Approche dirigée par les modèles pour l'implantation de bases de données massives sur des SGBD NoSQL

Amal Ait Brahim

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 1 Capitole

Présentée et soutenue par

Amal AIT BRAHIM

Le 31 octobre 2018

Approche dirigée par les modèles pour l'implantation de bases de données massives sur des SGBD NoSQL

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Gilles ZURFLUH

Jury

Mme Elisabeth METAIS, Rapporteur
M. Omar BOUSSAID, Rapporteur
M. Jean-Michel BRUEL, Examinateur
Mme Fatma ABDELHEDI, Examinateur
M. Gilles ZURFLUH, Directeur de thèse
M. Olivier TESTE, Président

Résumé

La transformation digitale des entreprises et plus largement celle de la société, entraine une évolution des bases de données (BD) relationnelles vers les BD massives. Dans les systèmes informatiques décisionnels actuels, les décideurs doivent pouvoir constituer des bases de données, les réorganiser puis en extraire l'information pertinente pour la prise de décision. Ces systèmes sont donc naturellement impactés par cette évolution où les données sont généralement stockées sur des systèmes NoSQL capables de gérer le volume, la variété et la vélocité.

Nos travaux s'inscrivent dans cette mutation ; ils concernent plus particulièrement les mécanismes d'implantation d'une BD massive sur un SGBD NoSQL. Le point de départ du processus d'implantation est constitué d'un modèle contenant la description conceptuelle des données et des contraintes d'intégrité associées.

Peu de travaux ont apporté des solutions automatiques complètes portant à la fois sur les structures de données et les contraintes d'intégrité. L'objectif de cette thèse est de proposer une démarche générale qui guide et facilite la tâche de transformation d'un modèle conceptuel en un modèle d'implantation NoSQL. Pour ceci, nous utilisons l'architecture MDA qui est une norme du consortium OMG pour le développement dirigé par les modèles.

A partir d'un modèle conceptuel exprimé à l'aide du formalisme UML, notre démarche MDA applique deux processus unifiés pour générer un modèle d'implantation sur une plateforme NoSQL choisie par l'utilisateur :

- Le processus de transformation d'un DCL,
- Le processus de transformation des contraintes associées,

Nos deux processus s'appuient sur :

- Trois niveaux de modélisation : conceptuel, logique et physique, où le modèle logique correspond à une représentation générique compatible avec les quatre types de SGBD NoSQL : colonnes, documents, graphes et clé-valeur,
- Des métamodèles permettant de vérifier la validité des modèles à chaque niveau,
- Des normes de l'OMG pour formaliser l'entrée du processus et l'ensemble des règles de transformation.

Afin de vérifier la faisabilité de notre solution, nous avons développé un prototype composé de deux modules. Le premier applique un ensemble de règles de transformation sur un modèle conceptuel et restitue un modèle NoSQL associé à un ensemble de directives d'assistance. Le second module complète le premier en intégrant les contraintes qui ne sont pas prises en compte dans le modèle physique généré. Nous avons montré également la pertinence de nos propositions grâce à une évaluation réalisée par des ingénieurs d'une société spécialisée dans le décisionnel.

Remerciements

Je remercie très sincèrement :

Monsieur Gilles ZURFLUH, Professeur et Directeur de la faculté d'informatique à l'Université Toulouse Capitole, pour sa totale disponibilité à mon égard, pour son aide, ses encouragements, ses critiques et ses précieux conseils qu'il m'a donnés et sans lesquels ce travail n'aurait pas été possible. Je tiens également à lui exprimer toute ma reconnaissance pour la confiance qu'il m'a toujours témoignée, pour m'avoir laissé une grande liberté d'action et pour m'avoir offert d'excellentes conditions pour mener à bien mes travaux de recherche.

Monsieur Olivier TESTE, Professeur à l'Université Toulouse 2 Jean Jaurès et responsable de l'équipe Systèmes d'Information Généralisées (SIG), pour m'avoir si bien accueillie au sein de son équipe afin que je puisse mener à bien cette thèse, pour avoir accepté de juger mon travail et pour l'honneur qu'il me fait en participant au jury.

Madame Elisabeth METAIS, Professeur au Conservatoire National des Arts et Métiers (CNAM) de Paris, pour avoir accepté d'être rapporteur de ce mémoire. Je tiens à lui exprimer ma plus vive reconnaissance pour la lecture approfondie de ce mémoire qui a permis d'en améliorer la qualité, pour ses remarques pertinentes et pour sa participation au jury.

Monsieur Omar BOUSSAID, Professeur à l'Université Lyon 2, pour avoir accepté d'être rapporteur de ce mémoire, pour ses critiques pertinentes, ses remarques constructives qui m'ont permis d'améliorer grandement la qualité de ce mémoire, ainsi que pour sa participation en tant que membre de mon jury. Je tiens tout particulièrement à le remercie pour les discussions que nous avons eues à Tanger et ses nombreux conseils.

Madame Fatma ABDELHEDI, Directrice du laboratoire CBI^2 au sein de la société TRIMANE, qui a participé à mes travaux et qui m'a offert la possibilité de valider mes propositions à partir d'applications réelles ; je la remercie également d'avoir participé à ce jury.

Monsieur Jean-Michel BRUEL, Professeur à l'Université Toulouse 2 Jean Jaurès, pour l'intérêt qu'il a porté à mes travaux en examinant ce mémoire et pour avoir accepté d'être membre de mon jury.

Madame Faten ATIGUI, Maître de conférences au Conservatoire National des Arts et Métiers (CNAM) de Paris, pour l'aide qu'elle m'a fournie lors de mon apprentissage de l'architecture MDA ainsi que pour sa participation à mes travaux de recherche.

Les gestionnaires de la Faculté d'informatique Michèle CUESTA et Florence THERY pour la disponibilité, l'aide généreuse et la gentillesse dont elles ont toujours fait preuve à mon égard.

Les collègues enseignants avec qui j'ai collaboré pour dispenser des cours et avec qui j'ai pu échanger sur la pédagogie. Plus particulièrement, je tiens à remercier Madame Geneviève PUJOLLE et Monsieur Khalid TAZI.

Les thésards, Lee THAN pour sa bonne humeur et sa sympathie et Rabah TIGHILT FERHAT pour sa totale disponibilité à mon égard lors de mes répétitions, pour ses remarques, ses encouragements et pour tous les échanges constructifs sur nos recherches. J'ai eu le plaisir de les connaître et de partager avec eux de si bons moments.

Pour finir, je souhaiterais remercier toute ma famille qui n'a eu de cesse de me soutenir et de croire en moi tout au long de mes études ; notamment, mes chers parents, mes sœurs Hanane et Sara et mes frères Faisal et Aimad, à qui je dédie cette thèse. Je leur suis très reconnaissante de m'avoir appris à être ambitieuse et à ne jamais baisser les bras. Mes remerciements ne pourront jamais égaler leur assistance et leur amour qui m'a apporté du soutien au moment où j'avais besoin d'aide. C'est grâce à eux et pour eux que je suis qui je suis aujourd'hui.

Je tiens particulièrement à exprimer ma profonde gratitude à ma mère Houria, la maman parfaite sur tous les plans qui a consacré toute sa vie pour notre bonheur, et à mes sœurs Hanane et Sara pour l'accompagnement de tous les instants. Merci pour les sacrifices qu'elles ont dû faire pendant mes longues années d'études et d'absence et pour le soutien indéfectible dont elles ont fait preuve lors des moments difficiles que j'ai passés pour préparer ma thèse. Qu'elles sachent que vivre loin d'elles était l'épreuve la plus difficile de ma vie.

Sommaire

Liste des Figures

Liste des Tableaux

Préambule

Nos travaux s'inscrivent dans le domaine de l'informatique décisionnelle. Selon la chaîne décisionnelle présentée dans la figure P.1, les données sont d'abord extraites de sources (BD de production, données publiques, documents du Web, …), elles sont ensuite prétraitées puis réorganisées pour être manipulées par des décideurs humains via des logiciels d'analyse. Nos travaux se situent en amont de ce processus décisionnel, au niveau des sources de données.

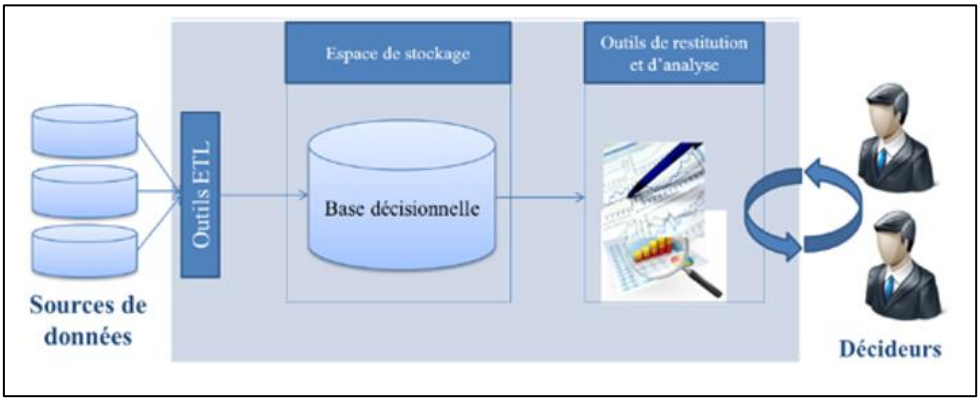


Figure P.1 : La chaîne décisionnelle

L'informatique décisionnelle a connu ces dernières années des évolutions notoires liées notamment à l'apparition des données massives (Big Data). Avant l'avènement des Big Data, les sources de données étaient principalement constituées de bases de données relationnelles, de fichiers informatiques et de documents formatés en HTML ou XML. A présent, les systèmes de décision intègrent des bases de données massives dans leurs sources. Ce nouveau type de sources sert à alimenter une base décisionnelle ; mais comme nous le verrons dans l'application Big Data présentée au chapitre 1, les utilisateurs peuvent aussi réaliser des analyses (souvent limitées à des calculs simples ou du « reporting ») directement sur la source. C'est le contexte de notre étude comme le montre la figure P.2.

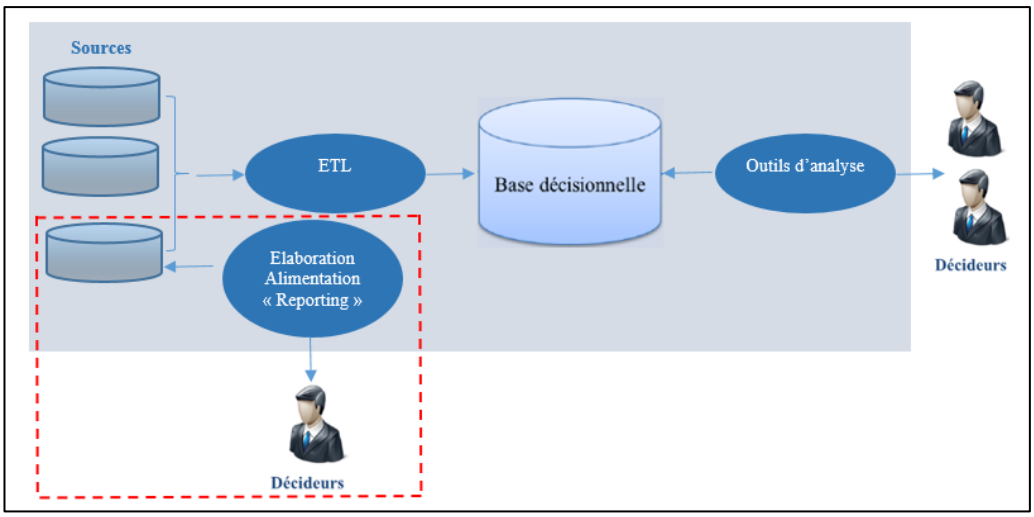


Figure P.2 : La chaîne décisionnelle étendue

Des utilisateurs (généralement des décideurs) peuvent-ils eux-mêmes créer, implanter, alimenter et analyser une base de données de type Big Data ? Pour répondre à cette question, nous avons été confrontés à deux verrous majeurs :

- Le processus d'implantation des bases de données massives
- L'autonomie des utilisateurs.

1. Les sources de données massives

Les entreprises développent des applications manipulant des bases de données massives et l'on constate que les systèmes décisionnels intègrent de plus en plus ces nouvelles sources de données.

C'est le cas dans l'application médicale qui a motivé nos travaux et que nous présentons dans le chapitre 1. Dans le cadre de programmes de suivi de pathologies, des médecins mettent en place des bases de données de type Big Data qu'ils seront amenés à alimenter, à interroger et à analyser quotidiennement pendant plusieurs années.

D'autre part, l'émergence des données massives, notamment dans le domaine des réseaux sociaux, a conduit les géants du numérique (« GAFA ») à développer des systèmes de gestion de données permettant le stockage des Big Data et apportant souplesse et performances : les systèmes NoSQL. Ainsi, le mécanisme d'implantation d'une base de données massive dans un SGBD NoSQL constitue le premier verrou auquel nous nous sommes confrontés.

2. L'autonomie des utilisateurs

Depuis l'avènement de l'informatique dans les entreprises (années 60) puis dans le grand public (années 70), on constate une tendance constante de l'industrie du logiciel : celle de permettre à des utilisateurs non informaticiens d'accéder à des procédures nécessitant des connaissances et des savoir-faire complexes. Ainsi, les tâches qui exigeaient auparavant une grande technicité et qui étaient confiées à des informaticiens, sont de plus en plus réalisées par des systèmes numériques fermés ; ceux-ci sont alors mis en œuvre par les utilisateurs eux-mêmes, détenteurs de connaissances métier mais étrangers aux techniques informatiques. Le pionnier en la matière est le langage SQL apparu dans les années 70 et qui permet à des non informaticiens d'écrire des requêtes relativement complexes sur une base de données. Ce langage de haut niveau permet de s'affranchir des contraintes techniques de bas niveau. On peut également citer les travaux récents en matière de traitement du langage naturel qui attestent du souci de rapprocher les systèmes numériques des utilisateurs finaux ; ceux-ci ne maîtrisant pas les codes informatiques [Silberztein et al., 2018].

Les systèmes décisionnels n'échappent pas à cette tendance ; un des objectifs de la recherche est donc de permettre aux utilisateurs finaux (analystes, décideurs, experts en mégadonnées notamment) de s'approprier certaines technologies.

Parmi celles-ci, on peut citer l'élaboration de sources Big Data, l'extraction et le nettoyage des données, la construction d'un cube, l'analyse et la fouille des données basée sur des algorithmes particulièrement complexes.

Dans l'application médicale que nous présentons dans le chapitre 1, les médecins, détenteurs de compétences métier mais non spécialistes en informatique, pourraient être capables de créer des bases de données et d'assurer leur maintenance en toute autonomie. Bien entendu, le recours à un informaticien n'est pas exclu ; il sera possible chaque fois que la complexité des modèles et des opérations le nécessitera.

Il s'agit du second verrou auquel nos travaux tentent de répondre : permettre à des utilisateurs non informaticiens d'implanter une base de données massives.

Notre objectif est donc de permettre à des utilisateurs d'élaborer une base de données massive sans recourir à un informaticien.

3. Périmètre et validation de nos travaux

Nos travaux ont donc pour objet de permettre à des utilisateurs de créer une BD massive sans recourir à l'intervention d'informaticiens. Mais certains aspects se situent hors de notre champ d'étude. Il en est ainsi de trois problématiques qui font l'objet de recherches par ailleurs :

- Le point de départ de l'élaboration d'une BD massive est constitué par un modèle conceptuel des données (DCL d'UML) ; or, nous n'avons pas étudié le processus d'élaboration de ce modèle qui relève de l'ingénierie des exigences [Anton, 1996].

- La solution que nous proposons dans cette thèse génère plusieurs modèles physiques pour certains SGBD NoSQL ; le choix du meilleur modèle physique selon des critères liés aux performances des traitements a été étudié par ailleurs, notamment dans les travaux de [Gomez et al., 2018].

- Une BD massive peut être distribuée sur plusieurs sites, notamment pour répondre à des problématiques de sécurité dans le cloud, chaque site étant doté d'un SGBD NoSQL spécifique [Li et al., 2017]. Cet aspect n'est pas étudié dans nos travaux.

Au sein de notre laboratoire, nous avons développé un système informatique pour expérimenter notre processus de création d'une BD massive. Afin de valider nos travaux, ce logiciel a été utilisé dans un cadre professionnel par des ingénieurs de la Société Trimane située à Saint Germain en Laye.

Chapitre 1 : Contexte et Problématique

Ces dernières années ont vu l'explosion des données générées et accumulées par les dispositifs informatiques de plus en plus nombreux et diversifiés. Les BD constituées sont désignées par l'expression « Big Data » et sont caractérisées par la règle dite des « 3V ». Celle-ci est due au volume des données qui peut dépasser plusieurs téraoctets et à la variété de ces données qui sont qualifiées de complexes. De plus, ces données sont souvent saisies à très haute fréquence et doivent donc être filtrées et agrégées en temps réel pour éviter une saturation inutile de l'espace de stockage.

Les techniques d'implantation classiques, basées principalement sur le paradigme relationnel, connaissent des limites pour gérer les BD massives [Angadi et al., 2013]. Ainsi, de nouveaux systèmes de stockage et de manipulation des données ont été développés. Regroupées sous le terme NoSQL, ces systèmes sont bien adaptés pour gérer de gros volumes de données dont les schémas sont flexibles. Ils apportent aussi de grandes capacités de passage à l'échelle et de bonnes performances en temps de réponse [Angadi et al, 2013].

Cependant, en raison de la complexité des schémas des BD à implanter et de la spécificité des modèles physiques actuels associés aux SGBD NoSQL, les utilisateurs d'applications Big Data sont confrontés à la difficulté d'implanter une BD NoSQL.

L'objectif de cette thèse est de répondre à cette problématique, en proposant une solution qui assiste un utilisateur pour implanter une BD massive sur un SGBD NoSQL. Mais cet objectif est double puisqu'il doit aussi permettre à un décideur d'implanter lui-même une BD massive présentant une complexité modérée.

Plan du chapitre. Ce chapitre présente le contexte de notre thèse ainsi que la problématique traitée. Les sections 1.1 et 1.2 définissent respectivement ce que sont les Big Data et le NoSQL. La section 1.3 justifie la nécessité de la modélisation conceptuelle des BD massives. La section 1.4 présente les principes de l'architecture dirigée par les modèles (MDA) que nous avons utilisée pour formaliser notre solution. La section 1.5 présente notre problématique et l'illustre grâce à une étude de cas. La section 1.6 introduit notre contribution et finalement la section 1.7 donne le plan de ce mémoire.

1.1 Données massives : « Big Data »

Le Big Data est un type de BD ayant des caractéristiques spécifiques. Nous présentons dans cette section les trois règles qui ont été largement utilisées pour définir le Big Data. Il s'agit des règles « 3V », « 4V » et « 5V ».

Règle « 3V » : Une description du Big Data a été établie dans un rapport de recherche en 2001 par l'analyste Douglas Laney du groupe Gartner [Douglas, 2001]. Elle définit les enjeux inhérents à la croissance des données comme étant tridimensionnels. Il s'agit du *Volume*, de la *Variété* et de la *Vélocité*. En 2012, Gartner[1] a donné une définition plus détaillée des Big Data comme suit :

> **Définition 1.** « *Les Big Data sont des données volumineuses, très variées, générées et traitées à grande vitesse. Ces données exigent des formes efficaces et innovantes de traitement de l'information pour permettre une meilleure prise de décision* » [Douglas, 2001].

Règle « 4V » : Le cabinet d'analyse IDC[2] (International Data Corporation) a délimité, dans un rapport de 2011 [Gantz and Reinsel, 2011], quatre dimensions pour caractériser le Big Data, à savoir : le *Volume*, la *Variété*, la *Vélocité* et la *Valeur*.

> **Définition 2.** « *Les Technologies Big Data décrivent une nouvelle génération de technologies et d'architectures conçues pour extraire économiquement de la valeur à partir de grands volumes de données très variées, en permettant leur capture et leur analyse à grande vitesse* » [Gantz and Reinsel, 2011].

Règle « 5V » : Deux nouveaux V sont apparus en 2013 dans un article de recherche [Demchenko et al., 2013] où le groupe SNE[3] (System and Network Engineering) propose une définition plus large pour le Big Data au travers de la règle des 5V.

> **Définition 3.** « *Les Technologies Big Data visent à traiter des données de grand volume, grande vélocité et grande variété pour extraire de la valeur, assurer une forte véracité des données originales et obtenir des informations qui exigent des formes novatrices de traitement des données, afin d'améliorer la prise de décision et le contrôle des processus* » [Demchenko et al., 2013].

Les définitions associées aux dimensions utilisées dans les règles ci-dessus sont les suivantes :

- Volume : représente la taille de l'ensemble des données à traiter,

[1] https://www.gartner.com/en
[2] https://www.idc.com/about
[3] https://ivi.fnwi.uva.nl/sne/

- Variété : fait référence à la diversité des sources, des types et des formats de données,
- Vélocité : correspond à la vitesse à laquelle les données sont collectées et traitées,
- Valeur : équivaut au profit que l'on peut tirer de l'usage des Big Data,
- Véracité : recouvre la qualité et la fiabilité des données ; c'est la dimension qualitative des Big Data.

Actuellement 90% des SGBD sont relationnels. Mais face aux caractéristiques du Big Data, les systèmes relationnels rencontrent des limites. Les principaux problèmes de ces systèmes sont :

- La mise à l'échelle horizontale : une BD relationnelle a été principalement conçue pour des configurations à serveur unique [Kumar et al., 2015] ; mettre à l'échelle cette base consiste à la distribuer sur plusieurs serveurs. Ceci pose des contraintes financières (le coût des serveurs) et techniques (le nombre de serveurs est généralement limité à 10). De plus, gérer des tables sur différents serveurs reste une tâche complexe,

- La définition d'un modèle lors de la création de la BD et avant la saisie des données : dans un contexte Big Data, l'utilisateur doit être capable d'intégrer facilement de nouvelles données. Les SGBD relationnels n'offrent pas cette souplesse ; le modèle relationnel est difficile à modifier de façon incrémentale sans affecter les performances ou mettre la BD hors ligne.

Ainsi, les technologies de stockage ont dû évoluer pour introduire de nouveaux SGBD qui sont capables de gérer des données volumineuses, variées et qui peuvent évoluer très rapidement. Il s'agit des SGBD NoSQL [Han et al, 2011] présentés dans la section suivante.

1.2 NoSQL

Le terme NoSQL désigne un type de systèmes de gestion de base de données qui va au-delà des systèmes relationnels associés au langage SQL en acceptant des structures de données plus complexes. Selon leurs modèles physiques, les BD gérées par ces systèmes se répartissent en quatre catégories : colonnes, documents, graphes et clé-valeur [Angadi et al., 2013]. Chacune d'elles offrant des fonctionnalités spécifiques. Par exemple, dans une BD orientée-documents comme MongoDB[4], les données sont stockées dans des tables dont les lignes peuvent être imbriquées. Cette organisation des données est couplée à des opérateurs qui permettent d'accéder aux données imbriquées [Kumar et al., 2015].

[4] https://www.mongodb.com/

Le choix de la catégorie du SGBD la plus adaptée à une application donnée, est lié à la nature des traitements (les requêtes) appliqués sur les données. Mais ce choix n'est pas exclusif puisque, dans chaque catégorie, les SGBD peuvent assurer tous les types de traitements, au prix parfois d'une certaine lourdeur ou d'une programmation plus importante.

Dans ce qui suit, nous présentons les modèles de données adoptés par chaque catégorie de SGBD NoSQL.

1.2.1 Modèle orienté-colonnes

Le modèle orienté-colonnes est un modèle structuré où les données sont organisées en familles de colonnes, ce qui équivaut au concept de table dans le modèle relationnel. Les lignes possèdent un identifiant appelé clé de ligne et sont composées d'un ensemble de valeurs ; chacune est associée à une colonne. Ainsi, la recherche d'une valeur revient à parcourir la séquence : *clé de ligne -> famille de colonnes -> colonne*.

Bien que le modèle orienté-colonnes se rapproche du modèle relationnel, l'organisation des données dans les deux modèles est différente [Abadi et al., 2008]. En opposition à ce que l'on trouve dans une BD relationnel où les colonnes sont statiques et présentes dans chaque ligne, dans une BD orientée-colonnes les colonnes sont dynamiques et apparaissent uniquement dans les lignes concernées. Autrement dit, chaque ligne a un nombre différent de colonnes (cf. figure 1.1) et on peut lui ajouter à tout moment de nouvelles colonnes ; on gagne ainsi en extensibilité au niveau du modèle de données.

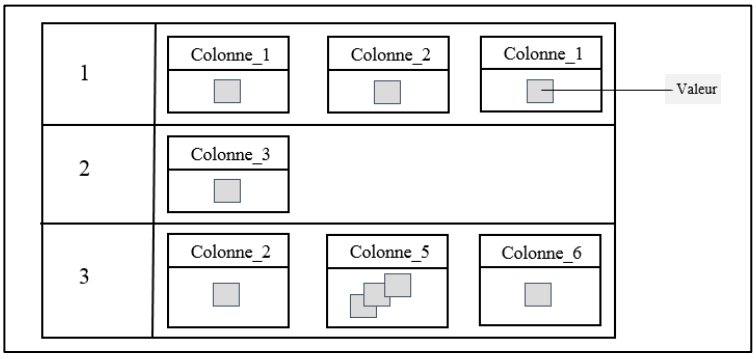


Figure 1.1 - Organisation d'une famille de colonnes dans une BD orientée-colonnes

De plus, le modèle orienté colonnes a pour avantage d'améliorer l'efficacité du stockage et d'éviter de consommer de l'espace par rapport au modèle relationnel. En effet, du fait de leur conception par allocation de blocs, dans un SGBD relationnel, une colonne vide consommera quand même de l'espace. Dans un SGBD orienté-colonnes, le coût de stockage d'une colonne vide est 0.

Cassandra[5], HBase[6] et Accumulo sont des exemples de SGBD où les données sont stockées suivant un model orienté-colonnes.

1.2.2 Modèle orienté-documents

Les données dans un modèle orienté-documents sont organisées en collections de documents. Un document est identifié par une clé à laquelle correspond un agrégat de couples clé-valeur qui peuvent être hiérarchisés (cf. figure 1.2). Cela veut dire que la valeur peut-elle même contenir une ou plusieurs paires clé-valeur.

Au sein de la même collection, les documents peuvent être de structures différentes. Autrement dit, les couples utilisés pour définir les documents d'une collection ne sont pas forcément les mêmes. De plus, comme les autres modèles NoSQL, le modèle orienté-documents est flexible ; on peut ajouter des couples à chaque insertion d'un nouveau document.

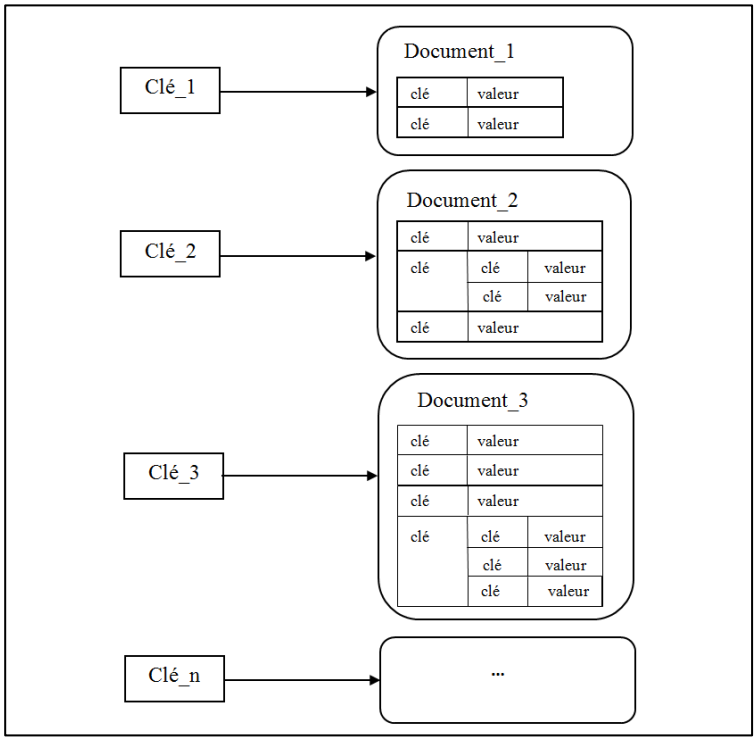


Figure 1.2 - Organisation d'une collection dans une BD orientée-documents

Les SGBD orientés-documents, comme MongoDB, couchDB [7] et couchDB server, apportent des fonctionnalités avancées pour la manipulation des

[5] Datastax. (2012). Apache Cassandra™ Documentation. From http://www.odbms.org/wp-content/uploads/2013/11/cassandra10.pdf.
[6] Apache HBase Team. (2017). Apache HBase ™ Reference Guide, version 3(0). From https://hbase.apache.org/book.html

[7] Apache CouchDB 2.1 Documentation". [Online]. Available: http://docs.couchdb.org/en/2.1.1/.

documents. Avec le principe d'imbrication du modèle de données qu'ils adoptent (le modèle orienté-documents), nous pouvons modéliser les données de façon à ce qu'elles supportent des fonctionnalités plus avancés d'interrogation, telles que la capacité de manipuler le contenu du document (recherche en plein texte) [Arora and Aggarwal, 2013].

1.2.3 Modèle orienté-graphes

Ce modèle organise les données sous forme de nœuds et de relations. Les nœuds et les arcs (c'est-à-dire les relations) peuvent porter un ensemble de propriétés exprimées sous la forme de paires clé-valeur (cf. figure 1.3).

Ce modèle est utile pour stocker et interroger des données complexes fortement liées ; c'est le cas par exemple des données issues des réseaux sociaux.

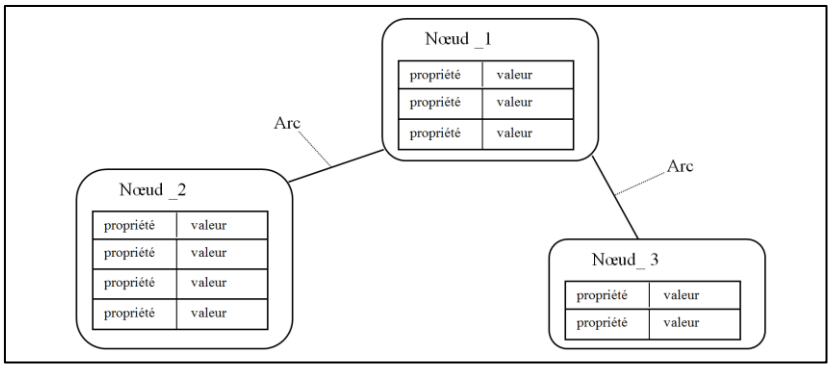


Figure 1.3 - Organisation des données dans une BD orientée-graphes

Les SGBD orientés-graphes les plus courants sont Neo4j [8], OrientDB [9] et FlockDB[10].

1.2.4 Modèle orienté clé-valeur

Il s'agit du modèle NoSQL le plus basique et qui a été le précurseur dans les SGBD NoSQL. Il organise les données sous forme de paires clé-valeur (cf. figure 1.4), où la clé est le point d'entrée unique qui permet d'accéder à la donnée. Les modèles de données orientés colonnes, documents et graphes présentés précédemment sont des évolutions du modèle clé-valeur.

Du fait de leur modèle d'accès simplifié qui utilise exclusivement la clé, les SGBD qui adoptent le modèle clé-valeur, comme Redis[11] et Riak[12], permettent d'atteindre des performances élevées en termes de temps d'accès aux données. Cependant, ils n'offrent que des fonctionnalités simplifiées en termes

[8] https://neo4j.com/
[9] https://orientdb.com/
[10] https://db-engines.com/en/system/FlockDB
[11] https://redis.io/
[12] http://docs.basho.com/

d'expression des requêtes [Angadi et al., 2013]. Ainsi, ces SGBD sont principalement utilisés dans des contextes où les besoins en termes de requêtes sont très réduits.

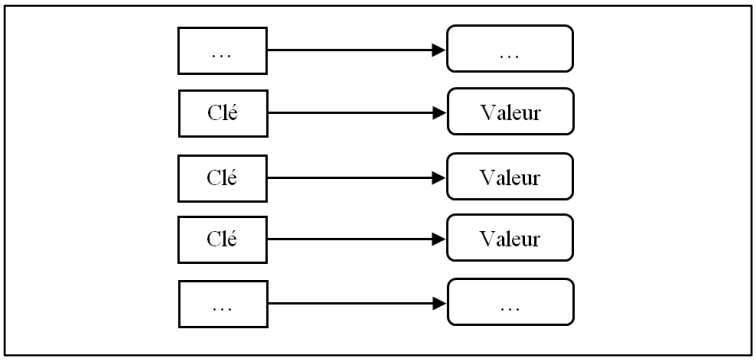


Figure 1.4 - Organisation des données dans une BD clé-valeur

1.3 Modélisation conceptuelle des Big Data

L'absence de la définition d'un schéma (schema less) lors de la création d'une BD, est une caractéristique que l'on trouve dans la plupart des SGBD NoSQL. Autrement dit, dans une table, le nom et le type des attributs de chaque ligne sont spécifiés au moment de la saisie de la ligne. Le mécanisme de schema less est en opposition à ce que l'on trouve dans les SGBD relationnels où le schéma des tables est défini à la création de cette table. Cette propriété offre une souplesse indéniable en facilitant l'évolution du schéma au fur et à mesure de l'alimentation des tables ; par exemple, elle permet l'ajout de nouveaux descripteurs pour une ligne existante ou bien la modification de la structure d'une ligne sans modifier les autres lignes préalablement stockées. Mais cette propriété concerne exclusivement le niveau physique (l'implantation) d'une BD [Herrero et al., 2016].

Or, le schéma conceptuel d'une BD (exempt de toute technique d'accès et de stockage) est un élément de connaissance sémantique essentiel pour une gestion des données efficace [Abelló, 2015], [Herrero et al., 2016], [Daniel et al., 2016]. Il fournit un haut niveau d'abstraction et vise à faire abstraction de caractéristiques techniques et il est utile, voire indispensable, pour :

- effectuer un contrôle lors de la saisie des données,
- appliquer des traitements, par exemple exprimer des requêtes qui sont des traitements particuliers permettant l'extraction de données et leurs mises à jour.

Notons que dans certains cas, la structure de la BD peut ne pas être totalement connue a priori. En effet, dans un contexte Big Data, nous distinguons les deux types de BD suivants : BD massive et réservoir de données (« Data Lake »). Une BD massive représente une extension d'une BD de production qui vérifie les 3V ;

Un Data Lake est une BD massive dont les traitements sont différés (souvent des traitements décisionnels). Les données ne sont pas structurées en fonction des traitements puisque ceux-ci ne sont pas connus au moment de la constitution de la BD. Il n'y a donc pas un schéma des données complet lors de la création du Data Lake ; les données brutes seront nettoyées, agrégées et structurées au moment de leur analyse, en fonction du besoin des utilisateurs. Par exemple, la surveillance de sites Web nécessite l'enregistrement de flux de données temporels dans des « logs » pendant l'exploitation des sites ; la survenue d'un incident entrainera l'analyse de l'historique de ces logs pour tenter d'expliquer l'événement ; la nature des traitements d'analyse sera fonction du type d'incident survenu.

Nos travaux s'intègrent dans le premier cas (BD massive). En effet dans le type d'applications qui a motivé nos travaux (cf. section 1.5), la connaissance de la structure de la BD est nécessaire.

Une BD massive est associée au critère de la variété des données. Elle pourra contenir des données de type standard (nombres, dates, chaînes de caractères) mais aussi des données multimédia (textes, images, graphiques, documents). Le modèle des classes du langage UMLsoa [13] est un formalisme conceptuel permettant de décrire ces données complexes sous forme d'objets ; il est aussi largement reconnu comme un modèle sémantique performant. UML est donc considéré comme le langage le plus adapté à la description d'une BD massive [Abelló, 2015].

Comme le montre la figure 1.5, un schéma conceptuel est composé de deux éléments : un diagramme décrivant la structure des données et le complément de contraintes non présentes dans le diagramme. Ces contraintes sont généralement formalisées avec le langage OCL qui est un langage d'expression de contraintes adapté au diagramme de classes d'UML.

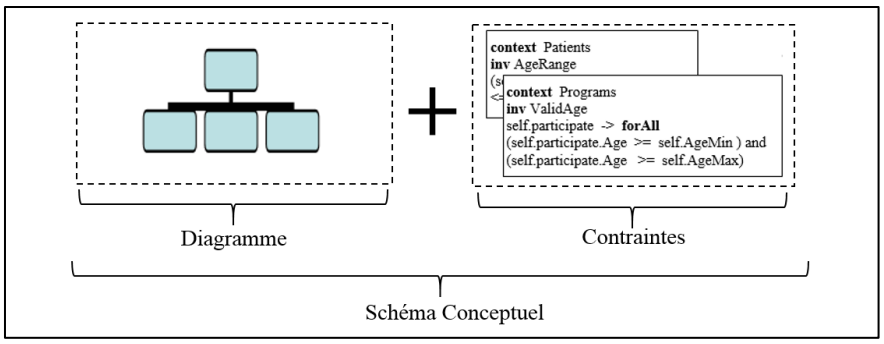


Figure 1.5 – Contenu du schéma conceptuel

13 "About the Unified Modeling Language Specification Version 2.5." [Online]. Available: http://www.omg.org/spec/UML/2.5/.

1.4 Architecture Dirigée par les Modèles (MDA)

Pour faire face à la complexité des applications informatiques, les spécialistes du logiciel se sont orientés vers l'ingénierie dirigée par les modèles (IDM, ou MDE pour Model Driven Engineering) [Combemale, 2008] en s'appuyant sur la modélisation. Il s'agit d'un paradigme qui se caractérise par une démarche permettant de générer une application à partir de modèles.

Le principe de base de l'IDM consiste à :

- Adopter des modèles comme éléments centraux dans le processus de développement d'une application,
- Automatiser les transformations entre ces modèles.

Dans cette optique, plusieurs spécifications ont été proposées telles que l'Architecture Dirigé par les Modèles (MDA pour Model Driven Architecture) [Hutchinson et al., 2011] proposée par l'Object Management Group[14] (OMG). Il s'agit d'une approche construite sur les mêmes concepts de base que l'IDM, à savoir le modèle et le métamodèle.

Les travaux de Bézivin et Gerbé [Bézivin and Gerbé., 2001] ainsi que ceux de Seidewitz [Seidewitz, 2003], définissent un modèle comme une description simplifiée d'un système sous la forme d'un ensemble de faits pour répondre à un objectif précis. Ainsi, le modèle doit capturer les informations nécessaires et suffisantes pour répondre à des questions sur le système modélisé.

Les concepts manipulés dans le modèle ainsi que les relations entre ces concepts doivent être exprimés dans un langage de modélisation bien défini. Ceci se fait au travers d'un métamodèle qui définit la syntaxe d'un langage spécifique de modélisation.

L'objectif de MDA est de décrire séparément les spécifications fonctionnelles et les spécifications d'implantation d'une application sur une plateforme donnée [Hutchinson et al., 2011]. Pour ceci, elle utilise trois modèles représentant les niveaux d'abstraction de l'application. Il s'agit (1) du modèle d'exigences (CIM pour Computation Independent Model) dans lequel aucune considération informatique n'apparait, (2) le modèle d'analyse et de conception (PIM pour Platform Independent Model) indépendant des détails techniques des plateformes d'exécution et (3) le modèle de code (PSM pour Platform Specific Model) spécifique à une plateforme particulière. La figure 1.6 montre ces modèles que nous décrivons comme suit :

- **CIM** : ce modèle décrit les besoins fonctionnels de l'application en utilisant le vocabulaire du métier (exprimés par un spécialiste du domaine de

[14] The Object Management Group (OMG): http://www.omg.org/

l'application). Les détails de la structure et de l'implantation de l'application restent encore indéterminés à ce niveau.

Les besoins recensés dans le CIM seront pris en compte dans les constructions des PIM et des PSM.

- **PIM** : ce modèle représente la logique métier spécifique à l'application en faisant abstraction des aspects techniques liés à la technologie de mise en œuvre.

- **PSM** : ce modèle correspondant à la spécification d'une application après projection sur une plateforme technologique donnée.

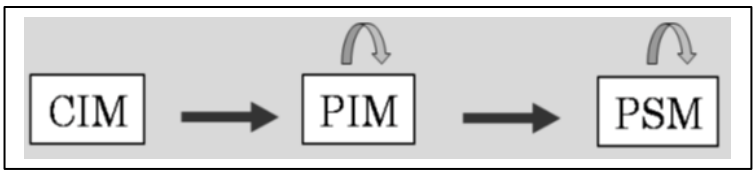


Figure 1.6 – Modèles MDA[15]

Le passage entre les différents modèles MDA se fait via une succession de transformations, essentiellement entre PIM et PSM. Notons toutefois que MDA permet plusieurs possibilités de transformations entre les modèles (cf. figure 1.6).

Une transformation correspond à l'application d'un ensemble de règles qui décrivent comment dériver un modèle cible à partir d'un modèle source. Les transformations de modèles peuvent être classées selon deux catégories : (1) la transformation d'un modèle vers du texte (M2T pour Model-To-Text) lorsque le résultat généré est du code et (2) la transformation d'un modèle vers un autre modèle (M2M pour Model-To-Model) quand le résultat est un modèle.

Pour exprimer les règles de transformation, l'OMG a défini le langage QVT[16] (Query/View/Transformation) ; il s'agit d'un langage standardisé de transformation de modèles que nous présentons dans le chapitre 5.

1.5 Problèmes traités et Motivation

Le stockage d'une BD massive sur un SGBD NoSQL soulève de nombreux problèmes ; nous en retenons deux.

Le premier problème est lié au passage du modèle conceptuel décrivant la BD massive vers un modèle physique NoSQL. Actuellement, ce passage s'avère

[15] OMG, M. (2003). Guide Version 1.0. 1, 2003. *Object Management Group*.

[16] OMG, Q. (2011). Meta Object Facility (MOF) 2.0
Query/View/Transformation, v1. 1.

fastidieux en raison de la complexité des modèles des BD à implanter et de la spécificité des modèles actuels associés aux SGBD NoSQL.

Un deuxième problème concerne le maintien de la cohérence des données. Dans les SGBD NoSQL, la gestion des contraintes dans les solutions NoSQL actuelles est limitée. Peu de mécanismes sont offerts pour garantir la cohérence des données stockées et, par conséquent, leur qualité.

Ce sont donc ces deux problèmes que nous allons traiter dans cette thèse par une approche basée sur MDA. L'utilité est d'assister l'utilisateur qui est chargé d'élaborer un modèle physique NoSQL à partir d'un modèle conceptuel.

Pour illustrer et motiver notre travail, nous utilisons un cas extrait d'une application médicale dont la base de données est décrite dans le formalisme UML. Il s'agit de la mise en place de programmes scientifiques consacrés au suivi d'une pathologie déterminée, qui regroupent une cinquantaine d'établissements hospitaliers européens (hôpitaux, cliniques et centres de soins spécialisés)

L'objectif premier d'un tel programme est de collecter des données significatives sur l'évolution temporelle de la pathologie, d'étudier ses interactions avec des maladies opportunes et d'évaluer l'influence de ses traitements à court et moyen termes. La durée d'un programme est décidée lors de son lancement et peut atteindre entre trois et dix ans.

Au début du programme, un médecin-coordonnateur est nommé au sein de chaque établissement participant ; il a la charge de :

- Déterminer une cohorte de patients sur la base du volontariat (en moyenne une trentaine de patients),
- Fixer les protocoles de soins en accord avec les objectifs du programme,
- Affecter des praticiens de l'établissement pour réaliser les mesures, les soins, les traitements et les consultations.

Avant le démarrage du programme, chaque coordonnateur doit saisir l'ensemble des données de départ :

- Les caractéristiques de chaque participant : identification, date de naissance, coordonnées de l'organisme social, médecin traitant, personne(s) de confiance,
- Les éventuels antécédents médicaux du patient (maladies graves, opérations chirurgicales, accidents) ; à cette occasion, peuvent être enregistrés des documents sous forme d'image : comptes rendus, radiographies, séquences vidéo,

- Les professionnels de santé affectés au programme : identification, spécialité, etc.,

- Le protocole de soin trimestriel spécifique à chaque patient et comportant les éléments suivants : mesures à collecter, traitements et contrôles à effectuer ; chacun de ces éléments étant associé à une fréquence ou à une date de réalisation et, le cas échéant, affecté à un professionnel de santé.

