Traitement des règles OCL

1) Introduction

Après une première réalisation d'un générateur d'applications à partir de diagrammes UML, avec comme environnement de déploiement Versata ; il s'est avéré que le traitement des règles OCL, et particulièrement la représentation sémantique adoptée, interdisait des développements indispensables à l'obtention de règles vraiment respectées lors de l'exécution des applications.
Pour générer du code traduisant fidèlement les contraintes induites par les règles, il est impératif d'être capable de dériver chacune d'elles, du point de vue des classes citées dans son expression. D'autre part, les règles peuvent nécessiter un ordonnancement, indispensable à leur bonne exécution. Enfin, la génération d'un code performant, implique de supprimer les éventuelles redondances dans la représentation sémantique.
C'est pourquoi, l'objectif est d'aboutir à une représentation sémantique aisément manipulable ; permettant ainsi de : dériver de l'expression initiale des règles, les arbres sémantiques nécessaires à leurs vérifications ; projeter ses représentations du modèle logique sur le modèle d'implémentation ; ordonnancer puis factoriser les arbres obtenus ; et enfin de générer : le code (respectivement la documentation) pour l'environnement de déploiement (respectivement des spécifications).
Dans un premier temps, nous décrirons les étapes aboutissant à la construction de l'arbre sémantique d'une règle et à sa cohérence par rapport au modèle qu'elle contraint ; nous nous attacherons ensuite à expliciter les mécanismes mis en jeu pour construire la contrainte attachée à chaque classe du modèle, pour terminer avec la génération de code.

2) Contexte

Pour faciliter l'utilisation de l'atelier de génération, nous avons été amenés à modifier, voire enrichir, le langage OCL. L'ensemble des types standards n'inclut aucune notion d'instant ni de durée. C'est pourquoi nous avons introduit quatre types supplémentaires : DateTime, Date, Time et Duration.

- DateTime représente un instant, point sur l'axe du temps, avec une précision de la milliseconde ;
- Date représente un instant, point sur l'axe du temps, avec la précision d'un jour. Ce qui implique, que toute opération attachée à ce type comporte un paramètre, indiquant quel est le fuseau horaire à prendre en considération pour la réaliser ;
- Time représente un instant de la journée, point sur l'axe du temps, avec une précision de la milliseconde ;
- Duration représente une durée, c'est un nombre entier de millisecondes.

Des opérateurs ont également été adjoints aux collections :

- isUniqueOnSet retourne vrai si il n'existe pas deux éléments de la collection instance pour lesquelles les évaluations du nuplet de ses paramètres seraient égales;
- maxBy est un opérateur sur les collections, il équivaut à l'enchaînement de l'opérateurs sortedBy suivi de l'opérateur last ;
- minBy est un opérateur sur les collections, il équivaut à l'enchaînement de l'opérateurs sortedBy suivi de l'opérateur first ;

- max (respectivement min) retourne le plus grand (respectivement petit) élément d'une collections dont les éléments sont d'un type totalement ordonnée, généralement : décimal, entier, réel, chaîne de caractères, durée.

Enfin, une classe ocl::System accueille les méthodes statiques :

- username retourne la chaîne de caractères contenant le nom de l'utilisateur connecté ;
- isAUser retourne vrai si la chaîne de caractères passée en paramètre correspond à un nom d'utilisateur ;
- date retourne la date courante dans le fuseau horaire spécifié ;
- time retourne l'heure courante dans le fuseau horaire spécifié ;
- dateTime retourne l'instant courant.

3) Analyse des règles

Cette phase débute par l'enchaînement classique des étapes de l'analyse des langages, qui à partir d'un texte construit un arbre qui en représente le sens. La cohérence entre l'arbre sémantique ainsi obtenu et le modèle logique qu'elle est sensée contraindre, est ensuite vérifiée.

a) Constitution de l'expression OCL

Dans les diagrammes UML, les règles ont quatre stéréotypes possibles : check, state, formula, « default value ». Elles sont rédigées avec un sous ensemble de l'OCL, le langage de définition de contrainte d'UML, afin d'exprimer : des assertions (respectivement des calculs), pour les règles de stéréotypes check et state (respectivement formula et « default value »). Le texte de la règle, écrit par l'utilisateur, est complété par un entête adéquat afin d'obtenir une expression OCL.

b) Analyse lexicale

La chaîne de caractère de l'expression OCL est transformée en un flux de lexèmes dans lequel, symboles, mot-clés, constantes et identificateur sont étiquetés.

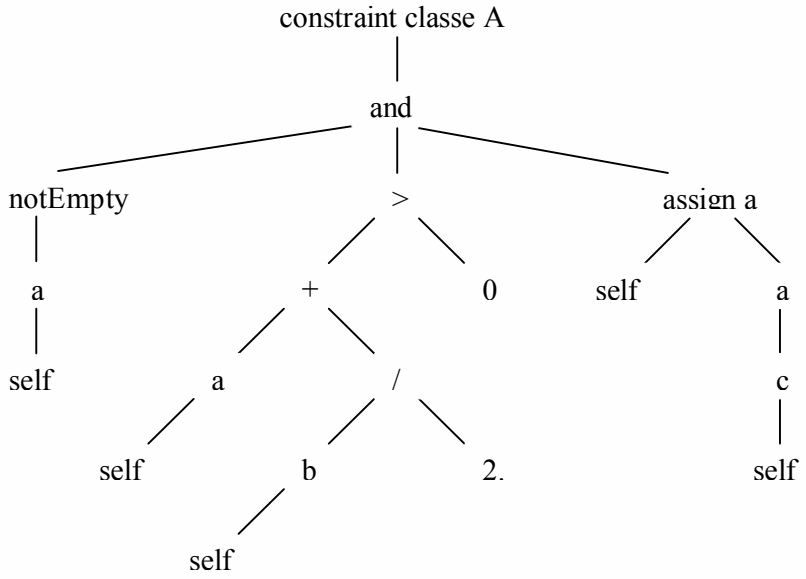
c) Analyse syntaxique

Cette étape construit un arbre syntaxique représentant l'expression OCL analysée. Chaque nœud de l'arbre représente une règle de la grammaire OCL satisfaite par l'expression analysée.

d) Construction de l'arbre sémantique

Cette étape construit l'arbre sémantique correspondant à partir de l'arbre syntaxique. Chacun de ses nœuds représente : une opération, une constante, une variable ou une instance. Chaque nœud opération comporte un certain nombre de nœuds fils qui correspondent à ses opérandes. Cet arbre est beaucoup plus compact que l'arbre syntaxique. En effet, tous les nœuds de l'arbre syntaxique correspondant à des règles de la grammaire OCL uniquement traversées lors de l'analyse de l'expression, n'ont pas d'équivalent dans l'arbre sémantique, car ils ne portent aucune signification.
La figure suivante donne un exemple d'arbre sémantique issue de la traduction de règles OCL attachées à une classe A.

contrainte de la classe A :
check : ((self.a + (self.b / 2)) > 0) and self? notEmpty()
valeur par défaut de l'attribut a : self.c.a



e) *Résolution des noms et vérification des types par rapport au modèle logique*

Il s'agit de parcourir récursivement l'arbre sémantique et de remplacer les éléments sémantiques « usage d'un attribut » et « instance» non définis, par les éléments adéquats : la variable ou la chaîne d'« usage d'un attribut » du modèle logique. Simultanément, chaque élément de l'arbre sémantique correspondant à une opération du méta-modèle des types standards de l'OCL est examiné ; les concordances des types et des cardinalités des nœuds opérandes avec ceux attendus par l'opérateur sont vérifiés.

