Traduction Automatique de contraintes OCL dans une BD NoSQL

Fatma Abdelhedi*,**, Amal Ait Brahim*, Gilles Zurfluh*

*IRIT - Université Toulouse Capitole - France
{Fatma.Abdelhedi, Amal.Ait−Brahim, Gilles.Zurfluh }@irit.fr,
**CBI^2 - Société Trimane - Saint Germain-En-Laye - France
Fatma.Abdelhedi@trimane.fr

Résumé. Les développeurs d'applications Big Data mettent en œuvre des systèmes NoSQL pour stocker et exploiter des BD massives. Ils transforment généralement 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 et la quasi-absence de mécanismes de gestion des contraintes. L'objet de nos travaux est donc d'assister le développeur en automatisant en grande partie le processus de transformation des modèles. Pour ceci, nous avons utilisé l'architecture MDA. À partir d'un modèle conceptuel qui décrit la structure des données et un ensemble de contraintes sur ces données, nous proposons des règles de dérivation pour générer (1) un modèle d'implantation destiné à une plateforme NoSQL et (2) le code permettant de vérifier les contraintes. Le premier point a été traité dans des travaux antérieurs. Dans cet article, nous étudions le deuxième point qui vise à proposer un processus automatique de traduction de contraintes. Ce processus est réalisé en deux étapes qui correspondent aux passages conceptuel vers logique puis logique vers physique. Cet article est consacré à la description du premier passage.

1 Introduction

Les développeurs d'applications Big Data stockent généralement les données sur des plateformes NoSQL (Abelló, 2015). Le point de départ du processus de stockage est constitué d'un modèle conceptuel qui contient (1) la structure des données et (2) un ensemble de contraintes destinées à assurer l'intégrité de ces données lors de leurs mises à jour (Herrero et al., 2016). Comme il est précisé dans la section « Etat de l'art », seuls quelques travaux ont présenté des processus de transformation de modèles conceptuels en modèles NoSQL contenant des contraintes d'intégrité.

Le maintien de la cohérence des données stockées constitue l'un des atouts majeurs des SGBD ; en effet, ceci garantit aux utilisateurs la qualité des données et, par conséquent, la qualité des systèmes d'information. Les systèmes NoSQL actuels offrent peu de mécanismes pour garantir cette cohérence (Angadi et al., 2013; Daniel et al., 2016). Ils prennent bien sûr en compte certaines contraintes dans le modèle physique, comme le type des valeurs ou l'identification des lignes dans une table (Cattell, 2011). Par contre, une contrainte telle que la prise

en compte des relations entre les tables est à la charge du développeur ; celui-ci doit décrire le mode d'implantation de chaque relation dans le modèle physique puis écrire le code.

Le développeur doit donc implanter les contraintes d'intégrité qui ne peuvent pas être prises en compte par le système NoSQL. Dans cet article, nous proposons un processus d'assistance permettant au développeur d'intégrer automatiquement les contraintes dans le processus de transformation du modèle conceptuel. L'ensemble de nos travaux s'appuie sur une application médicale Big Data d'où sont tirés les exemples illustrant les processus présentés. Il s'agit de la mise en œuvre de programmes nationaux et internationaux pour le suivi de cohortes de patients atteints de pathologies graves. Une description détaillée de notre étude de cas est donnée par Abdelhédi et al. (2017).

Nous présentons dans la section 2 les grandes lignes de notre approche de transformation de modèles conceptuels en modèles NoSQL. Les sections 3 et 4 détaillent les étapes du processus de transformation des contraintes. La section 5 décrit une expérimentation du processus proposé. Enfin, la section 6 positionne nos travaux par rapport à l'état de l'art.

2 Approche générale

Cet article se focalise sur le processus de transformation de contraintes d'intégrité associées aux données. Pour bien situer ce processus lié aux contraintes, il est nécessaire de connaitre le cadre dans lequel il s'insère ; il s'agit de la démarche UML2NoSQL présentée dans des travaux précédents (Abdelhédi et al., 2017). Nous rappelons uniquement les grandes lignes de cette démarche de transformation de modéles. UML2NoSQL est une démarche de type MDA permettant de générer des modèles physiques NoSQL à partir d'un modèle conceptuel UML (DCL). 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 adapté à un système NoSQL choisi par le développeur. 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 ou graphes. 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. 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 cet article est de compléter le processus en prenant en compte d'autres catégories de contraintes plus complexes.