Dès le démarrage du programme, les données sont collectées soit à la volée grâce à des capteurs (thermomètre, tensiomètre, …) à la disposition des patients, soit par l'intermédiaire des personnels médicaux utilisant des tablettes mobiles ou des matériels spécialisés (scanner, IRM, etc.).

La consultation d'un patient par un médecin peut être effectuée soit en chambre, lorsque le patient est hospitalisé, soit en consultation ambulatoire. Elle est nécessairement programmée, c'est-à-dire prévue dans le protocole trimestriel ou ajoutée par un médecin. Lors d'une consultation, le médecin établit et enregistre un compte rendu ; il peut également enregistrer des mesures et tout type de documents remis par le patient ; enfin il peut rédiger une ou plusieurs ordonnances (radiographie, médicaments non prévus dans le protocole, suspension temporaire ou définitive d'un traitement).

Cette étude de cas est donc un exemple typique d'application Big Data où l'utilisation d'un SGBD NoSQL est nécessaire compte tenu de la spécificité des données. En effet, les données collectées par les établissements impliqués dans le cadre du programme médical, présentent les caractéristiques généralement admises pour le Big Data (les 3 V). Le volume des données médicales recueilli quotidiennement auprès des patients, peut atteindre, pour l'ensemble des établissements et sur trois années, plusieurs téraoctets. D'autre part, la nature des données saisies (mesures, radiographie, scintigraphies, etc.) est diversifiée et peut varier d'un patient à un autre selon son état de santé. Enfin, certaines données sont produites en flux continu par des capteurs ; elles doivent être traitées quasiment en temps réel car elles peuvent s'intégrer dans des processus sensibles au temps (mesures franchissant un seuil qui impliqueraient l'intervention d'un praticien en urgence par exemple).

L'extrait de diagramme UML de la figure 1.7 montre quelques classes pour la base de données d'un programme médical.

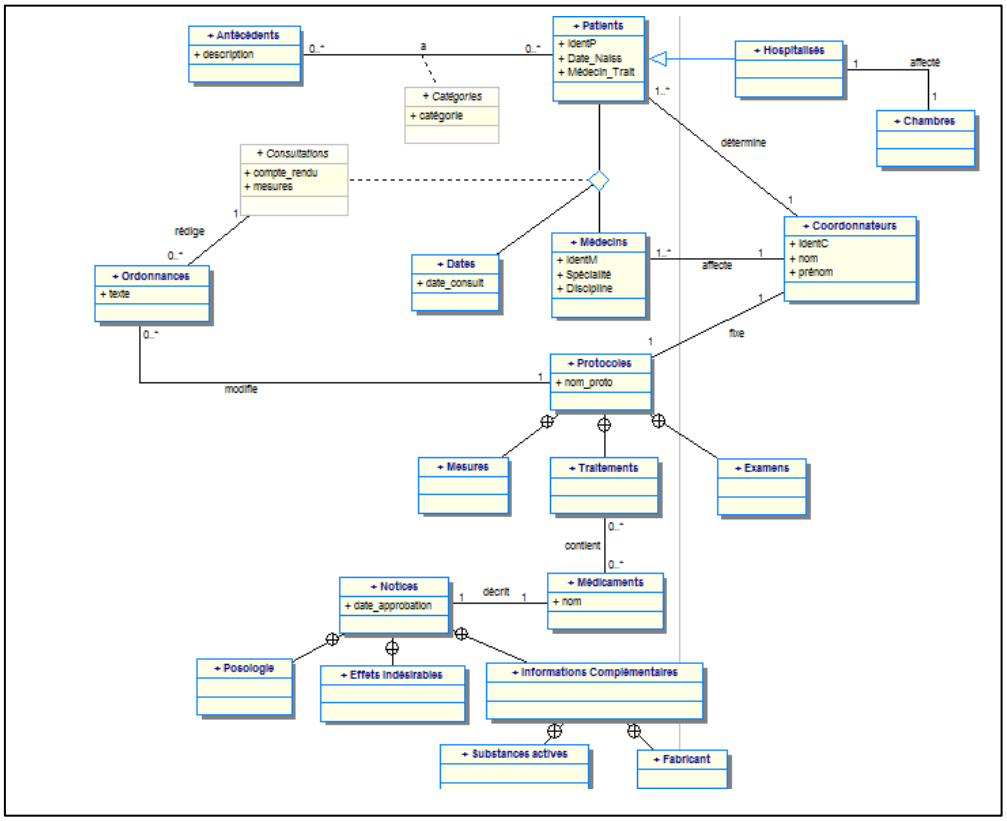


Figure 1.7 – Extrait d'un modèle de données

1.6 Notre contribution

Notre objectif est de répondre aux problèmes présentés dans la section précédente : le passage d'un modèle conceptuel incluant des contraintes en un modèle physique NoSQL. Nous disposons à l'entrée d'un modèle conceptuel de données exprimé à l'aide du formalisme UML et établi par un décideur (structures des données simples) ou un informaticien (cas plus complexes) suite à une étape d'analyse des besoins. Ce modèle comporte un DCL et un ensemble de contraintes d'intégrité exprimées en OCL. Pour assurer un passage automatisé de ce modèle vers un modèle physique NoSQL tout en intégrant la traduction de ses contraintes associées, nous proposons une approche dirigée par les modèles qui est composée de deux processus Object2NoSQL et OCL2Java. Ces processus se résument comme suit :

- Object2NoSQL :
    - **Point de départ** : Un diagramme de classes d'UML.
    - **Cible** : Un modèle physique NoSQL qui comporte :
        - Un modèle de données contenant les éléments nécessaires à l'implantation de la BD sur le SGBD NoSQL choisi par l'utilisateur,

- Un ensemble de directives d'assistance qui donne les modalités d'utilisation des attributs et d'implantation des relations sur le SGBD NoSQL.

- **Nature du processus** : transformations Model-To-Model : conceptuel > logique > physique. Ces transformations sont assurées par un ensemble de règles formalisées en QVT.

- **Description synthétique du processus** : Object2NoSQL permet de générer un modèle physique NoSQL à partir d'un DCL d'UML. Son principe consiste à produire un modèle logique NoSQL qui fait apparaitre principalement des tables et des relations binaires. Dans un second temps, ce modèle logique permet de générer un modèle physique propre au SGBD NoSQL choisi par l'utilisateur. Compte tenu de la généricité du modèle logique que nous avons proposé, le processus est capable de le transformer en un modèle physique pour l'une des plateformes d'implantation connues : colonnes, documents, graphes ou clé-valeur. Ce principe assure une vision générique de l'implantation des données et garantit l'indépendance de leur description vis-à-vis des spécificités techniques des plateformes NoSQL et de leurs évolutions. Ainsi, le processus Object2NoSQL passe par deux étapes. La première (Object2GenericModel) est chargée de transformer le DCL d'entrée (PIM conceptuel) en un modèle logique NoSQL (PIM logique). La deuxième transformation (GenericModel2PhysicalModel) génère, à partir du modèle logique, un modèle NoSQL (PSM) et un ensemble de directives d'assistance spécifiques au SGBD d'implantation choisi. Chacun de ces modèles (conceptuel, logique et physique) est conforme à un métamodèle que nous proposons pour décrire ses concepts.

- OCL2Java :

  - **Point de départ** : Des contraintes d'intégrité exprimées en OCL.

  - **Cible** : Un ensemble de méthodes java assurant la vérification des contraintes correspondantes.

  - **Nature du processus** : transformation Model-To-Model : conceptuel > logique formalisée en QVT suivie d'une transformation Model-To-Text : logique > physique formalisée en MOF2Text.

  - **Description synthétique du processus** : OCL2Java vise à compléter le processus Object2NoSQL décrit dans le point précédent en intégrant automatiquement d'autres contraintes qui ne sont pas prises en compte dans le modèle physique NoSQL généré. Pour chaque contrainte, le processus est réalisé en deux étapes successives qui produisent respectivement : (1) un modèle pivot de méthode java compatible avec des plateformes NoSQL de types différents (colonnes, documents, graphes et clé-valeur) puis (2) le code de la méthode à partir de son

modèle. Plusieurs codes peuvent être produits à partir du même modèle ; chacun d'eux est spécifique à un SGBD parce qu'il contient des requêtes d'accès aux données. Ce processus s'appuie sur l'utilisation des métamodèles OCL et Java pour effectuer les transformations.

Afin de vérifier la faisabilité de ces propositions, un prototype a été développé. Ce prototype est composé de deux modules :

- Le module de transformation du DCL : ce module consiste en un ensemble de règles de transformation QVT de type Model-To-Model, qui restituent un modèle NoSQL et un ensemble de directive d'assistance à partir du DCL entré par l'utilisateur. Ceci est réalisé en deux étapes : conceptuel > logique puis logique > physique.

- Le module de transformation des contraintes OCL : le rôle de ce module est d'intégrer les contraintes qui ne sont pas prises en compte dans le modèle physique généré par le module précédent. Pour ce faire, ce module prend en entrée des contraintes OCL et fournit en sortie des méthodes java en passant par un niveau logique où est défini un modèle de méthode java pour chaque contrainte. Ces méthodes vont permettre, dans un second temps, de vérifier les contraintes associées après leur exécution sur le SGBD d'implantation.

Nos propositions ont été également évaluées en comparant le temps nécessaire à notre processus automatisé avec le temps mis par un utilisateur pour obtenir manuellement un résultat équivalent. Pour un résultat (modèle physique et code de vérification des contraintes) quasiment identique, le temps de réalisation du processus a été réduit de 90% environ.

1.7 Organisation du mémoire

Cette thèse est organisée comme suit :

Le chapitre 2 situe l'état de l'art en matière d'implantation des BD sur les SGBD NoSQL. Nous commençons par présenter les propositions dans le contexte des entrepôts de données ; ces propositions ont généralement porté sur la transformation d'un modèle multidimensionnel en un modèle NoSQL. Ensuite, nous présentons les travaux qui ont étudié les mécanismes d'implantation d'une BD relationnelle sur des SGBD NoSQL. Finalement, nous présentons les solutions qui traitent de la transformation d'un modèle conceptuel UML en un modèle NoSQL. Ce chapitre se termine par une évaluation des travaux de recherche actuels et passés au regard de notre problématique.

Le chapitre 3 décrit le processus Object2NoSQL qui vise à formaliser et à automatiser l'élaboration d'un modèle physique NoSQL à partir d'un DCL d'UML décrivant une BD massive. Nous détaillons dans ce chapitre les composants de ce processus en précisant, pour chacun, les trois éléments suivants : la source, la cible et les règles de transformation associées.

Le chapitre 4 présente le processus OCL2Java de traduction des contraintes qui a pour but de compléter le processus Object2NoSQL. Nous détaillons dans ce chapitre la première étape OCL2JavaModel de type Model-To-Model qui traduit une contrainte OCL en un modèle de méthode java. Nous commençons par définir la structure des métamodèles OCL et Java proposés pour effectuer la transformation, puis nous explicitons le passage entre les éléments des modèles source et cible correspondants. La deuxième étape JavaModel2JavaCode de type M2T Model-To-Text prend en entrée chaque modèle de méthode java généré dans l'étape précédente (OCL2JavaModel) et fournit en sortie le code de vérification de la contrainte OCL correspondante. Le passage du modèle de méthode java vers le code correspondant est assuré par un ensemble de règles de transformation MOF2Text. Ces règles de transformation ainsi que le langage utilisé pour les formaliser sont présentés dans le chapitre 5 intitulé Expérimentation et Validation.

Le chapitre 5 décrit la réalisation de notre prototype. Ce prototype permet d'assister un utilisateur pour transformer un DCL d'UML en un modèle NoSQL. Il permet aussi de traduire les contraintes associées au DCL transformé en des méthodes java assurant la cohérence de la BD lors des mises à jours. Ce chapitre présente également l'évaluation de notre processus automatisé et les résultats que nous avons obtenus en le comparant avec le processus réalisé manuellement.

Chapitre 2 : Etat de l'Art

Au cours de ces dernières années, le besoin de gérer de façon effective les données massives ne cesse de croitre [Mpinda et al., 2015]. Dans [Karnitis et al., 2015], les auteurs affirment que l'utilisation des SGBD NoSQL est devenu une nécessité pour répondre à ce besoin ; ceci est dû aux nouvelles exigences posées par les applications utilisant des bases de données massives, notamment un stockage efficace à l'échelle, une grande flexibilité et un accès plus rapide à des volumes de données considérables.

2.1 Modélisation conceptuelle des données complexes

Une des caractéristiques essentielles des BD massives correspond à la notion de variété, c'est-à-dire qu'une BD peut contenir des données définies par des types non standard. Il s'agit d'une évolution des BD relationnelles (BD de 2ème génération) permettant de stocker des données généralement qualifiées d'objets complexes (BD de 3ème génération) telles que du textes, des graphiques, des documents, des séquences vidéo [Darmont et al., 2007]. Pour implanter de telles bases de données, des études ont porté sur la modélisation conceptuelle des objets, ceci dans les années 1990/2000.

La modélisation des données complexes a fait l'objet de nombreux travaux de recherche ; nous allons nous focaliser sur trois d'entre eux : [Tanasescu et al., 2005], [Pedersen and Jensen, 1999] et [Midouni et al., 2009] que nous avons considérés comme les travaux les plus marquants dans ce contexte.

L'approche de [Tanasescu et al., 2005] consiste à concevoir un diagramme UML pour identifier et représenter conceptuellement les données complexes afin de les préparer au processus de modélisation multidimensionnelle.

Dans le domaine médical, [Pedersen and Jensen, 1999] ont proposé un modèle multidimensionnel pour les données complexes qui modélise les données temporelles et imprécises respectivement par l'ajout du temps de validité et des probabilités au modèle.

Nous pouvons citer aussi le travail [Midouni et al., 2009] qui s'intéresse au traitement de la complexité des données médicales ; ils ont étendu un modèle en constellation en introduisant de nouveaux concepts permettant la présentation des données biomédicales. Dans le même article, les auteurs ont proposé une approche de modélisation et d'implantation d'un entrepôt médical en se basant sur le modèle étendu.

Dans la mesure où ces travaux sont anciens (antérieurs à 2010), il ne nous a pas paru utile de les détailler dans ce mémoire. Il convient cependant de noter que la plupart des articles portant sur les objets complexes consacrent le modèle de classes d'UML pour décrire la sémantique des données au niveau conceptuel.

Récemment, d'autres travaux ont proposé des processus de transformation d'un modèle de bases d'objets complexes en un modèle NoSQL.

2.2 Multidimensionnel vers NoSQL

Dans le contexte des entrepôts de données, des travaux ont porté sur le stockage d'un cube de données (généralement très volumineux) sur un SGBD NoSQL ; Dans [Chevalier et al., 2015], [Dehdouh et al., 2015], [Scabora et al., 2016] et [Freitas et al., 2016], les auteurs ont développé des processus pour traduire un modèle multidimensionnel en un modèle NoSQL.

Dans [Chevalier et al., 2015a] les auteurs ont étudié le passage d'un modèle multidimensionnel en constellation vers des modèles NoSQL. Dans cette étude, les deux modèles NoSQL orienté-colonnes et orienté-documents ont été retenus. Pour chacun d'eux, quatre processus de traduction ont été proposés :

- Le processus de traduction plate : il repose sur une dénormalisation des données du modèle multidimensionnel évitant ainsi les jointures entre le fait et les dimensions.

- Le processus de traduction par imbrication : il consiste à rassembler, d'une part, les mesures du fait et d'une autre part, les attributs de chaque dimension.

- Le processus de traduction hybride : il normalise les données des dimensions et nécessite l'utilisation d'auto-jointures entre les éléments (documents ou lignes) de la même collection pour l'orienté-documents ou de la même table pour l'orienté-colonnes.

- Le processus de traduction éclaté : il normalise les données des dimensions et nécessite l'utilisation de jointures entre plusieurs collections ou tables selon le modèle cible (documents ou colonnes).

Chaque processus prend en compte la construction du treillis dans les deux modèles NoSQL retenus. Une expérimentation des processus proposés a été réalisée sur les SGBD HBase et MongoDB.

Dans un autre article [Chevalier et al., 2015b], les mêmes auteurs ont proposé des processus de transformation intra-modèles et inter-modèles :

- Processus intra-modèles : les transformations s'appliquent sur des modèles de même type (orienté-documents ou orienté-colonnes).

- Processus inter-modèles : les conversions reposent sur des modèles de type différent (orienté-documents vers orienté-colonnes et inversement).

Ces deux processus consistent à convertir les données entre les différentes structures d'implantation définies dans [Chevalier et al., 2015a] : plate, imbriquée, hybride et éclatée.

L'article [Dehdouh et al., 2015] traite de l'implantation des entrepôts de données classiques mais de grande taille sur des systèmes NoSQL orientés-colonnes. Le processus d'implantation repose sur une architecture à 3 niveaux : conceptuel, logique et physique. Pour transformer un modèle conceptuel multidimensionnel en un modèle logique orienté-colonnes, trois solutions distinctes sont proposées :

- NLA (Normalized Logical Approach) : applique l'approche R-OLAP normalisé au modèle conceptuel multidimensionnel en étoile. Les tables de faits et de dimensions sont transformées en familles de colonnes séparément. Les mesures et les attributs des dimensions sont traduites par des colonnes simples.

- DLA (Denormalized Logical Approach) : applique l'approche R-OLAP dénormalisé au modèle conceptuel multidimensionnel en étoile. Les faits et les dimensions sont regroupés dans une même famille de colonnes appelée BigFactTable. Les mesures et les attributs des dimensions sont stockés dans des colonnes simples.

- DLA-CF (Denormalized Logical Approach) : identique à l'approche DLA, Les faits et les dimensions sont regroupés dans une même famille de colonnes. Mais les attributs de mesure sont rassemblés dans une famille de colonnes. Chaque dimension est également convertie en famille de colonnes.

Ces trois solutions utilisent les spécificités des systèmes Big Data en privilégiant certains choix de stockage des faits et des dimensions. Le passage vers le modèle physique est montré au travers d'une expérimentation particulière sur le système HBase ; les performances des trois solutions sont comparées.

Les auteurs de [Scabora et al., 2016] proposent d'appliquer la dénormalisation des données comme une tentative d'approximation d'un modèle multidimensionnel en étoile vers le modèle de données spécifique au SGBD NoSQL HBase. Leur solution porte sur la création d'une seule table HBase. Celle-ci est composée de deux familles de colonnes ; la première regroupe le fait et les dimensions qui sont fréquemment utilisées et la deuxième stocke les autres dimensions.

Dans [Freitas et al., 2016] les auteurs utilisent également la dénormalisation des données dans la solution de transformation des modèles proposée. Cette solution est compatible avec les deux types de SGBD NoSQL : orienté-colonnes et orienté-documents. A partir d'un modèle multidimensionnel en étoile, un ensemble de règles de transformation est appliqué pour générer un modèle conceptuel E-R ; celui-ci sera transformé dans un second temps en un modèle physique NoSQL. Selon le type de SGBD cible, le principe de dénormalisation appliqué est le suivant :

- Pour un SGBD orienté-colonnes : Toutes les tables sont stockées dans une seule famille de colonnes,
- Pour un SGBD orienté-documents : toutes les tables sont regroupées dans un seul document.

2.3 Relationnel vers NoSQL

D'autres travaux [Li, 2010], [Arora et al., 2013], [Mpinda et al., 2015] et [Rocha et al., 2015] ont étudié les mécanismes d'implantation d'une base de données relationnelle sur des systèmes NoSQL. Le volume des données manipulé justifie à lui seul ce choix d'implantation.

Dans [Li, 2010], la méthode proposée est basée sur des règles permettant la transformation d'un modèle relationnel en un modèle HBase ; les relations entre les tables (clés étrangères) sont traduites par l'ajout des familles de colonnes contenant des références.

[Arora et al., 2013] traite le passage d'un modèle relationnel vers le modèle orienté-documents de MongoDB. Un processus semi-automatique est proposé dans ce travail. A partir d'une base de données stockée sur le SGBD MySQL, l'utilisateur sélectionne la liste des tables qui vont être transformées en des collections dans la base de données MongoDB. Ensuite, Le processus s'exécute pour générer automatiquement des fichiers texte au format Pentaho Data Integration [Faster, 2014] ; ces fichiers chargent dans un second temps les données dans MongoDB depuis MySQL.

Dans [Mpinda et al., 2015], les auteurs décrivent un processus de migration d'un modèle de base de données relationnelle vers une structure adaptée à un SGBD NoSQL de type orienté-colonnes. Ce processus est composé de deux étapes : (1) la construction d'un modèle d'une base de données relationnelle et (2) la conversion de ce modèle en un autre spécifique aux bases de données orientées-colonnes.

Les auteurs dans [Rocha et al., 2015] propose un framework automatique qui permet de migrer une BD relationnelle vers une BD NoSQL. Deux SGBD ont été retenus dans ce travail : MySQL pour la source et MongoDB pour la cible. Le principe de ce framework est de préserver la façon dont les données sont modélisées au niveau de la source (i.e. dans la base de données relationnelle source), puis d'utiliser un modèle de données équivalent pour stocker ces données dans le SGBD NoSQL cible MongoDB. Pour ce faire, le processus s'appuie sur une couche d'abstraction entre une base de données relationnelle et un stockage NoSQL sous-jacent.

Des travaux similaires [Batra, 2016], [Lee and Zheng, 2015], [Vajk et al., 2013] et [Schram and Anderson, 2012] consistent à appliquer un ensemble de règles de correspondance entre le modèle relationnel et un modèle NoSQL. Ces travaux ciblent principalement les modèles NoSQL orienté-colonnes et orienté-documents.

2.4 UML vers NoSQL

Aujourd'hui, le modèle de données UML représente une référence en matière de représentation de modèles de bases de données complexes [Abelló, 2015]. Ce modèle conceptuel permet—de décrire la sémantique des objets métiers complexes ; il peut donc être appliqué à la description des bases de données massives qui contiennent des données de types et de formats variés.

A notre connaissance, très peu de travaux ont étudié la transformation d'un modèle conceptuel UML en un modèle physique NoSQL. On peut citer cependant les articles suivants : [Li et al., 2014], [Feng et al., 2015] et [Daniel et al., 2016].

Les auteurs de [Li et al., 2014] présentent une approche MDA pour traduire un diagramme de classes UML vers le modèle de données HBase. L'idée de base est de construire des métamodèles correspondant au diagramme de classes UML et au modèle de données orienté-colonnes de HBase, puis de proposer des règles de transformation entre les éléments des deux métamodèles proposés. Ces règles permettent de transformer un DCL directement en un modèle d'implantation spécifique au système HBase.

Un travail similaire [Feng et al., 2015] propose une approche dirigée par les modèles pour transformer directement un diagramme de classes UML en un modèle physique spécifique au SGBD Cassandra.

Dans [Daniel et al., 2016] les auteurs décrivent le passage d'un modèle conceptuel UML/OCL vers un modèle NoSQL orienté-graphes. Les règles assurant ce passage sont spécifiques aux BD orientées-graphes qui sont généralement utilisées pour stocker et interroger des données complexes fortement liées comme les données issues des réseaux sociaux.

Dans le tableau 2.1, nous synthétisons les travaux [Li et al., 2014], [Feng et al., 2015] et [Daniel et al., 2016] dont la solution proposée est proche de la nôtre. Nous considérons les caractéristiques suivantes :

- Transformation des structures de données,
- Transformation des contraintes,
- Niveaux de modélisation,
- Compatibilité avec les SGBD NoSQL.

	Transformation des structures de données	Transformation des contraintes	Niveaux de modélisation			Compatibilité avec les SGBD NoSQL			
			Conceptuel	Logique	Physique	Orienté-Colonnes	Orienté-Documents	Orienté-Graphes	Orienté Clé-Valeur
[Li et al., 2014]	x		x		x	x			
[Feng et al., 2015]	x		x		x	x			
[Gwendal et al., 2016]	x	x	x		x			x	

Tableau 2.1 - Tableau comparatif des processus de transformation d'un DCL en un modèle NoSQL

2.5 Travaux connexes

Dans certains SGBD NoSQL, les possibilités de structuration des données sont nombreuses. Chacune d'elles peut avoir un impact positif ou négatif sur la qualité des applications, notamment en matière de temps d'accès aux données, coût de navigation dans ces structures et la redondance de données qui peut être engendrée. Les travaux de [Gomez et al., 2018] ont étudié la qualité de structuration des données dans les SGBD NoSQL. Ils ont proposé un ensemble de métriques structurelles pour des modèles NoSQL. Ces métriques permettent d'assister l'utilisateur pour choisir la structuration des données la plus adaptée selon des critères de qualité tels que la lisibilité et la maintenabilité des modèles de données. Pour définir ces métriques, les auteurs se sont appuyés, entre autres, sur les métriques utilisées en Génie logiciel.

2.6 Positionnement

A présent, nous effectuons le positionnement de nos travaux au regard des articles de recherche que nous venons de présenter et dont les problématiques et/ou les solutions proposées sont proches des nôtres.

Les articles [Chevalier et al., 2015], [Dehdouh et al., 2015], [Scabora et al., 2016] et [Freitas et al., 2016] s'inscrivent dans le contexte de l'entreposage des données puisqu'ils étudient les règles de passage d'un modèle multidimensionnel en un modèle physique NoSQL. Deux SGBD NoSQL ont été retenues : le système orienté-colonnes HBase et le système orienté-documents MongoDB. Bien que le point de départ du processus (un modèle multidimensionnel) se situe au niveau conceptuel, ce modèle ne présente pas les mêmes caractéristiques qu'un DCL d'UML en terme de complexité ; notamment, il comporte exclusivement deux classes Faits et Dimensions et un type de lien unique entre ces deux classes. Dans notre processus, nous considérons des classes d'objets comportant des attributs atomiques et multivalués, des relations d'association, de composition, d'agrégation et d'héritage ainsi que des classes d'associations.

Par ailleurs les auteurs de [Li, 2010], [Arora et al., 2013], [Mpinda et al., 2015] et [Rocha et al., 2015] traitent de la transformation d'un modèle relationnel en un modèle physique NoSQL. Ces travaux répondent bien aux attentes concrètes des entreprises qui, face aux évolutions récentes de l'informatique, souhaitent stocker leurs bases de données volumineuses dans des systèmes NoSQL. Mais, la source du processus de transformation, ici un modèle relationnel, ne présente pas la richesse sémantique que l'on trouve dans les BD massives et que l'on peut exprimer dans un DCL (notamment grâce aux différents types de liens entre classes : agrégation, composition, héritage, …).

D'autre part, les travaux présentés dans [Li et al., 2014], [Feng et al., 2015] et [Daniel et al., 2016] ont pour objet de spécifier un processus de transformation MDA d'un modèle conceptuel (DCL) vers un modèle physique NoSQL. Concernant la structure des données, les processus de transformation présentés dans ces travaux ne proposent pas un niveau intermédiaire (le niveau logique) qui permettrait de rendre le résultat du processus indépendant d'une plateforme particulière. Ainsi, chacun d'eux s'applique uniquement à un seul type de systèmes NoSQL (orienté-colonnes dans [Li et al., 2014] [Feng et al., 2015] et orienté-graphes dans [Daniel et al., 2016]).

Enfin, concernant les contraintes d'intégrité, le processus proposé dans [Li et al., 2014] et [Feng et al., 2015] n'en tient pas compte. Dans le travail [Daniel et al., 2016], une fois le modèle orienté-graphes créé, une autre transformation est effectuée pour traduire les expressions OCL définies au niveau conceptuel en des requêtes exprimées dans le langage Gremlin. Il s'agit d'un langage de requêtes spécifique aux systèmes de type orienté-graphes et incompatible avec les autres types de systèmes NoSQL (colonnes et documents). Dans notre processus, le

niveau logique est compatible avec les quatre catégories de systèmes (colonnes, documents, graphes et clé-valeur).

Chapitre 3 : Processus Automatisé de Transformation d'un DCL

Rappelons ici les points essentiels à la base de notre démarche d'implantation.

L'absence d'un modèle (schemaless) lors de la création d'une BD, est une caractéristique que l'on trouve dans la plupart des SGBD NoSQL. Cette propriété offre une souplesse indéniable en facilitant l'évolution des structures de données pendant leur exploitation. Mais elle concerne exclusivement le niveau physique, c'est-à-dire le niveau de l'implantation d'une BD.

Pour mettre en œuvre les traitements métiers (applications lourdes, traitements décisionnels), il est préférable, voire indispensable, de disposer d'un modèle conceptuel qui décrit la sémantique des données en faisant abstraction des aspects physiques (structures d'implantation, mécanismes d'accès, etc.). [Abelló, 2015].

Le langage UML offre un standard de modélisation pour décrire les données complexes sous forme d'objets ; son modèle conceptuel des données est largement reconnu comme un modèle sémantique performant. Dans notre application médicale présentée dans le chapitre 1, la BD contiendra à la fois des données standard (coordonnées des patients et des personnels soignants, descriptifs de consultations, etc.), des données multimédia (textes explicatifs, notices de médicaments, radiographies, extraits de dossiers médicaux, …), des tableaux de grande dimension (mesures produites par des capteurs), etc. Nous avons donc formalisé les données contenues dans la BD médicale par un diagramme de classes du langage UML.

Nos travaux sont destinés à un utilisateur qui est chargé d'implanter une BD massive sur un SGBD NoSQL. Afin de l'assister dans cette tâche, l'objectif de cette thèse est de transformer un modèle conceptuel UML décrivant la BD massive en un modèle physique NoSQL. Dans ce chapitre, nous détaillons la solution que nous proposons afin de répondre à cet objectif. Il s'agit d'une approche dirigée par les modèles ; elle vise à formaliser et à automatiser l'élaboration d'un modèle physique NoSQL à partir d'un modèle conceptuel de données exprimé en UML.

Notons enfin que ce processus automatisé peut être mis en œuvre par un informaticien (cas d'une BD complexe) ou directement par un décideur (cas d'une BD simple dont les classes et liens sont peu nombreux). Afin de ne pas distinguer ces deux types d'acteurs, nous emploierons le terme utilisateur pour les désigner dans la suite de ce mémoire.

Plan du chapitre. La section 3.1 décrit notre approche. Les sections 3.2 et 3.3 définissent respectivement les transformations automatiques vers les niveaux logique et physique.

3.1 Aperçu de notre processus

Les travaux que nous présentons visent à assister un utilisateur dans l'implantation d'une BD massive sur un SGBD NoSQL. Pour ce faire, nous proposons le processus Object2NoSQL qui assure le passage du modèle conceptuel vers un modèle physique NoSQL. Un aperçu de notre approche est illustré dans la figure 3.1. Nous disposons à l'entrée d'un modèle conceptuel UML qui décrit des structures d'objets métiers. A partir de ce modèle, notre processus lui applique une chaîne de transformations qui le traduit en une succession de modèles pour aboutir à un modèle physique NoSQL.

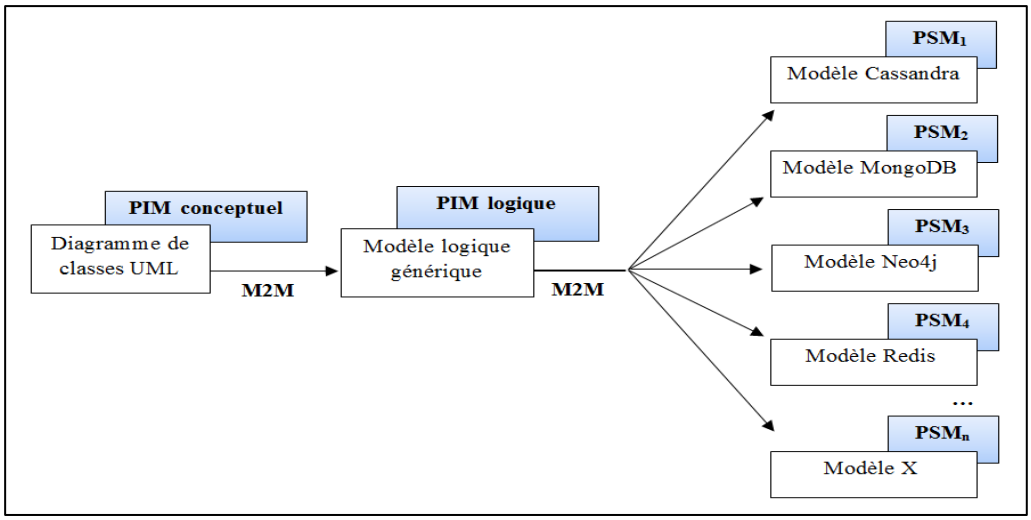


Figure 3.1 - Les niveaux de modélisation de notre approche

Compte tenu des intérêts de l'architecture dirigée par les modèles (MDA) présentés dans chapitre 1, nous l'utilisons pour formaliser et automatiser notre démarche. Généralement, une démarche MDA propose trois points de vue associés à leurs modèles respectifs : le CIM, le PIM et le PSM. Etant donné que l'entrée de notre processus est un modèle conceptuel des données (DCL d'UML), et que nous n'avons pas étudié le processus d'élaboration de ce modèle, nous retenons uniquement les niveaux PIM et PSM.

Au niveau PIM, nous considérons deux sortes de modèles : Le PIM conceptuel et le PIM logique :

- PIM conceptuel – Il s'agit d'un modèle qui décrit les données sous ses seuls aspects métier. Conformément à la justification donnée dans l'introduction de ce chapitre, nous décrivons le PIM conceptuel sous la forme d'un diagramme de classes (DCL) d'UML.

- PIM logique – L'introduction de ce modèle intermédiaire est justifiée en se référant à l'architecture ANSI/SPARC. Cette architecture fait apparaître les niveaux de description conceptuel (description métier) et interne (description technique). Le niveau interne peut être décomposé en niveau logique et

niveau physique (voir la figure 3.2). Le niveau logique décrit l'organisation technique des données en faisant abstraction des spécificités d'un SGBD particulier. Il correspond à une vision générique de l'implantation des données et garantit l'indépendance de la description de ces données vis-à-vis des plateformes d'implantation.
Dans notre scénario, le modèle logique correspond à un modèle générique NoSQL pouvant être implanté sur différentes plateformes NoSQL de type colonnes, documents, graphes ou clé-valeur. Son intérêt est d'apporter une certaine stabilité dans notre processus en limitant les impacts liés aux évolutions technologiques (fréquentes) des plateformes NoSQL (voire à leur remplacement). Toute évolution des caractéristiques techniques d'un SGBD apparaitra dans le modèle physique (PSM) mais n'affectera pas le modèle générique (PIM logique). Le processus de transformation logique-physique sera alors relancé et il n'y aura aucune incidence sur la transformation conceptuel-logique. Ceci simplifie les transformations et constitue un gain de temps notable pour les utilisateurs.

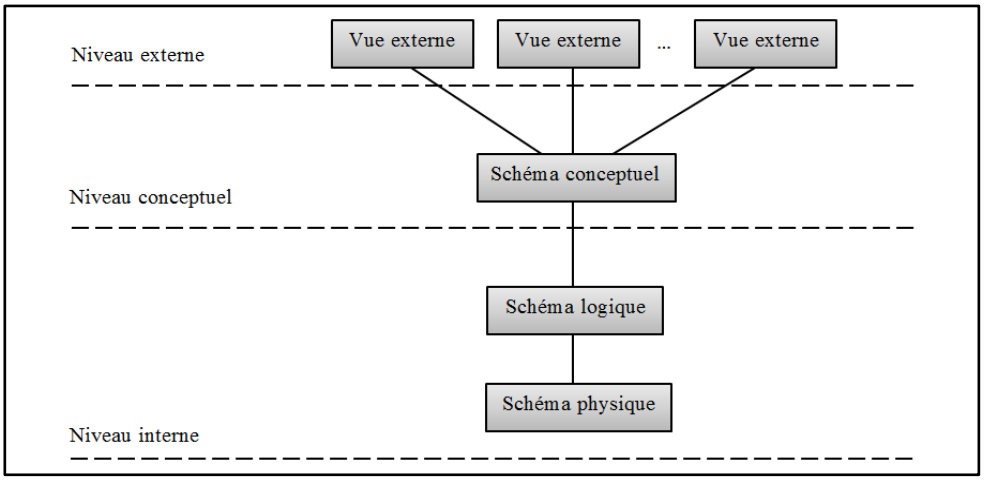


Figure 3.2 - Architecture ANSI-SPARC pour la description des données

Au niveau PSM, nous adoptons des modèles physiques qui correspondent à des plateformes NoSQL particulières. Pour illustrer notre étude, nous avons choisi un représentant de chaque type de SGBD NoSQL : Cassandra (orienté-colonnes), MongoDB (orienté-documents), Neo4j (orienté-graphes) et Redis (orienté clé-valeur).

Le passage d'un modèle à un autre se fait en utilisant des transformations M2M (Model-To-Model). Nous allons formaliser par la suite ces transformations en utilisant le standard QVT (Query View Transformation) défini par l'OMG pour la transformation de modèles (voir les Sections 3.2.3 et 3.3.3).

Notons que chaque passage conceptuel-logique et logique-physique est automatique. Mais le processus global conceptuel-physique peut, dans certains cas, nécessiter une intervention humaine (pour choisir le mode d'implantation

des liens). Nous parlons dans ce cas de processus automatisé. La figure 3.3 montre les différents composants de notre processus.

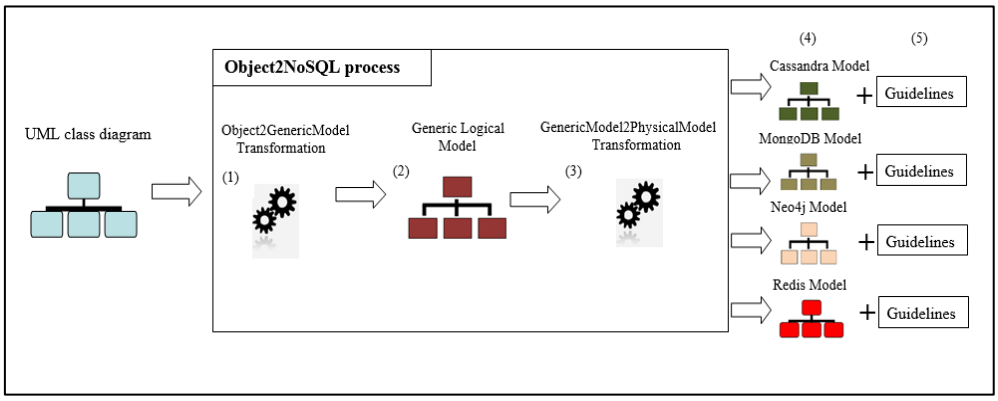


Figure 3.3 - Aperçu de processus Object2NoSQL

La transformation Object2GenericModel (1) est la première étape dans le processus Object2NoSQL. Elle traduit le diagramme de classes UML d'entrée en un modèle générique NoSQL (2) ; ce modèle est conforme au métamodèle du PIM logique présenté dans la section suivante. La transformation GenericModel2PhysicalModel (3) est la deuxième étape qui génère des modèles physiques NoSQL (PSM) (4) et un ensemble de directives d'assistance (5) spécifiques à chaque système à partir du modèle générique.

Les directives d'assistance sont utiles à l'utilisateur :

- pour implanter les données, notamment les relations, selon les techniques inhérentes à un SGBD choisi,
- pour élaborer l'interface de l'application qui permettra entre autre de saisir les données.

En effet, selon la manière dont nous définissons leur modèle physique de données, les SGBD NoSQL peuvent être classés en trois catégories :

(a) Des SGBD où le modèle est fixé préalablement à toute alimentation de données. Autrement dit, comme dans les systèmes relationnels, chaque table doit être spécifiée en précisant le nom et le type des attributs ; la saisie des données ne peut se faire qu'une fois le modèle entièrement défini ; c'est par exemple le cas de Cassandra (système orienté-colonnes) et de Riak TS (système clé-valeur),

(b) Des SGBD où seulement une partie du modèle est déclaré avant l'alimentation ; généralement, il s'agit de préciser les noms des tables ; par exemple dans les systèmes MongoDB (système orienté-documents) et HBase (système orienté-colonnes),

(c) Des SGBD où le modèle est spécifié au fur et à mesure de la saisie des données. L'utilisateur insère chaque ligne en précisant le nom de la table ainsi

que les noms des attributs ; c'est le cas des systèmes Neo4j (système orienté-graphes) et Redis (système clé-valeur).

Dans les deux dernières catégories, les systèmes n'exigent pas la définition complète du modèle physique avant la saisie des données. Cependant, l'utilisateur doit avoir connaissances des attributs (leurs nom et types) et de la manière d'implanter les relations, d'où l'intérêt des directives d'assistance.

Dans les sections suivantes, nous détaillons les composants du processus Object2NoSQL en précisant, pour chacun, les trois éléments suivants : (a) la source, (b) la cible et (c) les règles de transformation associées.

3.2 La transformation Object2GenericModel

Cette section présente la première transformation dans notre processus qui traduit le PIM conceptuel (diagramme de classes UML) en un PIM logique (modèle générique NoSQL). Nous commençons par la définition des source et cible de cette transformation, avant de présenter les règles qui lui sont associées.

3.2.1 La source : PIM conceptuel

Avant d'assurer un passage automatique d'un DCL vers le modèle générique NoSQL, nous formalisons préalablement les concepts présents dans le modèle de données d'UML.

Définition 1. Un diagramme de classes DCL est défini par (N, C, L, C^{asso}) où

- N est le nom du diagramme,
- C = $\{c_1, \dots, c_n\}$ est un ensemble de classes,
- L = $\{l_1, \dots, l_m\}$ est un ensemble de liens,
- C^{asso} = $\{c_1^{asso}, \dots, c_k^{asso}\}$ est un ensemble de classes d'associations.

Nous rappelons qu'une classe définit la structure (attributs) et le comportement (opérations) d'un ensemble d'objets ayant une sémantique et des propriétés communes. Comme le montre la définition 2, notre approche prend en compte uniquement la partie structurelle d'une classe.

Définition 2. ∀ i ∈ [1..n], une classe c_i ∈ C est définie par (N, A^c) où:

- c_i.N est le nom identifiant la classe,
- c_i. A^c = $\{a_1^c, \dots, a_q^c\} \cup \{Id^c\}$ est un ensemble d'attributs de la classe c_i, avec q ≥ 1, où :
  - ∀ j ∈ [1..q], le schéma d'un attribut $a_j^c \in A^c$ est un couple (N, c) où:
    - a_j^c.N est le nom identifiant l'attribut,
    - a_j^c.c est la classe qui définit l'attribut ; c peut être une classe prédéfinie, c'est-à-dire un type de données prédéfini (String, Integer, Date,…) ou une classe définie explicitement par l'utilisateur.
  - Id^c est un identificateur d'objets. Il s'agit d'un attribut particulier de c que nous avons défini afin de créer des références permettant d'accéder aux objets de c (voir section 3.3). Comme tout attribut, Id^c possède un nom Id^c.N et un type noté « Oid ».

Un lien est une association entre deux ou plusieurs classes d'objets ; il exprime des connexions sémantiques entre les objets. La définition 3 explicite ce concept en se limitant aux quatre types de liens les plus fréquents : la composition, l'agrégation, l'héritage et l'association.

Définition 3. ∀ i ∈ [1..m], un lien l_i ∈ L est défini par (N, Ty, CP^l) où:

- l_i.N est le nom identifiant le lien,
- l_i.Ty est le type de lien. Dans nos travaux nous considérons uniquement les types suivants : l'association, la composition, l'agrégation et l'héritage,
- l_i. CP^l = $\{cp_1^l, \dots, cp_f^l\}$ est un ensemble de couples, avec f ≥ 2 est le degré de l_i. ∀ j ∈ [1..f], $cp_j^l = (c, cr^c)$ où:
  - cp_j^l.c est une classe liée,
  - cp_j^l . cr^c est la cardinalité placée du côté de c. Notons que $cp_j^l.cr^c$ contiendra la valeur *Nulle* si aucune cardinalité n'est indiqué à côté de c ; C'est le cas d'un lien d'héritage et d'un lien n-aire. Dans ce dernier, les cardinalités sont assez difficiles à interpréter ; généralement elles ne sont pas précisées.

Dans un DCL d'UML, une classe d'associations possède à la fois les caractéristiques d'une classe et celles d'un lien. Ainsi, la définition d'une classe d'associations (Définition 4) est composée de deux parties : la première (en rouge) correspond à la définition d'une classe (Définition 1) et la deuxième (en vert) correspond à la définition d'un lien (Définition 2).