4) Construction des contraintes globales

Cette phase consiste à traduire les contraintes attachées au modèle logique, en contraintes attachées au modèle d'implémentation. Il s'agit de déduire, pour chaque classe d'implémentation, une contrainte, dite globale, qui synthétisera l'ensemble des règles implicitement ou directement induites par les contraintes exprimées sur le modèle logique.

a) *Projection du modèle logique sur le modèle d'implémentation*

Les règles sont exprimées par l'utilisateur en se référençant au modèle logique. Le processus de génération comporte une étape dite de « dénormalisation », au cours de laquelle un modèle d'implémentation est construit. Dans celui-ci, les généralisations, les relations de composition optionnelles ou obligatoires et de cardinalité maximale un, sont dénormalisées selon les vœux de l'utilisateur. Cette étape génère des règles

qui, en s'ajoutant à la contrainte globale, viennent compenser la fusion des classes du modèle logique.
Les règles générées dépendent du caractère obligatoire ou facultatif de la dénormalisation. Les relations d'héritage et les relations de compositions de cardinalité minimale égale à zéro (respectivement les relations de composition de cardinalité minimale égale à un) correspondent à des dénormalisations facultatives (respectivement obligatoires).
Soient n classes $\mathbf{A_i}$, i? [0..n] tel que la classe $\mathbf{A_{i+1}}$ est dénormalisée dans la classe $\mathbf{A_i}$ par la dénormalisation $\mathbf{d_i}$;
$\mathbf{d_i}$, la dénormalisation, elle possède un attribut $\mathbf{attr_i}$ si elle correspond à une relation de composition ;
$\mathbf{as_{ij}}$, les attributs scalaires obligatoires de la classe $\mathbf{A_i}$;
$\mathbf{ar_{ik}}$, les attributs obligatoires associés à un rôle de la classe $\mathbf{A_i}$;

i) Projection des contraintes implicites du modèle logique

Afin de rétablir les contraintes implicites au modèle logique dans le modèle d'implémentation, on ajoute à la contrainte globale de la classe $\mathbf{A_0}$, les contraintes suivantes :
Soit $\mathbf{C^p_i}$, $\mathbf{C^{p'}_i}$, des contraintes partielles ;
Soit $\mathbf{C_i}$, la contrainte issue de la dénormalisation de di ;
Si $\mathbf{d_i}$ est obligatoire :

Si $\mathbf{A_i}$ a au moins une classe ancêtre :

$\mathbf{C^p_i}$? (and $_i$ $\mathbf{S_{A0}}$(.$\mathbf{attr_i}$).$\mathbf{as_{ij}}$->size() > 0) (and $_k$ $\mathbf{S_{A0}}$(.$\mathbf{attr_i}$).$\mathbf{ar_{ik}}$->size() > 0) and $\mathbf{S_{A1}}$.oclIsKindOf($\mathbf{S_{Ai}}$)

Sinon

$\mathbf{C^p_i}$? (and $_i$ $\mathbf{S_{A1}}$(.$\mathbf{attr_i}$).$\mathbf{as_{ij}}$->size() > 0) (and $_k$ $\mathbf{S_{A1}}$(.$\mathbf{attr_i}$).$\mathbf{ar_{ik}}$->size() > 0)

Sinon

$\mathbf{C^p_i}$? $\mathbf{S_{A1}}$(.$\mathbf{attr_i}$).oclIsKindOf($\mathbf{S_{Ai}}$)

$\mathbf{C^{p'}_i}$? $\mathbf{C_{i+1}}$ and $\mathbf{C^p_i}$

Si $\mathbf{d_i}$ est facultative :

Si $\mathbf{A_i}$ a au moins une classe ancêtre :

$\mathbf{C_i}$? not $\mathbf{S_{A1}}$(.$\mathbf{attr_i}$).oclIsKindOf($\mathbf{S_{Ai}}$) or $\mathbf{C^{p'}_i}$

Sinon

$\mathbf{C_i}$? $\mathbf{S_{A1}}$(.$\mathbf{attr_i}$).isEmpty() or $\mathbf{C^{p'}_i}$

Sinon

$\mathbf{C_i}$? $\mathbf{C^{p'}_i}$

ii) Projection des règles

Lors de la projection d'une règle, l'instance et les usages d'attribut du modèle logique sont transformés en leurs homologues du modèle d'implémentation. Il est important de tenir compte du caractère facultatif de la dénormalisation, afin de compléter la règle pour qu'elle ne soit vérifiée que lorsque l'analogue de l'instance du modèle logique est présente dans le modèle d'implémentation.

b) Calcul des règles dérivées

Pour une règle donnée, chacun de ses éléments sémantiques « usage d'un attribut » ou « affectation » correspondant à une navigation vers une classe différente de la classe à laquelle est attachée la règle, donnent lieu à une règle dérivée sur cette autre classe. En effet, si dans une règle intervient une instance ou un des champs d'une instance d'une autre classe que celle à laquelle la règle est rattachée ; alors, une modification de cette instance ou d'un des champs intervenants, est susceptible de changer l'évaluation de cette règle. Afin que la règle soit constamment vérifiée, il est indispensable de l'exprimer du point de vue de cette autre classe.

i) Dérivation avec un niveau d'« usage d'un attribut »

Les « usage d'un attribut » correspondant à une association qui est dénormalisée ne sont pas pris en considération.
Soient deux classes **A** et **B** ; on notera dans la suite :
$\mathbf{S_A}$, une instance de la classe **A** ;
$\mathbf{S_B}$, une instance de la classe **B** ;
b, le nom du rôle final de l'association qui va de **A** à **B** ;
$\mathbf{R(S_A,b)}$, une règle **R** de **A** fonction de $\mathbf{S_A}$ et **b** ;
$\mathbf{valB_i}$, le $i^{éme}$ des n champs de **B** impliqués dans l'évaluation de $\mathbf{R(S_A,b)}$;
$\mathbf{b^i}$, la rôle inverse de **b** ;
$\mathbf{R'(S_B,b)}$, la règle dérivée de $\mathbf{R(S_A,b)}$ en exploitant **b** ;
Selon la nature, composition ou agrégation, et la cardinalité minimum du rôle $\mathbf{b^i}$ de la relation qu'entretiennent **A** et **B** la règle dérivée $\mathbf{R'(S_B,b)}$ sera différente.