Une contrainte est une expression donnant une restriction ou des informations complémentaires sur un élément du modèle. Elle peut porter par exemple sur une table, une ligne, un attribut ou une relation entre deux lignes. Son application assure la cohérence de la base de données lors des mises 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 et qui a été normalisé par l'OMG [1] ; il s'agit du langage OCL (Object Constraint Language) défini dans la section 3.

L'idée de cet article est de transformer des contraintes OCL en méthodes Java qui seront exécutées au niveau physique. Pour réaliser cette transformation, nous proposons le processus OCL2Java qui prend en entrée des contraintes OCL et fournit en sortie des méthodes programmées en Java. Comme le montre la figure 1, pour chaque contrainte, le processus est

1. http ://www.omg.org/spec/OCL/2.4/

réalisé en deux étapes successives qui produisent respectivement : (1) un modèle de méthode java : il s'agit d'un modèle pivot compatible avec des plateformes NoSQL de types différents (colonnes, documents et graphes). Cette compatibilité est principalement liée au fait que ce modèle fait abstraction du langage de requête de chaque système NoSQL et (2) le code de la méthode : à partir du modèle précédent, le processus génère le code java de la méthode. 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 propre à chaque système. 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 de chaque système NoSQL. Ainsi, toute évolution de langages ou tout changement du système nécessitera une nouvelle transformation logique/physique ; mais il sera inutile de remonter au niveau supérieur où sont exprimées les contraintes OCL. La figure 1 montre les différents composants du processus OCL2Java ainsi que la généralisation des processus de transformation des modèles et de transformation des contraintes. Comme nous l'avant précisé ci-dessus, 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.

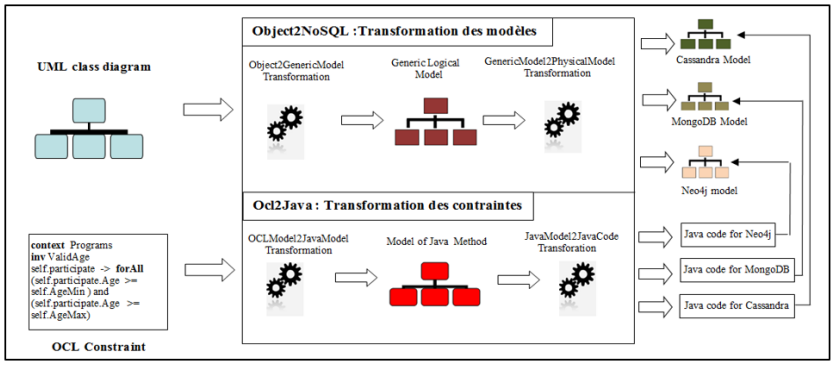

FIG. 1 – *Aperçu de la démarche UML2NOSQL*

3 Transformation OCL2JavaModel

Dans cette section, nous présentons la première étape dans notre processus de transformation des contraintes. Il s'agit de la transformation OCL2JavaModel qui traduit une expression OCL en un modèle d'une 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.

3.1 La source : Invariants 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 notre travail, nous avons choisi d'imposer un nom à tous les invariants.<expressionsOCL> est le corps de l'invariant qui est composé d'une ou plusieurs expressions OCL. Le contexte est utilisé dans une expression grâce au mot-clef $self$.

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 [2]. 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 donnons des exemples pour illustrer les différents concepts qui les composent.

3.1.1 Expression de définition : Let-In

Cette expression est utilisée pour définir 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.

Définition 2. Une expression ex^{letIn} est définie par (ex°,ex^{u}) où :

- ex^{o} = ($operande_1$, $operateur$, $operande_2$) est une expression opération (cf. section 3.1.4), 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 :

2. http ://www.omg.org/spec/OCL/2.4/

- $operande_2 = self.a^{c^{ctx}}$, 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 (voir section 3.1.3) ; il s'agit généralement de l'opération $forAll()$ (voir Définition 6).
- $operande_2 = ex^o$, où ex^o est une expression opération (voir section 3.1.4) définie comme suit : $ex^o = self.role^c -> operation()$,

- ex^u est une expression utilisant la variable définie par ex^o. Généralement, ex^u correspond à une expression conditionnelle (voir Définitions 3 et 4).