Définition 4. ∀ i ∈ [1..k], une classe d'associations $c_i^{asso} \in C^{asso}$ est défini par (N, $A^{c^{asso}}$, $CP^{c^{asso}}$) où:

- c_i^{asso}.N est le nom identifiant la classe d'associations,
- $c_i^{asso}.A^{c^{asso}} = \{a_1^{c^{asso}}, \dots, a_p^{c^{asso}}\} \cup \{Id^{c^{asso}}\}$ est un ensemble d'attributs de la classe d'associations, avec p ≥ 1, où :
  - ∀ j ∈ [1..p], un attribut $a_j^{c^{asso}} \in A^{c^{asso}}$ est défini par (N, c) où:
    - $a_j^{c^{asso}}$.N est le nom identifiant l'attribut,
    - $a_j^{c^{asso}}$.c est la classe qui définit l'attribut,
- $Id^{c^{asso}}$ est un identificateur de liens. Il s'agit d'un attribut particulier permettant de distinguer les liens de c^{asso}. Comme tout attribut, $Id^{c^{asso}}$ possède un nom et un type noté Oid.

- $c_i^{asso}.CP^{c^{asso}} = \{cp_1^{c^{asso}}, \dots, cp_l^{c^{asso}}\}$ est un ensemble de couples, avec l ≥ 2. ∀ r ∈ [1..*l*], $cp_r^{c^{asso}} = (c, cr^c)$ où:
  - $cp_r^{c^{asso}}$.c est une classe liée,
  - $cp_r^{c^{asso}}.cr^c$ est la cardinalité placé du côté de c. Notons que $cp_r^{c^{asso}}.cr^c$ peut contenir la valeur *Nulle* si aucune cardinalité n'est indiqué à côté de c (c'est le cas généralement pour une classe d'associations reliant 3 ou plusieurs classes).

Nous présentons ces différents concepts à travers un métamodèle défini par l'OMG[17] et que nous avons adapté à notre PIM conceptuel. Ce métamodèle ainsi que tous les métamodèles présentés dans ce mémoire sont implantés et validés en utilisant la plateforme Eclipse Modeling Framework (EMF) présentée dans le chapitre 5.

Comme l'illustre la figure 3.6, le métamodèle du PIM conceptuel montre les principaux éléments composant un modèle UML ainsi que leurs caractéristiques structurelles. Un diagramme de classes UML est composé de *Classes*, de *Liens* et de *Classes d'Associations*. Une *Classe* est composée d'un ou plusieurs *Attributs* correspondant à ses caractéristiques structurelles. Un lien relie deux ou plusieurs classes. Ainsi, il se compose d'au moins deux extrémités (LinkEnd), chacune représentant une connexion entre le lien et une classe ; la partie structurelle d'un lien est donc définie par ses extrémités. Comme nous l'avons présenté dans la définition 4, une *Classe d'Association* est un élément possédant à la fois les caractéristiques d'une classe et celles d'un lien. Autrement dit, une classe d'association peut être considérée comme un lien qui possède également les caractéristiques d'une classe ou comme une classe qui possède également les caractéristiques d'un lien.

[17] *https://www.omg.org/spec/UML/2.4.1/About-UML/*

Ce métamodèle permet d'exprimer certaines formes de *Contraintes*, notamment les contraintes structurelles (les attributs dans les classes, les différents types de relations entre classes, les cardinalités, etc.) et les contraintes de type (typage des attributs). Ces contraintes sont très utiles, mais se révèlent insuffisantes. UML permet de spécifier explicitement des contraintes additionnelles sur des éléments de modèle (*constrainedElement*). Une contrainte désigne une restriction/condition exprimée sous forme d'instruction dans un langage textuel naturel ou formel ; cette instruction constitue le corps (*body*) de la contrainte. Le chapitre 4 traite de ces contraintes ainsi que de leur intégration dans notre approche.

3.2.2 La cible : PIM logique

Notre modèle générique fait apparaître principalement des tables et des relations binaires. Dans cette section, nous définissons chacun de ces concepts et nous proposons un métamodèle pour les décrire.

Définition 5. Une base de données BD est définie par un triplet (N, T, R) où:

- N est le nom de la BD,
- T = $\{t_1, \dots, t_n\}$ est un ensemble de tables,
- R = $\{r_1, \dots, r_h\}$ est un ensemble de relations binaires.

Une table est un regroupement de lignes n'ayant pas nécessairement les mêmes schémas. Chaque ligne se présente sous la forme d'un couple (Identificateur : Valeur) ; la valeur d'une ligne correspond à un ensemble de couples (attribut : valeur). Tout attribut peut être atomique (de type standard : Integer, Float, String, etc.) ou complexe (composé d'un ensemble d'autres attributs). La figure 3.4 illustre des exemples de schémas de lignes dans notre modèle générique NoSQL.

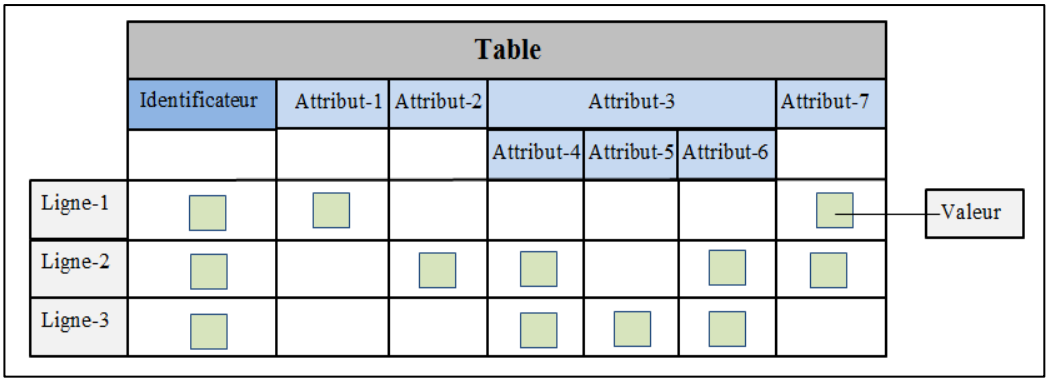


Figure 3.4 – Exemples de schémas de lignes dans une table logique

Définition 6. ∀ i ∈ [1..n], une table t_i ∈ T est défini par (N, A^t) où:

- t_i.N est le nom identifiant la table,
- t_i. $A^t = \{a_1^t, \dots, a_q^t\} \cup \{Id^t\}$ est un ensemble d'attributs qui seront utilisés pour définir les lignes de t, où :
  - ∀ i ∈ [1..q], le schéma d'un attribut $a_i^t \in A^t$ est un couple (N, Ty) où:
    - a_i^t.N est le nom identifiant l'attribut,
    - a_i^t.Ty est le type de l'attribut.
  - Id^t est un identificateur de lignes. Il s'agit d'un attribut particulier de t qui possède un nom Id^t.N et un type noté « Rid ». Nous allons utiliser cet attribut par la suite (voir section 3.3) afin de créer des liens entre les tables au niveau physique conformément aux principes des liens spécifiés dans le langage ODL de l'ODMG [Bartels et al., 1997].

Notons qu'au niveau logique, nous avons uniquement des relations binaires entre les tables. Ainsi, un lien conceptuel de degré n (avec n > 2) est traduit par une nouvelle table et n relations binaires émanant de cette table.

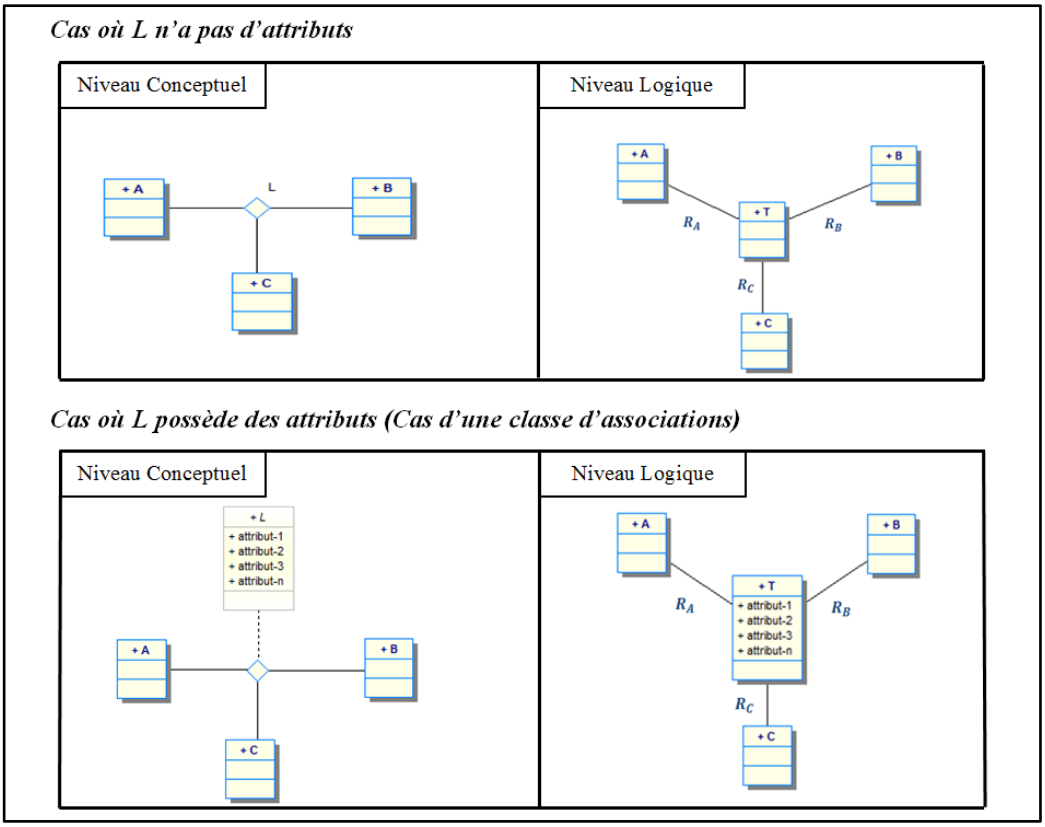


Figure 3.5 – Transformation d'un lien n-aire au niveau logique

Par exemple, dans la figure 3.5, le lien conceptuel L entre les classes A, B et C est traduit en une nouvelle table T, et trois relations binaires : R_A reliant T et A,

R_B reliant T et B et R_C reliant T et C. Dans le cas où L comporte des attributs, ceux-ci sont reportés dans T.

Nous proposons une formalisation de cette traduction conceptuel-logique des liens n-aires dans la section suivante : Règles de transformation. Nous montrons aussi, dans la seconde transformation GenericModel2PhysicalModel (section 3.3), que cette traduction peut être implantée sous différentes formats physiques et sur des plateformes NoSQL distinctes.

Définition 7. ∀ i ∈ [1..h], une relation binaire r_i ∈ R est définie par (N, CP^r) où :

- r_i.N est le nom désignant la relation,
- $r_i.CP^r$ = $\{cp_1^r, cp_2^r\}$ est un ensemble de deux couples, où : ∀ i ∈ {1,2} $cp_i = (t_i, cr^{t_i})$ où:
  - $cp_i.t_i$ est une table liée,
  - $cp_i.\, cr^{t_i}$ est la cardinalité placé du côté de t_i.

Nous décrivons les concepts de notre PIM logique présentés dans cette section à travers le métamodèle de la figure 3.7. Comme nous l'avons présenté dans la Définition 5, une base de données est composée de *Tables* et de *Relations Binaires*. Une table est un regroupement de *Lignes* ; chaque ligne étant constituée d'un *Identificateur* et d'une *Valeur*. Toute valeur est composée d'un ou plusieurs *Couples* (Attribut : Valeur). Un *Attribut* comporte un nom et un type, et il possède une *Valeur* qui peut être soit *Atomique*, soit *Complexe* (i.e. composée d'autres attributs). Nous présentons ceci dans le métamodèle par la contrainte {*XOR*} ; une contrainte UML prédéfinie.

Les tables sont liées entre elles par des relations binaires. Une relation binaire se compose de deux extrémités (*BRelationshipEnd)*.

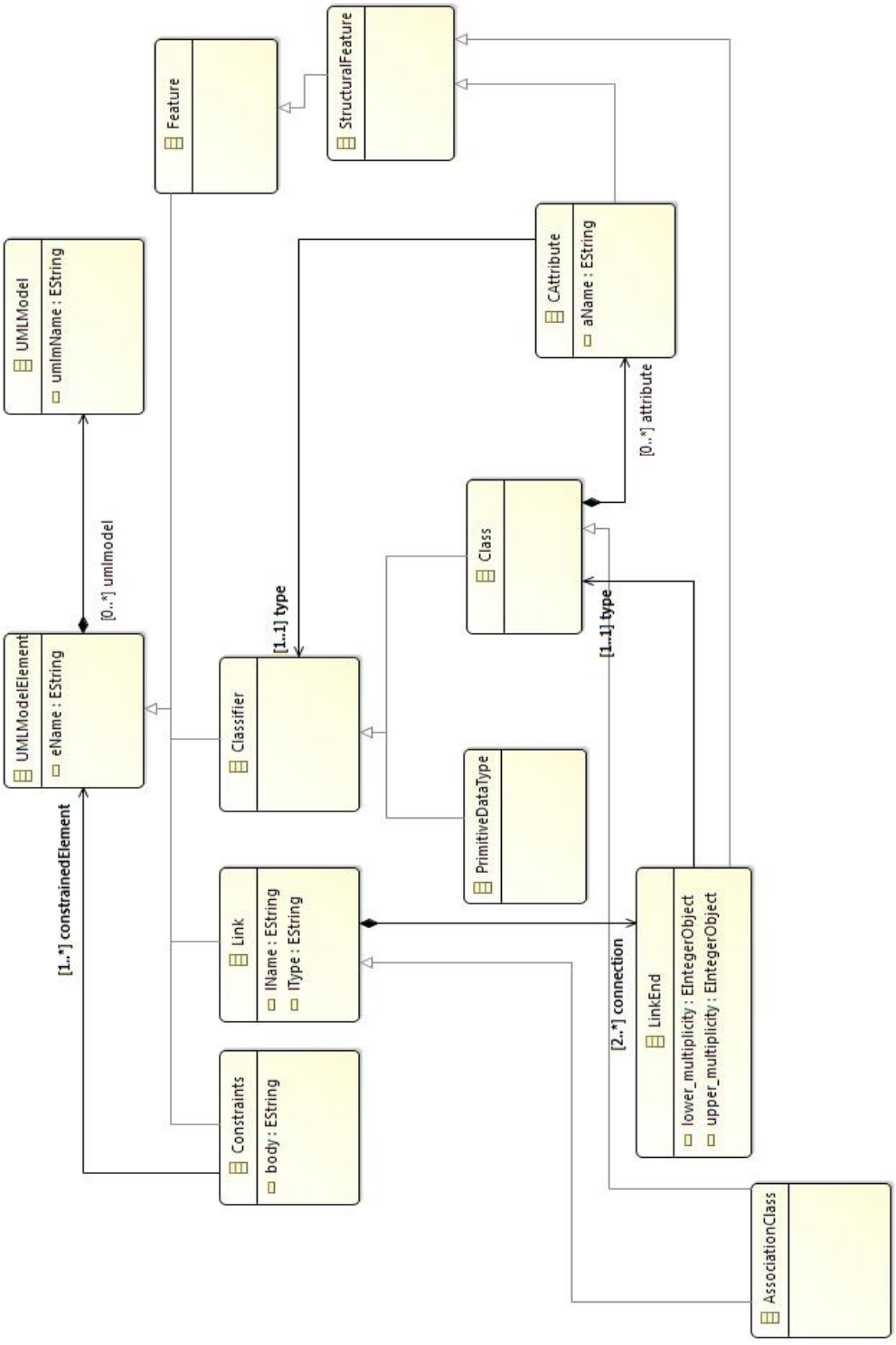


Figure 3.6 – Métamodèle du PIM conceptuel

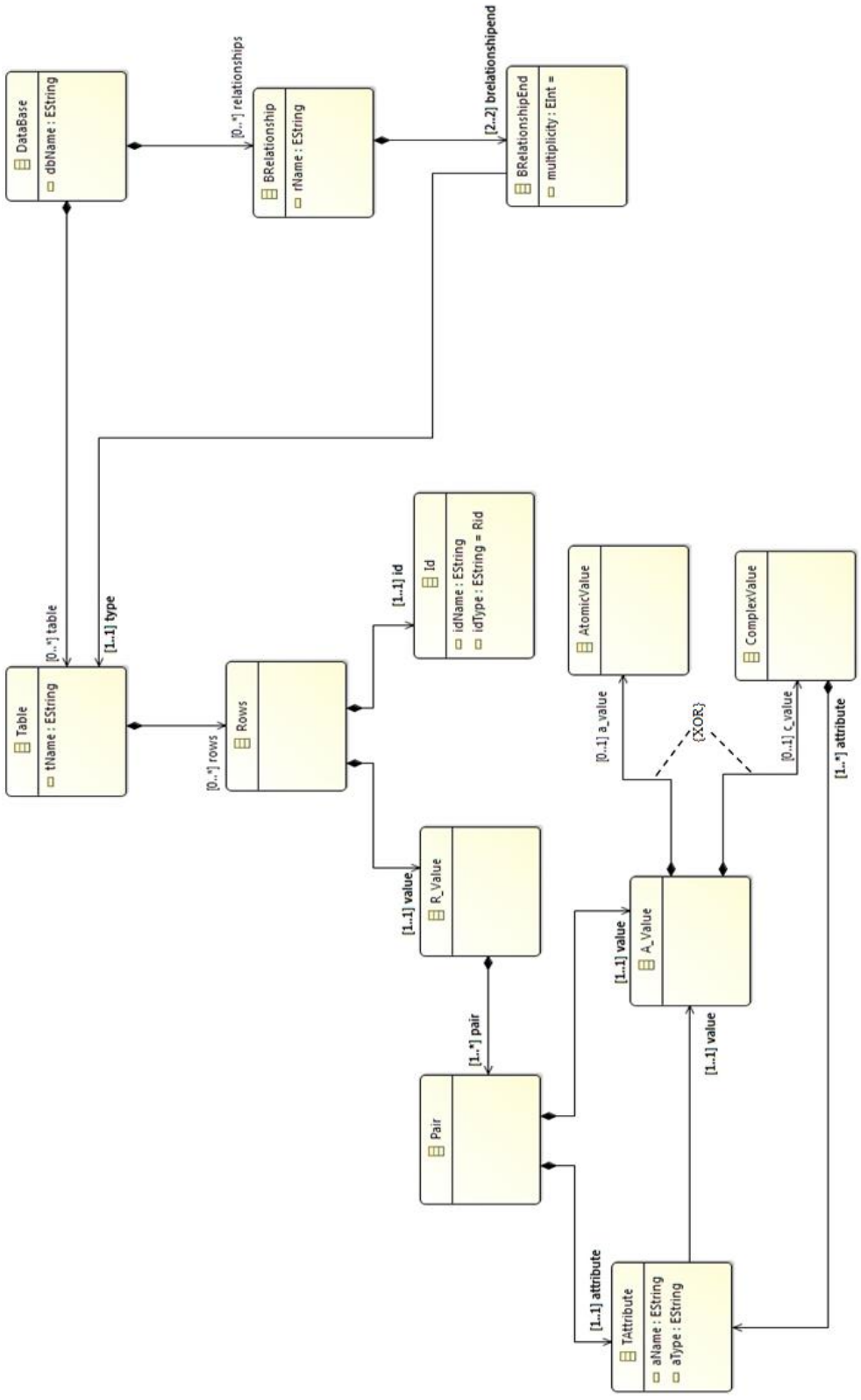


Figure 3.7 – Métamodèle du PIM logique

3.2.3 Les règles de transformation

Après avoir formalisé les concepts présents dans le modèle source (Diagramme de Classes UML) et dans le modèle cible (modèle Générique NoSQL) de la transformation UMLtoGenericModel, nous allons présenter le passage automatique du PIM conceptuel au PIM logique. Ce passage est réalisé par une chaîne de transformations que nous explicitons dans cette section.

- R1 : Chaque diagramme de classes DCL est transformé en une base de données BD, où BD.N = DCL.N.
- R2 : Chaque classe c ∈ C est transformée en une table t ∈ T, où :
  - t.N = c.N,
  - Chaque attribut de classe $a^c \in c.A^c$ est transformé en un attribut de table a^t, où $a^t.N = a^c.N$, $a^t.Ty = a^c.C$, puis ajouté à la liste des attributs de son conteneur transformé t tel que $a^t \in t.A^t$,
  - L'identificateur d'objets de c est transformé en un identificateur de lignes de t, où $Id^t.N = Id^c.N$ et $Id^t.Ty$ = Rid, puis ajouté à la liste des attributs de t tel que $Id^t \in t.A^t$.
- R3 : Chaque lien l ∈ L de degré 2 reliant deux classes c_1 et c_2 est transformé en une relation r ∈ R reliant les tables t_1 et t_2 qui correspondent aux classes c_1 et c_2 , où r.N = l.N, $r.Cp^r = \{(t_1, cr^{c_1}),(t_2, cr^{c_2})\}$.
- R4 : Chaque lien l ∈ L de degré n (avec n > 2) se traduit par (1) une nouvelle table t^l ayant son propre attribut d'identification Id^{t^l} où $t^l.N$ = l.N, $t^l.A = \{Id^{t^l}\}$ et (2) un ensemble de n relation binaires $\{r_1, \dots, r_n\}$, ∀ i ∈ [1..n] r_i relie t^l à une autre table t_i correspondant à une classe liée c_i , où $r_i.N = (t^l.N)\_(t_i.N)$ et $r_i.Cp^r = \{(t^l, nulle), (t_i, nulle)\}$.
- R5 : Chaque classe d'associations c_{asso} entre n classes $\{c_1, \dots, c_n\}$ (avec n ≥ 2) est transformée comme un lien de degré strictement supérieur à 2 en (1) une nouvelle table t^{asso} où $t^{asso}.N$ = l.N, $t^{asso}.A = c_{asso}.A^{asso}$ et (2) un ensemble de n relation binaires $\{r_1, \dots, r_n\}$, ∀ i ∈ [1…n] r_i relie t^{asso} à une autre table t_i correspondant à une classe liée c_i, où $r_i.N = (t^{asso}.N)\_(t_i.N)$ et $r_i.Cp^r = \{(t^{asso}, nulle), (t_i, nulle)\}$.

Nous avons aussi formalisé ces règles de transformation en utilisant le formalisme graphique du langage QVT (Query/View/Transformation), qui est un standard défini par l'OMG pour exprimer des transformations de modèles. Une transformation QVT entre deux modèles candidats (source et cible) est spécifiée grâce à un ensemble de relations. Chaque relation est composée des éléments suivants :

- « Domains » : chaque domaine désigne un modèle candidat et un ensemble d'éléments à relier.

- « Relation Domain » : permet de spécifier le type de relation entre les domaines, elle peut être marquée comme « Checkonly » (C) ou « Enforced » (E). Un domaine « Chekonly » permet de vérifier s'il existe une correspondance valide qui satisfait la relation ; alors qu'un domaine « Enforced » permet de créer un élément dans le modèle si le lien de correspondance n'est pas vérifié. Pour chaque domaine, le nom de son métamodèle sous-jacent doit être spécifié.

- La clause « When » : décrit les pré-conditions qui doivent être remplies pour réaliser la transformation.

- La clause « Where »: détermine les post-conditions qui doivent être remplies par tous les éléments du modèle participant à la relation.

Dans la suite, nous détaillons certaines règles de transformations appliquées dans le passage du PIM conceptuel vers le PIM logique en QVT graphique.

Relation « Main ». Cette relation est le point d'entrée du processus de transformation. La partie gauche de la figure 3.8 montre les éléments du modèle UML source (uml : UML) transformés en éléments du modèle générique NoSQL cible (gn : NoSQL) présenté par la partie droite de la figure. Le modèle UML est transformé en une base de données de même nom. La clause "where" spécifie que les classes, les liens n-aires et les classes d'associations sont transformées en tables, et que les liens binaires sont transformés en relations bianires.

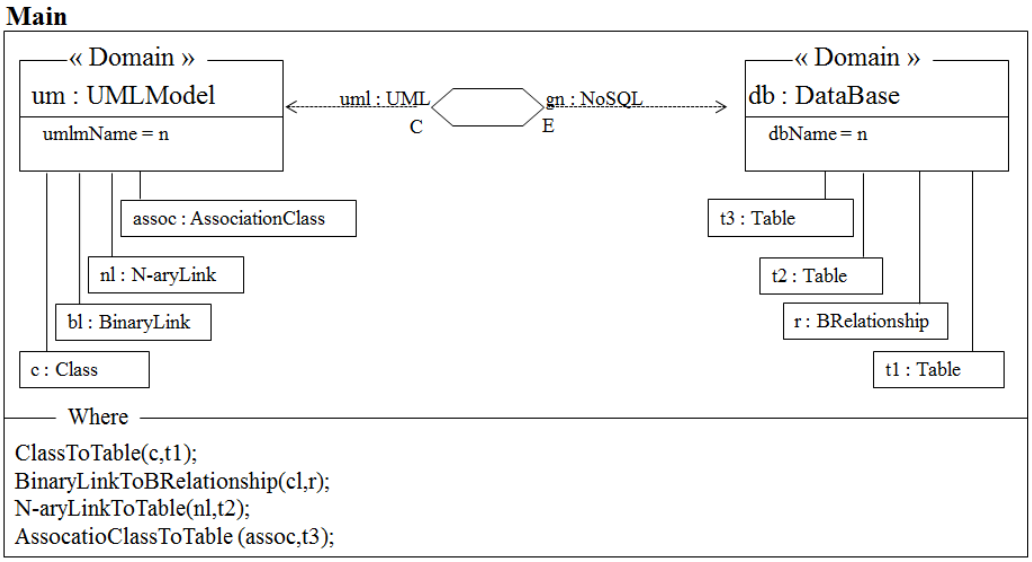


Figure 3.8 – Relation Main de la transformation du PIM conceptuel en PIM logique

Relation « ClassToTable». Une classe est transformée en une table dont le nom correspond au nom de la classe d'entrée. Tous les attributs de la classe sont transformés en attributs de la table, ceci en appliquant la relation « CAttributeToTAttribute » indiquée dans la clause "where" de la figure 3.9. Les liens binaires et n-aires auxquels participe la classe, sont transformés respectivement en relations binaires et tables.

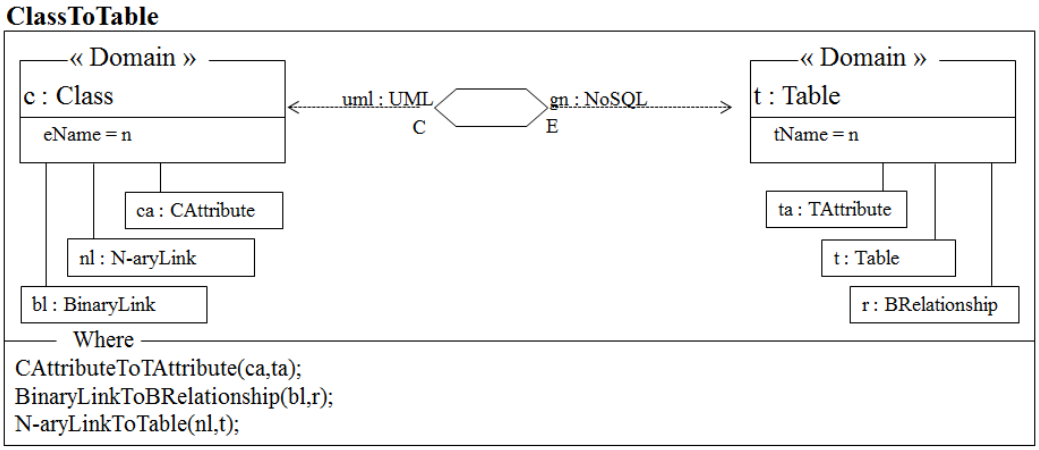


Figure 3.9 – Relation de transformation de classes en tables

Relation « N-aryLinkToTable ». Une fois les classes transformées en tables (la pré-condition « ClassToTable » de la clause « When »), le lien est converti en :

- Une nouvelle table portant le nom du lien (cf. figure 3.10)
- Un ensemble de relations binaires qui relient cette table aux autres tables liées (relation "ClassToBRelationship" de la clause "Where").

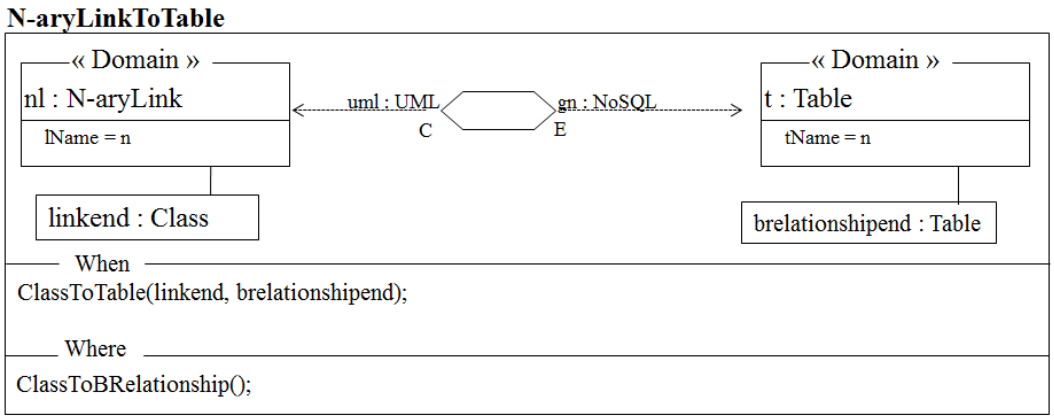


Figure 3.10 – Relation de transformation de liens n-aires en tables

3.3 La transformation GenericModel2PhysicalModel

Selon les principes de l'approche MDA, un PIM logique permet de générer plusieurs PSM associés à des plateformes distinctes. Ce principe assure l'indépendance du niveau logique face aux spécificités techniques des SGBD NoSQL ainsi qu'à leurs évolutions. Dans cette section, nous présentons tout d'abord les plateformes NoSQL d'implantation que nous avons choisies pour illustrer notre travail ainsi que leurs modèles de données. Ensuite, nous décrivons le passage du modèle générique vers le modèle physique associé à chacun de ces systèmes.

3.3.1 La source : PIM logique

L'entrée de la transformation GenericModel2PhysicalModel correspond à la sortie de la transformation Object2GenericModel présentée dans la section précédente. L'entrée de ce processus de transformation est donc un modèle générique NoSQL compatible avec les quatre types de SGBD NoSQL : colonnes, documents, graphes et clé/valeur ; ce modèle se situe au niveau logique de notre démarche illustrée dans la figure 3.1.

3.3.2 La cible : PSM

La sortie de la transformation GenericModel2PhysicalModel correspond à des modèles physiques NoSQL. Pour illustrer notre proposition, nous avons choisi d'implanter notre modèle générique sur un représentant de chaque type de SGBD NoSQL: Cassandra (orienté colonnes), MongoDB (orienté documents), Neo4j (orienté graphes) et Redis (orienté clé/valeur). Dans cette section, nous définissons les différents concepts du modèle de données associé à chacun de ces systèmes.

3.3.2.1 Modèle Cassandra

Apache Cassandra est un SGBD NoSQL distribué de type orienté-colonnes, initialement basé sur le modèle BigTable de Google ; il emprunte également des caractéristiques au système Dynamo d'Amazon. Le modèle de données de Cassandra a été conçu pour permettre une flexibilité maximale et des temps de réponse rapides sur de gros volumes. Ce modèle se base sur les concepts de keyspace, de famille de colonnes et de colonne.

Un keyspace est un conteneur de haut niveau qui contient tous les autres éléments du modèle. A l'intérieur d'un keyspace se trouvent une ou plusieurs familles de colonnes ; chacune d'elles est un regroupement de lignes identifiées par une clé (primary key) et définies par des colonnes. Les lignes à l'intérieur d'une famille de colonnes n'ont pas nécessairement les mêmes schémas (ne sont pas forcément définies par les mêmes colonnes).

Formellement :

Définition 8. Un keyspace KS est défini par (N, F) où:

- N est le nom du keyspace,
- F = $\{f_1, \dots, f_n\}$ est un ensemble de familles de colonnes.

Définition 9. ∀ i ∈ [1..n], le schéma d'une famille de colonnes $f_i \in$ F est un couple (N, CL^f) où:

- f_i.N est le nom désignant la famille de colonnes,
- $f_i.CL^f = \{cl_1^f, \dots, cl_q^f\} \cup \{PrimaryKey^f\}$ est l'ensemble de colonnes qui seront utilisés pour définir les lignes de f, où :
  - ∀ i ∈ [1..q], le schéma d'une colonne $cl_i^f \in CL^f$ est un couple (N, Ty) où:
    - cl_i^f. N est le nom désignant la colonne,
    - cl_i^f.Ty est le type de la colonne.
  - $PrimaryKey^f$ est une colonne spécifique de f jouant le rôle d'un identificateur de lignes. Comme toute colonne, $PrimaryKey^f$ possède un nom $PrimaryKey^f$.N et un type (standard).

3.3.2.2 Modèle MongoDB

MongoDB est un SGBD NoSQL de type orienté documents conçu pour offrir de hautes performances en lecture et en écriture ainsi qu'une mise à l'échelle automatique de la base de données. Ce système stocke les données en format BSON (Binary JSON) qui est une représentation textuelle de données en clé/valeur. La clé est un identificateur unique associé à un agrégat de champs que l'on nomme document.

Ainsi, une ligne dans une base de données MongoDB est un document en format BSON. Un document est toujours identifié par une clé, noté _id, et consiste en un ensemble de champs composés d'un nom et d'une valeur. La valeur d'un champ peut être atomique ou complexe (i.e. qu'elle inclut d'autres documents). Les documents sont regroupés dans des collections ; chacune d'elles peut contenir des documents avec des structures différentes (i.e. les documents d'une collection ne possèdent pas forcément les mêmes champs). Les collections sont contenues dans une base de données que nous appelons par la suite une base de données mongodb (BD^{md}) pour faire la distinction avec une base de données du niveau logique (BD).

Formellement :

Définition 10. Une base de données mongodb BD^{md} est définie par (N, CLL) où :

- N est le nom de la base,
- CLL = $\{cll_1, \dots, cll_n\}$ est un ensemble de collections.

Définition 11. ∀ i ∈ [1..n], le schéma d'une collection cll_i ∈ CLL est un couple (N, FL) où:

- cll_i.N est le nom identifiant la collection,
- cll_i.FL = $FL^a \cup FL^{cx} \cup \{Id^{cll}\}$ est l'ensemble de champs atomiques $Fl^a = \{fl_1^a, \dots, fl_r^a\}$ et complexes $Fl^{cx} = \{fl_1^{cx}, \dots, fl_s^{cx}\}$ qui seront utilisés pour définir les documents de cll_i, où :
  - ∀ i ∈ [1..r], le schéma d'un champ atomique $fl_i^a \in FL^a$ est un couple (N, Ty) où:
    - fl_i^a.N est le nom identifiant le champ,
    - fl_i^a.Ty est le type de champ.
  - ∀ j ∈ [1..s], le schéma d'un champ complexe $fl_j^{cx} \in FL^{cx}$ est un couple (N, FL') où:
    - fl_j^{cx}.N est le nom identifiant le champ,
    - fl_j^{cx}.FL' est l'ensemble de champs imbriqués dans fl_j^{cx} où FL'⊂ FL.
  - Id^{cll} est un champ spécial qui doit exister dans chaque document de cll_i afin de l'identifier d'une manière unique dans la collection. Ce champ porte un nom fixe noté _id.

3.3.2.3 Modèle Neo4j

Neo4j est un SGBD NoSQL de type orienté graphes qui fournit une grande souplesse pour représenter les liens entre les données. Nous rappelons que ce type de systèmes améliore les performances des applications dans les scénarios qui nécessitent le stockage et l'interrogation des données fortement liées (cf. chapitre 1).

Neo4j représente les données sous forme d'un graphe permettant de décrire un ensemble d'objets et leurs relations binaires. Les objets sont appelés les nœuds et une relation entre deux objets est appelé un arc (« Vertex » et « Edge» en anglais). Une ligne dans une base de données Neo4j correspond à un nœud qui possède un nom, appelé « label », et un ensemble de propriétés de type clé/valeur (la clé indiquant le nom d'une propriété auquel correspond une valeur). Un nœud peut avoir plusieurs arcs qui pointent sur d'autres nœuds. Tout comme un nœud, les arcs possèdent également un nom et des propriétés ; de plus, chacun d'eux est composé de deux extrémités qui correspondent au nœud de départ et au nœud d'arrivée.

Formellement :

Définition 12. Un graphe est défini par (V, E) où :

- V = $\{v_1, \dots, v_n\}$ est un ensemble de nœuds,
- E = $\{e_1, \dots, e_m\}$ est un ensemble d'arcs.

Définition 13. ∀ i ∈ [1..n], le schéma d'un nœud v_i ∈ V est un couple (L, PR^v) où:

- v_i.L est le label (ou nom) du nœud,
- $v_i.PR^v = \{pr_1^v, \dots, pr_q^v\} \cup \{Id^v\}$ est un ensemble de propriétés, où :
  - ∀ k ∈ [1..q], le schéma d'une propriété $pr_k^v \in PR^v$ est un couple (N, Ty) où:
    - pr_k^v .N est le nom identifiant la propriété,
    - pr_k^v.Ty est le type de la propriété.
  - Id^v est une propriété spéciale de v_i que l'utilisateur peut définir en lui associant la contrainte « Is Unique » afin d'identifier le nœud dans le graphe. En effet, dans un graphe Neo4j, deux ou plusieurs nœuds peuvent avoir le même label. Ainsi, celui-ci ne peut pas jouer le rôle d'un identificateur.

Définition 14. ∀ j ∈ [1..m], le schéma d'un arc e_j ∈ E est défini par un triplet (L, PR^e, $(extrem_1, extrem_2)$) où:

- e_i.L est le label de l'arc,
- $e_i.PR^e = \{pr_1^e, \dots, pr_p^e\}$ est un ensemble de propriétés, où :
  - ∀ l ∈ [1..p], le schéma d'une propriété $pr_l^e \in PR^e$ est un couple (N, Ty) où:
    - pr_l^e .N est le nom identifiant la propriété,
    - pr_l^e.Ty est le type de la propriété.
- $e_i.(extrem_1, extrem_2)$ est une paire de nœuds. Les nœuds $extrem_1$ et $extrem_2$ sont les extrémités de l'arc e_i.

3.3.2.4 Modèle Redis

Redis est un SGBD NoSQL de type clé-valeur, fréquemment utilisé dans les applications où le schéma évolue continuellement. Le principe général de Redis est de déposer, dans un espace de stockage, des valeurs associées à des clés. Les clés sont uniques. La valeur peut être atomique (une chaîne de caractères, un entier, une date, etc.) ou complexe (composée d'un ensemble de couples (clé : valeur)).

Redis n'exige pas de déclarer un modèle avant l'alimentation des données ; l'organisation de l'espace de stockage suit la Définition 15 :

Définition 15. Un espace de stockage ES est défini par un ensemble de couples CP = $CP^a \cup CP^{cx}$, où :

- $CP^a = \{cp_1^a, ..., cp_n^a\}$ est un ensemble de couples dont la valeur est atomique. ∀ i ∈ [1..n], un couple $cp_i^a \in CP^a$ est de la forme (Key :Value) où cp_i^a.Value est une valeur appartenant à un type atomique généralement prédéfini (Chaîne de caractères, Entier, Date,…),
- $CP^{cx} = \{cp_1^{cx}, ..., cp_m^{cx}\}$ est un ensemble de couples dont la valeur est complexe. ∀ j ∈ [1..m], un couple $cp_j^{cx} \in CP^{cx}$ est aussi de la forme (Key :Value) où cp_j^{cx}.Value = $\{cp_1^a, ..., cp_q^a\}$ est un ensemble de couples dont la valeur est atomique.

Nous décrivons le modèle de données utilisé par chaque système considéré à travers un métamodèle. Les figures 3.11, 3.12, 3.13 et 3.14 montrent respectivement les métamodèles des PSM Cassandra, MongoDB, Neo4j et Redis.

