-(2)) est une extension de la première. Elle permet à des arcs de type `playedBy` d'associer des types de rôles (`RoleType`) à des joueurs de rôles (`RolePlayer`). Le fait qu'un type de rôles soit associé par un arc de type `playedBy` à un joueur de rôles constitue une représentation simplifiée du fait que le dernier est le joueur unique pour tous les rôles instanciés du type de rôles en question.

Exemple de rôles et de leurs joueurs de rôles

La Figure 111 représente que la personne `Jean` occupe deux rôles «employé» représentés par `EmpCAM124` et `EmpHEC124`. Ceci est indiqué par des arcs de type `playedBy` de `EmpCAM124` et de `EmpHEC124` à `Jean`.

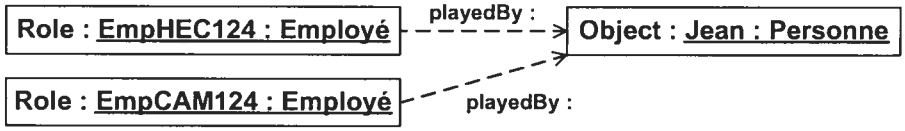


Figure 111 : Rôles et Joueurs de rôles

Si un `Role` (respectivement un `RoleType`) s'implique dans une relation (`Relation`) ou dans un type de relations (`RelationType`), alors il représente le rôle (respectivement le type de rôles) qu'un `RolePlayer` (respectivement un `RolePlayerType`) attaché à ce premier joue dans la relation ou dans le type de relations en question. Ceci aide à la distinction explicite des participants surtout dans une relation (`Relation`) ou un type de relations (`RelationType`) n-aires. Ce sujet est analysé plus loin dans la section suivante portant sur la représentation de relations et de types de relations.

5.3.4.3 Relations et types de relations

RelationType, Relation

Un `RelationType` représente un type de relations (*type relationnel*) entre les éléments de M1. Au niveau M1, une `Relation` représente au niveau des instances une relation instanciée d'un type de relations de M1. Chaque relation est classée en rapport à son modèle structurel tel que le nombre d'éléments qu'elle relie. Une relation peut être

unaire, binaire ou ternaire, etc.

Un `RelationType` est défini par sa ou ses structures de définition. Une `Relation`, un `RelationType` et un arc visent tous à lier des éléments mais il existe une différence de nature entre eux. Puisque `Relation` et `RelationType` sont des méta-nœuds conformes à `MMetaNode`, une `Relation` ou un `RelationType` est de nature un nœud donc est conforme à un méta-nœud. Par contre, un arc représente une liaison entre deux nœuds, entre deux arcs ou entre un nœud et un arc. Il se conforme à un méta-arc conforme à `MMetaArc`. La Notation 4 présente nos notations graphiques pour les relations et les types de relations de M1.

***Notation 4** : Notations graphiques pour relations et types de relations de M1*

Afin de faciliter graphiquement la distinction avec des nœuds d'autres types, une relation (`Relation`) ou un type de relations (`RelationType`) peut être représenté par un rectangle donc les coins sont arrondis. Par exemple, le type de relations `travailler` dans l'exemple 8 (page 122), la relation `jean-travailler` de type `travailler` dans l'exemple 12 (page 128).

relArcType/actType/objType

`relArcType` sert à indiquer des `Type` qui participent à un `RelationType` (Figure 112). Dans un `RelationType`, un participant étant un `Type` pourrait être spécifié, par exemple, comme (i) un acteur de l'action (c'est-à-dire qu'il exercera l'action) ou (ii) un objet de l'action (c'est-à-dire qu'il subira l'action). Sous-types de `relArcType`, les méta-arcs `actType` et `objType` visent respectivement à expliciter les cas (i) et (ii).

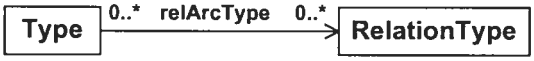


Figure 112 : Structure initiale de *relArcType*

Type de relations: travailler (exemple 8)

La Figure 113 montre une structure du type de relations `travailler`. Dans cette structure, `travailler` implique un seul type d'objets `Personne` en tant que l'acteur de l'action `travailler` (ce qui est indiqué par un arc de type `actType` de `Personne` à `travailler`).

Figure 113 : Une structure du type de relations *travailler*

actAsType

Un type de rôles peut avoir différents types de joueurs de rôles dépendamment des structures du type de relations dans lesquelles il s'implique. Le méta-arc `actAsType` permet de spécifier les types de joueurs de rôles (`RolePlayerType`) pour un type de rôles (`RoleType`) s'impliquant dans une structure d'un type de relations. Voir l'exemple suivant.

Type de relations: gérer (exemple 9)

La Figure 114 montre une structure du type de relations `gérer` entre des gestionnaires et des organisations. Dans ce type de relations, le type statique de rôles «gestionnaire» (`Gestionnaire`) est impliqué comme l'acteur de l'action (ce qui est indiqué par un arc, soit arc_1*, de type `actType` de `Gestionnaire` à `gérer`), et le type statique d'objets «organisation» (`Organisation`) est impliqué comme l'objet de l'action (ce qui est indiqué par un arc de type `objType` de `Organisation` à `gérer`). De plus, (i) le type de rôles `Gestionnaire` est joué par `PermEmp`, le type de rôles «employé permanent» (ce qui est indiqué par un arc, soit* arc_2*, de type `actAsType` de `PermEmp` à* arc_1*); (ii) le type de rôles `PermEmp` est joué par `Personne`, le type de personnes (ce qui est indiqué par un arc, soit* arc_3*, de type `actAsType` de `Personne` à* arc_2*).*

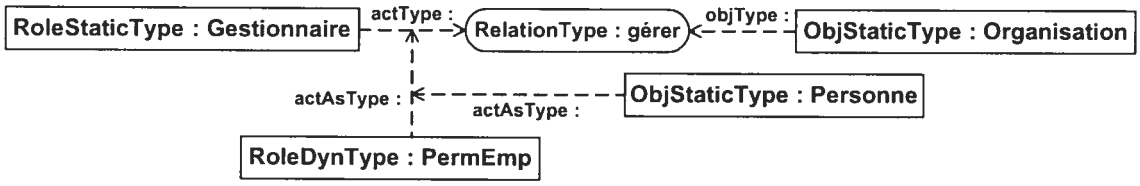


Figure 114 : Une structure du type de relations *gérer*

Afin de représenter le cas tel que (i) dans l'exemple 9 (cf. Figure 114), nous avons la structure suivante de `actAsType`: le méta-arc `actAsType` peut unir `RolePlayerType` à `relArcType` (Figure 115). Veuillez prendre note de la Règle 24 (Annexe IV).

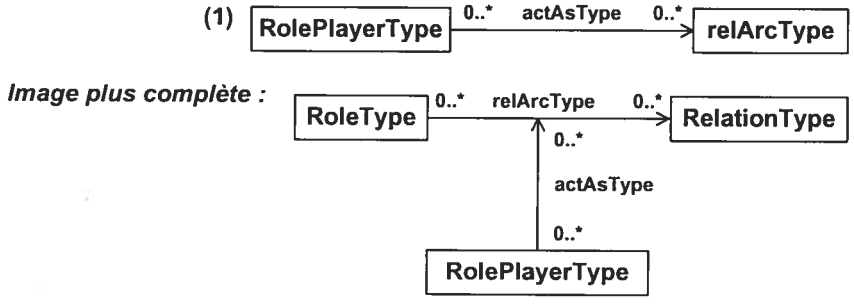


Figure 115 : Une structure initiale de *actAsType*

Pour représenter le cas tel que (ii) dans l'exemple 9 (cf. Figure 114), nous avons une autre structure de `actAsType`: le méta-arc `actAsType` peut unir `RolePlayerType` à `actAsType` (Figure 116). Veuillez prendre note de la Règle 25 (Annexe IV).

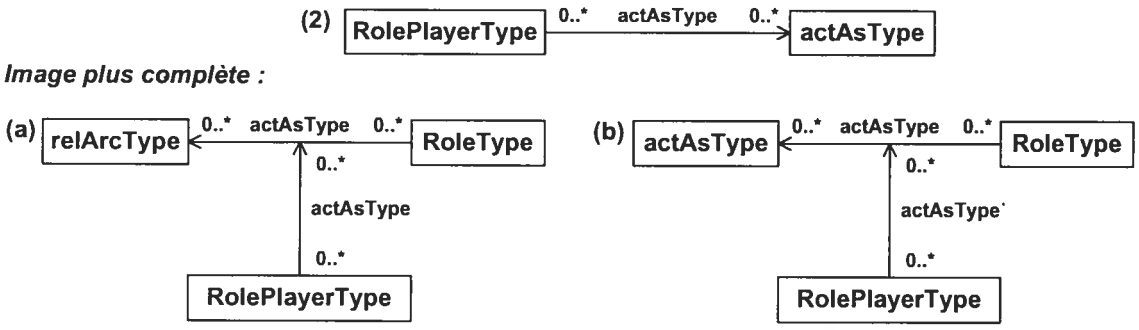


Figure 116 : Une autre structure initiale de *actAsType*

Les lecteurs peuvent se référer à la section III.3.15 (page iii-13) de l'Annexe III pour plus de détails sur la sémantique de chacune de ces structures de `actAsType`. Il est à souligner que dans une structure d'un type de relations, s'il existe un type de rôles dont les types de joueurs (`RolePlayerType`) ne sont pas indiqués, ces types de joueurs seront par défaut attachés à ce type de rôles par des arcs de type `playedByType`.

actAs

Pour une structure d'un type de relations, le méta-arc `actAs` permet d'indiquer les joueurs de rôles (`RolePlayer`) ou les types de joueurs de rôles (`RolePlayerType`) qui, sous un type de rôles, sont impliqués dans toutes les relations instanciées du type de relations selon la structure en question. Voir l'exemple ci-après.

Joueurs de rôles fixes pour types de rôles dans un type de relations (exemple 10)

Examinons le scénario suivant concernant le type de relations `gérer` décrit dans l'exemple 9 (page 123). Dans les relations de type `gérer`, l'élément `Organisation-EmpJean` désigne l'ensemble de toutes les organisations gérées par le rôle `EmpJean` seul dans le sens que chacune de ces organisations participe à une ou des relations de type `gérer` dans lesquelles les rôles «gestionnaire» ne sont occupés que par `EmpJean`, un rôle «employé permanent»; `Gestionnaire-EmpJean` désigne l'ensemble des rôles «gestionnaire» (`Gestionnaire`) joués fixement par `EmpJean`. Comment peut-on modéliser le type de relation `gérer` entre les participants `Gestionnaire-EmpJean`, `EmpJean` et `Organisation-EmpJean` ?

La Figure 117 présente une modélisation de ce scénario. Étant donné qu'une organisation peut ne pas appartenir pendant toute son existence au type `Organisation-EmpJean`, `Organisation-EmpJean` est un sous-type dynamique de `Organisation` et est conforme

à ObjDynType. Supposons qu'un rôle «gestionnaire», une fois déclaré comme une instance du type Gestionnaire-EmpJean, reste comme une instance permanente de ce type. Gestionnaire-EmpJean est un sous-type statique de Gestionnaire et est conforme à RoleStaticType. Alors Gestionnaire-EmpJean (respectivement Organisation-EmpJean) hérite de son parent Gestionnaire (respectivement Organisation) le type de relations gérer mais avec des contraintes additionnelles. À ce type de relations, l'élément Organisation-EmpJean est relié par un arc de type objType (comme l'objet de l'action) alors que Gestionnaire-EmpJean est relié par un arc de type actType (comme l'acteur de l'action). À cet l'arc de type actType de Gestionnaire-EmpJean à gérer, le rôle EmpJean est relié par un arc de type actAs, ce qui permet de représenter que les rôles «gestionnaire» de type Gestionnaire-EmpJean dans une relation de type gérer sont occupés par EmpJean seul.

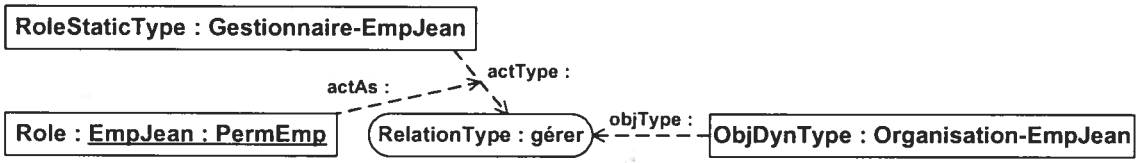


Figure 117 : Joueurs d'un type de rôles participant directement à un type de relations

Afin de représenter les arcs de type actAs tels que celui illustré dans la Figure 117 (exemple 10) concernant la spécification des joueurs de rôles fixes pour un type de rôles qui est un participant direct d'un type de relations, il faut avoir une structure de actAs comme celle-ci: le méta-arc actAs peut unir RolePlayer à relArcType (Figure 118-(1)). Étendue de cette structure, la structure suivante de actAs permet d'indiquer des types de joueurs de rôles fixes pour un type de rôles qui est un participant direct d'un type de relations: le méta-arc actAs peut unir RolePlayerType à relArcType (Figure 118-(2)). Veuillez prendre note de la Règle 26 (Annexe IV).

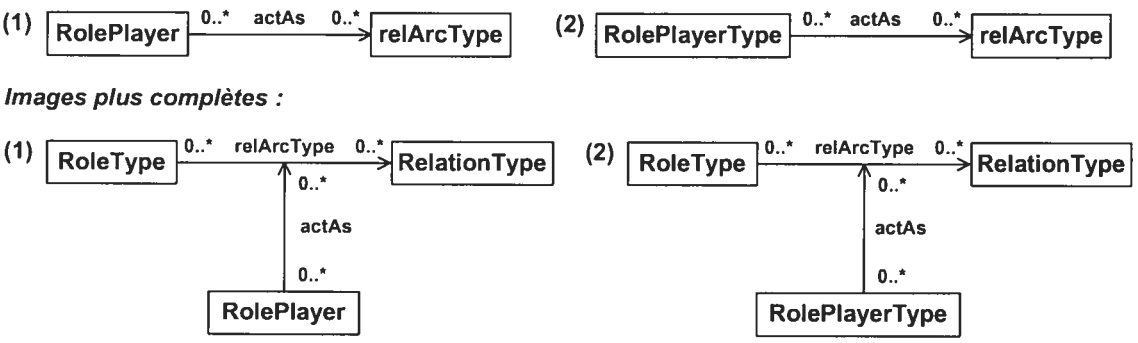


Figure 118 : Deux structures initiales de *actAs*

La description plus détaillée sur la sémantique de ces deux structures de `actAs` se trouve dans la section III.3.16 (page iii-16) de l'Annexe III.

Étudions l'exemple suivant.

Joueurs de rôles fixes pour les types de rôles impliqués dans un type de relations (exemple 11)

Voici un autre scénario concernant le type de relations `gérer` décrit dans l'exemple 9 (page 123). Dans les relations de type `gérer`, le type statique de rôles `Gestionnaire-Jean` désigne l'ensemble des rôles «gestionnaire» (`Gestionnaire`) joués par les rôles «employé permanent» dans le groupe `PermEmp-Jean`; le type dynamique de rôles `PermEmp-Jean` désigne l'ensemble des rôles «employé permanent» (`PermEmp`) joués par la seule personne `Jean`; le type dynamique d'objets `Organisation-Jean` désigne l'ensemble de toutes les organisations (`Organisation`) gérées par la seule personne Jean dans le sens que chacune de ces organisations participe à une ou des relations de type `gérer` dans lesquelles les rôles «employé permanent» impliqués sous les rôles «gestionnaire» ne sont occupés que par `Jean`. Comment peut-on modéliser le type de relation `gérer` entre les participants `Gestionnaire-Jean`, `PermEmp-Jean`, `Jean` et `Organisation-Jean` ?

La Figure 119 montre une représentation du scénario en question. Dans cette figure, au type de relations `gérer`, l'élément `Organisation-Jean` est relié par un arc de type `objType` (comme l'objet de l'action) alors que `Gestionnaire-Jean` est relié par un arc de type `actType` (comme l'acteur de l'action). À cet arc de type `actType` de `Gestionnaire-Jean` à `gérer`, le type de rôles `PermEmp-Jean` est relié par un arc de type `actAsType` (pour spécifier dans ce cas que les rôles «gestionnaire» de type `Gestionnaire-Jean` ne sont joués que par les rôles «employé permanent» de type `PermEmp-Jean`) À cet arc de type `actAsType`, `Jean` est relié par un arc de type `actAs` (pour spécifier dans ce cas que les rôles «employé permanent» de type `PermEmp-Jean` sont joués seulement par `Jean`).

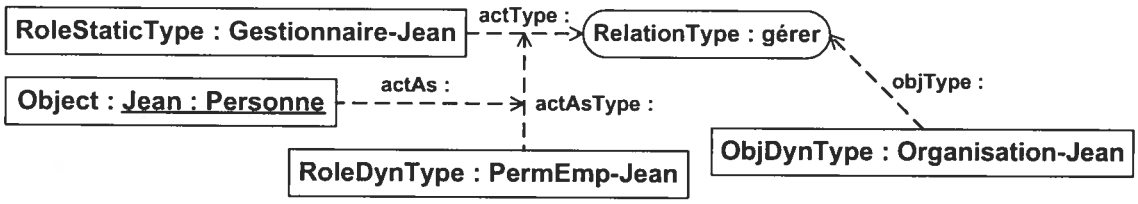


Figure 119 : Joueurs d'un type de rôles impliqué dans un type de relations

En vue de représenter les arcs de type `actAs` tels que celui illustré dans la Figure 119 (exemple 11), correspondant à la structure de `actAsType` illustrée dans la Figure 116 (page 124), les structures de `actAs` illustrées dans la Figure 120 permettent de spécifier

des joueurs de rôles (`RolePlayer`) particuliers ou des types de joueurs de rôles (`RolePlayerType`) qui sont fixés pour un type de rôles (`RoleType`) impliqué dans le cadre d'une structure d'un type de relations. C'est-à-dire que le méta-arc `actAs` peut unir `RolePlayer` ou `RolePlayerType` à `actAsType`. Veuillez prendre note de la Règle 27 (Annexe IV). La sémantique de `actAs` dans ces deux structures est décrite de façon détaillée à la section III.3.16 (page iii-16) de l'Annexe III.

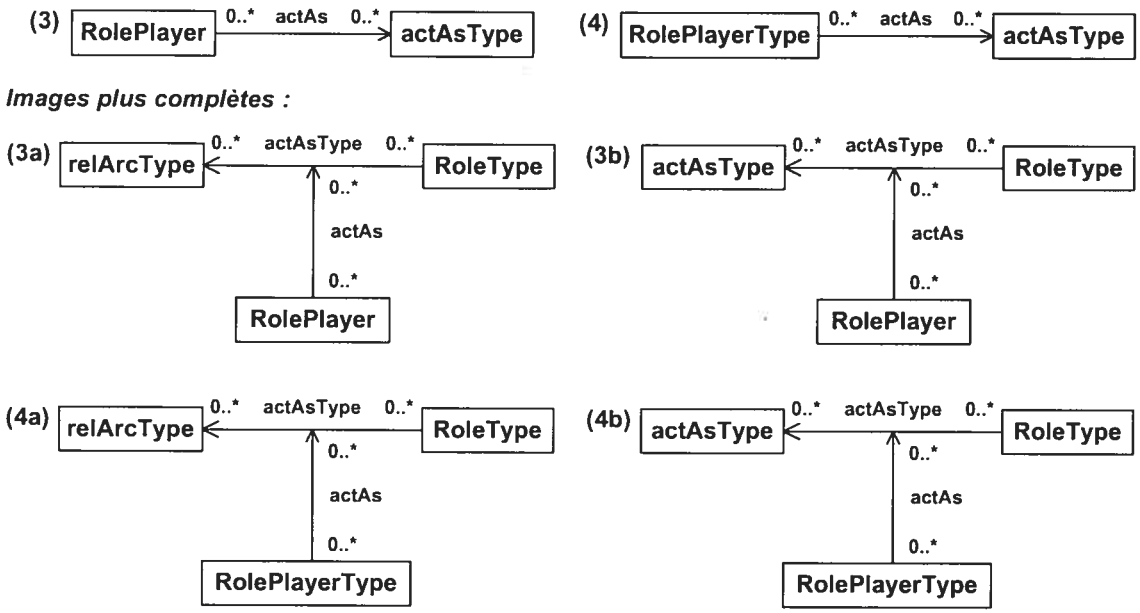


Figure 120 : Deux autres structures initiales de *actAs*

instOf

Les liens d'instanciation entre les relations et les types de relations sont représentés par des arcs de type `instOf` (Figure 121). Voir l'exemple de définition de relations (exemple 12) en bas.

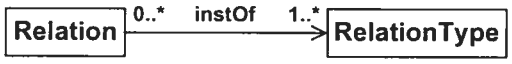


Figure 121 : Structure initiale de *instOf* pour les relations et les types de relations

relArc/act/obj pour la représentation de relations

Le méta-arc `relArc` permet de représenter des arcs qui indiquent des participants fixes à une relation (`Relation`) de niveau M1. Sous-types de `relArc`, les méta-arcs `act` et `obj` visent respectivement à expliciter des participants pris comme acteurs de l'action et des participants pris comme objets de l'action. Les structures de `relArc` sont illustrées par la Figure 122. Chacune de ces structures est étudiée ci-après.

Figure 122 : Structures initiales de *relArc* pour la représentation des relations

La structure de `relArc` illustrée dans la Figure 122-(1) permet à des `relArc` d'indiquer des instances (`TypeInstance`) impliquées dans une relation (`Relation`). L'exemple suivant montre une application de cette structure et illustre comment une relation est définie.

Définition d'une relation (exemple 12)

La Figure 123 exprime l'énoncé «la personne Jean travaille». Dans cette figure, l'élément `Jean` est relié par un arc de type `act` à l'élément `jean-travailler`. Les éléments `Jean` et `jean-travailler` représentent respectivement la personne Jean et une relation de type `travailler`. Ceci est indiqué par des arcs de type `instOf`. La structure `Structure-travailler` contextualise une structure du type de relations `travailler` (cf. l'exemple 8, page 122). Ce que `Context-jean` contextualise est une «instance» de ce que `Structure-travailler` contextualise, ce qui est indiqué par un arc de type `instModelOf` de `Context-jean` à `Structure-travailler`.

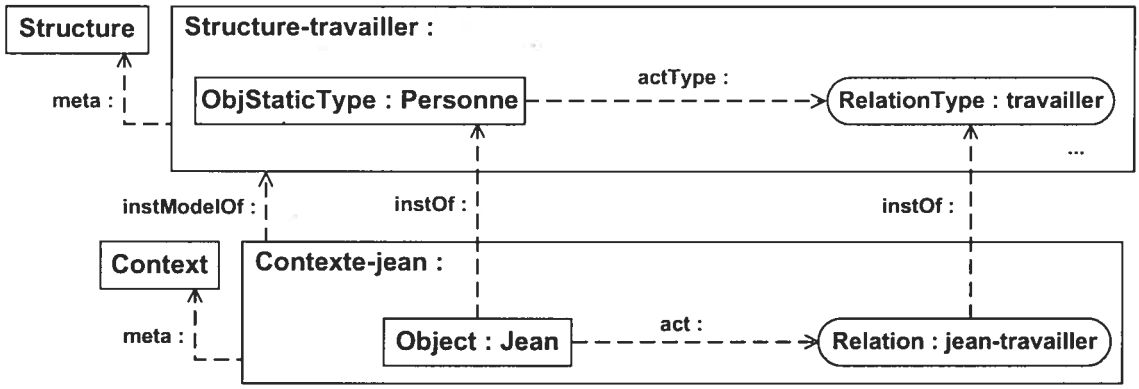


Figure 123 : Exemple de définition d'une relation

La structure de `relArc` illustrée dans la Figure 122-(2) permet à des `relArc` de relier des types (`Type`) à des relations (`Relation`). Le fait qu'un type soit rattaché par un arc de type `relArc` (respectivement `act` ou `obj`) à une relation est interprété comme une représentation simplifiée du fait que chaque instance de ce type soit rattachée par un arc de type `relArc` (respectivement `act` ou `obj`) à cette relation. Cette structure, d'une part, enrichit le pouvoir de l'expression de notre formalisme et d'autre part, nous offre un moyen de représenter plus simplement et efficacement certains cas de relations à

modéliser. L'exemple ci-après (exemple 15) en présente une illustration. Dans l'aspect de la modélisation de situations du monde réel, l'exemple 48 (page iii-17, Annexe III) nous donne plus de justifications sur l'avantage du fait qu'un type puisse s'engager dans une relation.

Type d'objets participant à une relation (exemple 13)

Continuons l'exemple sur le type de relations `travailler`. *Le fait que les personnes travaillent ensemble est interprété comme le fait que celles-ci participent ensemble à une relation de type* `travailler`. *Soit* `PersonneGroupe1` *un type dynamique d'objets représentant un groupe de personnes.*

Le fait que toutes les personnes dans le groupe `PersonneGroupe1` *travaillent ensemble soit représenté par le fait que chaque instance de type* `PersonneGroupe1` *est attachée par un arc de type* `act` *à une relation (soit* `personneGroupe1-travailler`$_i$*) de type* `travailler`. *Ceci est représenté simplement par le fait que* `PersonneGroupe1` *est relié à* `personneGroupe1-travailler`$_i$ *par un arc de type* `act`, *tel que l'illustre la Figure 124.*

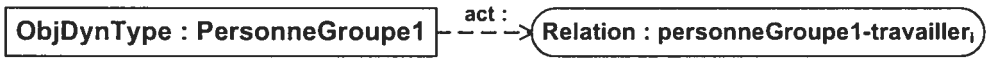


Figure 124 : Type participant à une relation

relArc/act/obj pour la représentation de types de relations

Afin de satisfaire à notre besoin «Types relationnels entre types et/ou instances» (page 39), la Figure 125 illustre d'autres structures de `relArc` permettant de représenter des arcs qui indiquent des participants fixes à un type de relations (`RelationType`) de M1. Sous-types de `relArc`, les méta-arcs `act` et `obj` permettent d'expliciter respectivement des participants pris pour des acteurs de l'action et des participants pris pour des objets de l'action.

Figure 125 : Structures initiales de *relArc* pour la représentation de types de relations

La Figure 125-(3) spécifie que des `relArc` peuvent lier des instances (`TypeInstance`) à un type de relations (`RelationType`). Le fait qu'une instance soit attachée par un arc de type `relArc` (respectivement `act` ou `obj`) à un type de relations signifie que cette première est rattachée par un arc de type `relArc` (respectivement `act` ou `obj`) à chaque relation instanciée de ce type de relations.

Instance participant à un type de relations (exemple 14)

Continuons l'exemple sur le type de relations `gérer` *décrit dans l'exemple 9 (page 123). Supposons que l'élément* `ISO` *désigne l'organisation ISO et que le type statique de rôles* `GestionnaireDeISO` *désigne l'ensemble des rôles «gestionnaire» (*`Gestionnaire`*) qui gèrent l'organisation* `ISO` *seul dans le sens que chacun de ces rôles «gestionnaires» participe à une ou des relations de type* `gérer` *où* `ISO` *est la seule organisation. Le type de relations* `gérer` *entre les participants* `GestionnaireDeISO` *et* `ISO` *peut être modélisé tel qu'illustré par la Figure 126. Dans cette figure, au type de relations* `gérer`*, le type* `GestionnaireDeISO` *est relié par un arc de type* `actType` *(comme l'acteur de l'action) alors que l'élément* `ISO` *est relié par un arc de type* `obj` *(comme l'objet de l'action).*

Figure 126 : Exemple de type relationnel entre une instance et un type

La structure de `relArc` illustrée dans la Figure 125-(4) (page 129) est inspirée de celle de la Figure 122-(2) (page 128). Elle permet à des `relArc` d'indiquer des types (`Type`) impliqués entièrement dans un type de relations (`RelationType`). Le fait qu'un type soit rattaché par un arc de type `relArc` (respectivement `act` ou `obj`) à un type de relations est interprété comme si chacune des instances de ce premier est rattachée par un arc de type `relArc` (respectivement `act` ou `obj`) à ce type de relations. Ceci signifie également que toutes les instances de ce premier sont rattachées par des arcs de type `relArc` (respectivement `act` ou `obj`) à chacune des relations instanciées de ce type de relations. Évidemment, cette structure nous offre un moyen de représenter plus simplement et efficacement certains cas de types de relations à modéliser. Ceci est démontré par l'exemple suivant.

Type d'objets participant entièrement à un type de relations (exemple 15)

Examinons le scénario suivant concernant le type de relations `travailler` *présenté dans l'exemple 8 (page 122). Supposons que* `toto`*,* `tata`*,* `tutu`*, et* `Jean` *représentent les personnes (*`Personne`*) et sont toutes les instances d'un type dynamique d'objets* `PersonneGroupe1` *représentant un groupe de personnes travaillant ensemble (cf. l'exemple 13, page 129).* `PersonneGroupe2` *désigne le type des personnes parmi lesquelles une ou plusieurs peuvent travailler ensemble avec toutes les personnes dans le groupe* `PersonneGroupe1`*. Comment peut-on modéliser* `PersonneGroupe2` *?*

Vu qu'une personne peut ne pas appartenir durant toute son existence au type

PersonneGroupe2, celui-ci est donc un sous-type dynamique de Personne et il est conforme à ObjDynType. PersonneGroupe2 hérite de son parent Personne le type de relations travailler. De façon plus restreinte, la structure du type de relations travailler auquel PersonneGroupe1 et PersonneGroupe2 s'engagent engendre les relations selon le schéma suivant. À chacune de ces relations, toutes les instances de PersonneGroupe1 (et peut-être avec une ou plusieurs instances de PersonneGroupe2) participent ensemble comme des acteurs de l'action. La structure peut être modélisée telle qu'illustrée par la Figure 127. À l'élément travailler, l'élément PersonneGroupe2 est relié par un arc de type actType et PersonneGroupe1 est relié par un arc de type act.

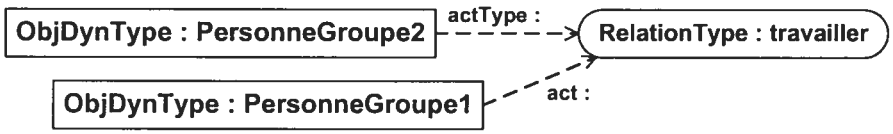


Figure 127 : Type impliqué entièrement dans un type de relations

5.3.4.4 Représentation de caractéristiques de types/instances

Attribute, AttrInst, chrcType, chrc, isKey, dataOfType

Ces éléments permettent de spécifier les caractéristiques des types et instances de types de niveau M1.

Les attributs (Attribute) symbolisent la connaissance représentant des caractéristiques des types (Type). Un attribut peut être caractérisé comme simple ou composé selon son modèle structurel, comme attribut clé ou ordinaire, et aussi comme multiple, unique, obligatoire ou optionnel, dépendamment des types qu'il caractérise (Définition 10).

Définition 10 : Caractéristiques des attributs

Un attribut est dit simple s'il n'a pas d'attributs, autrement dit, s'il est atomique. Par contre, un attribut est dit composé, s'il est constitué par d'autres attributs vus comme ses composants. Également, nous avons des instances d'un attribut simple et celles d'un composé.

Soient *T* un type à caractériser et *A* un attribut. *A* est dit un attribut clé pour *T* si la valeur de cet attribut *A* pour une instance de type *T* est différente de celle pour une autre instance de type *T*. Sinon *A* est dit un attribut ordinaire pour *T*.

Les contraintes de l'arité minimale/maximale de *A* pour *T* nous permet aussi de représenter d'autres caractéristiques de l'attribut *A*. Par exemple:

- *A* est dit multiple pour *T* si une instance de *T* peut avoir plusieurs valeurs en même temps pour son attribut *A*. L'arité maximale de *A* pour *T* doit être supérieure ou égale à la minimale et d'une valeur supérieure à 1.
- *A* est dit unique pour *T* si chaque instance de *T* a une et une seule valeur pour l'attribut *A* à chaque moment. Les arités minimale et maximale de *A* pour *T* doit être de valeur 1.
- *A* est dit obligatoire pour *T* si chaque instance de *T* a au moins une valeur pour l'attribut *A* à chaque moment. L'arité minimale de *A* pour *T* doit être inférieure ou égale à la maximale et d'une valeur supérieure à 0. Alors, si un attribut est unique pour un type, il est aussi obligatoire pour ce dernier.
- *A* est dit optionnel pour *T* si une instance de *T* peut n'avoir aucune valeur pour l'attribut *A*. L'arité minimale de *A* pour *T* doit être de valeur 0 et la maximale doit être d'une valeur supérieure à 0.

Le méta-arc `chrcType` permet d'attacher des attributs à des types (Figure 128). Comme un attribut composé est constitué par des attributs, ces derniers peuvent être regardés comme les attributs du premier. Ceci explique pourquoi nous spécifions que `Attribute` est aussi un sous-type de `Type` afin que `Attribute` puisse être relié à lui-même par `chrcType`. Le méta-arc `isKey` permet de désigner les attributs clés ou ordinaires pour un type. Sa structure (Figure 129) permet de couvrir les situations où un attribut est un attribut clé pour un type mais un attribut ordinaire pour un autre type. Si un arc de type `chrcType` qui attache un attribut *A* à un type *T* est relié par un arc de type `isKey` à `True` (l'instance prédéfinie de `Boolean` et représentant la valeur logique vrai (cf. la section III.2.7, page iii-3, Annexe III)), alors *A* est un attribut clé de *T*. Les autres caractéristiques d'un attribut (multiple, unique, obligatoire ou optionnel) sont spécifiées sous la forme de contraintes de l'arité (cf. la Définition 10). La section 5.3.4.6 (page 135) donnera plus de détails sur la représentation de ce type de contraintes.

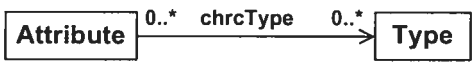


Figure 128 : Structure initiale de *chrcType*

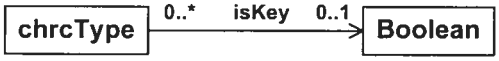


Figure 129: Structure initiale de *isKey*

Un attribut peut aussi être attaché à un type de données primitives, par exemple à `String` pour des chaînes de caractères, à `Integer` pour des entiers, etc. Ceci est indiqué par un arc de type `dataOfType` (cf. Figure 130).

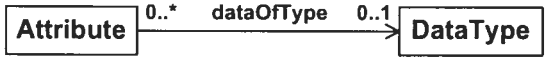


Figure 130 : Structure initiale de *dataOfType*

Les `AttrInst` (Figure 131) représentent les instances d'attributs (ou bien les valeurs d'attributs). `chrc` permet d'attacher des instances d'attributs à des instances à caractériser (Figure 132-(1)). Le fait qu'une instance d'un attribut caractérise (`chrc`) un type (Figure 132-(2)) spécifie que cette première est partagée par toutes les instances de ce type comme la valeur de l'attribut en question.

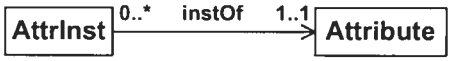


Figure 131 : Structure initiale de *instOf* pour les instances d'attributs et les attributs

Figure 132 : Structures initiales de *chrc*

L'exemple suivant (exemple 16) illustre comment spécifier des types et des instances avec leurs attributs. Il illustre également le comportement de `valueOf/depend` dans la représentation des interprétations d'éléments. L'exemple 52 (page iii-24) et l'exemple 53 (page iii-28) dans l'Annexe III expliquent plus en détail comment notre métamodèle permet la spécification d'attributs pour les types/instances de niveau M1.

Types et instances avec leurs attributs (exemple 16)

Soit la situation suivante. Un passeport permet d'identifier une personne comme citoyen d'un pays. Un passeport (`Passeport`) a un et un seul numéro d'identification (`IdNuméro`) et doit comporter le nom de son titulaire (`NomTitul`). Tel qu'illustré par la Figure 133, le type `Passeport` est caractérisé par deux attributs, `IdNuméro`, `NomTitul` (ce qui est indiqué par des arcs de type `chrcType`). `IdNuméro` est l'attribut clé de `Passeport` (ce qui est indiqué par un arc de type `isKey`), ce qui signifie que deux `Passeport` différents ont deux `IdNuméro` différents.

La Figure 133 illustre aussi comment une instance de `Passeport` est définie avec ses attributs. L'élément `Pca2005` représente un passeport (ce qui est indiqué par un arc de type `instOf` entre `Pca2005Jean` et `Passeport`) dont le numéro d'identification est

«CA123456» et le nom du titulaire est «Levesque Jean». Il est caractérisé par deux instances d'attributs (`AttrInst`) qui sont représentés par les éléments `Pca2005-IdNuméro` et `Jean-NomTitul` (ce qui est indiqué par des arcs de type `chrc`) et qui prennent respectivement `IdNuméro` et `NomTitul` pour leurs types (ce qui est indiqué par des arcs de type `instOf`). Les éléments `Pca2005-IdNuméro` et `Jean-NomTitul` sont interprétés en langage universel respectivement comme «CA123456» et «Levesque Jean». La Figure 134 montre la représentation simplifiée de `Pca2005`.

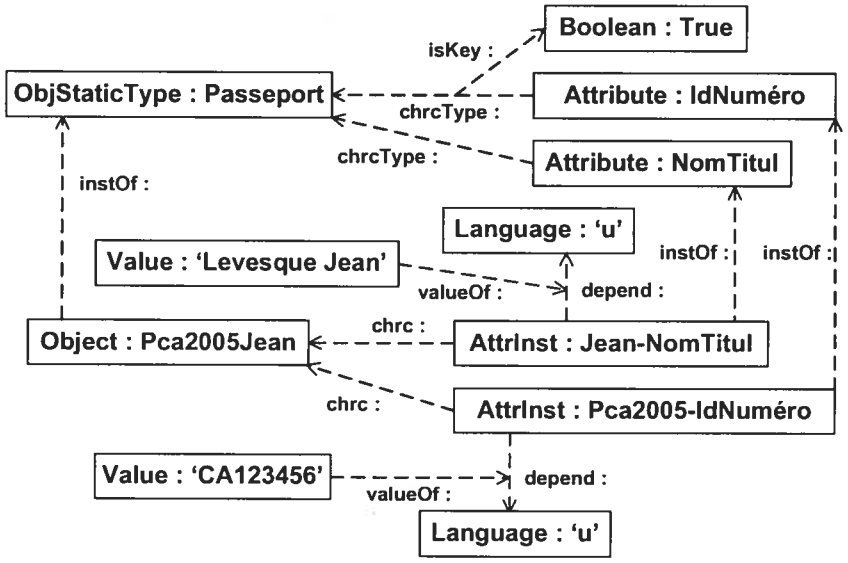


Figure 133 : Objet et ses valeurs attributs

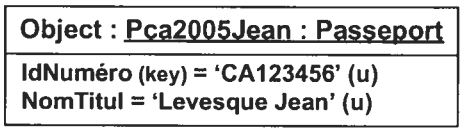


Figure 134 : Représentation simplifiée de l'objet *Pca2005Jean*

5.3.4.5 Instanciation de types de M1

Comme nous avons présenté, le méta-arc `instOf` implémente la relation d'instanciation au niveau M1. Il permet d'indiquer respectivement pour un `AttrInst`, `Object`, `Role`, `Relation` son ou ses types `Attribute`, `ObjType`, `RoleType`, `RelationType`. Il est important de noter que si un type de M1 est instancié, tous ses attributs doivent aussi être instanciés et que ses instances doivent se conformer à sa structure de définition. Il est à noter qu'un type de M1 peut être spécifié comme abstrait ou non, ce qui est indiqué par des arcs de type `isAbst` (cf. la Figure 135, et la section III.3.26, page iii-29, Annexe III).

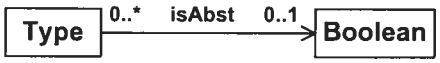


Figure 135 : Structure initiale de *isAbst*

5.3.4.6 Représentation de contraintes structurelles

LabelledInteger, interval

Similaires aux types `MLabelledInteger` et `Minterval` de M3, le méta-nœud `LabelledInteger` permet de représenter les entiers étiquetés et le méta-arc `interval` permet de former des intervalles fermés dont les bornes inférieures et supérieures sont des entiers étiquetés (Figure 136).

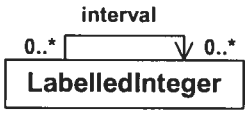


Figure 136 : Structure initiale de *interval*

arity, totalCard, localCard, iterate

Ces méta-arcs permettent de représenter au niveau M1 les types de contraintes structurelles suivants:

BesoinM2 10 : Contraintes de l'arité min/max

BesoinM2 11 : Contraintes de cardinalité totale min/max pour un type relationnel

BesoinM2 12 : Contraintes de cardinalité locale min/max pour un type relationnel

BesoinM2 13 : Contraintes de l'itération min/max pour un type relationnel

Nous présentons maintenant ces méta-arcs en mettant l'accent sur le principe permettant de spécifier les contraintes structurelles. Ce principe est le même que celui permettant de spécifier les contraintes de cardinalité sur méta-arcs que nous avons vu dans la section 5.2.2.3 (page 98). Les informations complémentaires à ce sujet se trouvent dans la section 4 (page iii-7) de l'Annexe III.

arity

Le type `arity` vise à représenter toutes les sortes de contraintes de l'arité, énumérées dans le BesoinM2 10 - «Contraintes de l'arité min/max» (page 36). Les structures de `arity` sont montrées dans la Figure 137. Celles illustrées respectivement aux Figure 137-(1),(2),(3) permettent respectivement de supporter les besoins suivants: (1) BesoinM2 10-(i) - «Contraintes de l'arité min/max sur un attribut pour un type»; (2) BesoinM2 10-(ii) - «Contraintes de l'arité min/max sur la composition d'un type

d'objets»; et (3) BesoinM2 10-(iii) - «Contraintes de l'arité min/max sur l'occupation d'un type de rôles». Les structures de `arity` illustrées aux Figure 137-(4),(5),(6) permettent de remplir le BesoinM2 10-(iv) - «Contraintes de l'arité min/max d'un type relationnel». Et celle à la Figure 137-(7) permet de combler le BesoinM2 10-(v) - «Contraintes de l'arité min/max sur l'implication d'un type d'objets/rôles sous un type de rôles dans un type relationnel». Pour de plus amples explications, voir les spécifications des types `relArcType`, `actType`, `objType`, `actAsType` (cf. la section 5.3.4.3, page 121), `playedByType` (cf. la section 5.3.4.2, page 120), `composeType` (cf. la section 5.3.4.1, page 118) et `chrcType` (cf. la section 5.3.4.4, page 131).

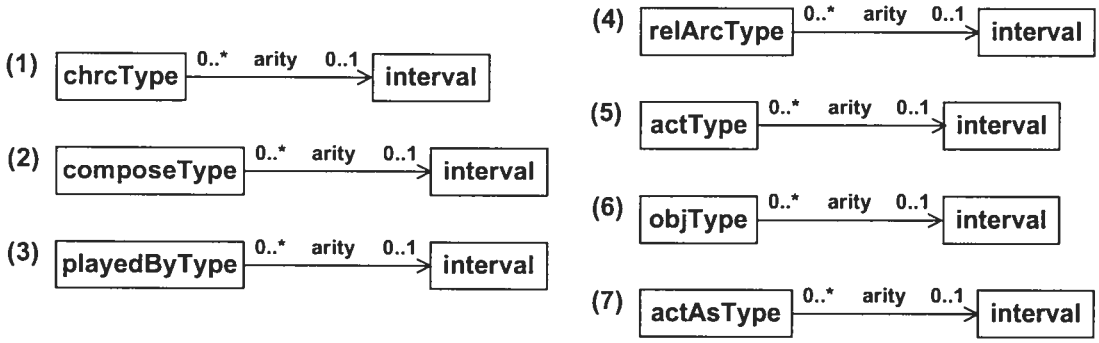


Figure 137 : Structures de *arity*

Étant donné que `chrcType`, `composeType`, `actAsType`, `playedByType`, et `relArcType` sont les sous-types directs de `forType`, et que `actType` et `objType` sont les sous-types directs de `relArcType`, les structures de `arity` illustrées dans la Figure 137 sont obtenues par le sous-typage (cf. la Règle 1, Annexe IV) de la structure de `arity` suivante: le méta-arc `arity` est défini pour relier `forType` à `interval` avec les cardinalités source 0..* et les cardinalités destination 0..1 (Figure 138).

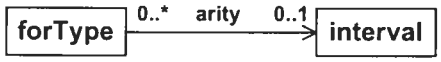


Figure 138 : Structure initiale de *arity*

L'exemple suivant montre comment spécifier des contraintes de l'arité.

Spécification des contraintes de l'arité min/max sur la composition d'un type d'objets

La Figure 139 illustre comment seront définies les contraintes de l'arité sur l'arc de type `composeType` de `Vitre` (représentant le type de vitres) à `Fenêtre` (représentant le type d'objets fenêtres). L'arité maximale sur cet arc est de valeur 5 tandis que l'arité minimale prend la valeur 1. Ceci spécifie qu'il y a d'une à 5 vitres au maximum qui peuvent être composantes

d'une fenêtre.

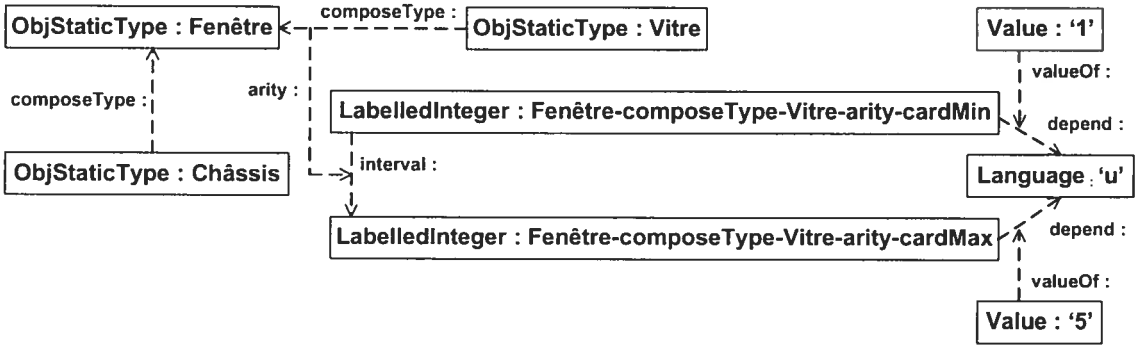


Figure 139 : Exemple de définition des contraintes de l'arité

La Figure 139 peut être représentée plus simplement comme dans la Figure 140. Les arités minimale/maximale sur l'arc de type `composeType` *de* `Châssis` *(représentant le type de châssis) à* `Fenêtre` *ne sont pas spécifiées et elles prennent donc les valeurs [1..1]. Ceci indique qu'il y a toujours un et un seul châssis parmi les composants d'une fenêtre. La Notation 5 présente notre notation liée aux contraintes de l'arité.*

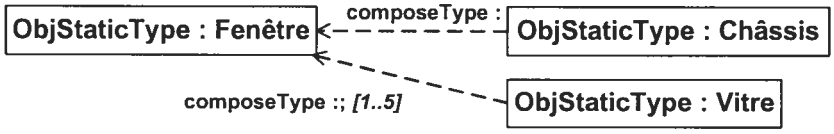


Figure 140 : Version simplifiée de la Figure 139

Notation 5 *: Notations pour les contraintes de l'arité minimale/maximale*

Dans les illustrations, si l'arité minimale (ou maximale) sur un arc de type `chrcType`, ou `composeType`, `playedByType`, `relArcType`, `actType`, `objType`, `actAsType` n'est pas explicitement spécifiée ou affichée, la valeur pour cette arité est 1 (ou *). Les arités minimale et maximale s'écrivent entre deux crochets ([..]), en italique et sur l'arc auquel elles sont attachées. Ces dernières sont séparées par les points-virgules avec les autres contraintes imposées sur l'arc en question s'il en existe.

totalCard

Le type `totalCard` permet de spécifier les contraintes de cardinalités totales abordées dans le BesoinM2 11 - «Contraintes de cardinalité totale min/max pour un type relationnel» (page 36). `totalCard` est défini pour relier chacun des types `relArcType` et `actAsType` à `interval` avec les cardinalités source 0..* et les cardinalités destination 0..1 (Figure 141). En remplaçant `relArcType` dans la Figure 141-(1) par les

sous-types `actType` et `objType`, nous obtenons deux autres structures de `totalCard`. Voir aussi les spécifications des types `relArcType`, `actType`, `objType` et `actAsType` (cf. la section 5.3.4.3, page 121).

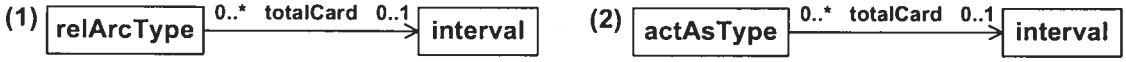


Figure 141 : Structures initiales de *totalCard*

Les contraintes de cardinalité totale sont spécifiées d'une façon similaire à celles de l'arité. Voir l'exemple 42 (page iii-9, Annexe III) pour plus de détail.

***Notation 6** : Notations pour les contraintes de cardinalité totale minimale/ maximale*

Dans les illustrations, si la cardinalité totale minimale (ou maximale) sur un arc de type `relArcType`, ou `actType`, `objType`, `actAsType` n'est pas explicitement spécifiée ou affichée, cette cardinalité prendra la valeur 0 (ou *). Les cardinalités totales minimale et maximale s'écrivent entre deux crochets ([..]), en souligné et sur l'arc auquel elles sont attachées. Ces dernières sont séparées par les points-virgules avec les autres contraintes imposées sur l'arc en question s'il en existe.

localCard

Le type `localCard` sert à spécifier les contraintes de cardinalités locales présentées dans le BesoinM2 12 - «Contraintes de cardinalité locale min/max pour un type relationnel» (page 37). `localCard` est défini pour relier chacun des types `relArcType` et `actAsType` à `interval` avec les cardinalités source 0..* et les cardinalités destination 0..1 (Figure 142). En remplaçant `relArcType` par les sous-types `actType` et `objType`, nous obtenons deux autres structures de `localCard`. Voir plus loin les spécifications des types `relArcType`, `actType`, `objType` et `actAsType` (cf. la section 5.3.4.3, page 121).

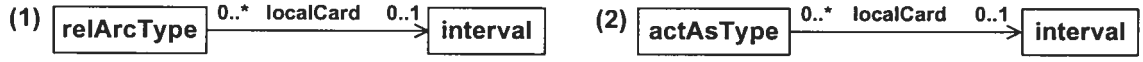


Figure 142 : Structures initiales de *localCard*

Les contraintes de cardinalité locale sont spécifiées d'une façon similaire à celles de l'arité. Voir l'exemple 43 (page iii-9, Annexe III) pour l'illustration.

***Notation 7** : Notations pour les contraintes de cardinalité locale minimale/ maximale*

Dans les illustrations, si la cardinalité locale minimale (ou maximale) sur un arc de

type `relArcType`, `actType`, `objType`, `actAsType` n'est pas explicitement spécifiée ou affichée, celle-ci prendra la valeur 0 (ou *). Les cardinalités locales minimale et maximale s'écrivent entre deux crochets ([..]) et sur l'arc auquel elles sont attachées. Ces dernières sont séparées par les points-virgules avec les autres contraintes imposées sur l'arc en question s'il en existe.

iterate

Ce méta-arc (Figure 143) permet de spécifier les contraintes de l'itération pour un type de relations (cf. BesoinM2 13 - «Contraintes de l'itération min/max pour un type relationnel», page 37). Ces contraintes capturent le nombre d'instances de ce type de relations dont les participants dans une instance sont également ceux dans une autre.

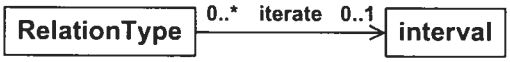


Figure 143 : Structure initiale de iterate

De même, les contraintes de l'itération pour un type de relations sont spécifiées d'une façon similaire à celles de l'arité. Voir l'exemple 44 (page iii-10, Annexe III) pour la démonstration. Les notations pour les contraintes de l'itération pour un type de relations sont présentées dans la Notation 8.

Notation 8 *: Notations pour les contraintes de l'tération minimale/maximale pour un type de relations*

Dans les illustrations, si le nombre de l'itération minimal (ou maximal) pour un type de relations n'est pas explicitement spécifié ou affiché, la valeur pour ce nombre est 1 (ou *). Les nombres de l'itération minimal et maximal pour un type de relations s'écrivent entre deux crochets ([min..max]) et suivant le nom du type de relations en question.

Ci-après est un exemple d'une situation avec différents types de contraintes structurelles. Voir aussi l'exemple 20 (page 165) et plusieurs autres exemples présentés dans l'Annexe III tels que l'exemple 45 (page iii-11), l'exemple 46 (page iii-11), l'exemple 47 (page iii-14), l'exemple 49 (page iii-20), l'exemple 50 (page iii-22) et l'exemple 51 (page iii-23).

Spécification du type de relations travailler (exemple 17)

La Figure 144 montre une structure du type de relations `travailler` *avec des contraintes relatives à ce type.*

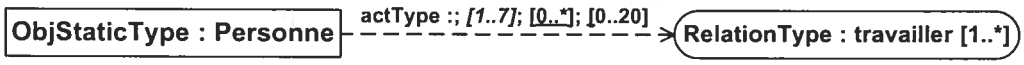


Figure 144 : Une structure du type de relations *travailler*

Dans cette structure, `travailler` *implique un seul type d'objets* `Personne` *en tant qu'acteur de l'action* `travailler`*. Les contraintes (implicites et explicites) de l'arité et de cardinalités (totales et locales) sur l'arc de type* `actType` *de* `Personne` *à* `travailler` *sont interprétées comme suit:*

(1) Il y a d'une à 7 personnes au maximum pouvant travailler ensemble, c'est-à-dire celles-ci peuvent s'associer ensemble à une relation de type `travailler`*. Ceci est indiqué par les contraintes notées comme [1..7] sur l'arc de type* `actType` *de* `Personne` *à* `travailler`*.*

(2) Une personne (`Personne`*) peut ne participer à aucune, ou participer à une ou plusieurs relations de type* `travailler`*, ce qui est indiqué par les contraintes notées comme* [0..*] *sur l'arc de type* `actType` *de* `Personne` *à* `travailler`*.*

(3) On peut trouver au maximum vingt personnes pouvant chacune s'associer à une relation de type `travailler` *avec un même groupe de personnes, ce qui est indiqué par les contraintes notées comme* [0..20] *sur l'arc de type* `actType` *de* `Personne` *à* `travailler`*.*

Les contraintes de l'itération pour `travailler` *spécifient:*

(4) Une relation de type `travailler` *peut se répéter plusieurs fois. Ainsi, les mêmes personnes peuvent participer plusieurs fois ensemble à une relation de type* `travailler`*. Ceci est indiqué par les contraintes notées comme [1..*] suivant le nom du type* `travailler`*.*

5.3.4.7 Hiérarchisation entre les types : le sous-typage et l'héritage

`subType` implémente la relation de sous-typage entre des types de M1 autres que les modèles alors que `inherit` implémente l'héritage. Les `ObjType`, les `RoleType`, les `RelationType`, et les `Attribute` peuvent respectivement être organisés en hiérarchie de spécialisation (ou d'héritage) selon les liens de type `subType` (ou `inherit`) qui existent entre eux. En tenant compte de la nature différente entre les types statiques et dynamiques (cf. la Définition 9, page 118), il est à noter qu'un `ObjStaticType` (respectivement `RoleStaticType`) ne peut être un sous-type d'un `ObjDynType` (respectivement `RoleDynType`). Tel que démontré à la section 2.1.3 (page 12), le type relationnel d'héritage (`inherit`) est plus général que celui de sous-typage (`subType`)

dans le sens que, le fait qu'un type `A` est relié à un type `B` par un lien de sous-typage signifie aussi que `A` est implicitement relié à `B` par un lien d'héritage. Les structures de `subType` sont donc déduites de celles de `inherit` (Figure 145) en remplaçant `inherit` par `subType`. Le sous-typage et l'héritage multiples sont permis.

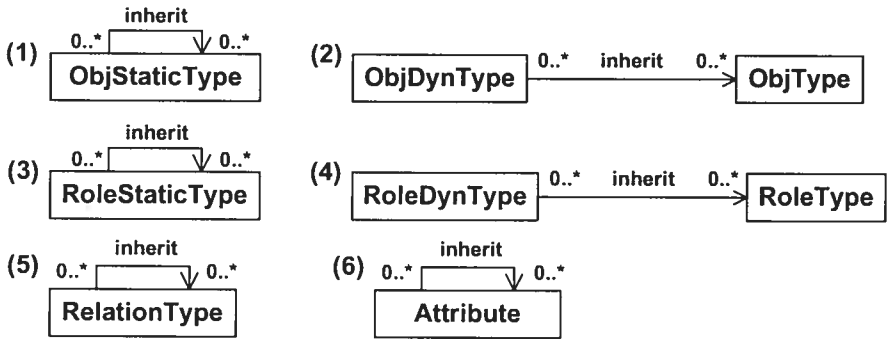


Figure 145 : Structures de *inherit*

En ce qui concerne la spécialisation entre les types de relations, chaque structure initiale d'un type de relations est plus spécifique que celles du ou des parents de ce type. La spécialisation entre les structures de types de relations est expliquée plus en détail à la section 7.2.2.1 (page 208) du Chapitre 7 (page 194). La section III.3.27 (page iii-29, Annexe III) présente différents exemples de sous-typage entre des types statiques/dynamiques d'objets, entre des types statiques/dynamiques d'objets, entre des types de relations ou entre des attributs.

Concernant l'effet de l'héritage, un type enfant peut hériter d'un type parent seulement les attributs et les types de relations héritables. Dans notre contexte, nous déterminons si un attribut (ou un type de relations) d'un type parent est héritable ou non (cf. la Définition 25, page iii-2, Annexe III) pour un type enfant, dépendamment des visibilités des éléments impliqués, et en suivant la Définition 25 (page iii-2, Annexe III). Voir dans la section 5.3.5 (page 141) notre solution concernant la spécification de la visibilité pour les éléments de M1.

5.3.5 Spécification de la visibilité pour les éléments

Les valeurs de visibilité (`Visibility`) sont inspirées de celles dans UML: publique (+), protégé (#) et caché (-). Nous considérons la visibilité d'un élément dans un modèle (cf. la Définition 23, page iii-2, Annexe III) et celle d'un attribut pour un type caractérisé (cf. la Définition 24, page iii-2, Annexe III).

Le méta-arc `visib` a pour but de spécifier la visibilité d'un élément (Figure 146).

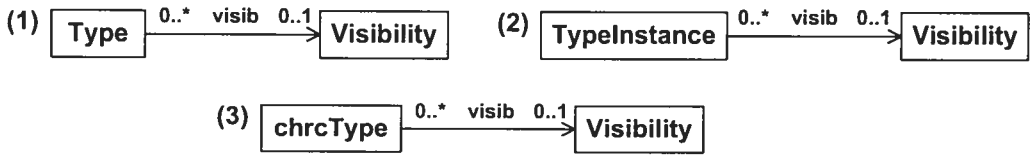


Figure 146 : Structures initiales de *visib*

Comme un élément (type ou instance) est défini dans un et un seul modèle, la structure de `visib` illustrée dans la Figure 146-(1) (respectivement Figure 146-(2)) permet à `visib` d'attacher une visibilité à un type (respectivement une instance), soit x, dans le modèle dans lequel x est défini. Quant à la structure de `visib` illustrée dans la Figure 146-(3), elle vise à spécifier la visibilité d'un attribut pour un type de M1. Cette structure permet de combler les cas où un attribut peut avoir différentes visibilités dépendamment des types caractérisés par l'attribut. Voici un exemple.

Visibilité d'un attribut pour un type (exemple 18)

La Figure 147 spécifie que le numéro téléphonique (`NuméroTel`) d'une personne (`Personne`) est privé alors que celui d'un rôle employé (`Employé`) est publique.

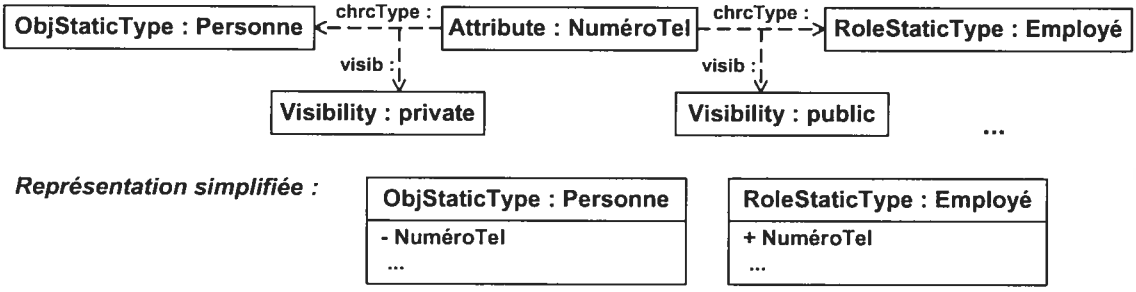


Figure 147 : Attributs avec leurs visibilités

5.3.6 Représentation de modèles et de leurs liens

Model, sem

`Model` permet la contextualisation des connaissances au niveau M1. Un modèle doit se conformer à un et un seul métamodèle (cf. la Règle 18, Annexe IV). Le méta-arc `sem` (Figure 148) implémente cette relation de conformité entre les modèles de niveau M1 (`Model`) et leurs métamodèles (`MMetaModel`) au niveau M2.

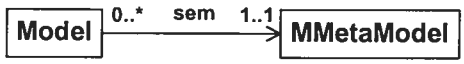


Figure 148 : Structure initiale de *sem*

Parmi les modèles (`Model`) au niveau M1, nous distinguons: les structures (`Structure`) pour les types de relations, les contextes (`Context`) pour les relations, les cycles d'états (`ObjCycle`) pour les types d'objets, les cycles d'états (`RoleCycle`) pour les types de rôles et les modèles de conditions (`IfThenModel`).

Structure, Context, defAs

Le méta-nœud `Structure` permet de contextualiser les structures des types de relations au niveau M1. Une structure d'un type de relations définit comment ce type associe ses participants. Chaque structure dite initiale d'un type de relations (`RelationType`) lui sera reliée par un arc de type `defAs` (Figure 149).

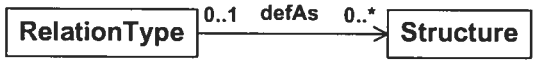


Figure 149 : Structure initiale de *defAs*

Le méta-nœud `Context` vise à contextualiser les modèles des relations au niveau M1. Pour une relation instanciée d'un type de relations selon une structure de ce type de relations, le modèle de cette relation indique comment cette relation associe ses participants. `Context` permet donc de distinguer explicitement ce type de modèles avec d'autres types de modèles au M1 telles que `Structure`, `IfThenModel`, `ObjCycle`, `RoleCycle`.

De plus, `Structure` est un sous-type de `RolePlayerType`, et `Context` est celui de `RolePlayer`. Ainsi, une `Structure` (`Context`) compte pour un `RolePlayerType` (`RolePlayer`). Ceci autorise des `Context` et des `Structure` à s'impliquer dans des `Relation` ou dans des `RelationType`.

ObjCycle, RoleCycle, cycleOf, startOf, prevOf et endOf

Les méta-nœuds `ObjCycle` et `RoleCycle` permettent respectivement de contextualiser les cycles d'états pour les types d'objets et ceux pour les types de rôles. Un cycle d'états pour un type d'objet (ou pour un type de rôles) spécifie les passages entre les états sur la base de certains critères qu'un objet (ou un rôle) de ce type peut passer durant son existence. Ci-après, un exemple de ce cas.

Cycle de vie de personnes (exemple 19)

La Figure 150 illustre le cycle de vie `Cycle-Vie` de personnes (ce qui est indiqué par l'arc de type `cycleOf` du modèle `Cycle-Vie` au type `Personne`). En basant sur l'âge d'une

personne, une personne peut commencer par l'état enfant (ce qui est indiqué par l'arc de type `startOf` *du type* `Enfant` *à* `Cycle-Vie`*). Puis, elle peut passer de l'état enfant à l'état adolescent (ce qui est indiqué par l'arc de type* `prevOf` *du type* `Enfant` *au type* `Adolescent`*). Ensuite, elle peut passer de l'état adolescent à l'état adulte (ce qui est indiqué par l'arc de type* `prevOf` *du type* `Adolescent` *au type* `Adulte`*). Finalement, elle peut passer de l'état adulte à l'état «personne âgée» (ce qui est indiqué par l'arc de type* `prevOf` *du type* `Adulte` *au type* `PersonneÂgée`*).*

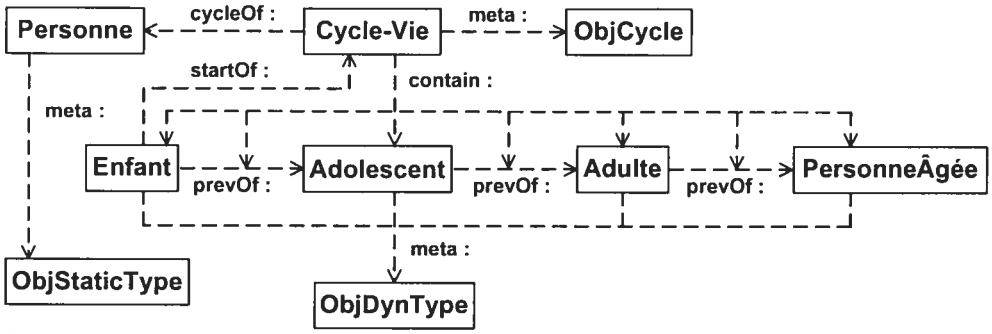


Figure 150 : Cycle de vie d'une personne

Dans un cycle d'états pour un type d'objets/rôles (soit `T`), les types représentant les états de `T` dans ce cycle sont considérés comme des sous-types dynamiques, disjoints et complets de `T`. Autrement dit, chaque instance de `T` peut, selon de son état, être classifiée de façon dynamique sous ces sous-types mais elle ne peut pas être classifiée à un instant donné sous plus d'un de ces sous-types.

La Figure 151 illustre les structures des méta-arcs `cycleOf`, `startOf`, `prevOf` et `endOf` qui permettent de spécifier les cycles d'états pour les types d'objets/rôles.