3.1.2 Expressions conditionnelles : If et Implies

Dans un invariant, certaines expressions sont dépendantes d'autres expressions. Pour exprimer ceci, OCL offre les deux formes suivantes :

Syntaxe de la forme1 : *if* <expression_1> *then* <expression_2> *else* <expression_3>

Définition 3. Une expression If-Then-Else ex^{cdt} est définie par (ex^{cdt}, ex^{then}, ex^{else}). Chacune des trois expressions ex^{cdt}, ex^{then}, ex^{else} correspond à une expression opération ex^o. Cette dernière peut être définie selon les deux 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 (voir Définition 2),
    - $operande_1 = self.a^{c^{ctx}}$ (voir Définition 1 cas où $operande_2 = self.a^{c^{ctx}}$),
  - $operateur$ correspond à un opérateur OCL prédéfini.
  - $operande_2$ est une valeur de type OCL prédéfini.
- $ex^o = self.role^c -> operation()$ (voir Définition 7).

Syntaxe de la forme 2 : <expression_1> *implies* <expression_2>

Définition 4. Une expression implies ex^{ip} est définie par (ex_1, ex_2), où $\forall i \in [1,2]$, ex_i est soit une expression opération ex^o (voir section 3.1.4) , soit une expression requête ex^r (voir section 3.1.3). Dans le premier cas ($ex_i = ex^o$), deux formes peuvent être considérés :

- 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 $self.role^c.a^c$ (voir 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 (voir Définition 7).

Dans le deuxième cas ($ex_i = ex^r$), il s'agit généralement d'une expression utilisant l'opération $forAll()$ (voir Définition 6).

3.1.3 Expressions itératives / Expressions requête

Les expressions itératives, appelées également les expressions requêtes, sont des expressions OCL permettant d'exprimer des contraintes sur les éléments d'une collection. Leur syntaxe générale 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 : $< element > . < propriete >$. Pour cela, il est nécessaire de déclarer $< element >$ dans l'expression $< expression\_declataion >$; $< element >$ jouera dans ce cas le rôle d'un itérateur.

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 $\{o_1, ..., o_n\}$, 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 est de 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 cet, nous considérons l'opération
- $forAll$ (avec un ou deux itérateurs),
- 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 $ex^s.a.forAll$

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. 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 (n'en 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).

Définition 6. Dans une opération $forAll$ les deux cas suivants sont considérés :

- 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 et a un attribut de o, $operateur$ correspond à un opérateur OCL prédéfini. $operande_2$ est une valeur de type OCL prédéfini.
- Si le nombre d'itérateur = 2 : ex^a = $ex^{ip}.ex_2$, avec 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 retournés par ex^s.

3.1.4 Expressions opération

Une expression opération est un type d'expressions OCL qui est utilisé pour définir les expressions : Let-In, If-Then-Elseet Implies et les expressions requêtes, 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 classifier 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, les opérations de base sur les collections qui sont fréquemment utilisées dans le cadre d'une expression requête. Nous complétons dans cette section la présentation des opérations OCL prédéfinies.

Définition 7. . 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, ..., 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 (voir section 3.1.3).
- $operation$ 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 : {$size()$, $isEmpty()$, $notEmpty()$}.

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.

Définition 8. 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 ci-dessus en fonction des expressions dans lesquelles ils sont utilisés.

3.2 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. 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).

Définition 9. 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,
- T^r est le type de la valeur retournée par m. Dans notre cas c'est le type Booléen,
- N est le nom désignant m,
- PR = pr_1, ..., 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.
- IS = {is_1,.. , 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.

3.2.1 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 10. 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.

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

- $condition$: est une instruction logique is^l (voir Définition 13),
- IS^v = {is^v_1, .., is^v_n} : est un ensemble d'instructions à exécuter si $condition$ = vrai,
- IS^f = {is^f_1, .. , is^f_m} : est un ensemble d'instructions à exécuter si $condition$=faux.

3.2.3 Instruction logique

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

Définition 12. Une instruction logique is^l peut être définie selon les 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. 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.

3.2.4 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 13. Une boucle foreach b^{fe} est définie par (is^{di}, $entete^{b^{fe}}$, $corps^{b^{fe}}$), 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) (voir Définition 11), $entete^{b^{fe}}$: 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, .. , is_n} : est un ensemble d'instructions java formant le corps de b^{fe}.