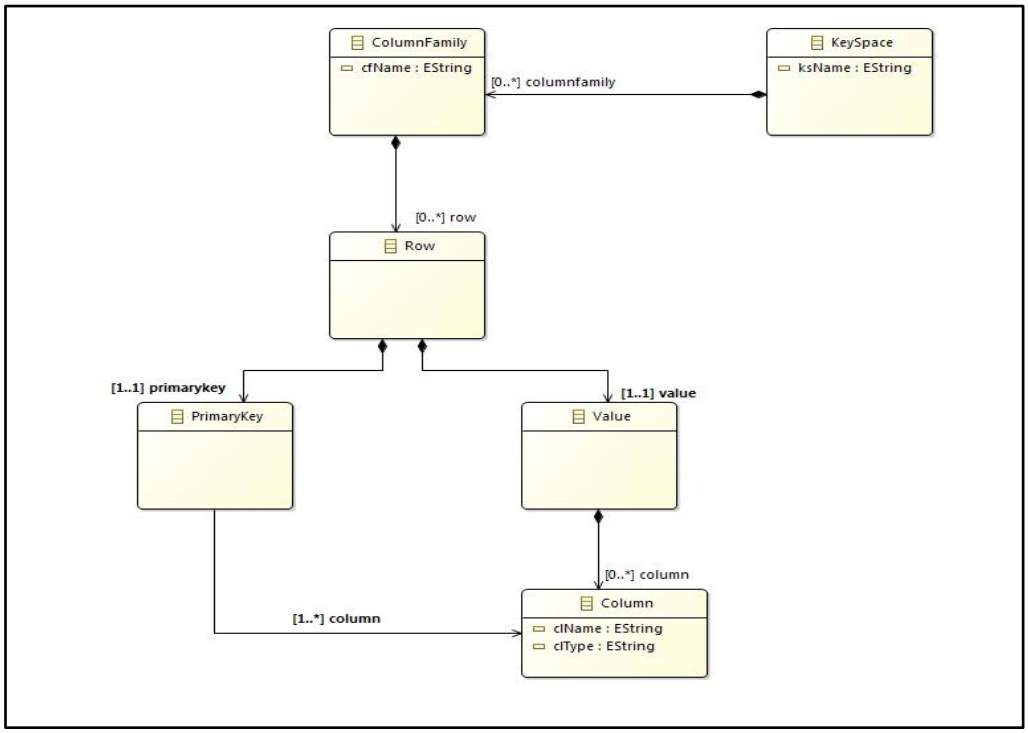


Figure 3.11 – Métamodèle du PSM Cassandra

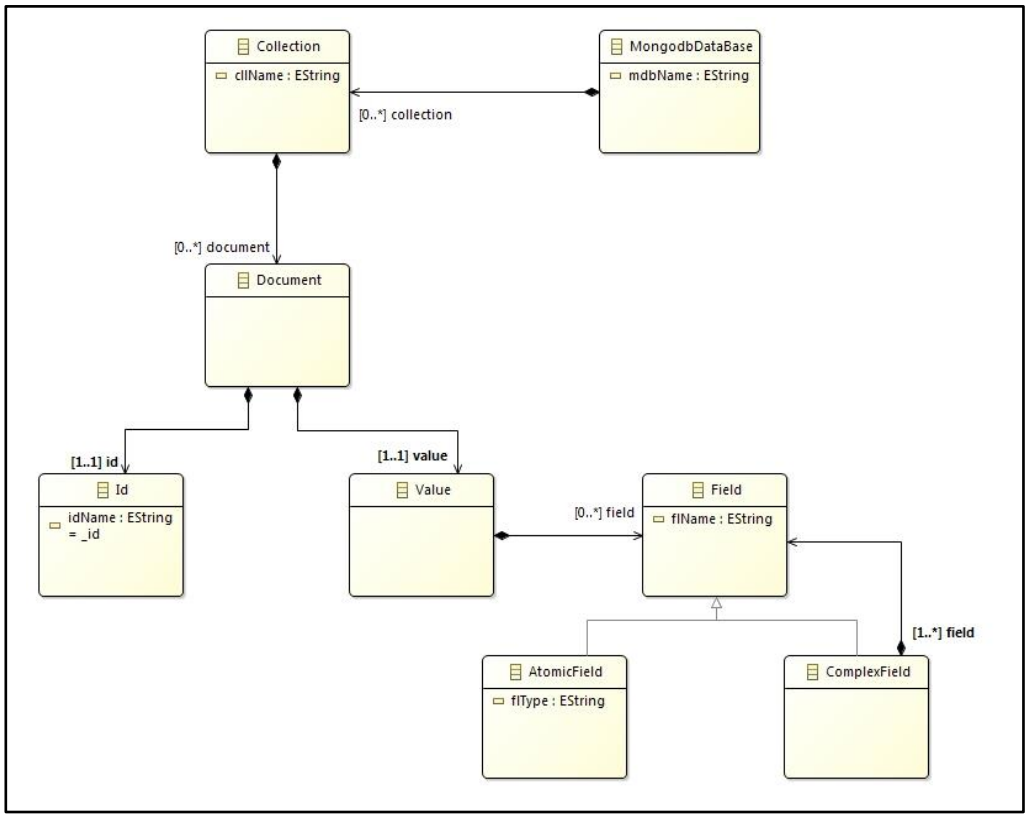


Figure 3.12 – Métamodèle du PSM MongoDB

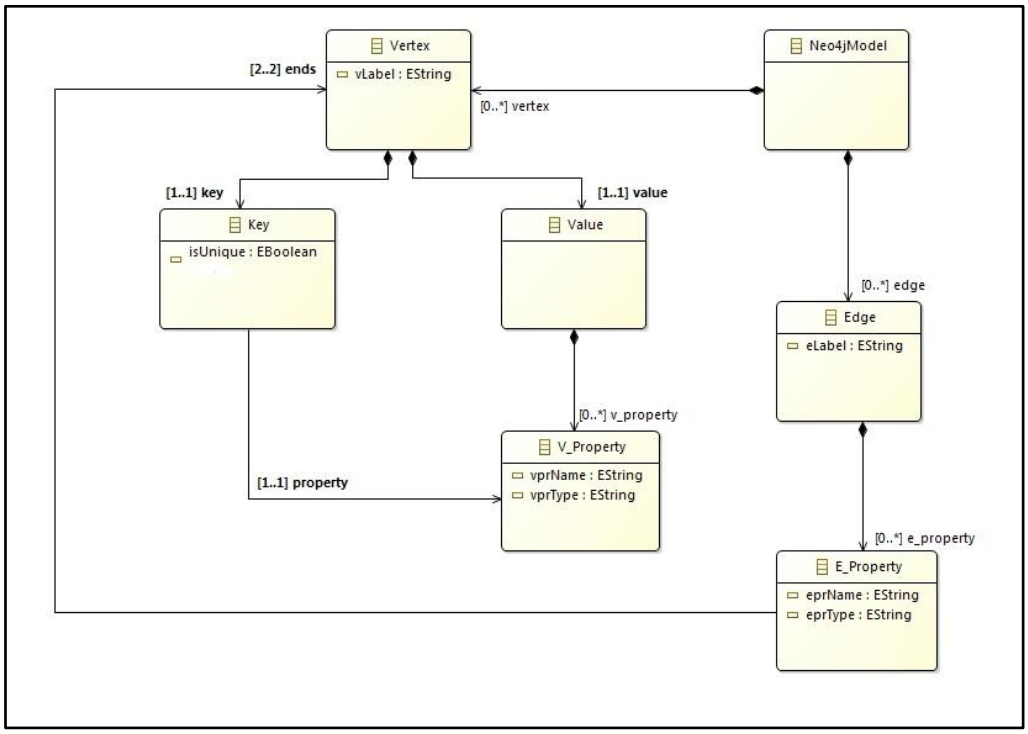


Figure 3.13 – Métamodèle du PSM Neo4j

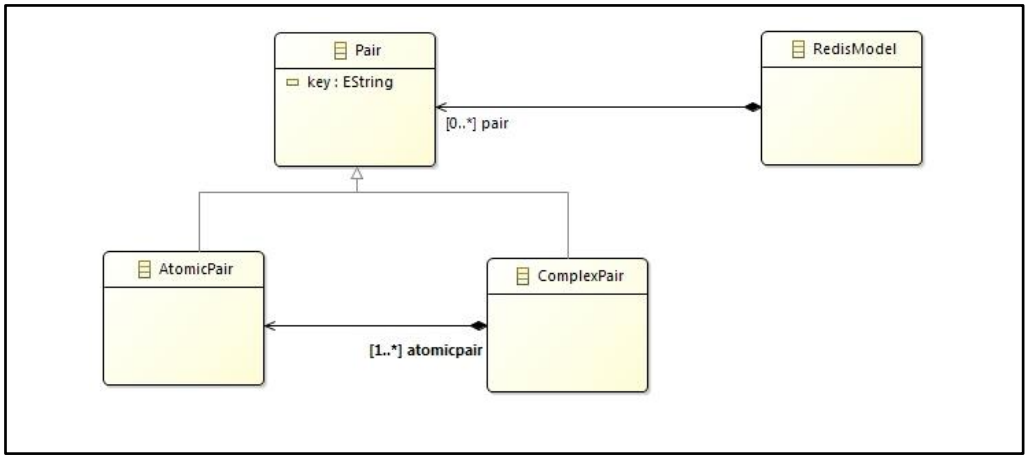


Figure 3.14 – Métamodèle du PSM Redis

3.3.3 Les règles de transformation

Nous présentons dans cette section le passage automatique du PIM logique vers chaque PSM décrit dans la section précédente. Ce passage est réalisé par l'application d'une série de transformations que nous explicitons par la suite.

Pour chaque système, deux types de transformations peuvent être considérés : (1) les transformations qui génèrent les éléments nécessaires à l'implantation de la base de données sur le SGBD NoSQL ; ce sont les éléments qui doivent être déclarés préalablement avant de commencer l'alimentation, et (2) les transformations qui créent les directives d'assistance utiles pour implanter les traitements ; ces directives donnent les modalités d'utilisation des attributs et d'implantation des relations (voir Section 3.1).

3.3.3.1 Vers le PSM Cassandra

Comme dans les systèmes relationnels, le modèle de données sous Cassandra doit être fixé préalablement. Il en est ainsi du nom de la BD, des noms de familles de colonnes et des colonnes. L'implantation des relations est précisée dans les directives.

a. Les éléments nécessaires à l'implantation de la BD

- R1 : Chaque base de données BD est transformée en un keyspace KS, où : KS.N = BD.N.
- R2 : Chaque table t ∈ BD est transformée en une famille de colonne f ∈ F, où :
    - f.N = t.N,

- Chaque attribut $a^t \in t$. A^t est transformé en une colonne cl, où cl.N = a^t.N, cl.Ty = a^t.Ty, puis ajouté à la liste des colonnes de son conteneur transformé f tel que cl ∈ f.CL.

- L'identificateur de lignes de t est transformé en un identificateur de lignes de f, où $PrimaryKey^f$.N = Id^t.N, puis ajouté à la liste des colonnes de f tel que $PrimaryKey^f$ ∈ f.CL.

b. Les directives d'assistance

- R3 : Généralement, une relation entre tables peut être implantée de trois manières : (1) l'ajout de références dans une des tables liées, (2) l'imbrication des données ou (3) la création d'une nouvelle table contenant des références. Comme Cassandra ne supporte pas l'imbrication, uniquement les solutions (1) et (3) peuvent être utilisées pour exprimer des relations entre les familles de colonnes. Ces deux solutions portent sur l'utilisation des colonnes de référence. Une colonne de référence est une colonne qui représente la valeur de la clé d'une famille de colonnes connexe. Autrement dit, les valeurs d'une colonne de référence doivent exister dans la colonne PrimaryKey d'une famille de colonnes liée ; notons que cette contrainte (qui s'apparente à la contrainte d'intégrité référentielle des systèmes relationnels) n'est pas gérée automatiquement par le système Cassandra ; sa vérification reste donc à la charge de l'utilisateur. Ainsi, nous définissions la règle 3 qui transforme les relations du niveau logique comme suit : pour chaque relation r reliant deux tables t_1 et t_2, trois solutions de transformation peuvent être considérées :

  Solution 1: r est transformée en une colonne cl référençant f_2 (la famille de colonnes correspondant à t_2), où cl.N = (f_2 .N)_Ref et cl.Ty = PrimaryKey^{f2}.Ty, et puis ajoutée à la liste des colonne de f_1 (la famille de colonnes correspondant à t_1) tel que cl ∈ f_1.CL. Lors de l'instanciation de f_1, la colonne de référence cl prendra une ou plusieurs valeurs (selon les cardinalités de r ; cf. tableau 3.1) de la PrimaryKey de f_2.

  Solution 2: r est transformée en une colonne cl référençant f_1 (la famille de colonnes correspondant à t_1), où cl.N = (f_1 .N)_Ref et cl.Ty = PrimaryKey^{f1}.Ty, et puis ajoutée à la liste des colonne de f_2 (la famille de colonnes correspondant à t_2) tel que cl ∈ f_2.CL. Lors de l'instanciation de f_2, la colonne de référence cl prendra une ou plusieurs valeurs (selon les cardinalités de r ; cf. tableau 3.1) de la PrimaryKey de f_1.

  Solution 3: r est transformée en une nouvelle famille de colonnes f composée de deux colonnes cl_1 et cl_2 référençant respectivement f_1 et f_2 (les familles de colonnes correspondantes aux tables liées $t_1$et t_2), où f.N = r.N, f.CL = { cl_1, cl_2 }, cl_1 .N = (f_1 .N)_Ref, cl_1 .Ty = PrimaryKey^{f1} .Ty, cl_2 .N = (f_2.N)_Ref et cl_2.Ty = PrimaryKey^{f2}.Ty. Lors de l'instanciation de f, les colonnes de référence cl_1 et cl_2 prendront respectivement une valeur de la PrimaryKey de f_1 et de f_2 .

Selon les cardinalités de r, les colonnes de référence utilisées dans chacune de ces solutions peuvent être monovaluées ou multivaluées. Le tableau 3.1 indique le type de la colonne de référence en fonction des cardinalités de la relation et de la solution de transformation choisie.

Relation	Solution	Type de la colonne de référence
r = (N,{(t_1,*),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Multivalued
	Solution 3	Monovalued
r = (N,{(t_1,1),(t_2,*)})	Solution 1	Multivalued
	Solution 2	Monovalued
	Solution 3	Monovalued
r = (N,{(t_1,1),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Monovalued
	Solution 3	Monovalued
r = (N,{(t_1,*),(t_2,*)})	Solution 1	Multivalued
	Solution 2	Multivalued
	Solution 3	Monovalued

Tableau 3.1. Types de colonnes de référence

3.3.3.2 Vers le PSM MongoDB

Sous MongoDB, seulement une partie du modèle est déclaré avant la saisie des données. Il s'agit de préciser le nom de la base de données et les noms des collections. Les noms des champs ainsi que de la manière d'implanter les relations sont spécifiés dans les directives d'assistance.

a. Les éléments nécessaires à l'implantation de la BD

- R1 : Chaque base de données BD est transformée en une base de données mongodb BD^{MD}, où BD^{MD}.N = BD.N.
- R2 : Chaque table t ∈ BD est transformée en une collection cll ∈ BD^{MD}, où cll.N = t.N.

b. Les directives d'assistance

- R3 : Chaque attribut de table $a^t \in t$. A^t est transformé en un champ atomique fl^a, où fl^a.N = a^t.N, fl^a.Ty = a^t.Ty, puis ajouté à la liste des champs de son conteneur transformé cll tel que $fl^a \in$ cll. FL^a, où cll est la collection qui correspond à la table t.

- R4 : L'identificateur de lignes d'une table t est transformé en un identificateur de documents de la collection cll correspondant à t, où Id^{cll}.N = Id^t.N, puis ajouté à la liste des champs de cll tel que $Id^{cll} \in$ cll.FL.

- R5 : Le système MongoDB permet d'exprimer des relations entre des objets liés en utilisant des champs de référence ou en imbriquant les documents. Un champ de référence est un champ représentant l'identifiant (_id) d'un document, inséré dans un autre document afin d'assurer la relation entre les deux documents. Autrement dit, les valeurs d'un champ de référence doivent exister dans le champ _id du document référencé. Tout comme Cassandra, cette contrainte n'est pas gérée automatiquement par le système MongoDB ; sa vérification reste à la charge de l'utilisateur.

  Notons qu'avec le système MongoDB, les relations peuvent exister entre les documents de la même collection ou entre les documents de différentes collections. Dans notre cas, toutes les relations que nous avons à implanter existent entre des collections différentes. En effet, une collection dans une base de données MongoDB peut contenir des documents représentant des objets de types différents. Par exemple, nous pouvons stocker dans la même collection des documents représentant des patients, des médecins, des consultations. Dans notre scénario et par rapport à l'application médicale qui est à la base de notre travail, une collection (niveau physique) correspond à une classe (niveau conceptuel), elle contient donc des documents ayant la même sémantique. Ainsi, nous n'avons que des relations entre des collections distinctes.

  Nous définissions la règle 3 qui transforme les relations du niveau logique comme suit : pour chaque relation r reliant deux tables t_1 et t_2, cinq solutions de transformation peuvent être considérées :

  Solution 1: r est transformée en un champ fl référençant un document dans cll_2 (la collection correspondant à t_2), où fl.N = (cll_2.N)_Ref et fl.Ty = Id^{cll_2}.Ty, et puis ajoutée à la liste des champs de cll_1 (la collection correspondant à t_1) tel que fl $\in$ cll_1.FL. Lors de l'instanciation de cll_1, le champ de référence fl prendra une ou plusieurs valeurs (selon les cardinalités de r ; cf. tableau 3.2) de l'identifiant _id d'un document existant dans cll_2.

  Solution 2: r est transformée en un champ fl référençant un document dans cll_1 (la collection correspondant à t_1), où fl.N = (cll_1.N)_Ref et fl.Ty = Id^{cll_1}.Ty, et puis ajoutée à la liste des champs de cll_2 (la collection correspondant à t_2) tel que fl $\in$ cll_2.FL. Lors de l'instanciation de cll_2, le

champ de référence fl prendra une ou plusieurs valeurs (selon les cardinalités de r ; cf. tableau 3.2) de l'identifiant _id d'un document existant dans cll_1.

Solution 3: r se traduit par l'imbrication d'un document d dans cll_2 (la collection correspondant à t_2) dans cll_1 (la collection correspondant à t_1), où d ∈ $cll_1.FL^{cx}$.

Solution 4: r se traduit par l'imbrication d'un document d dans cll_1 (la collection correspondant à t_1) dans cll_2 (la collection correspondant à t_2), où d ∈ $cll_2.FL^{cx}$.

Solution 5: r est transformée en une nouvelle collection cll, cll.N = r.N, cll.Fl = $\{fl_1, fl_2\}$, fl_1.N = (cll_1.N)_Ref, fl_1.Ty = Id^{cll_2}.Ty, fl_2.N = (cll_2.N)_Ref et fl_2.Ty = Id^{cll_2}.Ty.cll_1 et cll_2 étant les collections correspondantes aux tables liées $t_1$et t_2.

Relation	Solution	Type du champ de référence
r = (N,{(t_1,*),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Multivalued
	Solution 5	Monovalued
r = (N,{(t_1,1),(t_2,*)})	Solution 1	Multivalued
	Solution 2	Monovalued
	Solution 3	Monovalued
r = (N,{(t_1,1),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Monovalued
	Solution 5	Monovalued
r = (N,{(t_1,*),(t_2,*)})	Solution 1	Multivalued
	Solution 2	Multivalued
	Solution 5	Monovalued

Tableau 3.2. Types de champs de référence

Selon les cardinalités de r, les champs de référence utilisés dans les solutions 1, 2 et 5 peuvent être monovalués ou multivalués. Le tableau 3.2 indique le type du champ de référence en fonction des cardinalités de la relation et de la solution de transformation choisie.

3.3.3.3 Vers le PSM Neo4j

Le modèle des données sous le système Neo4j est spécifié au fur et à mesure de la saisie des données. Le nom du nœud ainsi que les noms des propriétés le constituant sont précisés par l'utilisateur au moment de l'insertion de chaque nœud. Ainsi, aucun élément ne doit être fixé avant de commencer l'alimentation. Pour Neo4J, seules les directives d'assistance sont générées.

a. Les directives d'assistance

- R1 : Chaque table t ∈ BD est transformée en un nœud v ∈ V, où :
  - v.L = t.N,
  - Chaque attribut $a^t \in t$. A^t est transformé en une propriété pr^v, où pr^v.N = a^t.N, pr^v.Ty = a^t.Ty, puis ajouté à la liste des propriétés de son conteneur transformé v tel que $pr^v \in v.\ PR^v$,
  - L'identificateur de lignes de t est transformé en un identificateur du nœud v, où Id^v.N = Id^t.N, puis ajouté à la liste des propriétés de v tel que $Id^v \in v.\ PR^v$. De plus, nous associons à Id^v la contrainte « Is Unique ».
- R2 : Chaque relation r reliant deux tables t_1 et t_2 est transformée en un arc e reliant les deux nœuds v_1 et v_2 correspondant aux tables liées, où :
  - e.L = r.N,
  - e.$(extrem_1, extrem_2) = (v_1, v_2)$.

3.3.3.4 Vers le PSM Redis

Sous Redis, aucun modèle ne doit être fixé préalablement avant de commencer l'alimentation. Autrement dit, la saisie des données se fait sans contrôle préalable par rapport à un modèle physique. Ce dernier est spécifié au fur et à mesure de la saisie des données ; l'utilisateur insère chaque couple (clé : valeur) en précisant sa clé ainsi que les clés des couples le constituant s'il s'agit d'un couple complexe.

Ainsi, notre processus de transformation ne génère pas un modèle physique spécifique au système Redis, mais un ensemble de directives qui donne, à l'utilisateur, les modalités d'utilisation des clés et d'implantation des relations.

a. Les directives d'assistance

- R1 : Sous Redis, la notion de table n'est pas explicite. Ainsi, nous imposons la règle selon laquelle tous les couples de 1er niveau dans l'espace de stockage qui présentent une clé de la forme XXXX_i, avec i un indice numérique quelconque, seront considérés comme appartenant à la table XXXX. Dans le

processus de transformation et selon ce principe, chaque couple cp^{cx} dans l'espace de stockage Redis va correspondre à une ligne d'une table t qui existe au niveau logique, où :

- cp^{cx}.Key = [t.N]_i ; i est un indice faisant référence à une ligne de t (Par exemple : Patient_1, Medecin_5 …),
- Chaque attribut $a^t \in t$. A^t est transformé en un couple cp^a, où cp^a.Key = a^t.N, puis ajouté à la liste de couples de son conteneur transformé cp^{cx} tel que $cp^a \in cp^{cx}$.Value,

- R2 : La création des relations sous Redis s'effectue par des couples de référence. Un couple de référence est un couple dont la valeur peut être atomique (elle correspond à la clé du couple référencé) ou complexe (composée d'autres couples de référence atomiques). Ainsi, pour chaque relation r qui existe au niveau logique entre deux t_1 et t_2, trois solutions peuvent être considérées :

Solution 1: r est transformée en un couple cp^a_{ref} référençant cp^{cx}_2 (le couple correspondant à une ligne de t_2), où cp^a_{ref}.Key = [t_2.N]_Ref et cp^a_{ref}.Value = cp^{cx}_2.Key, et puis ajoutée à la liste de couples de cp^{cx}_1 (le couple correspondant à une ligne de t_1) tel que $cp^a_{ref} \in cp^{cx}_1$.Value.

Solution 2: r est transformée en un couple cp^a_{ref} référençant cp^{cx}_1 (le couple correspondant à une ligne de t_1), où cp^a_{ref}.Key = [t_1.N]_Ref et cp^a_{ref}.Value = cp^{cx}_1.Key, et puis ajoutée à la liste de couples de cp^{cx}_2 (le couple correspondant à une ligne de t_2) tel que $cp^a_{ref} \in cp^{cx}_2$.Value.

Solution 3: r est transformée en un couple cp^{cx}_{ref} composé de deux couples de référence cp^a_{ref1} et cp^a_{ref2} référençant respectivement cp^{cx}_1 et cp^{cx}_2 , où cp^{cx}_{ref}.Key = r.N, cp^{cx}_{ref}.Value = $\{cp^a_{ref1}, cp^a_{ref2}\}$, cp^a_{ref1}.Key = (t_1.N)_Ref, cp^a_{ref1}.Value = cp^{cx}_1.Key, cp^a_{ref2}.Key = (t_2.N)_Ref, cp^a_{ref2}.Value = cp^{cx}_2.Key.

3.3.4 Choix d'implantation des liens

Pour certains SGBD NoSQL (Cassandra, MongoDB notamment), plusieurs solutions permettent d'implanter les relations entre les tables du niveau logique ; par exemple la solution des références ou celle des imbrications. Le choix d'une solution, qui est effectuée selon des critères propres à chaque SGBD, n'est pas sans conséquence puisqu'elle influe notamment sur les performances des traitements. Des travaux [Gomez et al., 2018] ont permis de définir des critères et des métriques pour effectuer le choix le plus pertinent au regard des exigences de l'application et des possibilités du SGBD. Ces travaux sont pour l'instant partiels puisqu'ils concernent uniquement le système MongDB avec le format de données JSON. Nous avons donc différé l'intégration de ces techniques dans notre processus et nous avons considéré qu'elles constituaient des perspectives à nos travaux (voir Conclusion Générale, section Perspectives).

Pour l'utilisateur, notre processus de transformation automatisé se déroule selon l'un des deux modes suivant :

1- Le SGBD choisi par l'utilisateur offre une technique d'implantation unique des liens (cas du système Neo4J par exemple) : le processus est alors automatique et génère directement un schéma physique.
2- Le SGBD choisi propose plusieurs solutions pour implanter les liens (cas du système MongoDB par exemple) : le processus est semi-automatique et propose à l'utilisateur les différentes solutions d'implantation des liens au travers d'une interface de dialogue. Pour chaque lien, l'utilisateur choisit l'implantation la plus adaptée aux exigences de l'application.

Dans le mode n° 2, on constate que le recours à un choix technique exige des connaissances informatiques. Par conséquent, dans le cas où un décideur met en œuvre notre processus de transformation de modèles, le mode n° 1 doit être privilégié ; si le mode n° 2 est retenu, un informaticien fournit une assistance ponctuelle au décideur pour choisir les solutions d'implantation des liens.

3.4 Conclusion

Synthèse

Les applications Big Data utilisent les SGBD NoSQL pour stocker des BD. Ceci nécessite généralement la transformation d'un modèle conceptuel décrivant une BD massive en un modèle physique NoSQL. Cette tâche manuelle s'avère fastidieuse en raison de la spécificité des modèles NoSQL. Dans ce chapitre, nous avons proposé le processus Object2NoSQL qui automatise en grande partie la transformation des modèles. Ce processus est basé sur MDA qui offre un cadre formel aux mécanismes de transformation des modèles. Il est destiné à un utilisateur qui est chargé d'implanter une BD massive sur un SGBD NoSQL.

Object2NoSQL consiste en une chaîne de transformations qui utilise un modèle générique situé à un niveau intermédiaire entre un DCL (niveau conceptuel) et un modèle d'implantation NoSQL (niveau physique). Ce modèle générique est compatible avec les quatre types de SGBD NoSQL : colonnes, documents, graphes et clé-valeur. Il fait apparaître principalement des tables et des relations binaires entre ces tables. Pour prendre en compte une évolution technologique du système dans le modèle physique, l'utilisateur remontera uniquement au niveau logique sur lequel il appliquera de nouvelles transformations, sans se soucier du modèle conceptuel initial. Ainsi, lors d'une évolution technologique ou d'un changement du système, notre processus de transformation est raccourci et donc simplifié en procurant un gain de temps sensible à l'utilisateur.

Les modèles de données (conceptuel, logique et physique) utilisés par notre processus sont conformes à des métamodèles que nous avons proposés pour réaliser les passages : conceptuel vers logique puis logique vers physique. Aussi,

nous avons automatisé chaque passage en utilisant des transformations de type M2M formalisées avec le standard QVT.

Par ailleurs, nous avons proposé plusieurs solutions pour implanter les relations binaires dans les modèles physiques selon les possibilités des SGBD (références mono ou multivaluées, imbrications). Le choix de la solution la plus adaptée selon des critères liés aux performances des traitements a été étudié par ailleurs [Gomez et al., 2018].

Les travaux de ce chapitre ont été présentés dans les publications suivantes : [Abdelhedi et al., 2016a], [Abdelhedi et al., 2016b], [Abdelhedi et al., 2016c], [Abdelhedi et al., 2017a], [Abdelhedi et al., 2017b], [Abdelhedi et al., 2017c], [Abdelhedi et al., 2018a] et [Abdelhedi et al., 2018b].

Positionnement

L'étude de l'état de l'art montre que les travaux existants relatifs à la transformation d'un modèle conceptuel UML en un modèle physique NoSQL se focalisent sur un seul type de SGBD NoSQL.

Les auteurs de [Li et al., 2014] présentent une approche MDA assurant le passage directe d'un DCL d'UML vers le modèle physique du SGBD orienté-colonnes HBase. Dans [Daniel et al., 2016] les auteurs proposent une solution de transformation d'un modèle conceptuel UML en un modèle physique NoSQL ; cette solution repose sur l'utilisation d'un ensemble de règles qui sont spécifiques aux BD orientées-graphes. Ainsi, chacun de ces travaux s'applique uniquement à un seul type de systèmes NoSQL : orienté-colonnes dans [Li et al., 2014] et orienté-graphes dans [Daniel et al., 2016].

La solution Object2NoSQL que nous avons proposée présente deux avantages majeurs :

- elle est compatible avec les quatre types de SGBD NoSQL : orientés colonnes, documents, graphes et clé-valeur ;

- elle introduit un niveau logique intermédiaire qui assure l'indépendance des niveaux conceptuel et physique.

En effet, le niveau logique apporte une certaine stabilité dans notre processus de transformation lors de l'évolution des caractéristiques physiques du SGBD NoSQL utilisé. Toute évolution technique impactera uniquement le niveau physique mais n'affectera pas le niveau logique. La transformation logique-physique sera alors relancée, mais il n'y aura aucune incidence sur la transformation conceptuel-logique (il est inutile de remonter au modèle conceptuel).

Chapitre 4. Processus de Traduction des Contraintes

Les utilisateurs d'applications Big Data stockent généralement les données sur des plateformes NoSQL. Le point de départ du processus de stockage est constitué d'un modèle conceptuel qui décrit la sémantique des données en l'absence de considérations techniques.

Tout modèle conceptuel contient (1) la structure des données et (2) un ensemble de contraintes destinées à assurer la cohérence de ces données lors de leurs mises à jour. Comme il est précisé dans le chapitre 2, seuls quelques travaux ont présenté des processus de transformation de modèles conceptuels en modèles NoSQL en associant des contraintes.

Le maintien de la cohérence des données stockées constitue l'un des atouts majeurs des SGBD puisqu'il garantit aux utilisateurs la qualité des données et, par conséquent, la qualité des systèmes d'information. Les SGBD NoSQL actuels offrent peu de mécanismes pour garantir cette cohérence. Ils prennent bien sûr en compte certaines contraintes dans le modèle physique, comme le type des valeurs ou l'identification par une clé de chaque ligne dans une table. Par contre, la prise en compte des relations entre les tables est à la charge de l'utilisateur ; celui-ci doit décrire le mode d'implantation de chaque relation dans le modèle physique puis écrire le code permettant d'établir les liens en garantissant la cohérence des données lors de leur saisie.

Ainsi, le mécanisme d'implantation des contraintes d'intégrité, qui ne peuvent pas être prises en compte directement dans le modèle physique, constitue un autre verrou auquel nous nous sommes confrontés.

Nous proposons à l'utilisateur un processus permettant de traduire automatiquement des contraintes lors de la transformation du modèle conceptuel en un modèle physique.

L'originalité de notre processus d'implantation réside dans la généralisation de la démarche utilisée. Nous transposons les principes utilisés pour la transformation des structures des données sur la transformation des contraintes d'intégrité. En effet, nous utilisons l'architecture MDA et sa métamodélisation pour spécifier les opérations de transformation des modèles.

Plan du chapitre. La section 4.1 introduit le processus de transformation des contraintes. Les sections 4.2 et 4.3 détaillent les étapes de ce processus.

4.1 Aperçu de notre processus

Ce chapitre se focalise sur le processus de transformation de contraintes d'intégrité associées à la description conceptuelle des données. Pour bien situer ce processus lié aux contraintes, il est nécessaire de connaitre le cadre dans lequel il va s'insérer ; il s'agit de la démarche Object2NoSQL présentée dans le chapitre précèdent. Nous rappelons uniquement les grandes lignes de cette démarche de transformation de modèles.

Object2NoSQL est une démarche de type MDA permettant de générer des modèles physiques NoSQL à partir d'un modèle conceptuel UML. Son principe consiste à produire un modèle générique NoSQL qui fait apparaitre principalement des tables et des relations binaires. Dans un second temps, ce modèle générique permet de générer un modèle physique adapté à un système NoSQL choisi par l'utilisateur. Compte tenu de la généricité du modèle logique que nous avons proposé, le processus est capable de transformer ce modèle en un modèle physique pour l'une des plateformes d'implantation connues : colonnes, documents, graphes ou clé-valeur. Ce principe assure une vision générique de l'implantation des données et garantit l'indépendance de leur description vis-à-vis des spécificités techniques des plateformes NoSQL et de leurs évolutions.

Notre processus Object2NoSQL se compose de deux étapes (cf. figure 4.1). La première est chargée de transformer le diagramme de classes UML d'entrée (PIM conceptuel) en un modèle générique NoSQL (PIM logique). La deuxième transformation génère un modèle physique NoSQL (PSM) et un ensemble de directives d'assistance spécifiques au système d'implantation choisi par l'utilisateur. Dans ce processus, nous n'avons considéré que des contraintes simples telles que les types de données et l'unicité des identifiants. L'objectif de nos travaux est de compléter notre processus en prenant en compte d'autres catégories de contraintes plus complexes.

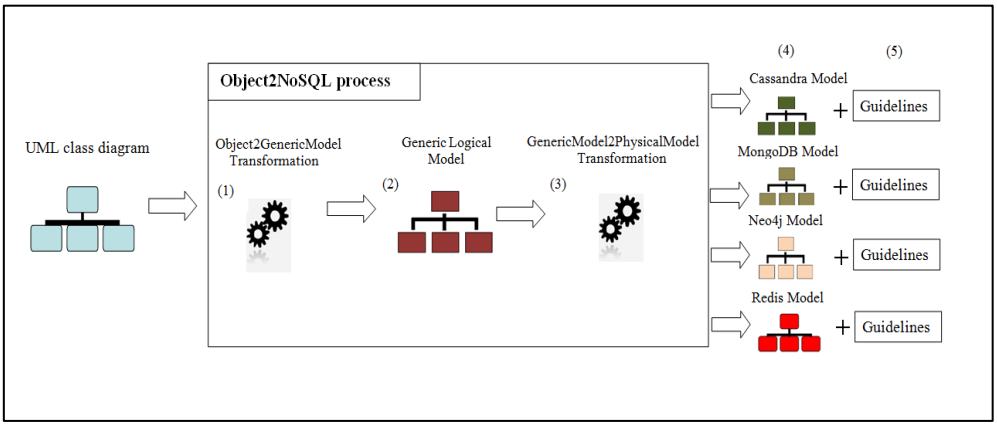


Figure 4.1 – Processus Object2NoSQL de transformation d'un DCL

Une contrainte est une expression donnant une propriété des données, une restriction ou des informations complémentaires sur un élément du modèle [Liu

et al., 2000]. Elle peut porter par exemple sur une table, une ligne, un attribut ou une relation entre deux lignes. Son objectif est d'assurer la cohérence de la base de données après chaque mise à jour. Nous adoptons une expression des contraintes sous la forme d'un langage basé sur la théorie des ensembles et la logique des prédicats. Il s'agit du langage OCL (Object Constraint Language) normalisé par l'OMG.

Les SGBD NoSQL ne disposant pas de système d'intégrité évolué, l'idée est de transformer des contraintes OCL en méthodes Java. Ces méthodes seront mises en œuvre au niveau physique, lors des mises à jour, pour vérifier les contraintes sur une BD NoSQL. Pour faire ceci, nous proposons le processus OCL2Java qui prend en entrée des contraintes OCL associées à un DCL et fournit en sortie des méthodes programmées en Java. Comme le montre la figure 4.2, pour chaque contrainte, le processus est réalisé en deux étapes successives qui produisent respectivement :

- un modèle de méthode java : il s'agit d'un modèle pivot (de niveau logique) compatible avec des plateformes NoSQL de types différents (colonnes, documents, graphes et clé-valeur). Cette compatibilité est principalement liée au fait que ce modèle fait abstraction du langage de requête de chaque système NoSQL.

- le code de la méthode : à partir du modèle précédent, le processus génère le code java de la méthode (de niveau physique). Plusieurs codes peuvent être produits à partir du même modèle ; chacun d'eux est spécifique parce qu'il contient des requêtes d'accès aux données. Ces requêtes sont exprimées dans le langage de requête associé à chaque SGBD.

L'intérêt d'introduire un niveau intermédiaire situé entre les contraintes OCL (niveau conceptuel) et le code (niveau physique) est de généraliser notre processus de transformation des contraintes à trois niveaux (conceptuel, logique et physique). En effet, le niveau logique sert à limiter les impacts liés aux spécificités des langages de requête propres à chaque système NoSQL. Ainsi, toute évolution de tels langages ou tout changement du système nécessitera uniquement une nouvelle transformation logique/physique ; il sera donc inutile de remonter au niveau supérieur des contraintes OCL.

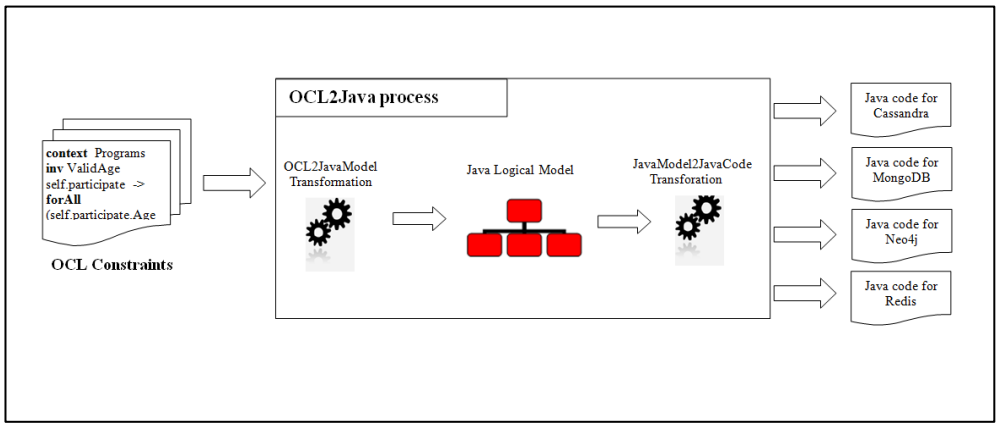


Figure 4.2 – Processus OCL2Java de transformation des contraintes

La figure 4.2 montre les différents composants du processus OCL2Java. Comme nous l'avons précisé précédemment, la transformation des contraintes se fait en deux étapes successives. La première correspond à la transformation OCL2JavaModel qui traduit une contrainte OCL en un modèle de méthode java. Il s'agit d'une transformation Model-To-Model où les modèles source et cible sont respectivement conformes aux métamodèles OCL et Java proposés dans la section suivante. JavaModel2JavaCode est la deuxième transformation de type Model-To-Text qui génère le code java d'une méthode à partir de son modèle. Les contraintes OCL peuvent donc être vérifiées après une exécution des méthodes java correspondantes sur le système d'implantation choisi.

4.2 Transformation OCL2JavaModel

Dans cette section, nous présentons la première étape de notre processus de transformation des contraintes. Il s'agit de la transformation OCL2JavaModel qui traduit une expression OCL en un modèle de méthode java. Nous commençons par définir la structure des métamodèles OCL et Java que nous proposons pour effectuer cette transformation ; nous explicitons ensuite le passage entre les éléments des modèles source et cible correspondants.

4.2.1 La source : Invariant OCL

OCL est un langage formel standardisé par l'OMG. Il permet de spécifier des informations supplémentaires qui n'ont pas pu l'être avec le formalisme de modélisation standard d'UML. OCL permet d'exprimer deux types de contraintes : (1) des invariants de classes et (2) des pré/post-conditions à l'exécution d'une opération. Dans le cadre de nos travaux sur la transformation des contraintes, nous considérons uniquement le premier type portant sur la description des données.

Un invariant est une condition que toutes les instances d'une classe doivent respecter. Cette condition est définie sous la forme d'une expression OCL qui

forme le corps de l'invariant. Tout invariant est lié à un contexte spécifique ; c'est la classe sur laquelle s'applique la contrainte. La syntaxe de création d'un invariant est de la forme suivante :

context <nomClasse>
inv <nomContrainte> :
<expressionsOCL>

Context et inv sont des mots clés. <nomClasse> est la classe à laquelle l'invariant est attaché. <nomContrainte> est un nom optionnel qui peut être donné à la contrainte ; dans nos travaux, nous avons choisi d'imposer un nom à tous les invariants pour les désigner en Java. <expressionsOCL> est le corps de l'invariant qui est composé d'une ou plusieurs expressions OCL.

Le contexte peut être utilisé à l'intérieur d'une expression grâce au mot-clef self.

Exemple:

A partir du DCL de notre application médicale (cf. figure 1.7), nous pourrions écrire l'invariant suivant :

context Patients
inv AgeValide :
(self.age >=0) and (self.age <= 120) -- l'âge de chaque patient est compris entre 0 et 120 ans

Définition 1. Un invariant i est défini par (C, N, EX) où :

- C est le contexte de i,
- N est le nom désignant i,
- $EX = \{ex_1, \ldots, ex_m\}$ est un ensemble d'expressions OCL formant le corps de i.

Afin d'assurer le passage automatique d'un invariant vers un modèle d'une méthode java, nous avons dû formaliser préalablement les expressions OCL figurant dans la documentation de l'OMG. Dans cette documentation, le concept « Expression OCL » est défini informellement dans des contextes différents. Ainsi, il est difficile de proposer une formalisation exhaustive qui définit précisément tous les types possibles d'expressions OCL. Nous nous sommes donc limités aux expressions fondamentales qui sont fréquemment utilisées. Il s'agit des expressions itératives, des expressions opération, des expressions conditionnelles et des expressions de définition.

Dans ce qui suit, nous définissons chacune de ces expressions et nous illustrons les différents concepts qui les composent.

4.2.1.1 Expression de définition : Let-In

Cette expression est utilisée pour définir en OCL (i.e. déclarer et initialiser) une variable afin qu'elle puisse être utilisée dans l'expression qui suit le *In*. L'intérêt est de faciliter l'écriture d'un invariant et d'éviter les répétitions.

Syntaxe :

Let <nomVar> : <typeVar> = <expression_1> *In* <expression_2>

<nomVar> et <typeVar> correspondent respectivement au nom et au type de la variable à définir. <expression_1> est une expression d'initialisation ; <expression_2> est l'expression dans laquelle la variable sera utilisée.

Exemple:

L'invariant suivant permet d'assurer l'unicité du numéro de sécurité sociale de chaque patient :

```
context Programme
inv NssUnique :
Let isUnique : Boolean = self.patient -> forAll(p1, p2 : Patients | p1 <> p2
implies p1.nss <> p2.nss) In
if isUnique  = True
then return 1
else return 0
endif
```

Les expressions If-Then-Else et forAll() utilisées dans cet exemple seront présentées dans la suite.

Définition 2. Une expression Let-In ex^{letIn} est définie par (ex^o, ex^u), où :

- ex^o = ($operande_1$, $operateur$, $operande_2$) est une expression opération (cf. section 4.2.1.4), où :
  - $operande_1$ = (N^v, T^v) : est une expression permettant de déclarer une variable v, où :
    - N^v est le nom de v,
    - T^v est le type de v.
  - $Operateur.N$ = « = »,
  - $operande_2$ est une expression permettant d'initialiser la variable v déclarée par $operande_1$. Elle peut correspondre à l'une des expressions suivantes :
    - $operande_2 = self.a^{c^{ctx}}$, où :
      - $self$: fait référence à une classe c^{ctx} représentant le contexte de l'invariant dans lequel l'expression Let-In est incluse,
      - $a^{c^{ctx}}$: est un attribut de c^{ctx} défini par (N, T), où N et T sont respectivement le nom et le type de $a^{c^{ctx}}$.
    - $operande_2 = self.role^c.a^c$, où :
      - $role^c$: est le rôle d'une classe c dans l'association qui la relie à la classe c^{ctx},
      - a^c = (N, T) : est un attribut de c.
    - $operande_2$ = ex^r, où ex^r est une expression requête (cf. section 4.2.1.3) ; il s'agit généralement d'une expression utilisant l'opération exists() (cf. Définition 8) ou l'opération forAll() (cf. Définition 7).
    - $operande_2 = ex^o$, où ex^o est une expression opération qui peut correspondre à l'une des expressions suivantes :
      - ex^o = $self.role^c$ -> $operation()$ (cf. Définition 11), où : $operation()$ correspond généralement à l'opération size () (cf. section 4.2.1.4).
      - ex^o = ($operande_1'$, $operateur$, $operande_2'$) (cf. Définition 12). Dans ce cas, $operande_1'$ et $operande_2'$ correspondent respectivement à un attribut $self.a^{c^{ctx}}$ et une valeur de type OCL prédéfini. $operateur$ est un opérateur OCL prédéfinis sur les entiers ou les réels, par exemple : +, -, *, /.
- ex^u est une expression utilisant la variable définie par ex^o. Généralement, ex^u correspond à une expression conditionnelle ($If - Then - Else$ ou $Implies$) (cf. Définitions 3 et 4).

4.2.1.2 Expressions conditionnelles : If et Implies

Dans un invariant, certaines expressions dépendent d'autres expressions. Pour exprimer ceci, OCL offre les deux syntaxes suivantes :

Syntaxe de la forme1 :

If <expression_1> then <expression_2> else <expression_3> endif

Définition 3. Une expression If-Then-Else ex^{ite} contient trois expressions opération. Chacune d'elles peut être définie selon les 2 formes suivantes :

- ex^{o} = ($operande_1$, *operateur*, $operande_2$), où :
  - Pour $operande_1$, deux cas peuvent être considérés :
    - $operande_1 = N^{v}$: correspond à une variable déclaré et initialisé par une expression Let-In (cf. Définition 2),
    - $operande_1 = self.a^{c^{ctx}}$ (cf. Définition 2, cas où : $operande_2 = self.a^{c^{ctx}}$),
  - *Operateur* correspond à un opérateur de comparaison OCL prédéfini ; par exemple : <=,>=,<>,=.
  - $operande_2$ est une valeur de type OCL prédéfini (Integer, Real, Boolean, String, etc.).
- ex^{o} = $self.role^{c}$ -> *operation()* (cf. Définition 11), où : *operation()* est une opération retournant un booléen ; il s'agit généralement des opérations isEmpty() et notEmpty() (cf. section 4.2.1.4).

Exemple :

context Patients
inv CategoriePatient :
if age <=12 then Catégorie = "enfant"
else Catégorie = " adulte"
endif

Syntaxe de la forme 2:

<expression_1> implies <expression_2>

Cette expression s'interprète de la façon suivante : Si <expression_1> est vraie, alors <expression_2> doit être vraie également. Si <expression_1> est fausse, alors l'expression complète est vraie.

Définition 4. Une expression Implies ex^{ip} est définie par (ex_1, ex_2), où ∀ i ∈ [1,2], ex_i est soit une expression opération ex^o (cf. section 4.2.1.4), soit une expression requête ex^r (cf. section 4.2.1.3).

Dans le premier cas ($ex_i = ex^o$), deux formes peuvent être considérées :

- ex^o = ($operande_1$, *operateur*, $operande_2$), où : *operateur* est un opérateur OCL prédéfini. Pour $operande_1$ et $operande_2$, plusieurs manières sont disponibles pour les définir ; nous citons les plus utilisées :
  - $operande_1$ et $operande_2$ sont deux attributs a_1 et a_2 ; chacun d'eux peut être défini en utilisant les expressions suivantes : *self.* $a^{c^{ctx}}$ ou *self.* $role^c.a^c$ (cf. Définition 2),
  - $operande_1$ est un attribut défini en utilisant les expressions *self.* $a^{c^{ctx}}$ ou *self.* $role^c$. a^c et $operande_2$ une valeur de type OCL prédéfini.
- ex^o = *self.*$role^c$ *-> operation()* : une expression opération (cf. Définition 11),

Dans le deuxième cas ($ex_i = ex^r$), il s'agit généralement d'une expression utilisant l'opération exists() (cf. Définition 8) ou l'opération forAll() (cf. Définition 7).

4.2.1.3 Expressions itératives / Expressions requête

Les expressions itératives, appelées également les expressions requête, sont des expressions OCL permettant d'exprimer des contraintes sur les éléments d'une collection. Notons que dans la suite du mémoire, nous utiliserons le terme « collection » de la norme OCL qui correspond à un ensemble d'objets. La syntaxe générale d'une expression requête est la suivante :

Syntaxe :

<expression_source> -> <operation> (<expression_declaration> | <expression_argument>)

L'expression <expression_source> retourne une collection d'éléments. L'expression <expression_argument> porte sur les propriétés de ces éléments et elle est évaluée pour chacun d'eux en appliquant l'opération <operation>.