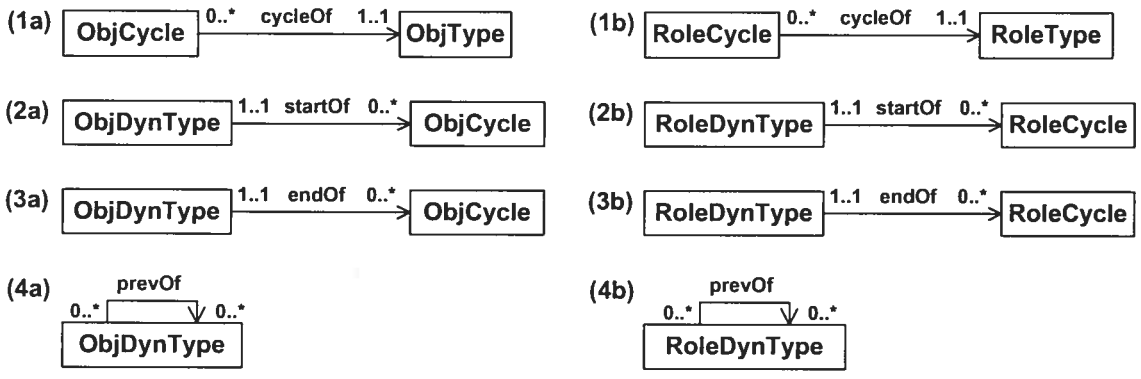


Figure 151 : Structures initiales de *cycleOf*, *startOf*, *prevOf* et *endOf*

Dans l'Annexe III, la section 9 (page iii-34, Annexe III) illustre plus précisément comment spécifier les cycles d'états et leur intégration avec des exemples concrets.

IfThenModel

Le méta-nœud `IfThenModel` sert à contextualiser des préconditions et des postconditions dans la représentation des règles définies au niveau M1. La représentation de règles de niveau M1 sera présentée plus en détail à la section 5.3.7 (page 147).

Nous présentons ci-dessous, les autres méta-arcs qui visent aussi à gérer les modèles au niveau M1 ainsi que leur contenu.

defIn, contain, extend, import

Les structures de ces méta-arcs sont illustrées à la Figure 152. Similaires respectivement aux éléments `MdefIn`, `Mcontain` de M3, les éléments `defIn`, `contain` visent à représenter les liens de *contenant* entre un modèle de M1 et ses éléments. Similaire à `Mextend` de M3, l'élément `extend` de M2 implémente l'extension entre les modèles de M1. Quant à lui, `import` implémente l'importation d'éléments publiques entre les modèles de M1. Son comportement est décrit par la Règle 32 (Annexe IV).

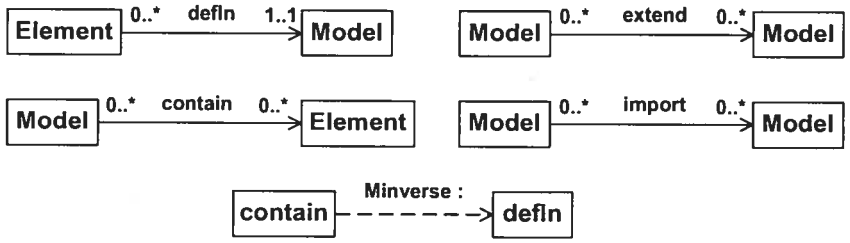


Figure 152 : Structures initiales de *defIn*, *contain*, *extend*, *import*

Les informations plus détaillées sur ces méta-arcs se trouvent dans les sections de III.3.35 (page iii-40) à III.3.38 (page iii-41) de l'Annexe III.

modelOp, join, diff, intersection, result, infer, restrict, instModelOf, transform, semAs

Sous-types de `modelOpArc`, ces méta-arcs (Figure 153) ont pour but de représenter certaines opérations entre modèles de M1. Les méta-arcs `modelOp`, `join`, `diff`, `intersection`, `result`, `infer` et `restrict` de M2 implémentent respectivement les mêmes fonctions que les méta-arcs `MmodelOp`, `Mjoin`, `Mdiff`, `Mintersection`, `Mresult`, `Minfer` et `Mrestrict` de M3 mais applicables au niveau M1. Les méta-arcs `instModelOf`, `transform` et `semAs`, quant à eux, concernent des fonctions plutôt applicables aux modèles de niveau M1 qu'aux modèles au niveau méta. `instModelOf` vise à représenter l'opération d'instanciation de modèles abordée dans la section 2.3.2

(page 21). Cette opération est expliquée en détail à la section 7.2.2.2 (page 211). Entre modèles de M1, les arcs de type `transform` représente les liens de *transformation* alors que ceux de type `semAs` représente les liens d'*équivalence*.

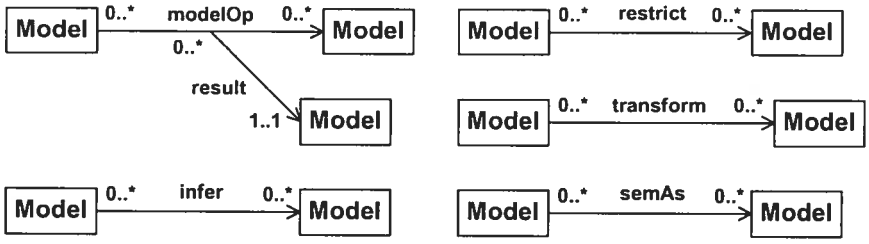


Figure 153 : Méta-arcs *modelOp*, *result*, *infer*, *restrict*, *instModelOf*, *transform*, *semAs*

Les informations plus détaillées sur ces méta-arcs se trouvent dans la section III.3.32 (page iii-39) et les sections de III.3.39 (page iii-41) à III.3.44 (page iii-42) de l'Annexe III.

viewOf

Le méta-arc `viewOf` (Figure 154) vise à indiquer des modèles considérés comme des vues (*views*) différentes, correspondants à différents niveaux d'abstraction ou à différents points de vue pour un élément (qui peut désigner un modèle (cf. Figure 154-(1)), un type d'objets (cf. la Figure 154-(2)), ou un objet (cf. la Figure 154-(3)). La sémantique et le comportement de `viewOf` sont spécifiés plus en détail à la section III.3.45 (page iii-42, Annexe III).

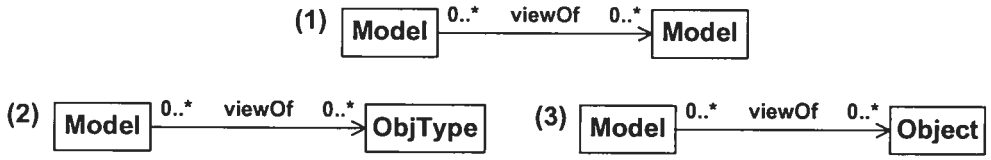


Figure 154 : Structures initiales de *viewOf*

inheritModel

`inheritModel` (Figure 155) implémente la relation d'héritage entre modèles de M1. Par cette relation d'héritage, un modèle enfant peut réutiliser les éléments non cachés d'un modèle parent, et en outre, il peut hériter de ce modèle parent les attributs et types de relations héritables (cf. la Règle 31, Annexe IV). Voir les Définition 23 (page iii-2), Définition 24 (page iii-2) et Définition 25 (page iii-2) pour l'effet de la visibilité d'éléments sur l'accessibilité d'éléments.

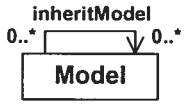


Figure 155 : Structure initiale de *inheritModel*

5.3.7 Représentation de règles

Rule, IfThenModel, Ref, EveryRef, if, then

Similaires respectivement aux éléments `MRule`, `MIfThenModel`, `Mif` et `Mthen` de M3, les éléments `Rule`, `IfThenModel`, `if`, et `then` de M2 (Figure 156) s'utilisent pour structurer les règles sémantiques qui contraignent les types/instance de niveau M1. Et similaires respectivement aux éléments `MRef` et `MEveryRef` de M3, les éléments `Ref` et `EveryRef` de M2 permettent le traitement des variables dans les règles de niveau M1. La Règle 19 (Annexe IV) sur l'attribution d'éléments à une variable, et la Règle 20 (Annexe IV) sur la combinaison des modèles de préconditions (respectivement de postconditions) dans une règle, s'appliquent toujours.

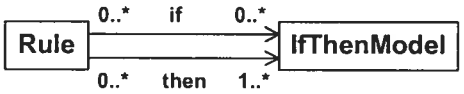


Figure 156 : Structures initiales de *if, then*

fulfil

Similairement au rôle du méta-arc `Mfulfil` de niveau M3, le méta-arc `fulfil` de niveau M2 s'utilise pour spécifier les contextes d'application des règles au M1. Nous considérons les structures de `fulfil` suivantes :

(1) `fulfil` peut indiquer pour un type de joueurs de rôles (`RolePlayerType`) les règles à remplir (Figure 157-(1)).

(2) `fulfil` peut indiquer des règles spécifiant les critères afin qu'une instance d'un type puisse passer d'un type d'états à un autre type d'états (Figure 157-(2)). La Règle 33 (Annexe IV) est à noter. L'exemple 35 (page 186) illustre son application.

(3) `fulfil` peut indiquer des règles spécifiant les critères appliqués à l'occupation d'un type de rôles pour un type de joueurs de rôles (Figure 157-(3)). Ceci est précisé par la Règle 34 (Annexe IV).

(4) `fulfil` peut indiquer des règles à appliquer lors de la transformation entre modèles (Figure 157-(4)). Ceci est décrit par la Règle 35 (Annexe IV).

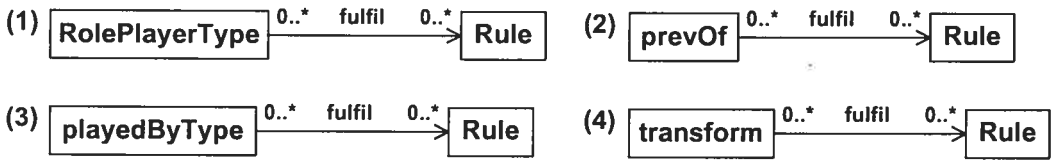


Figure 157 : Structures initiales de *fulfil*

Voir la section III.3.48 (page iii-44, Annexe III) pour divers exemples d'applications.

where, assign, assignedTo, valueIn, noValueIn, List, ListOfDiffEle, memberOf

Similaire à `Mwhere` de M3, le méta-arc `where` de M2 permet de représenter les cas où une règle peut être personnalisée en remplaçant d'abord une ou des variables dans cette règle par des éléments concrets, puis être appliquée à une situation. D'une manière similaire aux éléments `Massign`, `MassignedTo`, `MvalueIn`, `MnoValueIn`, `MList`, `MListOfDiffEle` et `MmemberOf` de M3, les éléments `assign`, `assignedTo`, `valueIn`, `noValueIn`, `List`, `ListOfDiffEle` et `memberOf` de M2 supportent, au niveau M1, le traitement des variables et des listes, nécessaire à la représentation des règles définies au niveau M1. Les structures initiales de `where`, `assign`, `assignedTo`, `valueIn`, `noValueIn` et `memberOf` sont illustrées à la Figure 158. La sémantique et le comportement de ces méta-éléments sont décrits en détail aux sections III.3.49 (page iii-45), III.3.50 (page iii-45) et III.3.51 (page iii-45) de l'Annexe III.

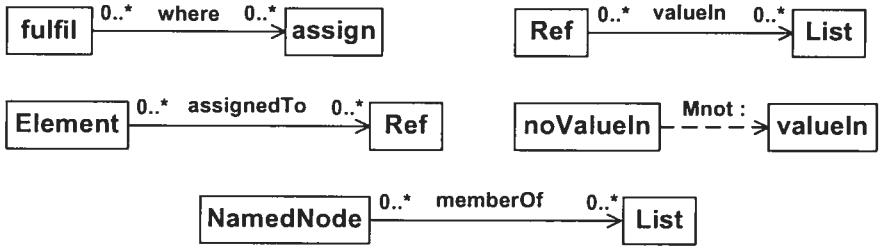


Figure 158 : Méta-arcs *where*, *assignedTo*, *valueIn*, *noValueIn*, *memberOf*

5.3.8 Représentation de contraintes

Relativement à la représentation de contraintes, notre métamodèle supporte la représentation de contraintes structurelles et la représentation de règles. Nous considérons la représentation d'autres types de contraintes nécessaires à la représentation de connaissances. Par exemple :

(1) L'opérateur de la négation logique entre les types de relations est implémenté par le méta-arc `not`.

(2) Le méta-arc `subset` est défini pour représenter les contraintes de sous-ensemble ou bien de dépendance d'inclusion.

(3) Le méta-arc `xor` sert à mettre en œuvre l'opérateur «ou exclusif» («xor» en anglais) logique au niveau M1.

Les structures des méta-arcs `subset` et `not` illustrées dans la Figure 159 sont faciles à comprendre. Leurs exemples d'applications sont présentés dans les sections de l'évaluation : section 6.2.2.8 (page 179) et section 6.2.2.15 (page 190).

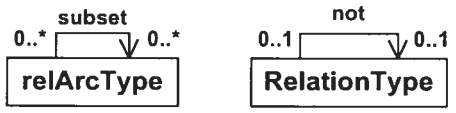


Figure 159 : Structures initiales de *not*, *subset*

Nous expliquons ci-après la raison d'être des structures de `xor` illustrées dans la Figure 160.

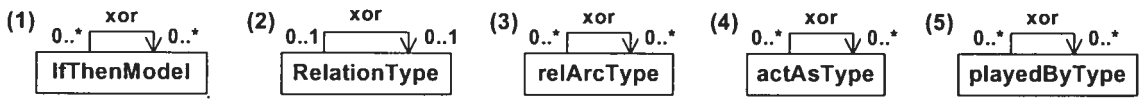


Figure 160 : Structures initiales de *xor*

L'opérateur «ou exclusif» logique retourne *true* si et seulement si une seule des conditions (c'est-à-dire un seul des opérandes) est évaluée vraie. Au niveau M1, afin de distinguer plus facilement les conditions (ou bien hypothèses) et les faits (cf. la Définition 8, page 40, et l'exemple 6, page 39), les `IfThenModel` sont à contextualiser des conditions, et les modèles non conformes à `IfThenModel` sont à contextualiser des faits. Nous définissons alors une structure de `xor` entre `IfThenModel` (Figure 160-(1)) et la Règle 37 (Annexe IV) exprime la sémantique de la structure. La structure de `xor` illustrée dans la Figure 160-(2) sert à représenter la contrainte d'exclusivité entre les types de relations (cf. la Définition 11, page 181) et la Règle 38 (Annexe IV) est à retenir. Les autres structures de `xor` illustrées dans les Figure 160-(3),(4),(5) servent respectivement à représenter la contrainte d'exclusion sur les types de relations (cf. la Définition 13, page 183), celle sur les types de rôles (cf. la Définition 14, page 183) et la contrainte d'exclusivité pour un type sur des types de rôles (cf. la Définition 12, page 181), vis-à-vis des trois règles sémantiques retenues, Règle 39, Règle 40 et Règle 41 (Annexe IV). Les illustrations d'utilisation de ces structures de `xor` sont présentées dans les sections de l'évaluation: section 6.2.2.9 (page 180) et section 6.2.2.10 (page 183).

5.3.9 Représentation d'usagers

Pour représenter au niveau M1 les utilisateurs et leurs interactions avec les objets de connaissances, notre métamodèle dispose des éléments `User` et `create`, similaires à `MUser` et `Mcreate` de M3. Nous pouvons donc spécifier quel élément (`Element`) est créé par quel utilisateur (`User`) (Figure 161).

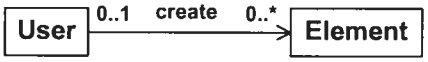


Figure 161 : Structure initiale de *create*

5.3.10 Autres éléments

Nous définissons également les éléments dans le but de supporter d'autres fonctionnalités telles que la comparaison entre les éléments, la représentation de termes équivalents et la représentation d'opérations mathématiques.

Comparaison entre éléments

D'une manière similaire aux méta-arcs `Mcompare`, `M=`, `M==`, `M!=`, `MforValues`, `M>`, `M<`, `M<=`, `M>=` de M3, les méta-arcs `compare`, `=`, `==`, `!=`, `forValues`, `>`, `<`, `<=`, `>=` de M2 permettent de supporter, au niveau M1, différents types de comparaison entre les éléments (Figure 162). Voir la section III.3.56 (page iii-47, Annexe III) pour plus de détails à ce sujet.

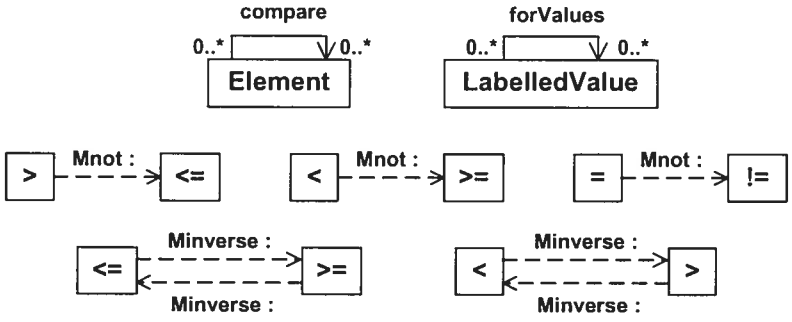


Figure 162 : Structures initiales de *compare*, *forValues*, *>*, *<*, *<=*, *>=*, *=*, *!=*

Représentation de termes équivalents

Afin d'indiquer des termes équivalents ou de faire la correspondance entre les éléments, le méta-arc `Meqv` est utilisé au niveau méta et `eqv` est utilisé au M1 (Figure 163).

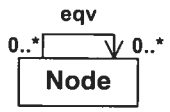


Figure 163 : Structure initiale de *eqv*

Opérations mathématiques

Les méta-arcs +, -, *, / sont définis pour coder respectivement les opérateurs des opérations mathématiques : «+» de l'addition, «-» de la soustraction, «*» de la multiplication et «/» de la division. Une telle opération prend deux valeurs (`LabelledValue`) comme entrées et elle retourne une valeur comme résultat, ce qui est indiqué par des arcs de type `resultVal`. Les méta-arcs +, -, *, / sont les sous-types de `math`, ce qui permet de déduire, par le sous-typage, les structures de +, -, *, /, `resultVal` à partir des structures de `math` et `resultVal` illustrées dans la Figure 164.

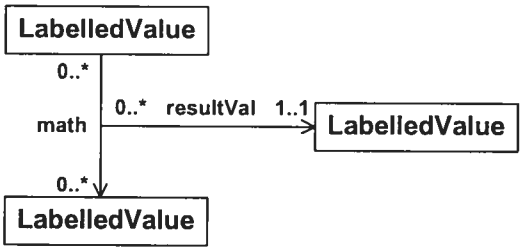


Figure 164 : Structures initiales de *math*, *resultVal*

5.4 Synthèse

Compte tenu des besoins présentés dans le deuxième chapitre concernant la représentation de modèles répartis à tous les niveaux de modélisation, nous avons présenté dans ce chapitre, tous les éléments du méta-métamodèle (M3), et du métamodèle (M2) de notre formalisme ainsi que leurs interactions. Le méta-métamodèle se définit lui-même et il est extensible. Au moyen de ce méta-métamodèle, nous définissons, au niveau M2, un métamodèle en réponse à tous nos besoins concernant la modélisation du monde réel. L'évaluation du pouvoir d'expression de notre formalisme est présentée dans le prochain chapitre.

Chapitre 6. Évaluation du pouvoir d'expression

Ce présent chapitre porte sur l'évaluation du pouvoir d'expression de notre formalisme. Plus spécifiquement, nous aborderons les questions de: (i) l'évaluation de notre méta-métamodèle concernant la représentation des métamodèles, et (ii) l'évaluation de notre métamodèle concernant la représentation de modèles de connaissances.

6.1 Évaluation du méta-métamodèle

Cette section présente l'évaluation de notre méta-métamodèle (M3) au niveau théorique et pratique. L'évaluation théorique a pour finalité de montrer que M3 répond à tous les besoins essentiels identifiés à la section 3.2 (page 41). L'évaluation pratique consiste, pour sa part, à apporter des illustrations de mise en œuvre de M3 dans la représentation de métamodèles de types différents au niveau M2. Il est important de noter que M3 est extensible. En se basant sur le noyau de M3, il est possible de définir et d'ajouter de nouveaux éléments, si besoin est.

6.1.1 Évaluation théorique

BesoinM3 1 : Typage entre éléments de deux niveaux de modélisation consécutifs

Oui. Notre M3 supporte toutes les définitions de base qui sont nécessaires au niveau M3: méta-élément / élément; méta-nœud / nœud; méta-arc / arc (méta-lien / lien binaire).

Comme nous l'avons mentionné, la représentation de toute chose à modéliser repose fondamentalement sur les *nœuds* et les *arcs* (cf. la section 5.1, page 93): un modèle est représenté par un ensemble d'éléments et de relations entre ces éléments; un élément est représenté par un nœud ou un arc; une relation unaire, binaire ou n-aire qui unit un ensemble d'éléments peut être représentée par un nœud et un ensemble d'arcs. Étant donné qu'un arc code un lien binaire, unidirectionnel et qu'un méta-arc code un méta-lien binaire, unidirectionnel, nous prouvons ci-dessous que notre M3 permet également de représenter des méta-liens et des liens, binaires et bidirectionnels.

Un lien binaire et bidirectionnel entre deux éléments (dont chacun est un nœud ou arc), soit e_1 et e_2, est formé par un arc de e_1 à e_2 et par un autre arc de e_2 à e_1 où les deux méta-arcs de ces deux arcs sont inverses l'un de l'autre et ils forment le méta-lien

du lien binaire et bidirectionnel en question. Voici quelques exemples de méta-liens binaires et bidirectionnels dans notre M3. Les méta-arcs `M>` et `M<` forment un type de comparaison «supérieur-inférieur» qui est un méta-lien binaire et bidirectionnel, entre deux valeurs. Les méta-arcs `M<=` et `M>=` forment un type de comparaison «inférieur ou égal – supérieur ou égal» entre deux valeurs, et ce type est aussi un méta-lien binaire et bidirectionnel.

BesoinM3 2 : Passage entre niveaux types et instances

Oui. Dans notre formalisme, un méta-nœud est traité comme une instance de `MMetaNode` et comme un type de nœuds conformes à ce méta-nœud; un méta-arc est traité comme une instance de `MMetaArc` et comme un type d'arcs conformes à ce méta-arc.

BesoinM3 3 : Distinction entre différentes notions de conformité au niveau méta

Oui. `Mmeta` implémente le rapport de conformité, au niveau méta, entre éléments et méta-éléments (c'est-à-dire le rapport de conformité entre les éléments au niveau M2 et les méta-éléments au niveau M3 et celui entre les éléments au niveau M3). Alors que `Msem` implémente le rapport de conformité, au niveau méta, entre les modèles et les métamodèles (c'est-à-dire le rapport de conformité entre les modèles au niveau M2 et M3 et celui entre M3 et lui-même).

BesoinM3 4 : Hiérarchisations entre types

Oui. Notre M3 permet le sous-typage multiple entre les méta-éléments, y compris les méta-nœuds (étant des types non relationnels) et les méta-arcs (étant des types relationnels). Ce type de sous-typage est implémenté par `MsubType`.

BesoinM3 5 : Contraintes de cardinalité min/max pour un méta-lien

Oui. Dans notre formalisme, les contraintes de cardinalité maximale/minimale sur la source et sur la destination pour un méta-arc (c'est-à-dire un méta-lien) sont spécifiées par des arcs de type `Mcard`.

BesoinM3 6 : Distinction entre modèles de différentes natures

Oui. Notre M3 permet de distinguer, parmi les modèles (`MModel`) aux niveaux M2 et M3, les métamodèles (`MMetaModel`), les structures (`MStructure`) et les modèles de conditions (`MIfThenModel`).

BesoinM3 7 : Liens avec les modèles

Oui. Notre M3 supporte les types essentiels de liens avec les modèles. `MdefIn`, `Mcontain` permettent de représenter des liens de contenant entre un modèle et ses éléments. Le méta-arc `Mextend` représente l'extension de modèles. Le méta-arc `Msem` sert à représenter le lien de conformité entre un modèle et un métamodèle. Les méta-arcs `Mjoin`, `Mdiff`, et `Mintersection` permettent de représenter les opérations de jointure, de différence et d'intersection entre modèles; `Mresult` pour indiquer des modèles résultant de ces opérations; `Minfer` pour spécifier qu'un modèle infère un autre modèle; `Mrestrict` pour spécifier qu'un modèle est plus restrictif qu'un autre modèle. De plus, notre M2 définit d'autres types de relations relatives aux modèles. Par exemple, `MdefAs` permet de relier un méta-arc à une structure de définition propre à ce méta-arc; `Mif` et `Mthen` permettent d'indiquer respectivement pour une règle (`MRule`), le ou les modèles de préconditions (`MIfThenModel`) et le ou les modèles de postconditions (`MIfThenModel`).

6.1.2 Évaluation pratique

Nous avons montré que notre M3 répondait à tous les besoins. Ce M3 permet de représenter des métamodèles de types différents au niveau M2. Pour illustrer cette capacité, nous allons voir comment notre M3 met en œuvre la famille de métamodèles sNets, la famille de formalismes orienté-objet et la famille de métamodèles RDF/RDFS/OWL, pour représenter le monde réel. À noter que dans les figures d'illustrations, les liens traversant deux niveaux adjacents sont des liens de conformité entre les éléments et les méta-éléments ainsi que ceux entre les modèles et les métamodèles.

Notre M3 et sNets

Notre M3 permet d'implémenter tous les métamodèles de la famille des métamodèles sNets. Dans cette famille, les méta-entités se conforment à notre élément `MMetaNode`, les méta-liens se conforment à notre élément `MMetaArc`, et les univers sémantiques se conforment à notre élément `MMetaModel`. La Figure 165 montre un exemple de la représentation de `Marie` dans sNets dont on trouve les éléments nécessaires au niveau M2. Dans cette figure, `Marie` définie dans le modèle `UnModèleM1sNets` au niveau M1 est conforme à `sNetsObject` dans le contexte global. `Marie` est également une

instance de `Personne` dans le contexte local (ce qui est spécifié par un lien d'instanciation `sNetstype` sur l'axe horizontal). L'élément `Personne` est conforme à `sNetsClass`. Dans la famille des métamodèles sNets, `sNetsObject` et `sNetsClass` sont deux méta-entités qui représentent respectivement l'ensemble de tous les objets et celui de tous les types d'objets; ils sont donc conformes à `MMetaNode`. Le méta-lien `sNetstype` entre `sNetsObject` et `sNetsClass` est conforme à `MMetaArc`. Représentant un univers sémantique dans sNets, `M2sNet` est conforme à `MMetaModel` (ce qui est indiqué par un arc de type `Mmeta`), et le contenu de `M2sNet` est conforme au contenu de `NotreM3`, c'est-à-dire à notre M3 (ce qui est indiqué par un arc de type `Msem`).

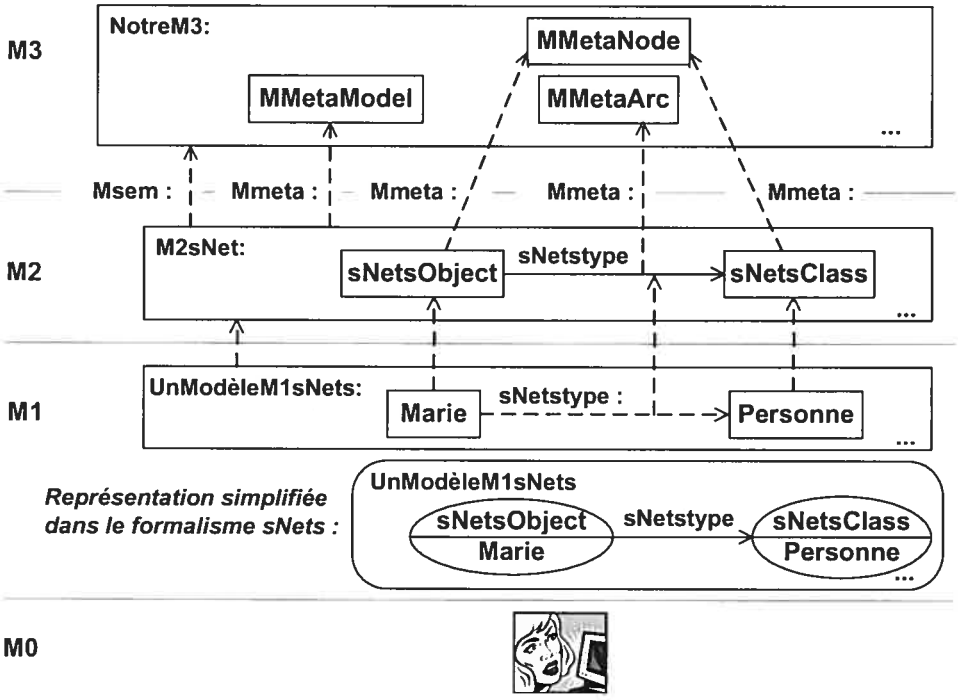


Figure 165. Marie et sNets

Notre M3 et les formalismes orienté-objet

Notre M3 permet aussi d'implémenter les formalismes orienté-objet, par exemple UML. Les métaclasses UML sont conformes à notre élément `MMetaNode`, les associations entre ces métaclasses sont conformes à notre élément `MMetaArc` et les packages, dans le métamodèle UML, sont conformes à notre élément `MMetaModel`.

La Figure 166 décrit partiellement le diagramme de classes dans le package *Core::Constructs* du métamodèle UML [113]. Dans ce diagramme, les métaclasses `Classifier`, `Class`, `Relationship`, `Association`, `Property`, `Operation` sont

conformes à MMetaNode, et elles représentent respectivement l'ensemble des classificateurs, des classes, des types relationnels, des associations, des attributs, et des opérations au niveau M1. Association est un sous-type de Classifier et de Relationship; Class est un sous-type de Classifier (ce qui est indiqué par des arcs de type MsubType). Le méta-arc memberEnd est une association entre Association et Property. Ceci spécifie qu'une Association détient au moins deux Property comme ses extrémités (AssociationEnd) et qu'un Property peut être une extrémité d'une seule Association. Le méta-arc attribute est une association entre Classifier et Property, ce qui spécifie qu'un Classifier peut prendre plusieurs Property pour ses attributs et qu'un Property peut être un attribut d'au plus un Classifier. Le méta-arc ownedAttribute est une association entre Class et Property. Ceci spécifie qu'une Class peut posséder plusieurs Property comme ses attributs propres et qu'un Property peut être un attribut d'au plus une Class. Le méta-arc ownedOperation est une association entre Class et Operation. Ceci indique qu'une Class peut posséder plusieurs Operation comme ses opérations/méthodes et qu'une Operation peut être une opération/méthode d'au plus une Class.

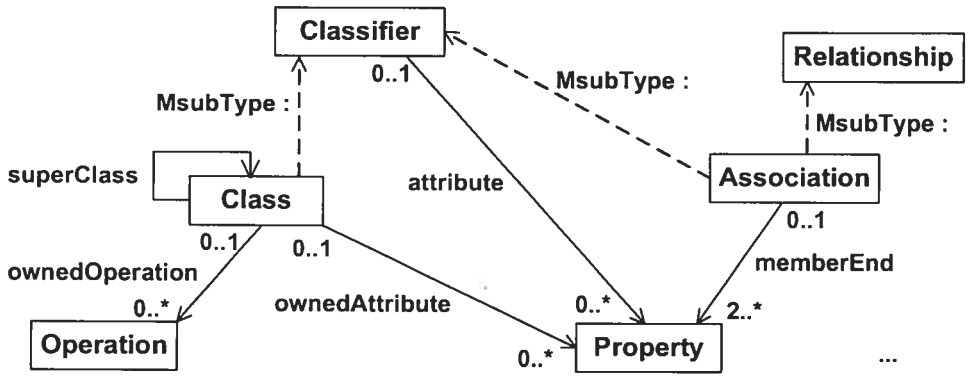


Figure 166. Diagramme de classes selon le package *Core::Constructs* de UML2.0

La Figure 167 illustre la représentation de Marie dans UML, un exemple de représentation des choses du monde réel. Dans cette figure, Marie, définie dans le modèle UnModèleUml au niveau M1, est conforme à Instance dans le contexte global. Elle est aussi une instance de Personne dans le contexte local (ce qui est spécifié par un lien d'instanciation («snapshot») de type classifier sur l'axe horizontal). Personne est conforme à Class. Dans UML, Instance et Class

représentent respectivement l'ensemble des instances et des classes. Ils sont donc conformes à `MMetaNode`. Le méta-lien `classifier` entre `Instance` et `Class` est conforme à `MMetaArc`.

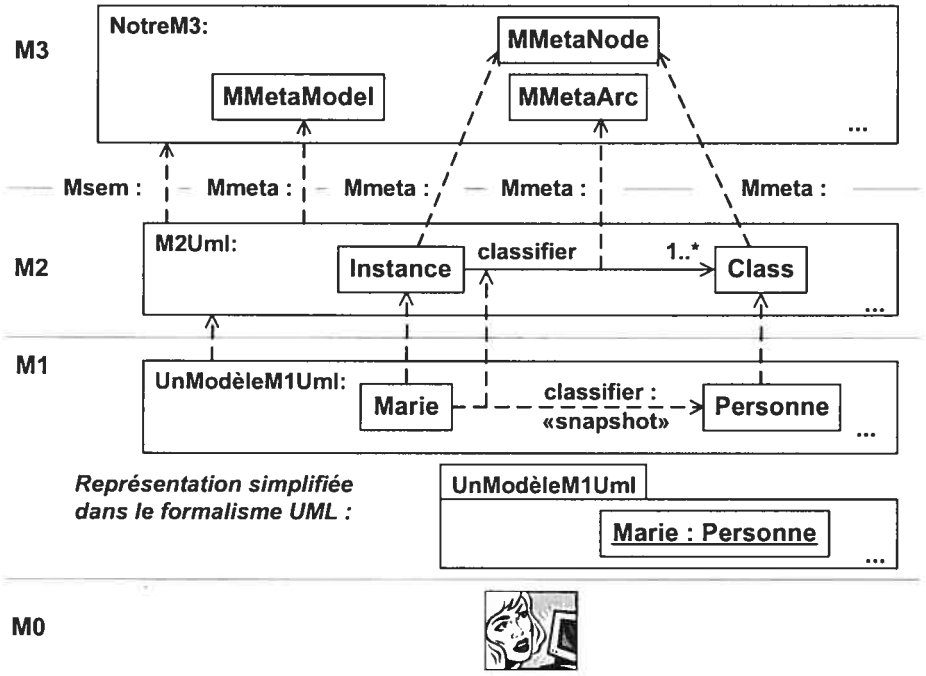


Figure 167. Marie et UML

Notre M3 et RDF/RDFS/OWL

Notre M3 permet également de représenter les formalismes utilisés dans le contexte du web sémantique, par exemple RDF/RDFS/OWL. Dans la famille de métamodèle RDF/RDFS/OWL, les éléments conformes à notre élément `MMetaNode` sont les métaclasses de ressources, par exemple : `xs:schema` / `rdf:RDF` / `owl:Ontology`; `xs:element`; `rdfs:Class` / `owl:Classe`; `rdf:resource` / `owl:Thing`; `rdf:Property` / `owl:ObjectProperty` / `owl:DatatypeProperty`; etc. Les éléments conformes à notre élément `MMetaArc` sont les méta-propriétés, par exemple : `rdf:type`; `rdfs:subClassOf` / `owl:subClassOf`; `rdfs:subPropertyOf`; `rdfs:domain`; `rdfs:range`; `owl:onProperty`; `owl:hasValue`; `owl:imports`; `owl:priorVersion`; etc. Également, les éléments conformes à notre élément `MMetaModel` sont les schémas, les espaces de noms, les ontologies qui contiennent la définition des vocabulaires XML / RDF / RDFS / OWL, par exemple: `xmlns:xs`; `xmlns:rdfs`; `xmlns:rdf`; `xmlns:owl`.

La représentation de `Marie` dans RDF/RDFS/OWL est illustrée par la Figure 168.

Dans cette figure, `Marie`, définie dans une ontologie `UneOntologie` au niveau M1, est conforme à `owlThing` dans le contexte global. Elle est en outre une instance de `Personne` dans le contexte local (ce qui est spécifié par un lien d'instanciation de type `rdfstype` sur l'axe horizontal). L'élément `Personne` est conforme à `owlClass`. Les éléments `owlThing` et `owlClass` représentent respectivement l'ensemble des individus et celui des classes; ils sont donc conformes à `MMetaNode`. Le méta-lien `rdfstype` entre `owlThing` et `owlClass` est conforme à `MMetaArc`.

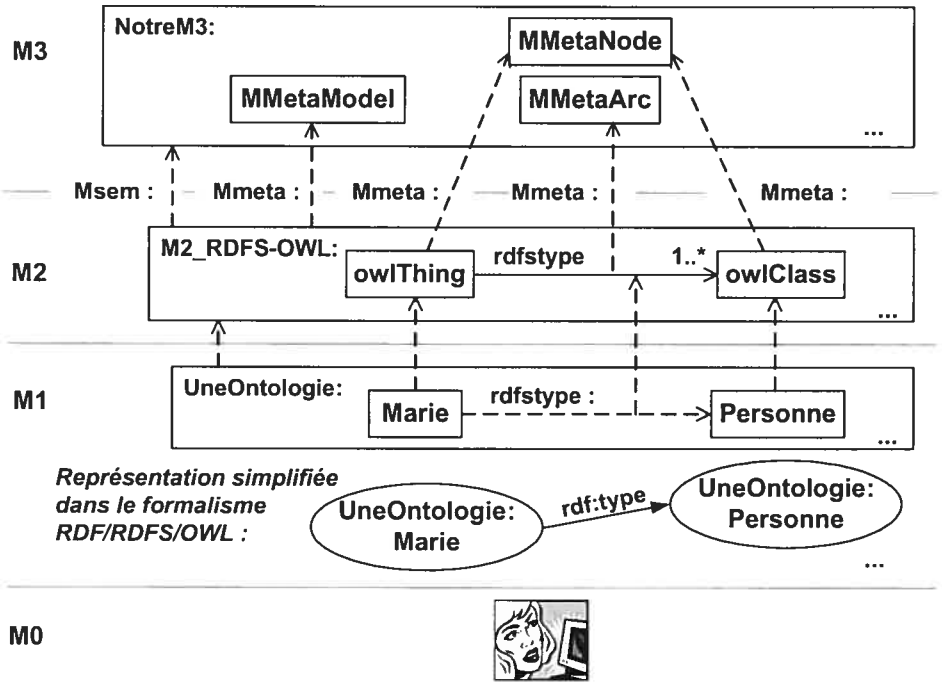


Figure 168. Marie et RDF/RDFS/OWL

6.2 Évaluation du métamodèle

L'évaluation de notre M2 est effectuée en deux étapes. La première étape vise à démontrer que notre M2 répond à tous nos besoins pour M2 (section 3.1) concernant la représentation des modèles de données et de connaissances au niveau M1. Afin d'illustrer le pouvoir d'expression de notre M2, la deuxième étape présente la façon dont nous pouvons modéliser au moyen de notre M2 différentes situations possibles, par exemple celles décrites dans les Problème 1 (page 34) et Problème 2 (page 37). Nous avons vu grâce aux exemples décrits dans ces deux problèmes qu'il était impossible de les modéliser par le biais des formalismes étudiés dans notre revue de littérature (cf. le Chapitre 4, page 43). Cette deuxième étape porte également sur la définition et la

modélisation des besoins concernant les contraintes.

Il est à souligner que notre M2 est extensible, c'est-à-dire que de nouveaux éléments, définis en se basant sur notre M3, peuvent être ajoutés et intégrés au présent M2.

6.2.1 Évaluation théorique

6.2.1.1 Typage entre éléments

BesoinM2 1 : Typage entre éléments du niveau M1

Oui. Notre M2 supporte toutes les notions de base nécessaires pour représenter au M1 le niveau de types et celui d'instances. Au M1, parmi les types (`Type`), nous distinguons : les attributs (`Attribute`); les types d'objets (`ObjType`); les types de rôles (`RoleType`); les types de relations (`RelationType`) (c'est-à-dire les *types relationnels* au M1); et les structures de types de relations (`Structure`). Parmi les instances des types (`TypeInstance`), nous distinguons : les instances d'attributs (`AttrInst`); les objets (`Object`); les rôles (`Role`); les relations (`Relation`) (c'est-à-dire les *relations* instanciés des *types relationnels* au M1); et les contextes de relations (`Context`). Il est à souligner qu'au M1, un type de relations (`RelationType`) ou une relation (`Relation`) peut unir des types (`Type`) et/ou des instances (`TypeInstance`), ce qui signifie que notre M2 permet de représenter tous les types de relations ainsi que les relations pouvant exister dans le monde réel.

BesoinM2 2 : Typage entre éléments de deux niveaux de modélisation consécutifs

Oui. Comme pour le BesoinM3 1 pour M3 (cf. la section 6.1.1, page 152).

6.2.1.2 Passage entre les niveaux des types et des instances

BesoinM2 3 : Passage entre niveaux types et instances

Oui. Comme nous l'avons mentionné dans notre formalisme, un élément peut être traité comme une instance à un niveau et comme type à un autre niveau. Dans le cadre des niveaux M2 et M1 en particulier, tout élément (c'est-à-dire nœud ou arc) de M1 est traité comme l'instance d'un méta-élément (méta-nœud ou méta-arc) de M2. De plus, parmi les éléments de M1, nous distinguons les types (`Type`) et les instances (`TypeInstance`). Une instance (`TypeInstance`) est traitée également comme une instance d'un type au niveau M1 (`Type`).

BesoinM2 4 : Distinction entre différentes notions de conformité

Oui. La Figure 58 (page 92) a illustré différentes notions de conformité supportées

par notre formalisme. `instOf` implémente l'instanciation entre les types et les instances au niveau M1. `Mmeta` implémente au niveau méta le rapport de conformité entre les éléments et les méta-éléments (le rapport de conformité entre les éléments au niveau M2 et les méta-éléments au niveau M3 et celui entre les éléments au niveau M3). Pour sa part, `meta` implémente le rapport de conformité entre les éléments au niveau M1 et les méta-éléments au niveau M2. Quant à lui, `Msem` implémente au niveau méta le rapport de conformité entre les modèles et les métamodèles (le rapport de conformité entre les modèles au niveau M2 et notre M3 et celui entre notre M3 et lui-même) alors que `meta` implémente celui entre les modèles de M1 et M2.

6.2.1.3 Classification et Hiérarchisation entre les types

BesoinM2 5 : Hiérarchisations entre types

Oui. Les types de M1 peuvent être organisés en hiérarchie. En effet, notre M2 autorise le sous-typage multiple (`subType`) ainsi que l'héritage multiple (`inherit`) au niveau M1, entre les types d'objets (`ObjType`), entre les types de rôles (`RoleType`), entre les types de relations (`RelationType`), et entre les attributs (`Attribute`). La section III.3.27 (page iii-29, Annexe III) a présenté différents exemples de hiérarchie entre les types de M1. Dans notre formalisme, un modèle représentant une structure (`Structure`) est considéré, par ailleurs, comme un type de M1. Les structures peuvent aussi être organisées en hiérarchie selon le rapport de spécialisation (`restrict`) ou d'héritage (`inheritModel`) qui existent entre elles.

BesoinM2 6 : Multi-classification et Connaissance partielle

Oui. Notre M2 permet la multi–classification. Du point de vue de l'usager, la multi-classification est un support à la connaissance partielle. Du point de vue du système, la connaissance partielle est supportée comme suit: seuls les éléments qui correspondent à l'information connue peuvent être créés, et avec l'ajout de l'information supplémentaire, le système peut migrer dynamiquement un élément instancié d'un type vers des types plus spécialisés. Reprenons l'exemple du problème sur la classification en cas de connaissances partielles posé dans l'exemple 2 (page 28). Prenons le cas où il n'y a aucun support de multi-classification. Si nous avons une voiture à 4 portes, 2 roues motrices, dont l'origine est inconnue (assemblée en Europe à partir de pièces provenant d'Asie), elle est classée dans le type `Véhicule-2RM-4portes`. Mais lorsque son origine est connue (soit l'Asie), elle est migrée vers le type `Véhicule-Asie-2RM-`

`4portes`. Pourtant, grâce au support de multi-classification, si son origine est inconnue, une voiture à 4 portes et 2 roues motrices est simplement classée dans les types `Véhicule-4portes` et `Véhicule-2RM`, et lorsque son origine est connue, soit l'Asie, cette voiture est de plus classée dans le type `Véhicule-Asie`.

6.2.1.4 Modèle de rôles

BesoinM2 7 : Objets, Rôles, Types d'objets, Types de rôles

Oui. Notre M2 définit les méta-éléments `Object`, `Role`, `ObjStatiqueType`, `ObjDynType`, `RoleStatiqueType`, et `RoleDynType` afin d'expliciter au niveau M1 les objets, les rôles, les types statiques d'objets, les types dynamiques d'objets, les types statiques de rôles, et les types dynamiques de rôles.

BesoinM2 8 : Rapports entre objets et rôles / types de rôles

Oui. Les critères répertoriés pour ce besoin sont satisfaits. Par la suite, nous vérifions, en détail, la réponse de notre formalisme à chacun de ces critères. À noter qu'un joueur de rôles peut être un objet, un rôle, une relation ou un contexte.

- (i) Un joueur de rôles peut simultanément occuper des rôles d'un même type ou de types différents.
- (ii) Un rôle est joué par un seul joueur à un instant donné.
- (iii) Différents types de joueurs de rôles peuvent jouer un même type de rôles, autrement dit, des instances (étant des joueurs de rôles) de types différents peuvent jouer un même type de rôles.
- (iv) Un joueur de rôles peut dynamiquement acquérir ou abandonner des rôles. Lorsqu'un joueur de rôles acquiert un rôle et devient le joueur de ce rôle, l'élément représentant ce rôle est rattaché à l'élément représentant ce joueur par un arc de type `playedBy`. Si un joueur abandonne un rôle, il est possible que l'arc de type `playedBy` entre eux soit supprimé. Une autre solution possible dans ce cas est l'utilisation d'un marqueur de temps pour indiquer la période où ce joueur occupe le rôle en question. Un rôle peut exister sans être attaché à un joueur particulier; et évidemment, un rôle cédé par un joueur pourra être acquis par un autre joueur. Ceci permet de satisfaire au critère (v) ci-dessous.
- (v) Un type de rôles peut être transféré d'un joueur de rôles à un autre joueur.
- (vi) Un rôle (ou type de rôles) peut être acquis ou cédé, indépendamment d'autres rôles (ou types de rôles).

- (vii) Les types de rôles peuvent dépendre des types relationnels. Dans notre contexte, un type de rôles peut toujours s'attacher à un type de relations, et un joueur de rôles doit participer à une relation instanciée de ce type de relations pour pouvoir jouer un rôle instancié du type de rôles en question.
- (viii) Différents types de rôles peuvent partager des structures ou des comportements communs. Ceci est supporté par le sous-typage et l'héritage entre les types de rôles. Notre M2 autorise le sous-typage multiple et l'héritage multiple entre les types de rôles. Il permet aussi d'expliciter qu'un type de rôles est statique ou dynamique.
- (ix) La séquence d'acquisition ou de cession des types de rôles peut représenter la restriction. Supposons qu'un type de rôles TR_1 puisse être occupé directement par un type de rôles TR_2 (ce qui est indiqué par un arc de type `playedByType` de TR_1 à TR_2), et que TR_2 puisse être occupé directement par un type de joueurs soit T (ce qui est indiqué par arc de type `playedByType` de TR_2 à T), et que TR_1 ne peut pas être occupé directement par T (ce qui est indiqué par le fait qu'il n'existe aucun arc de type `playedByType` associant TR_1 à T). Dans ce cas, une instance de T ne peut jouer qu'un rôle de type TR_1 sous rôles de type TR_2, c'est-à-dire que ce rôle est acquis directement par un rôle de type TR_2 occupé directement par l'instance de T en question.
- (x) L'état d'un joueur de rôles peut varier dépendamment du rôle ou du type de rôles sous lequel le joueur est adressé. Une relation peut aussi avoir des propriétés représentant partiellement l'état de ses participants.
- (xi) Comme un type de rôles peut avoir des propriétés, les propriétés d'un joueur de rôles peuvent être spécifiées par ses rôles des types de rôles correspondants.
- (xii) L'accès dépend du contexte. Si un joueur est accédé sous un rôle ou un type de rôles, les propriétés propres aux autres rôles ou types de rôles du joueur sont cachés. Notre M2 permet aussi de spécifier la visibilité d'un attribut pour un type ou celle d'un élément dans un modèle, et il dispose de moyens pour contrôler l'accès entre les types ainsi qu'entre les modèles.
- (xiii) Un joueur de rôles et ses rôles ont des identités distinctes.

6.2.1.5 Classification statique et classification dynamique

BesoinM2 9 : Classification dynamique et classification statique

Oui. Comme nous l'avons présenté ci-haut, notre M2 permet de distinguer

explicitement les types statiques d'objets (respectivement de rôles) et les types dynamiques d'objets (respectivement de rôles) par leurs méta-éléments `ObjStatiqueType` (respectivement `RoleStatiqueType`) et `ObjDynType` (respectivement `RoleDynType`). Ceci permet de spécifier si un type d'objets (de rôles) est un sous-type statique ou dynamique d'un autre type d'objets (de rôles), offrant ainsi un support permettant de répondre au besoin «Classification dynamique et Classification statique».

6.2.1.6 Contraintes structurelles

BesoinM2 10 : Contraintes de l'arité min/max

Oui. Les contraintes dans ce groupe sont spécifiées par des arcs de type `arity`, tout en respectant les structures de `arity`.

BesoinM2 11 : Contraintes de cardinalité totale min/max pour un type relationnel

Oui. Les contraintes dans ce groupe sont spécifiées par des arcs de type `totalCard`, tout en respectant les structures de `totalCard`.

BesoinM2 12 : Contraintes de cardinalité locale min/max pour un type relationnel

Oui. Les contraintes dans ce groupe sont spécifiées par des arcs de type `localCard`, tout en respectant les structures de `localCard`.

BesoinM2 13 : Contraintes de l'itération min/max pour un type relationnel

Oui. Les contraintes dans ce groupe sont spécifiées par des arcs de type `iterate`, tout en respectant les structures de `iterate`.

BesoinM2 14 : Contraintes de cardinalité min/max pour un méta-lien

Oui. Concernant les métamodèles définis par notre méta-métamodèles, les contraintes de cardinalité maximale/minimale sur la source et sur la destination pour un méta-arc (c'est-à-dire un type relationnel au niveau méta) sont spécifiées par des arcs de type `Mcard`. Pour illustration, l'exemple 20 (page 165) dans la partie de l'évaluation pratique démontrera comment notre M2 permet de modéliser la situation du Problème 1 (page 34).

6.2.1.7 Types relationnels entre types et/ou instances

BesoinM2 15 : Types relationnels entre types et/ou instances

Oui. Notre M2 permet de représenter des types relationnels entre les types et/ou les instances. Une illustration a été présentée dans l'exemple 14 (page 130). Les illustrations

plus complexes, concernant la modélisation des situations décrites dans le Problème 2 (page 37), seront présentées dans l'exemple 21 (page 166) et l'exemple 22 (page 167) dans la partie portant sur notre évaluation pratique.

6.2.1.8 Représentation de modèles et de leurs liens

BesoinM2 16 : Distinction entre modèles de différentes natures

Oui. La nature d'un modèle au niveau M1 est explicitée par son méta-élément au niveau M2. De cette façon, notre M2 permet de distinguer, parmi les modèles au niveau M1 (`Model`), les structures (`Structure`), les contextes (`Context`), les modèles d'états (`ObjCycle`, `RoleCycle` respectivement pour les types d'objets et pour les types de rôles), et les modèles de conditions (`IfThenModel`). Aussi, les modèles au niveau M2 (`MMetaModel`) sont les métamodèles des modèles au niveau M1 (`Model`).

BesoinM2 17 : Liens avec les modèles

Oui. Notre M2 définit différents méta-arcs pour représenter les types essentiels de liens avec les modèles. Par exemple, `defIn`, `contain` servent à représenter des liens de contenant entre un modèle et ses éléments. Le méta-arc `extend` sert à l'extension de modèles. `sem` sert à représenter le lien de conformité entre un modèle au M1 et son métamodèle au M2. `join`, `diff`, et `intersection` visent respectivement les opérations de jointure, de différence, et d'intersection entre modèles; et `result` pour indiquer des modèles résultant de ces opérations. `infer` a pour but de spécifier qu'un modèle infère un autre modèle; et `restrict` pour spécifier qu'un modèle est plus restrictif qu'un autre modèle. De plus, ce M2 dispose de méta-arcs pour représenter d'autres rapports relatifs à des modèles au niveau M1. Par exemple, `import` et `inheritModel` représentent aussi les moyens permettant la réutilisation d'éléments dans un modèle de l'extérieur; `defAs` est à relier un type de relations à une structure de définition propre à ce type; `cycleOf` permet de relier un modèle d'états au type d'objets ou de rôles correspondant; `if` et `then` permettent d'indiquer respectivement une règle (`Rule`) le ou les modèles de préconditions (`IfThenModel`) et le ou les modèles de postconditions (`IfThenModel`).

BesoinM2 18 : Liens entre modèles au niveau M1

Oui. Notre M2 dispose également des méta-arcs pour représenter les liens entre modèles mentionnés pour ce besoin. Effectivement, `semAs` a pour représenter les liens

d'équivalence entre les modèles; `viewOf` pour d'indiquer les modèles représentant des vues différentes pour un modèle (respectivement un type d'objets, un objet); `transform` pour les liens de transformation entre les modèles et `fulfil` pour les liens qui indiquent les règles appliquées lors d'une transformation entre deux modèles.

6.2.2 Évaluation pratique

Modélisation des situations présentées dans les problèmes 1 et 2

L'exemple suivant démontre comment notre M2 permet de modéliser la situation du problème 1 (page 34).

Représentation de la situation dans le problème 1 (page 34) (exemple 20)

La structure du type de relations `SignerContrats` peut être modélisée comme dans la Figure 169. `Fournisseur`, `Client` et `Témoin` peuvent être vus comme des types statiques de rôles et être impliqués comme les acteurs dans le type de relations `SignerContrats`, ce qui est indiqué par les arcs de type `actType` (disons respectivement les arcs arc_1, arc_2 et arc_3).

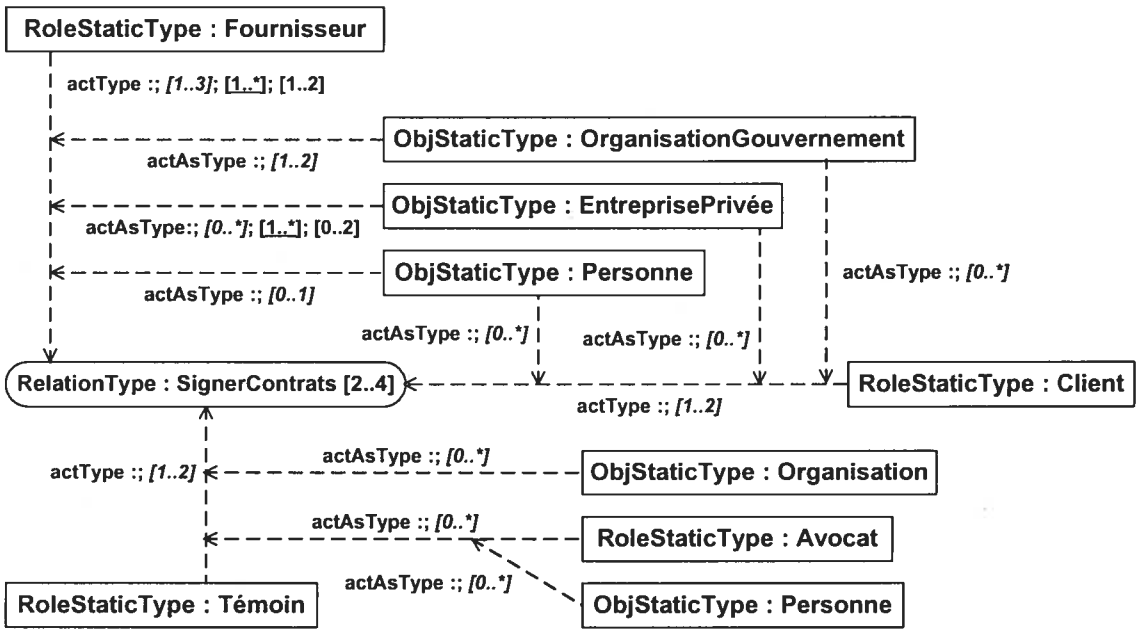


Figure 169 : Structure du type de relations *SignerContrats*

Donc, la (contrainte 1) est notée comme [1..3] sur l'arc arc_1 (qui est de type `actType` et relie `Fournisseur` à `SignerContrats`). La (contrainte 2) est notée comme [1..2] sur l'arc arc_2 (étant de type `actType` et associant `Client` à `SignerContrats`). Et la (contrainte 3) est notée comme [1..2] sur l'arc arc_3 (étant de type `actType` et associant `Témoin` à `SignerContrats`).

Ensuite, la (contrainte 4) est représentée comme suit : chacun des types `OrganisationGouvernement`, `EntreprisePrivée`, `Personne` *et chacun des arcs* arc_1, arc_2 *sont reliés par un arc de type* `actAsType`, *avec les contraintes de l'arité notées comme [0..*]; chacun des types* `Avocat`, `Organisation` *est rattaché à l'arc* arc_3 *par un arc de type* `actAsType`, *avec les contraintes de l'arité notées comme [0..*]; et le type* `Personne` *est rattaché à l'arc de type* `actAsType` *entre* `Avocat` *et* arc_3 *par un arc de type* `actAsType`, *avec les contraintes de l'arité notées comme [0..*].*

Puis, la (contrainte 5) est codée comme suit : à l'arc arc_1, *respectivement,* `OrganisationGouvernement` *est relié par un arc de type* `actAsType`, *avec les contraintes de l'arité notées comme [1..2],* `Personne` *est relié par un arc de type* `actAsType`, *avec les contraintes de l'arité notées comme [0..1],* `EntreprisePrivée` *est relié par un arc de type* `actAsType` *avec les contraintes de l'arité notées donc comme [0..*].*

Sur l'arc arc_1, *les (contrainte 6) et (contrainte 7) sont notées comme* [1..*]; [1..2]. *Les (contrainte 8) et (contrainte 9) sont codées comme* [1..*]; [0..2] *sur l'arc de type* `actAsType` *de* `EntreprisePrivée` *à l'arc* arc_1.

Finalement, la (contrainte 10) sur le nombre de l'itération de `SignerContrats` *est notée comme [2..4] suivant le nom du type* `SignerContrats`.

Les exemple 21 et exemple 22 ci-dessous présentent des illustrations plus complexes concernant la modélisation des situations décrites dans le problème 2 (page 37).

Représentation de la situation 1 dans le problème 2 (page 37) (exemple 21)

Cette situation concerne la modélisation du type `FournisseurTeximus-TémoinAvocatPapin-Client` *en rapport* `SignerContrats` *avec* `FournisseurTeximus` *(une instance de* `Fournisseur`*) et* `TémoinAvocatPapin` *(une instance de* `Témoin`*).*

Étant donné qu'un rôle client peut ne pas appartenir pendant toute son existence au type `FournisseurTeximus-TémoinAvocatPapin-Client`, *ce dernier est donc un type dynamique de rôles, conforme à* `RoleDynType`. `FournisseurTeximus-TémoinAvocatPapin-Client` *hérite de son parent* `Client` *le type de relations* `SignerContrats`. *Mais plus restrictivement, à ce type de relations,* `FournisseurTeximus-TémoinAvocatPapin-Client` *est relié par un arc de type* `actType` *alors que chacune de ces instances* `FournisseurTeximus` *et* `TémoinAvocatPapin` *est reliée par un arc de type* `act`. *Ceci peut être modélisé comme dans la Figure 170 ou encore la Figure 171. La structure de* `SignerContrats` *dans cette figure doit respecter les contraintes attachées à ses structures mères dont la structure initiale de*

SignerContrats est décrite dans le problème 1 (page 34) et illustrée dans la Figure 169 (page 165). Dans cette première structure, Teximus s'implique en tant que le joueur de FournisseurTeximus, Papin en tant que celui de AvocatPapin, et AvocatPapin en tant que celui de TémoinAvocatPapin.

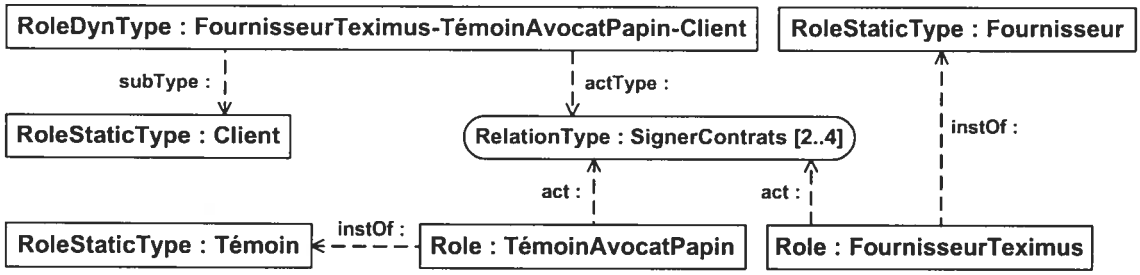


Figure 170 : Représentation de la situation 1 dans le problème 2

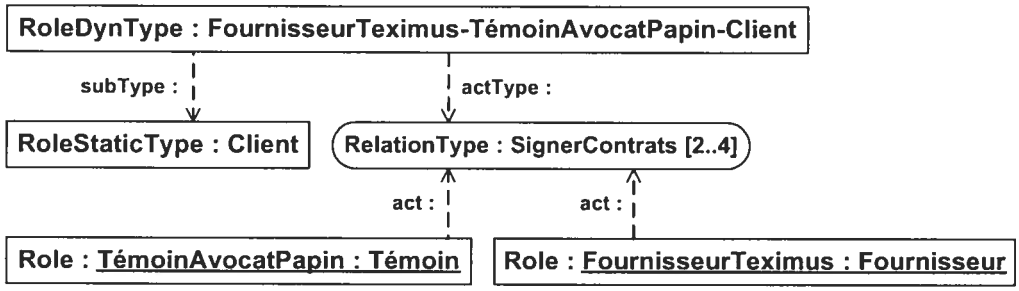


Figure 171 : Version simplifiée de la Figure 170

Représentation de la situation 2 dans le problème 2 (page 37) (exemple 22)

Cette situation concerne la modélisation du type Teximus-Papin-PersonneClient en respectant la structure suivante (S) de SignerContrats: chaque instance de Teximus-Papin-PersonneClient doit participer sous un rôle «client» à au moins une relation de type SignerContrats où Teximus (une instance de EntreprisePrivée) est le seul sous un rôle «fournisseur» et où la personne Papin (une instance de Personne) est le seul joueur, en tant qu'avocat, sous rôles de «témoin». Une représentation de la situation est illustrée par la Figure 172 et expliquée en détail par ce qui suit.

Soit Teximus-Papin-Client, un sous-type de Client représentant l'ensemble de tous les rôles client que les instances de Teximus-Papin-PersonneClient jouent dans les relations de type SignerContrats qui sont conformes à la structure S. Dans la structure S, Teximus-Papin-Client est relié à SignerContrats par un arc de type actType auquel Teximus-Papin-PersonneClient est rattaché par un arc de type actAsType avec les contraintes de cardinalité totale notées comme [1..]. Comme un rôle «client» peut ne pas appartenir durant toute son existence au type Teximus-Papin-Client, ce dernier est*

un sous-type dynamique de `Client` *et est conforme à* `RoleDynType`.