(1) B est associée à A par une agrégation de rôle final b

Si la cardinalité minimale du rôle $\mathbf{b^i} = 0$, alors $\mathbf{R'(S_B,b)}$ devient :
(not $\mathbf{S_B}$.oclIsNew() (and (or $\mathbf{S_B.valB_i <> S_B.valB_i@pre}$)$_{i\ ?\ [1..n]}$) and not $\mathbf{S_B.b^i}$->notEmpty()) implies $\mathbf{S_B.b^i}$->forAll($\mathbf{D_A}$ |$\mathbf{R(D_A,b)}$))

Si la cardinalité minimale du rôle $\mathbf{b^i} > 0$, alors $\mathbf{R'(S_B,b)}$ devient :
not $\mathbf{S_B}$.oclIsNew() (and (or $\mathbf{S_B.valB_i <> S_B.valB_i@pre}$)$_{i\ ?\ [1..n])}$ implies $\mathbf{S_B.b^i}$->forAll($\mathbf{D_A}$ |$\mathbf{R(D_A,b)}$))

(2) B est associée à A par une composition de rôle final b

La cardinalité minimale et maximale du rôle $\mathbf{b^i}$ est 1, alors $\mathbf{R'(S_B,b)}$ devient :
not $\mathbf{S_B}$.oclIsNew() (and (or $\mathbf{S_B.valB_i <> S_B.valB_i@pre}$)$_{i\ ?\ [1..n]}$) implies $\mathbf{R(S_B.b^i,S_B)}$

ii) Dérivation avec n niveaux d'« usage d'un attribut »

Soient une classes $\mathbf{A_0}$ et n classe $\mathbf{A_i}$, i? [1..n] ; on notera dans la suite :

$\mathbf{S_{A0}}$, une instance de la classe $\mathbf{A_0}$;
$\mathbf{S_{Ai}}$, une instance de la classe $\mathbf{A_i}$, i? [1..n] ;
$\mathbf{a_i}$, le nom du rôle final de l'association qui va de $\mathbf{A_{i-1}}$ à $\mathbf{A_i}$;
$\mathbf{R(S_A,b)}$, une règle **R** de **A** fonction de $\mathbf{S_A}$ et **b** ;

$\mathbf{valA_{nj}}$, le $j^{éme}$ des n champs de **A** impliqués dans l'évaluation de $\mathbf{R(S_A, a_{i,i\ ?\ [1..n]})}$;
$\mathbf{a_i^i}$, la rôle inverse de $\mathbf{a_i}$;
$\mathbf{R'(S_{An}, a_{i,i\ ?\ [1..n]})}$, la règle dérivée de $\mathbf{R(S_A, a_{i,i\ ?\ [1..n]})}$ en exploitant les $\mathbf{a_{i,\ i\ ?\ [1..n]}}$;

(1) A_n est associée à A via au moins une association à cardinalité multiple

$\mathbf{R'(S_{An}, a_{i,i\ ?\ [1..n]})}$ devient :
(not $\mathbf{S_{An}}$.oclIsNew() (and (or $_{j\ ?\ [1..n]}$ $\mathbf{S_{An}.valA_{nj}} <> \mathbf{S_{An}.valA_{nj}@pre}$)) (and $_k$ not $\mathbf{S_{An}}$.$(.\mathbf{a_i^i})_{i\ ?\ [1..k]}$ ->notEmpty())$_k$) implies $\mathbf{S_{An}}(.\mathbf{a_i^i})_{i\ ?\ [1..n]}$->forAll($\mathbf{D_{A0}}$ |$\mathbf{R(D_{A0}, a_i^i}_{,i\ ?\ [1..n]})$)
les valeurs de k sont choisies tel que la cardinalité minimum de $\mathbf{a_k^i}$ est nulle.

(2) A_n est associée à A avec une cardinalité 1

$\mathbf{R'(S_{An}, a_{i,i\ ?\ [1..n]})}$ devient :
(not $\mathbf{S_{An}}$.oclIsNew() (and (or $_{j\ ?\ [1..n]}$ $\mathbf{S_{An}.valA_{nj}} <> \mathbf{S_{An}.valA_{nj}@pre}$))$_j$) implies $\mathbf{R(S_{An}.\ (.a_i^i)_{i\ ?\ [1..n]}, S_{An})}$

iii) Dérivation d'un « usage d'attribut » de même type que la classe origine de la règle

$\mathbf{R'(S_B,b)}$ devient :
($\mathbf{S_B}$.oclIsNew() or $\mathbf{S_B.b} <> \mathbf{S_B.b@pre}$) implies $\mathbf{R(S_B.b,b)}$)

iv) Dérivation d'une « assignation »

$\mathbf{A_b(S_A, expr(S_A))}$, une affectation de l'expression $\mathbf{expr(S_A)}$ au rôle **b** de l'instance $\mathbf{S_A}$;
$\mathbf{A_b'(S_B, expr'(S_B))}$, l'affectation dérivée de $\mathbf{A_b(S_A, expr(S_A))}$ en exploitant **b**.
Si l'association **b** n'est pas dénormalisée, et que $\mathbf{expr(S_A)}$ ne dépends pas de **b** (sinon elle aurait déjà été dérivée), alors $\mathbf{A'_b(S_A, expr(S_A))}$ devient :
$\mathbf{A_b}$(if $\mathbf{S_B}$.oclIsDeleted() then Set{} else $\mathbf{expr(S_B.b^i)}$ endif)

c) Constitution d'une contrainte globale par classe

En conservant les notations précédemment évoquées la contrainte globale est :
n classe $\mathbf{A_i}$, i? [1..n] ;
$\mathbf{R_{ik}(S_{Ai},a_{ikl})}$, les règles **R** indicées par **ik** de $\mathbf{A_i}$ fonction de $\mathbf{S_{Ai}}$ et des $\mathbf{a_{ikl}}$ le sous ensemble des rôles reliant les classes $\mathbf{A_i}$ entre elles ;
$\mathbf{G_i}$, la contrainte globale de la classe $\mathbf{A_i}$.
La contrainte globale de $\mathbf{A_i}$ est la conjonction des contraintes attachées initialement à $\mathbf{A_i}$, et des contraintes attachées aux classes $\mathbf{A_j}$ qui admette une dérivation sur $\mathbf{A_i}$.
$\mathbf{G_i}$ est donc équivalente à :
and $_{ik}$ $\mathbf{R_{ik}(S_{Ai},a_{ikl})}$ (and $_{j\ ?\ i\ j\ ?\ [1..n]}$ $\mathbf{R_{jk}'(S_{Ai}, a_{jkl})}$)

5) Traitement des contraintes globales

Cette phase enchaîne les traitements assurant : qu'un sens donné est exprimé de façon univoque dans l'arbre sémantique de la contrainte globale, que les nœuds de l'arbre sémantique sont ordonnés, et enfin, que les factorisations sont effectuées.

a) Normalisation et simplification

La normalisation d'un arbre sémantique, consiste à le modifier afin de ne pas laisser subsister des sous-arbres qui ont le même sens mais qui mettent en jeu des nœuds

différents. Il s'agit d'associer à une signification donnée une unique représentation sous forme d'arbre sémantique. Ainsi, il est possible de :

- Détecter des simplifications de l'arbre, sans multiplier les motifs de recherche ;
- Faciliter les recherches de sous-arbres ayant un sens donné, avec un motif unique.