Il est recommandé de faire référence aux propriétés de l'élément courant dans l'expression <expression_argument> en utilisant la notation : <élément>.<propriété>. Pour cela, il est nécessaire de déclarer <élément> dans l'expression <expression_declaraion> ; <élément> jouera dans ce cas le rôle d'un itérateur.

Exemple :

L'expression suivante représente dans le contexte d'un programme médical les patients âgés de plus de 60 ans :

context Programme …
… self.patient -> select (p : Patients | p.age >= 60) …

expression_source opération expression_declaration expression_argument

Formellement :

Définition 5. Une expression requête ex^r est définie par (ex^s , $operation$, ex^d, ex^a) où :

- ex^s est une expression retournant une collection d'éléments $\{o_1, \ldots, o_n\}$, où ∀ i ∈ [1..n], o_i est un objet d'une classe c liée à la classe c^{ctx} représentant le contexte de l'invariant dans lequel l'expression requête est incluse . ex^s a la forme suivante : $self.role^c$, où :
  - $self$: fait référence à la classe c^{ctx},
  - $role^c$: est le rôle de la classe c dans l'association qui la relie à la classe c^{ctx}.
- $operation$: est une opération OCL désignée par un nom N. Dans nos travaux, nous considérons les opérations OCL : select, forAll (avec un ou deux itérateurs), exists, collect et isUnique,
- ex^d est une expression permettant de déclarer un objet o de c. Elle est définie par un couple (N, T), où :
  - N: est le nom de o,
  - T = c.N : est le type de o.
- ex^a : est une expression à évaluer pour tout objet o retourné par ex^s. Elle est définie ci-dessous dans les définitions 6, 7, 8, 9 et 10 en fonction de l'opération utilisée dans ex^r.

Le résultat final retourné par une expression requête (une collection d'objets, une collection de valeurs d'un attribut ou un booléen) dépend de l'opération <operation> utilisée : select, forAll, exists, collect, isUnique. Dans la suite, nous présentons chacune de ces opérations avec des exemples illustratifs.

a. Select

L'opération *select* permet de générer une sous-collection à partir de la collection retournée par l'expression ex^s. Cette sous-collection ne contient que les éléments vérifiant l'expression ex^a.

Définition 6. Une opération select est définie par (cll^{in}, ex^{a}, cll^{out}), où :

- cll^{in} = { o_1 ,…, o_n } est la collection source retournée par ex^{s} (cf. Définition 5),
- ex^{a} = ex^{o}, où ex^{o} est une expression opération (cf. section) de la forme suivante : ($operande_1$, $operateur$, $operande_2$), où :
  - $operande_1 = o.a$, où o est un objet déclaré préalablement dans ex^{d} (cf. Définition 5) et a un attribut de o,
  - $operateur$ correspond à un opérateur OCL prédéfini. Par exemple : <=,>=,<>,=,
  - $operande_2$ est une valeur de type OCL prédéfini (Integer, Real, Boolean, String, etc.).
- cll^{out} = {o_1,…, o_m} est la collection cible générée à partir de cll^{in}, où : ∀ j ∈ [1..m] o_j est un objet vérifiant ex^{a}.

Exemple :

Pour écrire une contrainte imposant qu'un programme médical doit posséder, parmi les patients impliqués, au moins un patient de plus de 60 ans, nous écrivons l'invariant suivant :

```
context Programme
inv ContrainteAge:
self.patient -> select (p : Patients | p.age>=60) -> notEmpty()
```

Notons que « notEmpty() » est une opération OCL prédéfinie que nous allons présenter dans la section suivante.

b. forAll

Cette opération représente le quantificateur universel ∀ ; elle permet d'exprimer une contrainte sur l'ensemble des éléments d'une collection. Le résultat retourné est de type Booléen. Autrement dit, l'expression requête utilisant l'opération forAll est évaluée à vrai si l'expression ex^{a} est vraie pour tous les objets retournés par l'expression ex^{s}. Sinon, l'expression requête est évaluée à faux.

Exemple :

L'invariant suivant spécifie que tous les patients impliqués dans le programme médical ont plus de 16 ans.

```
context Programme
inv ContrainteAge :
self.patient -> forAll(p : Patients | p.age>=16)
```

Notons que l'opération forAll possède une variante utilisant deux itérateurs ; chacun d'eux parcourra l'ensemble de la collection source. Il s'agit d'une

opération forAll classique (ne possédant qu'un seul itérateur) réalisée sur le produit cartésien de la collection retournée par l'expression <expression_source> sur elle-même (produit cartésien réflexif).

Exemple :

L'invariant suivant impose qu'il n'existe pas deux instances de la classe Patients pour lesquelles l'attribut NSS (Numéro de Sécurité Sociale) a la même valeur.

```
context Programme
inv ContrainteNSS :
self.patient -> forAll(p1, p2 : Patients | p1 <> p2 implies p1.nss <> p2.nss)
```

Ainsi, l'expression requête incluse dans cet invariant est évaluée à vrai si les NSS de tous les patients sont différents.

Définition 7. Une opération forAll est définie par (cll^{in}, ex^{a}, a^{out}), où :

- cll^{in} = { o_1 ,…, o_n } est la collection source retournée par ex^{s} (cf. Définition 5),
- La définition de ex^{a} dépend du nombre d'itérateurs utilisés :
  - Si le nombre d'itérateur = 1 : ex^{a} correspond à une expression opération ex^{o}= ($operande_1$, $operateur$, $operande_2$), où :
    - $operande_1 = o.a$, où o est un objet déclaré préalablement dans ex^{d} (cf. Définition 5) et a un attribut de o,
    - $operateur$ correspond à un opérateur OCL prédéfini.
    - $operande_2$ est une valeur de type OCL prédéfini (Integer, Real, Boolean, String, etc.).
  - Si le nombre d'itérateur = 2 : $ex^{a} = ex^{ip} . ex_2$, avec ex^{ip} une expression Implies (cf. Définition 4) et ex_2 une expression opération de la forme suivante : ($operande_1$, $operateur$, $operande_2$), où $operande_1$ et $operande_2$ correspondent à un attribut des objets de cll^{in}.
- a^{out} est un attribut de type booléen. a^{out} = vrai si :∀ $o_i \in cll^{in}$, avec i ∈ [1..n], l'expression ex^{a} est vraie. a^{out} = faux si : ∀ i ∈ [1..n], ∃ $o_i \in cll^{in}$ pour lequel l'expression ex^{a} est fausse.

c. Exists

Exists représente le quantificateur existentiel ∃. Elle permet de vérifier si l'expression < expression_argument> est vraie pour au moins un élément de la collection retournée par l'expression <expression_source>. Le résultat retourné est donc de type Boolean.

Exemple :

Pour exprimer une contrainte imposant que le programme médical doit posséder, parmi ses médecins impliqués, au moins un chirurgien, nous écrivant l'invariant suivant :

context Programme
inv ContrainteMédecin :
self.médecin -> exists(m : Médecins | m.spécialité = "chirurgie")

Définition 8. Une opération exists est définie par (cll^{in}, ex^{a}, a^{out}), où :

- cll^{in} = { o_1 ,…, o_n } est la collection source retournée par ex^{s} (cf. Définition 5),
- ex^{a} = ex^{o}, où ex^{o} est une expression opération de la forme suivante : ($operande_1$, *operateur*, $operande_2$), où : $operande_1 = o.\ a$, *operateur* correspond à un opérateur OCL prédéfini et $operande_2$ est une valeur de type OCL prédéfini (Integer, Real, Boolean, String, etc.) (cf. Définition 6, $ex^{a} = ex^{o}$).
- a^{out} est un attribut de type boolean. a^{out} = vrai si : ∀ i ∈ [1..n], ∃ o_i ∈ cll^{in} pour lequel l'expression ex^{a} est vraie. a^{out} = faux si : ∀ i ∈ [1..n], ∄ o_i ∈ cll^{in} pour lequel l'expression ex^{a} est vraie.

d. Collect

L'opération *collect* permet de créer une collection à partir de la collection retournée par l'expression ex^{s}. La nouvelle collection possède le même nombre d'éléments que la collection initiale, mais le type de ces éléments est différent (cf. l'exemple ci-après).

Cette opération fonctionne ainsi : pour chaque élément de la collection source, l'expression ex^{a} est évaluée. Le résultat est ensuite ajouté dans la collection cible. C'est le même principe de l'opérations *select* présentée précédemment. La différence est au niveau du type des éléments composant la collection retournée par chaque opération. Pour *select*, c'une collection d'objets et pour *collect*, il s'agit d'une collection de valeurs d'un attribut.

Notons que généralement dans le résultat retourné par l'opération collect, il peut y avoir des doublons.

Exemple :

Dans le contexte d'un programme médical, pour récupérer toutes les spécialités des médecins impliqués, nous pouvons utiliser l'opération collect comme suit :

context Programme …
… self.medecin -> collect (m : Medecins | m. spécialité)
…

Dans cet exemple, l'opération *collect* est appliquée à une collection de médecins et retourne une collection de valeurs de l'attribut spécialité.

Définition 9. Une opération collect est définie par (cll^{in}, ex^{a}, cll^{out}), où :

- cll^{in} = { o_1 ,…, o_n } est la collection source retournée par ex^{s} (cf. Définition 5),
- ex^{a} = o.a, où : o est un objet déclaré préalablement dans ex^{d} (cf. Définition 5) et a un attribut de o,
- cll^{out} = {v_1^a,…, v_m^a} est la collection cible générée à partir de cll^{in}, où : ∀ j ∈ [1..m] v_j^a est la valeur de l'attribut a des objets de cll^{in} indiqué dans l'expression ex^{d} (cf. Définition 5).

e. isUnique

Cette opération retourne un Booléen qui vaut vrai si l'évaluation de l'expression ex^{a} produit une valeur différente pour chaque élément de la collection retournée par l'expression ex^{s}. Autrement, le Booléen vaut faux.

Exemple :

Avec l'invariant suivant, nous pouvons vérifier qu'il n'y a pas des professionnels de santé possédant le même identifiant :

```
context Programme
inv ContrainteId :
self.prosanté -> isUnique(ps : ProSanté | ps.identifiant)
```

Définition 10. Une opération isUnique est définie par (cll^{in}, ex^{a}, a^{out}), où :

- cll^{in} = { o_1 ,…, o_n } est la collection source retournée par ex^{s} (cf. Définition 5),
- La définition de ex^{a} est identique à celle présentée précédemment dans Définition 9,
- a^{out} est un attribut de type booléen. a^{out} = vrai si : ∀ i,j ∈ [1..n] l'évaluation de l'expression ex^{a} pour o_i retourne une valeur différente de celle produite par l'évaluation de l'expression ex^{a} pour o_j, avec o_i et o_j sont des objets de cll^{in} et i ≠ j.

4.2.1.4 Expressions opération

Une expression opération en OCL est incluse pour définir les expressions : Let-In, If-Then-Else, Implies et les expressions requête, comme nous l'avons montré en définissant chacune de ces expressions. Aussi, les expressions opération peuvent être utilisées seules dans le corps d'un invariant, indépendamment des autres expressions OCL. Nous pouvons les classer en deux catégories : (1) des expressions opération portant sur des opérations OCL prédéfinies et (2) celles

utilisant des opérateurs OCL prédéfinis. Syntaxe 1 et Syntaxe 2 définissent respectivement la syntaxe générale de chacune de ces catégories :

Syntaxe 1 :

<expression_source> -> <operation> ()

L'expression <expression_source> est l'entrée de l'opération <operation> ; elle retourne une collection d'éléments. Le résultat retourné par une expression opération dépend de l'opération <operation> utilisée.

OCL propose plusieurs opérations prédéfinies. Nous avons présenté dans la section précédente (Expressions requête), les opérations de base sur les collections (select, forAll, exists, collect et isUnique) qui sont fréquemment utilisées dans le cadre d'une expression requête. Ces opérations portent sur les éléments de la collection d'entrée ; elles évaluent une expression (expression_argument) pour chacun de ces éléments. Nous rappelons que l'expression <expression_argument> porte sur les attributs des éléments d'entrée. Nous complétons dans cette section la présentation des opérations OCL prédéfinies en décrivant d'autres opérations portant sur la collection d'entrée.

Définition 11. Une expression opération ex^o utilisant une opération prédéfinie est définie par (ex^s, *operation)* où :

- ex^s est une expression retournant une collection d'éléments $\{e_1, \ldots, e_n\}$, où $\forall$ i $\in$ [1..n], e_i est soit un objet d'une classe c liée à la classe c^{ctx} représentant le contexte de l'invariant dans lequel l'expression opération est incluse, soit une valeur d'un attribut de c. Ainsi, ex^s peut être définie selon les deux manières suivantes :
  - $ex^s = (self, role^c)$, où :
    - $self$: fait référence à la classe c^{ctx},
    - $role^c$: est le rôle de la classe c dans l'association qui la relie à la classe c^{ctx}.
  - $ex^s = ex^r$ où ex^r est une expression requête qui retourne une collection d'objets ou de valeurs d'un attribut (cf. section 4.2.1.3).
- *Operation* est l'opération OCL à appliquer sur les éléments retournés par ex^s. Elle est désignée par un nom N. Dans nos travaux, N prendra une valeur dans l'ensemble suivant : {size(), isEmpty(), notEmpty()}.

Nous rappelons que nous ne considérons pas toutes les opérations OCL prédéfinies. Nous présentons uniquement celles les plus utilisées.

a. Size()

Cette opération retourne un entier correspondant au nombre d'éléments de la collection retournée par l'expression ex^s.

Exemple 1 :

L'invariant suivant précise que chaque établissement participant au programme médical possède un médecin-coordonnateur unique :

```
context Etablissement
inv ContrainteMedecinCo :
self.medecinco -> size() = 1
```

Expression opération

L'expression opération utilisée dans cet invariant applique l'opération size () sur la collection d'objets (les médecins coordinateurs) retournés par l'expression « self.medecinco ». Cet exemple illustre le premier cas des expressions opération où l'entrée est retournée par une expression de la forme : self.role. L'exemple 2 correspond au deuxième cas où l'entrée de l'expression opération correspond à une expression requête.

Exemple 2 :

Cet exemple se place dans le contexte du programme médical et indique que ce dernier doit posséder, parmi ses patients impliqués, au moins un patient de plus de 60 ans :

```
context Programme
inv ContrainteAge:
self.patient -> select (p : Patients | p.age>=60) -> size() >=1
```

Expression requête

Expression opération

b. isEmpty() / notEmpty

L'opération isEmpty() (resp. notEmpty) retourne un booléen qui vaut Vrai (resp. Faux) si la collection retournée par l'expression ex^s est vide (resp. n'est pas vide).

Exemple :

En utilisant une expression opération, nous pouvons spécifier que la collection de tous les patients sans antécédents, est vide.

```
context Programme
inv ContraintePatient :
self.patient -> reject( aAntécédents)->isEmpty()
```

Syntaxe 2 :

<operande_1> <operateur> <operande_2>

Le langage OCL possède un certain nombre d'opérateurs prédéfinis. Nous citons par exemple : '+', '-', '*', '/', '<', '>', '<>' '<=' '>=', 'and', 'or' et 'xor'. La syntaxe 2 correspond à la syntaxe générale des expressions opération utilisant ces opérateurs.

Généralement, <operande_1> correspond à un attribut caractérisant les objets d'une classe et <operande_2> correspond soit à un autre attribut (de la même classe ou d'une classe différente) soit à une valeur de type prédéfini (Integer, Real, String, Boolean, etc.).

Définition 12. Une expression opération ex^o utilisant un opérateur prédéfini est définie par ($operande_1$, $operateur$, $operande_2$), où : $operateur$ est un opérateur OCL désigné par un nom N. $operande_1$ et $operande_2$ sont définis dans les Définitions ci-dessus en fonction des expressions dans lesquelles ils sont utilisés.

Exemple :

Dans le cadre d'un programme médical, toutes les dates de consultation doivent être postérieures à la date du début du programme. Nous pouvons exprimer cette contrainte comme suit :

```
context Consultations
inv contrainteDate :
self.dateConsultation < self.programme.dateDebut
```

Nous présentons les différents concepts présentés dans cette section à travers le métamodèle OCL de la figure 4.3. Nous avons défini ce métamodèle en se référant aux spécifications de l'OMG[18].

[18] http://www.omg.org/spec/OCL/

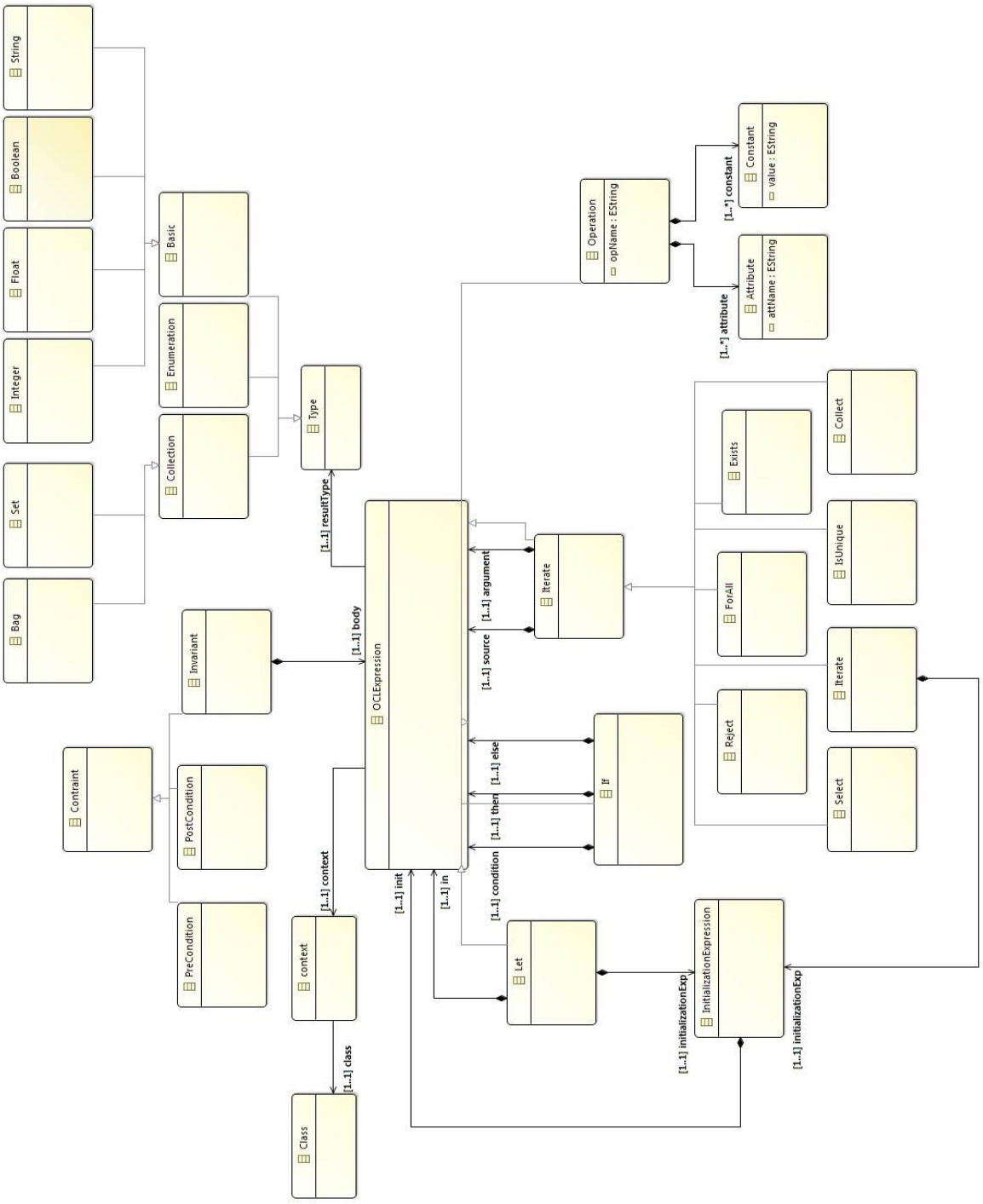


Figure 4.3 – Métamodèle OCL

4.2.2 La cible : Modèle d'une méthode Java

La source de la transformation OCL2JavaModel correspond à des invariants OCL et la cible à des modèles de méthodes java. Par la suite, chaque modèle sera transformé en un code dont l'exécution permettra de vérifier l'invariant correspondant. Compte tenu de la justification donnée dans la section 4.1, nous n'avons pas effectué une transformation directe d'un invariant OCL vers le code java spécifique à un SGBD NoSQL ; nous passons par un niveau logique.

Dans cette section, nous rappelons les concepts importants liés à une méthode java. Nous nous limitons aux éléments nécessaires pour transformer les expressions OCL considérées dans la section précédente (La source).

Une méthode java est associée à une classe et permet d'effectuer des traitements sur les objets de cette classe. Elle est définie par : (1) une visibilité (public, privé ou protégé), (2) un type de retour, (3) un nom, (4) une liste de paramètres et (5) un corps composé d'une ou plusieurs instructions java. Ainsi, nous pouvons formaliser une méthode java comme suit :

Définition 13. une méthode java m est définie par (C, V, T^r, N, PR, IS) où :

- C est la classe à laquelle m est associée,
- V est la visibilité de m. Dans notre cas, toutes les méthodes générées sont des méthodes publiques,
- T^r est le type de la valeur retournée par m. Dans notre cas, chaque méthode java permet de vérifier si un invariant est respecté ou non. Ainsi, la valeur retournée est de type Booléen,
- N est le nom désignant m,
- PR = $\{pr_1, \ldots, pr_n\}$ est l'ensemble de paramètres de m. $\forall$ i $\in$ [1..n], pr_i est défini par (N, T), où pr_i. N est le nom du paramètre et pr_i.T est son type. Notons que m peut ne pas avoir des paramètres ; dans cas, PR = $\emptyset$,
- IS = $\{is_1, \ldots, is_m\}$ est un ensemble d'instructions formant le corps de m.

Dans ce qui suit, nous définissons quelques instructions pouvant être utilisées dans le corps d'une méthode java. Nous rappelons que nous nous limitons aux instructions nécessaires à la transformation des invariants OCL supportés par notre processus.

4.2.2.1 Instruction de déclaration et d'initialisation

En java, on peut combiner la déclaration et l'initialisation d'une variable dans une seule instruction. La déclaration précise le type de la valeur que la variable va contenir suivi par son nom. Le type peut être : (1) un type standard (int, float, string, etc.) ou (2) une classe. L'initialisation de la variable déclarée porte sur l'utilisation de l'opérateur d'affectation "=".

Définition 14. Une instruction de déclaration et d'initialisation is^{di} d'une variable v est définie par (T^v, N^v, N^{op}, Vl^v), où :

- T^v est le type de la variable,
- N^v est le nom de la variable,
- N^{op} est le nom de l'opérateur utilisé pour affecter une valeur à v. Dans ce cas, il s'agit de l'opérateur « = »,
- Vl^v est la valeur à affecter à v.

Nous pouvons aussi créer une variable sans l'initialiser en utilisant l'instruction de déclaration is^d = (T^v, N^v).

4.2.2.2 Instruction de test

L'instruction de test If est une instruction de contrôle qui permet d'exécuter des instructions en fonction de la valeur d'une condition.

Définition 15. Une instruction de test is^{if} est définie par ($condition$, IS^v, IS^f), où :

- $condition$: est une instruction logique is^l (cf. Définition 16),
- IS^v = { is_1^v , …, is_n^v }: est un ensemble d'instructions à exécuter si $condition$ = vrai,
- IS^f = { is_1^f , …, is_m^f }: est un ensemble d'instructions à exécuter si $condition\ condition = faux$.

4.2.2.3 Instruction logique

Généralement, une instruction logique est construite en utilisant les opérateurs de comparaison : ==, !=, <, >, <= et >=. Les utilisations les plus fréquentes de ces opérateurs consiste à utiliser des variables et des résultats retournés par des méthodes prédéfinies.

Une autre façon pour définir une instruction logique consiste à combiner plusieurs instructions logiques avec les opérateurs « && » et « || ».

Définition 16. Une instruction logique is^l peut être définie selon l'une des deux manières suivantes :

- is^l = ($v_1, operateur^c$, v_2), où : v_1 et v_2 sont deux variables et $operateur^C$ est un opérateur de comparaison désigné par un nom N,
- is^l = ($is^m, operateur^C, v$), où :
  - $is^m = N^{obj}.N^{mtd}$ est une instruction faisant appel à une méthode java prédéfinie (cf. section 4.2.2.6 Méthodes prédéfinies) ; Elle est définie par le nom de la méthode précédé du nom d'un objet de la classe à laquelle appartient la méthode,
  - $operateur^C$: est un opérateur de comparaison désigné par un nom N,
  - v : est soit une valeur de type java prédéfini, soit une variable.

4.2.2.4 Instruction itérative / Boucle

Les instructions itératives ou bien les boucles sont utilisées pour exécuter plusieurs fois les mêmes instructions jusqu'à ce qu'une condition ne soit plus satisfaite. Les trois boucles de base sont : *for*, *while* et *do-while*. La boucle for a une variante ; c'est la boucle *foreach* que nous avons utilisé pour réaliser nos transformations.

La boucle *foreach* permet de parcourir une collection d'objets ; elle se déplace respectivement du premier élément au dernier. Sa syntaxe est la suivante :

```
Collection<ObjetX> collection = ...;
for(ObjetX objetX : collection){
  // Instructions
}
```

L'interprétation de cette syntaxe est la suivante : Pour chaque objet *objetX* de type *ObjetX* dans *collection*, on exécute *Instructions*.

Les valeurs qui sont successivement affectées à *objetX* lors de l'exécution de la boucle correspondent aux différents objets présents dans *collection*. Ainsi, le type de *objetX* doit être celui des objets de *collection*.

Définition 17. Une boucle foreach b^{fe} est définie par ($is^{di}, entête^{b^{fe}}$, $corps^{b^{fe}}$), où :

- is^{di} : est une instruction permettant de déclarer et d'initialiser une variable correspondant à une collection d'objets. Elle est définie par (T^v, N^v, N^{op}, Vl^v) (cf. Définition 14),
- $entête^{b^{fe}}$: est l'entête de la boucle qui est définie par (N^{cll}, T^o , N^o), où :
  - N^{cll} est le nom de la collection qu'on veut parcourir,
  - T^o est le type des objets de cll,
  - N^o est un nom local à la boucle donné à chaque objet de cll.
- $corps^{b^{fe}}$ = $\{is_1, \ldots, is_n\}$: est un ensemble d'instructions java formant le corps de b^{fe}.

4.2.2.5 Instruction break

Une instruction Break is^{br} est une instruction qu'on peut utiliser à l'intérieur d'une boucle ; elle fait sortir de la boucle si une condition est atteinte.

4.2.2.6 Méthodes prédéfinies

En Java, il existe plusieurs méthodes prédéfinies. Nous définissons dans cette section celles que nous avons utilisées dans notre processus de transformation des contraintes :

a. isExhausted()

Il s'agit d'une méthode qui teste si un ensemble contient plus de résultats. Elle retourne "True" s'il ne reste aucun élément dans l'ensemble ; autrement, elle retourne " False". Ainsi, la valeur retournée par isExhausted() ("True" ou " False") peut être utilisée pour vérifier si un ensemble est vide ou non.

b. getInt("attribut"),*getString*("attribut"),*getFloat*("attribut")

Elles récupèrent respectivement une valeur de type "Int", "Float" et "String" de l'attribut spécifié comme paramètre.

c. size()

Elle renvoie le nombre d'éléments contenus dans un ensemble.

d. all()

Retourne tous les éléments d'un ensemble.

A partir du métamodèle proposé dans la documentation Eclipse[19] et adapté à notre contexte, nous présentons les différentes instructions présentées dans cette section à travers le métamodèle de la figure 4.4.

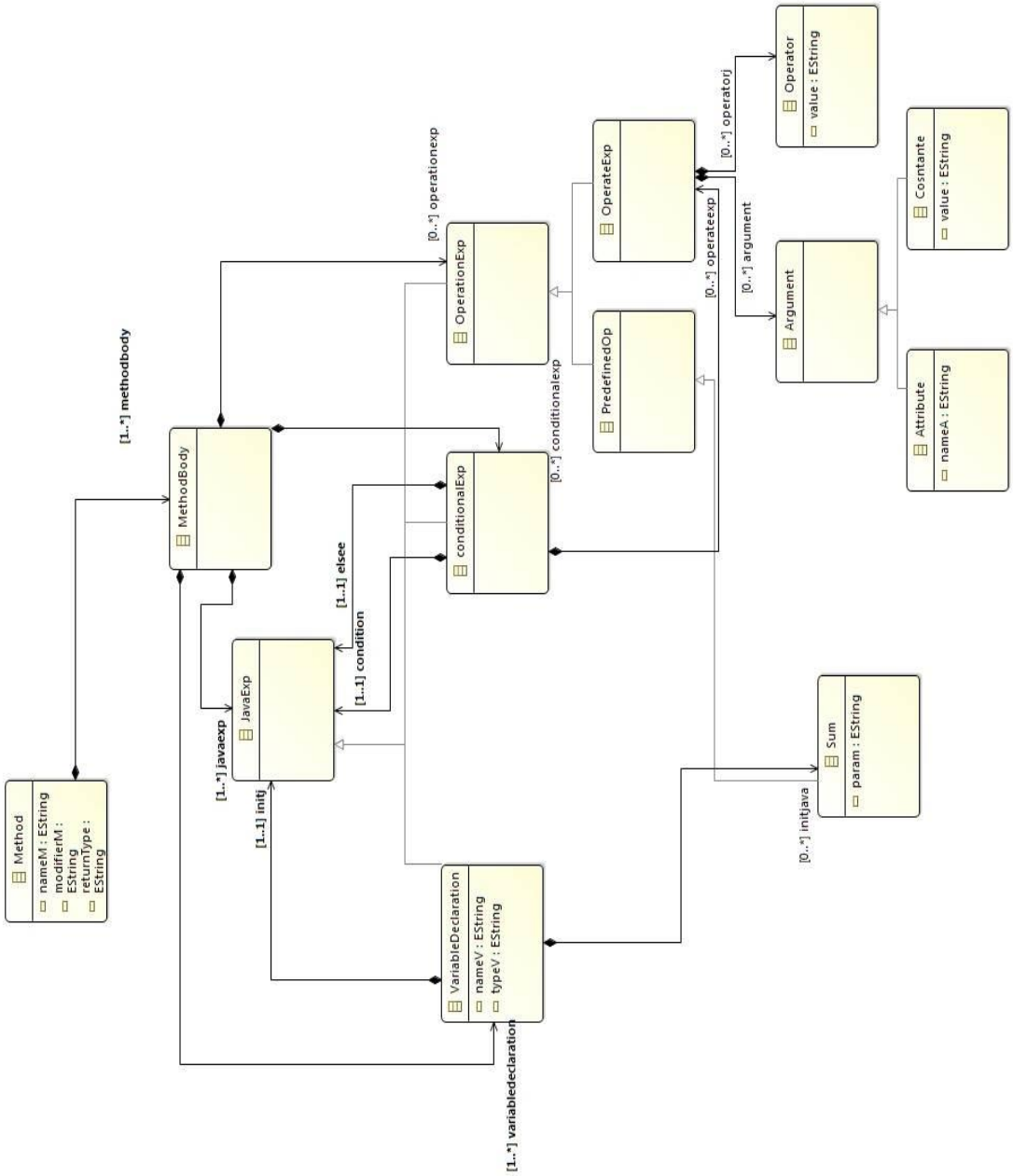


Figure 4.4 – Métamodèle Java

[19] https://help.eclipse.org/Javametamodel

4.2.3 Les règles de transformation

Afin de réaliser la transformation OCL2JavaModel, il est nécessaire de définir des règles décrivant le passage automatique d'un invariant OCL (niveau conceptuel) vers un modèle d'une méthode java (niveau logique). Ces règles de transformation sont définies dans cette section.

- **R1 :** chaque invariant i est transformé en une méthode java m, où m.N = i.N, m.C = i.C, m.V = "public" et m.T^r = "Boolean".

- **R2 :** chaque expression ex ∈ i.EX est transformée en une instruction is ∈ m.IS. Dans la section 4.2.1 (La source), nous avons considéré quatre types d'expressions OCL définissant le corps d'un invariant. Il s'agit de : (1) l'expression Let-In, (2) les expressions conditionnelles, (3) les expressions requête et (4) les expressions opération. Pour chacune de ces expressions, nous définissons ci-dessous la règle de transformation correspondante :

- **R3 :** Une expression ex^{letIn} = (ex^o, ex^u) est transformée en appliquant les règles 4 et 9.

- **R4 :** Dans une expression ex^{letIn} , l'expression ex^o = ($operande_1$, $operateur$, $operande_2$) est transformée en une instruction de déclaration et d'initialisation is^{di} = (T^v, N^v, N^{op}, Vl^v), où : $is^{di}.T^v = operande_1.T^v$, $is^{di}.N^v = operande_1.N^v$, $is^{di}.N^{op} = operateur.N$ et $is^{di}.Vl^v = operande_2$. Selon la valeur de $operande_2$, nous appliquons les règles 5, 6, 7 ou 8.

- **R5 :** Dans une expression ex^o définissant une expression ex^{letIn} , si $operande_2$ = *self.* $a^{c^{ctx}}$ alors nous transformons $operande_2$ en une méthode m^{att} permettant de retourner un attribut de la classe référencée par *self.* Cette méthode possède un nom : m^{att}.N = « getAtt » et deux paramètres : m^{att}.PR = $\{pr_1, pr_2\}$, où : $pr_1 = c^{ctx}$.N est le nom de la classe référencée par *self* et pr_2 = $a^{c^{ctx}}$.N le nom de l'attribut à récupérer. Notons qu'à cette étape, seule la signature de m^{att} est générée (i.e. son nom et la liste de ses paramètres). m^{att} sera transformée dans un deuxième temps (dans la seconde transformation du processus) en une requête de sélection écrite avec le langage d'interrogation propre à chaque SGBD NoSQL.

- **R6 :** Dans une expression ex^o définissant une expression ex^{letIn} , si $operande_2 = self.role^c.a^c$ alors nous transformons $operande_2$ en une méthode m^{att} permettant de retourner un attribut a^c d'une classe c ; celle-ci est liée à la classe référencée par *self* et son rôle est indiqué par $role^c$. m^{att} possède un nom : m^{att}.N = « getAtt » et deux paramètres : m^{att}.PR = $\{pr_1, pr_2\}$, où : $pr_1 = c$.N et pr_2 = a^c.N.

- **R7 :** Dans une expression ex^o définissant une expression ex^{letIn} , si $operande_2$ = ex^r alors nous transformons $operande_2$ en appliquant les règles 20 ou 28 selon l'opération utilisée dans l'expression requête ex^r.

- **R8 :** Dans une expression ex^o définissant une expression ex^{letIn} , si $operande_2$ = $self.role^c$ -> $size()$ alors nous transformons $operande_2$ en une instruction is^m de la forme suivante : m^{obj}.all().size(), où : les termes all() et size() correspondent à des méthodes prédéfinies (cf. Section 4.2.2.6 Méthodes prédéfinies) et m^{obj} une méthode permettant de retourner les objets d'une classe c dont le rôle est indiqué dans $role^c$. Tout comme m^{att} (voir R4), seule la signature de m^{obj} est générée à cette étape, i.e. son nom m^{obj}.N = « getObj » et ses paramètres : m^{obj}.PR = { pr_1 }, où : pr_1 = c.N. m^{obj} sera transformée plus tard (dans la seconde transformation) en une requête de sélection écrite avec le langage d'interrogation propre à chaque SGBD NoSQL. Nous appliquons le même principe pour générer la signature de m^{obj} dans une expression ex^{ite} si ex^o = $self.role^c$ -> $operation()$ (voir R13).

- **R9 :** Dans une expression ex^{letIn}, l'expression ex^u = ex^{ite} est transformée en appliquant la règle 10.

- **R10 :** une expression ex^{ite} = (ex^{cdt}, ex^{then}, ex^{else}) est transformée en une instruction de test is^{if} = ($condition$, IS^v , IS^f), où : is^{if}. $condition$ = ex^{ite}. ex^{cdt}, is^{if}. IS^v = {ex^{ite}. ex^{then}} et is^{if}. IS^f = {ex^{ite}. ex^{else}}. Nous rappelons que ex^{cdt}, ex^{then} et ex^{else} sont des expressions opération ex^o(cf. Définition 2). Selon la forme de ex^o, nous appliquons les règles 11 ou 12.

- **R11 :** Dans une expression ex^{ite} , si ex^o = ($operande_1$, $operateur$, $operande_2$) alors nous la transformons en une instruction logique is^l = (v_1, $operateur^c$, v_2), où : v_1 = N^v si $operande_1$ = N^v ou v_1 = m^{att} si $operande_1$ = $self.a^{c^{ctx}}$ (la signature de m^{att} est générée en appliquant la règle 5), $operateur^C$.N = $operateur.N$ et v_2 = $operande_2$.

- **R12 :** Dans une expression ex^{ite}, si ex^o = $self.role^c$ -> $operation()$ alors nous la transformons en une instruction is^l = (is^m, $operateur^c$, v), où : is^m est une instruction de la forme suivante : m^{obj} .isExhausted(), où : isExhausted() est une méthode prédéfinie (cf. Section 4.2.2.6 Méthodes prédéfinies). La signature de m^{obj} est générée en appliquant la règle 8, $operateur^c$.N = " == " et v = "True" si $operation.N$ = $isEmpty()$ ou v = "False" si $operation.$N = $notEmpty()$.

- **R13 :** une expression ex^{ip} = (ex_1,ex_2) est transformée en une instruction de test is^{if} = ($condition$, IS^v), où : is^{if}. $condition$ = ex^{ip}. ex_1, is^{if}. IS^v = {ex^{ip}. ex_2}. Nous rappelons que chacune des expressions ex_1 et ex_2 est soit une expression opération ex^o soit une expression requête ex^r (cf. Définition 4). Selon le cas, nous appliquons les règles 14 ou 17.

- **R14 :** Dans une expression ex^{ip}, si ex_i = ex^o , avec i ∈ [1,2], nous appliquons les règles 15 ou 16, selon la forme de ex^o.

- **R15 :** Dans une expression ex^{ip}, si ex^o = ($operande_1$, $operateur$, $operande_2$) alors nous transformons ex^o en une instruction logique is^l =

$(v_1, operateur^c, v_2)$, où : $v_1 = m_1^{att}$, $operateur^c$.N = $operateur$.N et $v_2 = m_2^{att}$ si $operande_2$ est un attribut ou $v_2 = operande_2$ si $operande_2$ est une valeur. La signature de m_1^{att} et m_2^{att} est générée en appliquant les règles 5 ou 6 en fonction des expressions utilisées ($self.a^{c^{ctx}}$ ou $self.role^c.a^c$) pour définir $operande_1$ et $operande_2$.

- **R16 :** Dans une expression ex^{ip}, si $ex^o = self.role^c \text{ -> } operation()$ alors nous appliquons la règle 12,

- **R17 :** Dans une expression ex^{ip}, si $ex_i = ex^r$, avec i ∈ [1,2], alors nous appliquons les règles 20 ou 27 selon l'opération utilisée dans l'expression ex^r.

- **R18 :** une expression requête ex^r = (ex^s , $operation$, ex^d , ex^a) est transformée en appliquant les règles 19, 20, 27 ou 29 selon la valeur de $operation$.N :

- **R19 :** Dans une expression ex^r , si $operation$.N = "$select$" alors nous transformons ex^r en une méthode m^s permettant de retourner les objets d'une classe qui vérifient ex^a. Tout comme m^{att} (voir R5) et m^{obj} (voir R8), seule la signature de m^s est générée à cette étape, i.e. son nom m^s.N = "$select$" et ses paramètres : m^s.PR = $\{pr_1, pr_2\}$, où :

  - $pr_1 = ex^d.T$: est le nom de la classe à partir de laquelle nous allons sélectionner les objets,

  - $pr_2 = is^l$: est la condition de sélection qui correspond à une instruction logique de la forme suivante : ($v_1, operateur^c$, v_2), où : $is^l.v_1 = ex^a.operande_1$, $operateur^C$ .N = $operateur$.N et $is^l.v_2 = ex^a.operande_2$.

  m^s sera transformée dans la deuxième étape du processus en une requête de sélection écrite avec le langage d'interrogation associé à chaque SGBD NoSQL.

- **R20 :** Dans une expression ex^r , si $operation$.N = "$exists$", "$forAll$" ou "$collect$" alors nous transformons ex^r en une boucle foreach b^{fe} = ($is^{di}, entête^{b^{fe}}$, $corps^{b^{fe}}$), où : $b^{fe}.is^{di}, b^{fe}.entête^{b^{fe}}$ et $b^{fe}.corps^{b^{fe}}$ sont générés en appliquant respectivement les règles 21, 23 et 23.

- **R21 :** Dans une expression ex^r , si $operation$.N = "$exists$", "$forAll$" ou "$collect$", alors $b^{fe}.is^{di}$ est définie ainsi : $is^{di}.T^v$ = "Collection", $is^{di}.N^v$ = "resultat", $is^{di}.N^{op}$ = "=", $is^{di}.Vl^v$ = "m^{obj}" (voir R8 pour la définition de m^{obj}), où : m^{obj}.N = « getObj » et m^{obj}.PR = $\{pr_1\}$, où $pr_1 = ex^d.T$. Notons qu'en fonction du SGBD NoSQL utilisé, " Collection " sera transformée au moment de la génération du code (la deuxième étape du processus) vers le type qui permet de stocker le résultat de la requête

correspondant à m^{obj}. Sous Cassandra par exemple, il s'agit du type "ResultSet".

- **R22 :** Dans une expression ex^r, si $operation$.N = "$exists$", "$forAll$" ou "$collect$", alors $b^{fe}.entête^{b^{fe}}$ est définie ainsi : $entête^{b^{fe}}.N^{cll}$ = "resultat", $entête^{b^{fe}}.T^o$ = " Object " et $entête^{b^{fe}}.N^o$ = " object ". Comme "Collection" (voir R21), "Object" sera transformée dans la deuxième étape du processus vers le type des éléments retournés par m^{obj}. Sous MongoDB par exemple, $entête^{b^{fe}}.T^o$ va correspondre à "Document".

- **R23 :** Dans une expression ex^r, si $operation$.N = "$exists$" alors $b^{fe}.corps^{b^{fe}} = \{is^d, is^{if}\}$, avec is^d une instruction de déclaration d'une variable de type booléen représentant le retour de l'opération $exists$ et is^{if} une instruction de contrôle permettant de vérifier qu'il existe au moins un objet de ex^s vérifiant ex^a. is^d et is^{if} sont générées en appliquant respectivement les règles 24 et 25.

- **R24 :** Dans une expression ex^r, si $operation$.N = "$exists$" ou "$forAll$" alors $corps^{b^{fe}}.is^d$ est définie ainsi : $is^d.T^v$ = "Boolean", $is^d.N^v$ = "out".

- **R25 :** Dans une expression ex^r, si $operation$.N = "$exists$" alors $corps^{b^{fe}}.is^{if}$. est définie ainsi :

  - $is^{if}.condition$ est générée en appliquant la R27.

  - $is^{if}.IS^v = \{is^l, is^{br}\}$, avec is^l une instruction logique définie par $(v_1, operateur^c, v_2)$, où : $is^l.v_1$ = "out" $is^l.operateur^c$ = "=" et $is^l.v_2$ = "True" et is^{br} une instruction Break (cf. 4.2.2.5),

  - $is^{if}.IS^f = \{is^l\}$, avec is^l une instruction logique définie par $(v_1, operateur^c, v_2)$, où : $is^l.v_1$ = "out" $is^l.operateur^c$ = "=" et $is^l.v_2$ = "False".

- **R26 :** Dans une expression ex^r, si $operation$.N = "$exists$" ou "$forAll$" alors $is^{if}.condition$ définissant $corps^{b^{fe}}$ correspond à une instruction logique is^l de la forme suivante : $(is^m, operateur^c, v)$, où :

  - $is^m.N^{obj}$ = "object" et $is^m.N^{mtd}$ est généré en fonction du type de l'attribut a indiqué dans $ex^a.operande_1$ (cf. Section 4.2.2.6). Par exemple, si a.T = "Int", alors $is^m.N^{mtd}$ = getInt("$ex^a.operande_1$"). Nous appliquons le même principe pour générer $corps^{b^{fe}}.is^m$ si $operation$.N = "$collect$" (voir R29),

  - $is^l.operateur^c = ex^a.Operateur$,

  - $is^l.v = ex^a.operande_2$.