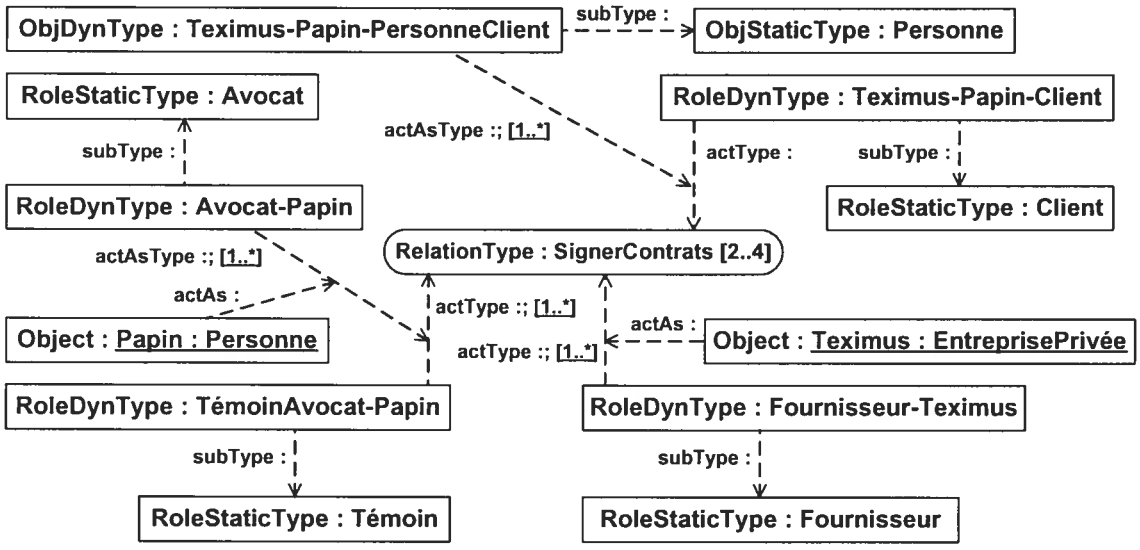


Figure 172 : Représentation de la situation 2 dans le problème 2

Dans les relations de type `SignerContrats` *conformes à la structure* `S`, `Teximus` *s'implique comme le seul joueur sous rôle «fournisseur», et* `Papin` *s'implique comme le seul joueur sous rôles «témoin», en tant qu'avocat. C'est-à-dire, dans ces relations : (i)* `Teximus` *joue un ensemble de rôles «fournisseur»; et (ii)* `Papin` *joue un ensemble de rôles «avocat» qui jouent un ensemble de rôles «témoin».*

L'ensemble des rôles «fournisseur» mentionné dans (i) peut avoir plusieurs membres et est donc représenté par un type soit `Fournisseur-Teximus`, *un sous-type de* `Fournisseur`. *L'ensemble des rôles «avocat» mentionné dans (ii) peut aussi avoir plusieurs membres et est représenté par un type soit* `Avocat-Papin`, *un sous-type de* `Avocat`. *Également, l'ensemble des rôles «témoin» mentionné dans (ii) peut avoir plusieurs membres et est représenté par un type soit* `TémoinAvocat-Papin`, *un sous-type de* `Témoin`. *Étant donné qu'un rôle fournisseur peut ne pas appartenir tout au long de son existence au type* `Fournisseur-Teximus`, *ce dernier est un type dynamique de rôles, conforme à* `RoleDynType`. *Pour une raison similaire,* `TémoinAvocat-Papin` *et* `Avocat-Papin` *sont des types dynamiques de rôles, conformes à* `RoleDynType`. *Donc, comme ce que représente la Figure 172, dans la structure* `S`:

- *le fait (i) peut être codé comme suit:* `Fournisseur-Teximus` *est relié à* `SignerContrats` *par un arc (*$\texttt{arc}_1$*) de type* `actType`, *avec les contraintes de cardinalité totale notées comme [1..*]; et* `Teximus` *est associé à* $\texttt{arc}_i$ *par un arc de type* `actAs`;

– *le fait (ii) peut être codé comme suit:* `TémoinAvocat-Papin` *est relié à* `SignerContrats` *par un arc (*arc_j*) de type* `actType`*, avec les contraintes de cardinalité totale notées comme [1..*]; Avocat-Papin est relié à* arc_j *par un arc (*arc_k*) de type* `actType`*, avec les contraintes de cardinalité totale notées comme [1..*]; et* `Papin` *est associé à* arc_k *par un arc de type* `actAs`*.*

Modélisation des situations suivant les besoins concernant les contraintes

Toute représentation de connaissances nécessite la représentation de contraintes. Nous avons montré dans la section 6.2.1.6 (page 163) que notre M2 permet de représenter toutes sortes de contraintes spécifiant le nombre d'instances d'un modèle structurel. De plus, nous montrons dans cette section, comment notre M2 permet de représenter les contraintes dont la plupart ont été utilisées pour valider le pouvoir d'expression du modèle uniforme des graphes conceptuels (cf. section 5.1.3 – [55]).

6.2.2.1 Contrainte de non nullité

L'attribut pour lequel cette contrainte est spécifiée ne peut pas prendre la valeur *nulle* (il doit être toujours renseigné). Notre formalisme n'autorise pas la valeur nulle. Si un attribut est spécifié comme obligatoire pour un type (cf. la Définition 10, page 131), la valeur concrète de cet attribut, pour une instance du type en question, doit être toujours renseignée. Par contre, si un attribut est spécifié comme optionnel pour un type (cf. la Définition 10, page 131), une instance de ce type peut être ajoutée ou créée même si elle n'a pas l'attribut en question (autrement dit, même si l'attribut en question n'est pas renseigné). Par exemple, la contrainte suivante: le numéro d'identification d'un passeport doit être obligatoirement renseigné. Ceci signifie qu'un passeport ne peut pas être ajouté ou créé sans que son numéro d'identification ne soit défini. Donc, l'attribut représentant les numéros d'identification des passeports est un attribut obligatoire pour le type de passeports (cf. la Définition 10, page 131).

6.2.2.2 Contrainte d'implication

Une contrainte d'implication est représentée par une règle de format «*Si ... alors ...*». Voir l'exemple 23.

Représentation d'une règle au niveau M1 (exemple 23)

Représentons la règle mentionnée dans l'exemple 6 (page 39): si une personne est admise à un programme de maîtrise en informatique, alors elle a complété un baccalauréat en

informatique ou en mathématique. Cette règle (disons `RègleAdmisMScInf`*) peut être interprétée entièrement comme suit: si (i) une personne* (`x`) *est admise à un programme de maîtrise en informatique, alors (ii)* `x` *a complété un baccalauréat en informatique ou (iii)* `x` *a complété un baccalauréat en mathématique. Dans cette règle, l'expression (i) représente la partie de pré-conditions, et les (ii) et (iii) représentent celle de post-conditions. Tel qu'illustré dans la Figure 173-(4) (page 170), l'élément* `RègleAdmisMScInf` *(représentant la règle) est donc associé par un arc de type* `if` *à* `RègleAdmisMScInf-ModèleIf`*, un modèle de conditions contextualisant l'expression (i). Cet élément est aussi associé par des arcs de type* `then` *à* `RègleAdmisMScInf-ModèleThen1` *et* `RègleAdmisMScInf-ModèleThen2`*, deux modèles de conditions contextualisant respectivement les expressions (ii) et (iii). Nous expliquons ci-dessous la mise en œuvre du contenu des modèles* `RègleAdmisMScInf-ModèleIf`*,* `RègleAdmisMScInf-ModèleThen1`*, et* `RègleAdmisMScInf-ModèleThen2`*.*

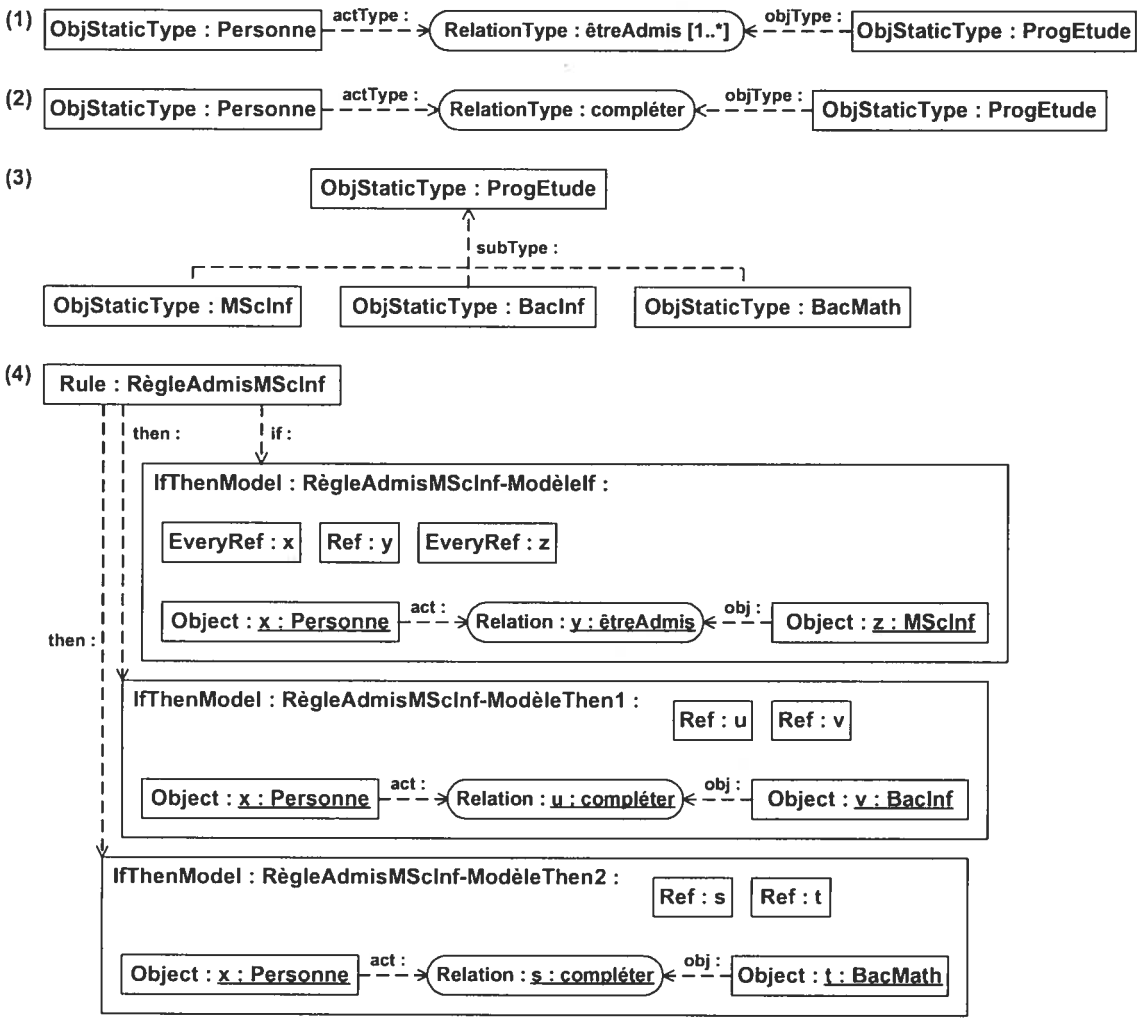


Figure 173 : Exemple de contrainte d'implication

Soient deux types statiques d'objets, `Personne` *et* `ProgEtude`*, qui représentent respectivement l'ensemble des personnes et celui des programmes d'études. La structure illustrée dans la Figure 173-(1) (page 170) sert à spécifier qu'une personne (*`Personne`*) peut être admise (*`êtreAdmis`*) à un programme d'études (*`ProgEtude`*). Celle dans la Figure 173-(2) (page 170) sert à spécifier qu'une personne peut compléter (*`compléter`*) un programme d'études. Soit trois types statiques d'objets,* `MScInf`*,* `BacInf`*, et* `BacMath`*, qui représentent respectivement l'ensemble de programmes de maîtrise en informatique, celui de baccalauréat en informatique, et celui de baccalauréat en mathématique. Ces trois types sont donc des sous-types de* `ProgEtude` *(la Figure 173-(3), page 170). Ils peuvent hériter de* `ProgEtude` *les types de relations* `êtreAdmis` *(cf. la Figure 173-(1), page 170) et* `compléter` *(cf. la Figure 173-(2), page 170). Les Figure 173-(1),(2),(3) (page 170) aident à mieux comprendre ce que représente la Figure 173-(4) (page 170).*

L'expression (i) peut être codée comme le modèle `RègleAdmisMScInf-ModèleIf` *(cf. la Figure 173-(4), page 170). L'élément* `x` *représente une variable désignant n'importe quelle personne (*`Personne`*) qui est admise (*`êtreAdmis`*) à un programme de maîtrise en informatique (*`MScInf`*). L'élément* `x` *est donc déclaré conforme à* `EveryRef`*, ce qui est indiqué par un arc de type* `meta` *de* `x` *à* `EveryRef` *ou plus simplement par la formule «*`EveryRef : x`*». Ce que* `x` *désigne comme une personne (un objet instancié de* `Personne`*) est codé d'un façon simplifiée par la formule «*`Object : x : Personne`*». Soit* `z` *une variable désignant n'importe quel programme de maîtrise en informatique auquel la personne* `x` *est admise. L'élément* `z` *est alors déclaré conforme à* `EveryRef`*, ce qui est indiqué par un arc de type* `meta` *de* `z` *à* `EveryRef` *ou simplement par la formule «*`EveryRef : z`*». Le fait que* `z` *désigne un programme de maîtrise en informatique (un objet instancié de* `MScInf`*) est codé d'un façon simplifiée par la formule «*`Object : z : MScInf`*». La personne* `x` *et le programme* `z` *sont reliés par une relation de type* `êtreAdmis`*. Cette relation est désignée par une variable soit* `y`*. Ceci est codé par les formules «*`Relation : y : êtreAdmis`*» et «*`Ref : y`*» et par le fait que* `x` *et* `z` *sont associés à cette relation (c'est-à-dire à* `y`*) respectivement par un arc de type* `act` *et un arc de type* `obj`*.*

L'expression (ii) peut être codée comme le modèle `RègleAdmisMScInf-ModèleThen1` *(cf. la Figure 173-(4), page 170). L'élément* `v` *est déclaré comme une variable conforme à* `Ref` *(ce qui est indiqué par la formule «*`Ref : v`*») et désignant un programme de baccalauréat en informatique que la personne* `x` *a complété (ce qui est indiqué par la formule «*`Object : z : BacInf`*»). La personne* `x` *et le programme* `v` *sont reliés par une relation de type* `compléter`*. Cette relation est désignée par une variable soit* `u`*. Ceci est indiqué: par les formules*

«Relation : u : compléter» et «Ref : u»; et par ce que x, v *sont associés à* u *respectivement par un arc de type* act *et un arc de type* obj.

D'une manière similaire, l'expression (iii) peut être codée comme le modèle RègleAdmisMScInf-ModèleThen2 *(cf. la Figure 173-(4), page 170). La variable* t *désigne un programme de baccalauréat en mathématique que la personne* x *a complété (ce qui est indiqué par les formules «Ref : t» et «Object : z : BacInf»). La variable* s *désigne une relation de type* compléter *entre la personne* x *et le programme* t. *Ceci est indiqué par les formules «Relation : s : compléter» et «Ref : s» et par le fait que* x *et* t *sont associés à* s *respectivement par un arc de type* act *et un arc de type* obj.

Le pouvoir de représentation de règle peut aider à combler divers types de contraintes présentés plus loin tels que la contrainte de clé unique, d'appartenance à un domaine, de dépendance fonctionnelle, etc.

6.2.2.3 Contrainte de clé unique

Les instances d'un type peuvent être distinguées grâce aux valeurs d'un attribut clé du type. Dans notre formalisme, des arcs de type `isKey` permettent d'indiquer les attributs clés pour les types.

Par exemple, vérifions la contrainte suivante. Le numéro d'identification d'un passeport est unique. Autrement dit, les passeports se distinguent grâce à leurs numéros d'identification. La Figure 133 (page 134) dans l'exemple 16 (page 133) a illustré comment spécifier que le type `Passeport` (représentant le type de passeports) possède `IdNuméro` (l'attribut représentant les numéros d'identification pour les passeports) comme son attribut clé. Il est à noter que la contrainte est équivalente à la règle suivante: si deux passeports sont distincts, alors leurs numéros d'identification sont différents.

6.2.2.4 Contrainte d'appartenance à un domaine

Une contrainte d'appartenance à un domaine pour un attribut (type) vise à déterminer l'ensemble de toutes les valeurs (instances) possibles pour cet attribut (type). Ceci peut être supporté par des modèles de définition de l'attribut (type). Des règles aident également à déterminer l'ensemble des valeurs (instances) possibles pour un attribut (type). Voir l'exemple 24. Il est aussi possible de recenser les valeurs (instances) possibles pour un attribut (type). Voir l'exemple 25.

Contrainte d'appartenance à un domaine (exemple 24)

Soit la définition suivante qui détermine le domaine du type d'âges de personnes: (a) l'âge d'une personne est un nombre supérieur ou égal à 0 et inférieur ou égal à 130, et (b) tous les

nombres dans l'intervalle [0..130] appartiennent à l'ensemble d'âges de personnes. Supposons que l'âge d'une personne soit vu comme une caractéristique de la personne et qu'il soit modélisé comme étant instancié de l'attribut `ÂgePersonne`*. Soit* `Nombre` *un attribut représentant l'ensemble des numéros;* `âgeMinPerssonne` *et* `âgeMaxPerssonne` *deux entiers étiquetés désignant l'âge minimal et l'âge maximal d'une personne et prenant les valeurs 0 et 130. Nous présentons la définition du domaine de* `ÂgePersonne` *suivant les contraintes (a) et (b).*

La contrainte (a) spécifie que l'ensemble des `ÂgePersonne` *est un sous-ensemble de celui des nombres dans l'intervalle [0..130]: (i) pour toute instance* `x` *de* `ÂgePersonne`*, alors (ii)* `x` *désigne un nombre, (iii) la valeur de* `x` *est supérieure ou égale à 0 et inférieure ou égale à 130. Ceci peut être représenté par la règle* `RègleÂgePersonne` *illustrée dans la Figure 174. Deux modèles,* `RègleÂgePersonne-ModèleIf` *et* `RègleÂgePersonne-ModèleThen`*, contextualisent respectivement pour cette règle la partie de pré-conditions (ce qui est indiqué par un arc de type* `if`*) et celle de post-conditions (ce qui est indiqué par un arc de type* `then`*).*

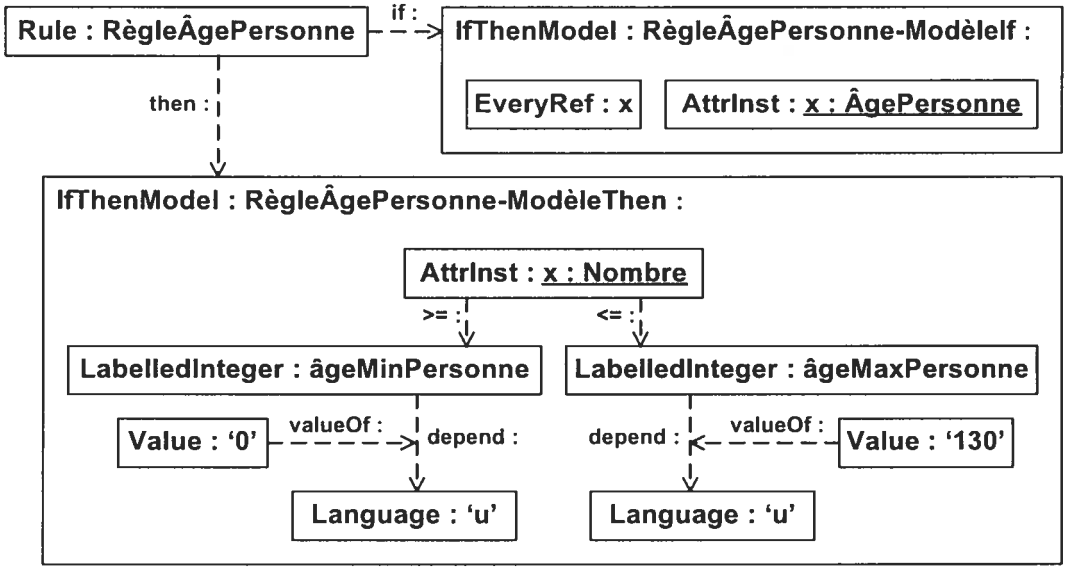


Figure 174 : Contrainte d'appartenance à un domaine

Le modèle `RègleÂgePersonne-ModèleIf` *exprime l'expression (i) par les formules «*`EveryRef : x`*» et «*`AttrInst : x : ÂgePersonne`*». Le modèle* `RègleÂgePersonne-ModèleThen` *exprime les expressions (ii) et (iii). Dans ce modèle, l'expression (ii) est représentée par la formule «*`AttrInst : x : Nombre`*»; et l'expression (iii) est représentée comme suit : la valeur de* `x` *est supérieure ou égale à celle de* `âgeMinPersonne` *(ce qui est indiqué par un arc de type* `>=` *de* `x` *à* `âgeMinPerssonne`*) et inférieure ou égale à celle de* `âgeMinPerssonne` *(ce qui est indiqué par un arc de type* `<=` *de* `x` *à* `âgeMaxPerssonne`*).*

Le modèle `RègleÂgePersonne-ModèleThen` peut être représenté dans une version simplifiée telle qu'illustrée dans la Figure 175.

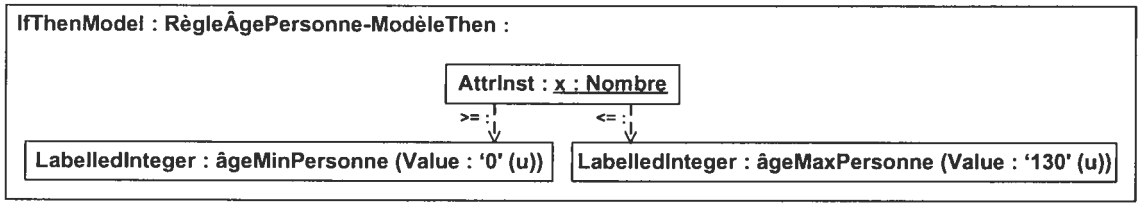


Figure 175 : Version simplifiée du modèle *RègleÂgePersonne-ModèleIf*

La contrainte (b) spécifie que l'ensemble des nombres dans l'intervalle [0..130] est un sous-ensemble de celui des `ÂgePersonne`: (i) pour tout nombre `x`, (ii) la valeur de `x` est supérieure ou égale à 0 et inférieure ou égale à 130, alors (iii) `x` devient une instance de `ÂgePersonne`. Cette contrainte peut être représentée telle que la règle `RègleÂgePersonne02` illustrée dans la Figure 176.

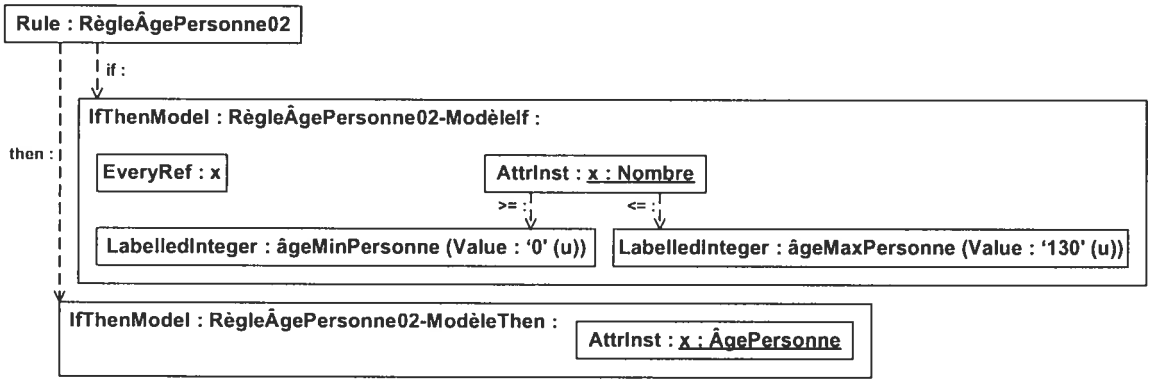


Figure 176 : Contrainte d'appartenance à un domaine

Pour cette règle, `RègleÂgePersonne02-ModèleIf` et `RègleÂgePersonne02-ModèleThen` représentent respectivement les modèles de pré-conditions et de post-conditions. Le modèle `RègleÂgePersonne02-ModèleIf` exprime les expressions (i) et (ii). L'expression (i) peut être codée par les formules «`EveryRef : x`» et «`AttrInst : x : Nombre`». L'expression (ii) peut être codée comme suit: la valeur de `x` est supérieure ou égale à celle de `âgeMinPersonne`, et inférieure ou égale à celle de `âgeMinPersonne`. Le modèle `RègleÂgePersonne02-ModèleThen` exprime l'expression (iii) simplement par la formule «`AttrInst : x : ÂgePersonne`».

Contrainte d'appartenance à un domaine – lister les instances d'un type (exemple 25)

Représentons par exemple, `toto`, `tata`, `tutu`, et `Jean`, toutes les instances de `PersonneGroupe1` (cf. l'exemple 48, page iii-17, Annexe III). Alors (a) `toto`, `tata`, `tutu`,

et Jean sont les instances de PersonneGroupe1, et (b) PersonneGroupe1 n'a pas d'instance autre que toto, tata, tutu, et Jean. L'expression (a) est exprimée par les assertions que toto, tata, tutu, et Jean sont les instances de PersonneGroupe1. Quant à l'expression (b), elle peut être exprimée par la règle RèglePersonneGroupe1 illustrée dans la Figure 177: si x désigne n'importe quelle instance de PersonneGroupe1, x doit être un des membres de ListePersonneGroupe1, une liste dont tous les membres sont toto, tata, tutu et Jean..

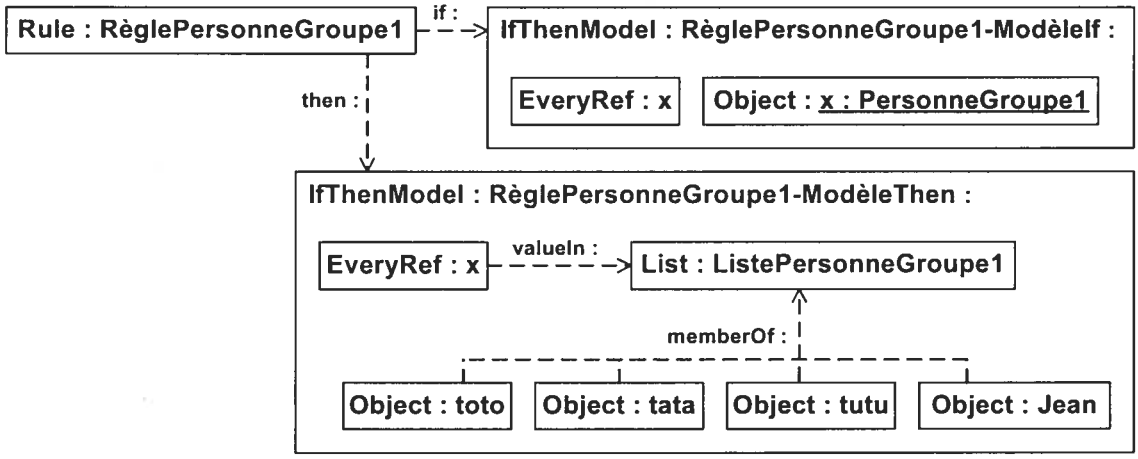


Figure 177 : Contrainte d'appartenance à un domaine - lister les instances d'un type

6.2.2.5 Contrainte de restriction de domaine

Une contrainte de restriction de domaine vise à déterminer l'ensemble des valeurs (instances) possibles pour un attribut (type). C'est un cas particulier de contrainte d'appartenance à un domaine (cf. la section 6.2.2.4, page 172). Voir l'exemple 26.

Contrainte de restriction de domaine (exemple 26)

Continuons l'exemple 24 (page 172). Soit la contrainte suivante: pour toute personne âgée, son âge est supérieur ou égal à 60 ans. Rappelons que Personne est le type représentant l'ensemble de toutes les personnes, que PersonneÂgée est le type de personnes âgées, et que ÂgePersonne est l'attribut représentant le type d'âges de personnes qui caractérise Personne. La contrainte peut être représentée par la règle RègleÂgePersonneÂgée dans la Figure 178 avec l'interprétation suivante: (i) pour tout x une personne âgée et (ii) pour tout y l'âge de x, alors (iii) la valeur de y est supérieure ou égale à 60.

Le modèle de pré-conditions RègleÂgePersonneÂgée-ModèleIf de la règle exprime les expressions (i) et (ii). L'expression (i) est représentée par les formules «EveryRef : x», «Object : x : Personne», et par un arc de type instOf de x à PersonneÂgée.

L'expression (ii) est représentée par les formules «EveryRef : y» et «AttrInst : y : ÂgePersonne», et par un arc de type chrc de y à x. Le modèle de post-conditions RègleÂgePersonneÂgée-ModèleThen de la règle exprime l'expression (iii) comme suit : la valeur de y est supérieure ou égale à celle de âgeMinPersonneÂgée (ce qui est indiqué par un arc de type >= de y à âgeMinPersonneÂgée). Et âgeMinPersonneÂgée représente un entier étiqueté désignant l'âge minimal d'une personne âgée et prend la valeur 60.

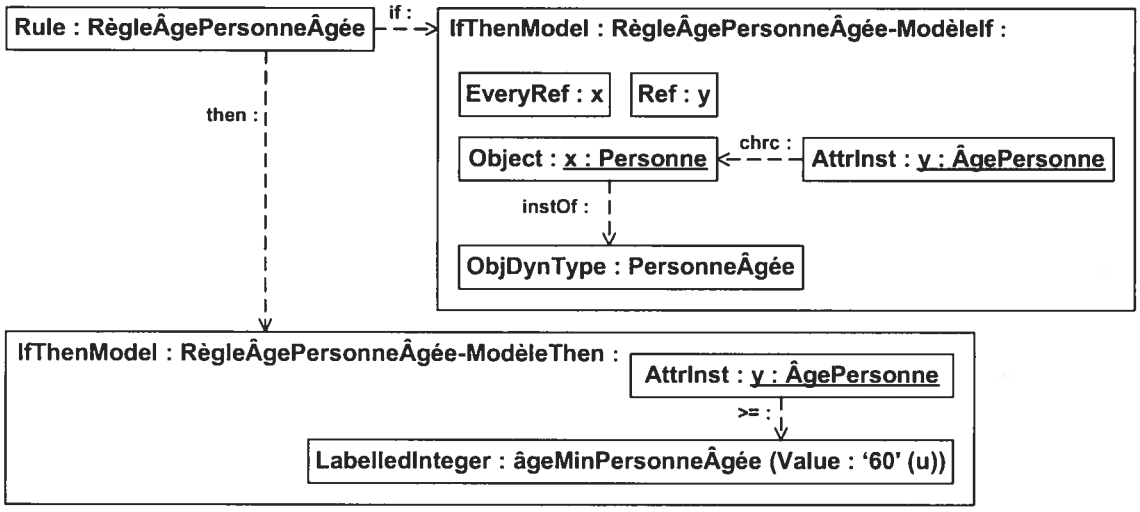


Figure 178 : Contrainte de restriction de domaine

6.2.2.6 Contrainte de dépendance fonctionnelle

Soit R un type de relations entre deux types X et Y. Dans le cadre d'une structure de R, il y a une dépendance fonctionnelle $X \xrightarrow{R} Y$ si pour toute paire t_i et t_j, instances de R, nous avons: $t_i[X] = t_j[X] \Rightarrow t_i[Y] = t_j[Y]$. Autrement dit, X détermine Y si toute instance de X correspond à une et une seule instance de Y [57].

Nous avons donc la définition étendue suivante pour le cas d'un type de relations n-aire. Soit R un type de relations entre les types $X_1, X_2, \ldots, X_m$ et Y. Disons X un ensemble de types, $X = \{X_1, X_2, \ldots X_m\}$. Dans le cadre d'une structure de R, il y a une dépendance fonctionnelle $X \xrightarrow{R} Y$ si pour toute paire t_i et t_j, instances de R, nous avons: $t_i[X] = t_j[X]$ (c'est-à-dire pour chaque k=1, 2, ..., m: $t_i[X_k] = t_j[X_k]$) $\Rightarrow t_i[Y] = t_j[Y]$. Autrement dit, l'ensemble X des types $X_1, X_2, \ldots$, et X_m détermine Y si toute instance de X correspond à une et une seule instance de Y.

Notre formalisme permet d'exprimer cette contrainte de dépendance fonctionnelle par le fait que dans la structure de R, la cardinalité locale minimale et maximale pour Y

prennent la valeur 1. Voir l'exemple 27.

Contrainte de dépendance fonctionnelle (exemple 27)

Étudions l'exemple suivant. Une fois embauchée par une organisation, une personne y jouera un rôle «employé» auquel correspond un profil d'employé. Nous pouvons dire que chaque profil d'employé (ProfilEmployé) appartient à (appartenir-à) un seul rôle employé (Employé), ce qui peut être représenté par une structure illustrée dans la Figure 179. Employé et ProfilEmployé représentent l'ensemble des employés et des profils d'employés. appartenir-à représente le type de relations d'appartenance entre les profils d'employé (ProfilEmployé) et les rôles employé (Employé). Dans cette structure de appartenir-à:

- *ProfilEmployé est l'acteur de l'action (ce qui est indiqué par un arc de type actType);*
- *Employé est l'objet de l'action (ce qui est indiqué par un arc de type objType);*
- *un ProfilEmployé participe à des relations de type appartenir-à avec un seul Employé (ce qui est indiqué par ce que les cardinalités locales minimale et maximale pour Employé dont la valeur est 1, notées comme [1..1] sur l'arc de type objType de Employé à appartenir-à).*

Figure 179 : Contrainte de dépendance fonctionnelle

6.2.2.7 Contrainte de dépendance fonctionnelle inter-relations

Soit S un ensemble non vide de types de relations, $S = (R_1, R_2, ..., R_p)$; pour chaque k=1, 2, ..., p, R_k un type de relations entre deux types X et Y. Il y a une dépendance fonctionnelle inter-relations $X \xrightarrow{S} Y$ si $\forall R_i \in S$, $\forall R_j \in S$, nous avons: $\forall t_i$, instance de R_i, et $\forall t_j$, instances de R_j, $t_i[X] = t_j[X] \Rightarrow t_i[Y] = t_j[Y]$. Autrement dit, pour toute paire R_i et R_j dans S, lorsque l'instance de X dans une instance de R_i est égale à celle dans une instance de R_j, les instances correspondantes de Y sont égales [57].

La définition suivante est étendue pour le cas de types de relations n-aires. Soit S un ensemble non vide de types de relations, $S = (R_1, R_2, ..., R_p)$; et pour chaque k=1, 2, ..., p, R_k un type de relations entre les types $X_1, X_2, ..., X_m$ et Y. Disons X un ensemble de types, $X = \{X_1, X_2, ..., X_m\}$. Il y a une dépendance fonctionnelle inter relations $X \xrightarrow{S} Y$ si $\forall R_i \in S$, $\forall R_j \in S$, nous avons : $\forall t_i$, instance de R_i, et $\forall t_j$, instances de R_j,

$t_i[X] = t_j[X]$ (c'est-à-dire pour chaque k=1, 2, ..., m: $t_i[X_k] = t_j[X_k]$) $\Rightarrow$ $t_i[Y] = t_j[Y]$. Autrement dit, pour toute paire R_i et R_j dans S, lorsque l'instance de X (c'est-à-dire l'ensemble des éléments: l'instance de X_1, celle de X_2, ..., et celle de X_m) dans une instance de R_i est égale à celle dans une instance de R_j, les instances correspondantes de Y sont égales.

Il est à noter qu'une instance de X peut participer à une instance de R_i (ou R_j) mais peut ne participer à aucune instance de R_j (ou R_i).

L'exemple 28 présente un exemple du cas.

Contrainte de dépendance fonctionnelle inter-relations (exemple 28)

Soit une contrainte de dépendance fonctionnelle inter-relations représentée par la règle suivante: si (i) un rôle employé dirige un département et travaille dans un département, alors (ii) ces deux départements doivent être le même. Autrement dit, un rôle employé ne peut pas diriger un département et travailler dans un autre département. Évidement dans ce cas, un rôle «employé» peut diriger un département mais ne travailler dans aucun département, ou il peut travailler dans un département mais ne diriger aucun département. La règle (disons `RègleEmployé01`) représentant la contrainte peut être codée telle qu'illustrée dans la Figure 180-(3).

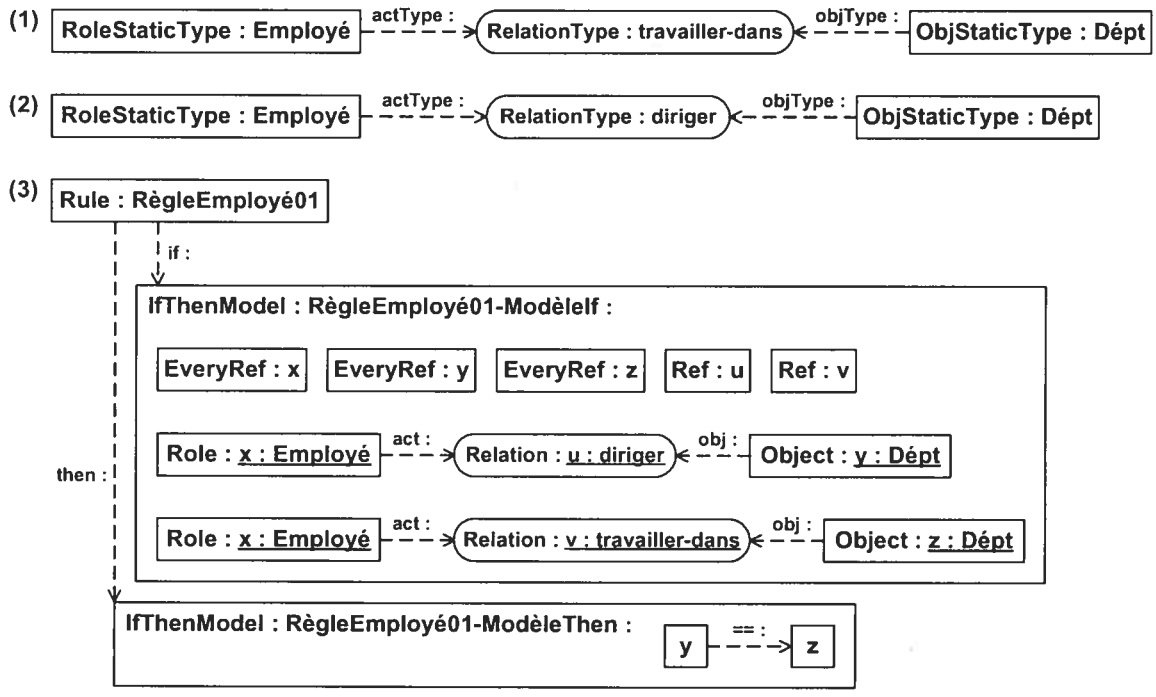


Figure 180 : Contrainte de dépendance fonctionnelle inter-relations

Dans cette figure, l'élément `RègleEmployé01` est associé par un arc de type `if` et par un arc de type `then` respectivement à `RègleEmployé01-ModèleIf` et à

RègleEmployé01-ModèleThen, deux modèles de conditions contextualisant respectivement les expressions (i) et (ii). Nous expliquons ci-dessous la mise en œuvre du contenu des modèles RègleEmployé01-ModèleIf et RègleEmployé01-ModèleThen..

Soit Employé, un type statique de rôles qui représente l'ensemble de rôles «employé»; et Dépt, un type statique d'objets qui représente l'ensemble des départements. La structure illustrée dans la Figure 180-(1) a pour but de spécifier qu'un rôle «employé» (Employé) peut travailler dans (travailler-dans) un département (Dépt). Et celle dans la Figure 180-(2) a pour but de spécifier qu'un rôle «employé» peut diriger (diriger) un département. Ces structures permettent d'expliquer des modèles de relations dans la Figure 180-(3).

Selon le modèle RègleEmployé01-ModèleIf (cf. la Figure 180-(3)) qui exprime l'expression (i), l'élément x *représente une variable désignant n'importe quel rôle employé (Employé), et les éléments* y *et* z *représentent deux variables désignant chacune n'importe quel département de telle sorte que:*

- *(i1) le rôle employé* x *dirige le département* y *(ce qui est indiqué par une relation (désignée par une variable soit* u*) de type* diriger *entre* x *et* y*);*
- *(i2) le rôle employé* x *travaille dans le département* z *(ce qui est indiqué par une relation (désignée par une variable soit* v*) de type* travailler-dans *entre* x *et* z*).*

L'expression (ii) appliquée à ce cas est interprétée par le fait que y *et* z *sont les mêmes. Ceci est indiqué par un arc de type* == *de* y *et* z*, comme représenté par le modèle RègleEmployé01-ModèleThen (cf. la Figure 180-(3)).*

6.2.2.8 Contrainte de dépendance d'inclusion

Soient R_1, R_2 deux types de relations. Soient T_1, T_2 deux types liés respectivement à R_1 et à R_2. Il y a une dépendance d'inclusion de T_2 à T_1 lorsque tout élément qui participe à une relation de type R_2 en tant qu'instance de T_2 avec un ensemble d'éléments (E) doit participer à une relation de type R_1 en tant qu'instance de T_1 avec ce même ensemble d'éléments (E). Autrement dit, pour le même ensemble d'éléments (E), l'ensemble des éléments dont chacun participe à une relation de type R_2 en tant qu'instance de T_2 avec E, est un sous-ensemble de l'ensemble des éléments dont chacun participe à une relation de type R_1 en tant qu'instance de T_1 avec E. Dans notre formalisme, les arcs de type subset permettent de spécifier les contraintes de sous-ensemble ou de dépendance d'inclusion (cf. la Règle 36, Annexe IV). Voir l'exemple 29.

Contrainte de dépendance d'inclusion (exemple 29)

Examinons la situation suivante. Une structure (S_1*) de* `travailler-dans`*, telle qu'illustrée dans la Figure 180-(1), spécifie qu'un rôle «employé» (*`Employé`*) peut travailler dans (*`travailler-dans`*) un département (*`Dépt`*). Une structure (*S_2*) de* `diriger`*, telle qu'illustrée dans la Figure 180-(2), spécifie qu'un rôle «employé» peut diriger (*`diriger`*) un département. Soit la contrainte de dépendance d'inclusion suivante: un rôle «employé» ne dirige que les départements dans lesquels il travaille. Cette contrainte peut être représentée par la règle suivante: si un rôle employé dirige un département, alors il doit travailler dans ce département. Cette règle (*`RègleEmployé02`*) peut être modélisée comme dans la Figure 181 avec cette interprétation: pour tout* `x` *un rôle employé et pour tout* `y` *un département, s'il existe* `u` *une relation de type* `diriger` *de* `x` *à* `y`*, alors il existe* `v` *une relation de type* `travailler-dans` *de* `x` *à* `y`*. Par ailleurs, la contrainte peut être spécifiée simplement comme dans la Figure 182. L'arc de type* `objType` *de* `Dépt` *à* `diriger` *(dans* S_2*) est relié par un arc de type* `subset` *à l'arc de type* `objType` *de* `Dépt` *à* `travailler-dans` *(dans* S_1*).*

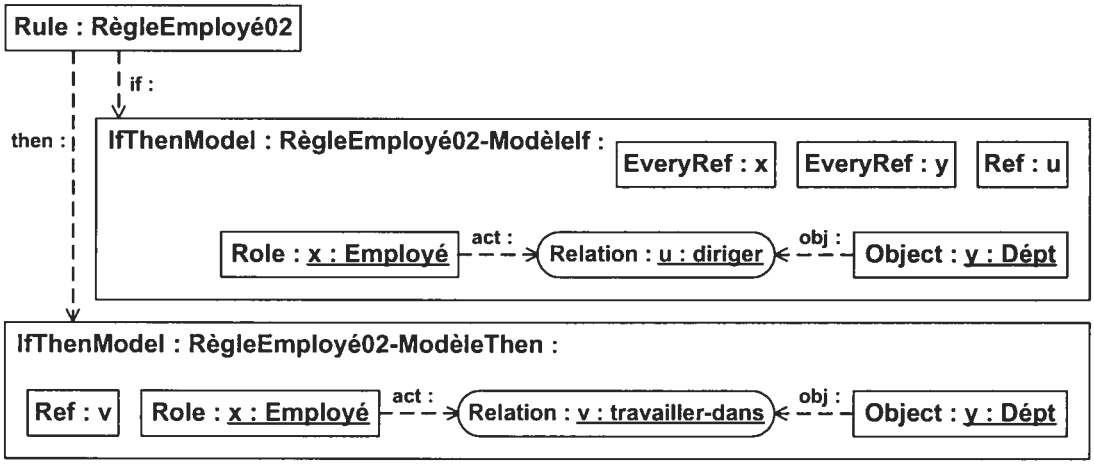


Figure 181 : Contrainte de dépendance d'inclusion sous forme d'une règle

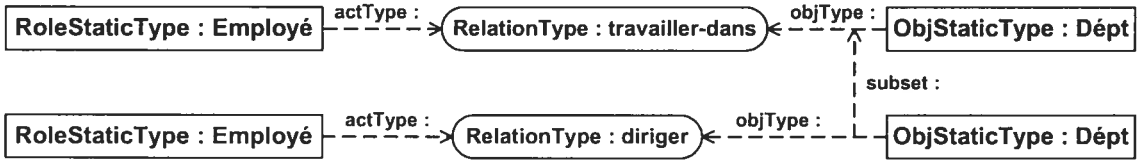


Figure 182 : Contrainte de dépendance d'inclusion

6.2.2.9 Contrainte d'exclusivité

Dans notre formalisme, une contrainte d'exclusivité (cf. la Définition 11) ou celle sur les types de rôles (cf. la Définition 12) peut être représentée par l'utilisation de

l'opérateur `xor`. L'exemple 30 et l'exemple 31 en présentent deux illustrations.

Définition 11 : Contrainte d'exclusivité

Soit deux types de relations R_1, R_2 dont chacun est défini entre un ensemble de types, $X = \{X_1, X_2, \ldots, X_n\}$.

Une contrainte d'exclusivité entre R_1 et R_2 s'exprime comme suit : $\forall\ t_1$, instance de R_1, et $\forall\ t_2$, instance de R_2: $t_1[X] \neq t_2[X]$ (c'est-à-dire il existe un k: $1 \leq k \leq n$, $t_1[X_k] \neq t_2[X_k]$).

Autrement dit, $\forall\ t_1$, instance de R_1, l'ensemble des instances $t_1[X] = \{t_1[X_1], t_1[X_2], \ldots, t_1[X_n]\}$ ne participent à aucune relation de type R_2; et réciproquement, $\forall\ t_2$, instance de R_2, l'ensemble des instances $t_2[X] = \{t_2[X_1], t_2[X_2], \ldots, t_2[X_n]\}$ ne participent à aucune relation de type R_1.

Définition 12 : Contrainte d'exclusivité pour un type sur des types de rôles

Soit n types de rôles TR_1, TR_2, ..., TR_n; et T un type de joueurs commun pour les types de rôles TR_1, TR_2, ..., TR_n. Une contrainte d'exclusivité pour T sur TR_1, TR_2, ..., TR_n exprime que si une instance de T joue un ou des rôles instanciés d'un parmi les types TR_1, TR_2, ..., TR_n, elle ne joue aucun rôle instancié d'un autre type parmi ces derniers types. Comme un type de rôles peut s'engager à des types de relations, la contrainte en question signifie aussi que: si une instance de T s'implique dans une relation sous un rôle instancié d'un parmi les types TR_1, TR_2, ..., TR_n, elle ne s'implique dans aucune relation sous un rôle instancié d'un autre type parmi ces derniers.

Contrainte d'exclusivité (exemple 30)

Soit la contrainte suivante à représenter: un rôle lecteur (`Lecteur`) ne peut pas à la fois réserver (`réserver`) et emprunter (`emprunter`) le même livre (`Livre`). La contrainte peut être exprimée telle qu'illustrée dans la Figure 183. Les deux types de relations `réserver` et `emprunter` sont liés par un arc de type `xor`. Cette contrainte peut également être exprimée en appliquant la structure de `xor` entre les modèles de conditions, tel que représentée à la Figure 184. Dans cette figure, deux modèles de conditions, `Modèle1` et `Modèle2`, sont associés par un arc de type `xor`, ce qui exprime la clause (`Modèle1 xor Modèle2`) et est contextualisé par `Modèle1xorModèle2`, un modèle désigné par un élément conforme à `Model` et non à `IfThenModel`. C'est-à-dire, la classe (`Modèle1 xor Modèle2`) est évaluée à vrai.

Autrement dit on a soit `Modèle1` *soit* `Modèle2` *vrai. Le* `Modèle1` *exprime l'expression (i): pour tout* `x` *un rôle lecteur, pour tout* `y` *un livre,* `x` *réserve* `y` *(ce qui est indiqué par une relation de type* `réserver` *entre* `x` *et* `y`*). Le* `Modèle1` *exprime l'expression (ii):* `x` *emprunte* `y` *(ce qui est indiqué par une relation de type* `emprunter` *entre* `x` *et* `y`*). Alors la clause (*`Modèle1 xor Modèle2`*) peut être interprétée comme suit: pour tout* `x` *un rôle lecteur, pour tout* `y` *un livre, soit (*`x` *réserve* `y`*) soit (*`x` *emprunte* `y`*). Ceci interprète donc bien la contrainte à représenter.*

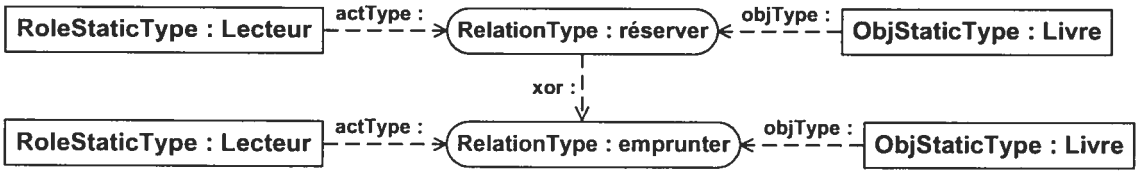


Figure 183 : Contrainte d'exclusivité

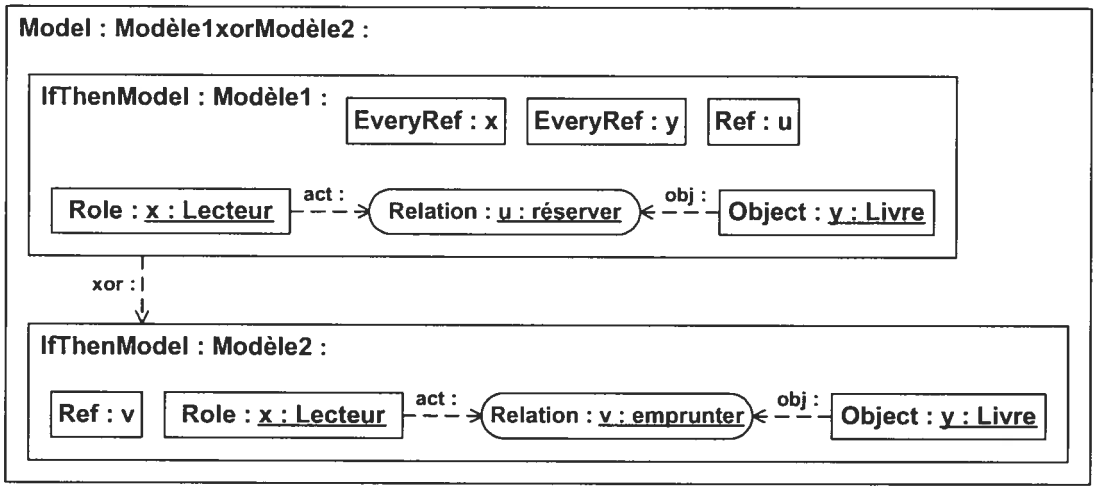


Figure 184 : Contrainte d'exclusivité sous forme d'une règle

Contrainte d'exclusivité pour un type sur types de rôles (exemple 31)

Soit la contrainte d'exclusivité suivante pour un type sur types de rôles : une personne (`Personne`*) ne peut pas jouer à la fois un rôle «avocat» (*`Avocat`*) et un rôle «procureur» (*`Procureur`*), même si ce n'est pas dans le même procès. Cette contrainte signifie aussi qu'une personne qui participe à une relation en tant qu'avocat (ou en tant que procureur) ne peut participer à aucune relation en tant que procureur (ou en tant qu'avocat). La contrainte peut être exprimée telle qu'illustrée à la Figure 185. L'arc de type* `playedByType` *de* `Avocat` *à* `Personne` *et celui de* `Procureur` *à* `Personne` *sont associés par un arc de type* `xor`*. Voir aussi l'exemple 33 (page 184) pour un autre cas concernant les types de rôles «avocat» et «procureur».*

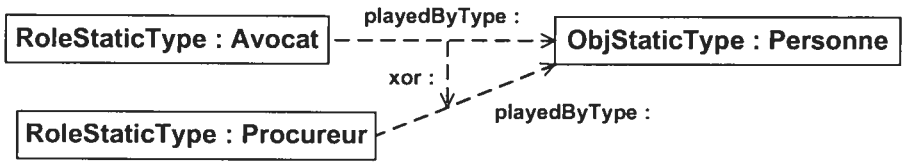


Figure 185 : Contrainte d'exclusivité pour un type sur types de rôles

6.2.2.10 Contrainte d'exclusion

Dans notre formalisme, une contrainte d'exclusion sur les types de relations (cf. la Définition 13) ou celle sur les types de rôles (cf. la Définition 14) est représentée par l'utilisation de l'opérateur `xor`. Voir l'exemple 32 et l'exemple 33.

Définition 13 : Contrainte d'exclusion sur types de relations

Soit n types de relations R_1, R_2, ..., R_n, et un type T commun impliqué dans R_1, R_2, ..., R_n. Une contrainte d'exclusion pour T sur R_1, R_2, ..., R_n dans les structures de R_1, R_2, ..., R_n exprime que : si une instance de T participe à une ou des relations instanciées d'un parmi les types de relations R_1, R_2, ..., R_n, elle ne participe à aucune relation instanciée d'un autre type parmi ces derniers.

Définition 14 : Contrainte d'exclusion sur types de rôles

- Soit deux types de rôles TR_1 et TR_2; et T un type de joueurs commun pour les types de rôles TR_1 et TR_2 impliqués dans un type de relations R. Une contrainte d'exclusion pour T sur TR_1 et TR_2 dans une structure de R exprime que: si une instance de T s'implique dans une relation de type R sous un rôle instancié d'un parmi les types TR_1 et TR_2, elle ne s'implique pas dans cette relation sous rôle instancié d'un autre type parmi ces derniers.
- Soit deux types de rôles TR_1 et TR_2; et T un type de joueurs commun pour les types de rôles TR_1 et TR_2 impliqués respectivement dans des types de relations R_1 et R_2. Une contrainte d'exclusion pour T sur TR_1 et TR_2, respectivement dans les structures des types de relations R_1 et R_2, exprime que: si une instance de T joue un ou des rôles de type TR_1 (ou TR_2) dans une relation de type R_1 (ou R_2), elle ne s'implique dans aucune relation de type R_2 (ou R_1), sous rôles de type TR_2 (ou TR_1).

Contrainte d'exclusion sur les types de relations (exemple 32)

La Figure 186 illustre une représentation de la contrainte d'exclusion sur les types de relations: un rôle employé (`Employé`) ne peut pas simultanément diriger (`diriger`) une

organisation gouvernementale (`OrganisationGouvernement`) et travailler dans (`travailler-dans`) une entreprise privée (`EntreprisePrivée`). Dans cette figure, l'arc de type `actType` de `Employé` à `diriger` et celui de `Employé` à `travailler-dans` sont associés par un arc de type `xor`.

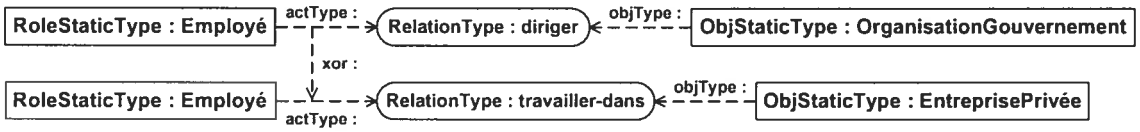


Figure 186 : Contrainte d'exclusion sur les types de relations

Contrainte d'exclusion pour un type sur types de rôles (exemple 33)

Soit la contrainte d'exclusion suivante pour un type sur types de rôles: une personne (`Personne`) ne peut pas jouer à la fois un rôle avocat (`Avocat`) et un rôle procureur (`Procureur`) dans le même procès (`Procès`). Dans ce cas, il est à souligner qu'une personne peut jouer à la fois un rôle «avocat» (`Avocat`) et un rôle «procureur» (`Procureur`), tel qu'illustré à la Figure 187 représente.

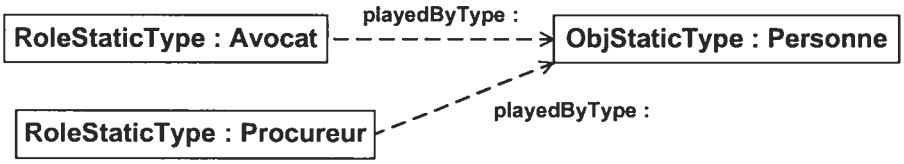


Figure 187 : Personne et les rôles

Alors, la contrainte peut être exprimée comme en témoigne la Figure 188.

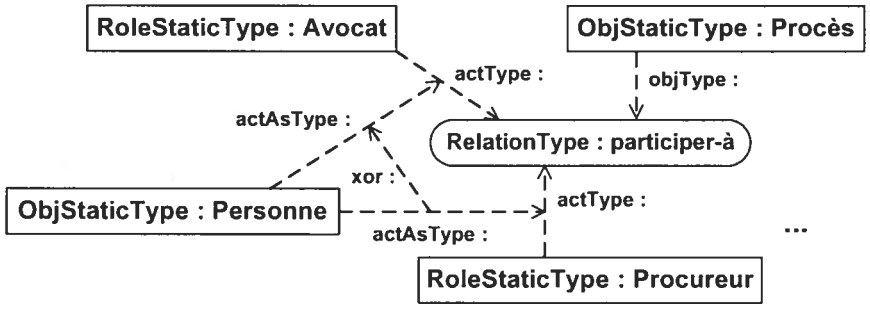


Figure 188 : Contrainte d'exclusion sur les types de rôles

Cette figure décrit une structure de `participer-à`, un type de relations qui implique les types de rôles `Avocat` et `Procureur` comme deux acteurs de l'action (ce qui est indiqué par des arcs de type `actType`) et qui implique le type d'objets `Procès` comme l'objet de l'action (ce qui est indiqué par un arc de type `objType`). Dans cette structure de `participer-à`, les types de rôles `Avocat` et `Procureur` sont joués par le type `Personne`, ce qui est indiqué

par des arcs de type `actAsType`*. Ces deux arcs de type* `actAsType` *sont associés par un arc de type* `xor`*, ce qui exprime la contrainte d'exclusion à représenter.*

6.2.2.11 Contrainte de contexte

Une contrainte de contexte représente une dépendance fonctionnelle entre deux modèles. Voir l'exemple 34.

Contrainte de contexte (exemple 34)

Soit la contrainte de contexte suivante: si (i) un rôle employé travaille dans un département appartenant à une organisation, alors (ii) ce rôle travaille pour un rôle employeur joué par cette organisation. La contrainte peut être représentée par la règle `RègleContexteEmployé` *(voir la Figure 189).*

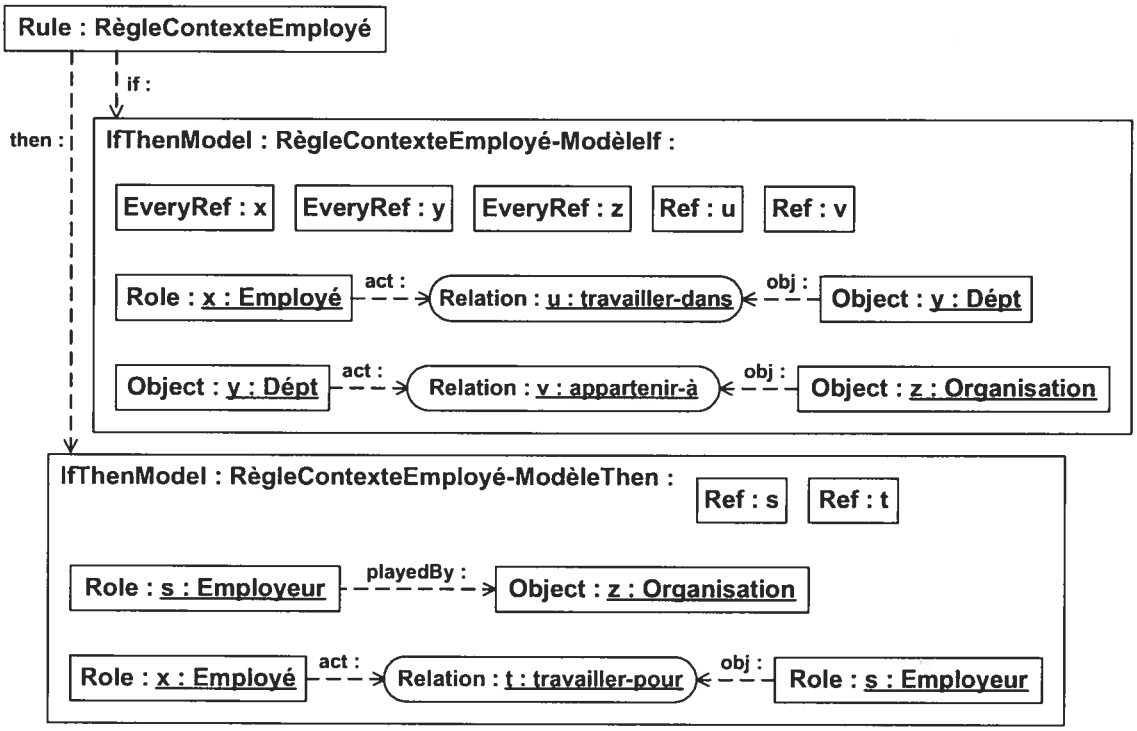


Figure 189 : Contrainte de contexte

Le modèle `RègleContexteEmployé-ModèleIf` *représente la partie de pré-conditions de la règle. Il exprime l'expression (i) comme suit : pour tout rôle employé* `x`*, pour tout département* `y`*, et pour toute organisation* `z` *de telle sorte que ces conditions soient satisfaites:*

- *(i1) le rôle employé* `x` *travaille dans (*`travailler-dans`*) le département* `y`*. Ceci est indiqué par une relation (désignée par une variable soit* `u`*) de type* `travailler-dans` *entre* `x` *et* `y`*.*
- *(i2) le département* `y` *appartient à (*`appartenir-à`*) l'organisation* `z`*. Ceci est indiqué*

par une relation (désignée par une variable soit v*) de type* appartenir-à *entre* y *et* z.

Le modèle RègleContexteEmployé-ModèleThen *représente la partie de post-conditions de la règle. Il exprime l'expression (ii) par les conditions suivantes:*

- *(ii1) il existe* s *un rôle employeur joué par l'organisation* z. *Ceci est indiqué par un arc de type* playedBy *de* s *à* z.
- *(ii2) le rôle employé* x *travaille pour (*travailler-pour*) le rôle employeur* s. *Ceci est indiqué par une relation (désignée par une variable soit* t*) de type* travailler-pour *entre* x *et* s.

6.2.2.12 Contrainte de migration

Toute contrainte de migration est liée à l'évolution des données et au mécanisme permettant de faire migrer dynamiquement un élément instancié directement d'un type vers des types plus représentatifs/pertinents, tout en respectant la définition de ces derniers.

Une contrainte de migration est donc exprimée par les modèles de définition des types impliqués. Par exemple, le modèle illustré à la Figure 170 (page 167) ou bien à la Figure 171 (page 167) (cf. l'exemple 21, page 166) exprime la définition du type FournisseurTeximus-TémoinAvocatPapin-Client. Ce type est un sous-type de Client. Il regroupe tous les rôles *client* dont chacun doit participer à au moins une relation instanciée de SignerContrats où FournisseurTeximus est le seul rôle *fournisseur* et où TémoinAvocatPapin est le seul rôle *témoin* (cf. la situation 1 du Problème 2, page 37).

De plus, le pouvoir de représenter les cycles d'états pour les types d'objets (ou de rôles) permet d'exprimer l'ordre de l'évolution. Par exemple, un adolescent (Adolescent) devient adulte (Adulte) et non l'inverse (ce qui est indiqué par un arc de type prevOf de Adolescent à Adulte, tel qu'illustré à la Figure 150 (page 144) de l'exemple 19 (page 143)).

Également, des critères s'appliquant au changement d'états d'un type peuvent être exprimés. L'exemple 35 présente un exemple du cas.

Contrainte de migration – critère de transition entre les types (exemple 35)

*Soit le critère suivant: «une personne adolescente (*Adolescent*) devient adulte (*Adulte*) à l'âge de 18» ou bien «toute personne passe du groupe d'adolescents au groupe d'adultes lorsque qu'elle atteint l'âge de 18 ans». Ceci peut être illustré tel que dans la Figure 190. Le*

passage d'une instance du type dynamique `Adolescent` *vers* `Adulte` *est activé suivant la règle* `RègleAdolescentVersAdulte` *(ce qui est indiqué par un arc de type* `fulfil` *qui associe à* `RègleAdolescentVersAdulte` *l'arc de type* `prevOf` *de* `Adolescent` *à* `Adulte`*). La règle* `RègleAdolescentVersAdulte` *spécifie que : (i1) pour toute personne (désignée par* `x`*), si (i2) elle est du groupe d'adolescents et (i3) son âge (désignée par* `y`*) atteint 18, alors (ii) elle passe au groupe d'adultes.*

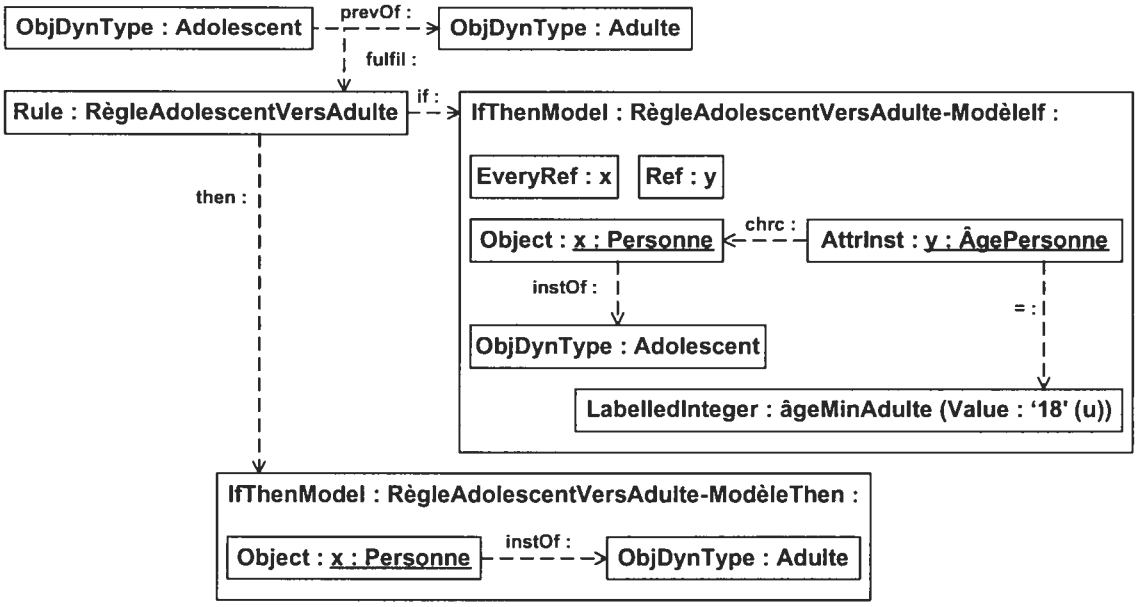


Figure 190 : Contrainte de migration – critère de transition entre les types

La partie de pré-conditions de la règle `RègleAdolescentVersAdulte` *est contextualisée par le modèle* `RègleAdolescentVersAdulte-ModèleIf`*. Dans ce modèle: l'expression (i1) est interprétée par les formules «*`EveryRef : x`*» et «*`Object : x : Personne`*»; l'expression (i2) par un arc de type* `instOf` *de* `x` *à* `Adolescent`*; l'expression (i3) par les formules «*`Ref : y`*» et «*`AttrInst : y : ÂgePersonne`*» et par un arc de type* `=` *de* `y` *à* `âgeMinAdulte`*; et* `âgeMinAdulte` *représente un entier étiqueté qui désigne l'âge minimal pour un adulte et prend la valeur 18.*

Le modèle `RègleAdolescentVersAdulte-ModèleThen` *contextualise la partie de post-conditions de la règle, et interprète l'expression (ii) par un arc de type* `instOf` *de* `x` *à* `Adulte`*. Évidemment, une fois que la personne* `x` *est vue comme une instance de* `Adulte`*, elle n'appartient plus au type* `Adolescent`*.*

6.2.2.13 Contrainte de couverture

Une contrainte de couverture exprime le fait que les sous-types représentent toutes les

alternatives pour le type parent dans le sens que l'ensemble des instances du type parent est l'union des ensembles des instances de ses sous-types. Par exemple, un rôle *employé* doit être un rôle «employé temporaire» ou «employé permanent».