3.3 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 $ex \in i.EX$ est transformée en une instruction $is \in m.IS$. 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 expression 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 transformé 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 système 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 la règle 17. **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 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 système NoSQL. Nous appliquons le même principe pour générer la signature de m^{obj}dans une expression ex^{ite} si ex^o = $self.role^c$ -> $operation()$. **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(voir Définition 3). 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 $operande_2$ = N^v ou v_1 = m^{att} si $operande_2$ = $self.a^{c^ctx}$ (la signature de m^{att}est générée en appliquant la règle 5), $operateur^C.N$ = " == " 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. 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 (voir Définition 4). Selon le cas, nous appliquons les règles 14 ou 16. **R14 :** Dans une expression ex^{ip}, si ex_i = ex^o, avec $i \in$ [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 $operande_2$ et $operande_2$. **R16 :** Dans une expression ex^{ip}, si ex^o = $self.role^c$ -> $operation()$ alors nous appliquons la règle 12 et si ex_i =ex^r, avec $i \in$ [1,2], alors nous appliquons la règle 17. **R17 :** Une expression requête ex^r est transformée en une boucle foreach b^{fe} = (is^{di}, $entete^{b^{fe}}$, $corps^{b^{fe}}$), où : $b^{fe}.is^{di}$, $b^{fe}.entete^{b^{fe}}$ et $b^{fe}.corps^{b^{fe}}$ sont générés en appliquant respectivement les règles 18, 19 et 20. **R18 :** Dans une expression ex^r, si $operation.N$ = "$forAll$", alors $b^{fe}.is^{di}$ est définie ainsi : $is^{di}.T^v$ = "$Collection$", $is^{di}.N^v$ = "$resultat$", $is^{di}.N^op$ = " = ", $id^{di}.Vl^v$ = "m^{obj}" (voir R8 pour la définition de m^{obj}), où : $m^{obj}.pr_1$ = $ex^d.T$. **R19 :** Dans une expression ex^r, si $operation.N$ = $forAll$, alors $b^{fe}.entete^{b^{fe}}$ est définie ainsi : $entete^{b^{fe}}$. N^{cll} = "$resultat$", $entete^{b^{fe}}$. T^o = "$Object$" et $entete^{b^{fe}}$. N^o = "$object$". **R20 :** Dans une expression ex^r, si $operation.N$ = "$forAll$" alors $b^{fe}.corps^{b^{fe}}$ = is^d, is^{if}, où is^d. T^v = "$Boolean$", $is^d.N^v$ = "out", $is^{if}.condition$ est 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 générée en fonction du type de l'attribut a indiqué dans ex^a . $operande_2$.

4 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 système NoSQL choisi par le développeur. Le passage du modèle de méthode java vers le code correspondant est assuré par un ensemble de règles de transformation formalisées en MOFM2T (Oldevik, 2006). Par manque de place, nous avons consacré cet article à la transformation OCL2JavaModel qui correspond au passage conceptuel vers logique. Nous ne décrivons donc pas le passage logique vers physique.

5 Expérimentation

Dans cette section, nous évoquons les techniques que nous avons utilisées pour mettre en œuvre la démarche présentées dans la figure 1. Etant donné que notre approche est dirigée par les modèles, nous avons utilisé un environnement technique adapté à la modélisation, la métamodèlisation et la transformation des modèles. Nous avons eu recours à la plateforme Eclipse Modeling Framework (EMF) qui utilise Ecore pour créer et manipuler les modèles. Notre choix du langage de transformation a été fondé sur des critères spécifiques à notre démarche. En effet, l'outil doit être intégré dans l'environnement EMF pour qu'il soit utilisé aisément avec les outils de modélisation et de métamodélisation. Ainsi, nous avons utilisé le langage QVT. La figure 4 montre un extrait du code QVT assurant la génération du modèle de méthode java à partir d'un invariant OCL.

Les règles de transformation que nous avons définies dans cet article permettent d'obtenir un modèle logique indépendant de toute plateforme d'implantation. Ce principe assure l'in-