- **R27 :** Dans une expression ex^r , si $operation$.N = "$forAll$" alors nous transformons ex^r en une boucle foreach b^{fe} = (is^{di}, $entête^{b^{fe}}$, $corps^{b^{fe}}$), où :

  - $b^{fe}.is^{di}$ et $b^{fe}.entête^{b^{fe}}$ sont générés en appliquant respectivement les règles 21 et 22.

  - $b^{fe}.corps^{b^{fe}}$ = { is^d, is^{if} } , avec is^d une instruction de déclaration d'une variable de type booléen représentant le retour de l'opération $forAll$ et is^{if} une instruction de contrôle permettant de vérifier qua chaque objet de ex^s vérifie $ex^a.is^d$ et is^{if} sont générées en appliquant respectivement les règles 24 et 28.

- **R28 :** Dans une expression ex^r , si $operation$.N = "$forAll$" alors $corps^{b^{fe}}.is^{if}$ est définie ainsi :

  - $is^{if}.condition$ est générée en appliquant la règle 26,

  - is^{if} . IS^v = { is^l }, avec is^l une instruction logique définie par (v_1, $operateur^c$, v_2), où : $is^l.v_1$ = "out" $is^l.operateur^c$ = "=" et $is^l.v_2$= "True".

  - is^{if} . IS^f = { is^l, is^{br} }, avec is^l une instruction logique définie par (v_1, $operateur^c$, v_2), où : $is^l.v_1$ = "out" $is^l.operateur^c$ = "=" et $is^l.v_2$= "False" et is^{br}une instruction Break (cf. Section 4.2.2.5).

- **R29 :** Si $operation$.N = "$collect$" alors nous transformons ex^r en une boucle foreach b^{fe} = (is^{di}, $entête^{b^{fe}}$, $corps^{b^{fe}}$), où :

  - $b^{fe}.is^{di}$ et $b^{fe}.entête^{b^{fe}}$ sont générés en appliquant respectivement les règles 21 et 22.

  - $corps^{b^{fe}}$ = { is^m }, avec is^m une instruction logique générée en en appliquant la règle 26.

La figure 4.5 suivante présente une vue synthétique des relations d'utilisation entre ces règles de transformation

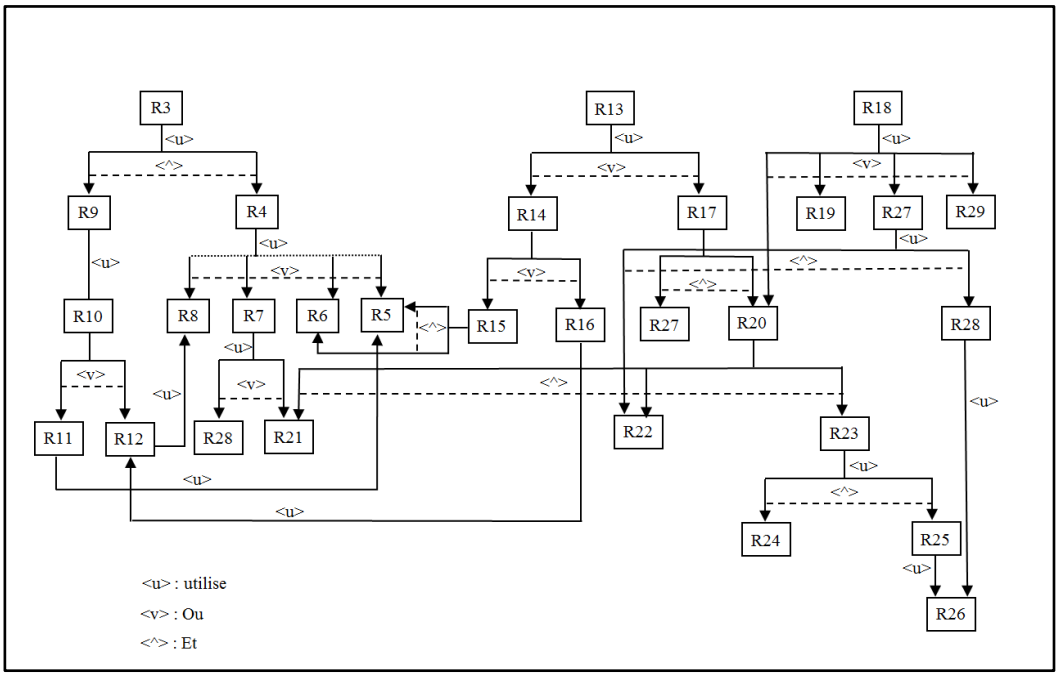


Figure 4.5 – Règles de transformation conceptuel -> logique

4.3 Transformation JavaModel2JavaCode

JavaModel2JavaCode est la deuxième étape dans notre processus de transformation des contraintes. Il s'agit d'une transformation M2T (Model2Text) qui prend en entrée chaque modèle de méthode java généré dans l'étape précédente (OCL2JavaModel) et fournit en sortie le code de vérification de la contrainte OCL correspondante. Ce code dépend des caractéristiques techniques propres au SGBD NoSQL choisi par l'utilisateur. La figure 4.6 montre les composants de cette transformation.

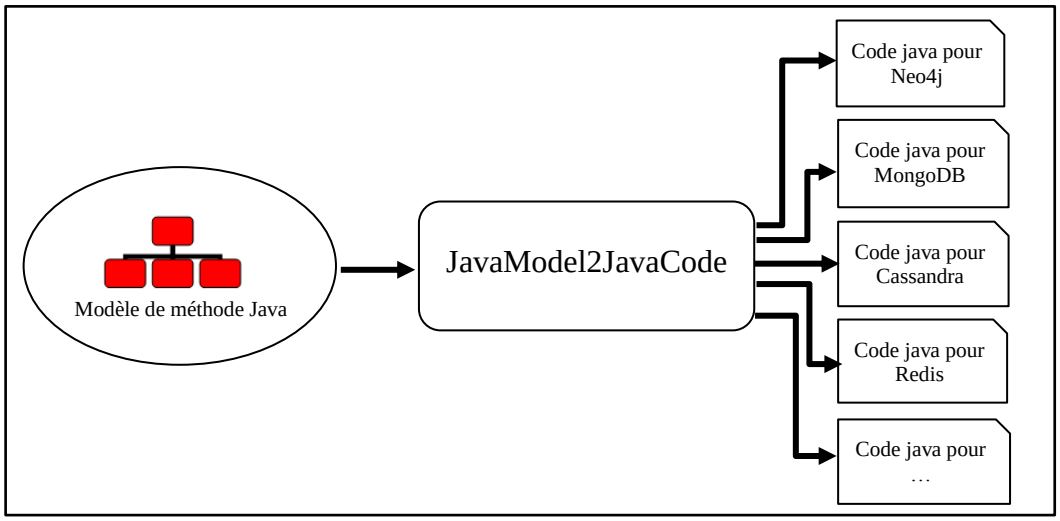


Figure 4.6 – Transformation JavaModel2JavaCode

Le passage du modèle de méthode java vers le code correspondant est assuré par une transformation formalisée en MOFM2T. Cette transformation ainsi que le langage utilisé pour la formaliser sont présentés dans le chapitre 5 « Expérimentation et Validation ».

4.4 Conclusion

Synthèse

Ce chapitre présente la deuxième partie de notre contribution portant sur la traduction automatique des contraintes OCL. Cette partie a pour objet d'apporter un complément au processus Object2NoSQL de transformation d'un DCL décrit dans le chapitre précédent. Ce dernier processus ne considère que des contraintes simples telles que les types de données et l'unicité des identifiants. Afin de le compléter, nous avons proposé le processus OCL2Java qui intègre des contraintes OCL plus complexes.

Quatre types d'expressions OCL sont supportés par notre processus. Il s'agit des expressions requête, des expressions opération, des expressions conditionnelles et des expressions de définition. La traduction de ces expressions s'appuie sur les éléments suivants :

- Utilisation de Trois niveaux de modélisation : il s'agit des niveaux conceptuel, logique et physique. Le premier décrit les contraintes associées à un DCL en utilisant le standard OCL. Le deuxième introduit un modèle pivot de méthode java pour chaque contrainte. Ce modèle fait abstraction des caractéristiques techniques propres aux SGBD NoSQL. A partir de chaque modèle défini au niveau logique, plusieurs codes java peuvent être générés au niveau physique. Chacun d'eux est spécifique à un SGBD NoSQL et

permet de vérifier la contrainte correspondante suite à une mise à jour de la BD.

- Formalisation des expressions OCL : nous avons formalisé les expressions OCL supportées par notre processus en nous appuyant sur les définitions informelles figurant dans la documentation de l'OMG.
- Définition des Métamodèles : nous avons proposé un métamodèle OCL en nous référant aux spécifications de l'OMG et un métamodèle Java en adaptant celui proposé dans la documentation Eclipse à notre contexte ; dans le métamodèle Java, nous nous sommes limités aux éléments nécessaires à la transformation des expressions OCL considérées comme source.
- Formalisation des règles de transformation M2M et M2T : deux types de règles de transformation sont utilisés par notre processus OCL2Java. Le premier M2M assure le passage conceptuel vers logique. Nous l'avons formalisé avec le standard QVT. Le second M2T correspond au passage logique vers physique et nous l'avons exprimé avec le standard MOFM2T.

Les travaux de ce chapitre ont été présentés dans les publications : [Abdelhedi et al., 2017d] et [Abdelhedi et al., 2018c].

Positionnement

Au regard des travaux [Li et al., 2014] et [Daniel et al., 2016], les plus proches à notre solution de transformation des modèles conceptuels UML, le processus décrit dans ce chapitre apporte une avancée significative.

En effet le processus proposé dans [Li et al., 2014] transforme un DCL d'UML en un modèle NoSQL propre au SGBD HBase, mais il ne tient pas compte des contraintes d'intégrité. Par ailleurs, dans le travail [Daniel et al., 2016], un processus de transformation des contraintes OCL est proposé. Mais ce processus décrit la correspondance entre les expressions OCL et des requêtes rédigées dans le langage Germlin. Il s'agit d'un langage de requêtes spécifique à un SGBD orienté-graphes mais incompatible avec les autres types de SGBD NoSQL (colonnes, documents et clé-valeur).

Ainsi notre processus de transformation des contraintes :

- généralise la démarche d'implantation des structures de données et des contraintes d'intégrité en utilisant l'architecture MDA
- s'appuie sur une décomposition à trois niveaux : conceptuel, logique et physique ; le niveau logique étant compatible avec les quatre types de SGBD NoSQL. Ce modèle pivot fait abstraction du langage de requête propre à chaque système NoSQL ; il sera donc aisé, dans un second temps, de le traduire en un code destiné à être exécuté sur un SGBD NoSQL donné.
- utilise le langage Java qui est interfacé avec la plupart des SGBD NoSQL.

Chapitre 5 : Expérimentation et Validation

Le présent chapitre décrit un prototype qui a permis de vérifier la faisabilité de nos propositions présentées dans les chapitres 3 et 4. Ce prototype est composé de deux modules qui transforment respectivement un DCL d'UML en un modèle NoSQL et ses contraintes en des méthodes java ; celles-ci assureront la cohérence de la BD lors des mises à jour. Les entrées de ce prototype, i.e. le DCL et les contraintes, sont fournies par l'utilisateur.

Ce chapitre présente également une évaluation que nous avons effectuée dans une entreprise afin de montrer la pertinence de nos propositions. Cette évaluation porte essentiellement sur la comparaison entre :

- Le résultat retourné par notre processus automatisé et celui obtenu par le processus manuel,
- Le temps de réalisation de chaque processus.

Plan du chapitre. La section 5.1 décrit les outils techniques utilisés pour implanter notre prototype. La section 5.2 présente une description du prototype. Les sections 5.2.1 et 5.2.2 détaillent les modules composant le prototype ; pour chaque module, nous décrivons les métamodèles, les modèles et les règles de transformation. La section 5.3 compare nos transformations automatiques avec celles mises en œuvre manuellement.

5.1 Outils d'implantation

Dans cette section, nous présentons succinctement les techniques que nous avons utilisées pour mettre en place l'environnement expérimental. Etant donné que notre approche est basée sur MDA, nous avons besoin d'une infrastructure adaptée à la métamodélisation, la modélisation et les transformations M2M (Model-To-Model) et M2T (Model-To-Text). C'est ainsi que nous avons retenu EMF (Eclipse Modeling Framework) [Budinsky et al., 2004] comme plateforme d'implantation.

EMF fournit un ensemble d'outils destinés à introduire une approche de développement dirigée par les modèles au sein de l'environnement Eclipse[20]. Ces outils apportent trois fonctionnalités principales. La première est la définition d'un métamodèle représentant les concepts utilisés par l'utilisateur, la deuxième est la création des modèles instanciant ce métamodèle et la troisième fonctionnalité correspond à la transformation de modèle vers modèle et de modèle vers texte.

Dans ce qui suit, nous présentons les outils d'EMF utilisés pour mettre en œuvre notre prototype.

5.1.1 Ecore

EMF utilise le langage de métamodélisation Ecore[21] pour la définition et la vérification des métamodèles. Ce langage se concentre sur la spécification structurelle des métamodèles [Combemale, 2008]. Il s'agit d'une implantation d'un sous ensemble du standard MOF (Meta Object Facility) de l'OMG[22]. La figure 5.1 présente les principaux concepts de métamodélisation utilisés dans Ecore.

S'inspirant de l'approche orientée-objet, le langage Ecore repose sur la notion de package (EPackage), de classe (EClass), d'attribut (EAttribute), de lien de référence (EReference), de type de données (EDataType), et d'énumération EEnum.

A la racine de la hiérarchie de ces concepts, se trouve EPackage. Il est composé de EClass et de EDataType. EClass représente les caractéristiques structurelles d'un objet à travers un ensemble de EAttribute et de EReference. EReference permet de définir une association entre deux classes du métamodèle. EDataType représente le type d'un attribut qui peut être soit un type de données prédéfini tels

[20] Projets EMF et SOA Tools Platform de Eclipse. Disponible www.eclipse.org/stp/ and www.eclipse.org/emf/

[21] ECore. The eclipse modeling framework project home page. http ://www.eclipse.org/emf

[22] Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Core Specification, January 2006. Final Adopted Specification

que String, Integer et Boolean, soit un type défini par l'usager en utilisant le type énumération de valeurs EEnum.

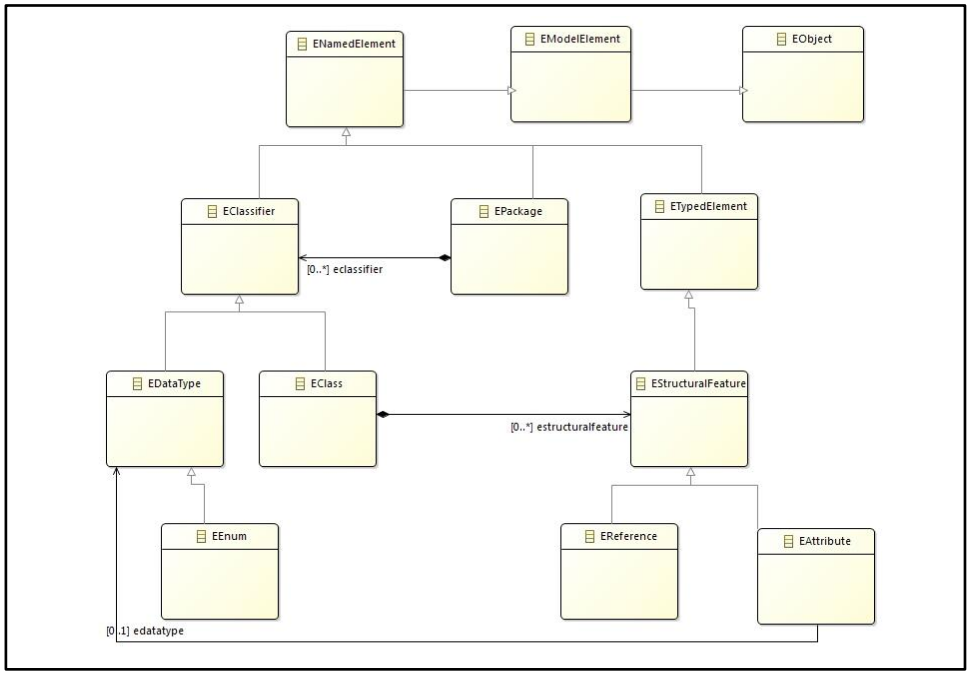


Figure 5.1 – Extrait simplifié du métamodèle Ecore [Budinsky et al., 2004]

5.1.2 XMI

EMF intègre la norme XMI (XML Metadata Interchange)[23] pour représenter des modèles en XML. Il s'agit d'un format de représentation de modèles qui indique comment ces derniers peuvent être traduits sous la forme de documents XML. Le principe de fonctionnement de XMI est de générer automatiquement des grammaires XML (des DTD) à partir d'un métamodèle (dans notre cas il s'agit d'un métamodèle Ecore) ; ceci permet de représenter les modèles instances sous forme de documents XML. Pour ceci, XMI s'appuie sur l'analogie entre les modèles et leur métamodèle d'une part, et les documents XML et leur DTD d'autre part. Cette analogie réside dans le fait qu'un métamodèle et un DTD sont identiques en ce qu'ils définissent respectivement les structures des modèles et des documents XML. La figure 5.2 illustre le principe de fonctionnement de XMI.

[23] OMG, XML Metadata Interchange (XMI) Specification. (http ://www.omg.org/technology/documents/formal/xmi.htm)

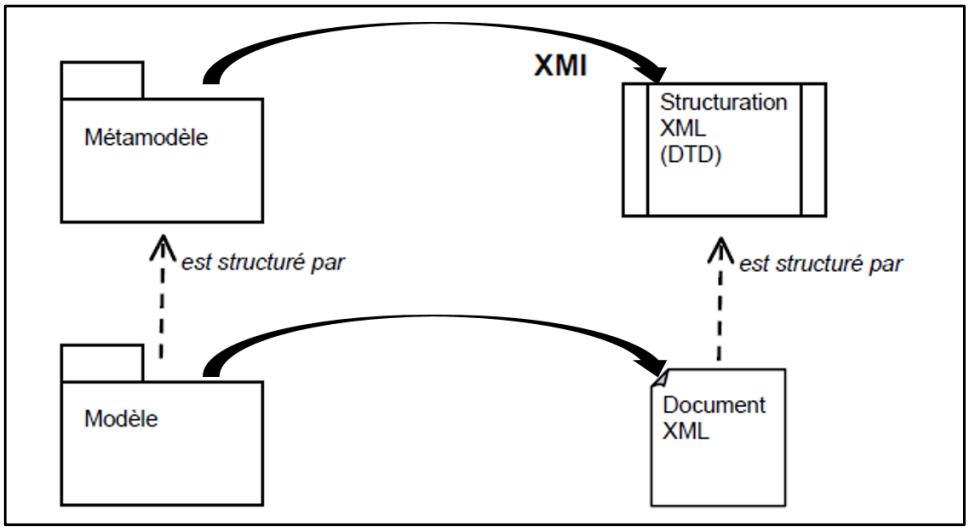


Figure 5.2 – Principe de fonctionnement de XMI [Blanc and Salvatori, 2011]

5.1.3 QVT

Les transformations de modèles sont au centre d'une approche MDA. Il est donc nécessaire de disposer d'un langage permettant d'accomplir les transformations de type Model-To-Model (M2M). Pour ce faire, l'OMG a proposé le standard QVT[24] (Query, View, Transformation). Il s'agit d'un langage de haut niveau intégré dans l'environnement EMF pour exprimer les transformations de modèles. Il permet d'établir le passage automatique entre les différentes phases d'une approche MDA.

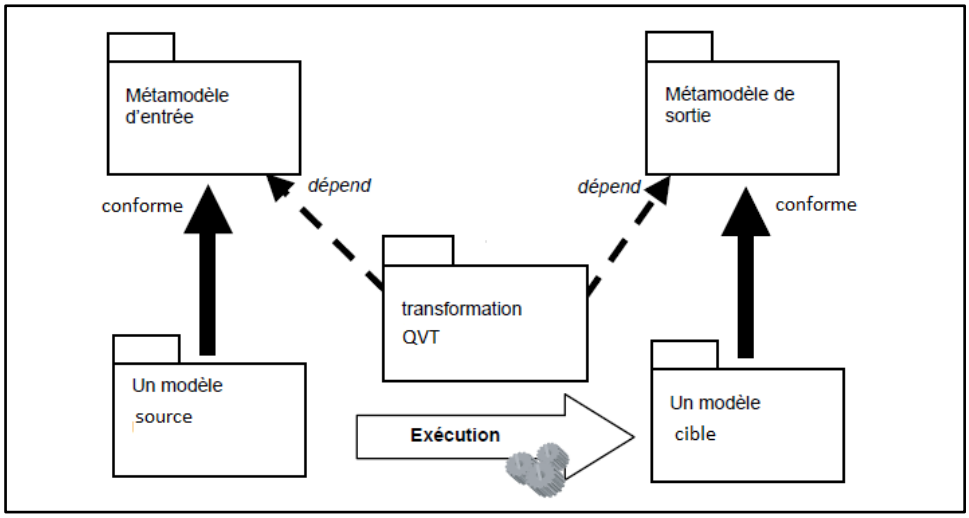


Figure 5.3 – Principe d'une transformation QVT [Blanc and Salvatori, 2011]

[24] OMG, Q. (2009). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, OMG (2008).

Le principe d'une transformation QVT consiste à exprimer des règles de correspondance structurelles entre les métamodèles source et cible de cette transformation ; ces règles seront ensuite interprétées par un moteur qui réalisera la transformation. Autrement dit, une transformation QVT prend en entrée un modèle source et renvoie en sortie un modèle cible. Chacun des modèles source et cible est conforme à un métamodèle. La figure 5.3 illustre ce principe.

5.1.4 MOFM2T

Un autre type de transformation utilisé dans un contexte MDA concerne la génération de textes à partir de modèle. Il s'agit de la transformation Model-To-Text (M2T). Contrairement à une transformation M2M, une transformation M2T ne nécessite pas obligatoirement la définition d'un métamodèle cible [Blanc and Salvatori, 2011]. Autrement dit, avec une transformation M2T, un document textuel (code source ou documentation) peut être généré à partir d'un modèle sans avoir à définir le métamodèle du texte.

Pour établir ce type de transformation, nous avons utilisé le projet Acceleo[25] qui est un générateur de code produisant du texte à partir de modèles EMF. Le langage utilisé par Acceleo est une implantation du langage MOFM2T[26] (MOF Model to Text) standardisé par l'OMG. Le principe de ce langage est de décrire la correspondance entre les éléments du modèle source et les éléments du texte. Pour ceci, MOFM2T utilise une approche par « Templates », où Template correspond à un texte contenant des espaces réservés dans lequel seront mis des informations extraites du modèle source. Ces espaces correspondent généralement à des expressions spécifiées sur les éléments qui seront utilisées dans le modèle et qui auront pour but de sélectionner et d'extraire des informations de ce modèle. Ces informations sont par la suite transformées en fragment de code en utilisant la bibliothèque de définition du langage cible.

Nous avons utilisé les langages QVT et MOFM2T pour transformer des modèles ; ils permettent respectivement des transformations M2M et des transformations M2T. Le choix de ces langages repose sur les deux points suivants :

- Il s'agit de deux normes édictées par l'OMG,
- La plateforme d'implantation EMF que nous avons utilisée, dispose de ces deux formalismes.

5.2 Description du prototype

L'objectif du prototype réalisé est de :

[25] http ://www.eclipse.org/acceleo/.
[26] https://www.omg.org/spec/MOFM2T/About-MOFM2T/

- Transformer un DCL d'UML en un modèle physique NoSQL qui comporte : (1) les éléments nécessaires à l'implantation de la BD et (2) un ensemble de directives d'assistance spécifiques à un SGBD NoSQL,

- Traduire les contraintes associées au DCL en des méthodes Java. Ces dernières seront appelées suite à une mise à jour de la BD afin de vérifier les contraintes correspondantes.

Notre prototype ne produit pas le modèle conceptuel (DCL + Contraintes) d'entrée. Ce dernier est établi par un décideur (structures des données simples) ou un informaticien (cas plus complexes) suite à une étape d'analyse des besoins. Ainsi, le démarrage du prototype nécessite deux entrées qui sont fournies par l'utilisateur. Il s'agit :

- Du diagramme de classes qui décrit des structures d'objets métiers,

- Des contraintes d'intégrité exprimées en OCL,

Comme illustré dans la figure 5.4, le prototype se compose des deux modules ; le module de transformation du DCL : Object2NoSQL et le module de transformation des contraintes : OCL2Java. Le premier consiste en un ensemble de règles de transformation de type Model-To-Model ; il restitue, à partir du DCL d'entrée, un modèle NoSQL et un ensemble de directives d'assistance spécifiques au SGBD choisi. Le second module a pour rôle de compléter le premier, en intégrant des contraintes qui ne sont pas prises en compte dans le modèle généré. Les sections 5.2.1 et 5.2.2 ci-dessous présentent respectivement l'architecture de chacun de ces modules.

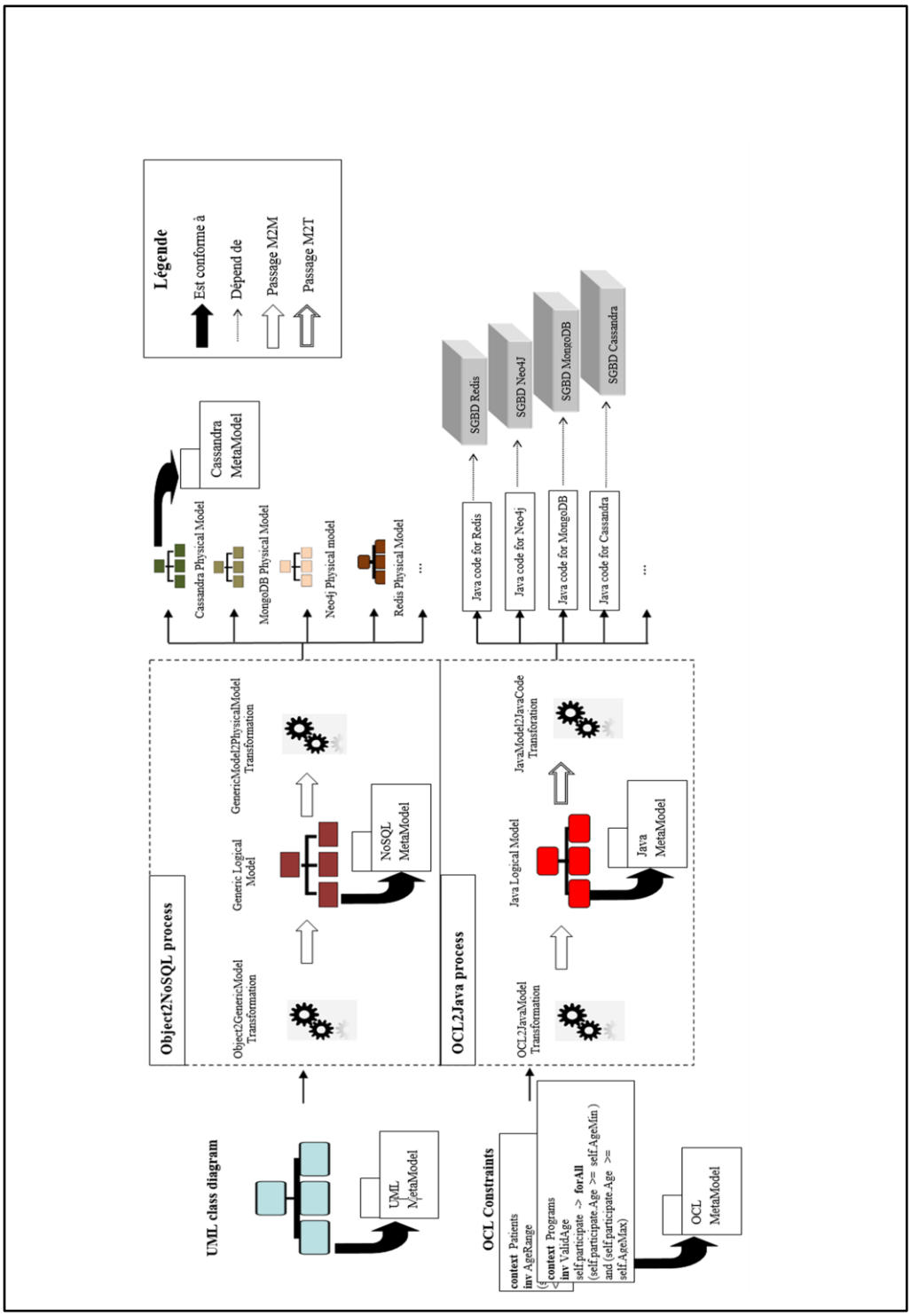


Figure 5.4 – Architecture du prototype

5.2.1 Module Object2NoSQL

Ce module est chargé de transformer un DCL d'UML en un modèle physique NoSQL. Conformément à la justification donnée dans la section 3.1 du chapitre 3, nous introduisons un niveau logique intermédiaire entre les niveaux conceptuel et physique. Ce niveau vise à décrire techniquement la structure des données sans préciser les caractéristiques propres à chaque SGBD. Autrement dit, le module Object2NoSQL s'exécute en deux étapes successives : conceptuel > logique puis logique > physique. Le passage d'un modèle à un autre se fait en utilisant des transformations de type M2M formalisées en QVT. La figure 5.5 montre les composantes du module Object2NoSQL.

Nous détaillons par la suite ce module en précisant l'entrée, la sortie, les transformations et l'implantation.

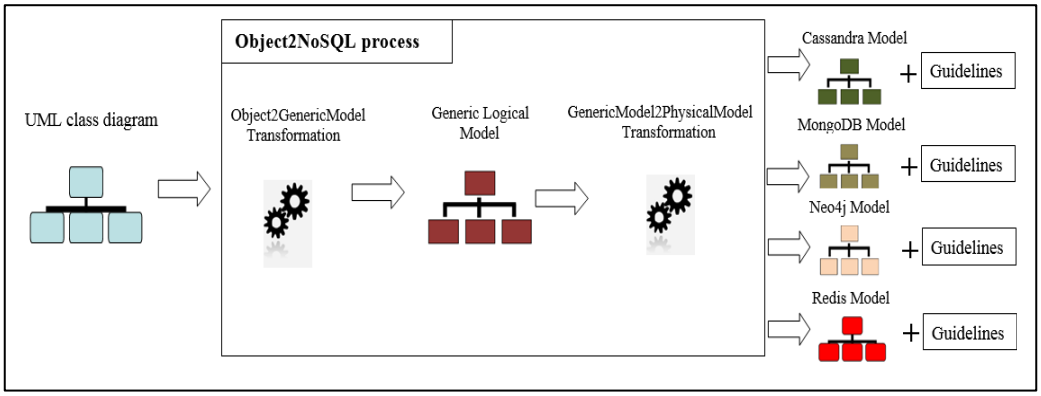


Figure 5.5 – Module Object2NoSQL

Entrée

L'entrée du module Object2NoSQL est un diagramme de classes du langage UML. Le choix du formalisme UML est justifié dans la section 1.3 du chapitre 1. L'utilisateur fourni le DCL d'entrée en instanciant le métamodèle du PIM conceptuel que nous avons proposé dans le chapitre 3 (cf. figure 3.6). Ce métamodèle montre les principaux éléments composant un modèle de données d'UML ainsi que leurs caractéristiques structurelles.

Sortie

Le résultat final retourné par le module Object2NoSQL est un modèle physique NoSQL (colonnes, documents, graphes ou clé-valeur) qui comporte :

- Un modèle de données contenant les éléments nécessaires à l'implantation de la BD NoSQL,
- Un ensemble de directives d'assistance qui donnent les modalités d'utilisation des attributs et d'implantation des relations selon les techniques inhérentes au SGBD NoSQL choisi.

Evidemment, pour une sortie donnée (modèle physique NoSQL), il est nécessaire d'enregistrer ses paramètres, c'est-à-dire son métamodèle et les règles de transformation qui permettent de le générer. Pour illustrer notre travail, nous avons choisi de produire des modèles physiques aux caractéristiques bien distinctes, ceci en utilisant les SGBD Cassandra, MongoDB, Neo4j et Redis. Si l'utilisateur souhaite utiliser un autre SGBD, le module doit être complété par l'ajout des nouveaux paramètres spécifiques à ce système.

Transformations

Le module Object2NoSQL est composé de deux transformations : Object2GenericModel et GenericModel2PhysicalModel. La première étape traduit le DCL d'entrée en un modèle générique NoSQL conforme au métamodèle du PIM logique que nous avons proposé dans le chapitre 3 (cf. figure 3.7). La seconde prend en entrée le modèle générique et génère les éléments nécessaires à l'implantation de la BD ainsi que l'ensemble de directives d'assistance propres au SBGD NoSQL choisi. Ces deux transformations sont réalisées par un ensemble de règles M2M formalisées en QVT. Si le choix du SGBD est fait au départ, les 2 transformations s'enchainent sans rupture.

Notons que le module de transformation du DCL est capable de transformer ce dernier en un modèle physique pour l'une des plateformes d'implantation NoSQL : colonnes, documents, graphes et clé-valeur. Comme nous l'avons mentionné ci-dessus, nous avons illustré dans ce mémoire notre contribution par un seul représentant de chaque type de SGBD NoSQL : Cassandra pour l'orienté-colonnes, MongoDB pour l'orienté-documents, Neo4j pour l'orienté-graphes et Redis pour l'orienté clé/valeur. Nos travaux présentés dans les publications [Abdelhedi et al., 2016c] et [Abdelhedi et al., 2018b] montrent que notre solution est aussi compatible avec d'autres SGBD NoSQL comme HBase (orienté-colonnes) et CouchDB (orienté-documents).

Implantation

La figure 5.6 montre les différentes étapes d'implantation du module de transformation du DCL : Object2NoSQL. Cette implantation nécessite la définition préalable d'un ensemble de métamodèles et de règles de transformation de type M2M.

Tout d'abord, nous avons créé les métamodèles ECORE qui sont illustrés par la figure 5.7. Il s'agit des métamodèles du PIM conceptuel (a), PIM logique (b), PSM Cassandra (c), PSM MongoDB (d), PSM Neo4j (e) et PSM Redis (f). Ces métamodèles décrivent respectivement la structure d'un DCL d'UML, du modèle générique NoSQL et des modèles physiques de Cassandra, MongoDB, Neo4j et Redis. Une description détaillée de ces métamodèles est donnée dans le chapitre 3 (cf. figures 3.11, 3.12, 3.13 et 3.14).

Ensuite, nous avons utilisé le langage QVT pour implanter les règles de transformation assurant les deux passages : conceptuel vers logique et logique

vers physique. Les figures 5.8 et 5.9 montrent des extraits du script QVT correspondant à chaque passage ; les commentaires figurant dans les scripts indiquent les règles utilisées.

Finalement, nous avons testé le module en suivant les étapes suivantes :

- **Etape 1 :** Instancier le métamodèle du PIM conceptuel pour produire le DCL (cf. figure 5.10) de l'application médicale décrite dans le chapitre 1. Ce DCL est stocké sous la forme d'un fichier XMI,

Le résultat d'exécution du script Object2GenericModel sur le DCL d'entrée est illustré dans la figure 5.11. Il s'agit d'un modèle générique NoSQL contenant principalement des tables reliées entre elles par des relations binaires.

- **Etape 2 :** Préciser le nom du SGBD à utiliser (MongoDB par exemple),

- **Etape 3 :** Choisir une solution d'implantation des relations parmi celles proposées par le module (imbrication ou références).

- **Etape 4 :** Exécuter le script GenericModel2Physical sur le modèle générique retourné à l'étape 1.

Le résultat final est illustré dans la figure 5.12. Il s'agit d'un modèle de données contenant les éléments nécessaires à l'implantation de la BD (a) et d'un ensemble de directives d'assistance spécifiques au SGBD MongoDB (b).

Notons que nous présentons uniquement un extrait des métamodèles et des scripts QVT utilisés par le module Object2NoSQL.

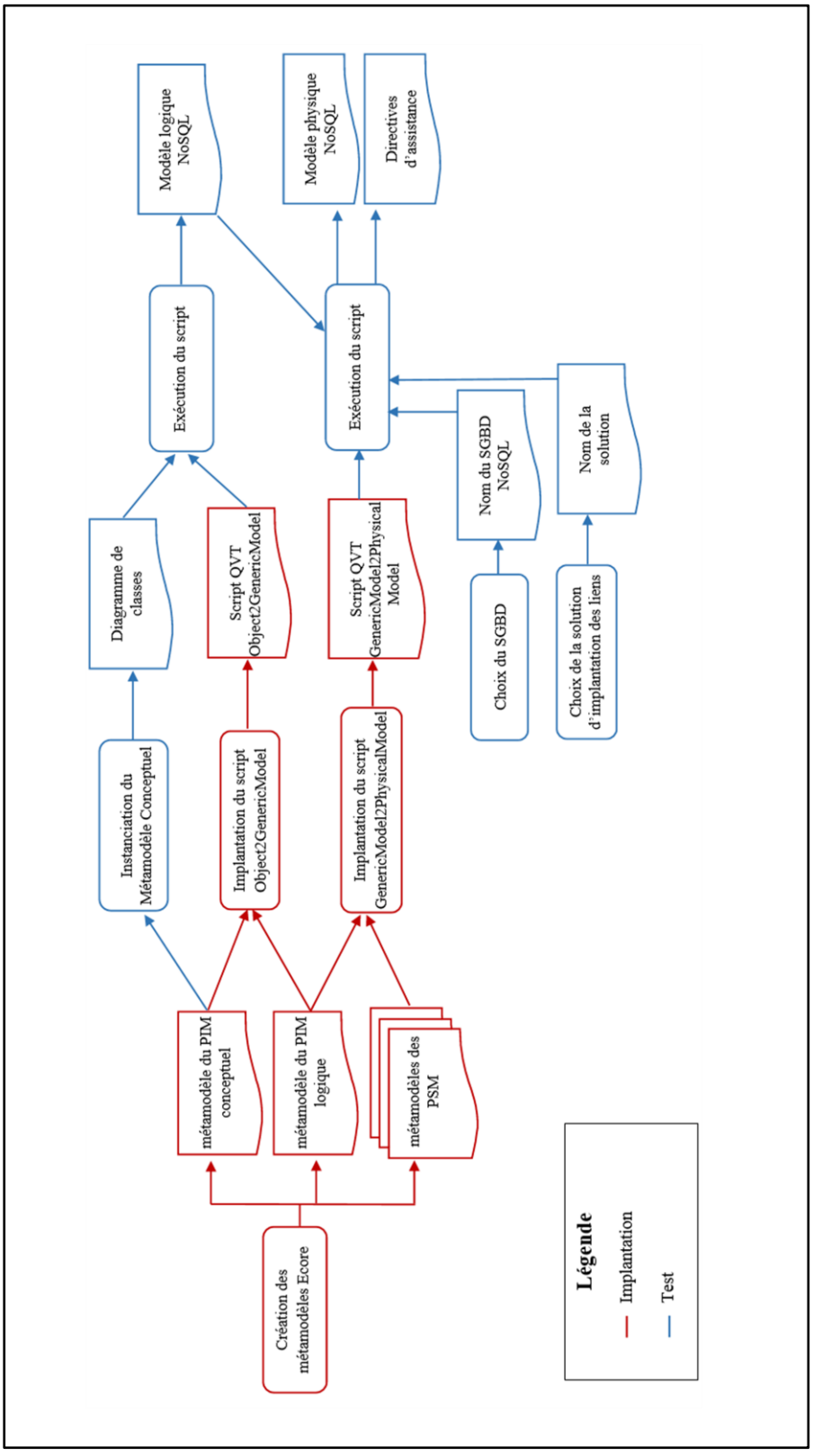


Figure 5.6 – Etapes d'implantation du module Object2NoSQL

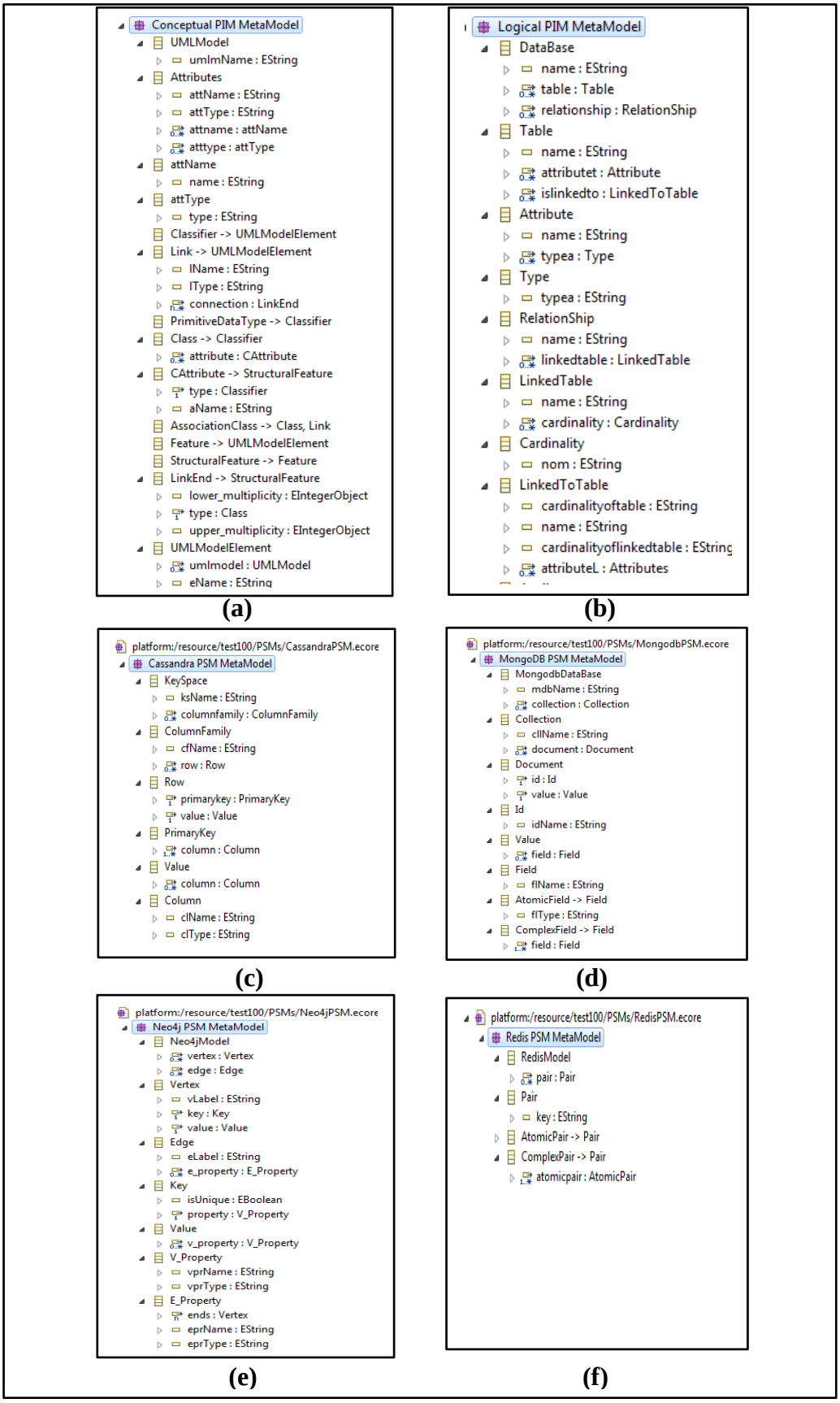


Figure 5.7 – Métamodèles Ecore du module Object2NoSQL

```
modeltype conceptualPIM uses "http://UMLClassDiagram.com";
modeltype logicalPIM uses "http://GenericLogicalModel.com";
transformation TransformationUmlToColumnsOrientedModel(in Source: conceptualPIM, out Target: logicalPIM);
main() {Source.rootObjects()[ClassDiagram] -> map toDataBase();}
mapping ClassDiagram::toDataBase():DataBase{name := self.name ; table:=self.classes -> map toClass();
relationship:=self.links -> toRelationship();}
-- Transforming Class to Table
mapping conceptualPIM ::Class::toClass():logicalalPIM::Table{name:=self.name;attributet:=self.attributec -> map
toAttribute();}
-- Transforming Attribute to Column
mapping conceptualPIM ::Attribute::toAttribute():logicalPIM::Attribute{if (self.attType="Oid"){aName:=self.attName;
aType:="Rid";}endif; {aName:=self.attName ; aType:=self.attType;}}
-- Transforming Binary Link to Relationship
mapping conceptualPIM ::Link::toRelationship():logicalPIM::RelationShip{name:=self.name ; linkedtable:=self.linkedclass ->
map toLinkedTable() ; typer:=self.typel -> map toRelationType();}
mapping conceptualPIM ::LinkedClass::toLinkedTable():logicalPIM::LinkedTable{name:=self.name ;
cardinality:=self.cardinality -> map toCardinalities();}
mapping conceptualPIM ::TypeL::toRelationType():logicalPIM::TypeR{typer:=self.typel;}
mapping conceptualPIM ::Cardinality::toCardinalities():logicalPIM::Cardinality{nom:=self.nom;}
-- Transforming n-ary link to Table
mapping conceptualPIM ::Class::toClass1():logicalPIM::Tables{tName:=self.cName;key:=self.ident -> map
toId();columns:=self.attributes -> map toRefColumns();}
mapping conceptualPIM ::Attributes::toRefColumns():logicalPIM::Columns{aName:=self.attName;aType:='Rid';}
-- Transforming Association Class to Table
mapping conceptualPIM ::Class::toClass2():logicalPIM::Tables{tName:=self.cName;key:=self.ident -> map
toId();columns:=self.attributes -> map toRefColumns();columns:=self.attributes -> map toColumns();}
mapping conceptualPIM ::Ident::toId():logicalPIM::Key{kname:=self.iName -> map tokName();ktype:=self.itype -> map
```