Notre formalisme permet d'exprimer une contrainte de couverture sous forme d'une règle «*Si ... alors ...*» dont la partie pré-condition (ou post-condition) peut être la combinaison de type «ou» logique des modèles de conditions. Voir l'exemple 36. Concernant les cycles d'états pour les types d'objets (ou de rôles) en particulier, les sous-types dynamiques d'un type parent qui forment un cycle d'états pour ce type parent peuvent donc être vus comme toutes les alternatives pour ce dernier. Alors le pouvoir de représenter des cycles d'états pour les types d'objets (ou de rôles) avec notre formalisme représente également une possibilité pour exprimer des contraintes de couverture. Voir la section 9 (page iii-34, Annexe III) pour plus de détails et d'illustrations.

Contrainte de couverture (exemple 36)

Soit la contrainte de couverture suivante : un rôle employé doit être un rôle «employé temporaire» ou «employé permanent». Cette contrainte peut être représenté par une règle telle que `RègleCouvertureEmployé` *dans la Figure 191, avec l'interprétation suivante: (i) pour tout* `x` *un rôle employé (*`Employé`*), (ii)* `x` *doit être un rôle «employé temporaire» (*`TempEmp`*) ou (iii)* `x` *doit être un rôle «employé permanent» (*`PermEmp`*).*

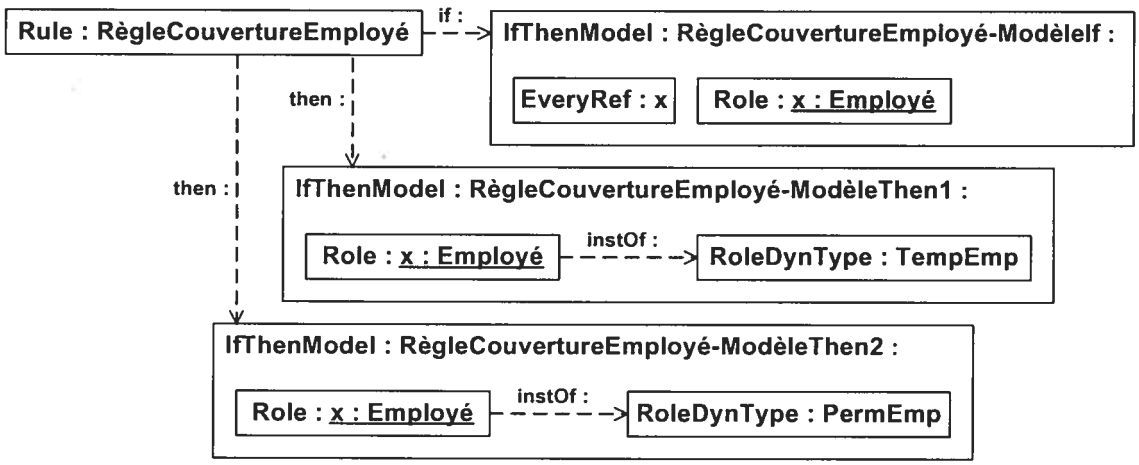


Figure 191 : Contrainte de couverture

L'élément `RègleCouvertureEmployé` *(représentant la règle en question) est donc associé par un arc de type* `if` *à* `RègleCouvertureEmployé-ModèleIf`*, un modèle de conditions contextualisant l'expression (i). Cet élément est aussi associé par des arcs de type* `then` *à* `RègleAdmisMScInf-ModèleThen1` *et* `RègleAdmisMScInf-ModèleThen2`*,*

deux modèles de conditions contextualisant respectivement les expressions (ii) et (iii). Nous expliquons ci-dessous la mise en œuvre du contenu des modèles `RègleAdmisMScInf-ModèleIf`, `RègleAdmisMScInf-ModèleThen1` *et* `RègleAdmisMScInf-ModèleThen2`.

Le modèle `RègleCouvertureEmployé-ModèleIf` *contextualise la partie de pré-conditions de la règle* `RègleCouvertureEmployé` *et exprime l'expression (i) par les formules «*`EveryRef : x`*» et «*`Role : x : Employé`*». `RègleCouvertureEmployé` est donc associé à ce modèle par un arc de type* `if`.

Le modèle `RègleCouvertureEmployé-ModèleThen1` *exprime l'expression (ii) par un arc de type* `instOf` *de* `x` *à* `TempEmp`. *Et le modèle* `RègleCouvertureEmployé-ModèleThen2` *exprime l'expression (iii) par un arc de type* `instOf` *de* `x` *à* `PermEmp`. *Ces deux modèles représentent, dans la règle* `RègleCouvertureEmployé`, *les modèles de post-conditions alternatifs.* `RègleCouvertureEmployé` *est donc associé à ces deux modèles par des arcs de type* `then`..

6.2.2.14 Contrainte de disjonction

Une contrainte de disjonction entre les types exprime le fait que ces types sont disjoints, en ce sens qu'aucun élément n'est à la fois une instance de deux parmi ces types. La contrainte peut être représentée sous forme d'une règle. L'exemple 37 présente une illustration. En particulier, les sous-types d'un type parent qui forment un cycle d'états pour ce type parent sont considérés comme des sous-types dynamiques, disjoints et complets du type parent. Donc le pouvoir de représenter des cycles d'états pour des types d'objets (ou de rôles) signifie également un moyen dans notre formalisme pour exprimer des contraintes de disjonction. Voir la section 9 (page iii-34, Annexe III) pour plus de détails.

Contrainte de disjonction (exemple 37)

Soit la contrainte de disjonction suivante: le type de rôles «employé temporaire» et celui de «employé permanent» sont disjoints. Cette contrainte peut être exprimée par la règle `RègleDisjonctionEmployé`, *illustrée à la Figure 192. Cette règle se traduit comme suit: (i1) pour tout* `x` *un rôle «employé temporaire» (*`TempEmp`*) et (i2) pour tout* `y` *un rôle «employé permanent» (*`PermEmp`*), (ii)* `x` *et* `y` *ne doivent pas représenter le même élément.*

Le modèle `RègleDisjonctionEmployé-ModèleIf` *contextualise la partie de pré-conditions de la règle* `RègleDisjonctionEmployé` *(ce qui est indiqué par un arc de type* `if`*). Ce modèle exprime l'expression (i1) par les formules «*`EveryRef : x`*» et «*`Role : x :`

TempEmp», et l'expression (i2) par les formules «`EveryRef : y`*» et «*`Role : y : PermEmp`*».*

Le modèle `RègleCouvertureEmployé-ModèleThen` *contextualise la partie de post-conditions de la règle* `RègleDisjonctionEmployé` *(ce qui est indiqué par un arc de type* `then`*). Il représente l'expression (ii) par un arc de type* `!=` *de* `x` *à y.*

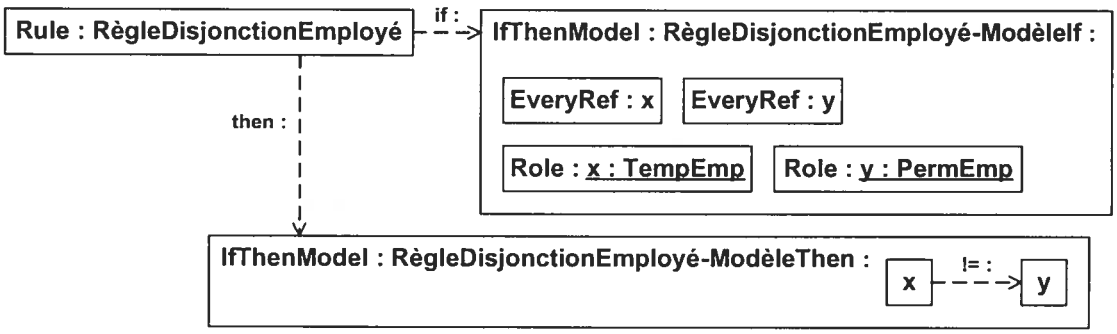


Figure 192 : Contrainte de disjonction

6.2.2.15 Contrainte d'implication d'informations négatives

Une contrainte d'implication d'informations négatives est liée à la représentation de phrases sous forme négative. Sa représentation avec notre formalisme se fait par l'utilisation de l'opérateur `not`. Un exemple du cas est présenté dans l'exemple 38.

Contrainte d'implication d'informations négatives (exemple 38)

Représentons par exemple la contrainte suivante: aucune salle de réunion ne se situe au 9ème étage. Soient les suppositions suivantes: `SalleRéunion` *et* `Étage` *sont deux types statiques d'objets qui représentent respectivement l'ensemble de salles de réunion et celui des étages. La structure illustrée à la Figure 193-(1) spécifie qu'une salle de réunion (*`SalleRéunion`*) se situe à (*`se-situer-à`*) un étage (*`Étage`*). La Figure 193-(2) spécifie que le type de relations* `ne-pas-se-situer-à` *exprime la négation de* `se-situer-à`*.*

La contrainte est équivalente à la règle «Toute salle de réunion ne se situe pas au 9ème étage». Elle peut être représentée par la règle `RègleSalleRéunion`*, illustrée à la Figure 193-(3) avec cette interprétation: (i) pour tout* `x` *une salle de réunion, (ii)* `x` *ne se situe pas au 9ème étage. Dans cette règle, le modèle de pré-conditions* `RègleSalleRéunion-ModèleIf` *exprime l'expression (i) par les formules «*`EveryRef: x`*» et «*`Object : x : SalleRéunion`*». Soit* `9èÉtage` *un objet représentant le 9ème étage. Le modèle de post-conditions* `RègleSalleRéunion-ModèleThen` *représente l'expression (ii) par le fait qu'il existe une relation (désignée par une variable soit* `y`*) de type* `ne-pas-se-situer-à` *entre* `x` *et* `9èÉtage`*.*

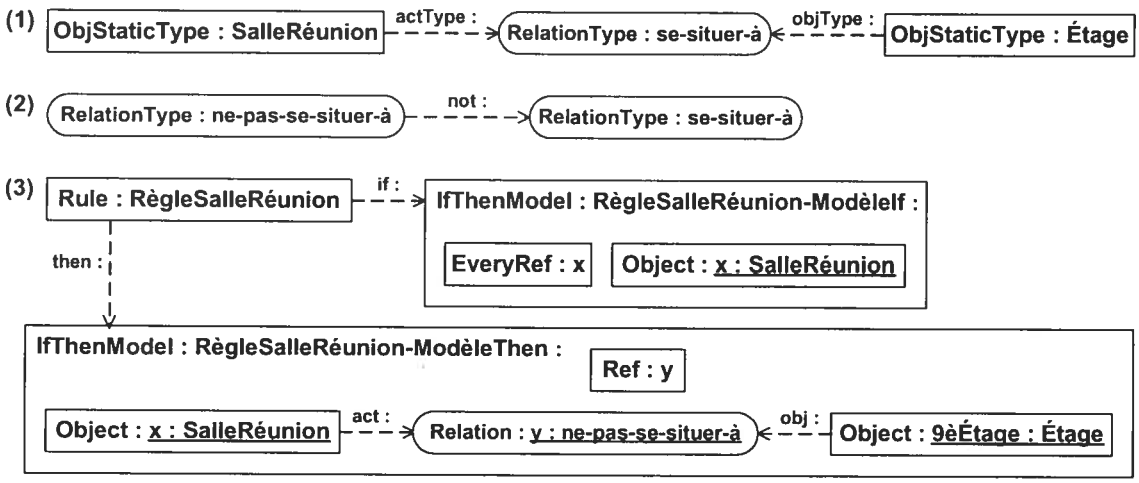


Figure 193 : Contrainte d'implication d'informations négatives

6.3 Synthèse

L'évaluation de notre formalisme, présentée dans ce chapitre, a démontré sa flexibilité et sa puissance dans la représentation de modèles. Le méta-métamodèle de notre formalisme se définit lui-même, il est extensible et remplit entièrement les besoins formulés pour un M3. Il permet donc de définir tous les types de métamodèles au niveau M2. Au moyen de ce méta-métamodèle, notre métamodèle est défini et remplit tous les besoins nécessaires pour la représentation de modèles de connaissances du monde réel.

Du côté des fonctionnalités concernant la représentation de modèles, notre formalisme dispose d'autres supports intéressants. Par exemple, il permet le multilinguisme dans la représentation des interprétations des éléments. Il intègre la représentation de règles sémantiques contraignant des éléments modélisés. Il prend en compte aussi la représentation des utilisateurs et de leurs interactions avec les objets de connaissances, ce qui joue un rôle important pour la gestion administrative de la base de modèles.

Nous présentons ci-dessous deux tableaux regroupant les évaluations des formalismes étudiés et de notre formalisme en réponse aux besoins essentiels pour la gestion de modèles.

BesoinM3 1 : Typage entre éléments de deux niveaux de modélisation consécutifs

BesoinM3 2 : Passage entre niveaux types et instances

BesoinM3 3 : Distinction entre différentes notions de conformité au niveau méta

BesoinM3 4 : Hiérarchisations entre types

BesoinM3 5 : Contraintes de cardinalité min/max pour un méta-lien

BesoinM3 6 : Distinction entre modèles de différentes natures

BesoinM3 7 : Liens avec les modèles

Noyaux réflexifs (au M3)	**Besoins pour M3 (*BesoinM3*)**						
	1	2	3	4	5	6	7
Notre M3	+	+	+	+	+	+	+
Réseaux sémantiques (RSs)	-	-	-	-	-	-	-
sNets	+-	+	+	+-	+-	+-	+-
Graphes conceptuels (GCs)	+-	+	-	+	-	+-	+-
Modèle uniforme des GCs	+-	+	-	+	+	+-	+-
CDIF	+-	+	-	+	+	-	+-
MOF	+-	+	-	+	+	-	+-
XML-XML Schema	+-	+	-	+	-	+-	+-
RDF-RDFS	+-	+	-	+	-	+-	+-
OWL	+-	+	-	+	+	+-	+-

Notations:

«+» : remplir le besoin correspondant;

«+-»: remplir partiellement le besoin correspondant et à développer pour le remplir;

«-» : ne pas remplir et à développer pour remplir le besoin correspondant

Table 3: Notre M3 versus les M3 étudiés pour représenter et gérer les métamodèles

BesoinM2 1 : Typage entre éléments du niveau M1

BesoinM2 2 : Typage entre éléments de deux niveaux de modélisation consécutifs

BesoinM2 3 : Passage entre niveaux types et instances

BesoinM2 4 : Distinction entre différentes notions de conformité

BesoinM2 5 : Hiérarchisations entre types

BesoinM2 6 : Multi-classification et Connaissance partielle

BesoinM2 7 : Objets, Rôles, Types d'objets, Types de rôles

BesoinM2 8 : Rapports entre objets et rôles / types de rôles

BesoinM2 9 : Classification dynamique et classification statique

BesoinM2 10 : Contraintes de l'arité min/max

BesoinM2 11 : Contraintes de cardinalité totale min/max pour un type relationnel

BesoinM2 12 : Contraintes de cardinalité locale min/max pour un type relationnel

BesoinM2 13 : Contraintes de l'itération min/max pour un type relationnel

BesoinM2 14 : Contraintes de cardinalité min/max pour un méta-lien

BesoinM2 15 : Types relationnels entre types et/ou instances

BesoinM2 16 : Distinction entre modèles de différentes natures

BesoinM2 17 : Liens avec les modèles

BesoinM2 18 : Liens entre modèles au niveau M1

Métamodèle au M2	**Besoins pour M2 (*BesoinM2*)**																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Notre M2	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RSs	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
sNets	-	+-	+	-	-	-	-	-	-	-	-	-	-	+-	-	-	-	-
GCs	+	+-	+	-	+	+	+-	+-	-	-	-	-	-	-	+-	+-	+-	-
Modèle uniforme des GCs	+	+-	+	-	+	+	-	-	-	-	-	+	-	+	+-	+-	+-	-
Entité-Association	+-	+-	-	-	+-	-	+-	+-	-	-	-	+	-	+	-	-	-	-
UML	+	+-	+	-	+	+	+	+	-	+-	-	+	-	+	+-	+-	+-	+
XML-XML Schema	+-	+-	+	-	+	-	-	-	-	+-	-	-	-	-	+-	+-	+-	+-
RDF-RDFS	+-	+-	+	-	+	+	-	-	-	-	-	-	-	-	+-	+-	+-	+-
OWL	+-	+-	+	-	+	+	-	-	-	-	-	+	-	+	+-	+-	+-	+-

Notations:

«+» : remplir le besoin correspondant;

«+-» : remplir partiellement le besoin correspondant et à développer pour le remplir;

«-» : ne pas remplir et à développer pour remplir le besoin correspondant

Table 4: Notre M2 versus les M2 étudiés pour représenter et gérer les modèles au niveau M1

Chapitre 7. Mise en œuvre du cadre de modélisation

Nous traitons, dans ce chapitre, les deux problèmes suivants: (i) le stockage de modèles et (ii) la spécification d'opérations de manipulation de modèles, qui se sont révélés comme des problèmes fondamentaux de la mise en œuvre de notre formalisme pour un système de gestion de modèles. Nous présentons également notre prototype en cours de développement pour le système mentionné.

7.1 Stockage de modèles

Notre solution au problème de stockage de modèle consiste à emmagasiner les modèles de représentation de connaissances au sein d'une base de données relationnelles. Ce choix ne constitue qu'un essai de solution qui se veut la plus simple possible.

Forme Tbar

La possibilité de coder uniformément les éléments dans un formalisme peut simplifier la mise en œuvre de ce dernier. Donc nous avons choisi de mettre en œuvre les éléments (c'est-à-dire les nœuds et arcs) de notre formalisme de la même manière. Pour ce faire, nous avons introduit la notion de Tbar. Un Tbar permet de coder un nœud ou un arc. Un Tbar met en relation trois éléments et peut être noté comme un 3-uplet de forme (*source*, *metaArc*, *destination*), indiquant que l'élément *source* est associé à l'élément *destination* par un arc de type *metaArc*. La Figure 194 illustre la notation et la représentation des nœuds et des arcs en Tbar.

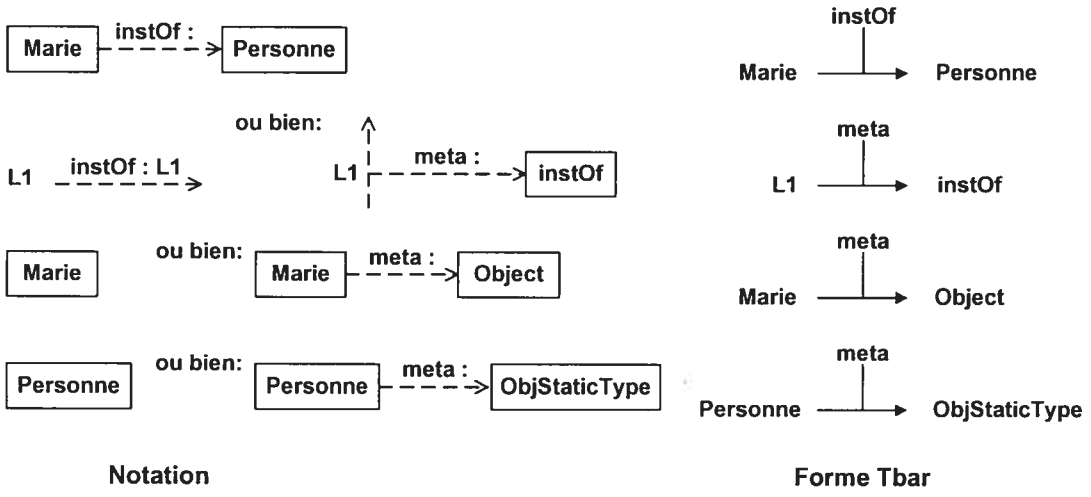


Figure 194 : Exemple 1 de nœuds et arcs en Tbar

Sur la première ligne de cette figure, nous avons à gauche la notation pour l'énoncé «Marie est une personne» et à droite la mise en œuvre sous la forme de Tbar: (`Marie`, `instOf`, `Personne`). Sur la deuxième ligne, il s'agit de la représentation de l'arc `L1` entre `Marie` et `Personne`. La mise en œuvre sous la forme de Tbar, (`L1`, `meta`, `instOf`), montre que `L1` est de nature `instOf`. Les troisième et quatrième lignes présentent respectivement la mise en œuvre des deux nœuds `Marie` et `Personne` de niveau M1. `Marie` est de nature objet: (`Marie`, `meta`, `Object`). `Personne` est de nature type statique d'objets: (`Personne`, `meta`, `ObjStaticType`).

Chaque élément aux extrémités d'un Tbar peut lui-même être représenté par un Tbar et ainsi de suite. Explorons par exemple, l'extrémité `instOf` des Tbar de la Figure 194. La première ligne à la Figure 195 présente que `instOf` est un nœud défini au niveau M2 et est de nature type d'arcs: (`instOf`, `Mmeta`, `MMetaArc`). La deuxième ligne présente que `MMetaArc` est de nature `MmetaNode`. La dernière ligne indique que `MMetaNode` est de nature `MMetaNode`, autrement dit que `MMetaNode` se conforme à lui-même. Le pouvoir de représenter un élément à une extrémité d'un Tbar par un Tbar nous permet ainsi de passer d'un élément à son méta-élément. Ce passage s'arrête lorsque l'élément réflexif (c'est-à-dire `MMetaNode` dans ce cas) est atteint.

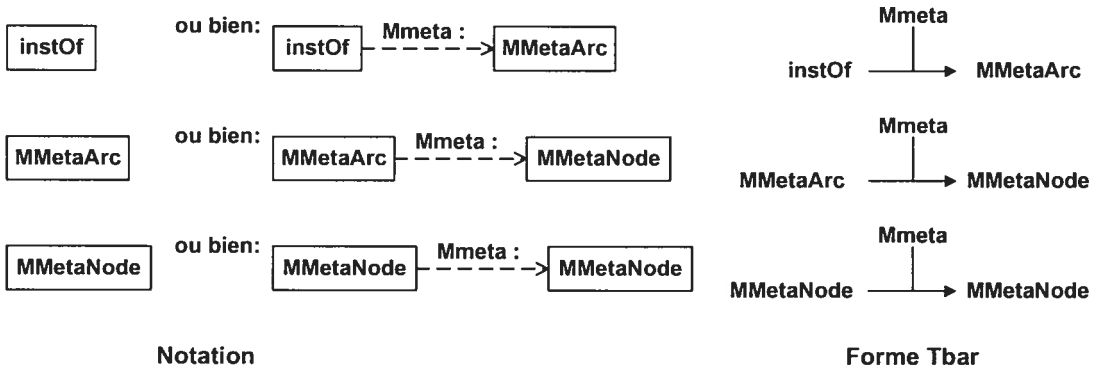


Figure 195 : Exemple 2 de nœuds et arcs en Tbar

Stockage de modèles au sein d'une base de données relationnelles

Nous venons de montrer comment les éléments (nœuds et arcs) dans notre formalisme peuvent être codés de la même manière par des Tbar. Par ailleurs, un élément peut être nommé et identifié par son libellé. Les éléments qui peuvent être libellés peuvent être entreposés dans une même table appelée *Tbar*, telle qu'illustrée à la Figure 196. Pour faciliter la compréhension, nous donnons, entre parenthèses, le label correspondant à chaque référence.

idTbar	TbarLabel	srce	metaArc	Dest
1	MMetaNode	null	88(Mmeta)	1(MMetaNode)
2	MMetaArc	null	88(Mmeta)	1(MMetaNode)
3	MMetaArc	null	88(Mmeta)	1(MMetaNode)
...	...	...	...	...
23	NotreM3	null	88(Mmeta)	1(MMetaNode)
...	...	...	...	...
88	Mmeta	null	88(Mmeta)	2(MMetaArc)
...	...	...	...	...
612	NotreM2	null	88(Mmeta)	12(MMetaModel)
613		612(NotreM2)	105(Msem)	42(NotreM3)
...	...	...	...	...
643	ObjStaticType	null	88(Mmeta)	640(ObjType)
		643(ObjStaticType)	76(MsubType)	640(ObjType)
...	...	...	...	...
794	Structure-instOf	null	88(Mmeta)	11(Mstructure)
795	instOf	null	88(Mmeta)	2(MMetaArc)
796		795(instOf)	125(MdefIn)	612(NotreM2)
797		795(instOf)	140(MdefAs)	794(Structure-instOf)
798		690(Object)	85(Msrce)	795(instOf)
799		640(ObjType)	86(Mdest)	795(instOf)
800		798(null)	125(MdefIn)	794(Structure-instOf)
801		799(null)	125(MdefIn)	794(Structure-instOf)
...	...	...	...	...
1897	ModèleM1	null	744(meta)	634(Model)
1898		1897(ModèleM1)	758(sem)	612(NotreM2)
1899	Nom	null	744(meta)	655(Attribute)
1900		1899(Nom)	1176(defIn)	1897(ModèleM1)
1901	Personne	null	744(meta)	643(ObjStaticType)
1902		1901(Personne)	1176(defIn)	1897(ModèleM1)
1903		1899(Nom)	995(chrcType)	1901(Personne)
...	...	...	...	...
2018	Marie	null	744(meta)	690(Object)
2019		2018(Marie)	1176(defIn)	1897(ModèleM1)
2020	L1	2018(Marie)	795(insOf)	1901(Personne)
2021	Nom-Marie	null	744(meta)	685(AttrInst)
2022		2021(Nom-Marie)	1176(defIn)	1897(ModèleM1)
2023		2021(Nom-Marie)	795(insOf)	1899(Nom)
2024		2021(Nom-Marie)	1134(chrc)	2018(Marie)
...	...	...	...	...

Figure 196 : Table *Tbar* pour le codage d'éléments libellés

Dans la table *Tbar*, chaque ligne indique qu'un élément (e) est enregistré comme un 5-uplet de forme (*idTbar, TbarLabel, source*, *metaArc*, *destination*) dans lequel: la relation entre *source*, *metaArc* et *destination* signifie le Tbar (*source*, *metaArc*, *destination*) correspondant à l'élément e; *TbarLabel* désigne le libellé de ce Tbar ou bien de l'élément e; et *idTbar* désigne le numéro de référence attribué par le système à ce Tbar ou bien à cet élément e. Ce principe permet de coder dans la même table *Tbar* les modèles répartis à tous les niveaux de modélisation. Un exemple de codage des éléments de niveau M3 est montré dans la Figure 196. La ligne 1 de la table *Tbar* déclare que 1 est la référence utilisée par le système pour l'élément libellé MMetaNode qui désigne l'élément réflexif de notre M3, attaché à lui-même par un arc instancié du type de référence 88 (Mmeta); la ligne 2 montre que l'élément de référence 1 (MMetaNode) est attaché à l'élément de référence 23 (NotreM3) par un arc instancié du type de référence 125 (MdefIn), indiquant que l'élément libellé MMetaNode est défini dans le méta-métamodèle NotreM3; etc. Un exemple de codage des éléments de niveau M2 est églament présenté dans la Figure 196. La ligne 612 déclare que l'élément de référence 612 (NotreM2) désignant un métamodèle est attaché à l'élément de référence 12 (MMetaModel) par un arc instancié du type de référence 88 (Mmeta); la ligne 613 montre que cet élément (NotreM2) est attaché à l'élément de référence 42 (NotreM3) par un arc instancié du type de référence 105 (Msem), exprimant que ce métamodèle est conforme au méta-métamodèle de NotreM3. Les lignes 795 et 796 indiquent que l'élément de référence 795 (instOf) est un méta-arc et est défini dans le métamodèle NotreM2. La ligne 797 indique que l'élément de référence 794 (Structure-instOf) représente une structure de instOf. Les lignes 798, 799, 800 et 801 présentent que dans la structure Structure-instOf, les éléments Objet et ObjType sont respectivement associés à instOf par un arc de type Msrce et un autre de type Mdest. La Figure 196 (page 196) illustre aussi le codage des éléments de niveau M1. Les lignes 1897 et 1898 déclarent que l'élément de référence 1897 (ModèleM1) désigne un modèle conforme à son métamodèle NotreM2. Les lignes 1901, 1902 et 1903 indiquent que l'élément de référence 1901 (Personne), désignant l'ensemble des personnes, est de nature type statique d'objets (ObjStaticType), qu'il est défini dans le modèle ModèleM1 et caractérisé par l'attribut Nom. Et les lignes de 2018 à 2024 codent que l'élément de référence 2018 (Marie) est de nature objet (Object), qu'il est défini dans ModèleM1,

qu'il représente une personne dont le nom est désigné par `Nom-Marie`, une instance d'attribut. La Figure 197 (page198) illustre en notation graphique (par des nœuds et arcs) une partie du contenu de la Figure 196 (page 196).

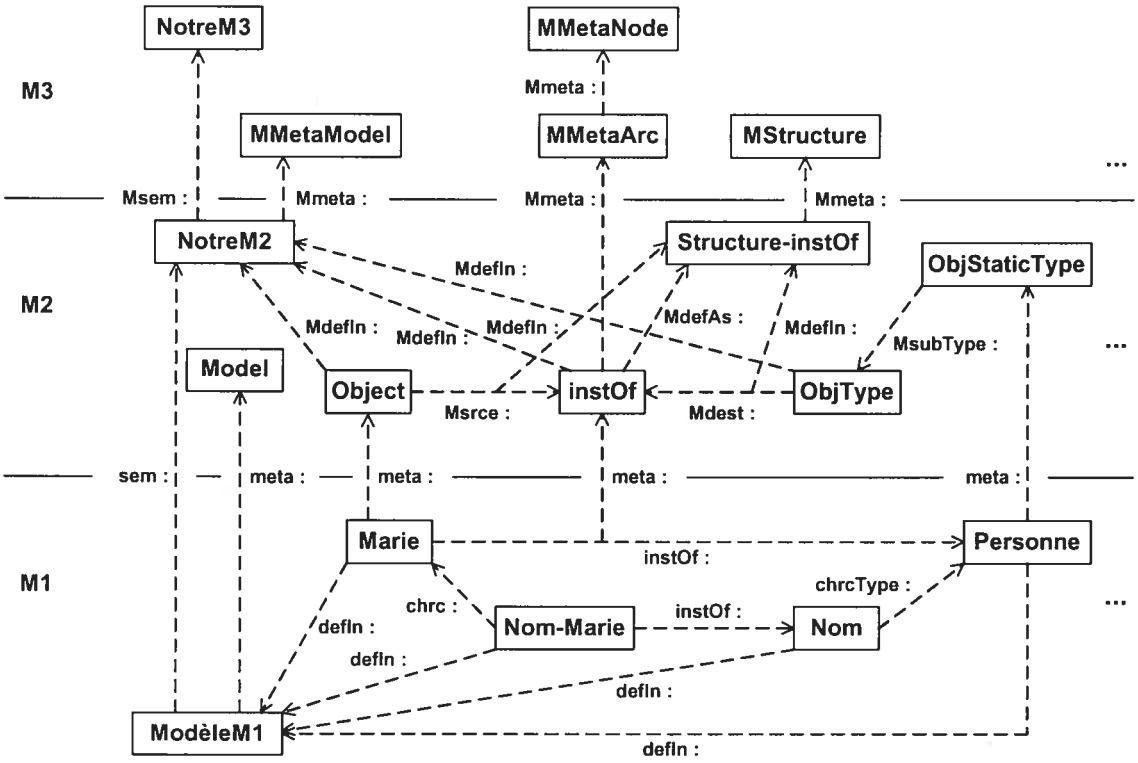


Figure 197 : Illustration graphique d'une partie de la Figure 196

Les éléments de nature valeurs (vus comme valeurs à représenter les interprétations des éléments libellés) sont traités différemment des éléments libellés (vus pour les objets), tel qu'illustré dans la section 2.1.4 (page 13): les objets peuvent changer ou être transformés en fonction du temps alors que les valeurs restent inchangées. Donc, les valeurs d'éléments libellés peuvent être stockées dans une table appelée *Value*. Dans cette table, chaque ligne est enregistrée comme un 4-uplet de forme (*idValue, idTbar, lang, value*) dans lequel: *value* désigne l'interprétation en langue *lang* de l'élément de référence *idTbar* (stocké dans la table *Tbar*), et *idValue* désigne la référence utilisée par le système pour cette *value* dans la table *Value*. La Figure 198 illustre cette table *Value*. Dans cette figure, la ligne 1 indique que l'élément `MMetaNode` (de référence 1 dans la table *Tbar*) est interprété comme «`MMetaNode`» en langue universelle (u). Les lignes 830 et 831 spécifient que l'élément `Personne` (de référence 1901 dans la table *Tbar*) est interprété comme «personne» en français (fr) et comme «person» en anglais (en). La

ligne 832 spécifie que l'élément `Nom-Marie` (de référence 2021 dans la table *Tbar*, et désignant le nom de `Marie`) est interprété comme «Levesque» dans toutes les langues (langue universelle u).

idValue	idTbar	lang	Value
1	1(MMetaNode)	u	MMetaNode
...	...	...	...
830	1901(Personne)	fr	personne
831	1901(Personne)	en	person
832	2021(Nom-Marie)	u	Levesque
...	...	...	...

Figure 198 : Table *Value* pour le stockage de valeurs d'éléments

Ainsi, la base de données peut comporter seulement deux tables, *Tbar* et *Value*, pour le codage des éléments libellés et pour le stockage de valeurs d'éléments libellés comme illustrées dans les Figure 196 (page 196) et Figure 198 (page 199).

7.2 Opérations sur les modèles

La manipulation des modèles est une facette aussi importante que l'expression quand vient le temps de définir des formalismes de modélisation. Il est donc important de montrer comment notre système de modélisation/métamodélisation se prête à réaliser les opérations importantes pour la manipulation des modèles répartis à tous les niveaux de modélisation.

Les opérations de manipulation de modèles peuvent se classifier en deux groupes: les opérations de base et les opérations avancées.

Les *opérations de base* concernent les fonctionnalités élémentaires pour la gestion de modèles. Nous pouvons compter les opérations de base suivantes (cf. la section 2.3.1, page 17, et la section 3.3, page 42):

- Les opérations indispensables pour l'acquisition de modèles: la création, la suppression, la modification (mise-à-jour, simplification), et la copie de modèles;
- Les opérations aidant à réduire le besoin de parcourir les éléments du modèle, lors du traitement de ce dernier: l'énumération de modèles, l'application de fonction;
- Les opérations permettant de déduire de nouveaux modèles à partir d'un modèle par le mécanisme de sous-typage: la restriction de types, la relaxation de types relationnels;

– Les opérations ensemblistes: la jointure, l'intersection, et la différence entre les modèles.

Les *opérations avancées* touchent les aspects plus complexes dans le traitement de modèles, tels que présentés aux sections 2.3.2 (page 20) et 3.3 (page 42): l'instanciation de modèles, la dérivation de modèles, la spécialisation entre les modèles, la contraction et l'expansion, la requête, la correspondance entre les modèles, la différence complexe entre les modèles, la fusion de modèles, la transformation et la migration de modèles.

Dans la suite de ce chapitre, nous allons décrire ces opérations de base listées ci-haut ainsi que certaines de ces opérations avancées qui touchent nos besoins, et montrer leur mise en œuvre avec notre système de modélisation/métamodélisation.

7.2.1 Opérations de base

Création de modèles

L'opération `CréationModèle(M)` vise à créer le modèle *M* qui est valide dans la base de modèles. Chaque élément contient sa sémantique et devient valide s'il respecte les contraintes qui lui sont rattachées. Alors la création d'un modèle peut impliquer la création d'autres éléments et nécessite automatiquement une étape de validation du modèle. Un modèle est valide si tous les éléments concernés sont valides. La base des connaissances ne contient que des éléments valides.

Le comportement de l'opération de création de modèles, `CréationModèle(M)`, peut donc être interprété comme suit, en utilisant d'autres opérations de base.

```
Opération CréationModèle(M){
  // Si M est un modèle valide, M est inséré dans la base de connaissances
  si estModèleValide(M) alors retourne InsertionModèle(M);
}
Opération estModèleValide(M){
  // Retourne Oui si tous les éléments de M sont valides, et retourne Non sinon
  pour chaque nœud, arc e de M faire
     si négation(estÉlémentValide(e)) alors retourne Non;
  retourne Oui;
}
Opération estÉlémentValide(e){
   /* Retourne Oui si l'élément e se conforme à son méta-élément et si e respecte les
```

contraintes concernées, sinon retourne *Non* */

}

Opération InsertionModèle(*M*) {

/* Insérer le modèle *M* dans la base de connaissances. Seulement de nouveaux éléments (nœuds, arcs) seront insérés. Puis retourner le modèle *M* */

}

L'insertion d'éléments à la base de connaissances se font par d'autres opérations de base, par exemple:

- InsertionÉlément(*e*) : cette opération vise à ajouter à la base de connaissances un nouvel élément valide *e*.
- InsertionÉléments(*E*): cette opération vise à ajouter à la base de connaissances les nouveaux éléments valides de l'ensemble *E*. L'opération InsertionÉlément(*e*) est effectuée pour chaque élément *e* de *E*.

Suppression de modèles

L'opération SuppressionModèle(*M*) consiste à supprimer le modèle *M* dans la base des connaissances. Dans un modèle, il peut exister des éléments définis dans ce modèle ainsi que des éléments réutilisés de d'autres modèles. Alors, lorsqu'un un modèle (*M*) est à supprimer, l'élément *M* (représentant le modèle *M*) et les arcs de cet élément sont à supprimer), ainsi que les éléments (nœuds, arcs) définis dans le modèle *M* et leurs arcs sont également à supprimer. La suppression d'un élément peut impliquer l'invalidité de d'autres éléments. Les éléments invalides peuvent aussi être à supprimer.

Le comportement de SuppressionModèle(*M*) peut donc être interprété comme suit:

Opération SuppressionModèle(*M*) {

// Suppression l'ensemble *E* des éléments contenus dans *M*

E := TousElements(*M*) ; // {*e* | *e* est un nœud ou arc contenu dans M_1}

varM := SuppressionÉlémentsDansModèle(*E*,*M*) ;

// Suppression de l'élément *M* dans la base de connaissances

retourne SuppressionÉlément(*M*) ; // SuppressionÉlément(*varM*)

}

Voici la spécification de certaines opérations de base appelées lors de l'exécution de l'opération de suppression de modèles:

- SuppressionÉlément(*e*) : cette opération vise à supprimer l'élément *e* dans la

base de connaissances. Lorsqu'un un élément (`e`) est supprimé, les arcs de cet élément sont aussi supprimés. La suppression d'un élément peut impliquer l'invalidité d'autres éléments. Les éléments invalides peuvent être à supprimer.

- `SuppressionÉléments(E)` : cette opération vise à supprimer les éléments de l'ensemble *E* présents dans la base de connaissances. `SuppressionÉlément(e)` est appliquée pour chaque élément `e` de *E*.
- `SuppressionÉlémentDansModèle(e,M)` : cette opération consiste à retourner le modèle *M* après la suppression de l'élément `e` dans *M* dans la base de connaissances. Il existe deux scénarios que voici:
  a) Si `e` est défini dans le modèle *M*, l'opération `SuppressionÉlément(e)` est appelée pour supprimer l'élément `e` dans la base de connaissances.
  b) Si `e` est défini dans un autre modèle que *M*, les groupes d'éléments suivants sont à supprimer: (i) les arcs de `e` définis dans *M*, (ii) les arcs entre `e` et *M*, et (iii) les éléments devenus invalides par la suppression des éléments dans les groupes (i) et (ii).
- `SuppressionÉlémentsDansModèle(E,M)` : cette opération consiste à retourner le modèle *M* dans la base de connaissances après avoir supprimé les éléments de l'ensemble *E* présents dans *M*. `SuppressionÉlémentDansModèle(e,M)` s'effectue pour chaque élément `e` de *E*.

Mise-à-jour des modèles

Quelques opérations de base concernent la mise-à-jour des modèles dans la base de connaissances. Certaines de ces opérations ont été spécifiées ci-haut.

- Ajout d'éléments à la base de connaissances: `InsertionÉlément(e)`, `InsertionÉléments(E)`.
- `AjoutÉlément(e,M)` : cette opération vise à ajouter le nouvel élément valide `e` au modèle *M*. Il s'agit d'associer `e` à l'élément *M* par un arc de type *défini-dans* (`e` est défini dans le modèle *M*) ou par un arc de type *contenu-dans* (le modèle *M* contient `e`). Si *M* existe dans la base de connaissances, l'ajout de `e` au modèle *M* peut impliquer l'insertion de nouveaux éléments à la base de connaissances. Par exemple, l'insertion de `e` si `e` est un nouvel élément à créer, l'insertion des arcs entre `e` et *M*.

- `AjoutÉléments(E,M)`: cette opération vise à ajouter les nouveaux éléments valides de l'ensemble *E* au modèle *M*. L'opération `AjoutÉlément(e,M)` peut s'appliquer à chaque élément *e* de *E*.
- Suppression d'éléments: `SuppressionÉlémentsDansModèle(E,M)`, `SuppressionÉlémentDansModèle(e,M)`, `SuppressionÉléments(E)`, `SuppressionÉlément(e)`, etc.
- Modification de valeurs et de libellés: les mises à jour des libellés, les mises à jour des interprétations d'éléments (y compris les mises à jour des valeurs d'attributs d'éléments).

Énumération

L'opération d'énumération, `Énumération(M,P)` retourne la liste *P* des éléments du modèle *M*. Cette liste indique l'ordre de ces éléments à visiter de manière à ce que chaque élément soit visité une seule fois. Nous pouvons appliquer les algorithmes de parcours en profondeur ou en largeur[6] qui sont très connus dans la théorie de graphes.

Application de fonction

L'opération `ApplicationFonction(f,M)` permet d'appliquer la fonction *f* à tous les nœuds du modèle *M*.

Copie de modèle

Nous spécifions les opérations suivantes concernant la copie de modèles:

- `CopieRéfNœud(V,U,e)`: cette opération retourne le modèle *U* après avoir copié la référence du nœud *e* du modèle *V* à *U*. Le nœud *e* dans *V* et sa copie dans *U* ont le même libellé ou la même référence. Aucun nouveau nœud n'est créé. Dans ce cas, l'opération `AjoutÉlément(e,U)` peut être appliquée.
- `CopieNœud(V,U,e,e')`: cette opération retourne le modèle *U* après avoir copié le nœud *e* du modèle *V* à *U*. Le nœud *e* dans *V* et sa copie (dont le libellé est désigné par *e'*) dans *U* sont libellés différemment. Le nouveau nœud *e'* est créé.
- `CopieArc(V,U,e)`: cette opération retourne le modèle *U* après avoir copié l'arc *e* du modèle *V* à *U*. L'arc *e* (entre deux éléments) dans *V* et sa copie *e'* (entre deux éléments copiés correspondants), défini dans *U*, sont deux arcs de même type mais

[6] http://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur; http://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur

de références différentes.

- `Copie(V,U)` : cette opération consiste à copier les éléments du modèle *V* dans le modèle *U*, puis elle retourne *U*. L'opération `CopieRéfNœud(V,U,e)` est appliquée pour chaque nœud *e* contenu dans *V*. `CopieArc(V,U,e)` est appliquée pour chaque arc *e* dans *V*. Si le modèle *U* à l'entrée de `Copie(V,U)` est vide, alors le modèle *U*, après l'exécution de `Copie(V,U)` est le modèle copié du modèle *V*.
- `Copie(V,U,Ev,Eu)` : cette opération se comporte comme `Copie(V,U)` excepté que *Eu* désigne la liste des libellés des nœuds qui représentent dans *U* les copies correspondantes des nœuds de *V* présents dans la liste *Ev*. L'opération `CopieRéfNœud(V,U,e)` s'exécute pour chaque nœud *e* dans *V* toutefois absent de la liste *Ev*. `CopieNœud(V,U,e,e')` s'exécute pour chaque nœud *e* dans *V* et présent dans la liste *Ev*, *e'* désigne le libellé correspondant dans la liste *Eu* pour la copie de *e*. Et `CopieArc(V,U,e)` s'exécute pour chaque arc *e* dans *V*.

  Par exemple, dans le cas où *V*=`'Structure-instOf'`, *U*=`'Structure-instOf2'`, *Ev*=`(Object,ObjType)` et *Eu*=`(Role,RoleType)`: l'exécution de `Copie(V,U,Ev,Eu)` permet de créer le modèle `Structure-instOf2`, une autre structure de `instOf`, copié de `Structure-instOf`. Toutefois, dans `Structure-instOf2`, les éléments `Object` et `ObjType` sont respectivement remplacés par `Role` et `RoleType`. La Figure 199 illustre ces deux modèles `Structure-instOf` et `Structure-instOf2`.

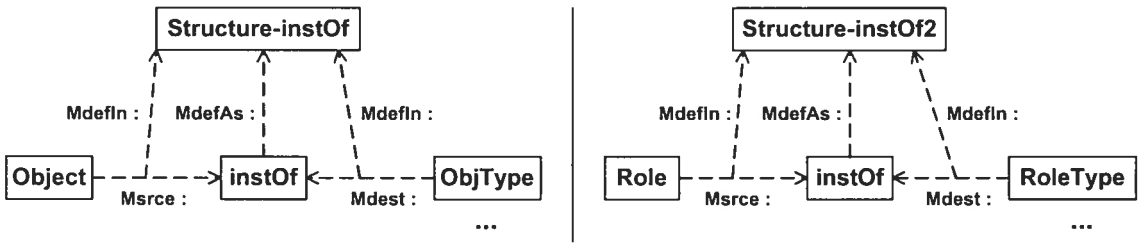


Figure 199 : Copie de modèles

Simplification de modèle

L'opération `Simplification(M)` consiste à supprimer les éléments dupliqués/redondants dans le modèle *M*, sans perdre d'information ni modifier la sémantique de *M*. Cette opération peut faire appel aux trois opérations primitives (illustrées dans la Table 5, page 205) par l'ordre-ci: (i) `JointureInterne(M)`, (ii)

`SuppArcsRédondants(M)`, et (iii) `SuppArcsDéduits(M)`.

- `JointureInterne(M)` : cette opération de jointure interne de modèle retourne le modèle M après avoir supprimé dans M les éléments dupliqués. Si deux nœuds (ou deux arcs) e_1 et e_2 dans M ont un même identifiant (c'est-à-dire un même libellé ou bien une même référence), ils sont considérés comme identiques. De ce fait, si l'un des deux est supprimé, par exemple e_1, les arcs de e_1 sont reliés à e_2 plutôt que d'être reliés à e_1.
- `SuppArcsRédondants(M)` : cette opération retourne le modèle M après avoir supprimé les arcs identiques et redondants dans M. Dans M, s'il existe deux arcs r_1 et r_2 identiques (cf. la Règle 23, Annexe IV) dont les références sont différentes, et que la suppression de l'un des deux, par exemple de r_1, ne modifie pas la sémantique de M, alors r_1 peut être supprimé en appelant `SuppressionÉlément(`r_1`,M)`. L'utilisateur peut intervenir dans cette phase pour déterminer si un arc est à supprimer ou non.
- `SuppArcsDéduits(M)` : cette opération retourne le modèle M après avoir supprimé dans M les arcs qui peuvent être déduits à partir de d'autres connaissances en appliquant les règles concernées. Dans l'exemple pour `SuppArcsDéduits(M)` illustré dans la Table 5 (page 205), comme `Adulte` est un sous-type de `Personne` et qu'il peut hériter des attributs de `Personne` (par la règle de sous-typage), l'arc de type `chrcType` de l'attribut `Nom` à `Adulte` devient inutile et peut donc être supprimé.

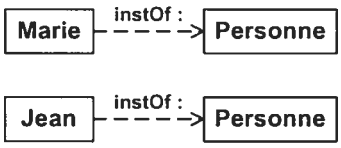
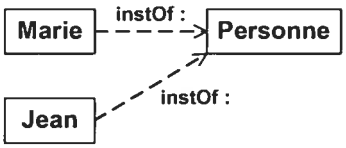
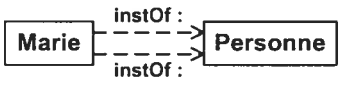
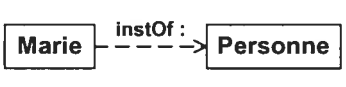
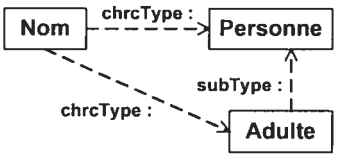
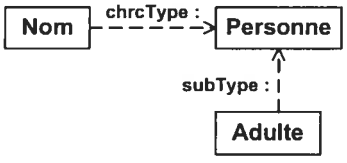
	Modèle à l'entrée	**Modèle à la sortie**
`JointureInterne(M)`		
`SuppArcsRédondants(M)`		
`SuppArcsDéduits(M)`		

Table 5 : Opérations primitives relatives à la simplification de modèles

Restriction de types / Relaxation de types relationnels

Nous spécifions ces deux opérations de restriction de types:

- `RestrictionType(V, U, t1, t2)`: cette opération consiste à remplacer un type t_1 dans le modèle V par un sous-type t_2 de t_1 pour en résulter un nouveau modèle U. Alors, cette opération peut être mise en œuvre par l'opération `Copie(V, U, Ev, Eu)` où $Ev=(t_1)$ et $Eu=(t_2)$.
- `RestrictionTypes(V, U, Ev, Eu)`: cette opération est une extension de `RestrictionType(V, U, t1, t2)` où Ev désigne la liste des types dans V à être remplacés par les sous-types correspondants dans la liste Eu pour en résulter le modèle U. Cette opération peut être implémentée par `Copie(V, U, Ev, Eu)`.

Le mécanisme de restriction de types vise à remplacer dans un modèle un type par un type plus spécifique, permettant ainsi l'émergence d'un nouveau modèle plus spécifique. Pour sa part, le mécanisme de relaxation de types relationnels vise à remplacer un type relationnel par un type plus général, permettant ainsi l'émergence d'un nouveau modèle plus général. Donc, la restriction de types est un cas simple de la spécialisation entre modèles, et la relaxation de types relationnels est un cas simple de la dérivation de modèles. La spécialisation entre modèles et la dérivation de modèles sont spécifiées en détail dans la section 7.2.2.1 (page 208).

Jointure/Intersection/Différence entre modèles

L'opération de jointure entre modèles, `JointureModèles(M1, M2)`, retourne le modèle qui est le résultat de l'union des deux modèles M_1 et M_2 en se basant sur les éléments communs de M_1 et M_2. Cette opération peut faire appel à l'opération de copie de modèles (`Copie(V, U)`) ainsi qu'à celle de jointure interne de modèle (`JointureInterne(M)`). Elle peut être définie comme suit:

```
Opération JointureModèles(M1, M2) {
  M3 : un modèle vide;
  varM3 := Copie(M1, M3) ; // copier les éléments du modèle M1 dans le modèle M3
  varM3 := Copie(M2, M3) ; // copier les éléments du modèle M2 dans le modèle M3
  // simplifier le modèle M3 obtenu en appliquant l'opération de jointure interne de M3
  retourne JointureInterne(M3) ;
}
```

L'opération d'intersection entre modèles, `IntersectionModèles(M1, M2)`, retourne le modèle qui contient tous et seulement les éléments présents à la fois dans les deux

modèles M_1 et M_2.

```
Opération IntersectionModèles(M1,M2){
  M3 : un modèle vide;
  // copier tous les éléments présents à la fois dans les deux modèles M1 et M2
  pour chaque (élément e de M1 et de M2) faire
     varM3 := CopieÉlément(M1,M3,e); // CopieÉlément(M1,varM3,e);
  retourne M3; // varM3
}
```

L'opération de différence entre modèles, `DifférenceModèles(`M_1`,`M_2`)`, retourne le modèle qui contient seulement les éléments présents dans le modèle M_1 mais non dans M_2.

```
Opération DifférenceModèles(M1,M2){
  M3 : un modèle vide;
  // copier tous les éléments présents dans le modèle M1 mais non dans M2
  pour chaque (élément e de M1 mais pas de M2) faire
     varM3 := CopieÉlément(M1,M3,e);
  retourne M3;
}
```

Il est à souligner que dans notre formalisme, les rapports entre M_1, M_2 et M_3 dans les opérations `JointureModèles(`M_1`,`M_2`,`M_3`)`, `IntersectionModèles(`M_1`,`M_2`,`M_3`)` et `DifférenceModèles(`M_1`,`M_2`,`M_3`)` sont représentés par des arcs de type `Mjoin`, `Mdiff`, `Mintersection` et `Mresult` si M_1, M_2 et M_3 appartiennent au niveau méta. Ils sont toutefois représentés par des arcs de type `join`, `diff`, `intersection` et `result` si M_1, M_2 et M_3 appartiennent au niveau M1. Par exemple, la Figure 200 montre un exemple de jointure entre modèles au niveau M1: le modèle `Mz` est le modèle résultant de l'union des deux modèles `Mx` et `My`.

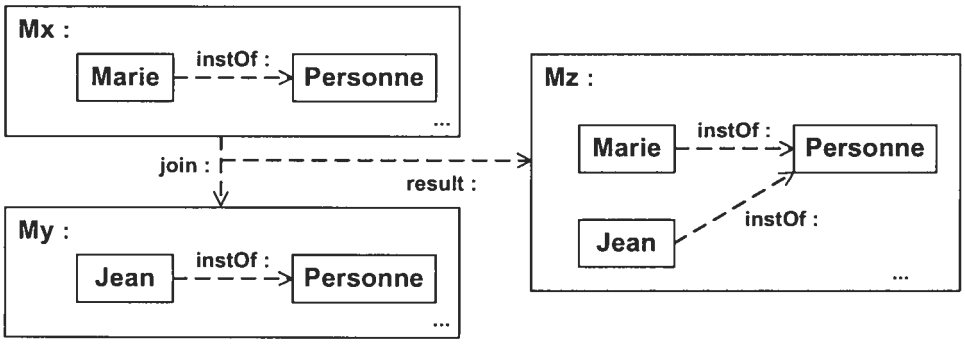


Figure 200 : Exemple de jointure entre modèles de niveau M1

7.2.2 Opérations avancées

Cette section est réservée pour les opérations traitant les problèmes suivants: la dérivation de modèles, la spécialisation entre modèles, l'instanciation de modèles, la migration de modèles, et la requête sur des modèles.

7.2.2.1 Dérivation de modèles / Spécialisation entre modèles

Si le mécanisme de dérivation de modèles vise à construire des modèles plus généraux, celui de spécialisation entre modèles vise à construire des modèles plus spécifiques. Dans notre formalisme, au niveau méta (ou au niveau M1), si un modèle `Mx` est dit *dérivé* d'un modèle `My`, c'est-à-dire si `My` est plus *restreint/spécifique* que `Mx`, `My` peut être attaché à `Mx` par un arc de type `Mrestrict` (ou de type `restrict`).

La spécialisation entre modèles au niveau méta est précisée par la Définition 15, sur la base des Définition 16 et Définition 17. La spécialisation entre modèles au niveau M1 est aussi précisée par la Définition 15 mais sur la base des Définition 16 et Définition 18.

Définition 15 : Spécialisation entre modèles

Soient m_1, m_2 deux modèles. m_1 est dit plus *restreint/spécifique* que m_2, s'il existe un sous-modèle m_3 de m_1 de sorte que m_3 soit identique à m_2 (cf. la Définition 16, page 208) excepté qu'une ou plusieurs structures dans m_2 sont restreintes à celles plus spécifiques dans m_3 (cf. les Définition 17 et Définition 18).

La Définition 16 spécifie le rapport entre modèles identiques ainsi que le rapport de sous-modèle entre modèles.

Définition 16 : Modèles identiques / Sous-modèle d'un modèle

Soient m_1, m_2 deux modèles. m_1 et m_2 sont dits *identiques* s'ils ont un même contenu en terme que m_1 contient tous les éléments de m_2 et que m_2 contient tous les éléments de m_1. m_1 est dit un *sous-modèle* de m_2 si m_2 contient tous les éléments de m_1.

Il est important de respecter les Règle 9, Règle 10 et Règle 11 (Annexe IV) sur la présence d'un lien ou d'un type relationnel dans un modèle.

La Définition 17 précise la spécialisation entre structures de méta-arcs. Par exemple, illustrées dans la Figure 102 (page 117), la structure (b) du méta-arc `name` est plus restreinte/spécifique que la structure (a).

Définition 17 : Spécialisation entre structures de méta-arcs

Soient R_1, R_2 des méta-arcs. Soit S_1 une structure de R_1 entre A_1, A_2; et soit S_2 une

structure de R_2 entre B_1, B_2. Supposons que les éléments A_1, A_2 et B_1, B_2 sont regroupés en paire en fonction de leurs arcs à R_1 et à R_2 : (A_1, B_1), (A_2, B_2).

S_1 est dite *plus restreinte* ou bien *plus spécifique* que S_2, si S_1 et S_2 respectent l'un des deux scénarios suivants:

– Dans le premier scénario, les conditions suivantes doivent être satisfaites.

(1a) R_1 et R_2 sont identiques;

(1b) A_i et B_i sont identiques pour chaque i=1, 2;

(1c) pour chaque i=1, 2 : ($1c_i$) les contraintes de cardinalités sur A_i doivent être les mêmes ou plus restreintes que celles sur B_i;

(1d) il existe un indice i égal à 1 ou 2 pour que ($1d_i$) les contraintes de cardinalités sur A_i doivent être plus restreintes que celles sur B_i.

– Suivant le deuxième scénario, les conditions suivantes doivent être satisfaites:

(2a) R_1 et R_2 sont identiques, ou R_1 est un sous-type de R_2;

(2b) l'élément A_1 doit être soit B_1 soit un sous-type de B_1, et de même pour A_2 et B_2;

(2c) R_1 est un sous-type de R_2, ou il existe un indice i égal à 1 ou 2 pour que l'élément A_i soit un sous-type de B_i;

(2d) en principe, pour chaque i=1, 2, ($2d_i$) les contraintes de cardinalités sur A_i doivent être les mêmes ou plus restreintes que celles sur B_i.

Les conditions ($1c_i$), ($1d_i$), et ($2d_i$) peuvent également s'exprimer comme suit. Soient $cardMinA_i$, $cardMaxA_i$ les cardinalités minimale et maximale de A_i, et $cardMinB_i$, $cardMaxB_i$ celles de B_i. La condition ($1c_i$) ou bien ($2d_i$) spécifient que «$cardMinA_i \geq cardMinB_i$ et $cardMaxA_i \leq cardMaxB_i$». Et ($1d_i$) spécifie que [$cardMinA_i$ .. $cardMaxA_i$] est un sous-intervalle de [$cardMinB_i$ .. $cardMaxB_i$] (soit «$cardMinA_i \geq cardMinB_i$ et $cardMaxA_i < cardMaxB_i$», soit «$cardMinA_i < cardMinB_i$ et $cardMaxA_i \leq cardMaxB_i$»).

La Définition 18 porte sur la spécialisation entre structures de types de relations. Cette définition tient compte des diverses caractéristiques structurelles des types de relations de niveau M1, par exemple, la participation possible des instances à un type de relations, les divers types de contraintes structurelles possibles sur les types de relations. À titre d'exemple, la structure de `SignerContrats` dans la Figure 169 (page 165) est plus générale que celles dans les Figure 170 (page 167) et Figure 172 (page 168).

Définition 18 : Spécialisation entre structures de types de relations

Soient R_1, R_2 deux types de relations; S_1 une structure de R_1 entre A_1, ..., A_n; et S_2 celle de R_2 entre B_1, ..., B_n. Supposons que les éléments A_1, ..., A_n et B_1, ..., B_n sont regroupés en paire en fonction de leurs arcs à R_1 et à R_2: (A_1, B_1), ..., (A_i, B_i), ..., (A_n, B_n). S_1 est dite plus restreinte ou plus spécifique que S_2 si les conditions suivantes sont remplies :

(i) si un élément soit B_i est un type impliqué entièrement dans R_2 selon S_2, alors les éléments A_i et B_i doivent être les mêmes et A_j (c'est-à-dire B_j) doit être aussi impliqué entièrement dans R_1 selon S_1;

(ii) si un élément soit B_j est une instance impliquée de manière fixe dans R_2 selon S_2, alors les éléments A_j et B_j doivent être les mêmes et A_j (c'est-à-dire B_j) doit être aussi impliquée de manière fixe dans R_1 selon S_1;

(iii) en plus, S_1 et S_2 suivent l'un des deux scénarios suivants:

- Dans le premier scénario, ces conditions doivent être satisfaites:

  (1a) R_1 et R_2 sont identiques;

  (1b) A_i et B_i sont identiques pour chaque i=1, 2, ..., n;

  (1c) pour chaque i=1, 2, ..., n: les contraintes de cardinalités sur A_i doivent être les mêmes ou plus restreintes que celles sur B_i;

  (1d) il existe un indice i de [1, n] tel que (1d$_i$) les contraintes de cardinalités sur A_i sont plus restreintes que celles sur B_i.

- Suivant le deuxième scénario, ces conditions doivent être satisfaites:

  (2a) R_1 et R_2 sont identiques ou R_1 est un sous-type de R_2;

  (2b) pour chaque i=1, ..., n: l'élément A_i doit être soit B_i, soit un sous-type de B_i, soit une instance de B_i;

  (2c) R_1 est un sous-type de R_2 ou il existe un indice i de [1..n] tel que l'élément A_i soit un sous-type de B_i ou soit une instance de B_i;

  (2d) Obligatoirement, pour chaque i=1, 2, ..., n: les contraintes de cardinalités sur A_i doivent être les mêmes ou plus restreintes que celles sur B_i.

Les conditions (1c), (1d), et (2d) peuvent aussi s'exprimer comme suit. Pour les contraintes de cardinalité d'une même sorte (c'est-à-dire celles qui sont indiquées par des arcs d'un même type dans le groupe de `card`), soient `cardMinA`$_i$, `cardMaxA`$_i$ les cardinalités minimale et maximale de A_i, et `cardMinB`$_i$, `cardMaxB`$_i$

celles de B_i. La condition (1c) ou bien (2d) spécifient que «$cardMinA_i \geq cardMinB_i$ et $cardMaxA_i \leq cardMaxB_i$». Et (1d) spécifie que [$cardMinA_i$ .. $cardMaxA_i$] est un sous-intervalle de [$cardMinB_i$ .. $cardMaxB_i$] (soit «$cardMinA_i \geq cardMinB_i$ et $cardMaxA_i < cardMaxB_i$» soit «$cardMinA_i < cardMinB_i$ et $cardMaxA_i \leq cardMaxB_i$»).

7.2.2.2 Instanciation de modèles

L'opération d'instanciation de modèles (cf. la section 2.3.2, page 21), `estInstModèle(`M_1`,`M_2`)`, consiste à déterminer si le modèle M_1 est une instance du modèle M_2. Si c'est le cas, dans notre formalisme, M_1 peut être relié à M_2 par un arc de type `instModelOf`. Les modèles à traiter ici, M_1 et M_2, appartiennent au niveau M1.

Comme au niveau M1, un modèle peut être un contexte (de nature `Context`), ou une structure (de nature `Structure`), etc., dans le cas où le modèle M_1 est instancié de M_2, l'opération `estInstModèle(`M_1`,`M_2`)` peut déterminer précisément, selon le cas, si le modèle M_1 est un contexte ou une structure et une instance directe ou indirecte de M_2. Cette opération peut être mise en œuvre en basant sur les définitions présentées ci-dessous.

La Définition 19 définit le rapport d'instanciation entre un contexte (de nature `Context`) et une structure (de nature `Structure`) correspondante (de niveau M1). La Figure 201 (page 212) montre un exemple. Le contexte `Contexte-marie` contextualise le modèle de la relation `marie-travailler` de type `travailler`. Il est une instance de la structure `Structure-travailler` du type de relations `travailler`, et il est attaché à `Structure-travailler` par un arc de type `instModelOf`.

Définition 19 : Rapport d'instanciation entre un contexte et une structure (de niveau M1)

Soient R un type de relations; S une structure de R entre A_1, ..., A_n; et C un contexte d'une relation $InstR$ entre B_1, ..., B_n. À noter que le fait qu'un type (T) s'implique entièrement dans une relation (respectivement type de relations) est interprété comme le fait que chaque instance de T s'implique dans la relation (respectivement le type de relations) en question. Supposons que les éléments A_1, ..., A_n et B_1, ..., B_n sont regroupés en paire en fonction de leurs arcs à R selon S et à $InstR$ selon C : (A_1, B_1), ..., (A_i, B_i), ..., (A_n, B_n).

C est dit une instance directe de S si les conditions suivantes sont satisfaites:

(a) *InstR* est une instance de *R*;

(b) Pour chaque i=1, ..., n:

- Si A_i est une instance impliquée dans *R* selon *S*, alors les éléments A_i et B_i doivent être les mêmes et B_i (c'est-à-dire A_i) doit aussi être impliquée dans *InstR* (selon *C*);
- Si A_i est type impliqué entièrement dans *R* selon *S* (c'est-à-dire que chaque instance de A_i s'implique dans *R* selon *S*), alors les éléments A_i et B_i doivent être les mêmes et B_i (c'est-à-dire A_i) doit être aussi impliqué dans *InstR* (selon *C*);
- Si A_i est un type (impliqué non entièrement dans *R* selon *S*), alors B_i doit être une instance de A_i;

(c) Et l'existence du modèle *C* de *InstR* ne viole aucune contrainte sur *R*.