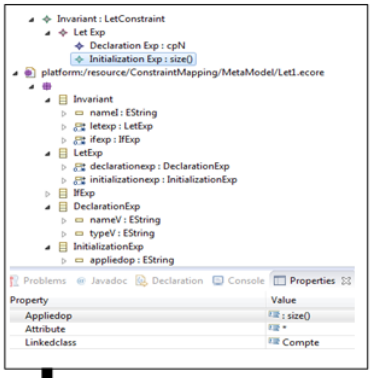

FIG. 2 – *SOURCE MODEL*

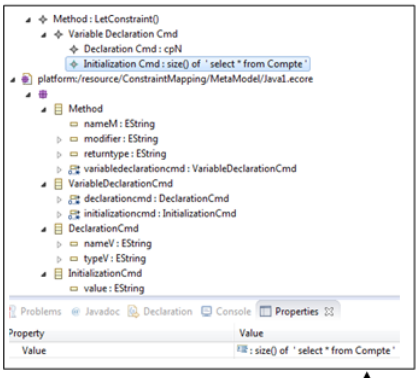

FIG. 3 – *TARGET MODEL*

```
modeltype OclM uses "http://OCLModel.com";
modeltype JavaM uses "http://JavaModel.com";
transformation OCL2JavaModel Transformation(in Source: OclM, out Target:
JavaM);
main() {
Source.rootObjects()[Invariant] -> map toMethod();}
mapping Invariant::toMethod():Method{
nameM := self.nameI+"()";
modifier:="public";
returntype:="Boolean";
variabledeclarationcmd:=self.letexp -> map toVariableDeclarationCmd();}
mapping OclM
::LetExp::toVariableDeclarationCmd():JavaM::VariableDeclarationCmd{
declarationcmd:=self.declarationexp -> map toDeclarationCmd();
initializationcmd:=self.initializationexp   -> map toInitializationCmd();}
mapping OclM ::DeclarationExp::toDeclarationCmd():JavaM::DeclarationCmd{
nameV:=self.nameV;
typeV:=self.typeV;}
mapping OclM ::InitializationExp::toInitializationCmd():JavaM::InitializationCmd{
value:=self.appliedop +" of "+" ' "+ "select " + self.attribute + " from "+
```

FIG. 4 – *REGLES QVT*

```
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.ResultSet;;
import com.datastax.driver.core.Session;
public class CassandraFile {
public static void connector(){
Cluster cluster; Session session;
cluster = Cluter.builder().addContactPoint("127.0.0.1").build();
session = cluster.connect("keyspace1");}
Public boolean exists (list l){
for (Iterator i = l.iterator(); e.hasNext();){
ElementType element = i.next(); if (exInJava){
return true;}} return false;}
Public list select(Collection cll){ List result;
for (Iterator i = cll.iterator(); e.hasNext();){
ElementType element = i.next();
if (exInJava) {result.add(element);}} return result;}
```

```
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.Mongo;
import  com.mongodb.BasicDBObject;
public class MongoDBFile {
public static void connector(){
DBCollection db=null;
try{
mongo = new Mongo("localhost",27017);
}catch (UnknownHostException e){ e.printStackTrace(); }
db=mongo.getDB("database1");}
Public Boolean forAll(Collection cll){
for (Iterator i = l.iterator(); e.hasNext();){
if (!exInJava){ return false;}}
return true;}
Public void predefinedOp(){
```

FIG. 5 – *Code java pour Cassandra et MongoDB*

dépendance du niveau logique face aux évolutions techniques des systèmes NoSQL. Ainsi, plusieurs codes peuvent être produits à partir du même modèle ; chacun d'eux dépend des caractéristiques techniques propres au système NoSQL choisi. La figure 5 présente des extrais de codes générés pour les plateformes Cassandra et MongoDB.

6 Etat de l'art et Positionnement de nos travaux

Récemment, des processus ont été proposés en vue d'implanter des BD sur des systèmes NoSQL. Ces études ont généralement porté sur la transformation des modèles. Dans le contexte des entrepôts de données, l'article (Dehdouh et al., 2015) traite de l'implantation des entrepôts de données 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. Toutes les trois utilisent les spécificités des systèmes Big Data en privilégiant certains choix de stockage des faits et des dimensions. Le passage vers le modèle physique n'est pas étudié mais celui-ci est montré au travers d'une expérimentation particulière sur le système HBase ; les performances des trois solutions sont comparées. D'autres travaux (Li, 2010; Vajk et al., 2013) ont étudié les mécanismes d'implantation d'une BD relationnelle sur des systèmes NoSQL. 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. Vajk et al. (2013) traite le passage d'un modèle relationnel vers le modèle orienté-documents de MongoDB. A notre connaissance, peu de travaux Li et al. (2014); Daniel et al. (2016) ont étudié la transformation d'un modèle conceptuel UML en un modèle physique NoSQL. 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. 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. A présent, nous effectuons le positionnement de nos travaux au regard des articles de recherche que nous venons de présenter et dont les problématiques sont proches des nôtres. L'article (Dehdouh et al., 2015) s'inscrit dans le contexte de l'entreposage des données puisqu'il étudie les règles de passage d'un modèle multidimensionnel en un modèle physique NoSQL. 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 des 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 et d'héritage ainsi que des classes d'associations. Par ailleurs les auteurs de (Li, 2010) et Vajk et al. (2013) traitent de la transformation d'un modèle relationnel en un modèle orienté-colonnes HBase. 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 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) 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) et orienté-graphes dans Daniel et al. (2016)). Enfin, concernant les contraintes d'intégrité, le processus proposé dans Li et al. (2014) n'en tient pas compte. Dans le travail Daniel et al. (2016), une fois le modèle orienté-graphes est créé, une autre transformation est effectuée pour traduire les expressions OCL définies au niveau conceptuel en des requêtes exprimées dans le langage Gremlin qui est un langage 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 compatibles avec les trois catégories de systèmes.