Figure 5.8 – Script QVT de la transformation Object2GenericModel

```
modeltype LogicalPIM uses "http://GenericLogicalModel.com";
modeltype CassandraPSM uses "http://CassandraModel.com";
transformation TransformationGenericModelToCassandraModel (in Source: LogicalPIM, out Target: CassandraPSM);
main() {Source.rootObjects()[DataBase] -> map toKeySpace();}
-- Transforming DataBase to KeySpace
mapping DataBase::toKeySpace():KeySpace{name := self.name;columnsfamily:=self.table -> map toColumnsFamily();}
-- Transforming Table to Columns-Family
Mapping LogicalPIM:: Table::toColumnsFamily():CassandraPSM::ColumnsFamily
{name:=self.name;column:=self.attributet -> map toColumn();referencecolumn:=self.islinkedto -> map toReferenceColumn();}
-- Transforming Attribute to Column
mapping LogicalPIM ::Attribute::toColumn():CassandraPSM::Column {name:=self.name;type:=self.typea -> map toType(); }
mapping LogicalPIM ::Type::toType():CassandraPSM::Type{if(self.typea = "Rid"){type:='Int';}endif; type:=self.typea;}
-- Transforming (1,*) RelationShip using a Monovalued Reference Column
mapping LogicalPIM ::LinkedToTable::toReferenceColumn():CassandraPSM ::ReferenceColumn {if(self.cardinalityoftable =
"*" and self.cardinalityoflinkedtable = "1"){name:=(self.name)+"_Ref"; type:='Int';}endif;}
-- Transforming (1,1) RelationShip using a Monovalued Reference Column
```

(a) script de Cassandra

```
modeltype LogicalPIM uses "http://GenericLogicalModel.com";
modeltype MongoDBPSM uses "http://MongoDBModel.com";
transformation TransformationGenericModelToCassandraModel
(in Source: LogicalPIM, out Target: MongoDBPSM);
main() {Source.rootObjects()[DataBase] -> map toDataBase();}
-- Transforming DataBase to MongoDB DataBase
mapping DataBase::toDataBase():MongoDataBase{name := self.name;collection:=self.table -> map toCollection();}
-- Transforming Table to Collection
mapping LogicalPIM ::Table::toCollection():MongoDBPSM::Collection {name:=self.name;}
atomicfield:=self.attributet -> map toAtomicField();complexfield:=self.islinkedto -> map toComplexField();}
-- Transforming Attribute to Atomic Field
mapping LogicalPIM ::Attribute::toAtomicField():MongoDBPSM::AtomicField {name:=self.name;}
-- Transforming Composition RelationShip using nested data
mapping LogicalPIM ::LinkedToTable::toComplexField():MongoDBPSM::ComplexField {if(self.relationshipType =
"Composition"){name:=self.name;atomicfield:=self.attributeL -> map toField();}endif;}
mapping LogicalPIM ::Attributes::toField():MongoDBPSM::AtomicField{name:=self.name;
```

(b) Script de MongoDB

Figure 5.9 – Script QVT de la transformation GenericModel2PhysicalModel

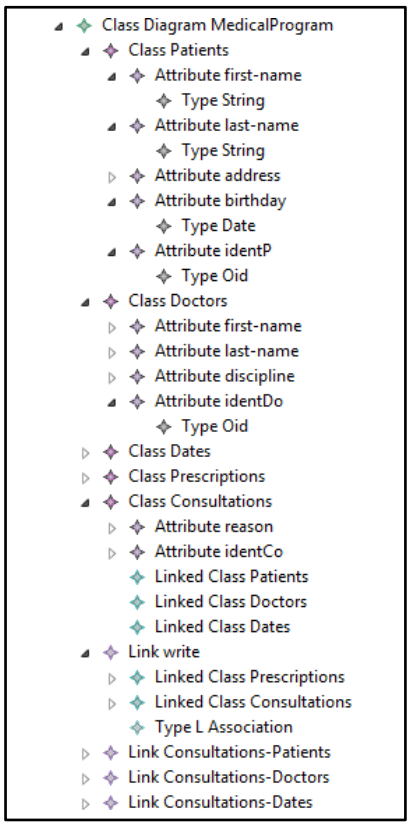


Figure 5.10 – PIM conceptuel

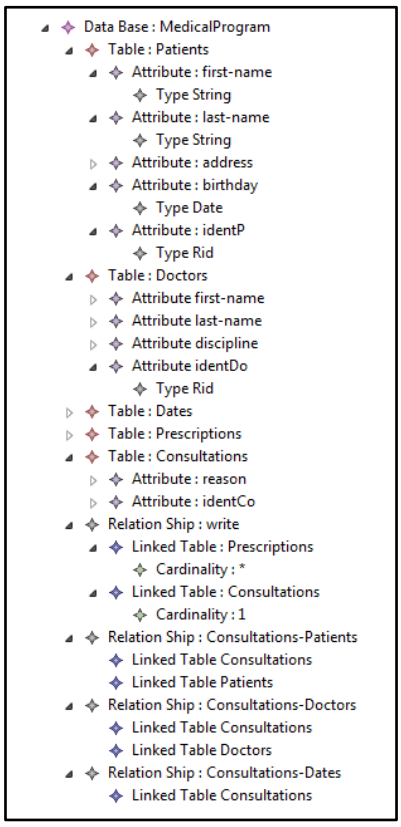
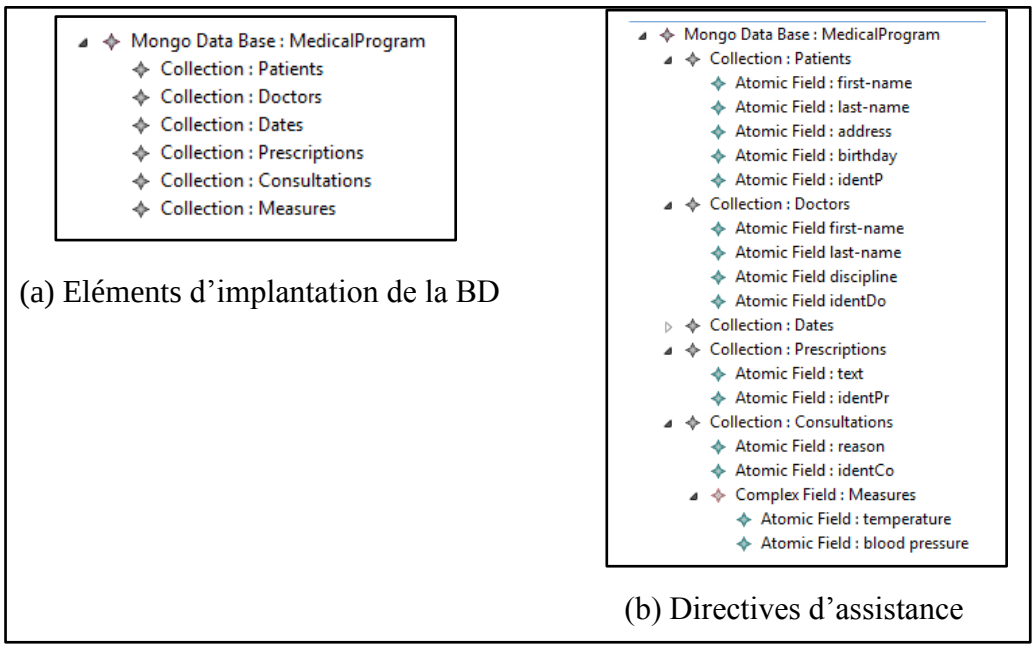


Figure 5.11 – PIM logique



(a) Eléments d'implantation de la BD

(b) Directives d'assistance

Figure 5.12 – PSM MongoDB

5.2.2 Module OCL2Java

Le module Object2NoSQL ne considère que des contraintes simples, telles que les types de données et l'unicité des identifiants, dans le modèle physique NoSQL généré. Pour assurer une meilleure cohérence des données stockées, il est donc nécessaire de disposer d'un outil automatique permettant de prendre en compte d'autres catégories de contraintes plus complexes. Pour ceci, nous proposons le module de transformation des contraintes OCL2Java qui vise à compléter le module de transformation du DCL Object2NoSQL.

OCL2Java se focalise sur la traduction des contraintes d'intégrité associées au DCL transformé par Object2NoSQL décrit dans la section 5.2.1. Pour chaque contrainte, le module OCL2Java s'exécute en deux étapes successives: OCL2JavaModel de type M2M puis JavaModel2JavaCode de type M2T (cf. figure 5.13). Ces étapes produisent respectivement :

- Un modèle pivot de méthode java indépendant de toute plateforme d'implantation NoSQL (colonnes, documents, graphes ou clé-valeur),
- Le code de la méthode à partir de son modèle. Plusieurs codes peuvent être produits à partir du même modèle ; chacun d'eux est spécifique à un SGBD parce qu'il contient des requêtes d'accès aux données.

L'intérêt d'introduire un niveau intermédiaire situé entre les contraintes OCL (niveau conceptuel) et le code (niveau physique) est expliqué dans la section 4.1 du chapitre 4.

Conformément aux principes de MDA, OCL2Java s'appuie sur l'utilisation des métamodèles pour effectuer ses transformations. Il s'agit des métamodèles OCL et Java que nous avons proposés et présentés dans le chapitre 4.

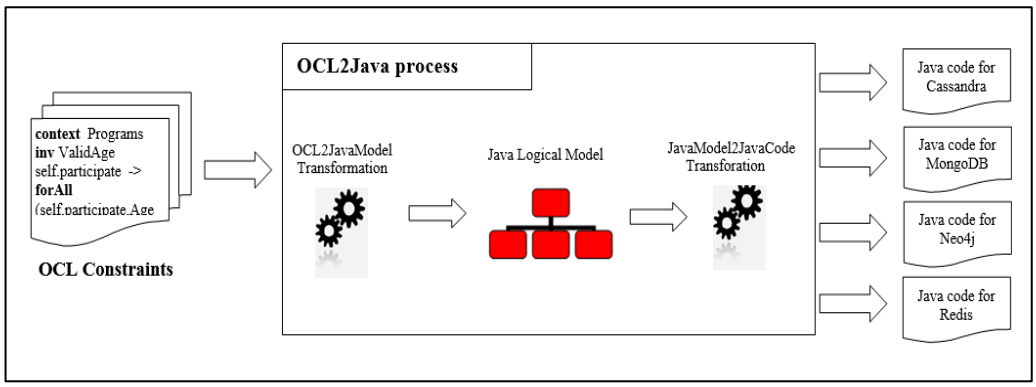


Figure 5.13 – Module OCL2Java

Pour bien situer le module des contraintes OCL2Java, nous détaillons ci-dessous ses différentes composantes (entrée, sortie, transformations), ainsi que les étapes de son implantation.

Entrée

L'entrée du module OCL2Java correspond à des contraintes d'intégrité. Nous avons adopté le langage OCL (Object Constraint Language) normalisé par l'OMG pour exprimer ces contraintes. Une description de ce langage est fournie dans la section 4.2.1 du chapitre 4. Nous rappelons que OCL permet d'exprimer deux types de contraintes : les invariants et les pré/post-conditions. Dans le cadre de nos travaux, nous considérons uniquement les invariants de classes.

La traduction automatique d'un invariant OCL vers une méthode java nécessite de formaliser préalablement les expressions OCL et de proposer un métamodèle décrivant les concepts présents dans chacune. Dans la documentation de l'OMG[27], les expressions OCL sont définies informellement dans des contextes différents.

Compte tenu de la variété des cas de contraintes figurant dans cette documentation, il était difficile de proposer une formalisation et un métamodèle exhaustifs pour définir précisément tous les types possibles d'expressions OCL. Nous nous sommes donc limités aux expressions qui sont fréquemment utilisées. Il s'agit d'expressions itératives, d'expressions opération, d'expressions conditionnelles et d'expressions de définition. Le chapitre 4 propose une définition formelle pour chacune de ces expressions (cf. section 4.2.1) et un métamodèle OCL (cf. figure 4.3) que nous avons défini en s'appuyant sur les spécifications de l'OMG.

L'utilisateur peut ainsi fournir l'entrée du module OCL2Java (i.e. l'ensemble des contraintes OCL à traduire) en instanciant le métamodèle proposé.

Sortie

Pour chaque contrainte OCL d'entrée, la sortie est un code java spécifique au SGBD NoSQL choisi par l'utilisateur. Ce code permet de vérifier la contrainte correspondante suite à une mise à jour de la BD.

Au niveau de la sortie, nous nous sommes limités aux instructions Java nécessaires à la transformation des expressions OCL que nous avons considérées comme entrée du module OCL2JavaModel (cf. section 4.2.2 du chapitre 4).

Transformations

Le module Object2NoSQL fait appel à deux transformations qui s'exécutent successivement pour chaque contrainte d'entrée. Il s'agit des transformations : OCL2JavaModel et JavaModel2JavaCode. La première est de type M2M et nous l'avons formalisée avec QVT. Elle produit un modèle de méthode java indépendant mais compatible avec les quatre types de plateformes NoSQL. La seconde est de type M2T et nous l'avons formalisée avec MOFM2T. Elle génère

[27] http://www.omg.org/spec/OCL/

un code java à partir du modèle retourné par OCL2JavaModel. Ce code contient des requêtes d'accès aux données qui sont exprimées avec le langage spécifique au SGBD choisi par l'utilisateur.

Dans ce module, le niveau logique fait apparaitre principalement des instructions de déclaration et d'initialisation, des instructions de test, des instructions logiques, la boucle foreach et les méthodes m^{att}, m^{obj} et m^{s} (cf. section 4.2.3 du chapitre 4). Nous rappelons que ces méthodes permettent de retourner respectivement : un attribut d'une classe, tous les objets d'une classe et uniquement les objets vérifiant une condition donnée.

Après le passage conceptuel > logique (Etape 2), seule la signature de ces méthodes est générée (i.e. le nom et la liste des paramètres). A partir de chaque signature, la transformation M2T génère une requête de sélection écrite avec le langage d'interrogation propre à un SGBD NoSQL.

Exemple :

L'invariant suivant spécifie que tous les patients impliqués dans le programme médical ont plus de 16 ans.

context Programme
inv ContrainteAge :
self.patient -> forAll(p : Patients | p.age >= 16)

Après le passage conceptuel vers logique, cette contrainte est transformée en une boucle b^{fe} = (is^{di}, $entête^{b^{fe}}$, $corps^{b^{fe}}$) (cf. Définition 32), où :

- $b^{fe}.is^{di}$ est définie ainsi : $is^{di}.T^{v}$ = "Collection", $is^{di}.N^{v}$ = "resultat", $is^{di}.N^{op}$ = "=", $is^{di}.Vl^{v}$ = "m^{obj}" . la signature de la méthode m^{obj} est définie par un nom N et un seul paramètre {pr_1}, où : m^{obj}.N = « getObj » et m^{obj}.PR = {pr_1}, avec pr_1 = "$Patients$".
- $b^{fe}.entête^{b^{fe}}$ est définie ainsi : $entête^{b^{fe}}.N^{cll}$ = "resultat", $entête^{b^{fe}}.T^{o}$ = "Object" et $entête^{b^{fe}}.N^{o}$ = "object".
- $b^{fe}.corps^{b^{fe}}$ = {is^{d}, is^{if}}, où :
  - $corps^{b^{fe}}.is^{d}$ est définie ainsi : $is^{d}.T^{v}$ = "Boolean", $is^{d}.N^{v}$ = "out".
  - $corps^{b^{fe}}$. is^{if} est définie ainsi : is^{if} . $condition$ = « age >= 16 », is^{if} . IS^{v} = { is^{l} }, avec is^{l} une instruction logique définie par (v_1, $operateur^{c}$, v_2), où : is^{l} . v_1 = "out" is^{l} . $operateur^{c}$ = "=" et is^{l} . v_2 = "True", is^{if} . IS^{f} = {is^{l}, is^{br}, is^{a}}, avec is^{l} une instruction logique définie par (v_1, $operateur^{c}$, v_2), où : is^{l} . v_1 = " out " is^{l} . $operateur^{c}$ = " = " et is^{l} . v_2 = " False " , is^{a} une instruction d'affichage, où is^{a}.N = « Afficher » et is^{a}.contenu = « ERREUR », is^{br} est une instruction Break.

La boucle suivante présente une vue synthétique du résultat obtenu :

```
Collection resultat = getObj (Patients);
        for (Object resultat : object)
{
            Boolean b;
            if (age >=16) {b= true;}
            else{
                b= false;
                Afficher("Erreur");
                break;
                }
}
```

Notons que le résultat retourné après le passage conceptuel vers logique est un modèle et pas du code. La boucle ci-dessus a pour but de simplifier la compréhension du résultat.

L'étape suivante consiste à générer le code java permettant de vérifier la contrainte « ContrainteAge » sur le SGBD NoSQL choisi par l'utilisateur.

S'il agit du SGBD Cassandra par exemple, nous obtenons le code suivant :

```
ResultSet result =session.execute(Select * FROM Patients);
        for (Row row:result)
{
            Boolean out;
            if(row.getInt("age") >= 16) {b= true;}
            else{b= false;System.out.println("Erreur");break;}
}
```

Et pour le SGBD MongoDB, nous obtenons ce résultat :

```
FindIterable result = database.getCollection("Patients").find();
        for (Document document : result)
{
            Boolean out;
            if(document.get("age") >= 16) { b= true;}
            else{b= false; System.out.println("Erreur");
                break;}
}
```

Implantation

Cette section présente la mise en œuvre du module OCL2Java. Ce dernier est réalisé en suivant les étapes illustrées dans la figure 5.14 :

Etape 1 : Nous avons créé en Ecore les métamodèles proposés dans le chapitre 4 pour OCL et Java (cf. figures 4.3 et 4.4). La figure 5.15 montre un extrait des fichiers Ecore correspondant à ces métamodèles : OCL (a) et Java (b).

Etape 2 : Nous avons utilisé le langage QVT décrit dans la section 5.1 pour implanter la première transformation M2M assurant le passage conceptuel vers logique. La figure 5.16 montre un extrait du script QVT correspondant à ce passage.

Etape 3 : Nous avons implanté le passage logique vers physique (Modèle vers Texte) avec le langage MOFM2T décrit dans la section 5.1. La figure 17 montre un extrait des scripts MOFM2T : pour Cassandra (a) et pour MongoDB (b). Chacun de ces scripts prend en entrée un modèle de méthode java généré à l'étape 2 et restitue le code java permettant de vérifier une contrainte OCL sur le SGBD NoSQL correspondant.

Pour tester le module de transformation des contraintes OCL2Java, nous avons instancié le métamodèle OCL pour créer des contraintes OCL (cf. figure 5.18). Ces contraintes représentent l'entrée du module.

L'exécution du script OCL2JavaModel sur les contraintes d'entrée renvoie des modèles logiques de méthodes java ; chacun de ces modèles correspond à une contrainte d'entrée (cf. figure 5.19). A partir de chaque modèle logique, nous pouvons générer plusieurs codes ; chacun d'eux est spécifique à un SGBD NoSQL. Ceci est assuré par le script JavaModel2JavaCode. La figure 5.20 présente un extrait du code de vérification des contraintes sur le SGBD Cassandra ; les commentaires figurant dans le code indiquent comment le script sera utilisé pour vérifier une contrainte sur la BD.

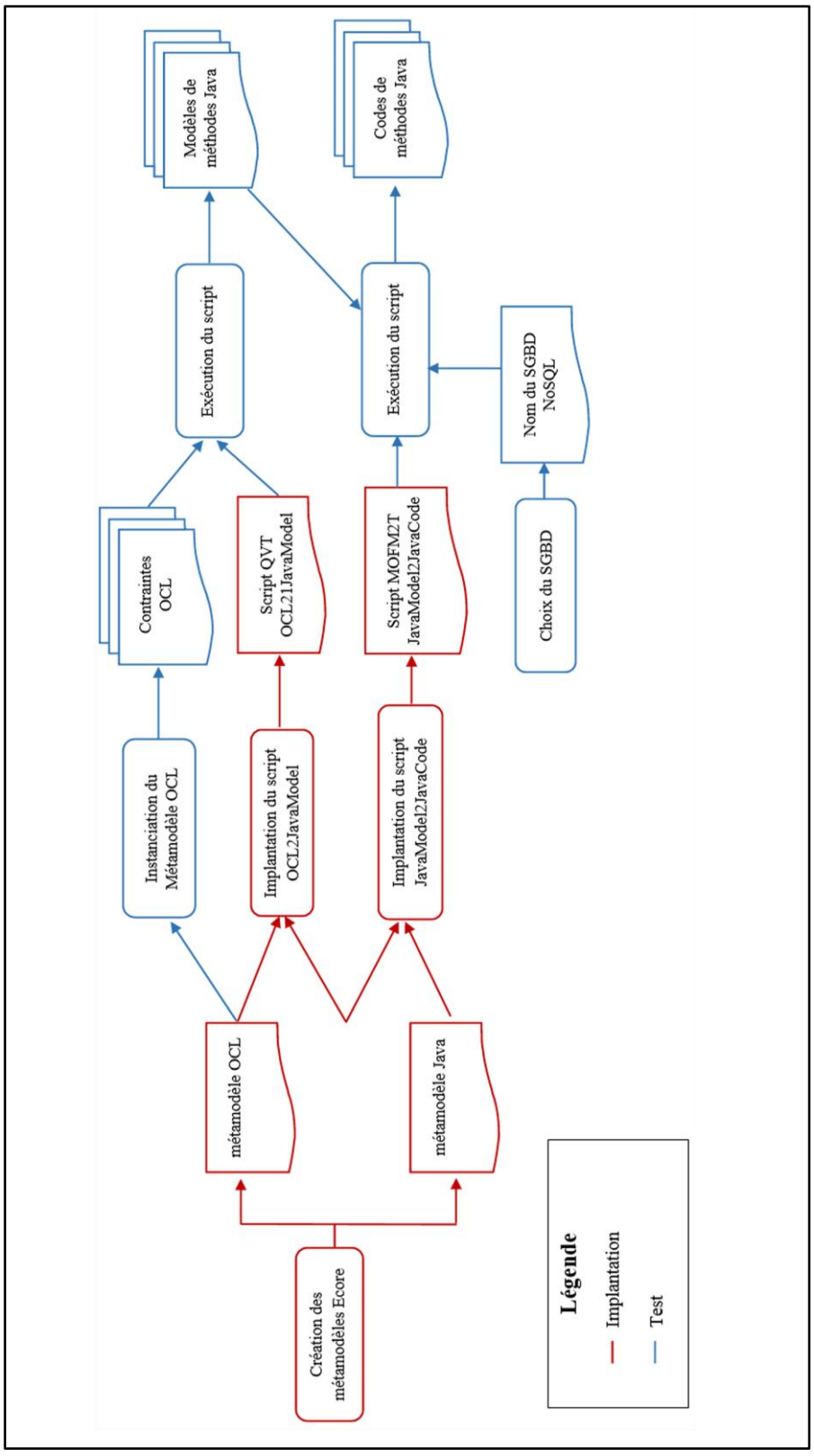


Figure 5.14 – Etapes d'implantation du module OCL2Java

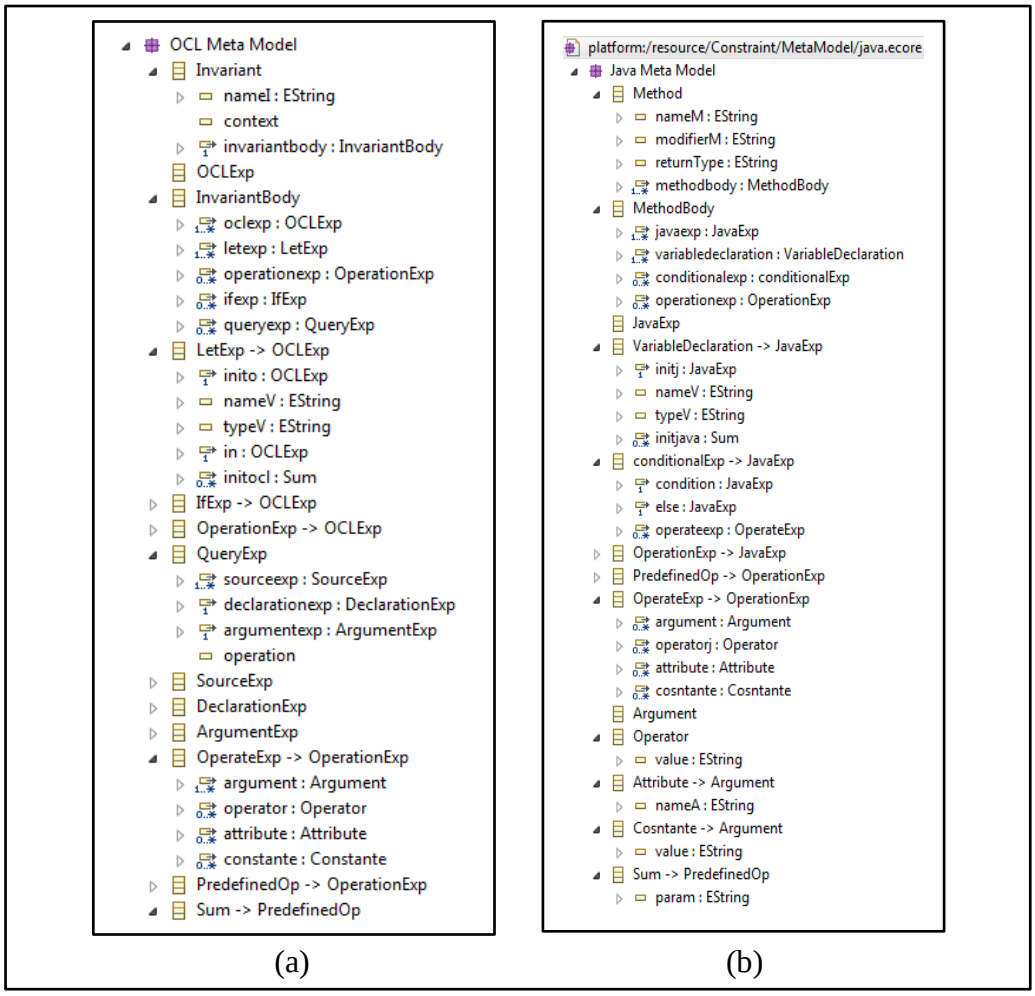


Figure 5.15 – Métamodèles Ecore du module OCL2Java

```
modeltype COM uses "http://ConstraintModel.com";
modeltype JVM uses "http://JavaModel.com";
transformation ConstraintTransformation(in Source: COM, out Target: JVM);
main() {Source.rootObjects()[Invariant] -> map toMethod();}
mapping Invariant::toMethod():Method{
nameM := self.nameI+"()";modifierM:="public";returnType:="boolean";methodbody:=self.invariantbody -> map
toMethodBody();}
mapping COM ::InvariantBody::toMethodBody():JVM::MethodBody{
javaexp:=self.oclexp -> map toJavaExpression(); variabledeclarationcmd:=self.letexp -> map
toVariableDeclarationCmd();conditionalexp:=self.ifexp -> map toConditionalExp();  }
//Transformation de l'expression Let en une instruction de déclaration et d'initialisation
mapping LetM ::LetExp::toVariableDeclarationCmd():JVM::VariableDeclarationCmd{
declarationcmd:=self.declarationexp -> map toDeclarationCmd();initializationcmd:=self.initializationexp -> map
toInitializationCmd();}
mapping LetM ::DeclarationExp::toDeclarationCmd():JVM::DeclarationCmd{nameV:=self.nameV;
typeV:=self.typeV;}
mapping LetM ::InitializationExp::toInitializationCmd():JVM::InitializationCmd{param:=self.appliedop +" of "+"
' "+ "select " + self.attribute + " from "+ self.linkedclass +" ";}
//Transformation de l'expression OCL If Then Else en une instruction de test
mapping COM ::IfExp::toConditionalExp():JVM::conditionalExp{operateexp:=self.operateexp -> map
toOperateExp();elseexp:=self.else -> map toElseExp();}
mapping COM ::OperateExp::toOperateExp():JVM::OperateExp{operatorj:=self.operatoro -> map toOperator();
attribute:=self.attribute -> map toAttribute();cosntante:=self.constante -> map toConstante();}
mapping COM ::Operator::toOperator():JVM::Operator{value:=self.operator;}
mapping COM ::Attribute::toAttribute():JVM::Attribute{nameA:=self.nameA;}
mapping COM ::Constante::toConstante():JVM::Cosntante{
```

Figure 5.16 – Script QVT de la transformation OCL2JavaModel

```
[template public generate (v : oclj)]
[comment @main /]
    [file ('script java', false, 'UTF-8')]
OMBLIST PROJECTS
OMBCC 'My_Project'
OMBCC 'Source'
Texttransformation JavaModel2JavaCode (in cs: "http://www.eclipse.org/emf"){
 cd.Datastax :: main(){
    file (self.Name.toLower())
  var l : List = self.invariant  ->  select (c)
stout.println(l.size() + "  " + l.isEmpty())
l ->forEach (cc : cs.EObject | cc.oclGetType = "Invariant"){
cc.Inv()
cs.Method :: MyMethod(){
CREATE Method 'self.getSignature()'
Var xt : l = self.Type  -> select(T)
xt  -> forEach(t:cs){
t.Type()
}
Var xn : l = self.Name  -> select(N)
xt  -> forEach(n:cs){
n.Name()
}
Var xs : l = self.Signature  -> select(S)
xs  -> forEach(s:cs){
s.Type()
}
Var xn : l = self.Name  -> select(N)
xt  -> forEach(n:cs){
n.Name()
}
Var xs : l = self.Signature  -> select(S)
xs  -> forEach(s:cs){
s.Type()
}
```

Figure 5.17 – Extrait du Script MOFM2T de la transformation JavaModel2JavaCode

```
public class CassConnector {
// SE CONNECTER AU KEYSPACE DE L'APPLICATION MEDICALE
Cluster cluster;
Session session;
cluster = Cluster.builder().addContactPoint("127.0.0.1").build();

 // Ouvrir la connexion
session = cluster.connect("MedicalProgram");

public static boolean constraint1 () {
String query = "SELECT * FROM InvolvedHospital";
ResultSet result =session.execute(query);
int i = result.all().size();

 if(i==0)
 return false;
else
return true;}
public static boolean constraint2 () {
if
(session.execute("SELECT * FROM Doctor").isExhausted()==false){
 return true;
            }
else{
 return false;}}
public static boolean constraint3 () {
String query = "SELECT * FROM Patient";
            ResultSet result =session.execute(query);
            for (Row row:result){
                if(row.getInt("age")>16)
                return true;
                else{
                break;
                return false;}}
                }
public static boolean constraint4 (){
ResultSet result = session.execute("SELECT * FROM
Patient");
for (Row row:result){
if (row.getInt("s") == "male"
&& row.getInt("nss").charAt(0) == 1)
return true
else
return false }
            // Fermer la connexion
             session.close();
              cluster.close();}
```

Figure 5.20 – Codes de vérification des contraintes sur le SGBD Cassandra

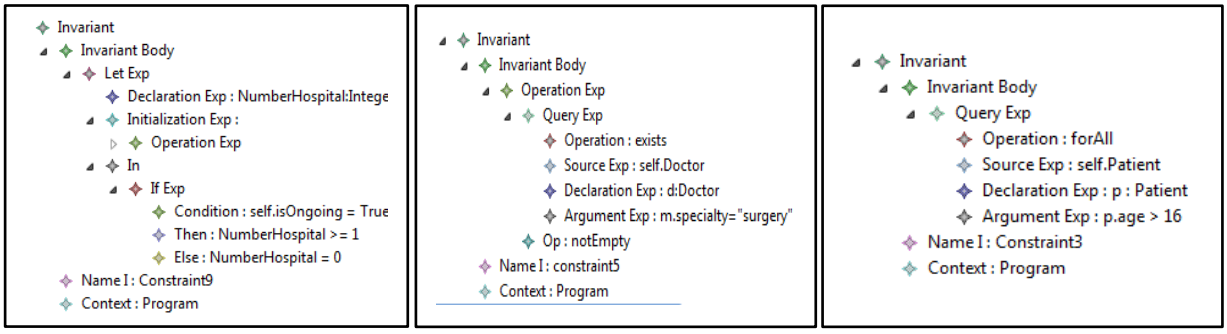


Figure 5.18 – Contraintes OCL

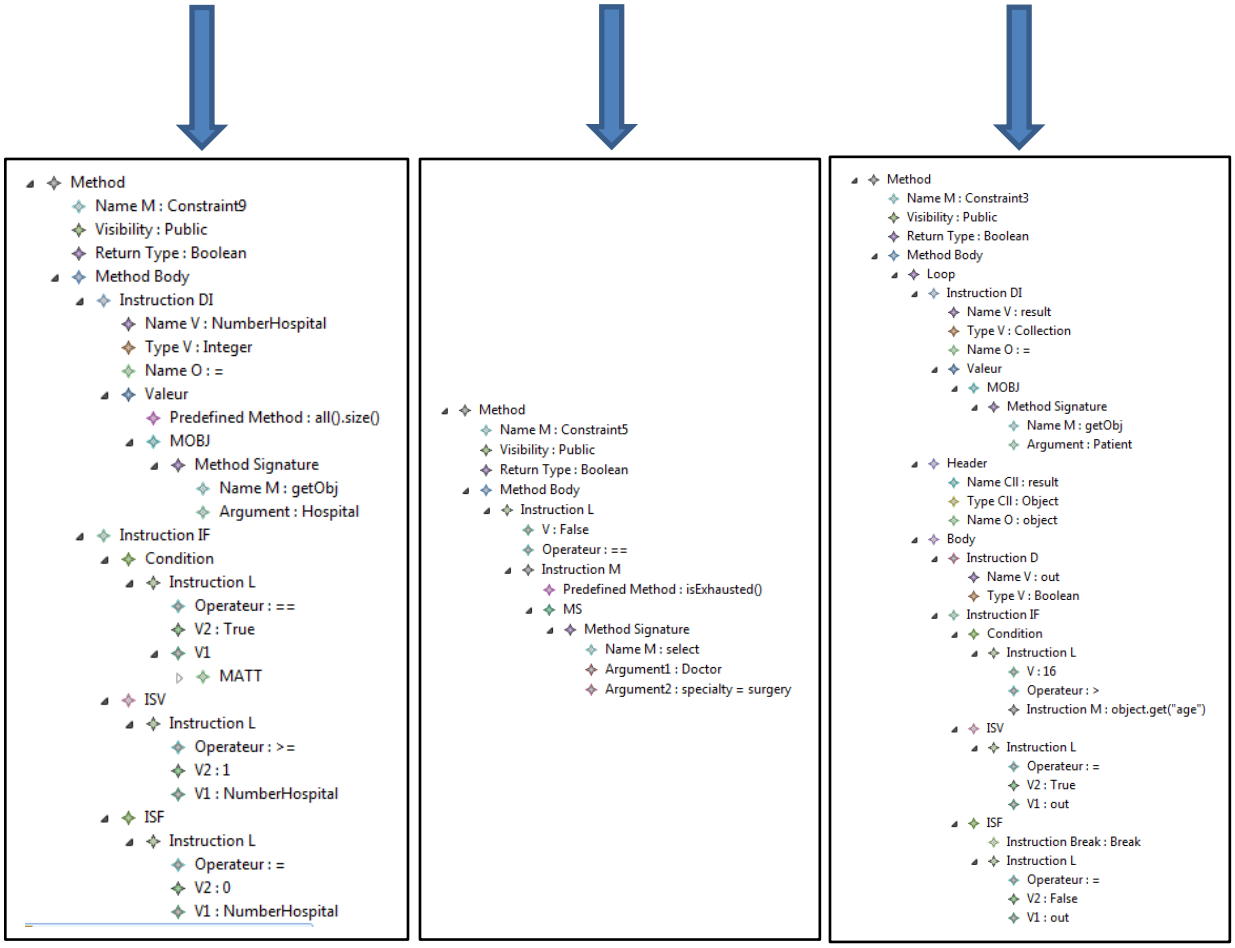


Figure 5.19 – Modèles de méthodes java

5.3 Validation

L'expérimentation présentée dans la section précédente a porté sur notre application médicale. Elle a permis de montrer la faisabilité de nos propositions. Ainsi grâce au développement logiciel de notre processus de transformations, un modèle conceptuel complet (structure + contraintes) peut être automatiquement traduit en un modèle physique NoSQL. A présent, il convient d'évaluer l'apport de ce processus dans une démarche professionnelle. L'objectif de nos travaux est bien de simplifier certaines tâches dans le développement d'applications Big Data ; plus précisément, il s'agit de réduire les ressources nécessaires à l'activité de transformation de modèles.

Travaillant en liaison avec une entreprise de développement d'applications décisionnelles située en région parisienne, la société TRIMANE[28], nous avons pu obtenir la mise en œuvre de notre logiciel sur deux applications réelles (pour des raisons de confidentialité, les noms des fichiers et des objets manipulés ont été changés). Ceci nous a permis de réaliser la validation de nos propositions. La première application traite de la mise à disposition des décisions des tribunaux, la seconde concerne l'étude de la disponibilité d'agents d'intervention. Il s'agit de deux applications décisionnelles développées par TRIMANE ; elles manipulent du texte et des photos et présentent des volumes de données importants (plusieurs téraoctets). Ces caractéristiques ont justifié le choix d'une implantation des données sur des systèmes NoSQL. Nous avons demandé une participation à deux employés de l'entreprise impliqués dans le développement de chacune des applications. Ces deux utilisateurs de la base de données ont des formations et des activités différentes :

- Un décideur (analyste de données) pour la première application dont le modèle de données présente une complexité modérée. Bien qu'il n'ait pas élaboré seul le modèle, ce non-informaticien comprend la sémantique des objets représentés. Sa formation lui permet également de définir le modèle d'implantation (simple) puis de rédiger des requêtes de type SQL pour élaborer des cubes de données.
- Un informaticien (ingénieur technique) qui a conçu en équipe le modèle de données de la seconde application. Ce spécialiste maîtrise évidemment les techniques d'implantation des bases de données ainsi que leur manipulation.

Ces deux acteurs ont, comme ils le font habituellement, transformé manuellement le modèle conceptuel des données en un modèle d'implantation. Le décideur a été convié à ne pas faire appel à un informaticien pour l'assister face à des difficultés d'implantation.

[28] http://www.trimane.fr/

5.3.1 Applications

L'application « Juris-Dépôt »

Cette application consiste à mettre la jurisprudence, c'est-à-dire l'ensemble des décisions des tribunaux français et européens, à la disposition de juristes tels que les avocats et les notaires. La jurisprudence est constituée de gros volumes de données textuelles contenant des arrêts du conseil d'état, des décisions de tribunaux, des avis contentieux, des ouvrages spécialisés, des revues d'analyse, etc. Ces documents sont généralement publiés dans des fonds documentaires numériques qui sont multiples, hétérogènes et dont les données sont difficiles à croiser.

Il s'agit de centraliser ces connaissances au fur et à mesure de leur publication en alimentant une base de données. Des analystes chargés de la saisie des données, établissent les liens internes explicites entre les documents stockés ; ils enregistrent également des liens externes vers des débats parlementaires ou bien vers des textes de loi (numérisés par ailleurs). Le volume de données traitées atteint plusieurs téraoctets. L'objectif de l'application est de permettre aux utilisateurs d'interroger et d'analyser les connaissances juridiques stockées et ceci avec un maximum d'efficacité.

Le modèle graphique suivant décrit la structure des données. Compte tenu de la simplicité de cette structure, l'implantation de la base de données a été confiée à un non informaticien qui fait partie de l'équipe d'analystes chargés de l'enregistrement des connaissances.

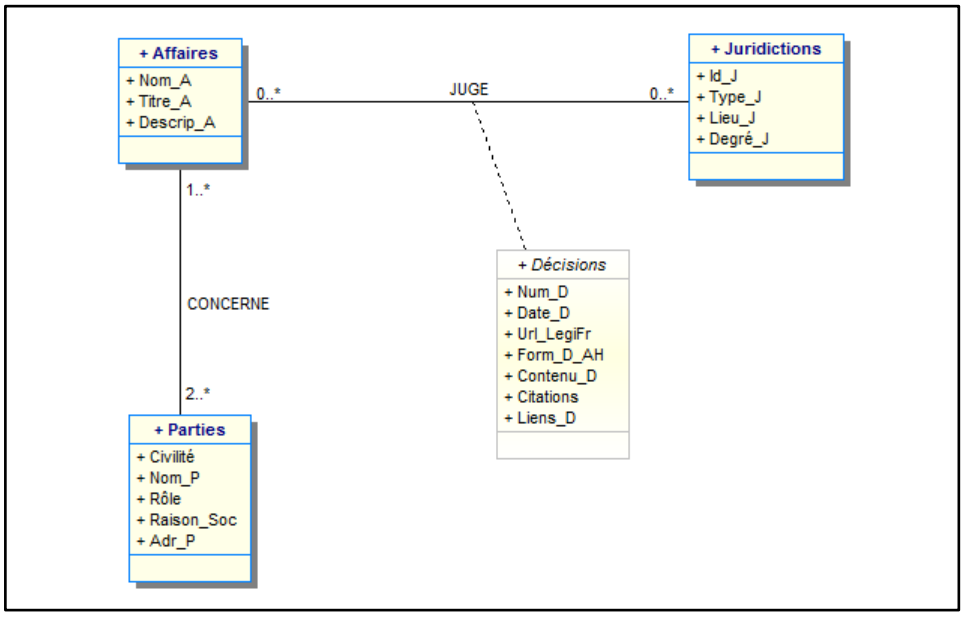


Figure 5.21 – DCL de l'application « Juris-Dépôt »

Ce modèle est complété par un dictionnaire des données qui donne, pour chaque attribut, un libellé explicatif, le type ainsi que les contraintes d'intégrité élémentaires et inter-attributs (non fourni ici).

Aucune contrainte d'intégrité additionnelle ne complète ce modèle dans la mesure où les données sont extraites de documents consolidés et publiés (site Légifrance notamment) dont la cohérence a déjà été contrôlée.

L'application « Disponibilités Des Agents»

Cette application est développée pour le compte d'un regroupement de communes situées à proximité d'une métropole régionale. Elle centralise les prévisions de disponibilité des agents de sécurité municipaux chargés du maintien de l'ordre public et répertorie les interventions pour chaque centre de sécurité communal ; ces données sont fournies quotidiennement par les responsables de centres et sont conservées une année en ligne. Les sorties sont répertoriées dans l'application chaque fois qu'une mission a fait l'objet d'une intervention : rixe, accident de circulation, mise en sécurité d'une zone, appel de la police judiciaire, etc. Chaque intervention fait l'objet d'un rapport, éventuellement de photos et de séquences vidéo, susceptibles d'être utilisés dans une procédure judiciaire.

Cette application a pour intérêt d'organiser la capacité de réaction de chaque centre et de permettre, le cas échéant, de faire appel à un centre voisin. Elle peut aussi faciliter la mobilisation de plusieurs centres pour faire face à une catastrophe, par essence imprévisible. En matière décisionnelle, l'ensemble des données fait l'objet d'analyses en différé pour mieux anticiper les besoins et évaluer la réactivité des centres.