Un contexte d'une relation est dit une instance indirecte d'une structure (*S*) d'un type de relations s'il est directement instancié d'une structure plus spécifique que *S* (cf. la Définition 18, page 210).

Un contexte (*C*) est dit une instance d'une structure (*S*) si son contenu est conforme à celui de *S* en ce sens que *C* n'est composé que des contextes de relations dont chacun constitue une instance d'une structure d'un type de relations dans *S*.

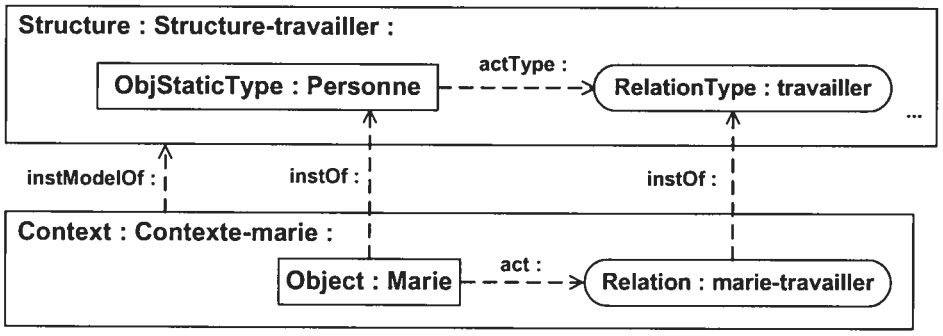


Figure 201 : Instanciation de modèles

La Définition 20 définit le rapport d'instanciation entre structures (de niveau M1). Dans ce cas, une instance d'une structure demeure une structure. Par exemple, la structure du type de relations `gérer` illustrée par la Figure 126 (page 130) est instanciée de celle à la Figure 114 (page 123); la structure de `SignerContrats` illustrée par la Figure 170 (page 167) est instanciée de celle à la Figure 169 (page 165). Clairement, une structure qui est une instance d'une autre structure est aussi une structure plus spécifique

que cette dernière.

Définition 20 : Rapport d'instanciation entre structures d'un même type de relations (de niveau M1)

Soient R un type de relations; S_1 une structure de R entre A_1, ..., A_n; et S_2 celle de R entre B_1, ..., B_n. Le fait qu'un type (T) s'implique entièrement dans un type de relations est interprété comme le fait que chaque instance de T s'implique dans le type de relations en question. Supposons que les éléments A_1, ..., A_n et B_1, ..., B_n sont regroupés en paire en fonction de leurs arcs à R selon S_1 et à R selon S_2: (A_1, B_1), ..., (A_i, B_i), ..., (A_n, B_n).

S_1 est dite une instance directe de S_2 si les conditions suivantes sont satisfaites:

(a) Pour chaque i=1, ..., n:

- Si B_i est une instance impliquée dans R selon S_2, alors les éléments A_i et B_i doivent être les mêmes et B_i (c'est-à-dire A_i) doit être aussi impliquée de façon fixe dans R selon S_1;
- Si B_i est un type impliqué entièrement dans R selon S_2 (chaque instance de B_i s'implique dans R selon S_2), alors les éléments A_i et B_i doivent être les mêmes et B_i (c'est-à-dire A_i) doit être aussi impliqué entièrement dans R selon S_1;
- Si B_i est un type (impliqué non entièrement dans R selon S_2), alors A_i doit être soit B_i, soit un sous-type de B_i, soit une instance de B_i;

(b) Il existe un indice i de [1..n] tel que l'élément A_i est une instance de B_i;

(c) Les contraintes sur R sont toujours respectées.

Une structure d'un type de relations est dite une instance indirecte d'une autre structure (S) de ce type de relations si elle est directement instanciée d'une structure plus spécifique que S (cf. la Définition 18, page 210).

Une structure (S_1) est dite une instance d'une autre structure (S_2) si S_1 est identique à S_2 excepté qu'au moins une structure (S) dans S_2 a été restreinte à une ou des instances (de S) dans S_1. Voir aussi la Définition 19 (page 211).

En se basant sur ces Définition 19 et Définition 20, la Définition 21 spécifie le rapport d'instanciation entre modèles (de niveau M1).

Définition 21 : Rapport d'instanciation entre modèles (de niveau M1)

Un modèle (m_1) est dit une instance d'un autre modèle (m_2) si m_1 est identique à m_2

excepté qu'au moins une structure (S) dans m_2 a été restreinte à une ou des instances (de S) dans m_1.

7.2.2.3 Migration de modèles

La migration de modèles est un mécanisme qui consiste à faire migrer dynamiquement les instances (objets/rôles) directes d'un type vers des types plus représentatifs/pertinents en respectant la définition de ces derniers (Définition 22). Tel que démontré dans la section 6.2.2.12 (page 186), notre formalisme permet de représenter les contraintes de migration. Cette section sur la migration de modèles souligne donc le côté opérationnel de ces contraintes.

Définition 22 : Rapport de migration entre modèles (de niveau M1)

Un modèle (m_1) est dit le modèle migré d'un autre modèle (m_2) si m_1 (le contenu) est identique à m_2 excepté que les instances directes des types dans m_2 sont migrés vers des types plus *représentatifs/pertinents* dans m_1 (en terme que les instances en question, suivant leurs états actuels, devraient être classées aux derniers types qu'aux premiers types), en respectant la définition des types impliqués.

L'opération de migration de modèles, `MigrationModèle(M)`, retourne le modèle migré (*Mmig*) du modèle *M* après avoir migré les objects et les rôles de *M* vers les types plus pertinents. Le fonctionnement de cette opération peut être décrit comme suit:

```
Opération MigrationModèle(M) {
  Mmig : un modèle vide;
  varMmig:= Copie(M,Mmig) ;//copier les éléments du modèle M au modèle Mmig
  E := {e | e représente un objet ou rôle contenu dans M} ;
  pour chaque (élément e de E) faire {
     // modifier Mmig en fonction de migrer e vers des types plus pertinents
     varMmig := MigrationElement(Mmig,e) ;
  } ; // pour
  retourne Mmig;
}
```

L'opération `MigrationElement(M,e)` consiste à migrer l'élément *e* contenu dans *M* vers des types plus pertinents, mettre à jour *M* conformément à la migration de *e*, puis retourner *M* comme le modèle résultant de cette migration. Le fonctionnement de

`MigrationElement(M,e)` peut être décrit en basant sur de d'autres opérations comme suit:

```
Opération MigrationElement(M,e){
  /* Types(e): retourner l'ensemble des types auxquels e est relié par des liens
  d'instanciation */
  varT1 := Types(e);
  /* TypesAbandonnés(e): retourner l'ensemble des types dans varT1
  abandonnés par e, c'est-à-dire les types qui ne seront plus un type direct de e une
  fois que e est migré vers de d'autres types plus pertinents */
  varT2 := TypesAbandonnés(e);
  /* EstMigréVersTypes(e,varT2): retourner l'ensemble des types plus
  pertinents qui deviennent des types directs de e et vers lesquels l'élément e est
  migré des types dans varT2 */
  varT3 := EstMigréVersTypes(e,varT2);
  /* MigrerElementDeTypesVersTypes(M,e,varT2,varT3):
    – supprimer dans le modèle M les arcs représentant les liens d'instanciation entre
      e et les types dans l'ensemble varT2,
    – ajouter dans le modèle M les arcs représentant les liens d'instanciation entre e et
      les types dans l'ensemble varT3
    – finalement, retourner le modèle M après les mises à jour effectuées ci-dessus */
  retourne MigrerElementDeTypesVersTypes(M,e,varT2,varT3);
}
```

7.2.2.4 Requête

Cette section porte sur notre solution permettant la recherche ou la sélection d'information (sur un modèle ou un objet, un type, etc.) dans la base de modèles.

Comme nous avons présenté dans la section 7.1 (page 194), la base de modèles est une base de données relationnelle qui comporte seulement deux tables, *Tbar* et *Value*, respectivement pour le codage des éléments libellés (cf. la Figure 196, page 196) et pour le stockage de valeurs d'éléments libellés (cf. la Figure 198, page 199). Pour cette solution de stockage de modèles, toute requête dans notre base de modèles peut être formulée au moyen d'un langage d'interrogation des bases de données relationnelles

(SQL) supporté par un système de gestion de bases de données disponible tel que MySQL, SQL Server, etc. À titre d'exemple, nous présentons ci-après la mise en œuvre en SQL Server de quelques requêtes fondamentales:

- L'opération `Code(id)` consiste à retourner le libellé d'un élément dont le Id est désigné par *id*:

  "**SELECT** TbarLabel **FROM** Tbar **WHERE** idTbar =" + *id*;

- L'opération `Id(label)` consiste à retourner le Id d'un élément dont le libellé est désigné par *label* :

  "**SELECT** idTbar **FROM** Tbar **WHERE** TbarLabel =" + *label*;

- L'opération `CodeMétaÉlement(label)` consiste à retourner le libellé du méta-élément d'un élément donné où le libellé de cet élément est désigné par *label*:

  "**SELECT** Tb1.TbarLabel **AS** metaEleCode **FROM** Tbar **AS** Tb1 **INNER JOIN** Tbar **AS** Tb2 **ON** Tb1.idTbar=Tb2.dest **AND** Tb2.TbarLabel =" + *label*;

- Soit `e` un élément dont le Id est désigné par *idDest*, soit `m_a` un méta-arc dont le Id est désigné par *idMetaArc*, l'opération `IdÉlementsSrce(idMetaArc, idDest)` consiste à retourner les Id de tous les éléments qui sont reliés à un élément `e` par des arcs de type `m_a`:

  "**SELECT** Tb1.idTbar **FROM** Tbar **AS** Tb1 **INNER JOIN** Tbar **AS** Tb2 **ON** Tb1.idTbar=Tb2.dest **AND** Tb2.metaArc=" + *idMetaArc* +" **AND** Tb2.dest=" + *idDest* +" **ORDER BY** 1";

  D'une façon similaire, l'opération `IdÉlementsDest(idMetaArc, idSrce)` est défini pour trouver les Id de tous les éléments auxquels un élément `e'` (dont le Id est désigné par *idSrce*) par des arcs de type `m_a` (dont le Id est désigné par *idMetaArc*).

- etc.

Voici un exemple d'application des opérations de requêtes définies ci-dessous. `IdÉlementsSrce(Id('defIn'), Id('ORG_Model'))` retourne tous les Id des éléments directement définis dans le modèle libellé 'ORG_Model' de niveau M1 (ces éléments sont reliés à ce modèles par des arcs de type `defIn`).

Nous présentons maintenant les solutions permettant à l'usager de spécifier ses requêtes dans notre base de modèles, ne connaissant ni la structure de cette base de modèles ni la mise en œuvre de notre formalisme.

Une solution possible consiste à définir un langage de requête générique, associé à notre formalisme mais indépendant de la mise en œuvre de ce dernier. Ce langage peut être similaire à celui que Lemesle a présenté dans [80] pour les recherches dans un modèle ou métamodèle sNets. Dans le cas où notre base de modèles est implémentée comme une base de données relationnelle, toute requête formulée dans le langage de requête en question, lors de son exécution, devrait être traduite (par un l'outil) en langage d'interrogation de la base de données relationnelle supporté par le système de gestion de bases de données qui gère notre base de modèles.

Une autre solution bien simple et supportée par notre formalisme, c'est de formuler une requête sous forme d'une règle. L'opération de requête, `Requête(Req)` retourne le résultat après avoir effectué la requête *Req* (sous forme d'une règle) dans la base de modèles.

Comme nous l'avons démontré dans les deux chapitres précédents (cf. la section 5.2.2.6, page 105, la section 5.3.7, page 147, et la section 6.2.2, page 165), notre formalisme permet de représenter les règles sémantiques réparties à tous les niveaux de modélisation pour contraindre les éléments représentés dans notre formalisme. Lorsqu'une règle est effectuée en tant qu'une requête (*Req*), la règle est interprétée d'une façon un peu particulière. La partie de pré-conditions (*PreCond*) de *Req* exprime les contraintes de la requête alors que la partie de post-conditions (*PostCond*) exprime les informations à retourner comme le résultat de cette requête. Ainsi, la partie *PreCond* de *Req* peut contenir uniquement des variables déclarées dans la parties *PostCond* et le résultat retourné après l'exécution de `Requête(Req)` est constitué par les éléments concrets qui peuvent remplacer les variables en question de telle sorte que chaque substitution de ces variables par les éléments concrets permette de vérifier les critères exprimés par la partie de pré-condition. La Figure 202 illustre un exemple de formuler une requête sous forme d'une règle: chercher tout élément (x) qui est le méta-élément de `Object` («`x:Object`»). La Figure 203 présente une autre requête plus complexe (cf. l'exemple 34, page 185) qui consiste à chercher tous les rôles «employés» (x) et les personnes (z) de telle sorte que ces personnes («`Object:z:Personne`») s'occupent de ces rôles «employés» («`Role:x:Employé`») et que ces rôles («`Role:x:Employé`») travaillent dans les départements («`Object:y:Dépt`») appartenant à l'organisation `UNICEF` («`Object:UNICEF:Organisation`»).

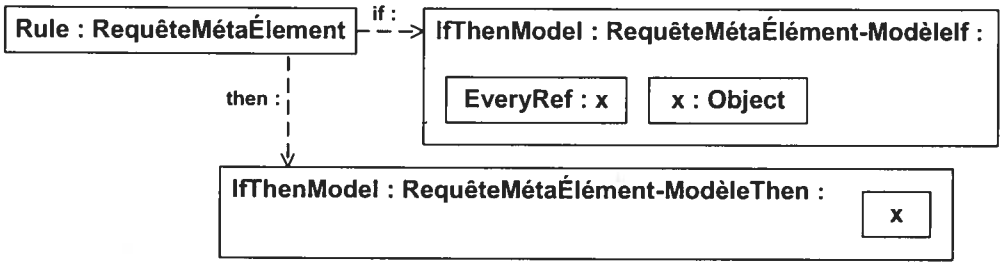


Figure 202 : Un exemple de requête sur le méta-élément de *Object*

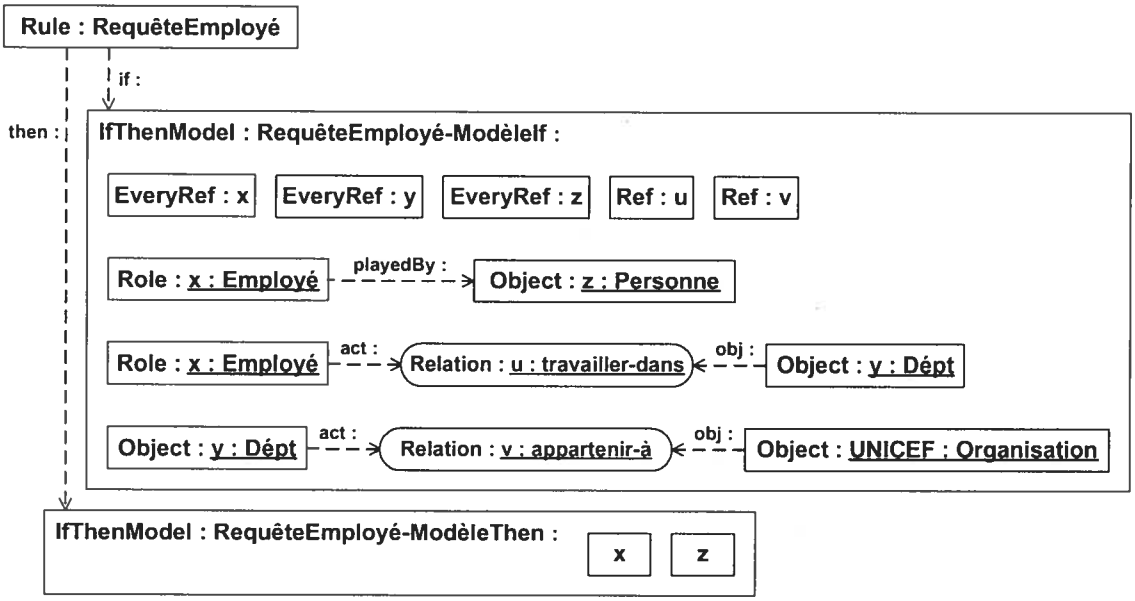


Figure 203 : Un autre exemple de requête

7.3 Prototype

Cette section présente le prototype implémentant notre formalisme pour la gestion de modèles.

Dans le cadre du stage de mes études de maîtrise, nous avons proposé un formalisme pour la représentation des connaissances statiques de l'entreprise et aussi mis en œuvre un prototype de validation ([41]). Nous fait évoluer notre formalisme dans le cadre de nos travaux de recherche de doctorat sur la gestion de modèles pour la gestion de connaissances et nous avons continué à travailler sur ce prototype.

Le prototype a été développé en langage JScript, exécuté par un navigateur Internet, interfacé avec une base de données relationnelle SQL Server ([38], [39], [41]). Ce prototype accepte à l'entrée des données représentées sous forme de document XML en vérifiant une partie des contraintes attachées aux données chargées. Dans ce prototype,

les modèles de représentation de connaissances sont stockés au sein d'une base de données relationnelle, comme présenté dans la section 7.1 (page 194). Le prototype permet également d'observer des types/objets entreposés au sein d'une base de données relationnelle (cf. la Figure 204, page 219, et la Figure 205, page 220).

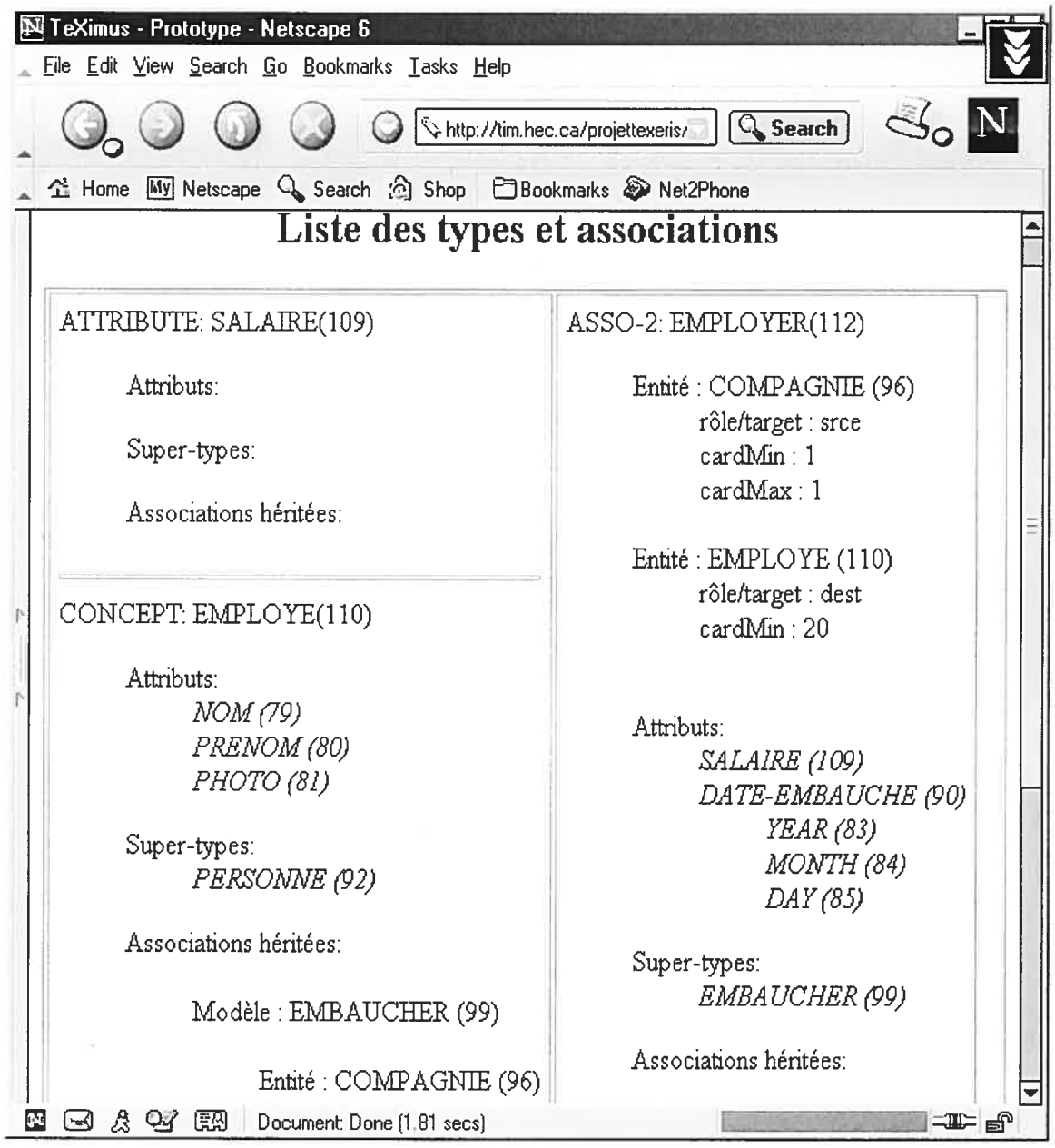


Figure 204 : Visualisation textuelle des types

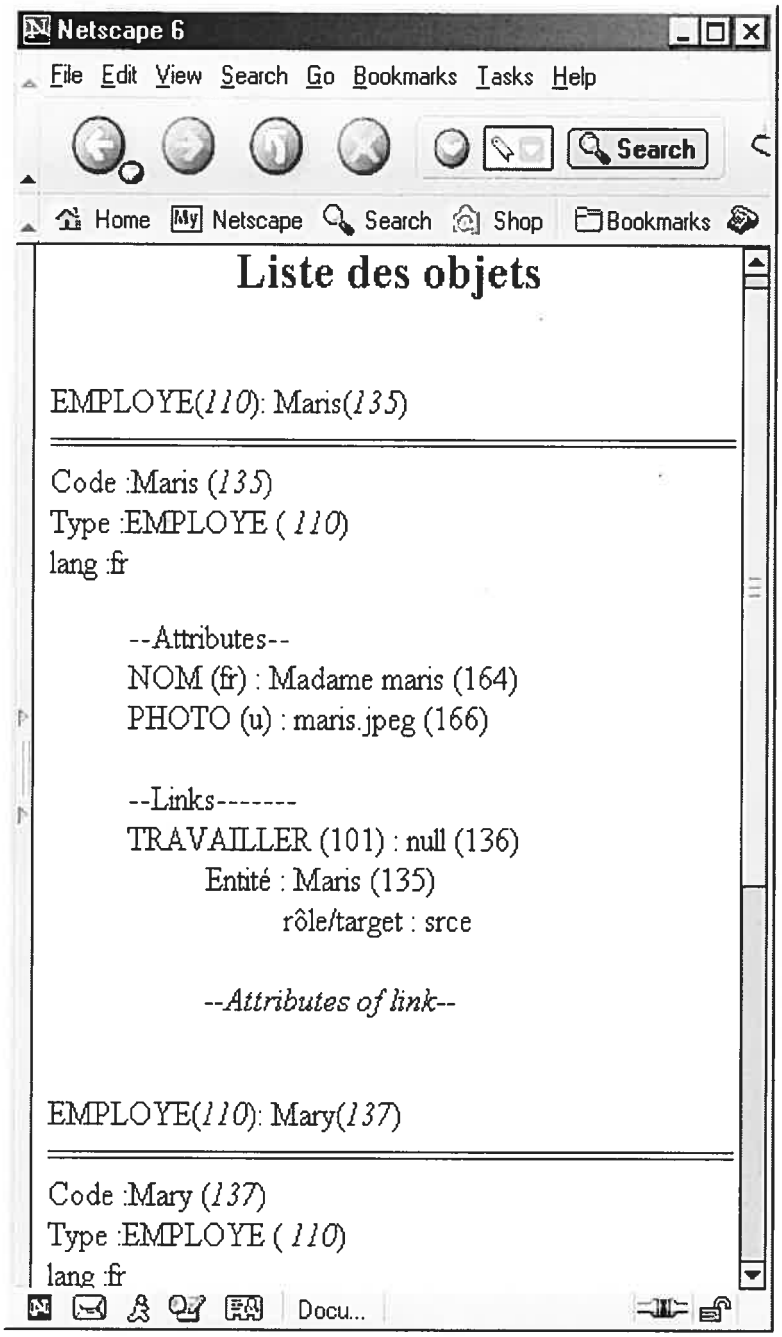


Figure 205 : Visualisation textuelle des objets

Nous présentons ci-dessous l'architecture (Figure 206) dans lequel s'intègre ce prototype (contrôleur de BD de modèles) présenté ci-dessus. Le composant central du prototype, l'interpréteur/validateur de modèles, consiste à interpréter et à valider un modèle. Il communique (lire/écrire) avec la base de modèles via le contrôleur de la base de modèles. Il fait appel à des opérations sur les modèles, ainsi qu'à des outils qui permettent d'éditer des modèles, d'éditer des règles de contrôle (pour la validation

syntaxique et sémantique de modèles) et de visualiser des modèles. Le fonctionnement de l'interpréteur/validateur de modèles suit les comportements des éléments du méta-métamodèle et ceux du métamodèle de notre formalisme que nous avons spécifiés dans le Chapitre 5 (page 92). La sémantique et le comportement pour chacune des opérations nécessaires pour notre besoin de manipulation de modèles sont précisés dans la section 7.2 (page 199).

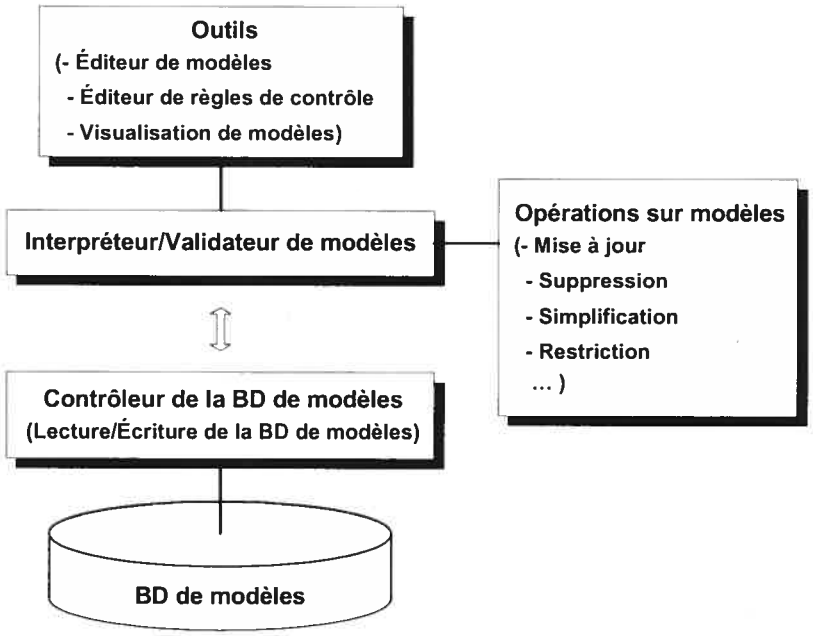


Figure 206 : Architecture d'un prototype de système de gestion de modèles

En résumé, les objectifs principaux de notre prototype sont les suivants:

1. Opéraliser le méta-métamodèle et le métamodèle proposés, autrement dit développer l'interpréteur/validateur de modèles. La sémantique de chaque modèle doit être respectée et la correction des modèles à charger dans la base de connaissances (la base de modèles) doit être vérifiée; les mécanismes supportés par notre formalisme tels que le sous-typage / l'héritage entre types, la multi-classification, etc. doivent être mis en marche.
2. Implémenter les opérations que nous avons spécifiées dans la section 7.2 (page 199) afin de répondre à nos besoins de manipuler et de gérer des modèles.
3. Développer des outils pour l'édition de modèles, la visualisation graphique des modèles, l'édition de règles de contrôle.

Nous sommes en voie de compléter le développement du prototype et remplir ses objectifs principaux.

7.4 Synthèse

Nous avons montré dans ce chapitre que notre formalisme est implémentable. Le problème crucial de sa mise en œuvre repose sur le stockage de modèles. Notre solution proposée est assez simple. Nous avons montré comment les éléments dans notre formalisme peuvent être codés de la même manière et comment les modèles (et leurs éléments) représentés dans notre formalisme peuvent être codés et stockés dans une base de données relationnelle. Nous avons également spécifié les opérations nécessaires pour nos besoins de manipulation des modèles présentés dans notre formalisme. Nous avons aussi montré la mise en œuvre de ces opérations avec notre système de modélisation/métamodélisation.

Chapitre 8. Conclusion et travaux futurs

La gestion de modèles prend une importance grandissante et suscite de nombreux travaux de recherche dans divers domaines comme la gestion des connaissances, la gestion de métadonnées, les ontologies, la qualité de service et le génie logiciel. Le défi commun, dans ce contexte, repose sur la représentation de modèles (répartis à tous les niveaux de modélisation) et sur le développement d'opérations pour la manipulation de modèles. Notre recherche s'inscrit dans le cadre de la gestion des modèles et porte sur ces deux axes de recherche importants: (i) la représentation de modèles et (ii) les opérations permettant de manipuler les modèles.

Au sujet de la représentation de modèles, nous avons identifié les besoins essentiels concernant la représentation des métamodèles et ceux concernant la représentation de modèles de connaissances du monde réel (cf. le Chapitre 3, page 26). Ces besoins nous permettent de mieux comprendre la complexité du monde à modéliser, d'identifier plus facilement les éléments pour chaque niveau de modélisation, et d'aboutir à une représentation la plus précise mais aussi la plus souple possible.

Plusieurs formalismes de modélisation peuvent être utilisés pour la représentation des modèles. Nos études sur ces formalismes (cf. la section 2.4, page 25, et le Chapitre 4, page 43) ont démontré qu'aucun ne répond complètement à tous les besoins recensés pour la gestion de modèles. De plus, l'uniformité dans la représentation de modèles est une exigence considérable. Ceci nous permet d'éviter l'utilisation d'un ensemble de modèles avec différents formalismes, et, lors des échanges entre les modèles, avec tous les problèmes de traduction et de pertes d'information.

C'est pourquoi nous avons spécifié un nouveau formalisme (cf. le Chapitre 5, page 92) permettant le développement d'un système de gestion des modèles. Ce formalisme est notamment basé sur les réseaux sémantiques permettant la flexibilité dans l'expression ainsi que la simplicité dans la représentation, facilitant ainsi la facilité de mise en œuvre.

Notre formalisme remplit tous les besoins recensés pour la gestion de modèles (cf. la section 6.1.1, page 152, et la section 6.2.1, page 159). Nous avons montré que par rapport aux noyaux réflexifs d'autres formalismes étudiés (cf. la section 6.1, page 152, et le Chapitre 4, page 43), le noyau réflexif de notre formalisme est le seul permettant de

définir tous les formalismes de modélisation. Également, en comparaison avec d'autres formalismes disponibles pour la modélisation du monde réel que nous avons étudiés (cf. la section 6.2.2, page 165, et le Chapitre 4, page 43), notre formalisme permet de représenter, d'une manière la plus fidèle, les situations du monde réel. Il est aussi le seul à permettre de modéliser les situations telles que celles décrites dans le Problème 1 (page 34) et le Problème 2 (page 37).

Il est à souligner que notre formalisme permet l'extensibilité. Plus précisément, le méta-métamodèle et le métamodèle proposés sont tous extensibles. Ils offrent la possibilité de définir et d'ajouter de nouveaux éléments au besoin, en se basant sur notre noyau réflexif.

Concernant les opérations sur modèles, nous avons spécifié les opérations nécessaires permettant la manipulation des modèles dans notre base de modèles (cf. la section 7.2, page 199). Ces opérations et notre formalisme constituent une base solide pour le développement d'un système de gestion de modèles. Le système peut fonctionner avec une mise en œuvre relativement simple au sein d'une base de données relationnelle (cf. le Chapitre 7, page 194).

Ainsi, notre travail de recherche dans le cadre de cette thèse a atteint les quatre objectifs que nous avons présentés dans l'introduction. Également, nous avons implémenté notre formalisme et commencé le développement d'un prototype d'un système de gestion des modèles qui met en œuvre ce formalisme (cf. la section 7.3, page 218). Cette réalisation complémentaire va atteindre l'objectif (5) mentionné dans l'introduction.

Travaux futurs

Comme nous l'avons mentionné dans la section 7.3 (page 218), il nous reste à compléter le développement de notre outil de validation (pour le méta-métamodèle et le métamodèle) et combler ses objectifs principaux.

Le métamodèle présenté dans cette thèse n'est pas complet. Perfectionner ce métamodèle fait aussi partie de nos plans futurs. Plus précisément, ce métamodèle pourrait être étendu afin de supporter d'autres fonctionnalités, par exemple, pour représenter la transformation entre modèles basés sur différents métamodèles.

Quant à la manipulation de modèles, certains sujets n'ont pas encore pu être approfondis tels que la correspondance entre modèles, la fusion de modèles, la

transformation de modèles basés sur différents métamodèles. C'est d'ailleurs pour cette raison que nous souhaitons également spécifier puis implémenter des opérations concernant ces sujets en tenant compte de résultats de travaux de recherche d'autres auteurs sur ces sujets (exemples: [43], [76], [80], [86], [88], [90], [104], [132], [133], [138], [139], [143], [150], [156], [162]).

Pour terminer nous voulons souligner que le formalisme présenté dans cette thèse est le seul, à notre connaissance, qui comble tous les besoins essentiels de la gestion de modèles répartis à tous les niveaux de modélisation. Ce formalisme est non seulement flexible, simple et naturel à concevoir, il est aussi facile à mettre en œuvre. Nous espérons que nos travaux de recherche apporteront des contributions utiles à la réalisation d'un système de gestion des modèles, particulièrement dans le cadre de la gestion de connaissances mais également dans divers autres domaines importants dont les bases de données, les ontologies, la qualité de service et le génie logiciel.

Références

1. Association Française pour l'Intelligence Artificielle: *Dossier Ingénierie des connaissances*. Extrait des N° 34 (pp.: 18 - 66) et 35 (pp.: 13 - 14). Juillet & octobre 1998.
2. Association Française pour l'Intelligence Artificielle: *Dossier Mémoire d'entreprise*. Extrait du N° 36 (pp.: 34 - 64). Janvier 1999.
3. Alagic S. and Bernstein P.A.: *A Model Theory for Generic Schema Management*. Proc. Database Programming Language: 8th International Workshop, DBPL 2001, Frascati, Italy, September 8-10, 2001. Volume 2397/2002, pp.: 228 – 246. Publisher: Springer-Verlag Heidelberg
4. American National Standard: *Conceptual Graph Standard*. 1999. http://www.bestweb.net/~sowa/cg/cgdpans.htm.
5. Ardon S.: *Intégration de l'utilisateur dans la gestion de la Qualité de Service pour des environnements hétérogènes*. Thèse de Doctorat, Université Pierre et Marie Curie, Paris VI. 2002.
6. Atzeni P. and Torlone R.: *Management of multiple models in an extensible database design tool*. In Advances in Database Technology - EDBT'96, 5th Int. Conf. on Extending Database Technology, Springer LNCS'1057, pp. 79-95. 1996.
7. Barclay R. O. and Murray P. C.: *What is knowledge management?*. 1997. http://www.media-access.com/whatis.html
8. Bechhofer S., Goble C., and Horrocks I.: *DAML+OIL is not enough*. International Semantic Web Working Symposium (SWWS). 2001.
9. Bellinger G., Castro D., and Mills A.: *Data, Information, Knowledge, and Wisdom*. http://www.systems-thinking.org/dikw/dikw.htm
10. Berners-Lee T., Hendler J., and Lassila O.: *The Semantic Web*. Scientific American. 2001.
11. Bernstein P.A. and Rahm E.: *Data Warehouse Scenarios for Model Management*. Proc. ER2000, LNCS 1920, Springer-Verlag, pp. 1-15. In International Conference on Conceptual Modeling, 2000.
12. Bernstein P.A.: *Panel: Is Generic Metadata Management Feasible?*. VLDB 2000.
13. Bernstein P.A.: *Generic Model Management - A Database Infrastructure for Schema Manipulation*. Proc. of 9th International Conference on Cooperative Information Systems, CoopIS'01, pp: 1-6. 2001.
14. Bernstein P.: *Applying model management to classical meta data problems*. In Proceedings of the Conference on Innovative Database Research (CIDR), 2003.
15. Bézivin J.: *On Different Interoperability Modes in Software Engineering : the Case of*

Modeling Activities at OMG. Software Engineering'98, Paris. December 1998.

16. Bézivin J. and Gerbé O.: *Towards a Precise Definition of the OMG/MDA framework*. Proceedings of the 16th Int. Conf. on Automated Software Engineering. 2001.
17. Bézivin J., Blay M., Bouzeghoub M., Estublier J., and Favre J.-M.: *Rapport de synthèse, Action spécifique CNRS sur l'Ingénierie Dirigée par les Modèles*. 2005.
18. Bernstein P.A., Levy A. Y., and Pottinger R.A.: *A Vision for Management of Complex Models*. Technical Report MSR-TR-2000-53. June 2000. Short version appeared in SIGMOD Record 29(4), pp.: 55-63. December 2000.
19. Bochmann G., Kerherve B., and Mohamed-Salem M.: *Quality of service management issues in electronic commerce*. Chapter 14, Electronic Commerce technology Trends: Challenges and Opportunities. IIR Publication Inc., MidrangeComputing 2000. pp.: 227-238.
20. Borst W. N.: *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD thesis, University of Twente, Enschede, 1997.
21. Brachman R.J., McGuiness D., Patel-Schneider P., Resnick L., and Borgida A.: *Living with CLASSIC : When and How to Use a KL-ONE-Like Language*. Morgan Kaufmann, 1991.
22. CDIF Technical Committee: *CDIF Framework for Modeling and Extensibility*. January 1994.
23. CDIF Technical Committee: *CDIF – Integrated Meta-model – Presentation Location and Connectivity Subject Area*. June 1996.
24. Cluet S., Delobel C., Siméon J., and Smaga K.: *Your Mediators Need Data Conversion!* SIGMOD 1998. Pages: 177–188. http://www.acm.org/
25. Corcho O., Fernández-López M., Pérez A.G.: *OntoWeb*. D1.1. Technical Roadmap. 2001.
26. Chastel Hervé: *Entité-Association*. Cours de base de données. 2001. http://deptinfo.cnam.fr/Enseignement/CycleA/BDPI/E-A/index.htm
27. Chein M.: *The CORALI Project: From Conceptual Graphs to Conceptual Graphs via Labelled Graphs*. Proc. of the 5th International Conference on Conceptual Structures, Seattle, USA, August 97. Lecture Notes in AI, volume 1257, pp. 65-79. 1997. Springer.
28. Chen Peter Pin-Shan: *The entity-relationship model-toward a unified view of data*. ACM Transactions on Database Systems, Vol. 1(1), pp.: 9-36. 1976.
29. Chu W.W., Zhang G.: *Associations and roles in object-oriented modeling*. ER'97, California, Lecture Notes in Computer Science, Vol. 1331, pp.: 257–270. Springer. 1997.
30. Connolly D., Harmelen F.v., Horrocks I., McGuinness D.L., Patel-Schneider P.F., and Stein L.A.: *DAML+OIL (March 2001) Reference Description*. W3C Note. 18 December

2001. http://www.w3.org/TR/daml+oil-reference

31. Claypool K.T., Rundensteiner E.A: *Sangam: Modeling Transformations For Integrating Now and Tomorrow*. SIGMOD Record (ACM Special Interest Group on Management of Data). 2001.
32. CMPT 354: *Database Systems and Structures*. Summer 1998 - http://www.cs.sfu.ca/CC/354/zaiane/material/notes/contents.html
33. Corazzon R.: *Descriptive and Formal Ontology*. http://www.formalontology.it/index.htm
34. Dahchour M.: *Integrating Generic Relationships into Object Models Using Metaclasses*. PhD thesis, Université catholique de Louvain, Belgium. Mar. 2001
35. Dieng R., Corby O., Giboin A., and Ribière M.: *Methods and Tools for Corporate Knowledge Management*. Rapport de recherche INRIA RR-3485. Short version In Proc. of the Workshop on Knowledge Acquisition, KAW'98, Banff, Canada.
36. Dinh L-A, Gerbé O., and Sahraoui H.: *Un méta-métamodèle pour la gestion de modèles*. The 2nd days on Model Driven Engineering (IDM'06), Lille, France. 2006.
37. Dinh L-A, Gerbé O., and Sahraoui H.: *Gestion de modèles: définitions, besoins et revue de littérature*. In: Actes des premières journées sur l'Ingénierie Dirigée par les Modèles (IDM05), Paris, France. 2005. pp.: 1–15.
38. Dinh T.-L.-A. and Gerbé O.: *A Metamodel for Knowledge Management*. RIVF'04 – Int. Conf. of French-Speaking or Vietnamese Computer Scientists, Hanoi, Vietnam. 2004. pp. 107-112.
39. Dinh T.L.A and Gerbé O.: *Un métamodèle pour la gestion de connaissances*. Cahier du GReSI no 03-03, HEC Montréal. Janvier 2003. http://gresi.hec.ca/cahier.asp
40. Dinh T.-L.-A.: *Métamodèle pour la gestion des modèles*. Partie orale de l'examen pré-doctoral, Université de Montréal. 2003.
41. DINH T.-L.-A.: *Spécification d'un métamodèle pour la représentation d'une mémoire d'entreprise & Prototype de validation*. Mémoire de maîtrise, Institut de la Francophonie pour l'Informatique. 2001.
42. Do H., Melnik S., Rahm E.: *Comparison of schema matching evaluations*. In Proceedings of the 2nd Int. Workshop on Web Databases (German Informatics Society). 2002.
43. Doan A.-H.: *Learning to Map between Structured Representations of Data*. PhD Thesis, University of Washington. 2002.
44. Do H.-H., and Rahm E.: *COMA - A System for Flexible Combination of Schema Matching Approaches*. In VLDB. 2002.
45. Ellis G., Levinson R.: *Proceedings of the Fourth International Workshop on Peirce: A Conceptual Graphs Workbench*. August 19, 1994.

46. Ernst J.: *Introduction to CDIF*. September 1997. http://www.eigroup.org/cdif/intro.html
47. Esch J.: *Contexts and Concepts, Abstraction Duals*. In Conceptual structures : current practices – Proc. of Second International Conference on Conceptual Structures, ICCS'94, College Park, Maryland, USA, August 16-20, 1994. Springer-Verlag. pp. 175-184.
48. Esch J.: *Contexts, Canons and Coreferent Types*. In Conceptual structures : current practices – Proc. of Second International Conference on Conceptual Structures, ICCS'94, College Park, Maryland, USA, August 16-20, 1994. Springer-Verlag. pp. 185-195.
49. Esch J. and Levinson R.: *An Implementation Model for Contexts and Negation in Conceptual Graphs*. In Conceptual structures : applications, implementation, and theory - Proceedings of Third International Conference on Conceptual Structures, ICCS '95, Santa Cruz, CA, USA, August 14-18, 1995. Springer. pp. 247-262.
50. Flatscher R.-G.: *Metamodeling in EIA/CDIF—Meta-Metamodel and Metamodels*. ACM Transactions on Modeling and Computer Simulation (TOMACS). Volume 12, Issue 4. October 2002. pp. 322 – 342.
51. Gandon Fabien. *Ontology Engineering: a survey and a return on experience*. Research Report of INRIA, RR4396, France - March 2002
52. Geoffrion A.M.: *Structured Modeling: Survey and Future Research Directions*, most recently updated June, 1999. Originally published in ORSA CSTS Newsletter, 15:1 (Spring, 1994). Updated May, 1996 and published in ITORMS, 1:3.
53. Gerbé O., Kerhervé B., and Srinivasan U.: *Model Operations for Quality-Driven Multimedia Delivery*. In Contributions to ICCS 2003. 2003.
54. Gerbé O. and Mineau G.: *The CG Formalism as an Ontolingua for Web-Oriented Representation Languages*. In Proceedings of the 10th International Conference on Conceptual Structures. Borovets, Bulgaria. 2002.
55. Gerbé O.: *Un modèle uniforme pour la modélisation et la métamodélisation d'une mémoire d'entreprise*. Thèse de doctorat, Université de Montréal (UdM). Janvier 2000.
56. Girard S., Favre J-M, Muller P-A, and Blanc X. editors: *Actes des premières journées sur l'Ingénierie Dirigée par les Modèles* (IDM05), Paris, France. 2005.
57. Godin R. and Missaoui R.: *Semantic query optimization using inter-relational functional dependencies*. Proc. Hawaii Int. Conf. on System Sciences (HICSS-24), Jan 8-11, 1991, vol. III. pp. 368-375.
58. Gustavsson Henrik, Lings Brian, Lundell Björn, Burman Per, and Palmquist Kristian: *An evaluation of CDIF support for behavioural modeling*. 1999.
59. Gerbé O., Mineau G., and Keller R.: *La métamodélisation et les graphes conceptuels*. Cahier du GReSI no 03-01, HEC Montréal. Janvier 2003.

60. Gottlob G., Schrefl M., and Rock B.: *Extending object-oriented systems with roles.* ACM Trans. Office Inform. Systems 14 (3). 1996. pp. 268–296.

61. Grunstein M. and Barthès J.-P.: *An Industrial View of the Process of Capitalizing Knowledge.* In J. F. Schreinemakers ed, Knowledge Management: Organization, Competence and Methodology, Proc. of the 4th Int. Symposium on the Management of Industrial and Corporate Knowledge (ISMICK'96), Rotterdam, the Netherlands, Ergon, October, 1996. pp. 258-264.

62. Gruber T.-R.: *Towards principles for the design of ontologies used for knowledge sharing. Formal Ontology in Conceptual Analysis and Knowledge Representation.* N. Guarino and R. Poli (Eds.). 1994. Kluwer.

63. Grunstein M.: *La capitalisation des connaissances de l'entreprise, système de production de connaissances.* L'entreprise apprenante et les Sciences de la Complexité. Aix-en-Provence. Mai 1995.

64. Guarino N. and Giaretta P.: *Ontologies and knowledge bases - Towards a terminological clarification. Toward Very Large Knowledge Bases.* Ed. IOS Press. 1995. pp. 25-32.

65. Halpin T.A. and Proper H.A.: *Subtyping and polymorphism in object-role modelling.* Data & Knowledge Engineering 15. 1995. pp. 251-281.

66. Ipsos-Reid & Microsoft Canada Co.: *Knowledge management: a success in most Canadian companies* [Study]. CMA Management. Vol. 75(4). Jun 2001. pp. 8-9.

67. ISO. ISO-IEC 10027*: Information technology - Information Resource Dictionary System (IRDS) – Framework.* ISO/IEC International standard. 1990.

68. ISO/IEC.: *Information Technology - Software Engineering Data Definition and Interchange - Transfer Format. Part 3: Encoding ENCODING.1* (CDIF - Transfer Format - Encoding ENCODING.1). Jan, 1994.

69. ISO/IEC.: *Information Technology - Software Engineering, Data Definition and Interchange - Integrated Meta-model, Part 1 : Foundation Subject Area.* Mai 1996.

70. ISO/IEC.: *FCD 15476-2: Information Technology - CDIF Semantic Metamodel - Part 2: Common.* 1998-01-05. http://www.omg.org/docs/cdif/98-06-08.pdf

71. Jézéquel J.-M., Gérard S., Mraidha C., and Baudry B.: *Approche unificatrice par les modèles.* Action spécifique CNRS sur l'Ingénierie Dirigée par les Modèles. 2005

72. Kerhervé B. and Gerbé O.: *Model Management for Quality of Service Support.* Proc. of the 14th Int. Conf. on Soft. & Syst. Engineering and their Applications, Vol. 1, France. 2001.

73. Knowledge Management Forum: *What is Knowledge Management.* KM Forum Archives -- The Early Days. Last updated: 8/3/2002. http://www.km-forum.org/what_is.htm

74. Kocura Pavel: *Semantics of Attribute Relations in Conceptual Graphs*. In Conceptual structures : logical, linguistic, and computational issues – Proc. of 8th Int. Conf. on Conceptual Structures, ICCS 2000, Darmstadt, Germany, 2000. Springer. pp. 235-248.

75. Kosky A.S.: *A Formal Model for Databases with Applications to Schema Merging*. 1994.

76. Kosky A.: *Transforming Databases with Recursive Data Structures*. PhD Thesis, University of Pennsylvania. 1996.

77. Kristensen B.B.: *Object-Oriented Modeling with Roles*. Proc. of the 2nd Int. Conf. on Object-Oriented Information Systems (OOIS'95), Ireland. 1995.

78. Kühne T.: *What is a Model ?*. Proceedings of Language Engineering for Model Driven Software Development. Dagstuhl Seminare 04101. 29.02-05.03, 2004.

79. Levinson R. and Ellis G.: *Multi-level hierarchical retrieval*. Knowledge Based Systems, vol. 5(3). September 1992. pp. 233-244.

80. Lemesle R.: *Techniques de Modélisation et de Méta-modélisation*. Thèse de Doctorat, Université de Nantes. 2000.

81. Lemesle R.: *Transformation Rules Based on Meta-modeling*. In Proceedings of EDOC'98, pages 113-122, La Jolla, CA, November 1998.

82. Lehmann Fritz and Rodin Ervin Y.: *Semantic Networks in Artificial Intelligence*. Pergamon Press, Oxford. 1992. Also published as a special issue of Computers and Mathematics with Applications, volume 23, number 2-9.

83. Lesca Humbert: *Veille Stratégique : Capitalisation des Signaux faibles et Développement d'une intelligence collective dans l'entreprise*. In Actes de la journée d'étude ADBS, Grenoble, France. 1997. pp. 11-20.

84. Loos P.: *Capture More Data Semantic Through The Expanded Entity-Relationship Model* (PERM). University of Münster, Institute of Business Informatics, Germany. 1997.

85. Lutz Carsten: *The Complexity of Description Logics with Concrete Domains*. Ph.D Theis, Internetseiten der Hochschulbibliothek, Hamburg. 2002.

86. Madhavan J., Bernstein P.A., Domingos P., and Halevy A.Y.: *Representing and Reasoning About Mappings between Domain Models*. 18th National Conference on Artificial Intelligence (AAAI 2002), Edmonton, Canada. 2002

87. Madhavan J., Bernstein P.A., and Rahm E.: *Generic schema matching with cupid*. In Proc. of the 27th International Conferences on Very Large Databases. 2001. pp. 49 - 58.

88. McBrien P. and Poulovassilis A.: *Data Integration by Bi-Directional Schema Transformation Rules*. In: 19th Int. Conf. on Data Engineering, ICDE'03. March 2003.

89. McBrien P.J. and Poulovassilis A.: *A uniform approach to inter-model transformations*. In Advanced Information Systems Engineering, 11th International Conference CAiSE'99,

volume 1626 of LNCS, pp. 333-348. Springer-Verlag, 1999.

90. Melnik S.: *Generic Model Management*. Ph.D Thesis. Lecture Notes in Computer Science, Springer. 2004
91. Melnik S., Rahm E., and Bernstein P.A.: *Rondo: A Programming Platform for Generic Model Management*. Proc. SIGMOD 2003. pp. 193 – 204.
92. Mihoubi H., Simonet A., Simonet M.: *Towards a declarative approach for reusing domain ontologies*. Information Systems, Vol. 23, Issue 6. September 1998. pp. 365-381.
93. Miller R.J., Haas L.M., Hernández M.A.: *Schema Mapping as Query Discovery*. VLDB 2000. pp. 77-88.
94. Mineau G.W., Missaoui R., Godinx R.: *Conceptual modeling for data and knowledge management*. Data & Knowledge Engineering, Vol. 33, Issue 2, May 2000. pp. 137-168.
95. Miras Dimitrios: *A Survey on Network QoS Needs of Advanced Internet Applications*. Internet2-QoS Working Group. November 2002.
96. Mitra P., Wiederhold G., and Kersten M.: *A Graph-Oriented Model for Articulation of Ontology Interdependencies*. In: Proceedings Conference on Extending Database Technology 2000 (EDBT'2000), Konstanz, Germany, 2000.
97. Morand B.: *Modèle de données Entité-Association*. 1998. http://www.iutc3.unicaen.fr/info/format/prog/info/acsi/donnees1/table.htm
98. Mugnier M.L., Chein M.: *Polynominal algorithms for projection and matching*. In Conceptual structures : theory and implementation – Proc. of 7th annual Workshop on Conceptual Graphs, Las Cruces, NM, USA, July 1992. Springer-Verlag. pp. 239-251.
99. Mugnier M.-L.: *Knowledge Representation and Reasonings Based on Graph Homomorphism*. RR-LIRMM 00-098. 2000.
100. Mullins Craig S.: *What is knowledge and can it be managed ?*. http://www.tdan.com/i008fe03.htm
101. Mylopoulos J., Borgida A., Jarke M., and Koubarakis M.: *Telos: Representing knowledge about information systems*. In ACM Trans. on Inform. Systems. Vol. 8(4). 1990. pp. 325-362.
102. Nahrstedt K.: *End-to-End QoS Guarantees in Networked Multimedia Systems*. ACM Computing Surveys Journal. Vol. 27, num. 4. December 1995. pp. 613-616.
103. Nonaka I.: *The knowledge-creating company*. Harvard Business Review. Nov-Dec. 1991. pp. 96-104.
104. Object Management Group (OMG.): *MDA Guide Version 1.0.1*. Joaquin Miller & Jishnu Mukerji ed., (2003)
105. OMG.: *Meta Object Facility (MOF) 2.0 Core Specification*. OMG Adopted Specification,

formal/06-01-01. 2006.

106. OMG.: *Meta Object Facility (MOF) 2.0 Core Specification*. OMG Adopted Specification, ptc/04-10-15. 2004.
107. OMG.: *Meta Object Facility (MOF) 2.0 Core Specification*. OMG Adopted Specification, ptc/03-10-04. 2003.
108. OMG.: *Meta Object Facility (MOF), version 1.4*, formal/02-04-03. April 2002.
109. OMG.: *UML 2.1.1 Infrastructure Specification*, formal/07-02-06. 2007.
110. OMG.: *UML 2.1.1 Superstructure Specification*, formal/07-02-05. 2007.
111. OMG.: *UML Diagram Interchange Specification*, v1.0, formal/06-04-04. 2006.
112. OMG.: *UML OCL Specification*, v2.0, formal/06-05-01. 2006.
113. OMG.: *UML 2.0 Infrastructure Specification*. formal/05-07-05. 2005.
114. OMG.: *UML 2.0 Superstructure Specification*. formal/05-07-04. 2005.
115. OMG.: *UML 2.0 Diagram Interchange convenience document*. ptc/05-06-04
116. OMG.: *UML 2.0 OCL convenience document*. ptc/05-06-06. 2005.
117. OMG.: *Unified Modeling Language: Infrastructure, Version 2.0*. OMG Adopted Specification, ptc/04-10-14. 2004.
118. OMG.: *Unified Modeling Language: Infrastructure. Version 2.0*. March 2003. http://www.omg.org/docs/ad/03-03-01.pdf
119. OMG.: *Unified Modeling Language: Superstructure. Version 2.0*. OMG Adopted Specification, ptc/03-08-02. August 2003. http://www.omg.org/docs/ptc/03-08-02.pdf
120. OMG.: *UML 2.0 Diagram Interchange draft adopted specification*. July 2003. http://www.omg.org/docs/ptc/03-07-03.pdf
121. OMG.: *UML 2.0 OCL 2nd revised submission, version 1.6*. OMG Document ad/2003-01-07. 2003.
122. OMG.: *Object Constraint Language Specification, version 1.1*. September 1997.
123. OMG.: *Software Process Engineering Metamodel. Version 1.0*. formal/02-11-14. November 2002.
124. OMG.: *Common Warehouse Metamodel (CWM) Specification - Version 1.1*, Volume 1. March 2003.
125. OMG.: *Common Warehouse Metamodel (CWM) Specification - Volume 2*, Extensions. February 2002.
126. OMG.: *XML Metadata Interchange (XMI). Version 2.0*, formal/03-05-02. May 2003.
127. Pan J.-Z., Horrocks I.: *Metamodeling Architecture of Web Ontology Languages*. International Semantic Web Working Symposium (SWWS). 2001.
128. Pan Dong: *The Application of Design Patterns in Knowledge Inference Engine*. Master

Thesis, University of Calgary. 1998.

129. Paquette G.: *Modélisation des connaissances et des compétences : un langage graphique pour concevoir et apprendre*. Sainte-Foy : Presses de l'Université du Québec. 2002.
130. Peltier M., Bézivin J., and Guillaume G.: *MTRANS: A general framework, based on XSLT, for model transformations*. WTUML - Workshop on Transformations in UML. 2001.
131. Pernici B.: *Objects with Roles*. Proceedings ACM-IEEE Conference of Office Information Systems (COIS). Boston 1990. pp. 205 – 215.
132. Pottinger R.A. and Bernstein P.A.: *Creating a Mediated Schema Based on Initial Correspondences*. IEEE Data Engineering Bulletin, Vol. 25, No.3. Sept. 2002. pp. 26-31.
133. Pottinger R.A. and Bernstein P.A.: *Merging Models Based on Given Correspondences*. University of Washington Technical Report UW-CSE-03-02-03. February 2003.
134. Poole J. and Campbell J.A.: *A Novel Algorithm for Matching Conceptual and Related Graphs*. In Conceptual structures : applications, implementation, and theory - Proceedings of Third International Conference on Conceptual Structures, ICCS '95, Santa Cruz, CA, USA, August 14-18, 1995. Springer. pp. 293-307.
135. Pottinger R.A.: *An Extensible System for Merging Two Models* (Doctoral Poster). The 28th International Conference on Very Large Databases, Hong Kong, August 20-23, 2002.
136. Prediger S.: *Nested Concept Graphs and Triadic Power Context Families: A Situation-Based Contextual Approach*. In Conceptual structures : logical, linguistic, and computational issues – Proc. of 8th International Conference on Conceptual Structures, ICCS 2000, Darmstadt, Germany, August 14-18, 2000. Springer. pp. 249-262.
137. Prediger S.: *Simple Concept Graphs: A Logic Approach*. In Conceptual Structures: Theory, Tools and Applications – Proc. of 6th International Conference on Conceptual Structures, ICCS'98, Montpellier, France, August 1998. Springer. pp. 225-239.
138. Rahm E. and Bernstein P.A.: *On Matching Schemas Automatically*. http://www.research.microsoft.com/research/db/ModelMgt/ (Short version: A survey of approaches to automatic schema matching. The International Journal on Very Large Data Bases. Vol. 10, Number 4/December 2001, pp.: 334 - 350. Springer-Verlag Heidelberg)
139. Revault N., Blanc X., and Perrot J-F.: *Traduction de méta-modèles*. In Langages et Modèles à Objets (Lmo'01), Vol 7 - n° 1-2/2001, R. Godin & I. Borne (ed), L'Objet - logiciel, bases de données, réseaux, pp. 95-111, Le Croisic, France, Janv (29-31), 2001, Hermès Science Publications, Paris.
140. Revault N., Sahraoui H.A., Blain G., and Perrot J.-F.: *A Metamodeling technique: The MétaGen system*. In Tools 16: Tools Europe'95, Prentice Hall, Versailles, France. 1995. pp. 127-139.

141. Rumbaugh J., Jacobson I., and Booch G.: *The Unified Modeling – User Guide*. Addison wesley. 1999.

142. Rumbaugh J., Jacobson I., and Booch G.: *The Unified Modeling – Language Reference Manual*. Addison wesley. 1998.

143. Sahraoui H.A.: *Application de la méta-modélisation à la génération d'outils de conception et de mise en œuvre de bases de données*. Thèse de Doctorat, Université P. et M. Curie (Paris 6), Paris, France. 1995.

144. Sowa J.-F.: *Building, Sharing, and Merging Ontologies*. Last Modified: 08/25/2001. http://www.jfsowa.com/ontology/ontoshar.htm

145. Sowa J.F.: *Knowledge Representation: Logical, Philosophical and Computational Foundations*. BooksCole. 2000.

146. Sowa J.F. and Borgida A.: *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA. 1991.

147. Sowa J.F.: *Using a lexicon of canonical graphs in a semantic interpreter*. In: M.W. Evens (Ed.), Relational Models of the Lexicon: Representing Knowledge in Semantic Networks, Cambridge University Press, Cambridge; 1988, pp. 113-137.

148. Sowa J.F.: *Conceptual Structures – Information processing in mind and machine*. Addison wesley 14472. 1984.

149. Sowa J.F.: *Semantic Networks*. http://www.jfsowa.com/pubs/semnet.htm

150. Sprinkle J.M.: *Metamodel driven model migration*. PhD Thesis, Vanderbilt University. August 2003.

151. Sahraoui H.A., Revault N., Blain G., and Perrot J-F.: *Un outil pour la conception de bases de données à objets*. In Revue Technique et Science Informatique (TSI), Vol. 17, n° 7. 1998. pp. 839-868

152. Schneider D.: *Modélisation de la démarche du décideur politique dans la perspective de l'intelligence artificielle*. Thèse de doctorat, Université de Genéve, 1994.

153. Stachowiak H.: *Gedanken zu einer allgemeinen Theorie der Modelle; Studium Generale*. Springer. 1965.

154. Steimann F.: *On the representation of roles in object-oriented and conceptual modelling*. Data & Knowledge Engineering 35. 2000. pp. 83-106.

155. Stein E.W.: *Organization memory: Review of concepts and recommendations for management*. International Journal of Information Management, Volume 15, Issue 1. February 1995. pp. 17-32.

156. Stumme G. and Madche A.: *FCA-Merge: Bottom-up merging of ontologies*. In 7th Intl. Conf. on Artificial Intelligence (IJCAI '01), Seattle, WA, 2001. pp. 225-230.

157. Stumme G.: *Using ontologies and formal concept analysis for organizing business knowledge*. In Proc. Referenzmodellierung 2001.

158. Theodorakis M., Analyti A., Constantopoulos P., and Spyratos N.: *A theory of contexts in information bases*. Information Systems, Volume 27, Issue 3. May 2002. pp. 151-191.

159. Tan J. and Zaslavsky A.: *Domain-Specific Metamodels for Heterogeneous Information Systems*. Proc. of the 36th Hawaii Int. Conf. on System Sciences (HICSS'03). 2003.

160. Terzi E., Vakali A., and Hacid M-S.: *Knowledge Representation, Ontologies, and the Semantic Web*. In Web Technologies and Applications - Proceedings of 5th Asia-Pacific Web Conference, APWeb 2003, Xian, China, April 23-25, 2003. Lecture Notes in Computer Science, Vol. 2642 / 2003. pp. 382 - 387. Springer-Verlag.

161. Ullman J.D.: *Principe of Database and Knowledge-Base Systems*. Volume 1, Standford university, Computer Science Press. Inc. 1988

162. Varro D. and Pataricza A.: *Mathematical Model Transformations*. Technical Report, Department of Measurement and Information Systems, Budapest University of Technology and Economics. May, 2001.

163. Vogel A., Kerhervé B., Bochmann G.v., and Gecsei J.: *Distributed multimedia applications and quality of service: a survey*. IEEE Journal of Multimedia Systems. Volume 2, Issue 2, Summer 1995. pp. 10 -19

164. Vasconcelos J., Kimble C., Gouveia F., and Kudenko D.: *A Group Memory System for Corporate Knowledge Management: An Ontological Approach*. 2000.

165. World Wide Web Consortium (W3C.): *Extensible Markup Language (XML) 1.1 (Second Edition)*. W3C Recommendation 16 August 2006. http://www.w3.org/TR/xml11/#elemdecls.

166. W3C.: *XML Schema 1.1 Part 1: Structures*. W3C Working Draft. 31 August 2006. http://www.w3.org/TR/xmlschema11-1/

167. W3C.: *XML Schema 1.1 Part 2: Datatypes*. W3C Working Draft 17 February 2006. http://www.w3.org/TR/xmlschema11-2/

168. W3C.: *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C Recommendation 04 February 2004- http://www.w3.org/TR/REC-xml

169. W3C.: *XML Schema Part 0: Primer Second Edition*. W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-0/

170. W3C.: *XML Schema Part 1: Structures Second Edition*. W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-1/

171. W3C.: *XML Schema Part 2: Datatypes Second Edition*. W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-2/

172. W3C.: *XML Schema: Formal Description*. W3C Working Draft, 25 September 2001. http://www.w3.org/TR/xmlschema-formal/

173. W3C.: *XML Schema Requirements*. W3C Note 15 February 1999. http://www.w3.org/TR/NOTE-xml-schema-req

174. W3C.: *XSL Transformation*. http://www.w3.org/TR/xslt

175. W3C.: *RDF Primer*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-primer/

176. W3C.: *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-concepts/#section-triples

177. W3C.: *RDF Semantics*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-mt/

178. W3C.: *RDF/XML Syntax Specification (Revised)*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-syntax-grammar/

179. W3C.: *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-schema/

180. W3C.: *OWL Web Ontology Language Overview*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-features/

181. W3C.: *OWL Web Ontology Language Guide*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-guide/

182. W3C.: *OWL Web Ontology Language Reference*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-ref/

183. W3C.: *OWL Web Ontology Language Semantics and Abstract Syntax*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-semantics/

184. Wieringa R., De Jonge W., and Spruit P.: *Using dynamic classes and role classes to model object migration*. Theory and Practice of Object Systems, Vol. 1(1), 1995, pp. 61–83

185. William R.C., Walter H., and Canning P.S.: *Inheritance is not subtyping*. Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. San Francisco, California, United States 1989, pp. 125 - 135

186. Willems M.: *Projection and Unification for Conceptual Graphs*. In Conceptual structures : applications, implementation, and theory – Proc. of Third Int. Conf. on Conceptual Structures, ICCS '95, Santa Cruz, CA, USA, August 14-18, 1995. Springer. pp. 278-292.

187. *Ontological Knowledge and Linguistic Coding*. Workshop at the 25th annual meeting of the German Linguistics Society, Munich, February 2003.

188. Yan Ling-Ling, Miller R.J., Haas Laura M., and Fagin Ronald: *Data-Driven Understanding and Refinement of Schema Mappings*. SIGMOD 2001, pp. 485 – 496.