Le tableau qui suit récapitule l'ensemble des motifs donnant lieu à une transformation de l'arbre. Les notations adoptées sont les suivantes :

pi, la ième proposition,
ci, la ième constante,
c, une constante,
p, q, deux expressions,
col, une collection d'instances.

Opérateur	Motif identifié	Transformation effectuée
Add	? $_{j? [1,m]}$ p_j + ? $_{i? [1,n]}$ c_i	? $_{j? [1,m]}$ p_j +c, c (= ? $_{I? [1,n]}$ c_i)
And	and ($p_1 \ldots p_{j-1}$ (and $q_1 \ldots q_m$) $p_{j+1} \ldots p_n$)	and ($p_1 \ldots p_{j-1} q_1 \ldots q_m p_{j+1} \ldots p_n$)
	and ($p_1 \ldots p_j \ldots p_k \ldots p_n$), p_j = ? p_k	false
	and ($p_1 \ldots pj \ldots p_k \ldots p_n$)), p_j = p_k	and ($p_1 \ldots p_{k-1} p_{k+1} \ldots p_n$)
	and(=(a,b) =(a,b,c))	=(a,b,c)
	and ($p_1 \ldots p_{j-1}$ p_j $p_{j+1} \ldots p_n$), p_j : true	and ($p_1 \ldots p_{j-1}$ $p_{j+1} \ldots p_n$)
	and ($p_1 \ldots p_{j-1}$ p_j $p_{j+1} \ldots p_n$), p_j : false	false
	and ($p_1 \ldots p_j \ldots p_k \ldots p_n$), p_j : a>c_1, p_k a>c_2	and ($p_1 \ldots p_{j-1}$ a>max(c_1, c_2) $p_{j+1} \ldots p_{k-1}$ $p_{k+1} \ldots p_n$)
	and ($p_1 \ldots p_j \ldots p_k \ldots p_n$), p_j : c_1>a, p_k c_2>a	and ($p_1 \ldots p_{j-1}$ max(c_1, c_2)>a $p_{j+1} \ldots p_{k-1}$ $p_{k+1} \ldots p_n$)
	and ($p_1 \ldots p_j \ldots p_k \ldots p_n$), p_j : a>b, p_k :b>a	false
	and ($p_1 \ldots p_j \ldots p_n$), p_j : or ($q_1 \ldots q_{k-1} p_i q_{k+1} \ldots q_m$)	and ($p_1 \ldots p_j \ldots p_{j-1}$ $p_{j+1} \ldots p_n$)
	and ($p_1 \ldots p_i \ldots p_j \ldots p_n$), pi : or ($q_l \ldots q_k$) p_j : or ($q_1 \ldots q_l \ldots q_k \ldots q_m$)	and ($p_1 \ldots p_j \ldots p_{j-1}$ $p_{j+1} \ldots p_n$)
	and ($p_1 \ldots p_i \ldots p_j \ldots p_n$), p_i : a > b, p_j : not b > a	and ($p_1 \ldots p_i \ldots p_{j-1} p_{j+1} \ldots p_n$),
	and ($p_1 \ldots p_i \ldots p_n$), p_i : or ($q_1 \ldots q_m$)	or (and (q_1 $p_1 \ldots p_i \ldots p_n$)… and ($q_m$ $p_1 \ldots p_i \ldots p_n$)
	and (p_1)	p_1
Equal	equal ($p_1 \ldots p_i \ldots p_j \ldots p_n$), p_j = p_i	equal ($p_1 \ldots p_i \ldots p_{j-1}$ $p_{j+1} \ldots p_n$)
	equal ($p_1 \ldots c_1 \ldots c_2 \ldots p_n$)	false
	equal (p_1)	true
	equal ($p_1 \ldots p_i \ldots p_n$), p_i = false	and (not $p_1 \ldots$not p_{i-1} not $p_{i+1} \ldots$not p_n)
	equal ($p_1 \ldots p_i \ldots p_n$), p_i = true	and ($p_1 \ldots p_{i-1}$ $p_{i+1} \ldots p_n$)
	equal ($p_1 \ldots p_n$), p_i est du type booléen	or (and($p_1 \ldots p_n$) and(not $p_1 \ldots$not p_n))
	equal ($p_1 \ldots p_n$), p_i est du type réel	and (not p_1>p_2 not p_2>p_1 …not p_1>p_n not p_n>p_1)
Multiply	? $_{j? [1,m]}$$p_j$. ? $_{i? [1,n]}$$c_i$	? $_{j? [1,m]}$$p_j$ .c, c = ? $_{I? [1,n]}$$c_i$

Or	or ($p_1 \ldots p_{j-1}$ (or $q_1 \ldots q_m$) $p_{j+1} \ldots p_n$)	or ($p_1 \ldots p_{j-1} q_1 \ldots q_m p_{j+1} \ldots p_n$)
	or ($p_1 \ldots p_{j-1}$ p_j $p_{j+1} \ldots p_n$), p_j : true	true
	or ($p_1 \ldots p_{j-1}$ p_j $p_{j+1} \ldots p_n$), p_j : false	or ($p_1 \ldots p_{j-1}$ $p_{j+1} \ldots p_n$)
	or ($p_1 \ldots p_j \ldots p_k \ldots p_n$), p_j = ? p_k	true
	or ($p_1 \ldots pj \ldots p_k \ldots p_n$)), p_j = p_k	or ($p_1 \ldots p_{k-1} p_{k+1} \ldots p_n$)
	or (=(a,b) =(a,b,c))	=(a,b)
	or ($p_1 \ldots p_j \ldots p_k \ldots p_n$), p_j : a>c_1, p_k a>c_2	or ($p_1 \ldots p_{j-1}$ a>min(c_1, c_2) $p_{j+1} \ldots p_{k-1}$ $p_{k+1} \ldots p_n$)
	or ($p_1 \ldots p_j \ldots p_k \ldots p_n$), p_j : c_1>a, p_k c_2>a	or ($p_1 \ldots p_{j-1}$ min(c_1, c_2)>a $p_{j+1} \ldots p_{k-1}$ $p_{k+1} \ldots p_n$)
	or ($p_1 \ldots p_j \ldots p_k \ldots p_n$), p_j : a>b, p_k : not b>a	or ($p_1 \ldots p_j \ldots p_{k-1} p_{k+1} \ldots p_n$),
	or ($p_1 \ldots p_j \ldots p_n$), p_j : and ($q_1 \ldots q_{k-1} p_i q_{k+1} \ldots q_m$)	or ($p_1 \ldots p_j \ldots p_{j-1}$ $p_{j+1} \ldots p_n$)
	or ($p_1 \ldots p_i \ldots p_j \ldots p_n$), pi : and ($q_l \ldots q_k$) p_j : and ($q_1 \ldots q_l \ldots q_k \ldots q_m$)	or ($p_1 \ldots p_j \ldots p_{j-1}$ $p_{j+1} \ldots p_n$)
	or (p_1)	p_1
Xor	xor (p_1, p_2)	and (or (p_1, p_2), or (not p_1, not p_2)
Negate	-? $_{j? [1,m]}$ p_j	? $_{j? [1,m]}$ -p_j
	-c, c = val	c, c = -val
	--p	p
Not	not true (resp. not false)	false (resp. true)
	not not p	p
	not and($p_1 \ldots p_n$)	or (not $p_1 \ldots$not p_n)
	not or($p_1 \ldots p_n$)	and (not $p_1 \ldots$not p_n)
Divide	p/c, c = 1	p
	c_1/c_2	c = c_1/c_2
	c/p, c = 0	c, c = 0
Greater	$c_1 > c_2$, $c_1 > c_2$ (resp. $c_1 \leq c_2$)	true (resp. false)
	p > q, p est similaire à q	false
	c > col? size(), c ≤ 0	false
GreaterOrEqual	p ? q	not (q > p)
Implies	p ? q	Or (not p, q)
Less	p < q	q > p
LessOrEqual	p ≤ q	not (p > q)
Minus	p – q	p + (-q)
NotEqual	p ? q	not (p = q)
Let	let p = expr, p n'est pas utilisé	
GetOclType	p.getOclType(), p à pour type t	t
IsTypeOf	p.isTypeOf(t), t est abstrait	false
IsKindOf	p.isKindOf(t), le type de p est conforme à t	true
	p.isKindOf(t), t n'est pas conforme au type de p	false
	p.isKindOf(t), t n'a pas de type concret	false