La structure des données est représentée par le modèle graphique suivant. Dans un souci de simplification, la totalité des classes et des attributs n'est pas mentionnée. On constate cependant une certaine complexité du modèle justifiant que l'implantation de cette base de données soit confiée à un informaticien.

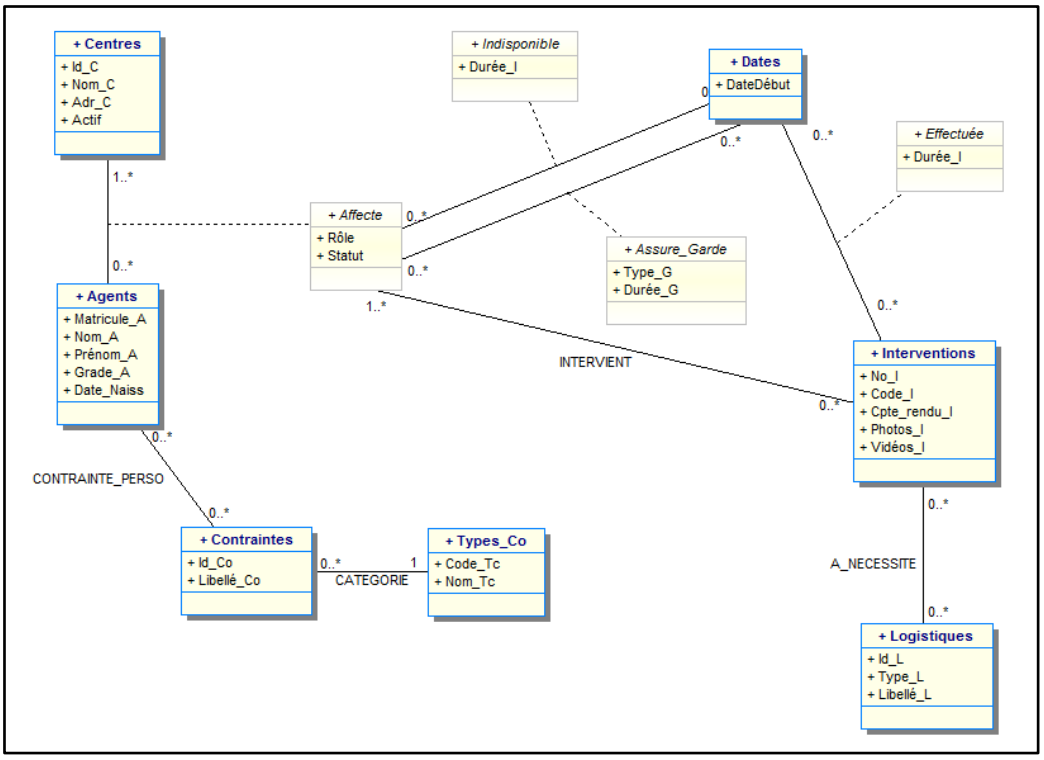


Figure 5.22 – Extrait du DCL de l'application « Disponibilités Des Agents »

Ce modèle est complété :

- par un dictionnaire des données (non fourni ici) ;
- par les contraintes d'intégrité additionnelles suivantes :

CI-1 : Un agent affecté à un centre doit être disponible pour être de garde sur une période donnée.

CI-2 : Un agent affecté à un centre peut assurer une intervention s'il est disponible sur la période de celle-ci.

Ces deux contraintes correspondent respectivement aux invariants OCL suivants :

context CENTRES
inv CI-1 :
Let DateFinI : Date = **self**.agents.affecté.indisponible.dates.DateDébut + **self**.agents.affecté.indisponible.Durée_I **In**
self.agents -> **forAll** (a : AGENTS | a.affecté.Rôle = 'Garde' **implies** a.affecté.assure_garde.dates.DateDébut > FinI)

context CENTRES
inv CI-2 :
Let AssureIntervention : Boolean = True **In**
if (self.affecté.interventions.dates.DateDébut >
(**self**.agents.affecté.indisponible.dates.DateDébut +
self.agents.affecté.indisponible.Durée_I))
then AssureSortie = True
else AssureSortie = False
endif

5.3.2 Mise en œuvre du processus de transformation

Le modèle conceptuel des données, constitué de la structure des données, du dictionnaire et des contraintes d'intégrité, représente le point de départ des processus d'implantation manuel et automatisé. Dans le processus automatisé, la structure et les contraintes sont saisies dans le format XMI (voir Section 5.1.2).

Modèle physique de l'application « Juris-Dépôt »

Le décideur impliqué dans le développement de cette application a utilisé le SGBD Neo4j pour implanter la base de données. Une description détaillée de ce système est fournie dans le chapitre 3 (cf. Section 3.3.2). Dans un premier temps, le décideur a appliqué son processus d'implantation habituel qui porte sur la transformation manuelle du modèle conceptuel des données en un modèle d'implantation. Dans un second temps, nous avons utilisé notre processus automatisé des transformations de modèles sur le même modèle conceptuel ; ce dernier a été saisi sous le format XMI. Nous présentons ci-dessous le résultat retourné par chaque processus (manuel et automatisé).

a. Résultat du processus manuel

Nœuds :

Affaires :
Propriétés : {Nom_A, Titre_A, Descrip_A}

Juridictions :
Propriétés : {Id_I, Type_J, Lieu_J, Degré_J}

Parties :
Propriétés : {Civilité, Nom_P, Rôle, Raison_Soc, Adr_P}

Décisions :
Propriétés : {Num_D, Date_D, Url_LegiFr, Form_D_AH, Contenu_D, Citations_D, Liens_D}

Arcs :

Concerne :
Extrémités : (Affaires, Parties)

Juge :
Extrémités : (Affaires, Juridictions)

Décisions_Affaires :
Extrémités : (Décisions, Affaires)

Décisions_Juridictions :
Extrémités : (Décisions, Juridictions)

Figure 5.23 – Modèle physique Neo4j établi par le décideur

b. Résultat du processus automatisé :

Comme nous l'avons vu dans la section 3.3.3.3, le modèle des données Neo4j est spécifié au fur et à mesure de la saisie des données ; aucun élément structurel ne doit être fixé avant de commencer l'alimentation de la base de données. Ainsi pour Neo4j, seules les directives d'assistance sont générées.

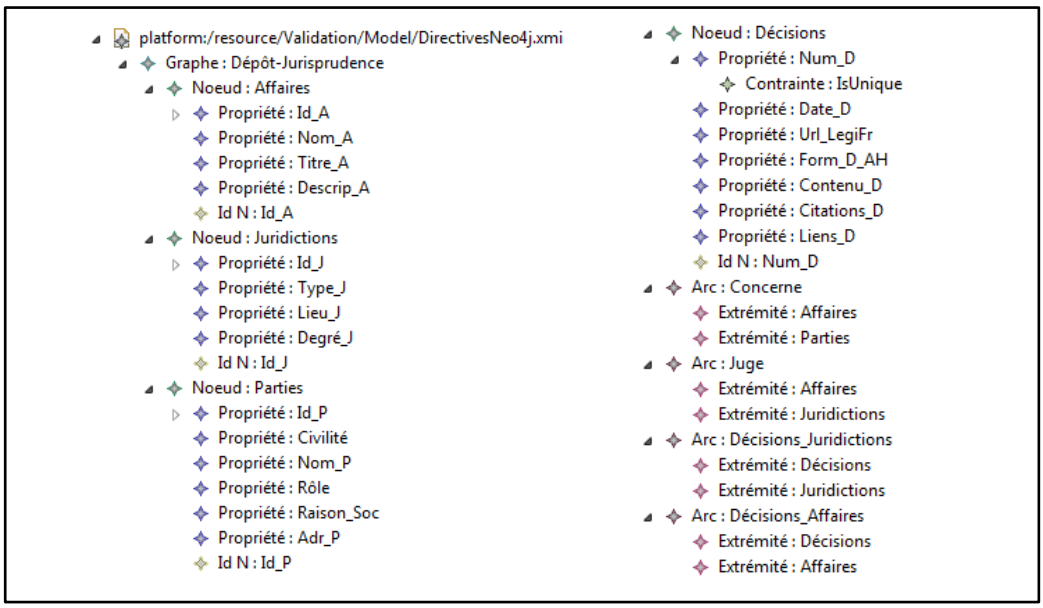


Figure 5.24 – Directives d'assistance produites par notre système

Pour l'application « Juris-Dépôt », le choix d'implantation des liens ne s'est pas posé dans la mesure où le système Neo4j dispose d'une seule solution pour implanter les liens.

Modèle physique de l'application « Disponibilités Des Agents »

Pour implanter la BD de cette application, l'informaticien chargé de cette tâche a choisi le SGBD Cassandra dont les caractéristiques sont présentées dans section 3.3.2. Pour générer le modèle d'implantation Cassandra, l'informaticien a choisi d'implanter les liens figurant dans le modèle conceptuel comme suit :

Lien Conceptuel	Type	Solution d'implantation
Contrainte_Perso	Binaire	Référence multivaluée côté Agents
Catégorie	Binaire	Référence monovaluée côté Contraintes
Intervient	Binaire	Référence multivaluée côté Interventions
A_Nécessité	Binaire	Référence multivaluée côté Interventions
Affecte, Indisponible, Assure_Garde et Effectuée	Classes d'Associations	Familles de colonnes contenant des références

Tableau 5.1 – Choix d'implantation des liens pour l'application Disponibilités Des Agents

Nous présentons ci-dessous le résultat de la transformation manuelle faite par l'informaticien et le résultat obtenu en utilisant notre processus automatisé.

a. Résultat du processus manuel :

Keyspace [DisponibilitéAgents] {

FC [Centres] :

Colonnes : (Id_C int, Nom_C text, Adr_C text, Actif boolean, PrimaryKey (Id_C))

FC [Agents] :

Colonnes : (Matricule_A int, Nom_A text, Prénom_A text, Grade_A text, Date_Naiss date, ContraintesRef set <int>, PrimaryKey (Matricule_A))

FC [Dates] :

Colonnes : (Id_D int, DateDébut date, PrimaryKey (Id_D))

FC [Interventions] :

Colonnes : (No_I int, Code_I int, Cpte_rendu_I text, Photos_I text, Vidéos_I text, AffecteRef set <int>, LogistiquesRef set <int>, PrimaryKey (No_I))

FC [Logistqiues] :

Colonnes : (Id_L int, Type_L text, Intitulé_L text, PrimaryKey (Id_L))

FC [Contraintes] :

Colonnes : (Id_Co int, Libellé_Co text, TypeRef int, PrimaryKey (Id_Co))

FC [Types_Co] :

Colonnes : (Code_Tc int, Nom_Tc text, PrimaryKey (Code_Tc))

FC [Affecte] :

Colonnes : (Id_Af int, Rôle text, Statut text, CentreRef int, AgentRef int, PrimaryKey (Id_Af))

FC [Indisponible] :

Colonnes : (Id_I int, Durée_I int, AffecteRef int, DateRef int, PrimaryKey (Id_I))

FC [Assure_Garde] :

Colonnes : (Id_G int, Type_G text, Durée_G int, AffecteRef int, DateRef int, PrimaryKey (Id_G))

FC [Effectuée] :

Colonnes : (Id_E int, Durée_I int, DateRef int, InterventionRef int, PrimaryKey (Id_E))

Figure 5.25 – Modèle physique Cassandra élaboré par l'informaticien

```
public class CassandraContr {
Cluster cluster;
Session session;
cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
session = cluster.connect("DiponibilitéAgents");public static boolean CI1 () {
 ResultSet result1 =session.execute("SELECT * FROM Agents");
 for (Row row:result1){
 ResultSet result2 =session.execute("SELECT * FROM Affecte where [Agents_Ref = row.getInt("Matricule_A")]");
for (Row row:result2){
ResultSet result3 =session.execute("SELECT * FROM Assure_Garde  where [Affecte_Ref = row.getInt("Id_Af")]");
 for (Row row:result3){
d1 =session.execute("SELECT DateDébut FROM Dates where [Id_D = row.getInt("Dates_Ref")]");}}
 }
 ResultSet result5 =session.execute("SELECT * FROM Agents");
  for (Row row:result5){
  ResultSet result6 =session.execute("SELECT * FROM Affecte where [Agents_Ref = row.getInt("Matricule_A")]");
  for (Row row:result6){
 d2 =session.execute("SELECT Durée_I FROM Indisponible where [Affecte_Ref = row.getInt("Id_Af")]");}}
 ResultSet result8 =session.execute("SELECT * FROM Agents");
   for (Row row:result8){
   ResultSet result9 =session.execute("SELECT * FROM Affecte where [Agents_Ref = row.getInt("Matricule_A")]");
  for (Row row:result9){
ResultSet result10 =session.execute("SELECT * FROM Indisponible where [Affecte_Ref = row.getInt("Id_Af")]");
  for (Row row:result10){
  d3 = session.execute("SELECT DateDébut FROM Dates where [Id_D = row.getInt("Dates_Ref")]");}
}}
if (d1>d3+d2) return True; else return false;}
session.close();
cluster.close();}
```

Figure 5.26 – Extrait du code de vérification des contraintes élaboré par l'informaticien

b. Résultat du processus automatisé :

Nous rappelons que le modèle de données sous Cassandra doit être fixé préalablement. Il en est ainsi du nom de la BD, des noms de familles de colonnes et des noms de colonnes ainsi que leur type ; ces quatre composantes représentent les éléments nécessaires à l'implantation de la BD. L'implantation des liens est précisée dans les directives d'assistance (voir Section 3.3.3.1).

En ce qui concerne les liens, les choix figurant dans le tableau 5.1 ont été repris et introduits dans notre système.

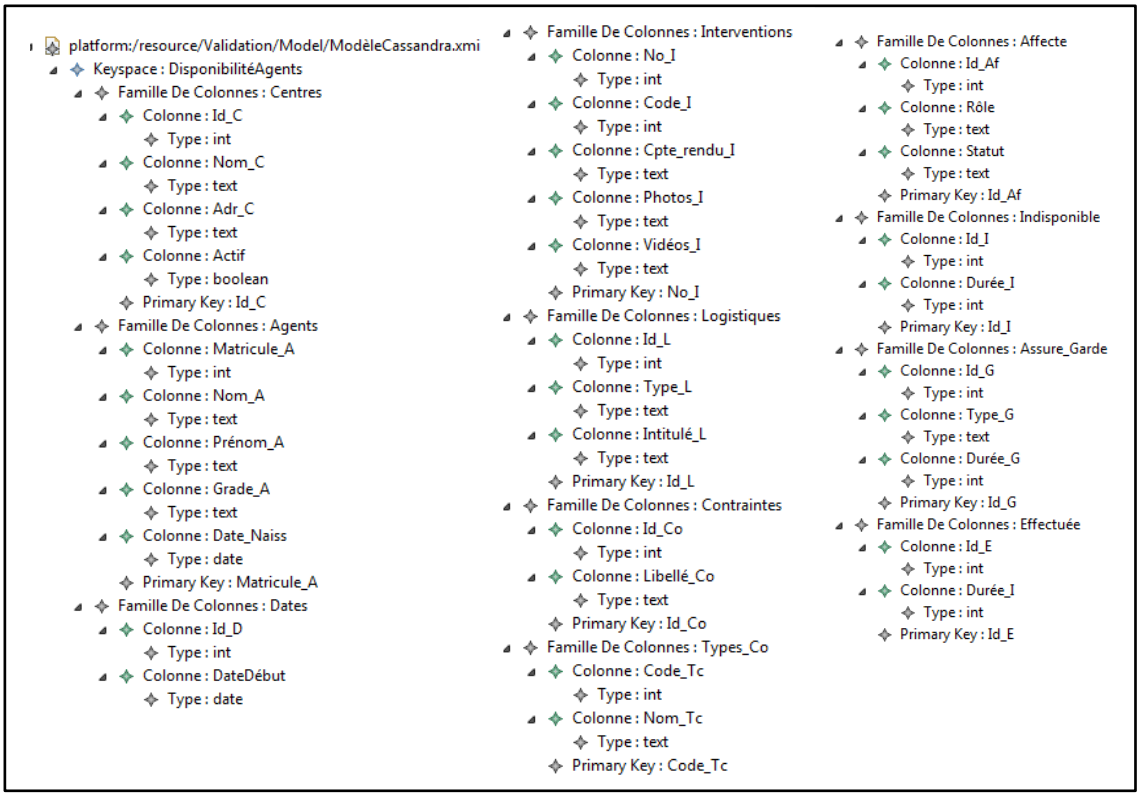


Figure 5.27 – Eléments d'implantation de la BD produits par notre système

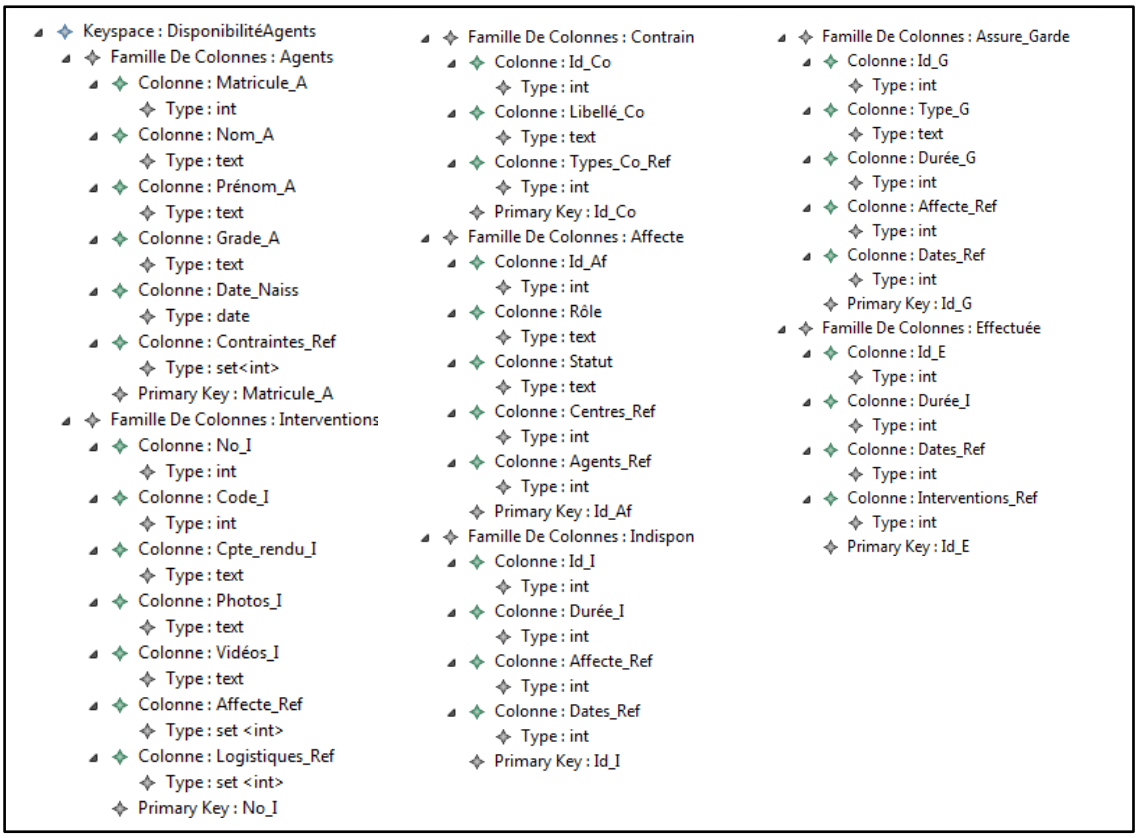


Figure 5.28 – Directives d'assistance produites par notre système

```
public class CassContraintes {
Cluster cluster;
Session session;
cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
session = cluster.connect("DiponibilitéAgents");public static boolean CI1 () {
 ResultSet result1 =session.execute("SELECT * FROM Agents");
  for (Row row:result1){
  ResultSet result2 =session.execute("SELECT * FROM Affecte_Agents where [Agents_Ref = row.getInt("Matricule_A")]");
 for (Row row:result2){
ResultSet result3 =session.execute("SELECT * FROM AssureGarde_Affecte where [Affecte_Ref = row.getInt("Affecte_Ref")]");
 for (Row row:result3){
 ResultSet result4 =session.execute("SELECT * FROM AssureGarde_Dates where [AssureGarde_Ref =
row.getInt("AssureGarde_Ref")]");
 for (Row row:result4){
 d1 = session.execute ("SELECT DateDébut FROM Dates where Id_D = [row.getInt("Dates_Ref")]");}}
 }}
 ResultSet result5 =session.execute("SELECT * FROM Agents");
 for (Row row:result5){
 ResultSet result6 =session.execute("SELECT * FROM Affecte_Agents where [Agents_Ref = row.getInt("Matricule_A")]");
 for (Row row:result6){
ResultSet result7 =session.execute("SELECT * FROM Indisponible_Affecte where [Affecte_Ref = row.getInt("Affecte_Ref")]");
 for (Row row:result7){
 d2 =session.execute("SELECT Durée_I FROM Indisponible where [Id_I = row.getInt("Indisponible_Ref")]");}
 }}
 ResultSet result8 =session.execute("SELECT * FROM Agents");
  for (Row row:result8){
  ResultSet result9 =session.execute("SELECT * FROM Affecte_Agents where [Agents_Ref = row.getInt("Matricule_A")]");
 for (Row row:result9){
ResultSet result10 =session.execute("SELECT * FROM Indisponible_Affecte where [Affecte_Ref = row.getInt("Affecte_Ref")]");
 for (Row row:result10){
 ResultSet result11 =session.execute("SELECT * FROM Indisponible_Dates where [Indisponible_Ref =
row.getInt("Indisponible_Ref")]");
 for (Row row:result11){
 d3 = session.execute ("SELECT DateDébut FROM Dates where Id_D = [row.getInt("Dates_Ref")]");}}
}}
if (d1>d3+d2) return True;
else return false;
}
session.close();
cluster.close();}
```

Figure 5.29 – Extrait du code de vérification des contraintes produit par notre système

5.3.3 Evaluation des résultats

Nous avons retenu deux critères, la qualité et le temps, pour comparer les résultats obtenus en mettant en œuvre successivement :

- la méthode manuelle de transformation de modèle,
- notre processus automatisé.

a. La qualité du modèle

Il s'agit d'évaluer la qualité des structures physiques de données résultantes. Autrement dit, on souhaite évaluer la pertinence de la transformation du modèle conceptuel (structures + contraintes) en un modèle physique NoSQL (structures + programmes).

Pour le décideur chargé d'implanter l'application « Juris-Dépôt », le résultat qu'il a obtenu manuellement a été comparé avec le modèle physique produit par notre processus automatisé ; les modèles sont équivalents.

Pour l'informaticien chargé de développer l'application « Disponibilités Des Agents », nous avons fait l'hypothèse que ses compétences lui permettaient d'obtenir manuellement un modèle physique pertinent (choix optimal des transformations). Ainsi la comparaison de son résultat avec celui produit par notre processus montre que les modèles générés sont équivalents. Les modèles obtenus ne sont pas identiques puisque les attributs de référence représentant les liens ne portent généralement pas les mêmes noms (voir les modèles présentés en section 5.3.2).

b. Le temps de transformation

D'autre part, nous avons comparé le temps nécessaire à notre processus automatisé avec le temps mis par le décideur et l'informaticien pour obtenir manuellement un résultat équivalent.

Pour un résultat (modèle physique et code de vérification des contraintes) quasiment identique, le temps de réalisation du processus a été considérablement réduit. Autrement dit, les transformations de modèles réalisées avec notre processus bénéficient d'un gain de temps significatif par rapport au processus manuel ; ceci est montré dans le tableau 5.2 ci-dessous.

<table>
<tr><th></th><th>« Juris-Dépôt »</th><th colspan="2">« Disponibilités Des Agents »</th></tr>
<tr><td rowspan="2">Processus Manuel</td><td rowspan="2">20 min</td><td>Modèle d'implantation</td><td>Code de vérification des contraintes</td></tr>
<tr><td>35 min</td><td>50 min</td></tr>
<tr><td rowspan="2">Processus Automatisé*</td><td rowspan="2">3 min</td><td>Modèle d'implantation</td><td>Code de vérification des contraintes</td></tr>
<tr><td>5 min</td><td>8 min</td></tr>
</table>

Tableau 5.2 – Temps de réalisation des processus manuel et automatisé

* Le temps tient compte de la saisie des données en entrée.

Bien entendu, le gain de temps entre le processus manuel et le processus automatisé s'accroit avec la complexité du modèle conceptuel de départ.

5.4 Conclusion

Dans ce chapitre, nous avons décrit la réalisation de notre prototype de transformation des modèles conceptuels UML. Ce prototype a permis de montrer la faisabilité de notre approche dirigée par les modèles. Il est composé de deux modules :

- Module de transformation du DCL : Object2NoSQL,
- Module de traduction des contraintes OCL : OCL2Java.

Le premier est chargé de transformer un DCL en un modèle physique NoSQL qui comporte : les éléments nécessaires à l'implantation de la BD NoSQL et un ensemble de directives d'assistance spécifiques au SGBD choisi par l'utilisateur. Le second module a pour objectif de compléter le premier en traduisant les contraintes associées au DCL transformé en des méthodes java. Ces contraintes sont exprimées en OCL.

Pour chaque module, nous avons présenté : l'entrée, la sortie, les transformations (conceptuel > logique > physique) et les différentes étapes de son implantation.

Pour tester le prototype, nous avons mené des expériences qui portent sur l'utilisation du modèle conceptuel de l'application médicale (présentée au chapitre 1). Pour ceci, nous avons instancié les métamodèles UML et OCL proposés dans les chapitres précédents pour générer des exemples d'instances (un DCL et des contraintes OCL). Ces dernières ont été fournies par la suite comme entrée des modules Object2NoSQL et OCL2Java.

Ce chapitre montre également la pertinence de nos propositions à travers une évaluation. Celle-ci a été effectuée dans le cadre d'un projet réalisé par la société TRIMANE. L'objectif est de comparer le temps de réalisation de notre processus automatisé avec le temps mis par un utilisateur pour obtenir manuellement un résultat équivalent (Modèle physique NoSQL et le code de vérification des contraintes).

Conclusion générale

Synthèse de nos travaux

Les travaux présentés dans ce mémoire de thèse se situent dans le contexte des Big Data, et plus particulièrement dans le cadre du stockage des données massives sur un SGBD NoSQL.

Ces systèmes sont bien plus adaptés que les systèmes relationnels pour gérer de gros volumes de données présentant des types et des formats variés. Cependant, leur utilisation soulève deux problèmes majeurs. D'une part le problème lié au passage d'un modèle conceptuel décrivant une BD massive vers un modèle d'implantation NoSQL. Ainsi la difficulté du processus d'implantation réside principalement dans :

- La complexité des structures de données massives qui prennent en compte la sémantique et qui doivent être stockées,
- La spécificité des modèles physiques d'implantation proposés par chaque SGBD NoSQL.

D'autre part, la gestion des contraintes d'intégrité assurée par ces SGBD est actuellement très limitée.

Dans ce mémoire, nous nous sommes intéressés au traitement de ces deux problématiques.

Pour ce faire, nous avons proposé une solution basée sur l'architecture MDA dont l'utilité est d'assister un utilisateur souhaitant stocker des données massives sur un SGBD NoSQL.

Notre solution MDA est composée de deux processus.

- Le processus Object2NoSQL de transformation d'un DCL est basé sur trois niveaux de modélisation : (1) conceptuel où les données sont formalisées par un DCL d'UML, (2) logique qui fait apparaître un modèle générique compatible avec les quatre types de SGBD NoSQL et (3) physique qui contient le modèle d'implantation propre à la plateforme NoSQL choisie par l'utilisateur. Notons qu'un modèle physique comporte deux volets : la description des structures de données et un ensemble de directives d'assistance. Le passage d'un niveau à l'autre est assuré par des transformations automatiques de type M2M (Model-To-Model) que nous avons formalisées avec le standard QVT. L'intérêt d'introduire un niveau logique entre les niveaux conceptuel et physique est de garantir l'indépendance de la description des données vis-à-vis des plateformes d'implantation. Ceci assure une certaine stabilité dans notre processus, simplifie les transformations et constitue un gain de temps notable lors des mises à jour physiques. En effet, à travers notre niveau logique, nous limitons les impacts liés aux évolutions technologiques des plateformes NoSQL, voire

à leur remplacement. Toute évolution des caractéristiques techniques d'un SGBD apparaitra au niveau physique mais n'affectera pas le niveau logique. Le passage logique vers physique sera alors relancé et il n'y aura aucune incidence sur le passage conceptuel vers logique.

- Le processus OCL2Java de traduction des contraintes a pour objet de généraliser le processus Object2NoSQL précédent. En effet, ce dernier prend uniquement en compte quelques contraintes (principalement les types de données et l'unicité des identificateurs) dans le modèle physique NoSQL généré. OCL2Java vise donc à intégrer d'autres contraintes plus complexes. Par analogie avec Object2NoSQL, nous avons défini OCL2Java en nous appuyant sur les trois niveaux conceptuel, logique et physique. Au niveau conceptuel, les contraintes sont exprimées avec le standard OCL. Compte tenu de la diversité des expressions OCL proposées par l'OMG, nous nous sommes limités à celles qui sont fréquemment utilisées. Il s'agit des expressions itératives, des expressions opération, des expressions conditionnelles et des expressions de définition. Il convient de noter que, dans la documentation de l'OMG, les expressions OCL sont définies informellement. Par conséquent, pour automatiser la transformation de ces expressions, nous avons dû les formaliser préalablement puis proposer un métamodèle. Pour chaque contrainte, le processus est réalisé en deux étapes successives qui produisent respectivement : (1) un modèle pivot de méthode java (niveau logique) puis (2) le code de la méthode à partir de son modèle (niveau physique). Plusieurs codes peuvent être produits à partir du même modèle ; chacun d'eux est spécifique à un SGBD parce qu'il contient des requêtes d'accès aux données exprimées avec le langage propre au SGBD.

Nous pouvons résumer les avantages de notre solution dans les points suivants :

- L'utilisation de MDA offre un cadre formel aux mécanismes de transformation des modèles,
- La formalisation de l'entrée des deux processus composant notre solution repose sur des standards de l'OMG : UML et OCL,
- Un modèle conceptuel décrit la sémantique des structures de données et des contraintes d'intégrité ; notre solution MDA prend en compte cette dualité,
- Les SGBD NoSQL stockent les données selon des principes d'implantation diversifiés ; notre solution offre à l'utilisateur le choix de ces différents modèles d'implantation,
- La généralisation de notre solution : l'implantation des structures et l'implantation des contraintes d'intégrité reposent sur l'architecture MDA et suivent la même démarche de transformation : conceptuel > logique > physique,
- L'utilisation du principe MDA de métamodèlisation permet de vérifier la conformité d'un modèle par rapport à un référentiel ;
- Les règles de transformation sont basées sur les normes de l'OMG : QVT et MOFM2T.

Afin de vérifier la faisabilité de notre solution, nous avons développé un prototype en utilisant la plateforme EMF (Eclipse Modeling Framework) [Budinsky et al., 2004]. Cette plateforme d'implantation présente un environnement complet et adapté à la métamodèlisation, la modélisation et la transformation des modèles. Elle dispose de l'ensemble d'outils nécessaires à la mise en œuvre de notre solution MDA. Nous avons ensuite effectué des expérimentations sur le DCL d'une application médicale afin de tester le fonctionnement des modules composant notre prototype.

En outre, notre solution a été validée dans un cadre professionnel par des ingénieurs de la Société Trimane spécialisée en informatique décisionnelle et située à Saint Germain en Laye.

Perspectives

Les perspectives envisageables à nos travaux portent principalement sur trois points.

Le premier concerne l'enrichissement de notre solution de transformation des modèles conceptuels UML. Actuellement, les deux processus Object2NoSQL et OCL2Java composant notre solution MDA sont limités aux éléments de base qui sont fréquemment utilisés. Nous souhaitons proposer des extensions qui portent principalement sur les types de liens et d'expressions OCL supportés par chaque processus. En effet, dans son état actuel, le processus Object2NoSQL de transformation du DCL considère uniquement les types de liens suivants : l'association, l'agrégation, la composition et l'héritage. D'autre part en ce qui concerne le processus OCL2Java de traduction des contraintes, les expressions OCL considérées correspondent aux expressions itératives (Select, Exists, forAll avec un et deux itérateurs et Collect), aux expressions opération, aux expressions conditionnelles (If-Then-Else et Implies) et aux expressions de définition (Let…In). Il serait alors possible de prendre en compte d'autres types de liens, notamment la dépendance et d'autres expressions OCL telles que les préconditions et les postconditions. Ceci nécessite typiquement (1) l'extension de la source et de la cible de chaque processus en formalisant les concepts liés aux nouveaux éléments que nous souhaitons ajoutés, puis (2) la mise à jour de l'ensemble des métamodèles et des règles de transformation utilisés.

Le deuxième point vise à automatiser le choix d'implantation des liens (voir section 3.3.4). Il s'agit du cas où le SGBD NoSQL retenu par l'utilisateur offre plusieurs solutions d'implantation des liens (MongoDB, Cassandra, etc.). Les caractéristiques de l'application seront saisies à l'entrée du processus et permettront à celui-ci d'effectuer les choix les plus appropriés. Notre processus sera donc, quel que soit le SGBD, entièrement automatique.

Le troisième point est lié à l'évolution des besoins de l'utilisateur. Cette évolution nécessite une mise à jour des modèles de données (ajout, modification ou suppression d'un élément du modèle). Elle peut se situer à deux niveaux : conceptuel et physique. Selon les principes de MDA, si les besoins évoluent au niveau conceptuel, il suffit de relancer le processus de transformation pour aboutir au nouveau modèle d'implantation correspondant. Lorsqu'il s'agit d'une évolution du modèle physique, il est nécessaire d'envisager un processus de retro-conception pour adapter le modèle conceptuel aux modifications apportées sur le modèle physique. Ceci est utile pour qu'un utilisateur puisse formuler ses requêtes d'accès aux données en s'appuyant sur un modèle conceptuel à jour.

Bibliographie

Abdelhedi, F., AIT Brahim, A., & Zurfluh, G. (2018a). Formalizing the Mapping of UML Conceptual Schemas to Column-Oriented Databases. *International Journal of Data Warehousing and Mining (IJDWM)*, *14*(3), 44-68.

Abdelhedi,F, Ait Brahim, A., Faten Atigui, Gilles Zurfluh (2018b). Towards Automatic Generation of NoSQL Document-Oriented Models. In : International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2018).

Abdelhedi, F., Ait Brahim, A., & Zurfluh, G. (2018c). Traduction Automatique de contraintes OCL dans une BD NoSQL , (EDA 2018).

Abdelhedi, F., Ait Brahim, A., Atigui, F., & Zurfluh, G. (2017a). Logical Unified Modeling For NoSQL DataBases. In *19th International Conference on Enterprise Information Systems* (ICEIS 2017) (pp. pp-249).

Abdelhedi,F., Ait Brahim, A., Faten Atigui, Gilles Zurfluh (2017b). Modeling Framework for NoSQL Systems. In : International Database Engineering & Applications Symposium (IDEAS 2017).

Abdelhedi, F., Ait Brahim, A., Atigui, F., & Zurfluh, G. (2017c). MDA-Based Approach for NoSQL Databases Modelling. In *International Conference on Big Data Analytics and Knowledge Discovery* (DaWaK 2017) (pp. 88-102). Springer, Cham.

Abdelhedi, F., Ait Brahim, A., Atigui, F., & Zurfluh, G. (2017d). UMLtoNoSQL: Automatic Transformation of Conceptual Schema to NoSQL Databases. In *Computer Systems and Applications* (AICCSA 2017), *2017 IEEE/ACS 14th International Conference on* (pp. 272-279). IEEE.

Abdelhedi, F., Ait Brahim, A., & Zurfluh, G. (2016a). L'implantation de sources de données dans un système NoSQL: formalisation des règles de passage conceptuel/logique (EDA 2016).

Abdelhedi, F., Ait Brahim, A., Atigui, F., & Zurfluh, G. (2016b). Processus de transformation MDA d'un schéma conceptuel de données en un schéma logique NoSQL (INFORSID 2016).

Abdelhedi, F., Ait Brahim, A., Atigui, F., & Zurfluh, G. (2016c). Big Data and Knowledge Management: How to Implement Conceptual Models in NoSQL Systems?. In IC3K-KMIS, 2016 (pp. 235-240).

Bartels, D., Berler, M., Eastman, J., Gamerman, S., Jordan, D., Springer, A., & Wade, D. (1997). *The object database standard: ODMG 2.0* (Vol. 131). R. G. G. Cattell, & D. K. Barry (Eds.). Los Altos, CA: Morgan Kaufmann Publishers.

Silberztein, M., Atigui, F., Kornyshova, E., Métais, E., Meziane. F. (2018). Natural Language Processing and Information Systems - 23rd International Conference on Applications of Natural Language to Information Systems, NLDB

2018, Paris, France, June 13-15, 2018, Proceedings, Springer 2018, ISBN 978-3-319-91946-1.

Gomez, P., Claudia Roncancio, C., & Casallas, R. (2018). Métriques structurelles pour l'analyse de bases orientées documents. In INFormatique des ORganisations et Systèmes d'Information et de Décision 2018.

Abelló, A. (2015, October). Big data design. In Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP (pp. 35-38). ACM.

Herrero, V., Abelló, A., & Romero, O. (2016, November). NOSQL design for analytical workloads: variability matters. In International Conference on Conceptual Modeling (pp. 50-64). Springer, Cham.

Daniel, G., Sunyé, G., & Cabot, J. (2016, November). UMLtoGraphDB: mapping conceptual schemas to graph databases. In International Conference on Conceptual Modeling (pp. 430-444). Springer, Cham.

Hutchinson, J., Rouncefield, M., & Whittle, J. (2011, May). Model-driven engineering practices in industry. In Proceedings of the 33rd International Conference on Software Engineering (pp. 633-642). ACM.

Bézivin, J., & Gerbé, O. (2001, November). Towards a precise definition of the OMG/MDA framework. In Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on (pp. 273-280). IEEE.

Angadi, A. B., Angadi, A. B., & Gull, K. C. (2013). Growth of New Databases & Analysis of NOSQL Datastores. International Journal of Advanced Research in Computer Science and Software Engineering, 3, 1307-1319.

Kumar, R., Charu, S., & Bansal, S. (2015). Effective way to handling big data problems using NoSQL Database (MongoDB). Journal of Advanced Database Management & Systems, 2(2), 42-48.

Arora, R., & Aggarwal, R. R. (2013). Modeling and querying data in mongodb. International Journal of Scientific and Engineering Research, 4(7), 141-144.

Abadi, D. J., Madden, S. R., & Hachem, N. (2008). Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (pp. 967-980). ACM.

Douglas, L., 2001. 3d data management: Controlling data volume, velocity and variety. *Gartner*. *Retrieved*, *6*, 2001.

Han, J., Haihong, E., Le, G., & Du, J. (2011, October). Survey on NoSQL database. In Pervasive computing and applications (ICPCA), 2011 6th international conference on (pp. 363-366). IEEE.

Demchenko, Y., Ngo, C., & Membrey, P. (2013). Architecture framework and components for the big data ecosystem. *Journal of System and Network Engineering*, 1-31.

Gantz, J., & Reinsel, D. (2011). Extracting value from chaos. *IDC iview*, *1142*(2011), 1-12.

Combemale, B. (2008). Ingénierie Dirigée par les Modèles (IDM)--État de l'art.

Seidewitz, E. (2003). What Models Mean. IEEE Software, 20(5):26–32.

Chevalier, M., El Malki, M., Kopliku, A., Teste, O., & Tournier, R. (2015a). Implementing Multidimensional Data Warehouses into NoSQL, 17th Int. In *Conf. on Enterprise Information Systems, vol. DISI, 172-183*.

Dehdouh, K., Bentayeb, F., Boussaid, O., & Kabachi, N. (2015, January). Using the column oriented NoSQL model for implementing big data warehouses. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (p. 469). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

Li, C. (2010, July). Transforming relational database into HBase: A case study. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on* (pp. 683-687). IEEE.

Vajk, T., Feher, P., Fekete, K., & Charaf, H. (2013, December). Denormalizing data into schema-free databases. In *Cognitive Infocommunications (CogInfoCom), 2013 IEEE 4th International Conference on* (pp. 747-752). IEEE.

Li, Y., Gu, P., & Zhang, C. (2014, April). Transforming UML class diagrams into HBase based on meta-model. In *Information Science, Electronics and Electrical Engineering (ISEEE), 2014 International Conference on* (Vol. 2, pp. 720-724). IEEE.

Budinsky, F., Steinberg, D., Ellersick, R., Grose, T. J., & Merks, E. (2004). Eclipse modeling framework: a developer's guide. Addison-Wesley Professional.

Blanc, X., & Salvatori, O. (2011). MDA en action: Ingénierie logicielle guidée par les modèles. Editions Eyrolles.

Object Management Group (OMG). MOF Model To Text Transformation Language (MOFM2T), version 1.0, January 2008. URL http://www.omg.org/spec/MOFM2T/1.0/PDF.

Liu, D., & McCluskey, T. L. (2000). *The OCL Language Manual*. Version 1.2. Technical report, Department of Computing and Mathematical Sciences, University of Huddersfield.

Anton, A. I. (1996, April). Goal-based requirements analysis. In Requirements Engineering, 1996. Proceedings of the Second International Conference on (pp. 136-144). IEEE.

Li, Y., Gai, K., Qiu, L., Qiu, M., & Zhao, H. (2017). Intelligent cryptography approach for secure distributed big data storage in cloud computing. Information Sciences, 387, 103-115.

Mpinda, S. A. T., Maschietto, L. G., & Bungama, P. A. (2015). From relational database to columnoriented nosql database: Migration process. International Journal of Engineering Research & Technology (IJERT), 4, 399-403.

Karnitis, G., & Arnicans, G. (2015, June). Migration of relational database to document-oriented database: structure denormalization and data transformation. In Computational Intelligence, Communication Systems and Networks (CICSyN), 2015 7th International Conference on (pp. 113-118). IEEE.

Darmont, J., Boussaid, O., Ralaivao, J. C., & Aouiche, K. (2007). An architecture framework for complex data warehouses. arXiv preprint arXiv:0707.1534.

Tanasescu, A., Boussaid, O., & Bentayeb, F. (2005, January). Preparing complex data for warehousing. In Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on (p. 30). IEEE.

Midouni, S. A. D., Darmont, J., & Bentayeb, F. (2009). Approche de modélisation multidimensionnelle des données complexes: Application aux données médicales. In 5èmes Journées francophones sur les Entrepôts de Données et l'Analyse en ligne (EDA 09) (pp. 155-166).

Pedersen, T. B., & Jensen, C. S. (1999, March). Multidimensional data modeling for complex data. In Data Engineering, 1999. Proceedings. 15th International Conference on (pp. 336-345). IEEE.

Scabora, L. C., Brito, J. J., Ciferri, R. R., & Ciferri, C. D. D. A. (2016). Physical data warehouse design on NoSQL databases OLAP query processing over HBase. In International Conference on Enterprise Information Systems, XVIII. Institute for Systems and Technologies of Information, Control and Communication-INSTICC.

Freitas, M. C., Souza, D. Y., & Salgado, A. C. (2016, April). Conceptual Mappings to Convert Relational into NoSQL Databases. In ICEIS (1) (pp. 174-181).

Chevalier, M., El Malki, M., Kopliku, A., Teste, O., & Tournier, R. (2015b). Benchmark for OLAP on NoSQL technologies comparing NoSQL multidimensional data warehousing solutions. In 9th IEEE International Conference on Research Challenges in Information Science, RCIS 2015, 480-485.

Rocha, L., Vale, F., Cirilo, E., Barbosa, D., & Mourão, F. (2015). A Framework for Migrating Relational Datasets to NoSQL1. Procedia Computer Science, 51, 2593-2602.

Faster, D. S. (2014). Pentaho Data Integration.

Lee, C. H., & Zheng, Y. L. (2015, June). Automatic SQL-to-NoSQL schema transformation over the MySQL and HBase databases. In Consumer Electronics-Taiwan (ICCE-TW), 2015 IEEE International Conference on (pp. 426-427). IEEE.

Batra, S. (2016). MONGODB versus SQL: a case study on electricity data. In Emerging Research in Computing, Information, Communication and Applications (pp. 297-308). Springer, Singapore.

Schram, A., & Anderson, K. M. (2012, October). MySQL to NoSQL: data modeling challenges in supporting scalability. In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity (pp. 191-202). ACM.

Feng, W., Gu, P., Zhang, C., & Zhou, K. (2015, December). Transforming UML Class Diagram into Cassandra Data Model with Annotations. In Smart City/SocialCom/SustainCom (SmartCity), 2015 IEEE International Conference on (pp. 798-805). IEEE.