Annexe I. Évaluation des formalismes étudiés

Cette annexe présente nos évaluations des formalismes étudiés en fonctions des besoins (cf. les Table I-1 et Table I-2) que nous avons recensés pour la gestion de modèles (cf. le Chapitre 3, page 26). Les formalismes choisis sont: Réseau sémantique, sNets, Graphes conceptuels, Modèle uniforme des GCs, Entité-Association, CDIF, UML, MOF, XML-XML Schema, RDF-RDFS et OWL.

BesoinM3 1 : Typage entre éléments de deux niveaux de modélisation consécutifs

BesoinM3 2 : Passage entre niveaux types et instances

BesoinM3 3 : Distinction entre différentes notions de conformité au niveau méta

BesoinM3 4 : Hiérarchisations entre types

BesoinM3 5 : Contraintes de cardinalité min/max pour un méta-lien

BesoinM3 6 : Distinction entre modèles de différentes natures

BesoinM3 7 : Liens avec les modèles

Table I-1 : Liste des besoins essentiels pour M3

BesoinM2 1 : Typage entre éléments du niveau M1

BesoinM2 2 : Typage entre éléments de deux niveaux de modélisation consécutifs

BesoinM2 3 : Passage entre niveaux types et instances

BesoinM2 4 : Distinction entre différentes notions de conformité

BesoinM2 5 : Hiérarchisations entre types

BesoinM2 6 : Multi-classification et Connaissance partielle

BesoinM2 7 : Objets, Rôles, Types d'objets, Types de rôles

BesoinM2 8 : Rapports entre objets et rôles / types de rôles

BesoinM2 9 : Classification dynamique et classification statique

BesoinM2 10 : Contraintes de l'arité min/max

BesoinM2 11 : Contraintes de cardinalité totale min/max pour un type relationnel

BesoinM2 12 : Contraintes de cardinalité locale min/max pour un type relationnel

BesoinM2 13 : Contraintes de l'itération min/max pour un type relationnel

BesoinM2 14 : Contraintes de cardinalité min/max pour un méta-lien

BesoinM2 15 : Types relationnels entre types et/ou instances

BesoinM2 16 : Distinction entre modèles de différentes natures

BesoinM2 17 : Liens avec les modèles

BesoinM2 18 : Liens entre modèles au niveau M1

Table I-2 : Liste des besoins essentiels pour M2

Notations dans les tables d'évaluation:

«+» : remplir le besoin correspondant;

«+-» : remplir partiellement le besoin correspondant et à développer pour le remplir;

«-» : ne pas remplir et à développer pour remplir le besoin correspondant

I.1 Réseau sémantique

Les réseaux sémantiques permettent la représentation de toute sorte d'information mais il manque la modularité et la structuration de la sémantique de ces informations [80]. Ainsi, les réseaux sémantiques ne répondent à aucun besoin compris dans notre liste des besoins essentiels pour la gestion de modèles. À noter que dans les réseaux sémantiques, seuls les liens binaires sont permis.

I.2 sNets

I.2.1 Évaluation suivant les besoins pour M3 (cf. Table I-1, page i-1)

BesoinM3		**Méta-métamodèle des sNets** [80]
1	+-	sNets supporte les entités (*nœuds*), méta-entités (*méta-nœuds*), relations (*liens*) binaires, métarelations (*méta-liens*) binaires mais non les relations (*liens*) entre relations ou entre entités avec relations.
2	+	sNets permet le passage entre niveaux types et instances.
3	+	Chaque entité (ou modèle) s'attache à sa méta-entité (son métamodèle) par un lien de conformité de type `meta (sem)`
4	+-	sNets permet l'héritage multiple entre méta-entités mais non la hiérarchisation entre métarelations.
5	+-	sNets permet de représenter les contraintes de cardinalité maximale d'une métarelation mais non celle de cardinalité minimale.
6	+-	Absence des notions: modèles structurels, modèles de conditions.
7	+-	sNets permet la représentation de deux métarelations inversées, la relation sémantique (`sem`) entre un modèle et son métamodèle et l'extension (`extends`) entre modèles. Il ne dispose pas de différents types de liens (la jointure, l'intersection, la différence, l'inférence, la restriction) entre modèles pour la gestion de modèles.

Table I-3 : Évaluation du méta-métamodèle des sNets suivant les besoins pour M3

I.2.2 Évaluation suivant les besoins pour M2 (cf. Table I-2, page i-1)

Puisque le noyau des sNets supporte les BesoinM3 1 et BesoinM3 5, tous deux

partiellement, aucun métamodèle conforme à ce noyau ne peut répondre entièrement au besoin «Typage entre éléments de deux niveaux de modélisation consécutifs» (BesoinM2 2) ni au besoin «Contraintes de cardinalité min/max pour un méta-lien» (BesoinM2 14) pour M2. Ainsi, aucun métamodèle conforme à ce noyau ne répond à tous les besoins pour M2.

I.3 Graphes conceptuels

I.3.1 Évaluation suivant les besoins pour M2 (cf. Table I-2, page i-1)

BesoinM2		Graphes conceptuels (GCs)
1	+	Les GCs supportent les notions «type» et «instance d'un type». Ils distinguent les types de concepts (*types non relationnels*) et les types de relations (*types relationnels*), aussi les concepts (*occurrences*) et les relations (*relations*).
2	+-	Concernant le support de nos notions de «*nœuds*» et «*liens binaires*»: présence de relations (*liens*) entre concepts (*nœuds*); mais absence de relations entre relations ou entre relations avec concepts.
3	+	Les GCs permettent de traiter un élément comme un type à un niveau et comme une instance à un autre niveau [55].
4	-	Les GCs ne prennent pas en compte différentes notions de conformité et ne définissent pas explicitement de liens de conformité.
5	+	Les types non relationnels et les relationnels sont organisés en hiérarchie.
6	+	Les GCs autorisent la multi-classification et la connaissance partielle [55].
7	+-	Sowa a distingué ([145], [148]) les types naturels (équivalente à notre notion «*type d'objets*») et les types de rôles mais : les types naturels et types de rôles coexistent dans une même hiérarchie de types; aucune différence syntaxique entre le côté des types naturels et celui des types de rôles; les types de rôles sont considérés comme des sous-types des types naturels.
8	+-	Des critères identifiés, seuls les critères (ii), (vii) et (viii) sont répondus. Un rôle est joué par un seul joueur à chaque moment (critère (ii)). Les types de rôles peuvent dépendre des relations (critère (vii)). Différents types de rôles peuvent partager des structures ou comportements communs (critère (viii)).
9	-	Absence de la distinction entre les classifications dynamique et statique.
10	-	Aucun de ces cinq besoins n'est considéré dans les GCs. Concernant les contraintes de cardinalités, les GCs disposent simplement du quantificateur existentiel (identifié par «∃») et universel (identifié par «∀»), ne permettant pas de modéliser les situations comme celle du Problème 1 (page 34).
11	-	
12	-	
13	-	
14	-	

15	+-	Les GCs permettent de représenter seulement certains types relationnels entre types et/ou instances [55] mais ne permettent pas encore de représenter ceux présentés par le Problème 2 (page 37).
16	+-	Les GCs disposent de la notion de contexte (comme des graphes emboîtés) mais ne traitent pas explicitement des modèles et de leurs relations.
17	+-	
18	-	

Table I-4 : Évaluation des Graphes conceptuels suivant les besoins pour M2

I.3.2 Évaluation suivant les besoins pour M3 (cf. Table I-1, page i-1)

BesoinM3		Graphes conceptuels (GCs)
1	+-	Comme pour le BesoinM2 2 pour M2.
2	+	Comme pour le BesoinM2 3 pour M2.
3	-	Comme pour le BesoinM2 4 pour M2.
4	+	Comme pour le BesoinM2 5 pour M2.
5	-	Comme pour le BesoinM2 14 pour M2.
6	+-	Comme pour le BesoinM2 16 pour M2.
7	+-	Comme pour le BesoinM2 17 pour M2.

Table I-5 : Évaluation des Graphes conceptuels (GCs) suivant les besoins pour M3

I.4 Modèle uniforme

I.4.1 Évaluation suivant les besoins pour M2 (cf. Table I-2, page i-1)

BesoinM2		Modèle uniforme des GCs
1	+	Comme dans le cas des GCs, pour les mêmes raisons.
2	+-	
3	+	
4	-	Un concept (*occurrence*) est associé à son type par la relation (*lien*) `type`. Mais absence de la notion de conformité entre modèles et métamodèles.
5	+	Les types non relationnels et les relationnels sont organisés en hiérarchie.
6	+	La multi-classification et la connaissance partielle sont supportées [55].
7	-	Comme le modèle uniforme des GCs ne tient pas compte de la notion de rôle (dans notre contexte), ces besoins relatifs à rôles n'y sont pas applicables.
8	-	
9	-	Comme dans le cas des GCs, pour les mêmes raisons.
10	-	Le modèle uniforme des GCs permet, parmi ces besoins, de représenter les contraintes de cardinalité retenues plutôt par les BesoinM2 12 et BesoinM2 14. Donc il ne permet pas de modéliser les situations comme celle du Problème 1 (page 34).
11	-	
12	+	
13	-	
14	+	
15	+-	Comme dans le cas des GCs, pour les mêmes raisons.

16	+-	Le modèle uniforme des GCs dispose des types pour représenter explicitement les contextes ainsi que différents types de graphes (les graphes de définition, les graphes de restrictions) mais ne traite pas explicitement des modèles et de leurs relations.
17	+-	
18	-	

Table I-6 : Évaluation du Modèle uniforme des GCs suivant les besoins pour M2

I.4.2 Évaluation suivant les besoins pour M3 (cf. Table I-1, page i-1)

BesoinM3		Modèle uniforme des GCs
1	+-	Comme pour le BesoinM2 2 pour M2.
2	+	Comme pour le BesoinM2 3 pour M2.
3	-	Comme pour le BesoinM2 4 pour M2.
4	+	Comme pour le BesoinM2 5 pour M2.
5	+	Comme pour le BesoinM2 14 pour M2.
6	+-	Comme pour le BesoinM2 16 pour M2.
7	+-	Comme pour le BesoinM2 17 pour M2.

Table I-7 : Évaluation du Modèle uniforme des GCs suivant les besoins pour M3

I.5 Entité-Association - Évaluation suivant les besoins pour M2 (cf. Table I-2, page i-1)

BesoinM2		Entité-Association
1	+-	La représentation de connaissances se fait uniquement au niveau des types et non au niveau des instances. Donc les besoins BesoinM2 3, BesoinM2 4, BesoinM2 6 et BesoinM2 15 ne sont pas applicables.
2	+-	Pour la notion de «*méta-lien binaire*», dans le métamodèle Entité-Association: présence de métarelations (*méta-liens*) entre méta-entités (*méta-nœuds*) mais non entre métarelations ou entre méta-entités avec métarelations.
3	-	*Non applicable.*
4	-	
5	+-	Absence de la hiérarchisation entre types relationnels.
6	-	*Non applicable.*
7	+-	Dans la famille de ce modèle ([28], [65]), les rôles n'ont pas de propriétés propres à lui [154] comme ceux dans notre contexte. Alors aucun de ces besoins n'est complètement rempli.
8	+-	
9	-	Entité-Association ne considère pas ce besoin.
10	-	Entité-Association permet, parmi ces besoins, de représenter seulement les contraintes de cardinalité retenues par les BesoinM2 12 et BesoinM2 14. Donc Entité-Association ne permet pas de modéliser les situations comme celle dans le Problème 1 (page 34).
11	-	
12	+	
13	-	
14	+	

15	-	*Non applicable.*
16	-	Le formalisme Entité-Association ne supporte pas la notion de modèle, donc ces besoins n'y sont pas appliqués.
17	-	
18	-	

Table I-8 : Évaluation de Entité-Association suivant les besoins pour M2

I.6 CDIF - Évaluation suivant les besoins pour M3 (cf. Table I-1, page i-1)

BesoinM3		CDIF
1	+-	Pour la notion de *méta-lien binaire*, CDIF permet les métarelations (*méta-liens*) binaires entre méta-entités (*méta-nœuds*) mais non les métarelations entre métarelations, ou entre méta-entités avec métarelations.
2	+	
3	-	CDIF ne définit pas explicitement de liens de conformité entre les objets (entités, relations, attributs, ...) et les méta-objets (méta-entités, métarelations, méta-attributs, ...) ni entre modèles et métamodèles.
4	+	Le sous-typage multiple pour les méta-entités et les métarelations
5	+	
6	-	CDIF dispose de la notion de « zones d'intérêt » (`SubjectArea`) pour organiser les éléments mais ne dispose d'aucune métarelation entre `SubjectArea`. Donc, aucun moyen pour distinguer les modèles de différentes natures comme modèles, métamodèles, modèles structurels, modèles de conditions.
7	+-	Aucune métarelation entre `SubjectArea`, aucun moyen pour expliciter le lien de conformité entre un modèle et son métamodèle, ni pour représenter les liens entre les « zones d'intérêt » (`SubjectArea`). Si la métarelation `IsUsedIn` (relie `CollectableMetaObject` à `SubjectArea`) signifie que les méta-objets collectables (`CollectableMetaObject`) sont utilisés dans des « zones d'intérêt », alors quel lien représente dans quelle « zone d'intérêt » les méta-objets collectables sont définis ? ou manque-t-il dans CDIF la métarelation pour indiquer qu'un méta-objet collectable est défini dans une « zone d'intérêt » ? Mais si la métarelation `IsUsedIn` signifie la métarelation de définition, alors elle représente aussi qu'un méta-objet collectable peut être défini dans plusieurs « zones d'intérêt ». Donc dans quelle « zone d'intérêt » ce méta-objet collectable est-il défini pour la première fois ?

Table I-9 : Évaluation de CDIF suivant les besoins pour M3

I.7 UML - Évaluation de UML selon nos besoins pour M2 (cf. Table I-2, page i-1)

BesoinM2		UML
1	+	
2	+-	Tel qu'expliqué dans l'évaluation du MOF selon le BesoinM3 1 (cf. la section I.8, page i-8), MOF ne supporte pas entièrement notre notion de «*lien binaire*»: MOF permet les liens entre *nœuds* mais non entre liens ni entre liens avec *nœuds*. Donc dans le métamodèle UML défini par MOF, il n'existe aucun *lien* entre *liens* ou entre *liens* avec *nœuds*.
3	+	UML permet le passage entre niveaux types et instances.
4	-	La relation de conformité (`instanceOf`) entre éléments et méta-éléments est distinguée avec l'instanciation locale entre instances et types au M1, mais est identifiée identique à celle entre modèles et métamodèles (cf. la Figure 28, page 59, et la Figure 29, page 59).
5	+	
6	+	
7	+	UML distingue les rôles statiques et dynamiques. Avec les caractéristiques du modèle de rôles dans UML2.x, nous pouvons considérer que UML2.x répond aux deux besoins.
8	+	
9	-	UML n'explicite pas la différence entre les classifications statique et dynamique et les représente par un même type de généralisation entre classificateurs (classes).
10	+-	Il est possible de spécifier les agrégations avec des contraintes de cardinalité. Ainsi le cas «Contraintes de l'arité min/max sur la composition d'un type d'objets» du BesoinM2 10 peut être supporté.
11	-	UML permet, parmi ces besoins, de représenter seulement les contraintes de cardinalité retenues par les BesoinM2 12 et BesoinM2 14. Donc UML ne permet pas de modéliser les situations comme celle du Problème 1 (page 34).
12	+	
13	-	
14	+	
15	+-	Dans UML2.0, les relations sont définies au niveau des classes et applicables au niveau des instances. Nous trouvons deux moyens permettant de supporter le besoin «Types relationnels entre types et/ou instances» : (i) des valeurs par défaut pour des attributs d'une classe; (ii) le power-type (`PowerTypes`). L'usager peut spécifier des valeurs concrètes à certains attributs d'une classe lors du raffinement de celle-ci. Cependant, uniquement des relations binaires sont représentables comme attributs d'une classe. Avec l'application des

		power-types, une classe est divisible en sous-classes plus spécifiques, chaque partition correspond à un power-type. Cependant, ce mécanisme ne convient pas lorsque chaque partition d'une classe se fait pour divers power-types combinés, tels que le cas décrit dans le Problème 2 (page 37), où une relation n-aires est présente.
16	+-	Concernant la modularité, UML supporte plusieurs notions comme les paquetages (`Package`), les composants (`Component`) et les modèles (`Model`), sans toutefois expliciter la notion de modèles structurels et celle de modèles de conditions.
17	+-	UML permet de représenter différents types de liens identifiés pour répondre à ce besoin. Par exemples, les liens de *contenant* entre un paquetage (modèle) et ses éléments; les liens d'*accès* (`import` et `merge`) entre paquetages pour la réutilisation des éléments et pour l'extension des paquetages. Cependant, dans UML, il faut clarifier la relation sémantique entre modèles et métamodèles qui n'est pas proprement identique à la relation `instanceOf` entre un élément et son méta-élément. Aussi, UML ne permet pas de représenter d'autres types de liens entre modèles cités dans le BesoinM2 17 comme la jointure, l'intersection, la différence, l'inférence, la restriction.
18	+	Concernant le développement de logiciels, UML considère plusieurs types de liens entre paquetages (modèles) au niveau M1, pouvant signifier les types de liens listés dans ce besoin. Exemples: la dépendance «trace» (le rapport d'équivalence entre modèles); la dépendance «refine» (le rapport entre un modèle et ses vues différentes); la dépendance de réalisation entre la spécification et une implémentation (peut signifier une transformation).

Table I-10 : Évaluation de UML suivant les besoins pour M2

I.8 MOF - Évaluation suivant les besoins pour M3 (cf. Table I-1, page i-1)

BesoinM3		MOF
1	+-	MOF supporte différentes notions retenues par ce besoin mais non entièrement la notion de «*lien binaire*». En fait, MOF dans [107] définit un lien entre deux objets mais un lien n'est pas vu comme un objet (*nœud*); MOF dans ([106], [105]) définit un lien entre deux éléments mais un lien n'est pas vu comme un élément (*nœud*). Alors, MOF ne couvre pas toutes les structures possibles pour un lien.

2	+	MOF permet le passage entre niveaux types et instance.
3	-	MOF2.0 ne fait pas de distinction entre la relation de conformité d'un élément à un méta-élément et la relation de conformité d'un modèle à un métamodèle, les représentant par la même relation `instanceOf` (cf. la Figure 28, page 59, et la Figure 29, page 59).
4	+	
5	+	
6	-	Concernant la modularité, MOF soutient le mécanisme de paquetages avec la notion de paquetages (`Package`). Aucun moyen n'est disponible pour distinguer explicitement différentes notions identifiées pour ce besoin, par exemple: *métamodèle* d'un modèle; *modèle structurel* d'un type relationnel; *modèle de conditions*.
7	+-	MOF2.0 supporte différents types de liens identifiés pour ce besoin; exemples: liens de *contenant* entre un paquetage (modèle) et ses éléments; liens d'extension des paquetages comme `PackageImport` (avec les mots-clés `import`, `access`), et `PackageMerge` (avec le mot-clé `merge`). Cependant, il faudrait clarifier dans MOF2.0 la relation sémantique entre modèles et métamodèles. En outre, MOF ne permet pas de représenter différents types de liens entre modèles comme la jointure, l'intersection, la différence, l'inférence, la restriction.

Table I-11 : Évaluation de MOF suivant les besoins pour M3

I.9 XML-XML Schema

I.9.1 Évaluation suivant les besoins pour M2 (cf. Table I-2, page i-1)

BesoinM2		XML-XML Schema
1	+-	XML Schema supporte la notion de type et celle d'instance, mais ne considère pas la distinction de nature entre les notions «*type non relationnel*» et «*type relationnel*» ni celle entre «*occurrence*» et «*relation*».
2	+-	Mêmes raisons que dans le cas du BesoinM2 1 pour M2.
3	+	XML Schema permet ce besoin.
4	-	Le type d'un élément ou attribut est toujours indiqué par l'attribut `type`. Cependant, XML Schema n'explicite pas le rapport de conformité entre un élément (ou attribut) (déclaré dans un schéma XML Schema) et une instance de l'élément (ou attribut) en question (déclarée dans un document XML

		conforme au schéma en question). Par exemple, dans la Figure 52 (page 78), à la ligne 1, le rapport de typage entre l'élément nommé `Adresse` et `TypeAdresse`, indiqué par l'attribut `type`, n'est pas comme celui entre l'élément `Adresse` avec `xs:element`, ni comme celui entre `Adresse` avec ses instances (occurrences) déclarées aux lignes 7 et 13 dans la Figure 51 (page 77).
5	+	Les types peuvent être organisés en hiérarchie grâce aux mécanismes d'extension (`extension`) et de restriction (`restriction`) de types.
6	-	La multi-classification n'est pas autorisée dans XML Schema.
7	-	*Non applicable*. XML Schema ne considère aucun modèle de rôles.
8	-	
9	-	Ce besoin n'est pas pris en compte.
10	+-	XML Schema permet de spécifier le nombre d'occurrences d'un élément (ou attribut) dans un type complexe, s'appliquant à certains types de contraintes présentés dans ce BesoinM2 10 tels que «Contraintes de l'arité min/max sur un attribut pour un type» et «Contraintes de l'arité min/max sur la composition d'un type d'objets»
11	-	*Non applicable.*
12	-	
13	-	
14	-	
15	+-	Le seul moyen permettant de répondre à ce besoin dans XML Schema est l'utilisation de l'attribut `fixed` dans la déclaration d'éléments et attributs. Pourtant, comme l'attribut `fixed` est de type `string` (le type représentant l'ensemble de chaînes de caractères), l'utilisation de `fixed` dans la déclaration d'éléments est applicable plutôt aux éléments de types simples mais non aux éléments de types complexes. Donc, il n'est pas évident de modéliser les situations telles que celles du Problème 2 (page 37).
16	+-	XML Schema supporte la notion de modèles au moyen des notions de schémas et espaces de noms. Par contre, il ne considère pas les notions de métamodèles, modèles structurels, modèles de conditions.
17	+-	XML Schema dispose de l'extension (`include/import`) de schémas, non des différents types de liens entre modèles cités pour ce besoin. Exemples: la conformité sémantique entre modèles et métamodèles, l'intersection, la différence, l'inférence et la restriction entre modèles.

18	+-	XML Schema permet la redéfinition (`redefine`) de schémas mais ne tient pas compte de différents types de liens entre modèles cités pour ce besoin tels que l'équivalence et la transformation entre modèles.

Table I-12 : Évaluation de XML-XML Schema suivant les besoins pour M2

I.9.2 Évaluation du noyau de XML Schema selon les besoins pour M3 (cf. Table I-1, page i-1)

BesoinM3		Noyau de XML Schema
1	+-	Comme pour le BesoinM2 2 pour M2.
2	+	Comme pour le BesoinM2 3 pour M2.
3	-	Comme pour le BesoinM2 4 pour M2.
4	+	Comme pour le BesoinM2 5 pour M2.
5	-	Comme pour le BesoinM2 14 pour M2.
6	+-	Comme pour le BesoinM2 16 pour M2.
7	+-	Comme pour le BesoinM2 17 pour M2.

Table I-13 : Évaluation du noyau de XML Schema suivant les besoins pour M3

I.10 RDF-RDFS

I.10.1 Évaluation suivant les besoins pour M2 (cf. Table I-2, page i-1)

BesoinM2		RDF-RDFS
1	+-	RDF/RDFS ne supportent pas les propriétés qui sont les *types relationnels* autres que binaires entre classes, ni les *relations* (instances des propriétés) autres que binaires, ni les attributs pour les *types relationnels* et *relations*.
2	+-	Absence de propriétés qui sont les *méta-liens* binaires entre propriétés ou entre classes (*méta-nœuds*) avec propriétés. Absence de *liens* binaires (instances des propriétés) entre liens ou entre ressources (*nœuds*) avec liens.
3	+	RDF/RDFS permettent le passage entre niveaux types et instance.
4	-	RDF/RDFS représentent les liens de conformité entre types et instances par `rdf:type` mais ne prend pas en compte la notion de conformité entre modèles et métamodèles.
5	+	Support du sous-typage multiple entre classe et celui entre propriétés.
6	+	RDF/RDFS autorisent la multi-classification.
7	-	*Non applicable*. RDF/RDFS ne considèrent aucun modèle de rôles.
8	-	
9	-	RDF/RDFS ne considèrent pas ce besoin.
10	-	~~Contraintes structurelles: RDF/RDFS ne supportent aucun type de contraintes~~

11	-	retenu par les besoins de ce groupe, ne permettent donc pas de modéliser les situations telles que celle du Problème 1 (page 34).
12	-	
13	-	
14	-	
15	+-	Les cas de types relationnels binaires et sans attributs entre types et instances sont représentables dans RDF/RDFS mais leur interprétation sémantique n'est pas bien précisée. Voir l'exemple 39.
16	+-	Pour la notion de modèles, RDF/RDFS fournissent les notions de schémas (`rdf:RDF`) et espaces de noms mais ne considèrent pas les notions de métamodèles, modèles structurels, modèles de conditions.
17	+-	Concernant ce besoin, RDF/RDFS permettent de représenter les liens d'accès entre espaces de noms. Cependant ils ne considèrent pas les différents types de liens entre modèles tels que le respect sémantique entre modèles et métamodèles, l'intersection, la différence, l'inférence et la restriction.
18	+-	RDF/RDFS disposent des relations (`rdfs:isDefinedBy`, `rdfs:seeAlso`) pour relier une ressource à une autre (qui peut être un schéma) qui donnent des informations supplémentaires sur la première mais ne considèrent pas différents types de liens entre modèles mentionnés dans ce besoin, par exemple l'équivalence et la transformation entre modèles.

Table I-14 : Évaluation de RDF-RDFS suivant les besoins pour M2

RDF/RDFS - Types relationnels entre types et/ou instances (exemple 39)

La Figure I-1 montre que `PersonneEmployée-Teximus` est une sous-classe de `Personne`, reliée par `travaille-pour` à une compagnie (étant également une organisation) particulière, `Teximus`. La présence de `travaille-pour` ici (à la ligne 14 dans la Figure I-1) est considérée comme instance de la propriété `travaille-pour` déclarée à la ligne 6 dans la Figure I-1. Soit `Marie` une instance de `PersonneEmployée-Teximus`, est-ce que l'on peut déduire que `Marie` sera associée à `Teximus` par un lien `travaille-pour` ? Ceci n'est pas précisé dans RDF/RDFS. Et si la classe `Personne` est prédéfinie comme le domaine d'application de `travaille-pour`, alors, en principe, l'élément `PersonneEmployée-Teximus` (étant sous-classe de `Personne`) sera vu comme instance de `Personne`. Ceci implique une contradiction.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                xml:base="http://www.exemple.org/test/exRDFS">
```

```
1.   <rdfs:Class rdf:ID="Personne"/>
2.   <rdfs:Class rdf:ID="Organization"/>
3.   <rdfs:Class rdf:ID="Compagnie">
4.     <rdfs:subClassOf rdf:resource="#Organization"/>
5.   </rdfs:Class>
6.   <rdf:Property rdf:ID="travaille-pour">
7.     <rdfs:range rdf:resource="#Organization"/>
8.   </rdf:Property>
9.   <rdf:Description rdf:ID="Teximus">
10.    <rdf:type rdf:resource="#Compagnie"/>
11.  </rdf:Description>
12.  <rdfs:Class rdf:ID="PersonneEmployée-Teximus">
13.    <rdfs:subClassOf rdf:resource="#Personne"/>
14.    <travaille-pour rdf:resource="#Teximus"/>
15.  </rdfs:Class>
...
</rdf:RDF>
```

Figure I-1 : RDF/RDFS - Exemple de type relationnel entre types et/ou instances

I.10.2 Évaluation du noyau de RDF/RDFS selon les besoins pour M3 (cf. Table I-1, page i-1)

BesoinM3		Noyau de RDF/RDFS
1	+-	Comme pour le BesoinM2 2 pour M2.
2	+	Comme pour le BesoinM2 3 pour M2.
3	-	Comme pour le BesoinM2 4 pour M2.
4	+	Comme pour le BesoinM2 5 pour M2.
5	-	Comme pour le BesoinM2 14 pour M2.
6	+-	Comme pour le BesoinM2 16 pour M2.
7	+-	Comme pour le BesoinM2 17 pour M2.

Table I-15 : Évaluation du noyau de RDF/RDFS suivant les besoins pour M3

I.11 OWL

I.11.1 Évaluation suivant les besoins pour M2 (cf. Table I-2, page i-1)

BesoinM2		OWL
1	+-	Mêmes raisons que dans le cas de RDF/RDFS.
2	+-	
3	+	
4	-	
5	+	
6	+	
7	-	*Non applicable*. OWL ne prend pas en compte aucun modèle de rôles.

8	-	
9	-	OWL ne considère pas ce besoin.
10	-	Concernant les types de contraintes recensés par ces besoins, OWL ne tient compte que des contraintes de cardinalité retenues par les besoins BesoinM2 12 et BesoinM2 14. OWL ne permet donc pas modéliser les situations comme celle du Problème 1 (page 34).
11	-	
12	+	
13	-	
14	+	
15	+-	OWL règle uniquement les cas de types relationnels binaires et sans attributs entre types et instances. Ceci se fait grâce au mécanisme permettant de spécifier que toutes les instances d'un type ont une même valeur particulière pour une propriété (cf. l'exemple 40).
16	+-	Comme RDF/RDFS, OWL ne remplit que partiellement chacun de ces trois besoins. Concernant la représentation de modèles et de leurs liens, en plus des supports mis en place par RDF/RDFS (cf. la section 4.5.2, page 81), OWL dispose d'autres supports: (i) la notion d'ontologie; (ii) les propriétés portant sur la mise en correspondance des éléments (classes, propriétés, individus) pour faciliter la fusion des ontologies; et (iii) les propriétés portant sur la gestion des versions d'ontologies pour faciliter la maintenance des ontologies. Toutefois, comme RDF/RDFS, OWL ne dispose pas de notions de métamodèles, modèles structurels, modèles de conditions (BesoinM2 16); OWL ne dispose pas non plus de certains types relationnels entre modèles (schémas, ontologies, espaces de noms) qui sont importants pour l'aspect de modélisation et gestion de modèles, par exemple, le rapport de respect sémantique entre un modèle et son métamodèle, l'intersection, la différence, l'inférence, la restriction (BesoinM2 17), le rapport de transformation entre modèles et l'équivalence entre modèles (BesoinM2 18).
17	+-	
18	+-	

Table I-16 : Évaluation de OWL suivant les besoins pour M2

OWL - Types relationnels entre types et/ou instances (exemple 40)

La Figure I-2 illustre la définition que les personnes employées de `Teximus` (les instances du type `PersonneEmployée-Teximus`) sont les personnes (les instances de `Personne`) dont chacune travaille pour (`travaille-pour`) la compagnie (étant également une organisation) `Teximus`.

```
...
<owl:Class rdf:ID="Personne"/>
```

```
<owl:Class rdf:ID="Organization"/>
<owl:Class rdf:ID="Compagnie">
   <rdfs:subClassOf rdf:resource="#Organization"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="travaille-pour">
   <rdfs:domain rdf:resource="#Personne"/>
   <rdfs:range rdf:resource="#Organization"/>
</owl:ObjectProperty>