	p.isKindOf(t), t a un seul type concret t_1	p.isTypeOf(t_1)
	p.isKindOf(t), t a n types concret t_1...t_n	or (p.isTypeOf(t_1)… p.isTypeOf(t_2))
AsType	p.asType(t), le type de p est conforme à t	p
Abs	a.abs()	if 0 > a then –a else a endif
Exists	col? exists(false)	false
	col? exists(true)	col? size() > 0
	col? exists(expr)	col? select(expr) ? size()> 0
ForAll	col? exists(c), c = true, false	c
Reject	col? reject(expr)	col? select(not expr)
AsBag	col? asBag()	col
AsSequence	col? asSequence(), col est une collection ordonnée	col
AsSet	col? asSet(), col est un ensemble	col
First	col? first()	col? at(1)
IsEmpty	col? isEmpty(), col est une collection à au moins un élément	false
	col? isEmpty()	not (col? size() > 0)
Last	col? last()	col? at(col? size())
NotEmpty	col? notEmpty(), col est une collection à au moins un élément	true
	col? notEmpty()	col? size() > 0
Size	p? size(), la cardinalité minimale et maximale de p sont égale à c	c
Symmetric Difference	col_1? SymmetricDifference(col_2)	col_1? union(col_2) ? difference(col1? intersection(col2))
If	if c then p_1 else p_2 endif, c = true	p_1
	if c then p_1 else p_2 endif, c = false	p_2
	if c then p_1 else p_2 endif, p_1 est similaire à p_2	p_1
	if c then p_1 else p_2 endif, p_1 et p_2 sont de type booléen	or (and (c, p_1), and (not c, p_2))

b) Ordonnancement

Cette étape du traitement consiste à ordonner partiellement les nœuds de l'arbre sémantique, afin que les contraintes suivantes soient respectées :
Les éléments sémantiques AttributeUsage, MaxByOperator, MinByOperator, FirstOperator, LastOperator, MaxOperator, MinOperator, SumOperator, s'ils sont liés à une instance dont la cardinalité minimale est nulle, doivent être précédés dans l'arbre sémantique d'un test d'existence de cette instance.
Des traitements similaires sont appliqués à l'élément sémantique AttributeUsage :

- si l'attribut est calculé par une formule ou une valeur par défaut, l'élément sémantique doit être précédé dans l'arbre sémantique de l'affectation de sa

valeur. Cependant, cette vérification n'est à opérer que si l'usage de l'attribut intervient directement sur l'instance dans laquelle est réalisée l'affectation.

- Si l'usage de l'attribut fait référence à la valeur qu'il avait avant la transaction courante ; il doit être précédé dans l'arbre sémantique d'un test assurant que l'instance portant l'attribut n'a pas été insérée durant la transaction courante.

Les opérandes de chaque nœud de l'arbre correspondant à un opérateur booléen, ancêtre à la fois de l'élément sémantique répondant aux critères précédemment évoqués et de la pré-condition associée, sont ordonnés par un algorithme de tri topologique. Si la pré-condition n'est pas trouvée dans l'arbre sémantique, une erreur est remontée à l'utilisateur.

c) Factorisation

Il s'agit de transformer l'arbre sémantique, en éliminant les redondances et en réintroduisant l'élément sémantique le plus approprié, quand un motif d'éléments sémantiques synonyme est repéré. Cette opération est le pendant du processus de normalisation qui consistait à restreindre la variété des éléments sémantiques pour diminuer le spectre des motifs à rechercher.

6) Transformation pour la cible

Cette dernière phase est dépendante de l'objectif du processus de génération et de l'éventuel environnement de déploiement choisi. Elle consiste à transformer l'arbre sémantique en un arbre formé de nœuds adaptés à la génération de code ou d'information pour une cible donnée.
L'objectif de cette étape est de transformer chaque nœud de l'arbre sémantique, en un nœud capable de générer du code pour une cible donnée ou de la documentation de spécification. Chaque élément sémantique délègue sa transformation à une classe implémentant une interface qui énumère toutes les transformations nécessitées par la variété des éléments sémantiques. Il est donc aisé de générer le code pour une cible différente, puisqu'il suffit de pourvoir le générateur d'une implémentation adaptée à celle-ci, sans modifier en quoi que se soit le reste du processus de traitement des règles.

a) Documentation de spécifications

b) Versata

La génération du code java pour Versata s'appuis sur l'interface de la classe DataObject.

i) Scission des contraintes globales

Les opérateurs sémantiques tels que : allInstances, AttributUsage d'un attribut à cardinalité multiple, et ceterae, génèrent du code qui ne peut s'exécuter de manière satisfaisante que dans un contexte où, toutes les instances ou modifications des instances sont connues. Or, dans Versata, les règles sont déclenchées au sein des opérations d'insertion, de mise à jour et de destruction des instances, à des instants qui ne garantissent pas, pour les opérateurs sus-cités l'accès à toutes les instances implicitement référencées. Ce qui signifie que l'exécution de règles impliquant de tels opérateurs doit être déclenchée juste avant que les modification des instances deviennent permanentes. Dans un souci d'optimisation, il est important de scinder les

contraintes globales en deux parties : l'une exécutée au plus près des modifications de l'instance, indépendante de collection d'instance ; l'autre exécutée lorsque toutes les informations sont prêtes à être stockées. Ceci permet d'interrompre la transaction à la première erreur, au plus près de l'instance fautive.

ii) Java

Phase d'optimisation liée aux opérateurs : x, y, z.
Remonté des erreurs

iii) SQL

Certain sous arbre-sémantiques peuvent être transformés en code SQL. Par exemple :

- les sous-arbres du type self.oclType().allInstances->isUniqueOnSet(x,y) deviennent un contraint IS UNIQUE sur les champs x, y.
- dans les sous-arbres du type self.oclType().allInstances->select(D | D.x = 0), D.x = 0 devient la partie conditionnelle de la requête clause.