7 Conclusion

Dans cet article, nous avons proposé un processus automatique destiné à aider un développeur pour traduire des contraintes OCL sur un système NoSQL. 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. Notre processus consiste en une chaîne de transformations qui utilise un modèle pivot pour chaque contrainte OCL. Ce modèle est compatible avec les trois types de systèmes NoSQL : colonnes, documents et graphes et fait abstraction du langage de requête associé à chaque système NoSQL. Ainsi, toute évolution ou changement du système préservera le niveau logique en impactant uniquement sa transformation vers le niveau physique. Dans le travail (Daniel et al., 2016) dont la solution proposée est proche de la nôtre, les transformations n'incluent pas un niveau logique et n'assurent donc pas la compatibilité avec tous types de systèmes NoSQL. Par ailleurs, nous avons proposé un métamodèle OCL à partir de la documentation de l'OMG ; ce métamodèle constitue le point de départ du processus de transformation. Les règles de transformation basées sur le métamodèle proposé ont été exprimées en langage QVT. Nous avons expérimenté notre démarche et nos modèles sur une application Big Data du domaine médical qui porte sur des programmes pluriannuels de suivi de pathologies. Comme perspectives à ce travail, nous envisageons de poursuivre l'enrichissement de notre approche de transformation de modèles conceptuels. Cet enrichissement vise à prendre en compte d'autres catégories de contraintes OCL.

Références

Abdelhédi, F., A. Ait Brahim, F. Atigui, et G. Zurfluh (2017). Mda-based approach for nosql databases modelling. In *International Conference on Big Data Analytics and Knowledge Discovery*.

Abelló, A. (2015). Big data design. In *Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP*.

Angadi, A. B., A. B. Angadi, et K. C. Gull (2013). Growth of new databases & analysis of nosql datastores. *International Journal of Advanced Research in Computer Science and Software Engineering*.

Cattell, R. (2011). Scalable sql and nosql data stores. *Acm Sigmod Record*.

Daniel, G., G. Sunyé, et J. Cabot (2016). Umltographdb : mapping conceptual schemas to graph databases. In *International Conference on Conceptual Modeling*.

Dehdouh, K., F. Bentayeb, O. Boussaid, et N. Kabachi (2015). 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)*.

Herrero, V., A. Abelló, et O. Romero (2016). Nosql design for analytical workloads : variability matters. In *International Conference on Conceptual Modeling*.

Li, C. (2010). Transforming relational database into hbase : A case study. In *2010 IEEE International Conference on Software Engineering and Service Sciences (ICSESS)*.

Li, Y., P. Gu, et C. Zhang (2014). Transforming uml class diagrams into hbase based on meta-model. In *2014 International Conference on Information Science, Electronics and Electrical Engineering (ISEEE)*.

Oldevik, J. (2006). Mofscript user guide. *Version 0.6 (MOFScript v 1.1. 11)*.

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

Summary

Big Data application developers use NoSQL systems to store massive DBs. They transform a conceptual model describing the massive DB into a NoSQL physical model. This manual task is complex because of the specificity of NoSQL models. Our purpose is to assist the developer to automatically transform models. For this, we use the MDA architecture. Starting from a conceptual model that describes the structure of data and a set of constraints, we propose a mapping rules that generate (1) a NoSQL physical model and (2) the code for checking constraints. The first point have been addressed in previous work. In this paper, we study the second point that aims to propose an automatic process for mapping constraints.