<Organization rdf:ID="Teximus"/>

<owl:Class rdf:ID="PersonneEmployée-Teximus">
   <owl:equivalentClass>
      <owl:Class>
         <rdfs:subClassOf rdf:resource="#Personne"/>
         <rdfs:subClassOf>
            <owl:Restriction>
               <owl:onProperty rdf:resource="#travaille-pour"/>
               <owl:hasValue rdf:resource="#Teximus"/>
            </owl:Restriction>
         </rdfs:subClassOf>
      </owl:Class>
   </owl:equivalentClass>
</owl:Class>
...
```

Figure I-2 : OWL – Exemple de type relationnel entre types et/ou instances

I.11.2 Évaluation du noyau de OWL en fonction des besoins pour M3 (cf. Table I-1, page i-1)

BesoinM3		Noyau de OWL
1	+-	Mêmes raisons que dans le cas du noyau de RDF/RDFS.
2	+	Comme pour le BesoinM2 3 pour M2.
3	-	Comme pour le BesoinM2 4 pour M2.
4	+	Comme pour le BesoinM2 5 pour M2.
5	+	Les contraintes de cardinalités sont représentées dans OWL.
6	+-	Comme pour le BesoinM2 16 pour M2.
7	+-	Comme pour le BesoinM2 17 pour M2.

Table I-17 : Évaluation du noyau de OWL suivant les besoins pour M3

Annexe II. Diagramme d'associations complet de notre méta-métamodèle

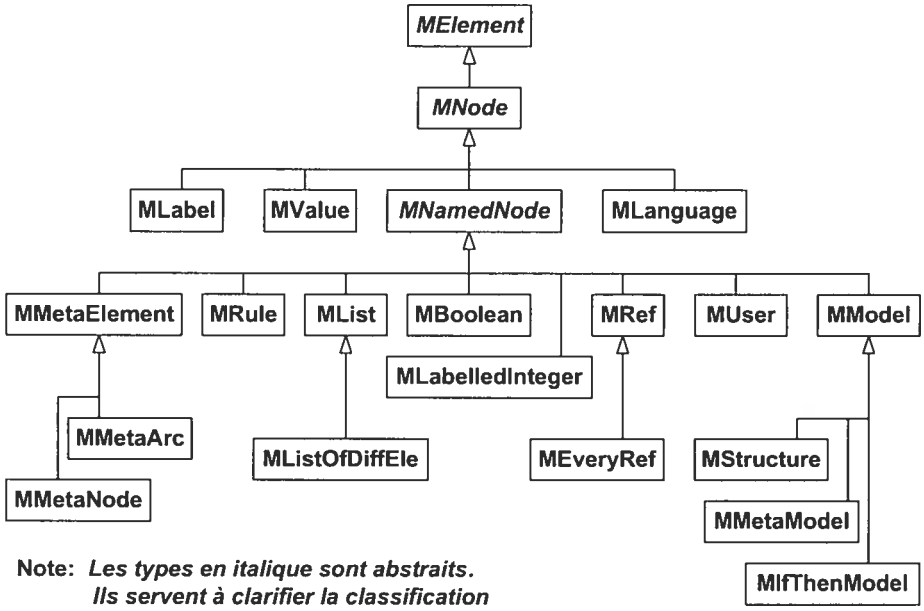


Figure II-1 : Méta-nœuds au niveau M3 : *MNode* et ses sous-types

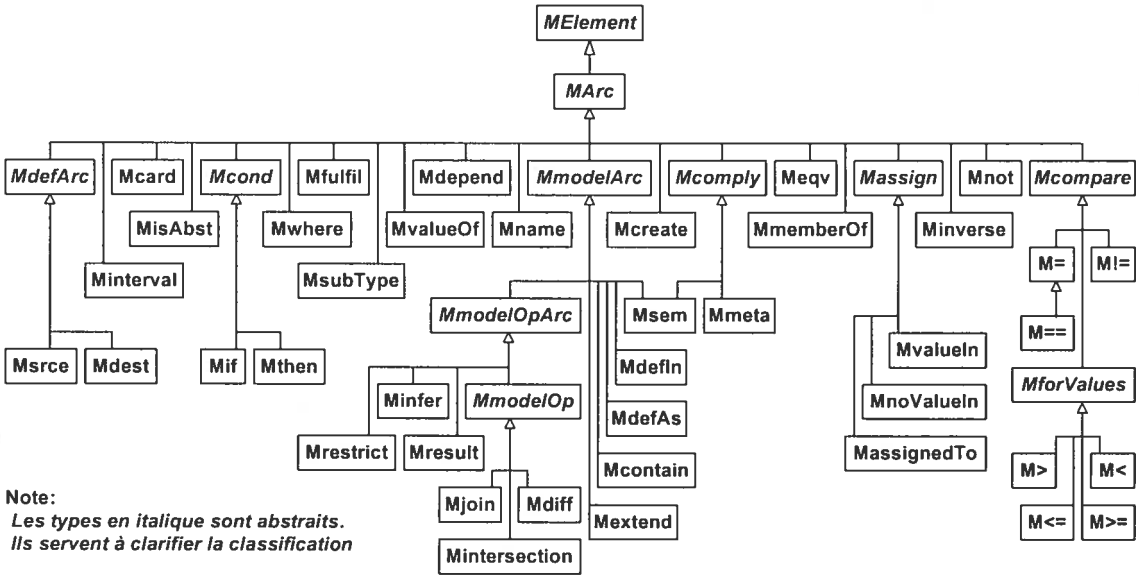


Figure II-2 : Méta-liens au niveau M3 : *MArc* et ses sous-types

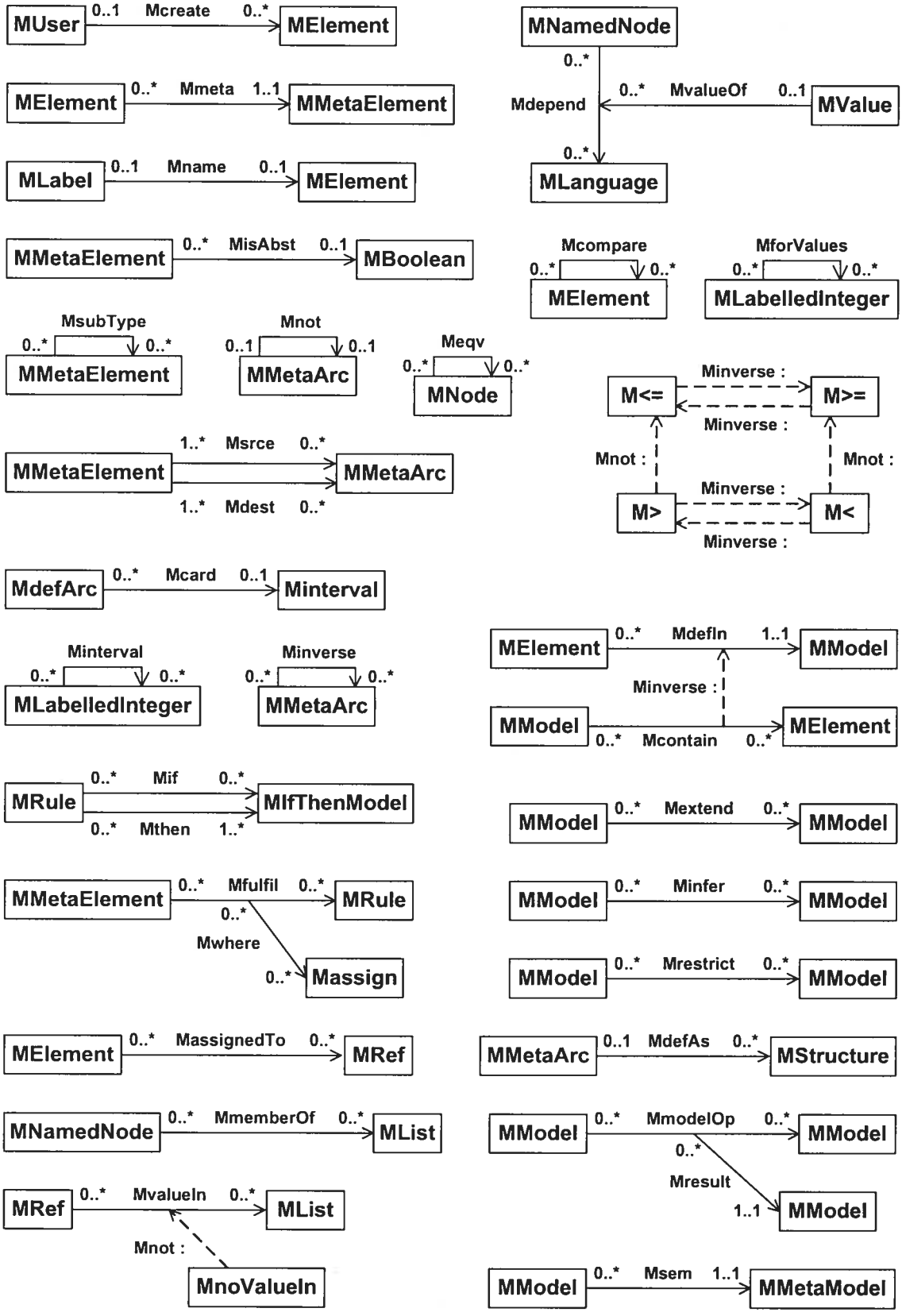


Figure II-3 : Diagrammes d'associations entre méta-éléments de M3

Annexe III. Éléments de notre métamodèle

Cette annexe présente la définition de chacun des méta-éléments définis dans notre métamodèle (M2) avec divers exemples de leurs applications.

III.1 Element

`Element` est conforme à `MMetaElement`. Il est un type abstrait, le super-type de tous les autres types de M2, et est subdivisé en deux: `Node`, `Arc`.

III.2 Méta-nœuds définis au M2

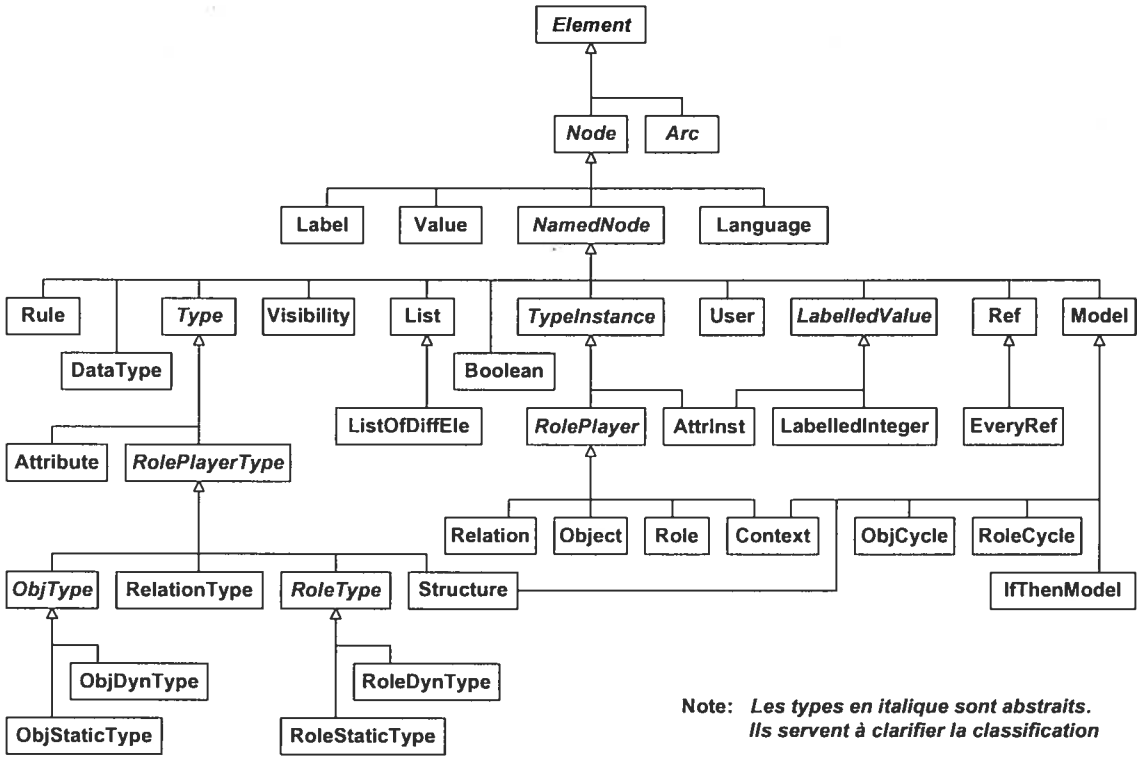


Figure III-1 : Hiérarchie des méta-nœuds au niveau M2 : *Node* et ses sous-types

III.2.1 Node

`Node` est un méta-nœud abstrait, le super-type de tous les autres méta-nœuds de M2. `Node` et ses sous-types se conforment à `MMetaNode`. `Node` est subdivisé en: `Label`, `Value`, `Language`, et `NamedNode`.

III.2.2 Label

`Label` représente l'ensemble des libellés à nommer les éléments de niveau M1.

III.2.3 Value, Language

`Language` et `Value` représentent vis-à-vis l'ensemble de langues et celui de valeurs

de l'univers du discours qui servent à représenter les interprétations des éléments de M1.

Nous traitons les langages comme des valeurs, c'est-à-dire, un `Language` et un `MLanguage` qui représentent un même langage peuvent être représentés par une même valeur, ce qui simplifiera l'interprétation des éléments. Le langage universel est représenté par «u», le français par «fr», l'anglais par «en», etc.

III.2.4 NamedNode

`NamedNode` est un type abstrait. Il représente l'ensemble des nœuds libellés. Les `NamedNode` sont traités comme objets. `NamedNode` est subdivisé en ces types-ci: `User`, `Visibility`, `Boolean`, `DataType`, `LabelledValue`, `Type`, `TypeInstance`, `Model`, `Rule`, `List`, et `Ref`.

III.2.5 User

Un `User` présente un utilisateur dans le système.

III.2.6 Visibility

Un `Visibility` représente une valeur de visibilité d'éléments: publique (+), protégé (#) et caché (-).

Définition 23 : Portée d'éléments

Soit `E` un élément défini dans un modèle `M`. La portée de `E` dépend de la valeur de la visibilité comme suit:

- `public` (publique): `E` est accessible de `M` et des modèles pouvant accéder à `M`.
- `protected` (protégée): `E` est accessible de `M` et de tous les modèles inclus dans `M` ainsi que tous les descendants de `M`.
- `private` (privée): `E` est accessible de `M` et des modèles inclus dans `M`.

Définition 24 : Portée d'attributs pour un type

Soit `T` un type défini dans un modèle `M`; et soit `A` un attribut pour un type `T`. La portée de `A` pour `T` dépend de la valeur de la visibilité comme suit :

- `private` (privée): `A` n'est accessible que depuis `T`.
- `protected` (protégée): `A` est accessible depuis `T` et depuis les types dérivés de `T`.
- `public` (publique): `A` est accessible de partout du moment que `T` est accessible.

Définition 25 : Type de relations héritable; Attribut héritable

Soit `R` un type de relations dont les éléments à unir sont (E_1, E_2, ..., E_n); soit `E` un élément héritier de E_1 par l'héritage et défini dans un modèle `M`. `R` devient héritable

pour E (c'est-à-dire une nouvelle structure de R avec les éléments (E, E_2, ..., E_n) à unir est déduite et accessible de M) si et seulement si R n'est pas caché et si tous les éléments E_2, ..., et E_n sont accessibles de M. Ce principe est tour à tour appliqué pour chaque élément héritier de E_1, E_2, ..., E_n pour déduire des structures possibles de R.

Un attribut d'un type est héritable si pour ce type cet attribut n'est pas caché (c'est-à-dire si l'attribut est publique ou protégé pour ce type).

III.2.7 Boolean

`Boolean` a seulement deux instances prédéfinies, `True` et `False`, représentant respectivement les valeurs logiques étiquetées : `True` pour le vrai, et `False` pour le faux. `True` est interprétée comme «1» en langage universel, «true» en anglais, et «vrai» en français; et `False` est interprétée comme «0» en langage universel, «false» en anglais, et «faux» en français.

III.2.8 LabelledValue

`LabelledValue` est abstrait et au sommet de la hiérarchie des types qui permettent de représenter des valeurs normalisées et étiquetées. Par exemple, la valeur «Papin» est le nom de `toto` et celui de `tutu` qui sont normalisés comme noms de personnes et étiquetés respectivement par `nom-de-toto` et `nom-de-tutu`.

`LabelledValue` est le super-type de `LabelledInteger` et `AttrInst`.

III.2.9 LabelledInteger

`LabelledInteger` représente l'ensemble des entiers étiquetés de M1. Un `LabelledInteger` est interprété comme un entier. Par exemple, l'élément `CardMin1` représente un entier étiqueté (conforme à `LabelledInteger`) qui vaut valeur 1.

III.2.10 DataType

`DataType` représente des types de données primitives comme ceux dans les langages de programmation. Par exemple, `String` pour des chaînes de caractères, `Integer` pour des entiers, `Float` pour des nombres réels, etc.

III.2.11 Type, TypeInstance

`Type` est abstrait et représente l'ensemble des types de M1. Il est subdivisé en: `Attribute` et `RolePlayerType`.

`TypeInstance` est abstrait et représente l'ensemble des éléments de M1 qui sont considérés comme les instances des types de M1. `TypeInstance` est subdivisé en: `AttrInst` et `RolePlayer`.

Une instance de type, c'est-à-dire une `TypeInstance`, n'est pas prise pour représenter un ensemble d'éléments. Les `TypeInstance` sont prises plutôt pour décrire et représenter des objets et choses de l'univers du discours ou dans le monde réel.

III.2.12 Attribute, AttrInst

`Attribute` représente l'ensemble des attributs pour caractériser des `Type` de M1.

`AttrInst` représente l'ensemble des instances des `Attribute`, c'est-à-dire les instances d'attributs, ou bien les valeurs d'attributs, pour caractériser des instances (`TypeInstance`) ou types d'instances (`Type`).

III.2.13 RolePlayer, RolePlayerType

`RolePlayer` représente l'ensemble des instances de types (`TypeInstance`) qui peuvent jouer des rôles et dits joueurs de rôles. `RolePlayer` est abstrait et subdivisé en: `Object`, `Role`, `Relation`, et `Context`.

`RolePlayerType` représente l'ensemble de types de joueurs de rôles. `RolePlayerType` est abstrait et subdivisé en: `ObjType`, `RoleType`, `RelationType`, et `Structure`.

III.2.14 Object, ObjType, ObjStaticType, ObjDynType

Un `Object` représente un objet de l'univers du discours ou dans le monde réel.

`ObjType` est un méta-nœud abstrait et représente l'ensemble des types d'objets. Sous-types de `ObjType`, `ObjStaticType` et `ObjDynType` représentent vis-à-vis l'ensemble des types statiques d'objets et l'ensemble des types dynamiques d'objets.

III.2.15 Role, RoleType, RoleStaticType, RoleDynType

Un `Role` représente un rôle ou bien une position qu'un `RolePlayer` peut jouer. Un `RoleType` représente un type de rôles ou bien de positions.

`RoleType` est un type abstrait et représentent l'ensemble des types de rôles. `RoleType` est subdivisé en deux : `RoleStaticType` pour les types statiques de rôles; et `RoleDynType` pour les types dynamiques de rôles.

III.2.16 Relation, RelationType

Un `RelationType` représente un type de relations (*type relationnel*) entre éléments de M1. Une `Relation` représente une relation instanciée d'un type de relations de M1.

III.2.17 Model

`Model` représente l'ensemble des modèles au niveau M1. `Model` est le super-type des types-ci : `Structure`, `Context`, `ObjCycle`, `RoleCycle`, et `IfThenModel`.

III.2.18 Structure, Context

`Structure` représente l'ensemble de structures des types de relations au niveau M1.

`Context` représente l'ensemble des modèles des relations au niveau M1. Ces modèles sont vus comme instances des structures des types de relations au M1 mais ne sont pas des structures.

`Structure` est un sous-type de `RolePlayerType`, et `Context` est celui de `RolePlayer`: un `Structure` (`Context`) est compté pour un `RolePlayerType` (`RolePlayer`), et peut s'impliquer dans des `Relation`/ `RelationType`.

III.2.19 ObjCycle, RoleCycle

`ObjCycle` (`RoleCycle`) représente l'ensemble de cycles d'états pour les types d'objets (pour les types de rôles).

III.2.20 IfThenModel

`IfThenModel` représente l'ensemble des modèles de conditions dans la représentation de règles définies au niveau M1.

III.2.21 Rule

`Rule` représente l'ensemble des règles nécessaires définies au M1 qu'un élément doit observer.

III.2.22 List, ListOfDiffEle

Un `List` représente au M1 une liste d'éléments. Plus spécifique, une `ListOfDiffEle` représente au M1 une liste d'éléments distincts deux à deux.

III.2.23 Ref, EveryRef

`Ref` et `EveryRef` de M2 visent à représenter les variables dans les règles définies au niveau M1. `Ref` correspond à l'interprétation du quantificateur existentiel (∃). `EveryRef` correspond à celle du quantificateur universel (∀). La Règle 19 (Annexe IV) sur l'attribution d'éléments à une variable est à noter.

III.3 Méta-liens définis au M2

La Figure III-2 illustre la hiérarchie des méta-arcs dans notre M2.

III.3.1 Arc

`Arc` est un type abstrait et est le super-type de tous les autres méta-arcs de M2. Un `Arc` représente au M1 un arc en général entre deux éléments et est le représentant de tous les autres arcs. `Arc` et ses sous-types se conforment à `MMetaArc`.

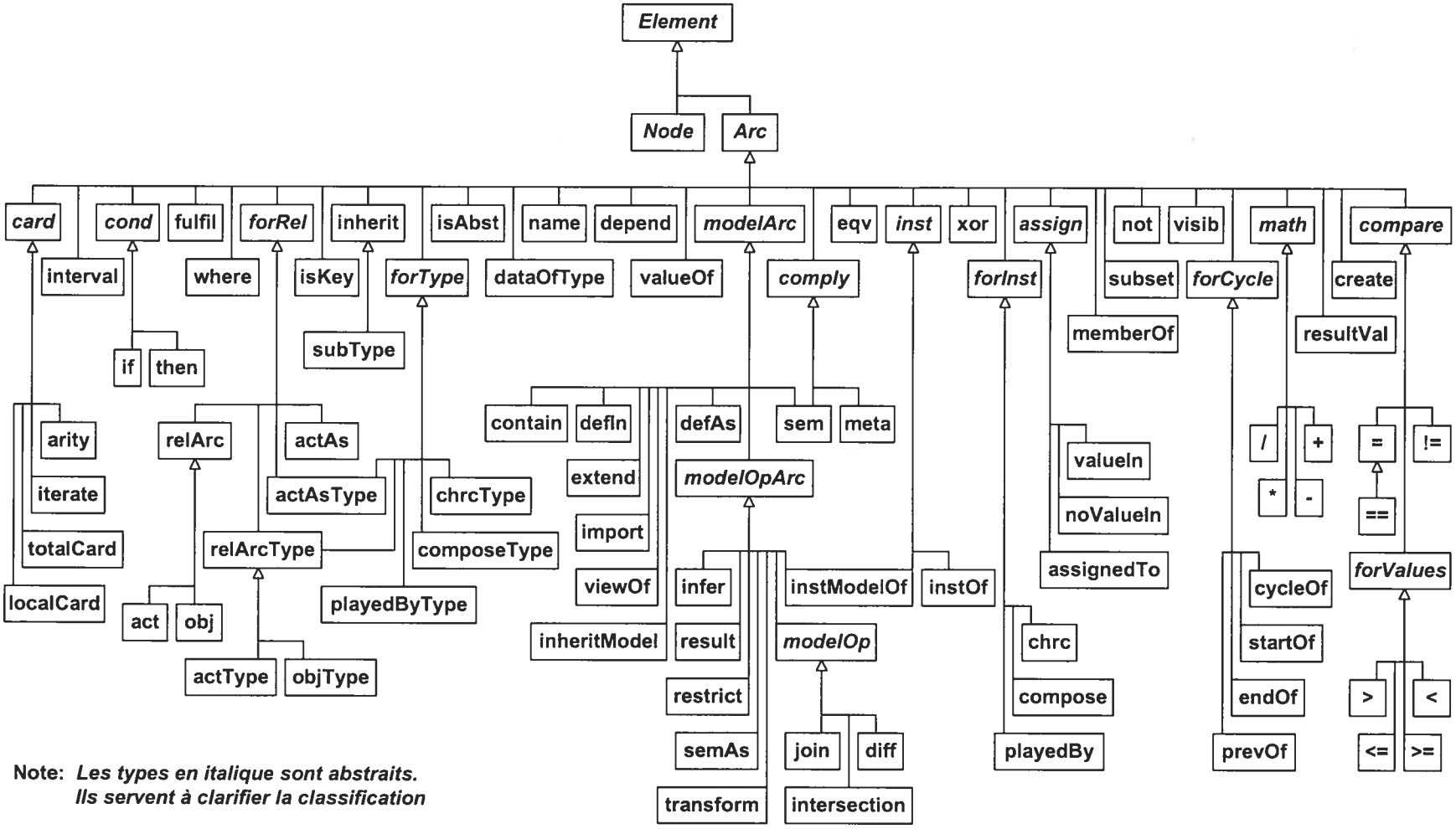


Figure III-2 : Hiérarchie des méta-arcs au niveau M2 : *Arc* et ses sous-types

1. Liens de conformités du niveau M1 au niveau M2

III.3.2 comply

`comply` est abstrait et regroupe les liens de conformité entre les éléments de niveau M2 et leurs instances de niveau M1. `comply` est représenté par `meta` et `sem`.

III.3.3 meta

`meta` représente l'ensemble des liens de conformité entre les éléments de M1 et leurs méta-éléments de M2. Chaque élément (nœud, arc) de M1 est conforme à un et un seul méta-élément (respectivement méta-nœud, méta-arc) de M2, et est attaché à ce dernier par un lien de conformité de type `meta`. La Règle 3 (Annexe IV) est à respecter.

III.3.4 sem

`sem` représente l'ensemble des liens de conformité entre les modèles au M1 et leurs métamodèles au M2. Un modèle (`Model`) se conforme (`sem`) à un et un seul métamodèle (`MMetamodel`). La Règle 18 (Annexe IV) est à noter.

2. Nommage d'éléments

III.3.5 name

`name` représente l'ensemble des liens à spécifier les noms des éléments de M1. Un libellé (`Label`) peut nommer (`name`) au maximum un élément (`Element`), et un élément peut être nommé par au maximum un libellé. Plus restrictivement, chaque nœud nommé (`NamedNode`) doit s'attacher à son libellé.

3. Interprétation d'éléments

III.3.6 depend, valueOf

`depend` et `valueOf` permettent de spécifier les interprétations des éléments de M1 selon différentes langues telles que le français, l'anglais, etc.

Le fait qu'une valeur est attachée par un arc de type `valueOf` à un arc de type `depend` liant un élément à une langue (cf. la Figure 103, page 117), montre que l'élément en question est attaché à des valeurs interprétées conformément à la langue.

4. Représentation de contraintes structurelles

III.3.7 interval

`interval` (cf. la Figure 136, page 135) permet de former au M1 les intervalles fermés dont les bornes inférieures et celles supérieures sont des entiers.

III.3.8 card

`card` regroupe les méta-arcs permettant de représenter au niveau M1 divers types de

contraintes structurelles listés dans les besoins de BesoinM2 10 (page 36) à BesoinM2 13 (page 37). `card` est abstrait et subdivisé en: `arity`, `totalCard`, `localCard`, `iterate`. Chacun de ces méta-arcs est restreint par les Règle 4 et Règle 8 (Annexe IV).

III.3.9 arity

`arity` représente l'ensemble des liens spécifiant les contraintes de l'arité (cf. le BesoinM2 10 - «Contraintes de l'arité min/max», page 36). `arity` est défini pour relier `forType` à `interval` avec les cardinalités source 0..* et les cardinalités destination 0..1 (cf. la Figure 138, page 136).

Spécification de contraintes de l'arité (exemple 41)

La Figure III-3 illustre comment sont définies les contraintes de l'arité sur l'arc de type `actType` de `Personne` (représentant le type de personnes) à `travailler` (représentant le type de relations `travailler`). L'arité maximale sur cet arc est de valeur 7 tandis que celle minimale prend la valeur 1. Ceci spécifie qu'il peut avoir d'un à sept personnes au maximum qui peuvent travailler ensemble (comme dans un groupe). La Figure III-3 peut être représentée plus simplement comme dans la Figure III-4. Voir la Notation 5 (page 137) pour notre notation liée aux contraintes de l'arité.

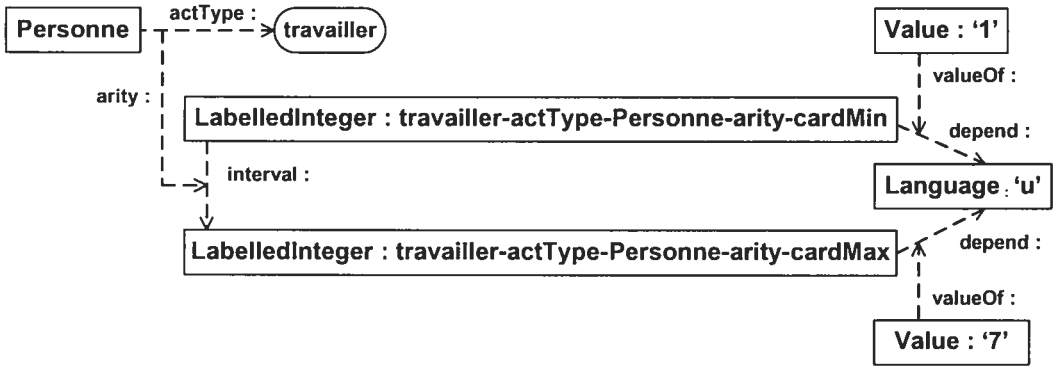


Figure III-3 : Exemple de définir les contraintes de l'arité

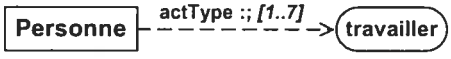


Figure III-4 : Version simplifiée de la Figure III-3

III.3.10 totalCard

`totalCard` représente l'ensemble des liens spécifiant les contraintes de cardinalités totales (cf. le BesoinM2 11 - «Contraintes de cardinalité totale min/max pour un type relationnel», page 36). `totalCard` est défini pour relier chacun des types `relArcType` et `actAsType` à `interval` avec les cardinalités source 0..* et les cardinalités destination 0..1 (cf. la Figure 141, page 138).

Spécification de contraintes de cardinalité totale (exemple 42)

La Figure III-5 illustre la définition des contraintes de la cardinalité totale sur l'arc de type `actType` *de* `Personne` *à* `travailler`*. La cardinalité totale minimale sur cet arc est de valeur 0 et celle maximale prend la valeur infinie. Ceci signifie qu'une personne peut ne participer à aucune, ou participer à une ou à plusieurs relations de type* `travailler`*. La Figure III-6 présente la version plus simple de la Figure III-5. Voir la Notation 6 (page 138) sur cardinalités totales minimale et maximale.*

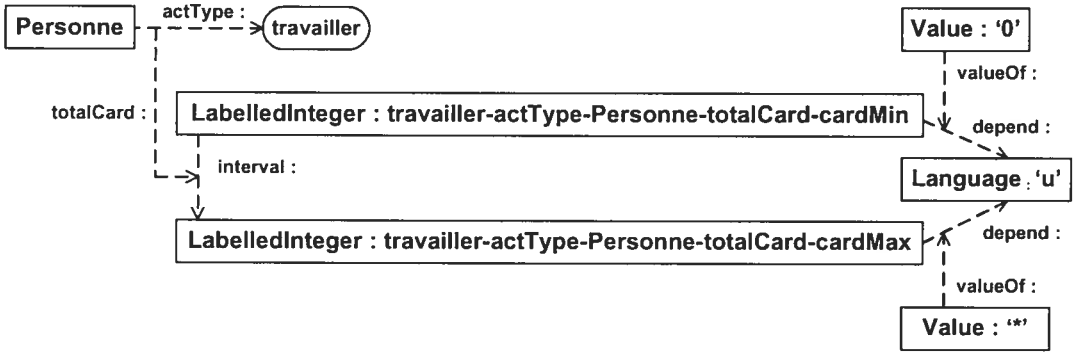


Figure III-5 : Exemple de définir les contraintes de cardinalité totale

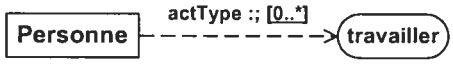


Figure III-6 : Version simplifiée de la Figure III-5

III.3.11 localCard

`localCard` représente l'ensemble des liens spécifiant les contraintes de cardinalités locales (cf. le BesoinM2 12 - «Contraintes de cardinalité locale min/max pour un type relationnel», page 37). `localCard` est défini pour relier chacun des types `relArcType` et `actAsType` à `interval` avec les cardinalités source 0..* et les cardinalités destination 0..1 (cf. la Figure 142, page 138).

Spécification de contraintes de cardinalité locale (exemple 43)

La Figure III-7 démontre comment sont définies les contraintes de la cardinalité locale sur l'arc de type `actType` *de* `Personne` *à* `travailler`*. La cardinalité locale minimale sur cet arc prend la valeur 0 alors que celle maximale prend la valeur 20. Ceci spécifie qu'il peut avoir au maximum vingt personnes dont chacune s'associe ensemble à une relation de type* `travailler` *avec un même groupe de personnes. Autrement dit, il est possible d'avoir au maximum vingt personnes dont chacune peut travailler ensemble avec un même groupe de personnes. La Figure III-8 illustre la version simplifiée de la Figure III-7. Voir la Notation 7 (page 138) sur cardinalités locales minimale et maximale.*

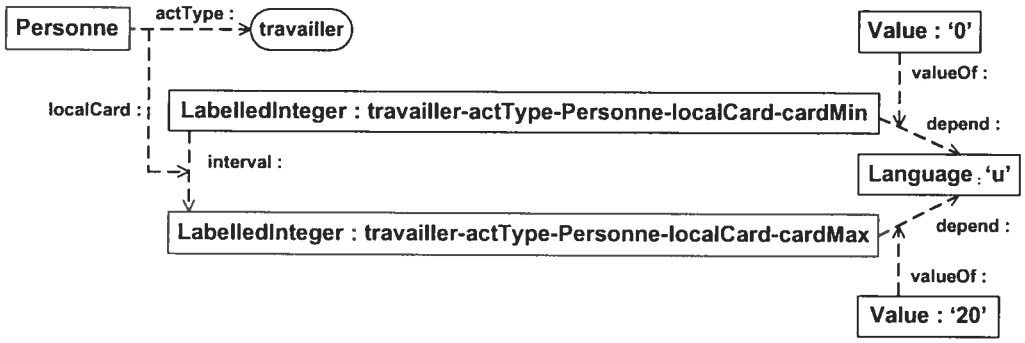


Figure III-7 : Exemple de définir les contraintes de cardinalité locale

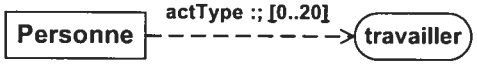


Figure III-8: Version simplifiée de la Figure III-7

III.3.12 iterate

`iterate` représente l'ensemble des liens spécifiant les contraintes de l'itération pour un type de relations (cf. le BesoinM2 13 - «Contraintes de l'itération min/max pour un type relationnel», page 37). `iterate` est défini pour relier `RelationType` à `interval` avec les cardinalités source 0..* et les cardinalités destination 0..1 (cf. la Figure 143, page 139).

Spécification de contraintes de l'itération pour un type de relations (exemple 44)

La Figure III-9 montre un exemple de comment spécifier des contraintes de l'itération pour un type de relations. Pour le type de relations `travailler`*, les mêmes personnes peuvent participer ensemble à une relation de type* `travailler` *plusieurs fois. Alors le nombre de l'itération minimale prend la valeur 1 et celui maximal prend la valeur infinie. Ceci est représenté plus simplement comme dans la Figure III-10. Voir la Notation 8 (page 139) sur itérations minimale et maximale pour un type de relations.*

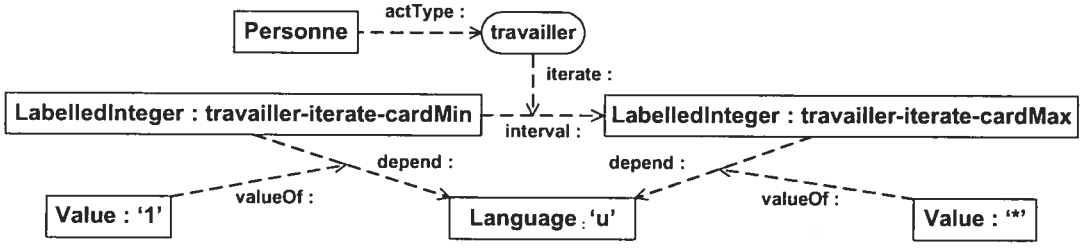


Figure III-9 : Exemple de définir les contraintes de l'itération

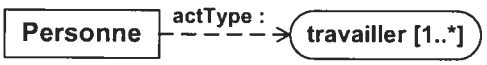


Figure III-10 : Version simplifiée de la Figure III-9

5. Spécification de relations et de types de relations

III.3.13 forRel

`forRel` est abstrait et est le super-type des méta-arcs qui permettent d'indiquer les participants d'une relation (`Relation`) ou d'un type de relations (`RelationType`) de niveau M1. `forRel` est subdivisé en: `relArcType`, `actAsType`, `actAs`, et `relArc`.

III.3.14 relArcType, actType, objType

Des `Type` qui peuvent s'engager directement à un type de relations (`RelationType`) sont donc reliés à ce dernier par des `relArcType` (cf. la Figure 112, page 122).

`actType` et `objType` sont les sous-types de `relArcType`. Dans un type de relations *TR*, un participant direct spécifié comme un acteur de l'action (resp. comme un objet de l'action) est associé à *TR* par un arc de type `actType` (resp. de type `objType`).

Spécification du type de relations travailler (exemple 45)

La Figure III-11 (ou bien Figure III-12) illustre une structure du type de relations `travailler` avec les contraintes relatives à ce type déjà décrites dans les exemple 41 (page iii-8), exemple 42 (page iii-9), exemple 43 (page iii-9), et exemple 44 (page iii-10). Voir aussi les Notation 5 (page 137), Notation 6 (page 138), Notation 7 (page 138), et Notation 8 (page 139).

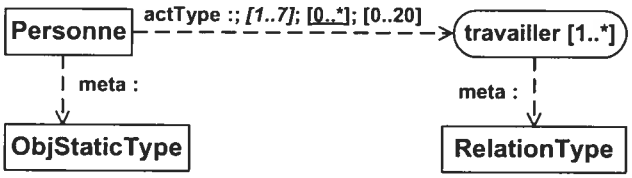


Figure III-11 : Une structure du type de relations *travailler*

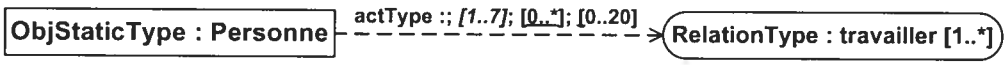


Figure III-12 : Version simplifiée de la Figure III-11

La Figure III-13 montre une autre structure du type de relations `travailler` avec des contraintes relatives à ce type. Dans cette structure, le type de rôles employé (`Employé`) est engagé comme l'acteur de l'action `travailler`.

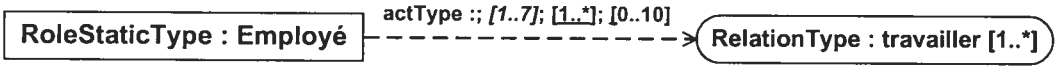


Figure III-13 : Une autre structure du type de relations *travailler*

Spécification du type de relations travailler-pour (exemple 46)

`travailler-pour` est un type de relations de `Personne` à `Organisation` et/ou à

Personne tout comme du type de rôles Employé au type de rôles Employeur. Nous pouvons représenter des énoncés tels que «la personne toto (comme l'employé) travaille pour la personne Fortin (comme l'employeur)», «les personnes toto, tata, tutu (comme les employés) travaillent ensemble pour la compagnie Teximus (comme l'employeur)».

La Figure III-14 présente une structure du type de relations travailler-pour dans lequel deux types statiques de rôles, Employé et Employeur, s'impliquent respectivement comme l'acteur et l'objet de l'action. Voir aussi les Notation 5 (page 137), Notation 6 (page 138), Notation 7 (page 138), et Notation 8 (page 139).

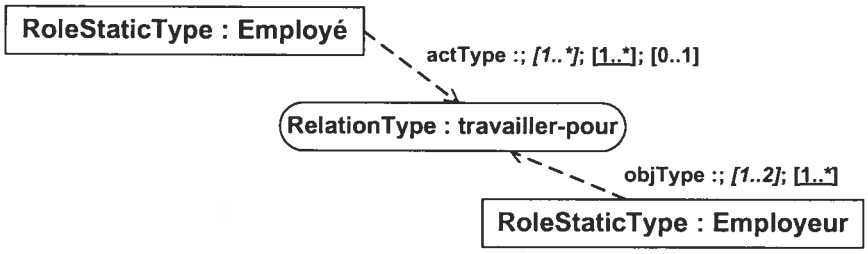


Figure III-14 : Une structure du type de relations *travailler-pour*

Les contraintes (implicites et explicites) de l'arité et de cardinalités (totales et locales) sur l'arc de type actType et celles sur l'arc de type objType de travailler-pour sont interprétées comme suit:

(1) Il y a un ou plusieurs employés qui travaillent en même temps pour de 1 à 2 employeurs. Autrement dit, les participants ensemble à une relation de type travailler-pour peuvent être : (1a) au moins un employé (ce qui est indiqué par les contraintes de l'arité notées comme [1..] sur l'arc de type actType de Employé à travailler-pour); et (1b) de 1 à 2 employeurs (ce qui est indiqué par les contraintes de l'arité notées comme [1..2] sur l'arc de type objType de Employeur à travailler-pour).*

(2) Chaque employé doit participer à au moins une relation de type travailler-pour, ce qui est indiqué par les contraintes de cardinalité totale notées comme [1..*] *sur l'arc de type actType de Employé à travailler-pour.*

(3) Un employé ne peut pas participer à nouveau à une relation de type travailler-pour avec les mêmes participants avec qui il s'est associé à une relation de type travailler-pour, ce qui est indiqué par les contraintes de cardinalité locale notées comme [0..1] *sur l'arc de type actType de Employé à travailler-pour*

(4) Chaque employeur doit participer à au moins une relation de type travailler-pour, ce qui est indiqué par les contraintes de cardinalité totale notées comme [1..*]

sur l'arc de type `objType` *de* `Employeur` *à* `travailler-pour`.

Le nombre de l'itération minimal (respectivement maximal) pour `travailler-pour` *n'est pas affiché, et prend donc la valeur 1 (respectivement *). Ceci spécifie cette contrainte:*

(5) Une relation de type `travailler-pour` *peut se répéter les mêmes participants.*

Pour ce type de relations `travailler-pour`*, il peut exister d'autres contraintes à représenter concernant divers éléments tels que des types de joueurs de rôles qui puissent s'occuper des rôles employé, employeurs, etc. Pour voir comment les contraintes de ce genre peuvent s'exprimer dans notre formalisme, voir les spécifications de* `actAsType` *(section III.3.15) et* `playedByType` *(section III.3.21).*

III.3.15 actAsType

Structure 1 (cf. la Figure 115, page 123): `actAsType` peut unir `RolePlayerType` à `relArcType`. Ceci permet de spécifier des types de joueurs de rôles (`RolePlayerType`) pour un type de rôles (`RoleType`) étant un participant direct d'un type de relations. La Règle 24 (Annexe IV) est à noter.

La sémantique de `actAsType` dans cette structure:

Soient *`RoleTypeA`* un type de rôles, *`RolePlayerTypeB`* un type de joueurs de rôles, et *`RelationTypeX`* un type de relations. Dans une structure de *`RelationTypeX`*, soient arc_1 un `relArcType` (c'est-à-dire un arc de type `relArcType` (ou `actType` ou `objType`)) associant *`RoleTypeA`* à *`RelationTypeX`*; et arc_2 un `actAsType` associant *`RolePlayerTypeB`* à arc_1. Donc, dans le type de relations *`RelationTypeX`*, *`RolePlayerTypeB`* s'implique comme le type de joueurs de rôles pour le type de rôles *`RoleTypeA`*. Autrement dit, un ou des joueurs de type *`RolePlayerTypeB`* peuvent participer à une relation de type *`RelationTypeX`* sous rôles de type *`RoleTypeA`*. Les contraintes qui concernent le type de relations *`RelationTypeX`* et qui sont attachées aux arcs arc_1, arc_2 ainsi que les contraintes qui concernent *`RoleTypeA`* et/ou *`RolePlayerTypeB`* doivent être respectées. Les contraintes de l'arité et de cardinalité totale/locale sur un arc de type `actAsType`, c'est-à-dire sur arc_2 dans ce cas, sont interprétées comme ci-après:

- L'arité minimale (maximale) indique le nombre minimal (maximal) des joueurs de type *`RolePlayerTypeB`* qui peuvent, sous rôles de type *`RoleTypeA`*, s'impliquer ensemble dans une relation de type *`RelationTypeX`*.
- La cardinalité totale minimale (maximale) indique le nombre minimal (maximal)

de relations de type *RelationTypeX* dans lesquelles un joueur de type *RolePlayerTypeB* peut impliquer sous un ou des rôles de type *RoleTypeA*.

- La cardinalité locale minimale (maximale) indique le nombre minimal (maximal) de joueurs de type *RolePlayerTypeB* dont chacun, sous un rôle de type *RoleTypeA*, peut impliquer dans une relation de type *RelationTypeX* avec un même groupe de participants restants (y compris des rôles et des joueurs de rôles).

Structure 2 (cf. la Figure 116, page 124): actAsType peut unir RolePlayerType à actAsType. Ceci permet de spécifier des types de joueurs de rôles (RolePlayerType) pour un type de rôles (RoleType) impliqué dans le cadre d'une structure d'un type de relations. La Règle 25 (Annexe IV) est à noter.

La sémantique de actAsType dans cette structure:

Soient *RoleTypeI* un type de rôles, *RolePlayerTypeJ* un type de joueurs de rôles, et *RelationTypeX* un type de relations. Supposons que dans une structure de *RelationTypeX*, arc_i représente un actAsType qui associe un actAsType (ou un relArcType) à *RoleTypeI*, et arc_j représente un actAsType qui associe *RolePlayerTypeJ* à arc_i. Alors, dans le type de relations *RelationTypeX*, *RolePlayerTypeJ* s'implique comme le type de joueurs de rôles pour le type de rôles *RoleTypeI*. Autrement dit, un ou des joueurs de type *RolePlayerTypeJ* peuvent s'impliquer dans une relation de type *RelationTypeX* sous rôles de type *RoleTypeI*. Il faut respecter les contraintes qui concernent le type de relations *RelationTypeX* et qui sont attachées aux arcs arc_i, arc_j ainsi que les contraintes qui concernent *RoleTypeI* et/ou *RolePlayerTypeJ*. Les contraintes de l'arité et de cardinalité totale/locale sur un arc de type actAsType, c'est-à-dire sur arc_j dans ce cas, sont interprétées d'une façon similaire à celles sur arc_2 que nous avons spécifiées dans la présentation sur la ***Structure 1*** de actAsType (cf. la Figure 115, page 123).

Un exemple d'utilisation de actAsType se trouve dans l'exemple 47.

Types de joueurs de rôles pour types de rôles dans un type de relations (exemple 47)

Reprenons l'exemple sur le type de relations travailler décrit dans l'exemple 45 (page iii-11). La Figure III-13 (page iii-11) a illustré une structure de travailler où le type de rôles employé (Employé) est engagé pour l'acteur de l'action travailler. La Figure III-15 montre une image plus complète de cette structure avec quelques contraintes ajoutées sur le type de relations travailler.

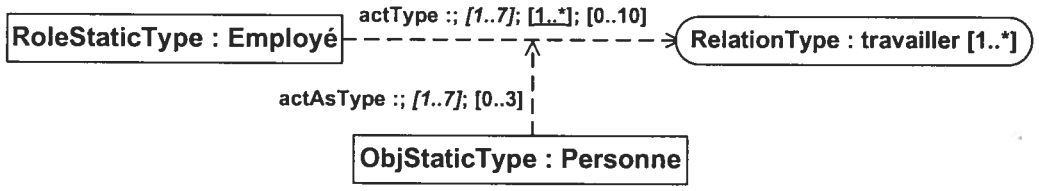


Figure III-15 : Types de rôles et leurs types de joueurs dans un type de relations

Les contraintes sur l'arc de type `actType` *de* `Employé` *à* `travailler` *sont interprétées comme suit:*

(1) Il y a d'un à sept rôles employés (`Employé`*) pouvant travailler ensemble en même temps, c'est-à-dire participer ensemble à une relation de type* `travailler`*. Ceci est indiqué par les contraintes de l'arité notées comme [1..7] sur l'arc de type* `actType` *de* `Employé` *à* `travailler`*.*

(2) Chaque employé doit participer à au moins une relation de type `travailler`*, ce qui est indiqué par les contraintes de cardinalités totales notées comme* [1..*] *sur l'arc de type* `actType` *de* `Employé` *à* `travailler`*.*

(3) Il y a au maximum dix rôles employé dont chacun peut s'associer à une relation de type `travailler` *avec les mêmes participants restants (c'est-à-dire les mêmes rôles employé ainsi que leurs joueurs restants qui sont impliqués ensemble dans une relation de type* `travailler`*). Ceci est indiqué par les contraintes de cardinalités locales notées comme* [0..10] *sur l'arc de type* `actType` *de* `Employé` *à* `travailler`*.*

Selon cette structure de `travailler`*, une personne (*`Personne`*) s'implique dans une relation de type* `travailler` *sous le rôle employé. Les contraintes sur l'arc de type* `actAsType` *de* `Personne` *à l'arc de type* `actType` *(de* `Employé` *à* `travailler`*) (cf. Figure III-15) sont interprétées comme suit:*

(4) Les arités minimale et maximale sur cet arc de type `actAsType` *sont respectivement de valeurs 1 et 7, et elles sont notées comme [1..7]. Ceci spécifie que dans chaque relation de type* `travailler`*, il y a d'une à sept personnes s'impliquant ensemble en tant qu'employés.*

(5) Les cardinalités totales minimale et maximale sur cet arc de type `actAsType` *ne sont pas affichées et sont donc de valeur* [0..*]*. Ceci spécifie qu'une personne peut ne participer à aucune, ou participer à une ou plusieurs relations de type* `travailler` *en tant qu'employés.*

(6) Les cardinalités locales minimale et maximale sur cet arc de type `actAsType` *sont notées comme* [0..3]*. Ceci représente qu'il existe au maximum trois personnes dont chacune, en tant qu'employé, peut participer à une relation de type* `travailler` *avec*

un même groupe de participants restants.

Les contraintes de l'itération pour `travailler` *spécifient:*

(7) Une relation de type `travailler` *peut se répéter plusieurs fois. C'est-à-dire, les mêmes employés peuvent participer ensemble plusieurs fois à une relation de type* `travailler`*. Ceci est indiqué par les contraintes notées comme* [1..*] *suivant le nom de ce type* `travailler`*.*

III.3.16 actAs

Structure 1 (cf. la Figure 118-(1), page 125): `actAs` peut unir `RolePlayer` à `relArcType`. Ceci permet d'indiquer des joueurs de rôles fixes pour un type de rôles qui est un participant direct d'un type de relations.

Structure 2 (cf. la Figure 118-(2), page 125): `actAs` peut unir `RolePlayerType` à `relArcType`. Ceci permet d'indiquer des types de joueurs de rôles fixes pour un type de rôles qui est un participant direct d'un type de relations.

La Règle 26 (Annexe IV) est à noter.

La sémantique de ces structures 1 et 2 de `actAs`:

Soient *`RoleTypeA`* un type de rôles, et *`RelationTypeX`* un type de relations. Dans une structure de *`RelationTypeX`*, soit arc_1 un `relArcType` (c'est-à-dire un arc de type `relArcType` (ou `actType` ou `objType`)) associant *`RoleTypeA`* à *`RelationTypeX`*. Un arc de type `actAs` d'un joueur de rôles à arc_1 (cf. Figure 118-(1)) représente que ce joueur de rôles participe à toute relation de type *`RelationTypeX`* sous rôles de type *`RoleTypeA`*. Un arc de type `actAs` d'un type de joueurs de rôles à arc_1 (cf. Figure 118-(2)) représente que toutes les instances de ce type de joueurs de rôles participent à chaque relation de type *`RelationTypeX`* sous rôles de type *`RoleTypeA`*. Clairement, les contraintes relatives au type de relations *`RelationTypeX`* doivent être satisfaites.

Structure 3 (cf. la Figure 120-(3), page 127): `actAs` peut unir `RolePlayer` à `actAsType`. Ceci permet d'indiquer des joueurs de rôles fixés pour un type de rôles impliqué dans le cadre d'une structure d'un type de relations.

Structure 4 (cf. la Figure 120-(4), page 127): `actAs` peut unir `RolePlayerType` à `actAsType`. Ceci permet d'indiquer des joueurs de rôles fixés pour un type de rôles impliqué dans le cadre d'une structure d'un type de relations.

La Règle 27 (Annexe IV) est à noter.

La sémantique de ces structures 3 et 4 de actAs:

Soit *RelationTypeX* un type de relations. Soit *RoleTypeJ* un type de rôles impliqué dans *RelationTypeX* selon une structure de *RelationTypeX*, ce qui est indiqué par arc_j, un arc de type actAsType. Un arc de type actAs d'un joueur de rôles (*RolePlayerE*) à l'arc arc_j représente que ce joueur de rôles *RolePlayerE* s'implique dans chaque relation de type *RelationTypeX* sous rôles de type *RoleTypeJ* (ce qui correspond à la sémantique de actAs dans la structure illustrée dans la Figure 120-(3) ou bien Figure 120-(3a),(3b)). Un arc de type actAs d'un type de joueurs de rôles (*RoleTypeE*) à l'arc arc_j représente que toutes les instances de type *RoleTypeE* s'impliquent dans chaque relation de type *RelationTypeX* sous rôles de type *RoleTypeJ* (ce qui correspond à la sémantique de actAs dans la structure illustrée dans la Figure 120-(4) ou bien Figure 120-(4a),(4b)). Évidemment, d'autres contraintes relatives au type de relations *RelationTypeX* doivent être respectées.

III.3.17 relArc, act, obj

relArc représente l'ensemble des arcs pour indiquer des participants fixes à une relation (Relation) ou à un type de relations (RelationType) de M1. Sous-types de relArc, les méta-arcs act et obj visent respectivement à expliciter des participants pris pour acteurs de l'action et des participants pris pour objets de l'action.

relArc/act/obj pour la représentation de relations

Structure 1 (cf. la Figure 122-(1), page 128): relArc peut unir TypeInstance à Relation pour indiquer des instances (TypeInstance) impliquées dans une relation.

Structure 2 (cf. la Figure 122-(2), page 128): relArc peut unir Type à Relation pour indiquer des types impliqués dans une relation. Le fait qu'un type est rattaché par un arc de type relArc (respectivement act ou obj) à une relation représente que toutes les instances de ce type s'impliquent directement dans cette relation.

Type d'objets participant à une relation (exemple 48)

Retournons au type de relations travailler étudié dans l'exemple 45 (page iii-11). Le fait que les personnes travaillent ensemble est interprété que celles-ci participent ensemble à une relation de type travailler. Soient toto, tata, tutu, Jean, et Marie les éléments représentant les personnes (Figure III-16). Soit que toto, tata, tutu, et Jean sont toutes les instances de PersonneGroupe1, celui-ci est un type dynamique d'objets représentant un groupe de personnes travaillant ensemble.

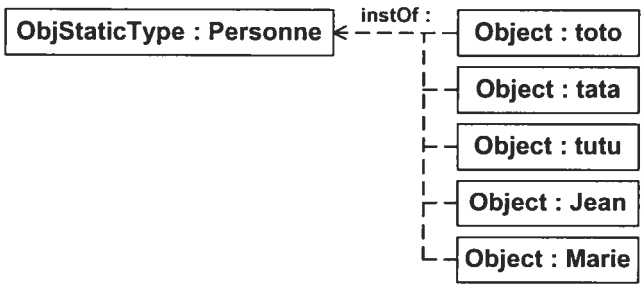


Figure III-16 : Exemple d'objets

Comment peut-on donc modéliser les relations de type `travailler` *dans les scénarios (a) et (b) suivants ? (a) Les personnes* `toto`, `tata`, `tutu`, `Jean` *travaillent ensemble plusieurs fois, soit* n_a *fois. (b)* `toto`, `tata`, `tutu`, `Jean` *travaillent aussi ensemble plusieurs fois avec la personne* `Marie`, *soit* n_b *fois.*

Étudions le scénario (a). Le scénario (a) signifie qu'a lieu n_a *fois le fait «les personnes* `toto`, `tata`, `tutu`, `Jean` *travaillent ensemble». Ce fait est représenté comme «*`toto`, `tata`, `tutu`, *et* `Jean` *sont reliés par des arcs de type* `act` *à une relation de type* `travailler`*»; par exemple, (a1) «*`toto`, `tata`, `tutu`, *et* `Jean` *sont reliés à* `personneGroupe1-travailler`$_i$ *par des* `act`*» (Figure III-17), la relation* `personneGroupe1-travailler`$_i$ *est illustrée plus en détail dans la Figure III-18.*

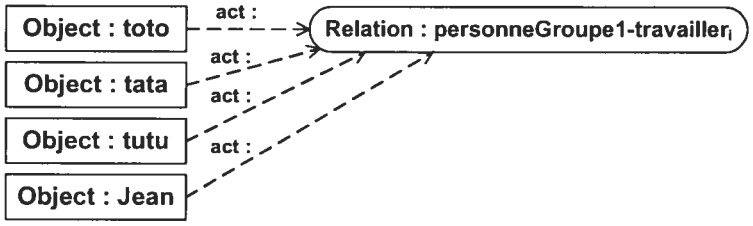


Figure III-17 : Modèle structurel de *personneGroupe1-travailler*$_i$

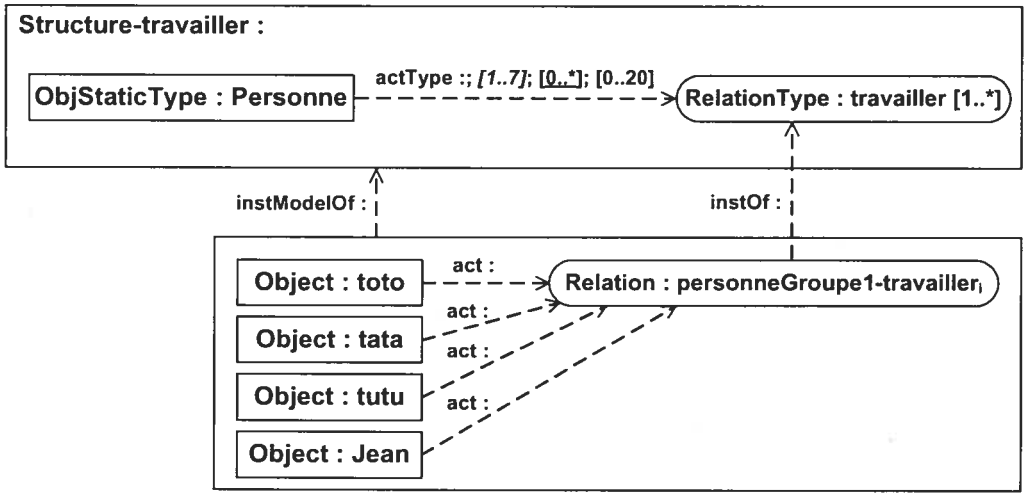


Figure III-18 : Exemple d'une relation de type *travailler*

Donc, le scénario (a) est représenté par n_a relations comme `personneGroupe1-travailler`$_i$ *dans (a1), c'est-à-dire chacune d'elles est instanciée du type* `travailler` *et à celle-ci les éléments* `toto`, `tata`, `tutu`, `Jean` *sont reliés par des arcs de type* `act`. *Étant donné que* `toto`, `tata`, `tutu`, *et* `Jean` *sont toutes les instances de* `PersonneGroupe1`, *la structure de* `relArc` *de* `Type` *à* `Relation` *permet d'interpréter (a1) comme (a2) «*`PersonneGroupe1` *est relié à* `personneGroupe1-travailler`$_i$ *par un arc de type* `act`*» (Figure III-19), autrement dit, (a1) et (a2) représentent au fond une même chose. Donc, le scénario (a) est représenté par n_a relations comme* `personneGroupe1-travailler`$_i$ *dans (a1) ou bien dans (a2). D'ailleurs, la représentation de (a2) (cf. Figure III-19) est visiblement plus simple que celle de (a1) (cf. Figure III-17). Alors, il est mieux de modéliser le scénario (a) par n_a relations comme* `personneGroupe1-travailler`$_i$ *dans (a2) que dans (a1). Il est à noter que ceci est possible seulement si* `relArc` *peut relier* `Type` *à* `Relation` *, ou bien si notre métamodèle autorise à un type de s'engager à une relation.*

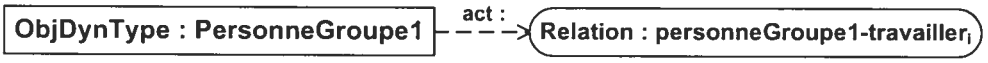


Figure III-19 : Équivalent de la Figure III-17 – Type participant à une relation

La structure de `relArc` *de* `Type` *à* `Relation` *prouve également son avantage dans la représentation du scénario (b). Ce scénario s'agit qu'a lieu n_b fois le fait «les personnes* `toto`, `tata`, `tutu`, `Jean`, *et* `Marie` *travaillent ensemble». Il est représenté donc par n_b relations comme* `personneGroupe2-travailler`$_i$ *dans (b1) «*`toto`, `tata`, `tutu`, `Jean`, *et* `Marie` *sont reliés à* `personneGroupe2-travailler`$_i$ *par des arcs de type* `act`*» (Figure III-20); c'est-à-dire ces relations sont de type* `travailler` *et à chacune d'elles les éléments* `toto`, `tata`, `tutu`, `Jean`, *et* `Marie` *sont reliés par des arcs de type* `act`.

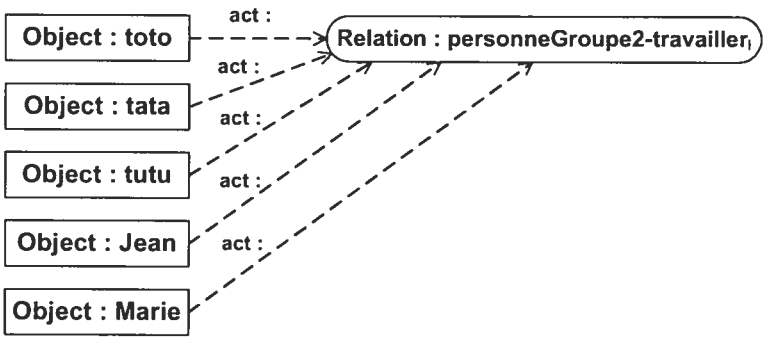


Figure III-20 : Modèle structurel de *personneGroupe2-travailler*$_i$

Comme nous avons analysé dans le scénario (a), les situations telles que (a1) et (a2) représentent en fait une même chose, ce qui implique que (b1) représente la même chose que (b2) «`PersonneGroupe1` *et* `Marie` *sont reliés à* `personneGroupe2-travailler`$_i$ *par*

des arcs de type `act`*» (Figure III-21). Alors, le scénario (b) est représenté par* n_b *relations comme* `personneGroupe2-travailler`$_i$ *dans (b1) ou bien dans (b2). De plus, la représentation de (b2) (cf. Figure III-21) est bien plus simple que celle de (b1) (cf. Figure III-20), il est mieux de modéliser (b) par* n_b *relations comme* `groupe2-travailler`$_i$ *dans (b2) que dans (b1).*

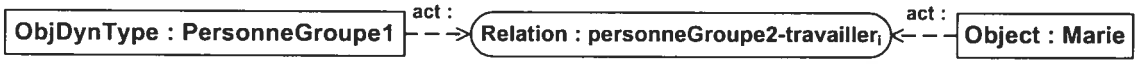


Figure III-21 : Équivalent de la Figure III-20 – Type participant à une relation

relArc/act/obj pour la représentation de types de relations

Structure 3 (cf. la Figure 125-(3), page 129): `relArc` peut unir `TypeInstance` à `RelationType`. Ceci permet d'indiquer des instances particulières impliquées dans un type de relations. Le fait qu'une instance est attachée par un arc de type `relArc` (respectivement `act` ou `obj`) à un type de relations signifie que cette première est rattachée par un arc de type `relArc` (respectivement `act` ou `obj`) à chaque relation instanciée de ce type de relations.

Structure 4 (cf. la Figure 125-(4), page 129): `relArc` peut unir `Type` à `RelationType`. Ceci permet d'indiquer des types impliqués entièrement dans une relation. Le fait qu'un type est rattaché par un arc de type `relArc` (respectivement `act` ou `obj`) à une relation représente que toutes les instances de ce premier sont rattachées par des arcs de type `relArc` (respectivement `act` ou `obj`) à chacune des relations instanciées de ce type de relations.

Rôle participant à un type de relations (exemple 49)

Continuons à examiner le type de relations `travailler-pour` *dans l'exemple 46 (page iii-11). Soit* `EmployeurTeximus` *l'élément désignant le rôle employeur (*`Employeur`*) de la compagnie ou l'organisation Teximus.* `Employé-EmployeurTeximus` *désigne le groupe de tous les rôles employé (*`Employé`*) qui travaillent pour* `EmployeurTeximus` *seul dans le sens que chacun de ces rôles employé participe à une ou des relations de type* `travailler-pour` *où* `EmployeurTeximus` *est le seul employeur. Comment peut-on modéliser le type* `Employé-EmployeurTeximus` *?*

Étant donné qu'un rôle employé peut ne pas appartenir pendant toute son existence au type `Employé-EmployeurTeximus`*,* `Employé-EmployeurTeximus` *est un sous-type de* `Employé` *et est conforme à* `RoleDynType`*. Chaque instance de* `Employé-EmployeurTeximus` *doit participer (comme un acteur de l'action) à une relation de type*

travailler-pour avec le seul rôle employeur EmployeurTeximus (comme l'objet de l'action). À noter que cette relation peut impliquer plusieurs instances de Employé-EmployeurTeximus. En appliquant la structure de relArc qui unit TypeInstance à RelationType, ceci peut être spécifié à niveau des types par la structure de travailler-pour comme suit (Figure III-22). Employé-EmployeurTeximus hérite de son parent Employé le type de relations travailler-pour. Et plus restrictivement, à ce type de relations, l'élément Employé-EmployeurTeximus est relié par un arc de type actType alors que EmployeurTeximus est relié par un arc de type obj. Cette structure doit respecter les contraintes attachées à ses structures mères dont la structure initiale de travailler-pour (cf. Figure III-14, page iii-12).

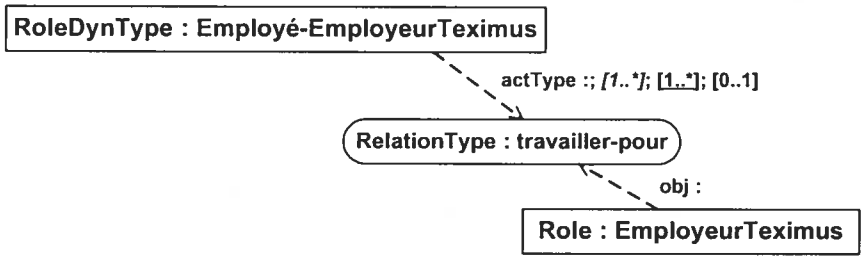


Figure III-22 : Type relationnel entre instance et type

6. Méta-arcs pour la spécification de types et instances de types de M1

III.3.18 forType

forType est abstrait et regroupe les méta-arcs relatifs à la spécification des types de M1. forType est subdivisé en: relArcType, actAsType, playedByType, composeType et chrcType.

III.3.19 forInst

forInst est abstrait et regroupe les méta-arcs relatifs aux TypeInstance et correspondants aux méta-arcs composeType, playedByType et chrcType. forInst est subdivisé en: compose, playedBy et chrc.

III.3.20 composeType, compose

composeType peut unir ObjType à ObjType lui-même (cf. la Figure 107, page 120) pour spécifier la composition des types d'objets. Les arcs de type composeType associent les types d'objets composants aux types d'objets composés correspondants.

compose peut unir Object à Object lui-même (cf. la Figure 107, page 120) pour représenter la composition des objets. Les arcs de type compose associent les objets composants aux objets composés correspondants. À chaque moment, un objet peut avoir

plusieurs composants mais être composant d'au plus un objet. Le méta-arc `compose` est antiréflexif et acyclique (cf. les Règle 6 et Règle 7, Annexe IV).

III.3.21 playedByType

`playedByType` (cf. la Figure 109, page 120) représente l'ensemble des arcs servant à indiquer des types de joueurs de rôles (`RolePlayerType`) pour un type de rôles.

Occupation de types de rôles (exemple 50)

La Figure III-23 spécifie que les personnes (`Personne`) peuvent jouer des rôles employé *(`Employé`) et* employeur *(`Employeur`) avec les contraintes présentées ci-après.*

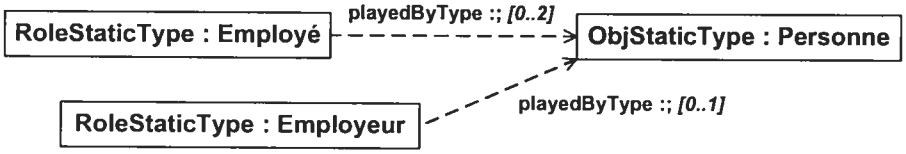


Figure III-23 : Occupation de types de rôles

L'occupation de rôles employé *pour des personnes est représenté par un arc de type `playedByType` de `Employé` à `Personne`. Une personne peut s'occuper en même temps deux rôles* employé *au maximum. Ceci est indiqué par les contraintes de l'arité notées [0..2] sur l'arc de type `playedByType` de `Employé` à `Personne`.*

L'occupation de rôles employeur *pour des personnes est représenté par un arc de type `playedByType` de `Employeur` à `Personne`. Une personne peut jouer ou non un rôle* employeur. *Et elle ne peut pas s'occuper plus d'un rôle* employeur *en même temps. Ceci est représenté par les contraintes de l'arité notées comme [0..1] sur l'arc de type `playedByType` de `Employeur` à `Personne`.*

III.3.22 playedBy

Structure 1 (cf. la Figure 110-(1), page 121): `playedBy` peut unir `Role` à `RolePlayer`. Un rôle est associé à son joueur de rôles par un arc de type `playedBy`. Un joueur de rôles peut jouer plusieurs rôles en même temps. Par contre, un rôle peut être chargé à chaque moment par un joueur de rôles au maximum. Le méta-arc `playedBy` est antiréflexif et acyclique (cf. les Règle 6 et Règle 7, Annexe IV).

Structure 2 (cf. la Figure 110-(2), page 121): `playedBy` peut unir `RoleType` à `RolePlayer`. Si un type de rôles *`TR`* est associé par un arc de type `playedBy` à un joueur de rôles *`JR`*, alors *`JR`* est le joueur unique pour tous les rôles de type *`TR`*.

III.3.23 chrcType, isKey

`chrcType` représente l'ensemble des arcs qui associent des attributs à des types de

M1 (cf. la Figure 128, page 132). `chrcType` est antiréflexif (cf. la Règle 6, Annexe IV).

Une instance d'un type peut ne pas avoir, ou peut avoir un ou plusieurs attributs d'un même type. Ceci est spécifié par les contraintes d'arités sur des attributs pour des types à caractériser (cf. la Définition 10, page 131). Si l'arité minimale ou maximale pour arc de type `chrcType` n'est pas indiquée, la valeur pour cette arité est 1.

`isKey` (cf. la Figure 129, page 132) unit `chrcType` à `Boolean` afin de désigner les attributs clés ou ordinaires pour un type (cf. la Définition 10, page 131). Si un arc de type `chrcType` qui attache un attribut `A` à un type `T` est relié par un arc de type `isKey` à `True`, alors `A` est un attribut clé de `T`. Un attribut peut être un attribut clé pour un type mais un attribut ordinaire pour un autre type.

Spécification des attributs pour un type (exemple 51)

*Soit qu'une personne (`Personne`) est caractérisée par les attributs suivants : un seul nom (`Nom`); un seul prénom (`Prénom`); aucun, un ou plusieurs surnoms (`Surnom`); une seule date de naissance (`DateDeNaissance`) constituée de jour (`Jour`), mois (`Mois`) et année (`Année`). La Figure III-24 illustre la représentation de `Personne`, un type statique d'objets. Les attributs `Nom`, `Prénom`, `Surnom`, `DateDeNaissance`, `Jour`, `Mois`, et `Année` sont attachés à leur méta-élément `Attribute` par des arcs de type `meta`. L'attribut `DateDeNaissance` est constitué des trois attributs simples `Jour`, `Mois`, et `Année` qui lui sont reliés, comme ses attributs uniques, par des arcs de type `chrcType`. `Nom`, `Prénom` et `DateDeNaissance` sont reliés par des arcs de type `chrcType` à `Personne` comme les attributs uniques de `Personne`, alors que `Surnom` est relié à `Personne`, comme un attribut multiple, par un arc de type `chrcType` avec les arités minimale et maximale de valeurs 0 et *.*

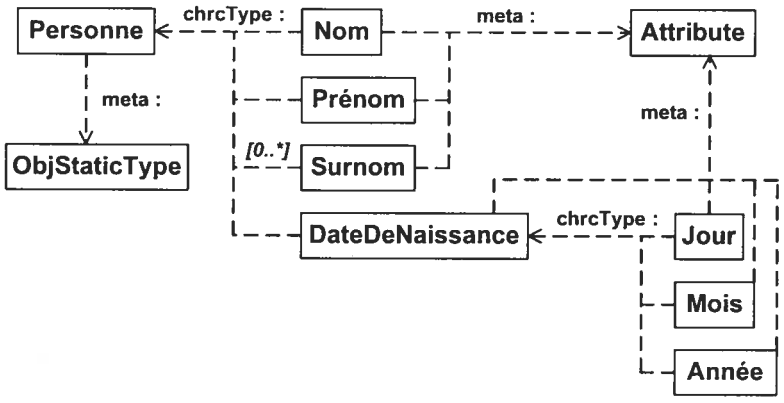


Figure III-24 : Spécifier des attributs pour un type

Voici un autre exemple. Supposons qu'un passeport (`Passeport`) est caractérisé non seulement par son numéro d'identification (`IdNuméro`) de type «attribut clé», mais encore par

d'autres attributs tels que le nom de son titulaire (`NomTitul`), le nom du pays livrant, etc. (cf. l'exemple 52, page iii-24). La Figure III-25 illustre la caractérisation du type `Passeport` avec les deux attributs `IdNuméro` et `NomTitul`.

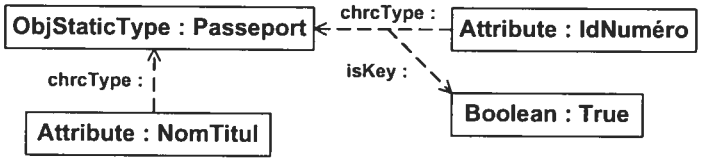


Figure III-25 : Attributs clés pour un type

Ces types `Personne` et `Passeport` avec leurs attributs peuvent être visualisés d'une manière simplifiée comme dans la Figure III-26. Cette figure montre aussi la représentation simplifiée d'autres types statiques/dynamiques d'objets/rôles, ou types de relations tels que les types `Retraité`, `Employé`, `PermEmp`, et `travailler-pour`. Un employé (`Employé`) est caractérisé par son numéro d'employés (`IdEmp`) de type «attribut clé», et son salaire (`Salaire`). Un(e) retraité(e) (`Retraité`) bénéficie de sa pension de retraite (`Pension`). Une relation de type `travailler-pour` est caractérisée par deux dates : date de début (`DateDébut`) et date de fin (`DateFin`).

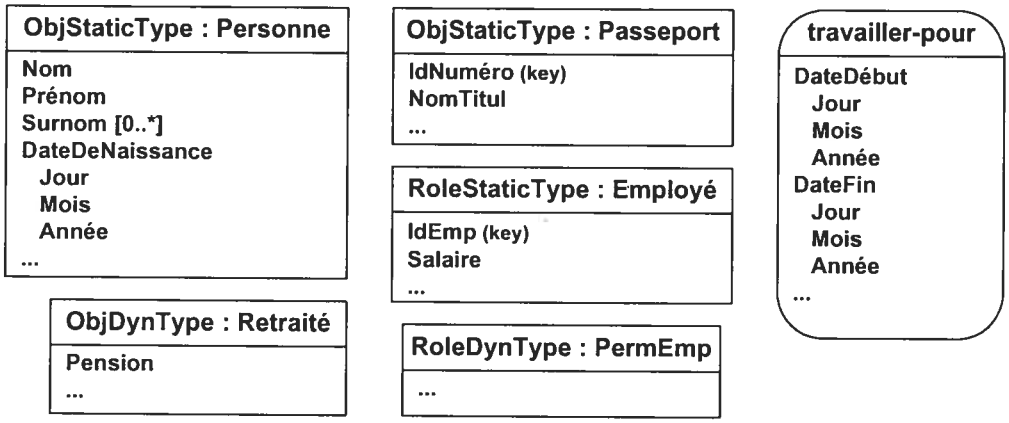


Figure III-26 : Représentation simplifiée de types avec leurs attributs

III.3.24 chrc

Structure 1 (cf. la Figure 132-(1), page 133): `chrc` peut unir `AttrInst` à `TypeInstance`. Des instances d'attributs ou bien valeurs d'attributs sont attachées à des instances à caractériser par des arcs de type `chrc`. Une instance peut avoir plusieurs valeurs d'attributs, et une valeur d'attributs peut caractériser plusieurs instances.

Instances d'un type avec leurs valeurs d'attributs (exemple 52)

Étudions la situation suivante. Un passeport permet d'identifier une personne comme citoyen d'un pays. Quoique chaque pays livre ses propres types de passeports, un passeport

(Passeport) a un et un seul numéro d'identification (IdNuméro), et doit comporter le nom de son titulaire (NomTitul). Soient les critères suivants à considérer : (i) deux Passeport différents ont deux IdNuméro différents; (ii) les NomTitul des Passeport d'une même personne doivent les mêmes, et certains dossiers administratifs sur une personne conservent les informations sur le ou les Passeport (telles que les IdNuméro, les NomTitul, etc.) de cette personne. Comment les attributs tels que IdNuméro, NomTitul ainsi que leurs instances seront modélisés ?

Afin de satisfaire le critère (i), le type IdNuméro sera modélisé comme un attribut clé du type Passeport. Tel qu'illustre la Figure III-27, le type Passeport est caractérisé par deux attributs, IdNuméro, NomTitul (ce qui est indiqué par des arcs de type chrcType). IdNuméro est l'attribut clé de Passport (ce qui est indiqué par un arc de type isKey).

Pour la même raison expliquée dans la section 2.1.4 (page 13), afin de modéliser les instances d'attributs (AttrInst), il y a seulement deux solutions. Les AttrInst sont traitées soit comme valeurs étiquetées dans la première solution et soit comme valeurs dans la deuxième.

a) Première solution: les AttrInst sont traitées comme valeurs étiquetées. C'est-à-dire, pour les instances d'un même attribut, ceux considérées comme identiques sont étiquetées identiquement; par contre, ceux considérées comme différentes sont étiquetées différemment.

La Figure III-27 illustre la spécification de Pca2005Jean, une instance de Passeport.

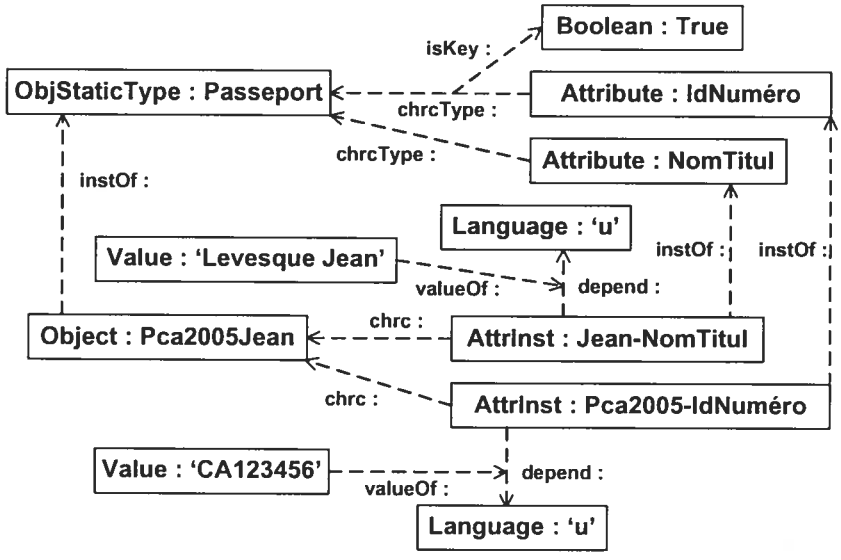


Figure III-27 : Objet *Pca2005Jean* et son type *Passeport*

Dans cette figure, l'élément Pca2005 représente un passeport (ce qui est indiqué par un arc de type instOf de Pca2005Jean à Passeport) dont le numéro d'identification est «CA123456» et le nom du titulaire est «Levesque Jean». Il est caractérisé par deux instances d'attributs (AttrInst) qui sont représentés par les éléments Pca2005-IdNuméro et

Jean-NomTitul (ce qui est indiqué par des arcs de type chrc) et qui prennent respectivement IdNuméro et NomTitul pour leurs types (ce qui est indiqué par des arcs de type instOf). Les éléments Pca2005-IdNuméro et Jean-NomTitul sont interprétés en langage universel respectivement comme «CA123456» et «Levesque Jean». La Figure III-28 présente une représentation simplifiée de Pca2005Jean.

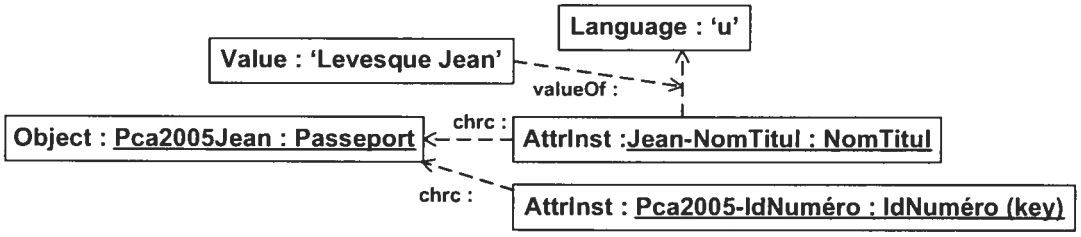


Figure III-28 : Représentation simplifiée de *Pca2005Jean*

Supposons que deux passeports Pca2005Jean (NomTitul = 'Levesque Jean' (u); IdNuméro ='CA123456' (u)) et Pfr2005Jean (NomTitul = 'Levesque Jean' (u); IdNuméro ='FR123456' (u)) sont à la personne Jean. Alors les NomTitul des Pca2005Jean et Pfr2005Jean sont toujours identiques, et peuvent être représentés par une même instance d'attributs: Jean-NomTitul. Comme l'illustre la Figure III-29, Pfr2005Jean est caractérisé par deux instances d'attributs (AttrInst), Pfr2005-IdNuméro et Jean-NomTitul, instanciées respectivement de IdNuméro et de NomTitul. L'élément Pfr2005-IdNuméro est interprété en langage universel comme «FR123456».

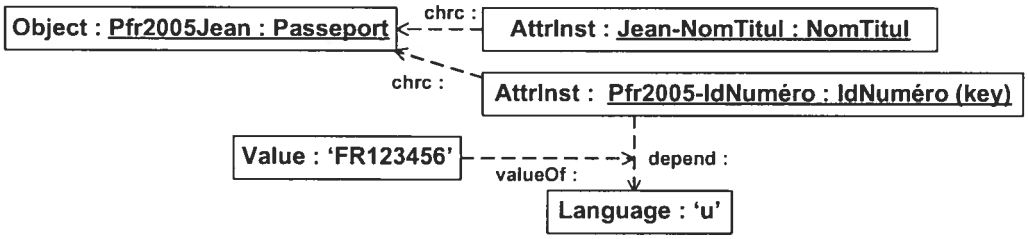


Figure III-29 : Représentation simplifiée de *Pfr2005Jean*

Soit Pfr2006Toto un passeport de la personne Toto avec les informations suivantes: Pfr2006Toto (NomTitul = 'Levesque Jean' (u); IdNuméro ='FR123490' (u)). Comme Jean et Toto sont deux personnes différentes, le fait que les NomTitul de leurs passeports ont une même valeur 'Levesque Jean' est une coïncidence. Alors le NomTitul des passeports de Jean et celui de Toto sont considérés comme deux instances d'attributs (AttrInst) différentes et sont donc étiquetés différemment. Comme l'illustre la Figure III-30, Pfr2006Toto est caractérisé par deux instances d'attributs (AttrInst), Pfr2006-IdNuméro (une instance de IdNuméro) et Toto-NomTitul (une instance de NomTitul). Les éléments Pfr2006-IdNuméro et Toto-NomTitul sont interprétés en langage universel respectivement comme

«FR123490» et «Levesque Jean».

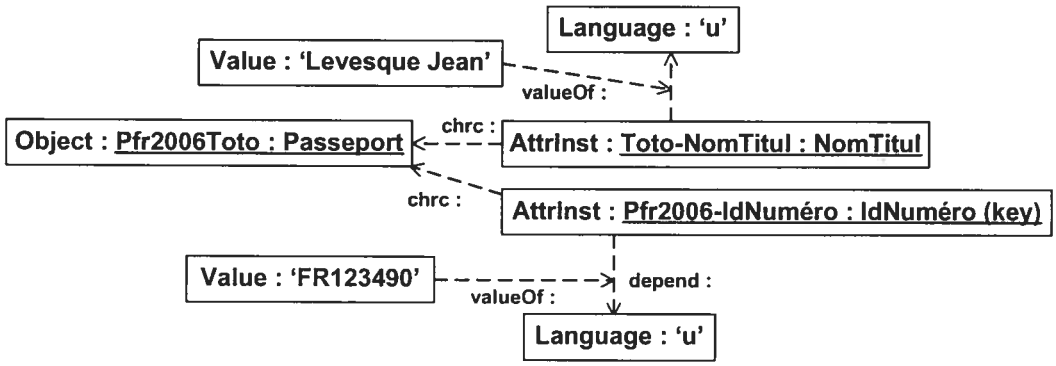


Figure III-30 : Représentation simplifiée de *Pfr2006Toto*

b) Deuxième solution: les `AttrInst` *sont traitées comme valeurs. L'inconvénient de cette solution se réside dans le support du critère (ii) dans cet exemple. À chaque fois que le* `IdNuméro` *(ou le* `NomTitul`*, etc.) d'un passeport est mis à jour à une nouvelle valeur, nous devons mettre à jour également à cette valeur le* `IdNuméro` *(ou le* `NomTitul`*, etc.) de ce passeport dans les dossiers concernés pour que ces insformations soient identiques. Plus le nombre de mises à jour de ce genre augmente, plus l'inconvénient se manifeste. Visiblement, la première solution présentée ci-haut corrige cet inconvénient.*

Ainsi, nous modélisons les instances d'attributs (`AttrInst`*) comme valeurs étiquetées.*

Les éléments `Pca2005Jean`*,* `Pfr2005Jean` *et* `Pfr2006Toto` *décrits ci-haut peuvent être représentés d'une manière simplifiée comme dans la Figure III-31. Similairement,* `Jean` *dans la Figure III-32 représente une personne concrète avec ses attributs en suivant la structure de son type* `Personne` *étudiée dans l'exemple 51 (page iii-23).*

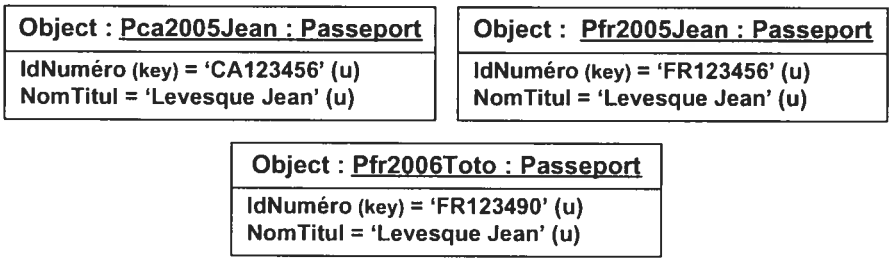


Figure III-31 : Représentation simplifiée de *Pca2005Jean*

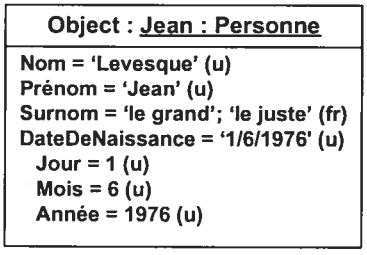


Figure III-32 : Représentation simplifiée de l'objet *Jean*

Structure 2 (cf. la Figure 132-(2), page 133): `chrc` peut unir `AttrInst` à `Type`.

Si une instance `I` d'un attribut `A` est liée par un arc de type `chrc` à un type `T`, toutes les instances de `T` partagent `I` comme la valeur pour l'attribut `A`. Un type peut avoir plusieurs instances d'attributs. Une instance d'attributs peut caractériser plusieurs types.

Concrétiser des attributs d'un type au niveau M1 (exemple 53)

La Figure III-33 illustre la concrétisation d'attributs d'un type de M1. Dans cette figure, `PorteFenêtre` représente l'ensemble de toutes les portes de fenêtres. Une porte de fenêtres (`PorteFenêtre`) est caractérisée par la couleur (`Couleur`), la largeur (`Largeur`) et la hauteur (`Hauteur`), ce qui est indiqué par des arcs de type `chrcType` reliant `Couleur`, `Largeur`, et `Hauteur` à `PorteFenêtre`. `PorteFenêtre-GroupeA` est un sous-type de `PorteFenêtre` (ce qui est indiqué par un arc de type `subType` en eux) et représente un ensemble de toutes les portes fenêtres d'une même dimension, la hauteur (`PFGroupeA-Hauteur`) soit 1m64 et la largeur (`PFGroupeA-Largeur`) soit 2m10. Cette figure peut être simplifiée comme la Figure III-34.

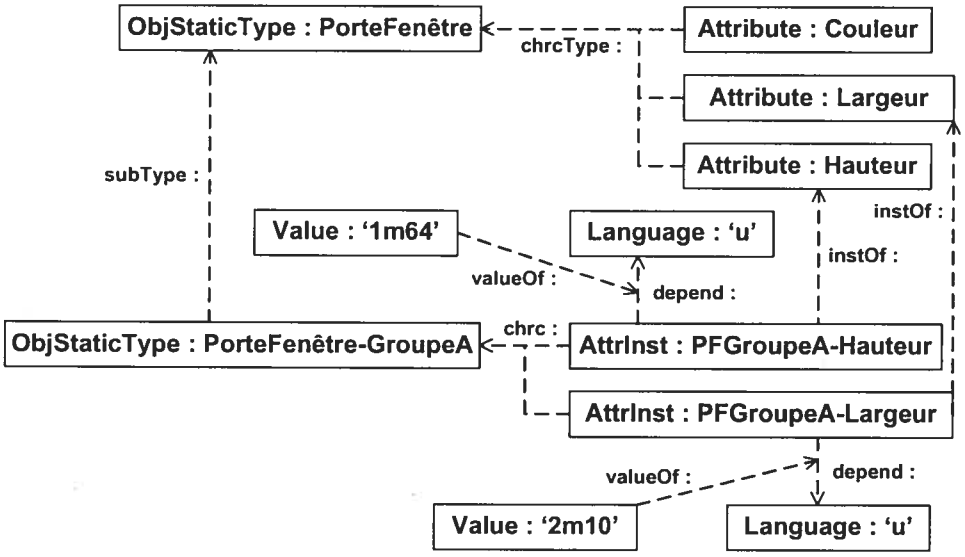


Figure III-33 : Concrétiser des attributs d'un type

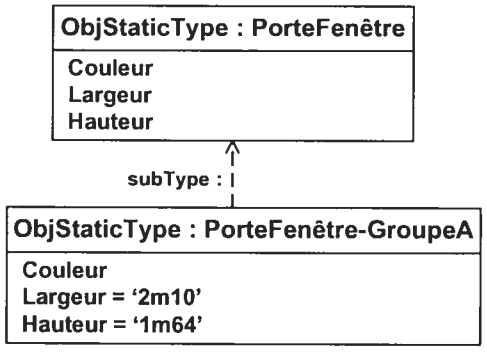


Figure III-34 : Représentation simplifiée de la Figure III-33

III.3.25 dataOfType