7) Améliorations possibles

Il serait souhaitable de modifier la représentation sémantique, afin d'éviter la duplication intempestive de sous-arbres. Ceci aurait plusieurs avantages : économiser l'espace mémoire utilisé par les représentations sémantiques ; diminuer le coût de comparaison de sous-arbres entre eux, puisque les relations d'identité seraient conservées, au lieu d'être recalculées en cas de nécessité ; préparer la phase de factorisation, car l'utilisation multiple d'un même sous-arbre en fait un candidat privilégié pour une mise en facteur.

8) Conclusion

Les objectifs ont ils été atteints ?
Le problème est-il entièrement ou partiellement résolu ?
Les résultats conduisent-ils à de nouvelles questions ?
Comment ce travail a t'il fait évoluer l'état de l'art ?
Dans quel domaine et dans quelle mesure les résultats peuvent infléchir ou modifier la situation actuelle ?
Que nous a apporté ce travail que nous n'attendions pas ?

Annexe A : Nomenclature des méthodes traitant les paramètres temporels

Les valeurs possibles des chaînes de caractères 'timeZoneStr' et 'localeStr' sont listées en annexe, à celles-ci s'ajoutent pour la détermination du fuseau horaire comme de la localisation trois valeur <user>, <company> et <system>.

- <user> et <company> sont des raccourcis pour accéder aux propriétés de nom TIME_ZONE (fuseau horaire), LOCALE_LANGUAGE, LOCALE_COUNTRY et LOCALE_VARIANT (localisation) positionnées pour chaque utilisateur au moyen de la console versata, ou globalement pour le Business Logic Server.
- <system> est un raccourci pour accéder au valeur par défaut de la machine sur laquelle s'exécute le Business Logic Server.

Si le paramètres timeZoneStr' ou 'localeStr' sont invalides, les valeurs par défaut sont alors recherchées dans l'ordre suivant spécifique, <user>, <company> et enfin <system> qui lui aboutit dans tout les cas.

Method name	*Return type*	*Parameters*	*Description*
		Ocl::System	
date	Date	String : timeZoneStr	Retourne la date exprimé dans le fuseau horaire défini par la chaîne de caractères
time	Time		Retourne l'heure GMT
dateTime	DateTime		Retourne un object DateTime représentant le nombre de millisecondes qui sépare le 1 janvier 1970 à 0 heure 0 minute 0 seconde et l'instant présent, le tout exprimé en temps GMT
userName	String		Retourne le nom de l'utilsateur qui manipule les données
isAUser	Boolean	String : userName	Vrai si il existe un utilisateur ayant pour nom la chaîne de caractères
		DateTime	
=	Boolean	DateTime: dateTime2	Vrai si les deux instants représentés par les deux objets DateTime sont égaux
between	Boolean	DateTime: dateTime2, DateTime : dateTime2	Vrai si l'instant représenté par l'objet DateTime est compris entre les deux instants représentés par les deux objets DateTime passés en paramètre
<	Boolean	DateTime: dateTime2	Vrai si l'instant représentés par le premier objet DateTime est strictement inférieur à celui représentés par l'object DateTime passé en paramètre
>	Boolean	DateTime: dateTime2	Vrai si l'instant représentés par le premier objet DateTime est strictement supérieur à celui représentés par l'objet DateTime passé en paramètre
<=	Boolean	DateTime: dateTime2	Vrai si l'instant représentés par le premier objet DateTime est inférieur à celui représentés par l'objet DateTime passé en paramètre
>=	Boolean	DateTime: dateTime2	Vrai si l'instant représentés par le premier object DateTime est supérieur à celui représentés par l'object DateTime passé en paramètre
min	DateTime	DateTime: dateTime2	Retourne l'objet DateTime dont l'instant est le plus petit
max	DateTime	DateTime: dateTime2	Retourne l'objet DateTime dont l'instant est le plus grand
-	Duration	DateTime: dateTime2	Calcul la durée qui sépare les deux objets DateTime, le résultat est algébrique
-	DateTime	Duration: duration	Retranche la durée passée en paramètre à l'objet DateTime
+	DateTime	Duration: duration	Ajoute la durée passée en paramètre à l'objet DateTime
getMillis	Integer	String : timeZoneStr	Retourne le nombre de millisecondes de l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
getSeconds	Integer	String : timeZoneStr	Retourne le nombre de secondes de l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
getMinutes	Integer	String : timeZoneStr	Retourne le nombre de minutes de l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
getHours	Integer	String : timeZoneStr	Retourne le nombre d'heures de l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
getDay	Integer	String : timeZoneStr	Retourne le numéro du jour dans le mois de l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères (1..31)
getMonth	Integer	String : timeZoneStr	Retourne le numéro du mois de l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères (0..11)
getLongYear	Integer	String : timeZoneStr	Retourne le nombre correspondant à l'année de l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
getDayOfYear	Integer	String : timeZoneStr	Retourne le numéro du jour dans l'année de l'objet DateTime exprimé dans le fuseau horaire défini par la chaine de caractères (1..365)
getMonthString	String	String : timeZoneStr, String : localeStr	Retourne le nom du mois de l'object DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères, et dans la langue défini par la localisation
getDayOfWeek	Integer	String : timeZoneStr	Retourne le numéro du jour dans la semaine de l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères (0..6)