`dataOfType` (cf. la Figure 130, page 133) représente l'ensemble des arcs à indiquer les types de données primitives (`DataType`) pour les attributs (`Attribute`). Un attribut peut être attaché à un seul type de données primitives. Si un attribut *A* est attaché à un type de données primitives *TD*, la valeur d'une instance de type *A* va être vu comme instance de type *TD*. Par exemple, `Nom` est un attribut attaché au type de données primitives `String`, la valeur d'une instance de type `Nom` est donc vue comme une chaîne de caractères telle que «Anna Trembley», «Papin», «Levesque», etc.

III.3.26 isAbst

`isAbst` (cf. la Figure 135, page 135) représente l'ensemble des arcs à indiquer au niveau M1 les types abstraits et les types non abstraits. Par défaut, un type (`Type`) est non abstrait. Un type relié à `True` par un arc de type `isAbst` sera un type abstrait.

7. Hiérarchisation entre types de M1

III.3.27 subType, inherit

`subType` implémente la relation de sous-typage multiple alors que `inherit` implémente l'héritage multiple entre les types de M1 autres que les modèles : entre les `ObjType`, entre les `RoleType`, entre les `RelationType`, et entre les `Attribute`. Voir aussi la section 5.3.4.7 (page 140) pour les spécifications sur `subType` et `inherit`. Les méta-arcs `subType` et `inherit` doivent remplir les Règle 5, Règle 6 et Règle 7 (Annexe IV). Concernant l'effet de sous-typage au niveau M1, les Règle 1, Règle 8, Règle 16, et Règle 28 (Annexe IV) sont appliquées. Pour l'effet de l'héritage, d'autres règles sont retenues : Règle 1, Règle 8, Règle 16, et Règle 29 (Annexe IV).

Ci-après sont des exemples de sous-typage entre des types de M1.

Classification de types d'objets (exemple 54)

Reprenons l'exemple de classification des véhicules (cf. l'exemple 2, page 28). Les véhicules (`Véhicule`) sont classés statiquement en types en fonction de divers critères. Par exemple, selon le nombre de portes, il y a les types-ci: les véhicules 2 portes (`Véhicule-2portes`); les 4 portes (`Véhicule-4portes`); ou les mini-fourgonnettes (`Véhicule-Van`). En fonction du mode de propulsion, nous avons ces deux types : les véhicules à 2 roues motrices (`Véhicule-2RM`) ou à 4 roues motrices (`Véhicule-4RM`). En basant sur l'origine des véhicules, nous avons trois autres suivants: les véhicules d'Europe (`Véhicule-Europe`); ceux d'Amérique (`Véhicule-Amérique`); et ceux d'Asie (`Véhicule-Asie`). Pour chacun

des types listés (cf. la Figure 12, page 29), il est à remarquer que si un véhicule est qualifié pour appartenir au type en question, alors durant tout au long de son existence, ce véhicule appartient toujours à ce dernier. Ceci signifie que ces types sont tous de nature de types statiques d'objets. Ces types peuvent être modélisés dans notre formalisme comme ce que la Figure III-35 présente.

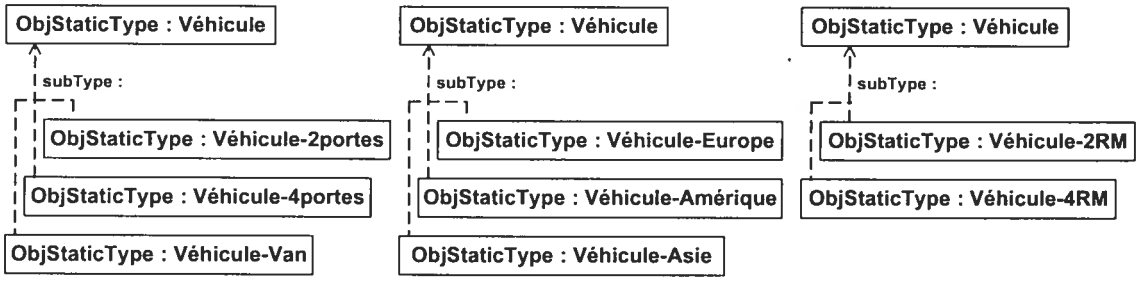


Figure III-35 : Différentes hiérarchies statique de véhicules basées sur les trois critères

Les véhicules peuvent aussi être classés dynamiquement en types en fonction de leurs états. Par exemple, à chaque moment, un véhicule est dans l'état en marche ou en panne, respectivement à son appartenance temporairement au type de véhicules en marche (`Véhicule-enMarche`) ou au type de véhicules en panne (`Véhicule-enPanne`). Donc, les types `Véhicule-enMarche` et `Véhicule-enPanne` sont de nature de types dynamiques d'objets, et sont des sous-types dynamiques de `Véhicule`. Ceci est modélisé dans notre formalisme comme dans la Figure III-36.

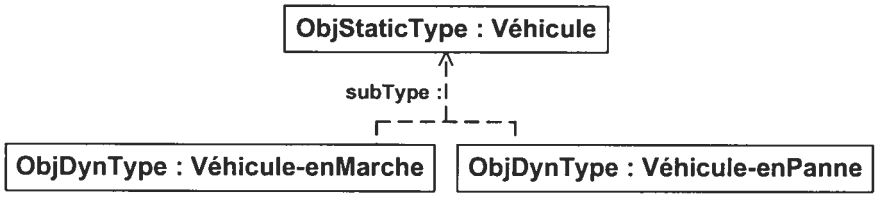


Figure III-36 : Une hiérarchie dynamique de véhicules basée sur l'état de véhicules

La Figure III-37 présente un exemple de sous-typage multiple. L'ensemble de tous les véhicules 2 portes, à 2 roues motrices et d'Amérique est représenté par un type statique d'objets, `Véhicule-Amérique-2RM-2portes`, qui est sous-type des types `Véhicule-Amérique`, `Véhicule-2RM`, et `Véhicule-2portes`. Voici un autre exemple. L'ensemble de tous les véhicules 2 portes, à 2 roues motrices, d'Amérique et en marche est représenté par un type dynamique d'objets, `Véhicule-Amérique-2RM-2portes-enMarche`, qui est sous-type des types `Véhicule-Amérique`, `Véhicule-2RM`, `Véhicule-2portes`, et `Véhicule-enMarche`. Le type `Véhicule-Amérique-2RM-2portes-enMarche` peut aussi être spécifié comme sous-type de `Véhicule-Amérique-2RM-2portes` et de `Véhicule-enMarche` (Figure III-38).

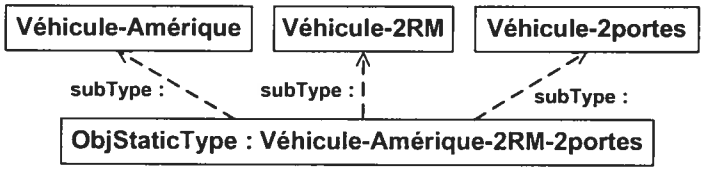


Figure III-37 : Sous-typage multiple – Type statique

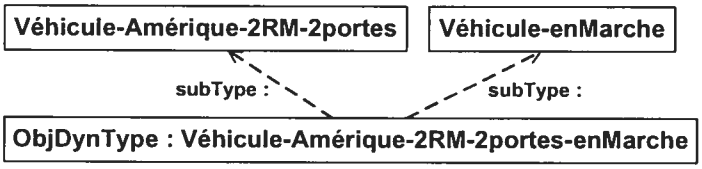


Figure III-38 : Sous-typage multiple – Type dynamique

Classification de types de rôles (exemple 55)

Par exemple, les rôles «employé» (`Employé`) sont divisés en catégories telles que les rôles «employé temporaire» (`TempEmp`), et les «employé permanent» (`PermEmp`) (Figure III-39).

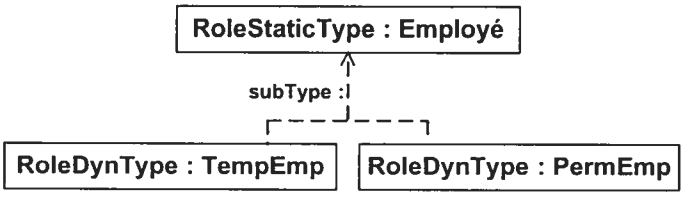


Figure III-39 : Hiérarchie dynamique de rôles

***Notation 9** : Notations graphiques pour les liens de sous-typage*

Un lien de sous-typage entre deux types, soit de `A` à `B`, peut être représenté simplement par une flèche de spécialisation (comme dans UML) de `A` à `B`. Dans les représentations graphiques, afin de distinguer plus clairement la classification statique et la classification dynamique des types d'objets (respectivement de rôles), un lien de sous-typage entre un type dynamique d'objets (respectivement de rôles) et un type d'objets (respectivement de rôles), soit de `A` à `B`, peut être représenté par une flèche de spécialisation de `A` à `B`, en pointillé.

Par exemple, les hiérarchies de spécialisations des véhicules dans les Figure III-37, Figure III-38, et Figure III-39 peuvent être illustrées respectivement comme dans les Figure III-40, Figure III-41 et Figure III-42.

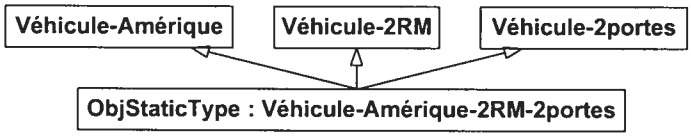


Figure III-40 : Représentation simplifiée de la Figure III-37

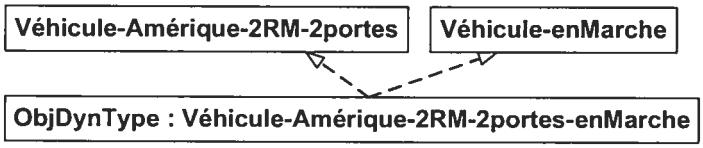


Figure III-41 : Représentation simplifiée de la Figure III-38

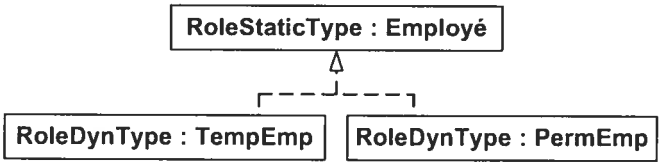


Figure III-42 : Représentation simplifiée de la Figure III-39

Classification de types de relations (exemple 56)

La situation présentée ici porte sur la classification de types de relations présentant des activités de mouvement. Les actions sur le déplacement (`se-déplacer`*) peuvent être classées en différents types, par exemple: marcher (*`marcher`*), se déplacer dans l'air (*`se-déplacer-dans-air`*), etc. Quant aux actions de voler (*`voler`*), il s'agit d'une combinaison des actions de se déplacer dans l'air (*`se-déplacer-dans-air`*) et de se soutenir dans l'air (*`se-soutenir-dans-air`*). Comme l'illustre la Figure III-43, le type de relations* `se-déplacer` *est super-type de différents autres types de relations dont* `marcher`, `se-déplacer-dans-air`, `voler`. *Le type de relations* `voler` *est sous-type des types de relations* `se-déplacer-dans-air` *et* `se-soutenir-dans-air`.

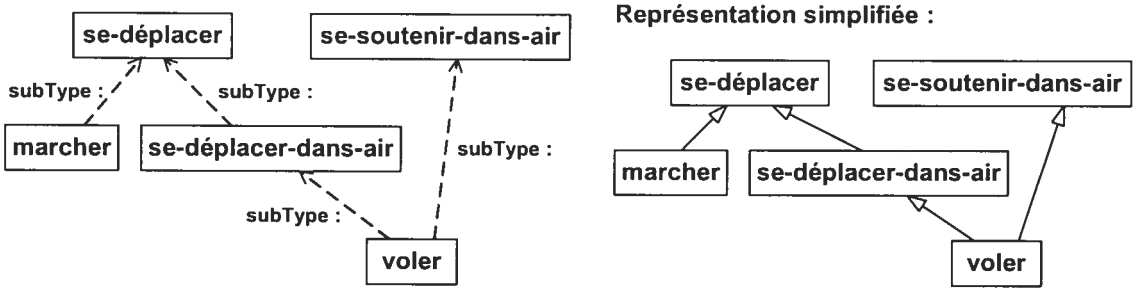


Figure III-43 : Exemple d'une hiérarchie de types de relations

Classification d'attributs (exemple 57)

Supposons qu'un attribut représentant une adresse en général (`Adresse`*) est composé des attributs suivants: le numéro (*`Numéro`*), le nom de la rue (*`NomRue`*), le nom de la ville (*`NomVille`*), le nom du province (*`NomProvince`*), le nom du pays (*`NomPays`*), et le code postal (*`CodePostal`*). Par exemple, l'adresse de la personne Anna, soit «4685, Edouard-Montpetit, Montréal, Québec, Canada, H3W1P6», est codée comme suit:* `Numéro` *= 4685;* `NomRue` *= Edouard-Montpetit;* `NomVille` *= Montréal;* `NomProvince` *= Québe;* `NomPays`

= *Canada; et* `CodePostal` = *H3W1P6.*

Supposons qu'un attribut représentant l'adresse d'un appartement (`AdresseAppt`*) est composé des attributs suivants: numéro (*`Numéro`*), nom de l'appartement (*`NomAppt`*), nom de la rue (*`NomRue`*), nom de la ville (*`NomVille`*), nom du province (*`NomProvince`*), nom du pays (*`NomPays`*), et code postal (*`CodePostal`*). Par exemple, l'adresse de Papin, soit «4283, appartement 2, Edouard-Montpetit, Montréal, Québec, Canada, H3W1P6», est codée comme suit:* `Numéro` = *4283;* `NomAppt` = *2;* `NomRue` = *Edouard-Montpetit;* `NomVille` = *Montréal;* `NomProvince` = *Québec;* `NomPays` = *Canada; et* `CodePostal` = *H3W1P6.*

Remarquons que l'adresse d'un appartement est une adresse en général mais ajoutée du nom de l'appartement. Donc, l'attribut `AdresseAppt` *est un sous-type de l'attribut* `Adresse`*, et ces deux types peuvent être modélisés comme ce que la Figure III-44 présente.*

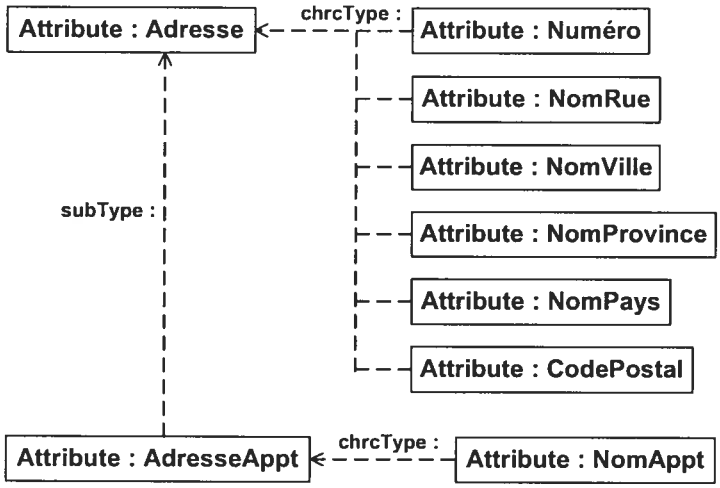


Figure III-44 : Exemple de spécialisation entre attributs

8. Spécification de la visibilité d'éléments de M1

III.3.28 visib

Structure 1 (cf. la Figure 146-(1), page 142): `visib` peut unir `Type` à `Visibility`.

Structure 2 (cf. la Figure 146-(2), page 142): `visib` peut unir `TypeInstance` à `Visibility`.

La sémantique de ces deux structures: Le fait qu'un type (respectivement une instance) x est attaché(e) à une visibilité v spécifie que v représente la visibilité de x dans le modèle dans lequel x est défini(e).

Structure 3 (cf. la Figure 146-(3), page 142): `visib` peut unir `chrcType` à `Visibility`. Le fait qu'un arc de type `chrcType` d'un attribut (`Attribute`) A à un type (`Type`) T soit relié par un arc de type `visib` à une visibilité (`Visibility`) v

représente que *v* est la visibilité de l'attribut *A* pour le type *T*. Un attribut peut avoir différentes visibilités : une visibilité est propre pour un type que l'attribut caractérise.

9. Représentation de cycles d'états

III.3.29 forCycle, cycleOf, startOf, prevOf, endOf

`forCycle` est abstrait et regroupe les méta-arcs servant à spécifier des cycles d'états (c'est-à-dire les `ObjCycle` et les `RoleCycle`). `forCycle` est représenté par les méta-arcs suivants: `cycleOf`, `startOf`, `prevOf`, et `endOf`.

`cycleOf` (cf. la Figure 151-(1a),(1b), page 144) représente l'ensemble des arcs à attacher les cycles d'états aux types d'objets ou de rôles correspondants.

Un cycle d'états (`ObjCycle`) est défini pour un et un seul type d'objets (`ObjType`) auquel il est rattaché par un arc de type `cycleOf`. Un cycle d'états (`RoleCycle`) est défini pour un et un seul type de rôles (`RoleType`) auquel ce premier est rattaché par un arc de type `cycleOf`. Plusieurs `ObjCycle` (respectivement `RoleCycle`) peuvent être rattachés par des arcs de type `cycleOf` à un même `ObjType` (respectivement `RoleType`). Les `ObjDynType` (`RoleDynType`) dans un `ObjCycle` (`RoleCycle`) pour un `ObjType` (`RoleType`) représentent les types d'états que ce dernier peut passer suivant l'ordre de ces types d'états spécifié par le `ObjCycle` (`RoleCycle`) en question.

`startOf` (respectivement `endOf`) représente l'ensemble des arcs à indiquer des états initiaux (respectivement finaux) dans les `ObjCycle` ou `RoleCycle`. Un `ObjDynType` (`RoleDynType`) initial d'un `ObjCycle` (`RoleCycle`) est attaché à ce `ObjCycle` (`RoleCycle`) par un arc de type `startOf` (cf. la Figure 151-(2a),(2b), page 144). Similairement, un `ObjDynType` (`RoleDynType`) final d'un `ObjCycle` (`RoleCycle`) est attaché à ce `ObjCycle` (`RoleCycle`) par un arc de type `endOf` (cf. la Figure 151-(3a),(3b), page 144). Un `ObjCycle` (`RoleCycle`) a un et un seul `ObjDynType` (`RoleDynType`) initial, mais peut n'avoir aucun, avoir un ou plusieurs `ObjDynType` (`RoleDynType`) finaux. Nous avons la Notation 10.

Notation 10 *: Notations graphiques pour un type dynamique initial/final*

Dans la représentation simplifiée d'un cycle d'états, le type d'états initial est remarqué par une flèche de forme « ●➔ », et un final par « ●— ».

`prevOf` (cf. la Figure 151-(4a),(4b), page 144) représente l'ensemble des arcs pour spécifier l'ordre de transition entre les types d'états dans un cycle d'états. Dans un

`ObjCycle` (`RoleCycle`), si un `ObjDynType` (`RoleDynType`) est lié par un arc de type `prevOf` à un autre `ObjDynType` (`RoleDynType`), alors le premier est un prédécesseur du dernier ou bien le dernier est un successeur du premier. Un `ObjDynType` (`RoleDynType`) peut avoir plusieurs successeurs ou prédécesseurs. Autrement dit, selon un `ObjCycle` (`RoleCycle`), et dépendamment du contexte, un `ObjType` (`RoleType`) peut passer à un parmi des `ObjDynType` (`RoleDynType`) possibles immédiatement après son `ObjDynType` (`RoleDynType`) actuel.

Les types dynamiques `ObjDynType` (`RoleDynType`) appartenant à un `ObjCycle` (`RoleCycle`) pour un `ObjType` (`RoleType`) sont considérés comme des sous-types dynamiques, disjoints et complets du `ObjType` (`RoleType`) en question. Autrement dit, chaque instance du `ObjType` (`RoleType`) en question peut, en fonction de son état, être classifiée dynamiquement à ces sous-types; et en aucun moment, elle ne peut pas être classifiée dynamiquement à plus d'un de ces sous-types.

Modélisation de cycles d'états pour types d'objets (exemple 58)

Une personne, durant son existence, peut passer par différents états comme enfant, adolescent, adulte, célibataire, mariée, ou salarié, au chômage, en retraite, etc. Ces états se relient selon de différents cycles dont le cycle de vie, celui d'état civil, ou celui d'état d'emploi.

Selon le cycle de vie `Cycle-Vie` d'une personne, une vraie personne (représentée par une instance de `Personne`), correspondement à son état enfant, adolescent, adulte, ou personne âgée, peut aussi être classée dynamiquement dans un des groupes `Enfant`, `Adolescent`, `Adulte`, `PersonneÂgée`.. La Figure 150 (page 144) de l'exemple 19 (page 143) a illustré comment ce cycle de vie est spécifié. La Figure III-45 présente sa représentation simplifiée.

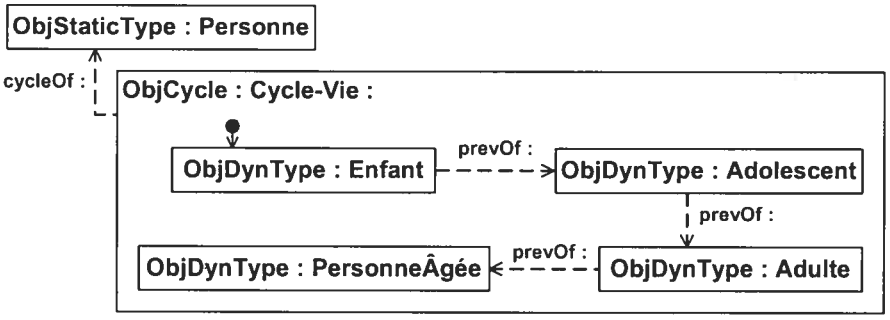


Figure III-45 : Version simplifiée de la Figure 150 (page 144)

De même, la Figure III-46 décrit le cycle d'état civil `Cycle-ÉtatCivil` pour `Personne`. Une personne à l'état célibataire, marié, veuf, ou divorcé peut être classée aux types `Célibataire`, `PersonneMariée`, `PersonneVeuve`, et `PersonneDivorcée` correspondants, et elle change entre ces états ou bien entre ces types selon l'ordre de transition

spécifié dans le cycle d'état `Cycle-ÉtatCivil`.

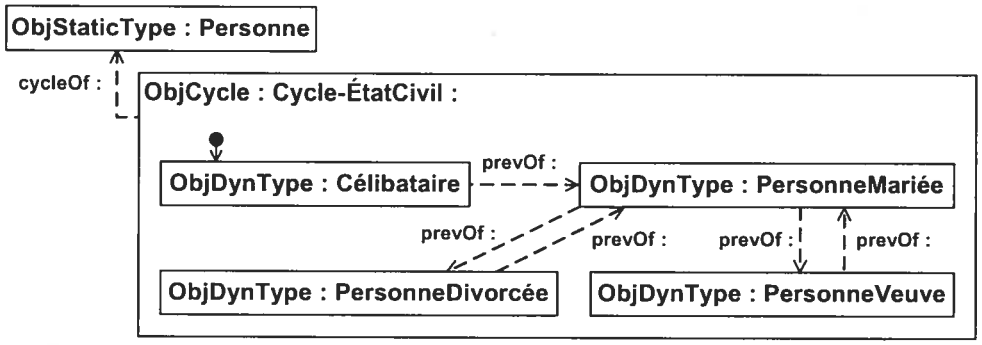


Figure III-46 : Cycle d'états civils d'une personne

À propos du cycle d'emploi, une personne commence plutôt avec un état de ne jamais avoir travaillé. Une fois qu'elle travaille et a son salaire, elle sera dans l'état salarié. D'ici, elle peut ensuite devenir sans-emploi ou bien au chômage, ou rester comme salariée jusqu'à sa retraite. Une personne sans-emploi peut trouver un travail à un moment donné et redevenir salariée, ou elle peut devenir en retraite. Il y a donc un cycle de transition entre deux états salarié et sans-emploi, ce qui implique durant la période de son premier état salarié à sa retraite, une personne peut passer à plusieurs reprises par l'état sans-emploi et/ou encore par l'état salarié. La Figure III-47 présente une modélisation de ce cycle d'emploi (`Cycle-ÉtatEmploi`*) avec les transitions entre les quatre états suivants: personne jamais travaillée (*`PersonneJamaisTravaillée`*), salarié (*`Salarié`*), personne sans-emploi (*`PersonneSansEmploi`*), et retraité (*`Retraité`*).*

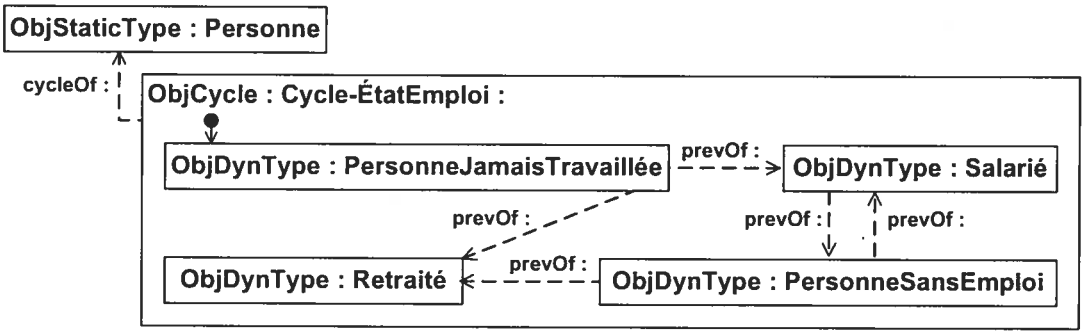


Figure III-47 : Cycle d'états d'emploi *Cycle-ÉtatEmploi*

Concrétisons ici plus autour de l'état sans-emploi d'une personne. Supposons que nous avons la politique suivante. Une personne salariée, une fois en état sans-emploi, bénéficiera d'abord une prestation de chômage dans une période. Elle peut terminer cette période en retournant à l'état salarié (lorsqu'elle retrouve un emploi) ou en devenant retraitée. Sinon, elle passera ensuite par la période où elle n'a pas de prestations de chômage. Aussi, elle peut terminer cette période en retournant à l'état salarié ou en devenant retraitée. Ainsi, le cycle de

sans-emploi pour une personne de l'état sans-emploi (PersonneSansEmploi) peut être représenté par une séquence: prestataire de chômage (PrestataireChômage), et puis non bénéficiaire de prestations de chômage (NonPrestataireChômage). Ce cycle peut être représenté par Cycle-PersonneSansEmploi qui est modélisé comme dans la Figure III-48 et est considéré comme un cycle d'états pour le type dynamique PersonneSansEmploi. Le contenu de la politique en question peut être modélisé comme ce qu'illustre la Figure III-49.

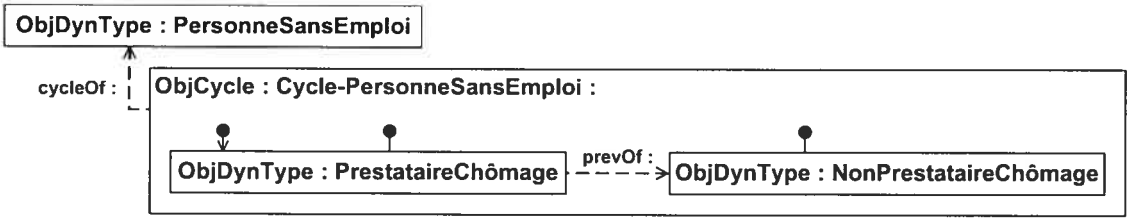


Figure III-48 : Cycle d'états du type dynamique *PersonneSansEmploi*

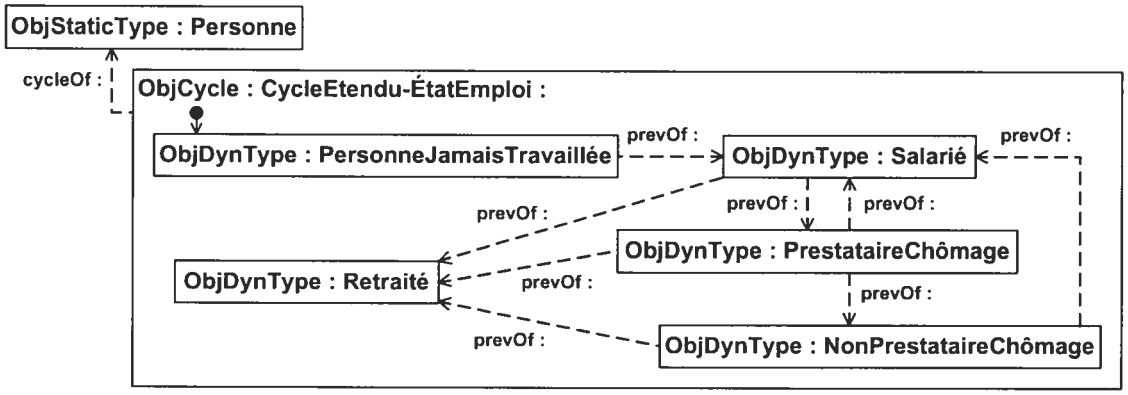


Figure III-49 : Intégration de *Cycle-ÉtatEmploi* et de *Cycle-PersonneSansEmploi*

Les cycles Cycle-Vie, Cycle-ÉtatCivil, Cycle-ÉtatEmploi, et Cycle-PersonneSansEmploi ont respectivement les types dynamiques initiaux suivants: Enfant, Célibataire, PersonneJamaisTravaillée, PrestataireChômage. Voir aussi la Notation 10 (page 34). Parmi ces quatre cycles, les trois premiers n'ont pas de types dynamiques finaux tandis que le dernier en a deux (PrestataireChômage et NonPrestataireChômage). PersonneJamaisTravaillée représente aussi le type dynamique initial dans le cycle CycEtendu-ÉtatEmploi décrit dans la Figure III-49.

Modélisation de cycles d'états pour types de rôles (exemple 59)

Voici quelques exemples de cycles d'états pour types de rôles. Un rôle employé dans un établissement, durant son existence, peut passer par divers états selon un cycle composé des états suivants: employé temporaire, employé permanent. Un rôle professeur dans une université, durant son existence, peut passer par divers états suivant un autre cycle composé des états suivants: professeur adjoint, professeur invité, professeur agrégé, et professeur titulaire.

La Figure III-50 illustre une représentation du cycle d'états `Cycle-Employé` *pour le type statique de rôles employé dans un établissement. Un rôle employé (*`Employé`*), durant son existence, peut passer par ces deux états: employé temporaire (*`TempEmp`*) et employé permanent (*`PermEmp`*).*

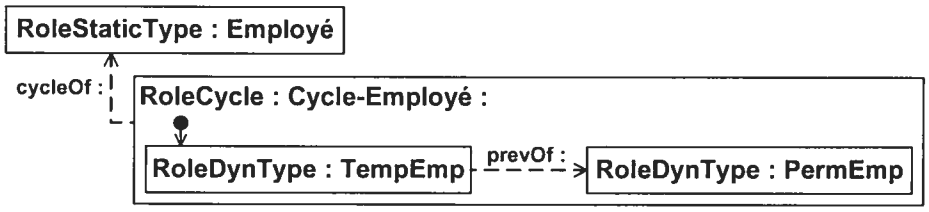


Figure III-50 : Cycle d'états d'employé *Cycle-ÉtatEmployé*

La Définition 26 spécifie le procédé pour intégrer deux cycles d'états, et souligne également le rôle qu'un `ObjDynType` (`RoleDynType`) initial et/ou final dans un cycle d'états joue dans l'intégration des cycles d'états. Par exemple, en suivant ce procédé, le cycle étendu d'emploi `CycEtendu-ÉtatEmploi` spécifié dans la Figure III-49 (page iii-37) résulte de l'intégration des cycles d'états `Cycle-ÉtatEmploi` (cf. la Figure III-47, page iii-36) et `Cycle-PersonneSansEmploi` (cf. la Figure III-48, page iii-37).

Définition 26 : Intégration de deux cycles d'états

Soit `Tdyn` un type dynamique dans un cycle d'états `Cyc`. Soit `Tdyn` ayant un cycle d'états `Cyc-Tdyn` dans lequel `I` représente le type dynamique initial, et dans lequel F_1, ..., F_n représentent le(s) finaux. Alors, l'intégration de `Cyc-Tdyn` dans `Cyc` deviendra un cycle d'états identique à `Cyc` excepté les conditions suivantes qui doivent être satisfaites:

- `Tdyn` sera supprimé,
- Les types dynamiques dans `Cyc-Tdyn` seront intégrés dans `Cyc` de sorte que les arcs de type `prevOf` initialement dans `Cyc-Tdyn` soient toujours maintenus.
- Chaque prédécesseur de `Tdyn` devient un prédécesseur de `I` par un arc de type `prevOf`.
- Chaque successeur de `Tdyn` devient un successeur de chacun des F_1, ..., F_n par des arcs de type `prevOf`.

10. Instanciation de types de niveau M1

III.3.30 inst

L'instanciation d'un type de niveau M1 est faite à chaque fois qu'un nouvel élément

est déclaré au M1 comme une instance du premier. `inst` est un méta-arc abstrait et désigne différents types de liens d'instanciations localement appliqués entre les instances (`TypeInstance`) et les types (`Type`) de M1. `inst` est représenté par `instOf` et `instModelOf`.

Lorsqu'un type de M1 est instancié, tous ses attributs doivent aussi être instanciés, et chaque instance du type en question doit se conformer à la structure de définition de ce type. La Règle 30 (Annexe IV) est à respecter.

III.3.31 instOf

Structure 1 (cf. la Figure 104, page 118): `instOf` peut unir `Object` à `ObjType`.

Structure 2 (cf. la Figure 108, page 120): `instOf` peut unir `Role` à `RoleType`.

Structure 3 (cf. la Figure 121, page 127): `instOf` peut unir `Relation` à `RelationType`.

La sémantique de ces trois structures: Les liens d'instanciations qui associent les objets/rôles/relations à leurs types d'objets/rôles/relations correspondants au niveau M1 sont représentés par des arcs de type `instOf`. La multi-classification est permise.

Structure 4 (cf. la Figure 131, page 133): `instOf` peut unir `AttrInst` à `Attribute` pour représenter les liens d'instanciations entre les attributs et leurs instances. Une instance d'attribut (`AttrInst`) est une instance directe d'un et d'un seul attribut (`Attribute`) et est attachée à ce dernier par un arc de type `instOf`.

III.3.32 instModelOf

`instModelOf` (cf. la Figure 152, page 145) représente l'ensemble des liens d'instanciation entre modèles de niveau M1. Le rapport d'instanciation entre modèles est spécifié par les Définition 19 (page 211), Définition 20 (page 213) et Définition 21 (page 213). Puisqu'une relation peut avoir plusieurs types de relations et aussi qu'une structure peut avoir plusieurs structures *mères*, entre modèles de niveau M1, un modèle (qui peut être un contexte ou une structure) peut se conformer à plusieurs modèles.

Une instance d'une structure est aussi une instance des structures plus générales que la structure en question et qu'une structure instanciée d'une structure est aussi une structure *fille* de la dernière, alors le rapport d'instanciation entre modèles est transitif. Ce rapport est aussi antiréflexif et acyclique. C'est-à-dire `instModelOf` doit remplir les Règle 5, Règle 6 et Règle 7 (Annexe IV).

11. Modèles et leurs liens

III.3.33 modelArc

`modelArc` est abstrait et représente au M1 l'ensemble de liens pouvant exister entre des modèles eux-mêmes et aussi entre des modèles et leurs éléments.

`modelArc` est subdivisé en: `sem`, `instModelOf`, `defAs`, `defIn`, `contain`, `extend`, `import`, `modelOpArc`, `modelOp`, `join`, `diff`, `intersection`, `result`, `infer`, `restrict`, `transform`, `semAs`, `viewOf`, et `inheritModel`.

III.3.34 defAs

Chaque structure dite *initiale* d'un type de relations (`RelationType`) sera reliée à ce type par un *lien de définition* de type `defAs`. Un type de relations (`RelationType`) peut être défini par (`defAs`) une ou plusieurs structures (`Structure`) initiales, et une structure peut représenter la définition d'un seul type de relations (cf. la Figure 149, page 143).

Si un type de relations n'a aucune structure initiale propre à lui, sa définition est déduite de la ou les structures initiales de son ou ses super-types. La structure de définition explicite d'un type de relations doit être plus spécifique que celles de ses super-types (cf. la Règle 17, Annexe IV).

Définition d'une structure d'un type de relations (exemple 60)

La Figure III-51 illustre le rapport de définition entre le type de relation `travailler` et `Structure-travailler`, une des structures de `travailler`. `travailler` et `Structure-travailler` ont été décrits dans l'exemple 45 (page iii-11).

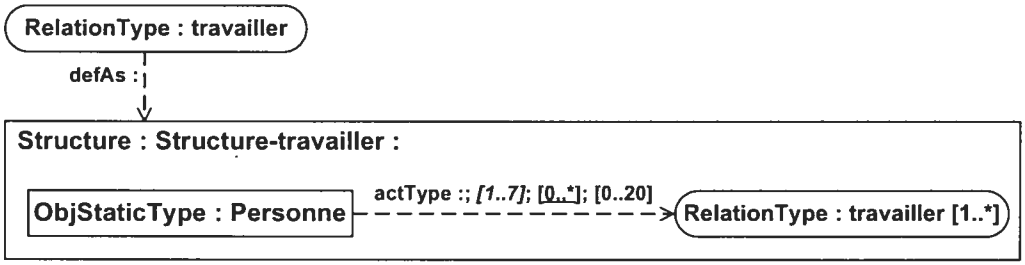


Figure III-51 : Définition d'une structure du type de relations *travailler*

III.3.35 defIn

`defIn` est défini pour unir `Element` à `Model` afin d'indiquer au M1 quel élément est défini dans quel modèle. Un élément est défini dans un et un seul modèle mais plusieurs éléments peuvent être définis dans un même modèle. Si un élément n'est pas spécifié explicitement dans quel modèle il est défini, par défaut, cet élément prendra son niveau

M1 comme le modèle dans lequel il est défini.

Si un élément est défini dans un modèle, le dernier contient le premier. `defIn` est restreint par les Règle 6, Règle 9, Règle 11 et Règle 12 (Annexe IV).

Les éléments définis dans un modèle sont généralement invisibles de l'extérieur. Au niveau M1, les arcs de types `defIn`, `extend`, `import`, `infer`, `restrict`, et `inheritModel` entre modèles sont les moyens permettant la réutilisation d'éléments entre modèles (cf. la Règle 13, Annexe IV).

III.3.36 contain

`contain` est défini pour unir `Model` à `Element` afin de représenter des liens d'appartenance entre un contenant et son contenu. Au niveau M1, un modèle peut contenir plusieurs éléments et un élément peut appartenir à plusieurs modèles.

`contain` inverse `defIn` et est relié à `defIn` par un arc de type `Minverse` (cf. la Règle 22, Annexe IV). `contain` est antiréflexif et acyclique (cf. les Règle 6 et Règle 7, Annexe IV), et aussi restreint par les Règle 10, Règle 11 et Règle 12 (Annexe IV).

III.3.37 extend

`extend` implémente l'extension de modèles au M1. Si un modèle m_1 est relié par un arc de type `extend` à un autre modèle m_2, tous les éléments dans m_2 sont considérés comme être dans m_1 (cf. la Règle 14, Annexe IV).

`extend` est acyclique (cf. la Règle 7, Annexe IV).

III.3.38 import

`import` est défini pour unir `Model` à `Model` et implémente l'importation d'éléments publiques entre modèles. La Règle 32 (Annexe IV) décrit la sémantique de `import`. Voir la Définition 23 (page iii-2) pour l'effet de la visibilité d'éléments (y compris de modèles) sur l'accès entre modèles.

III.3.39 modelOpArc

`modelOpArc` est abstrait et regroupe les méta-arcs permettant la représentation des opérations entre modèles au M1. `modelOpArc` est subdivisé en: `instModelOf`, `modelOp`, `result`, `infer`, `restrict`, `transform`, et `semAs`.

III.3.40 modelOp, join, diff, intersection, result

`modelOp` est abstrait et regroupe les opérations binaires entre modèles au M1 dont chacune retourne un et un seul modèle comme résultat (ce qui est indiqué par des arcs de

type `result`). `modelOp` est subdivisé en: `join`, `diff`, `intersection`.

`join`, `intersection` et `diff` servent à représenter respectivement au niveau M1 l'opération de jointure entre deux modèles, l'opération d'intersection entre modèles et l'opération de différence d'éléments entre modèles (cf. la section 2.3.1, page 17).

`join` et `intersection` sont symétriques (cf. la Règle 21, Annexe IV).

III.3.41 infer

Au M1, le fait qu'un modèle m_1 est relié par un arc de type `infer` à un autre modèle m_2 représente que m_2 est inféré de m_1 en appliquant nos règles d'inférence, y compris les règles de sous-typage (cf. les Règle 28 et Règle 1, Annexe IV).

III.3.42 restrict

Au M1, le fait qu'un modèle m_1 est relié par un arc de type `restrict` à un autre modèle m_2 spécifie que m_1 est plus restreint/spécifique que m_2 (cf. la Définition 16, page 208, la Définition 18, page 210, et la Définition 15, page 208). Entre les structures en particulier, `restrict` peut se comporter comme les liens de sous-typage. Une structure hérite tous les attributs et types de relations de toutes les structures auxquelles elle est reliée par des arcs de type `restrict` (cf. la Règle 28, Annexe IV). `restrict` est transitif, antiréflexif, et acyclique (cf. les Règle 5, Règle 6 et Règle 7 (Annexe IV)).

Exemple : Soient S_1 la structure du type de relations `travailler-pour` présentée dans Figure III-14 (page iii-12), et S_2 celle dans la Figure III-22 (page iii-21). Comme S_2 est plus spécifique que S_1, S_2 sera reliée à S_1 par un arc de type `restrict`.

III.3.43 transform

`transform` est défini pour unir `Model` à `Model` et représente l'ensemble des arcs exprimant le rapport de transformation entre modèles au M1.

III.3.44 semAs

`semAs` est défini pour unir `Model` à `Model` et représente l'ensemble des arcs exprimant le rapport d'équivalence entre modèles de M1. Soient m_1 et m_2 deux modèles représentant une même chose, m_1 et m_2 peuvent donc s'unir explicitement par un arc de type `semAs`.

III.3.45 viewOf

Structure 1 (cf. la Figure 154-(1), page 146): `viewOf` peut unir `Model` à `Model` pour représenter le rapport de vues différentes entre modèles.

Si des modèles sont rattachés par des arcs de type `viewOf` à un modèle, ils sont interprétés comme des *vues (aspects) différentes sur l'axe horizontal* pour le dernier. Par exemple, un modèle désignant un système physique peut avoir plusieurs modèles représentant des aspects différents du système tels que le modèle de structure, le modèle de fonctionnement, etc.

Si un ensemble de modèles, soient m_1, m_2, ..., et m_n forment une chaîne selon des arcs de type `viewOf` entre eux de telle sorte que par des arcs de type `viewOf`, m_2 soit relié à m_1, m_3 soit relié à m_2, et ainsi de suite, m_n soit relié à m_{n-1}, alors les modèles m_2, ..., et m_n sont interprétés comme des *vues différentes sur l'axe vertical* pour m_1 au cours du raffinement de m_1. Par exemple, vis-à-vis à des étapes de son raffinement, un modèle désignant un système physique peut avoir plusieurs vues telles que le modèle (à l'étape) d'analyse, le modèle (à l'étape) de conception, etc.

Structure 2 (cf. la Figure 154-(2), page 146): `viewOf` peut unir `Model` à `ObjType`.

Structure 3 (cf. la Figure 154-(3), page 146): `viewOf` peut unir `Model` à `Object`.

La sémantique de ces structures 2 et 3 :

La structure 2 (resp. structure 3) de `viewOf` permet à `viewOf` d'indiquer des modèles représentant des vues différentes pour un type d'objets (resp. un objet). Nous pouvons aussi distinguer les vues différentes sur l'axe horizontal et celles sur l'axe vertical pour le dernier.

Des modèles rattachés par des arcs de type `viewOf` à un type d'objets (resp. un objet) sont interprétés comme des *vues différentes sur l'axe horizontal* pour le dernier. Par exemple, un type de voitures (resp. une voiture) peut aussi avoir plusieurs modèles qui représentent respectivement sa structure, son fonctionnement, etc.

Si un type d'objets (respectivement un objet), soit E, et un ensemble de modèles, soient m_1, m_2, ..., et m_n, forment une chaîne selon des arcs de type `viewOf` entre eux de telle sorte que par des arcs de type `viewOf`, m_1 soit relié à E, m_2 soit relié à m_1, m_3 soit relié à m_2, ..., et m_n soit relié à m_{n-1}, alors les modèles m_1, ..., et m_n sont interprétés comme des *vues différentes sur l'axe vertical* pour E au cours du raffinement de E.

Le méta-arc `viewOf` est antiréflexif et acyclique (cf. Règle 6 et Règle 7, Annexe IV).

III.3.46 inheritModel

`inheritModel` est défini pour unir `Model` à `Model`. `inheritModel` implémente la relation d'héritage entre modèles de M1 et permet également de classifier ces modèles.

La Règle 31 (Annexe IV) présente le comportement de `inheritModel`. Voir les Définition 23 (page iii-2), Définition 24 (page iii-2), et Définition 25 (page iii-2) pour l'effet de la visibilité d'éléments (y compris de modèles) sur le comportement de l'héritage entre éléments.

12. Spécification de règles

III.3.47 cond, if, then

`cond` est abstrait et regroupe les `if` et les `then` qui permettent de structurer les règles de niveau M1. Au niveau M1, une règle (`Rule`) est attachée à sa partie de préconditions (`IfThenModel`) et à sa partie de postconditions (`IfThenModel`) respectivement par des arcs de type `if/then` (cf. la Figure 156, page 147). Similaire au cas de représentation des règles définies au niveau méta, la Règle 20 (Annexe IV) est à noter.

III.3.48 fulfil

Structure 1 (cf. la Figure 157-(1), page 148): `fulfil` peut unir `RolePlayerType` à `Rule`.

Règles contraignant un type de joueurs de rôles (exemple 61)

Une structure du type de relations `travailler` entre `Personne` est illustrée dans la Figure III-11 (page iii-11) et décrite dans l'exemple 45 (page iii-11). Cette structure et le type de relations `travailler` peuvent être reliés par des arcs de type `fulfil` à une règle. Cette règle spécifie que les personnes travaillant ensemble sont les personnes distinctes, et elle peut être codée comme suit: s'il existe deux arcs (de type `act`) qui associent respectivement deux variables soient `x` et `y` représentant des personnes (instances de `Personne`) à une relation de type `travailler`, alors il y a un arc de type `!=` entre `x` et `y` (c'est-à-dire `x` et `y` représentent deux personnes différentes).

Structure 2 (cf. la Figure 157-(2), page 148): `fulfil` peut unir `preOf` à `Rule`. Son comportement est décrit par la Règle 33 (Annexe IV).

Règles de transition entre types d'états pour un type d'objets/rôles (exemple 62)

Prenons pour l'exemple le cycle d'état civil pour le type `Personne` illustré dans la Figure III-46 (page iii-36) et décrit dans l'exemple 58 (page iii-35). L'arc de type `prevOf` qui associe `Célibataire` à `PersonneMariée` peut être relié par un arc de type `fulfil` à une règle. Cette règle spécifie qu'une personne (`Personne`) changera de l'état célibataire (`Célibataire`) à l'état marié (`PersonneMariée`) quand elle se marie avec une autre personne.

Structure 3 (cf. la Figure 157-(3), page 148): `fulfil` peut unir `playedByType` à

`Rule`. Son comportement est décrit par la Règle 34 (Annexe IV).

Règles relatives à l'occupation d'un type de rôles pour un type de joueurs de rôles (exemple 63)

Retournons à la Figure III-23 (page iii-22) décrite dans l'exemple 50 (page iii-22), l'arc de type `playedByType` *de* `Employé` *à* `Personne` *peut être relié par un arc de type* `fulfil` *à une règle. Et cette dernière spécifie qu'une personne (*`Personne`*) s'occupe d'un rôle* employé *(*`Employé`*) quand la personne a signé un contrat d'emploi (avec un ou des employeurs) correspondant à ce rôle.*

Structure 4 (cf. la Figure 157-(4), page 148): `fulfil` peut unir `transform` à `Rule`.

`fulfil` peut indiquer des règles à appliquer lors de la transformation entre modèles. Son comportement est décrit par la Règle 35 (Annexe IV).

III.3.49 assign, assignedTo, valueIn, noValueIn

`assign` est abstrait et regroupe les méta-arcs `assignedTo`, `valueIn`, et `noValueIn` qui implémentent au niveau M1 les fonctions d'affectation d'éléments concrets à des variables (cf. la Figure 158, page 148).

Un arc de type `assignedTo` d'un élément à une variable représente que cet élément est attribué à cette variable selon le cas d'application.

Un arc de type `valueIn` d'une variable à une liste spécifie que dans ce contexte, seulement les membres de cette liste sont des éléments attribuables à cette variable. `noValueIn` est la négation logique de `valueIn` et vice-versa, `valueIn` est la négation logique de `noValueIn`. Un arc de type `noValueIn` d'une variable à une liste spécifie qu'aucun membre de cette liste n'est attribuable à cette variable.

III.3.50 memberOf

`memberOf` représente l'ensemble des arcs à indiquer les éléments membres pour une liste (cf. la Figure 158, page 148).

Comme une liste récursive n'est pas autorisée, le méta-arc `memberOf` est à la fois antiréflexif (cf. la Règle 6, Annexe IV) et antisymétrique (cf. la Règle 15, Annexe IV).

III.3.51 where

`where` est défini pour unir `fulfil` à `assign` (cf. la Figure 158, page 148). D'autres structures de `where` sont déduites de cette structure de définition initiale en remplaçant le méta-arc `assign` par ses sous-types.

Soit r_1 un arc de type `fulfil` associant un élément (soit `eA`) à une règle (soit `RègleB`). Soit r_2 un `assign` (précisément, un arc conforme à un type dans le groupe

assign) contraignant des éléments attribuables à une variable (soit *variableX*) présente dans *RègleB*. Un arc de type where de r_1 à r_2 spécifie que l'élément *eA* doit remplir la règle *RègleB* où la variable *variableX* est contrainte par les éléments indiqués par r_2. À noter que deux éléments différents ne peuvent pas à la fois être attribués à une variable (la Règle 19, Annexe IV).

13. Représentation de contraintes

III.3.52 not

not implémente la négation logique, symétrique (cf. la Règle 21, Annexe IV) mais antiréflexive (cf. la Règle 6, Annexe IV), entre des types de relations (Figure 159).

Si deux types de relations sont reliés par un arc de type not, alors l'un est la négation logique de l'autre et réciproquement.

III.3.53 subset

subset représente l'ensemble des arcs pour spécifier les contraintes de sous-ensemble ou bien de dépendance d'inclusion. Un arc de type subset associe deux arcs de type relArcType. La Règle 36 (Annexe IV) définit le comportement de subset.

III.3.54 xor

Structure 1 (cf. la Figure 160-(1), page 149): xor peut unir IfThenModel à IfThenModel pour implémenter au M1 l'opérateur «ou exclusif» («xor» en anglais) logique. La Règle 37 (Annexe IV) exprime la sémantique de cette structure.

Structure 2 (cf. la Figure 160-(2), page 149): xor peut unir RelationType à RelationType pour représenter la contrainte d'exclusivité entre types de relations (Définition 11, page 181). La Règle 38 (Annexe IV) est à retenir.

Structure 3 (cf. la Figure 160-(3), page 149): xor peut unir relArcType à relArcType pour représenter la contrainte d'exclusion sur types de relations (Définition 13, page 183). La Règle 39 (Annexe IV) est à retenir.

Structure 4 (cf. la Figure 160-(4), page 149): xor peut unir actAsType à actAsType pour représenter la contrainte d'exclusivité sur types de rôles (Définition 14, page 183). La Règle 40 (Annexe IV) est à retenir.

Structure 5 (cf. la Figure 160-(5), page 149): xor peut unir playedByType à playedByType pour représenter la contrainte d'exclusivité pour un type sur des types de rôles (Définition 12, page 181). La Règle 41 (Annexe IV) est à retenir.

14. Activités d'usagers

III.3.55 create

Un arc de type `create` d'un utilisateur (`User`) à un élément (`Element`) représente que dans notre base de connaissance, cet élément est créé par cet utilisateur (cf. la Figure 161, page 150). Un utilisateur peut créer plusieurs éléments mais un élément peut être créé par un utilisateur au maximum.

15. Autres utilités

III.3.56 compare, =, ==, !=, forValues, >, <, <=, >=

`compare` est abstrait et à la racine de la hiérarchie des méta-arcs qui implémentent les opérateurs des opérations de comparaison entre deux éléments de M1. `compare` est subdivisé en : `=` (égal), `!=` (inégal, ou différent), et `forValues`. `forValues` est abstrait et regroupe: `>` (supérieur), `<` (inférieur), `<=` (inférieur ou égal) et `>=` (supérieur ou égal). `==` est un sous-type de `=`, et vise à représenter l'identité. C'est-à-dire qu'un arc de type `==` entre un élément e_1 et un élément e_2 indique que e_1 et e_2 désignent le même élément.

Toutes les comparaisons (`=`, `!=`, `>`, `<`, `<=`, `>=`) entre deux entiers (`LabelledInteger`) sont permises. Seulement la comparaison d'égalité (`=`) et celle de différence (`!=`) sont autorisées pour tous les éléments (`Element`).

Les méta-arcs `=` et `!=` sont symétriques (cf. la Règle 21, Annexe IV).

III.3.57 eqv

Un arc de type `eqv` entre deux nœuds signifie que ces derniers représentent des termes équivalents (cf. la Figure 163, page 151).

*III.3.58 math, +, -, *, /, resultVal*

`math` est abstrait et regroupe les méta-arcs `+`, `-`, `*`, `/` qui codent vis-à-vis les opérateurs des opérations mathématiques : « + » de l'addition, « - » de la soustraction, « * » de la multiplication, et « / » de la division. Une telle opération prend deux valeurs comme entrées et retourne une valeur comme le résultat, ce qui est indiqué par des arcs de type `resultVal`. Par le sous-typage, les structures de `+`, `-`, `*`, `/`, `resultVal` sont déduites des structures de `math` et `resultVal` illustrées dans la Figure 164 (page 151).

Les méta-arcs `+` et `*` sont symétriques (cf. la Règle 21, Annexe IV).

Annexe IV. Règles sémantiques

Cette annexe présente les règles sémantiques à noter dans notre méta-métamodèle (M3) et celles dans notre métamodèle (M2). Ces règles visent à restreindre des méta-éléments, surtout des méta-arcs. Elles nous aident à manifester davantage la sémantique de ces méta-éléments, et à mieux comprendre le comportement de notre M3/M2.

Règles sémantiques communes dans notre formalisme

Règle 1 : Restriction de type

Dans un modèle, un type peut être remplacé par un sous-type pour en résulter un autre modèle.

Règle 2 : Sous-typage entre méta-éléments

Un méta-nœud (méta-arc) ne peut pas être sous-type d'un méta-arc (méta-nœud).

Règle 3 : Rapport existentiel entre un nœud (arc) et son méta-nœud (méta-arc)

Un nœud (respectivement arc d'un élément source à un élément destination) peut exister si et seulement si son méta-nœud (respectivement méta-arc à relier le méta-élément source au méta-élément destination correspondants) est défini auparavant au niveau méta.

Règle 4 : Règle de cardinalité

Pour les contraintes des cardinalités minimale et maximale d'une même sorte pour un type (par exemple, sur la source ou la destination d'un méta-arc), la cardinalité correspondante au niveau instances du type en question doit être toujours supérieure ou égale à la minimale, et inférieure ou égale à la maximale.

Définition 27 : Réflexivité, antiréflexivité, transitivité, symétrie, antisymétrie, propriété acyclique d'un type relationnel binaire

Soient R un type relationnel binaire, X le domaine d'application de R, et Y celui de variation. Nous notons une instance de R entre deux éléments, soient x et y, sous la forme xRy. Donc pour tous x, y: si xRy, alors x et y sont instanciés respectivement de X et Y. Voici les règles définissant les propriétés réflexivité, antiréflexivité, transitivité, symétrie, antisymétrie, acyclique pour R:

- réflexive: si xRx pour tout x une instance de X;
- antiréflexive: il n'existe aucune instance de X soit x de telle sorte que xRx;
- transitive: pour tous x, y, z, si xRy et yRz alors xRz;
- symétrique: pour tous x, y, si xRy alors yRx;
- antisymétrique: pour tous x, y, si xRy et yRx alors x et y sont identiques;

– acyclique: il n'existe aucun cycle d'éléments, soit (x_1, ..., x_n, x_{n+1}), de telle sorte que $x_{n+1}Rx_1$, et que x_iRx_{i+1} pour tout i=1, ..., n.

Règle 5 : Méta-arc transitif

Soit R un méta-arc transitif. S'il y a un arc de type R entre deux éléments, soit de e_1 à e_2, et un autre arc de type R de e_2 à e_3, alors e_1 sera relié à e_2 aussi par un arc de type R (cf. la Définition 27, page iv-1).

Règle 6 : Méta-arc antiréflexif

Soit R un méta-arc antiréflexif. Il n'existe aucun élément qui ne relie pas à lui-même par un arc de type R (cf. la Définition 27, page iv-1).

Règle 7 : Méta-arc acyclique

Soit R un méta-arc acyclique. Il n'existe donc aucun cycle d'éléments, soit (e_1, ..., e_n, e_{n+1}), de telle sorte que e_{n+1} soit lié à e_1 par un arc de type R, et que e_i soit lié à e_{i+1} par un arc de type R pour tout i=1, ..., n (cf. la Définition 27, page iv-1).

Règle 8 : Respect de contraintes de cardinalité

Comme une instance d'un type est considérée aussi comme celle de n'importe quel super-type du type en question, alors à chaque fois un nouvel élément est déclaré, nous devons assurer que celui-ci respecte toujours les contraintes de cardinalité qui le touchent en suivant la Règle 4 (page iv-1).

Règle 9 : Règle de défini-dans relative aux types relationnels

Soient M un modèle, R un type relationnel, et S une structure initiale de R. Si R est défini dans M, alors S est considérée comme être définie dans M.

Règle 10 : Règle de contenu-dans relative aux types relationnels

Soient M un modèle, R un type relationnel, et S une structure initiale de R. Si R est contenu dans M, alors S est considérée comme être contenue dans M.

Règle 11 : Règle de défini-dans/ contenu-dans relative aux liens

Soient M un modèle, l un lien entre les éléments e_1, ..., e_n. Si l est défini ou contenu dans M, alors les éléments e_1, ..., e_n sont considérés comme être contenus dans M.

Règle 12 : Règle de défini-dans/ contenu-dans entre modèles

Soient m_1, m_2 deux modèles. Si m_1 est défini ou contenu dans m_2, alors m_1 peut accéder à tous les éléments de m_2.

Règle 13 : Accès pour réutilisation d'éléments entre modèles

Si un modèle soit m_1 peut accéder à un autre soit m_2 pour pouvoir dans m_1 réutiliser des

éléments dans m_2, nous pouvons aussi dans m_1 réutiliser des éléments qui sont définis dans d'autres modèles mais visibles pour des modèles accédant à m_2.

Règle 14 : Règle d'extension entre modèles

Soient m_1, m_2 deux modèles. Si m_1 étend m_2, alors m_1 contiendra tous les éléments de m_2.

Règle 15 : Méta-arc antisymétrique

Soit R un méta-arc antisymétrique. S'il y a un arc de type R entre deux éléments, soit de e_1 à e_2, et un autre arc de type R de e_2 à e_1, alors e_1 et e_2 seront un même élément (ce qui sera exprimé par un lien d'égalité entre e_1 et e_2) (cf. la Définition 27, page iv-1).

Règle 16 : Structure de définition d'un type relationnel

Soient R un type relationnel, et S une structure de R. S est dite une *structure de définition* de R (i) si S est la structure initiale de R, ou (ii) si S est déduite à partir d'une ou des structures initiales d'un ou des super-types de R.

Règle 17 : Rapport entre structures de définition

Chaque structure de définition d'un type relationnel doit être plus spécifique que toutes les structures de définition de tous ses super-types.

Règle 18 : Rapport entre un modèle et son métamodèle

Soient m_1, m_2 deux modèles. Si m_1 est le métamodèle de m_2, c'est-à-dire m_2 est relié à m_1 par un lien de conformité entre modèles et métamodèles, alors chaque élément défini dans m_2 a son méta-élément appartenant à m_1.

Règle 19 : Règle d'attribution d'éléments à une variable

Deux éléments différents ne peuvent pas simultanément être attribués à une variable.

Deux variables peuvent être remplacées par un même élément concret. Mais elles ne le peuvent pas si elles sont déclarées comme membres d'une liste d'éléments distincts deux à deux.

Règle 20 : Règle sur la combinaison des modèles de pré(post)conditions dans une règle

Si une règle est liée à plusieurs modèles de conditions comme préconditions (respectivement postconditions), alors sa partie de préconditions (respectivement de postconditions) est la combinaison de type «ou» logique des modèles de conditions présents dans cette partie.

Si une règle n'est liée à aucun modèle de conditions comme préconditions (c'est-à-dire qu'elle n'a pas la partie de préconditions), alors ce que la partie de postconditions représente doit être un fait ou bien être toujours vrai (cf. la Définition 8, page 38).

Règle 21 : Méta-arc symétrique

Soit R un méta-arc symétrique. S'il y a un arc de type R entre deux éléments, soit de e_1 à e_2, alors e_2 est aussi relié à e_1 par un arc de type R (cf. la Définition 27, page iv-1).

Règle 22 : Méta-arc Minverse

Soit R_1, R_2 deux méta-arcs, et soit qu'il y ait un arc de type `Minverse` de R_2 à R_1 (c'est-à-dire R_2 inverse R_1). S'il y a un arc de type R_1 entre deux éléments, soit de e_1 à e_2, alors e_2 est aussi relié à e_1 par un arc de type R_2.

Règle 23 : Arcs identiques

Soient deux arcs, r_1 reliant e_1 à e_2, et r_2 reliant e'_1 à e'_2. r_1 et r_2 sont considérés comme identiques s'ils sont d'un même type et si les deux séquences (e_1, e_2) et (e'_1, e'_2) sont identiques.

Règles contraignant des méta-éléments de M2

Règle 24 : Meta-lien actAsType en rapport avec relArcType

S'il existe un `actAsType` qui relie un type de joueurs de rôles à un `relArcType` (ou `actType`, `objType`), alors ce dernier associe un type de rôles à un type de relations.

Règle 25 : Meta-lien actAsType en rapport avec lui-même

Dans le cadre d'une structure d'un type de relations, s'il existe un `actAsType` qui relie un type de joueurs de rôles à un autre `actAsType`, alors ce dernier associe un `actAsType` (ou un `relArcType`) à un type de rôles.

Règle 26 : Meta-lien actAs en rapport avec relArcType

S'il existe un `actAs` qui relie un joueur de rôles ou un type de joueurs de rôles à un `relArcType` (ou `actType`, `objType`), alors ce dernier associe un type de rôles à un type de relations.

Règle 27 : Meta-lien actAs en rapport avec actAsType

Dans le cadre d'une structure d'un type de relations, s'il existe un `actAs` qui relie un joueur de rôles ou un type de joueurs de rôles à un `actAsType`, alors ce dernier associe un `actAsType` (ou un `relArcType`) à un type de rôles.

Règle 28 : Règle du sous-typage au M1

Soient T_1 et T_2 deux types au M1. Si T_1 hérite de T_2 par le sous-typage, T_1 hérite de T_2 toutes les propriétés (attributs, méthodes, relations, comportements, etc.).

Règle 29 : Règle de l'héritage au M1

Soient T_1 et T_2 deux types au M1. Si T_1 hérite de T_2 par l'héritage, T_1 hérite de T_2 les

attributs héritables et les types de relations héritables (cf. Définition 25, page iii-2).

Règle 30 : Instanciation au M1 en tenant compte la visibilité d'éléments

Soit x une instance de type T définie dans un modèle M. Si x est une relation (*lien*), alors T et les éléments liés à x doivent être tous accessibles de M. Si x n'est pas une relation (*occurrence*), alors T doit être accessible de M.

Règle 31 : Rapport de l'héritage entre modèles au M1

Soient m_1 et m_2 deux modèles au M1. Si m_1 est relié à m_2 par un arc de type `inheritModel`, les éléments non cachés (publiques ou protégés) dans m_2 sont réutilisables dans m_1, et en outre m_1 hérite de m_2 les attributs héritables (publiques ou protégés) et les types de relations héritables (cf. la Définition 25, page iii-2).

Règle 32 : Règle d'importation (import)

Soient m_1 et m_2 deux modèles au M1. Si m_1 est relié à m_2 par un arc de type `import`, les éléments publiques dans m_2 sont réutilisables dans m_1.

Règle 33 : Arcs de type fulfil entre des Rule et des arcs de type prevOf

Soit x un arc de type `prevOf` qui associe un type (soit T_1) à un autre type (soit T_2) et qui est défini dans un cycle d'états pour un type soit T_3. T_1 et T_2 sont donc deux sous-types dynamiques de T_3. Si x est associé à un ensemble de règles par des arcs de type `fulfil`, alors une instance de type T_3 qui est à l'état de type T_1 passera à l'état de type T_2 si elle remplit l'ensemble de règles en question.

Règle 34 : Arcs de type fulfil entre des Rule et des arcs de type playedByType

Soit x un arc de type `playedByType` qui associe un type de rôles (soit T_1) à un type de joueurs de rôles (soit T_2). Si x est associé à un ensemble de règles par des arcs de type `fulfil`, alors une instance de type T_2 qui s'occupe d'un rôle de type T_1 doit satisfaire l'ensemble de règles en question.

Règle 35 : Arcs de type fulfil entre des Rule et des arcs de type transform

Soit x un arc de type `transform` qui associe un modèle (soit m_1) à un autre modèle (soit m_2). Si x est associé à un ensemble de règles par des arcs de type `fulfil`, alors la transformation de m_1 en m_2 respecte l'ensemble de règles en question.

Règle 36 : Méta-arc subset

Soient R_1, R_2 deux types de relations. Soient T_1, T_2 deux types impliqués respectivement à R_1 et à R_2 de telle sorte que T_1 soit associé à R_1 par l'arc arc_1 un `relArcType` et que T_2 soit associé à R_2 par l'arc arc_2 un `relArcType`. Le fait que arc_2 est associé à arc_1 par un arc de type `subset` est interprété comme suit: tout élément qui participe à une relation

de type R_2 en tant qu'instance de T_2 avec un ensemble d'éléments (disons E) doit participer à une relation de type R_1 en tant qu'instances de T_1 avec l'ensemble d'éléments en question (c'est-à-dire avec E).

Règle 37: Arc de type xor entre deux modèles de conditions

Un arc de type `xor` entre deux `IfThenModel` est interprété que la combinaison de type «ou exclusif» logique des deux `IfThenModel` en question est évaluée à vraie si et seulement si ce qu'un seul de ces deux `IfThenModel` représente est vrai.

Règle 38: Arc de type xor entre deux types de relations

S'il existe un arc de type `xor` entre deux types de relations (R_1 et R_2), alors chacun des R_1 et R_2 est défini entre un même ensemble de types, $X = \{X_1, X_2, \ldots, X_n\}$, et il existe une contrainte d'exclusivité entre R_1 et R_2 (cf. la Définition 11, page 179).

Règle 39: Arc de type xor entre deux arcs de type relArcType

S'il existe un arc (arc_1) de type `xor` entre deux arcs (arc_2 et arc_3) de type `relArcType`, alors les deux arcs arc_2 et arc_3 doivent associer un même type soit T respectivement à deux types de relations, soit R_1, R_2, respectivement dans une structure (soit S_1) de R_1 et dans une autre (soit S_2) de R_2, et il existe une contrainte d'exclusion pour T sur les types de relations R_1, R_2 respectivement dans les structures S_1 et S_2 (cf. la Définition 13, page 181).

Règle 40: Arc de type xor entre deux arcs de type actAsType

S'il existe un arc (arc_1) de type `xor` entre deux arcs (arc_2 et arc_3) de type `actAsType`, alors les deux arcs arc_2 et arc_3 doivent associer un même type soit T respectivement à deux types de rôles soient TR_1 et TR_2, et il existe une contrainte d'exclusion pour T sur les types de rôles TR_1 et TR_2 dans des types de relations (cf. la Définition 14, page 181).

Règle 41: Arc de type xor entre deux arcs de type playedByType

S'il existe un arc (arc_1) de type `xor` entre deux arcs (arc_2 et arc_3) de type `playedByType`, alors les deux arcs arc_2 et arc_3 doivent associer un même type soit T respectivement à deux types de rôles soient TR_1 et TR_2, et il existe une contrainte d'exclusivité pour T sur les types de rôles TR_1 et TR_2 (cf. la Définition 12, page 179).

Annexe V. Stockage de modèles

Stockage d'éléments répartis à tous les niveaux de modélisation (exemple 64)

idTbar	TbarLabel	Srce	metaArc	dest
1	MMetaNode	Null	88(Mmeta)	1(MMetaNode)
2		1(MMetaNode)	125(MdefIn)	23(NotreM3)
3	MMetaArc	Null	88(Mmeta)	1(MMetaNode)
...	...	...	...	...
612	NotreM2	Null	88(Mmeta)	12(MMetaModel)
613		612(NotreM2)	105(Msem)	42(NotreM3)
...	...	...	...	...
643	ObjStaticType	Null	88(Mmeta)	640(ObjType)
		643(ObjStaticType)	76(MsubType)	640(ObjType)
...	...	...	...	...
794	Structure-instOf	Null	88(Mmeta)	11(Mstructure)
795	instOf	Null	88(Mmeta)	3(MMetaArc)
796		795(instOf)	125(MdefIn)	612(NotreM2)
797		795(instOf)	140(MdefAs)	794(Structure-instOf)
798		690(Object)	85(Msrce)	795(instOf)
799		640(ObjType)	86(Mdest)	795(instOf)
800		798(null)	125(MdefIn)	794(Structure-instOf)
801		799(null)	125(MdefIn)	794(Structure-instOf)
...	...	...	...	...
1897	ModèleM1	Null	744(meta)	634(Model)
1898		1897(ModèleM1)	758(sem)	612(NotreM2)
1899	Nom	Null	744(meta)	655(Attribute)
1900		1899(Nom)	1176(defIn)	1897(ModèleM1)
1901	Personne	Null	744(meta)	643(ObjStaticType)
1902		1901(Personne)	1176(defIn)	1897(ModèleM1)
1903		1899(Nom)	995(chrcType)	1901(Personne)
...	...	...	...	...
2018	Marie	Null	744(meta)	690(Object)
2019		2018(Marie)	1176(defIn)	1897(ModèleM1)
2020	L1	2018(Marie)	795(insOf)	1901(Personne)
2021	Nom-Marie	Null	744(meta)	685(AttrInst)
2022		2021(Nom-Marie)	1176(defIn)	1897(ModèleM1)
2023		2021(Nom-Marie)	795(insOf)	1899(Nom)
2024		2021(Nom-Marie)	1134(chrc)	2018(Marie)
...	...	...	...	...

Figure V-1 : Table Tbar pour le codage d'éléments (cf. la Figure V-3)

idValue	idTbar	lang	Value
...	...	...	...
830	1901(Personne)	fr	Personne
831	1901(Personne)	en	Person
832	2021(Nom-Marie)	u	Levesque
...	...	...	...

Figure V-2 : Table Value pour le stockage de valeurs d'éléments (cf. la Figure V-3)

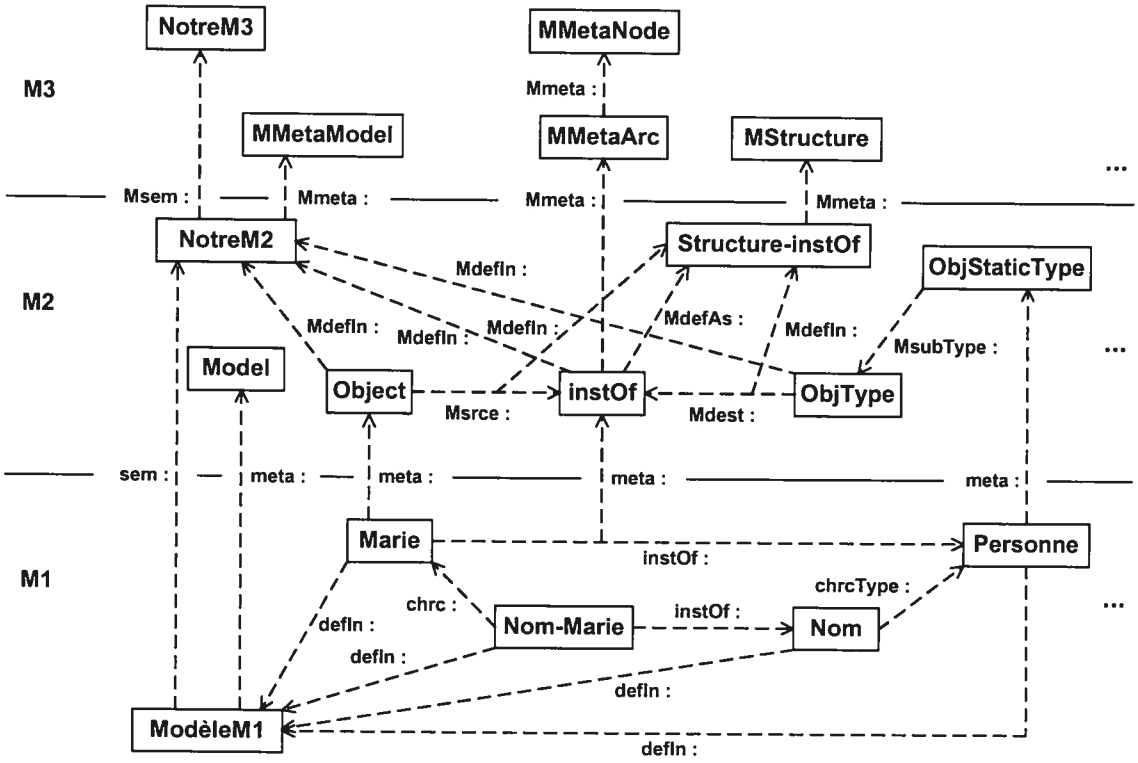


Figure V-3 : Illustration graphique d'une partie de la Figure V-1 et de la Figure V-2