getDayString	String	String : timeZoneStr, String : localeStr	Retourne le nom du jour de l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères, et dans la langue défini par la localisation
getWeekOfYear	Integer	String : timeZoneStr	Retourne le numéro de la semaine dans l'année de l'object DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères (1..53)
addMillis	DateTime	String : timeZoneStr, Integer : millis	Ajoute un nombre algébrique de millisecondes à l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
addSeconds	DateTime	String : timeZoneStr, Integer : seconds	Ajoute un nombre algébrique de secondes à l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
addMinutes	DateTime	String : timeZoneStr, Integer : minutes	Ajoute un nombre algébrique de minutes à l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
addHours	DateTime	String : timeZoneStr, Integer : hours	Ajoute un nombre algébrique de heures à l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
addDays	DateTime	String : timeZoneStr, Integer : days	Ajoute un nombre algébrique de jours à l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
addWeeks	DateTime	String : timeZoneStr, Integer : weeks	Ajoute un nombre algébrique de semaines à l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
addMonths	DateTime	String : timeZoneStr, Integer : months	Ajoute un nombre algébrique de mois à l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
addYears	DateTime	String : timeZoneStr, Integer : years	Ajoute un nombre algébrique d'années à l'objet DateTime exprimé dans le fuseau horaire défini par la chaîne de caractères
Date			
-	DateTime	Duration: duration	Retranche la durée passée en paramètre à l'objet Date
+	DateTime	Duration: duration	Ajoute la durée passée en paramètre à l'objet Date
getMillis	Integer	String : timeZoneStr	Retourne 0
getSeconds	Integer	String : timeZoneStr	Retourne 0
getMinutes	Integer	String : timeZoneStr	Retourne 0
getHours	Integer	String : timeZoneStr	Retourne 0
addMillis	DateTime	String : timeZoneStr, Integer : millis	Ajoute un nombre algébrique de millisecondes à l'objet Date exprimé dans le fuseau horaire défini par la chaîne de caractères
addSeconds	DateTime	String : timeZoneStr, Integer : seconds	Ajoute un nombre algébrique de secondes à l'objet Date exprimé dans le fuseau horaire défini par la chaîne de caractères
addMinutes	DateTime	String : timeZoneStr, Integer : minutes	Ajoute un nombre algébrique de minutes à l'objet Date exprimé dans le fuseau horaire défini par la chaîne de caractères
addHours	DateTime	String : timeZoneStr, Integer : hours	Ajoute un nombre algébrique d'heures à l'objet Date exprimé dans le fuseau horaire défini par la chaîne de caractères
Time			
=	Boolean	Time: time2	Vrai si les deux instants représentés par les deux objets Time sont égaux
between	Boolean	Time: Time2, Time: Time2	Vrai si l'instant représenté par l'objet Date est compris entre les deux instants représentés par les deux objets Date passés en paramètre. Attention le résultat peut être étrange si les dates ont été obtenu pour des fuseaux horaires différents).
<	Boolean	Time: time2	Vrai si l'instant représentés par le premier objet Time est strictement inférieur à celui représentés par l'object Time passé en paramètre
>	Boolean	Time: time2	Vrai si l'instant représentés par le premier objet Time est strictement supérieur à celui représentés par l'objet Time passé en paramètre
<=	Boolean	Time: time2	Vrai si l'instant représentés par le premier objet Time est inférieur à celui représentés par l'objet Time passé en paramètre
>=	Boolean	Time: time2	Vrai si l'instant représentés par le premier object Time est supérieur à celui représentés par l'object Time passé en paramètre
min	Time	Time: time2	Retourne l'objet Time dont l'instant est le plus petit
max	Time	Time: time2	Retourne l'objet Time dont l'instant est le plus grand
Duration			

=	Boolean	Duration: duration2	Vrai si les deux nombres de millisecondes représentés par les deux objets Duration sont égaux
<	Boolean	Duration: duration2	Vrai si le nombrede millisecondes représentés par le premier objet Duration est strictement inférieur à celui représentés par l'object Duration passé en paramètre
>	Boolean	Duration: duration2	Vrai si le nombrede millisecondes représentés par le premier objet Duration est strictement supérieur à celui représentés par l'objet Duration passé en paramètre
<=	Boolean	Duration: duration2	Vrai si le nombrede millisecondes représentés par le premier objet Time est inférieur à celui représentés par l'objet Duration passé en paramètre
>=	Boolean	Duration: duration2	Vrai si le nombrede millisecondes représentés par le premier object Duration est supérieur à celui représentés par l'object Duration passé en paramètre
min	Duration	Duration: duration2	Retourne l'objet Duration dont le nombre de millisecondes est le plus petit
max	Duration	Duration: duration2	Retourne l'objet Duration dont le nombre de millisecondes est le plus grand
+	Duration	Duration: duration2	Ajoute le nombre de millisecondes de l'objet Duration passé en paramètre
-	Duration	Duration: duration2	Retranche le nombre de millisecondes de l'objet Duration passé en paramètre
addMillis	Duration	Integer : millis	Ajoute un nombre algébrique de millisecondes à l'objet Duration
addSeconds	Duration	Integer : seconds	Ajoute un nombre algébrique de secondes à l'objet Duration
addMinutes	Duration	Integer : minutes	Ajoute un nombre algébrique de minutes à l'objet Duration
addHours	Duration	Integer : hours	Ajoute un nombre algébrique de heures à l'objet Duration
addDays	Duration	Integer : days	Ajoute un nombre algébrique de jours à l'objet Duration
addWeeks	Duration	Integer : weeks	Ajoute un nombre algébrique de semaines à l'objet Duration
addMonths	Duration	Integer : months	Ajoute un nombre algébrique de mois à l'objet Duration
asMillis	Integer		Retourne le nombre de millisecondes représenté par cet objet Duration
asSeconds	Real		Retourne le nombre de secondes représenté par cet objet Duration, la partie décimale représente une fraction de seconde
asMinutes	Real		Retourne le nombre de minutes représenté par cet objet Duration, la partie décimale représente une fraction de minute
asHours	Real		Retourne le nombre de heures représenté par cet objet Duration, la partie décimale représente une fraction d'heure
asDays	Real		Retourne le nombre de jours représenté par cet objet Duration, la partie décimale représente une fraction de jour
asWeeks	Real		Retourne le nombre de semaines représenté par cet objet Duration, la partie décimale représente un fraction de semaine
asMonths	Real		Retourne le nombre de mois de 31 jours représenté par cet objet Duration, la partie décimale représente une fraction de mois

Annexe B : Eléments sémantiques

a) Opérateurs commutatifs et associatifs

Opérateurs commutatifs et associatifs	
Nom	Description
AddOperator	Adition de n opérandes de type : réel, décimal, entier, durée
AndOperator	Et logique entre n opérandes booléens
EqualOperator	Test de l'égalité entre chacune des n opérandes, deux à deux
MultiplyOperator	Multiplication de n opérandes de type : réel, décimal, entier, durée
OrOperator	Ou logique entre n opérandes booléens

XorOperator	Ou exclusif entre n opérandes booléens

Opérateur unaire	
Nom	Description
NegateOperator	Opposé d'un opérande de type : réel, décimal ou entier
NotOperator	Négation d'un opérande booléen

b) Opérateurs binaires

Opérateurs binaires	
Nom	Description
DivideOperator	Division entre deux opérandes de type : réel, décimal, entier, durée
GreaterOperator	Test si le premier opérande est supérieur au second, pour les types : réel, décimal, entier, date heure, durée, date, heure, chaîne de caractères
GreaterOrEqualOperator	Test si le premier opérande est supérieur ou égal au second, pour les types : réel, décimal, entier, durée, date heure, date, heure, chaîne de caractères
ImpliesOperator	Implication logique : p? q ? ? p? q
LessOperator	Test si le premier opérande est inférieur au second, pour les types : réel, décimal, entier, durée, date heure, date, heure, chaîne de caractères
LessOrEqualOperator	Test si le premier opérande est inférieur ou égal au second, pour les types : réel, décimal, entier, durée, dateheure, date, heure, chaîne de caractères
MinusOperator	Soustraction du second opérande au premier, pour les types : réel, décimal, entier, durée
MinusDateOperator	Soustraction du second opérande au premier, pour les types : date heure, date, heure
NotEqualOperator	Test d'inégalité entre deux opérandes, pour tous les types
RangeCreator	Création d'un intervalle de valeurs entières
LetOperator	Définition d'une variable équivalente à une expression

c) Opérations d'instance

Opérations d'instance	
Nom	Description
GetOclType	Retourne le type de l'instance
IsTypeOf	Test d'égalité entre le type de l'instance et le type donné en paramètre ?
IsKindOf	Test d'égalité ou de relation d'héritage entre le type de l'instance et le type donné en paramètre ?
AsType	Conversion de l'instance en un type plus spécifique
IsDeleted	L'instance est-t-elle détruite ?
IsNew	L'instance est-t-elle nouvelle ?
IsUpdated	L'instance est-t-elle modifiée ?
AbsOperation	Calcul de la valeur absolue de l'instance, valable pour les types : réel, décimal, entier ou durée
DivOperation	Division euclidienne de l'instance par le paramètre, valable pour le type entier

FloorOperation	Calcul du plus grand entier inférieur à l'instance de type : réel, décimal, entier ou durée
MaxOperation	Maximum de l'instance et du paramètre pour les types : réel, décimal, entier ou durée
MinOperation	Minimum de l'instance et du paramètre pour les types : réel, décimal, entier ou durée
ModOperation	Reste de la division euclidienne de l'instance par le paramètre, valable pour le type entier
RoundOperation	Calcul l'entier le plus proche de l'instance de type : réel, décimal, entier ou durée
ConcatOperation	Concaténation de l'instance avec une chaîne de caractères
SubStringOperation	Extraction d'une sous chaîne de caractères de l'instance
ToDecimaOperation	Conversion de l'instance de type chaîne de caractères en un décimal
ToIntegerOperation	Conversion de l'instance de type chaîne de caractères en un entier
ToRealOperation	Conversion de l'instance de type chaîne de caractères en un réel
ToLowerOperation	Conversion de l'instance en une chaîne de caractères en minuscules
ToUpperOperation	Conversion de l'instance en une chaîne de caractères en majuscules
ToStringOperation	Conversion de l'instance en une chaîne de caractères
MethodCall	Appel d'une méthode de l'instance

d) Opérateurs d'instance

Opérateurs d'instance	
Nom	Description
LetOperator	Définition d'une variable équivalente à une expression
AttributUsage	Usage d'un attribut
ReverseRole	Usage des rôles opposés de l'association, à ceux portés par l'attribut rôle de cet élément sémantique
AllInstances	Ensemble de toutes les instances d'un type donné

e) Itérateurs sur les collections

Itérateurs sur les collections	
Nom	Description
CollectOperator	Construction de la collection, résultant de l'évaluation de l'expression passée en paramètre pour chaque élément de la collection instance
ExistsOperator	Test de la vérité de l'expression passée en paramètre, pour au moins un élément de la collection instance
ForAllOperator	Test de la vérité de l'expression passée en paramètre, pour tous les éléments de la collection instance
IsUniqueOperator	Test de l'unicité de l'évaluation de l'expression passée en paramètre pour chaque élément de la collection instance

IterateOperator	Accumulation des évaluations de l'expression passée en paramètre pour chaque élément de la collection instance, et retour du résultat de cette accumulation
RejectOperator	Construction de la collection, résultant des éléments de la collection instance, pour lesquels l'expression passée en paramètre est évaluée à faux
SelectOperator	Construction de la collection, résultant des éléments de la collection instance, pour lesquels l'expression passée en paramètre est évaluée à vrai
MaxByOperator	Retourne l'élément de la collection instance pour lequel l'évaluation de l'expression passée en paramètre est maximale
MinByOperator	Retourne l'élément de la collection instance pour lequel l'évaluation de l'expression passée en paramètre est minimale
SortedByOperator	Trie la collection instance par ordre croissant de l'évaluation, pour chacun de ses éléments, de l'expression passée en paramètre
IsUniqueOnSetOperator	Test de l'unicité de l'évaluation des nuplets d'expressions passées en paramètre pour chaque élément de la collection instance

f) Opérateurs sur les collections sans paramètre

Opérateurs sur les collections sans paramètre	
Nom	Description
AsBagOperator	Transformation de la collection instance en un sac
AsSequenceOperator	Transformation de la collection instance en une séquence
AsSetOperator	Transformation de la collection instance en un ensemble
FirstOperator	Extraction du premier élément de la collection instance
IsEmptyOperator	Test de l'absence d'éléments dans la collection instance
LastOperator	Extraction du dernier élément de la collection instance
MaxOperator	Retourne le plus grand élément de la collection instance, valable pour les types : réel, décimal, entier, durée, date heur, date, heure ou chaîne de caractères
MinOperator	Retourne le plus petit élément de la collection instance, valable pour les types : réel, décimal, entier, durée, date heur, date, heure ou chaîne de caractères
NotEmptyOperator	Test de la présence d'éléments dans la collection instance
SizeOperator	Calcul de la cardinalité de la collection instance
SumOperator	Calcul de la somme des éléments de la collection instance, valable pour les types : réel, décimal, entier ou durée

g) Opérateurs sur les collections avec un paramètre

Opérateurs sur les collections avec un paramètre	
Nom	Description
AppendOperator	Ajout d'un élément en queue de la collection instance

AtOperator	Retourne l'élément de la collection instance à la position passée en paramètre
CountOperator	Compte le nombre d'élément de la collection instance égal à l'instance passée en paramètre
DifferenceOperator	Calcul la différence entre l'ensemble instance et l'ensemble passé en paramètre
ExcludesOperator	Test de la non appartenance de l'instance passée en paramètre à la collection instance
ExcludesAllOperator	Test de la non appartenance des éléments de la collection instance passée en paramètre à la collection instance
ExcludingOperator	Exclusion de l'ensemble instance de l'élément passé en paramètre
IncludesOperator	Test de l'appartenance de l'instance passée en parameter à la collection instance
IncludesAllOperator	Test de l'appartenance des éléments de la collection instance passée en paramètre à la collection instance
IncludingOperator	Inclusion dans l'ensemble instance de l'élément passé en paramètre
IntersectionOperator	Intersection de la collection instance et de celle passée en paramètre
PrependOperator	Ajout d'un élément en tête de la collection instance
SymmetricDifference Operator	Différence symétrique entre l'ensemble instance et celui passé en paramètre
UnionOperator	Union de la collection instance et de celle passée en paramètre

h) Opérateur sur les collections avec deux paramètres

Opérateur sur les collections avec deux paramètres	
Nom	Description
SubSequenceOperator	Extraction de la séquence instance d'une sous séquence dont les bornes sont données par les paramètres

i) Autres opérateurs

Autres opérateurs	
Nom	Description
IfOperator	Calcul la première (respectivement la seconde) opérande si la valeur de vérité de la condition est vrai (respectivement fausse)
AssignOperator	Affectation de l'opérande à l'attribut de l'instance
AssignDefaultOperator	Affectation de l'opérande à l'attribut de l'instance si l'attribut est vide
Assign CollectionOperator	Affectation de la collection opérande à l'attribut de l'instance
AssignDefault CollectionOperator	Affectation de la collection opérande à l'attribut de l'instance si l'attribut est vide