

# Object Constraint Language (OCL)

## *Une introduction*

MASTER 2 IFPRU - MODULE INGÉNIERIE DES MODÈLES - FMIN310

Marianne Huchard

12 octobre 2009

## 1 Présentation générale

OCL est un langage formel, basé sur la logique des prédicats du premier ordre, pour annoter les diagrammes UML en permettant notamment l'expression de contraintes.

### 1.1 Objectif du langage

Voici les arguments avancés pour l'introduction d'OCL.

- Accompagner les diagrammes UML de descriptions :
  - précises
  - non ambiguës
- Eviter cependant les désavantages des langages formels traditionnels qui sont peu utilisables par les utilisateurs et les concepteurs qui ne sont pas rompus à l'exercice des mathématiques :
  - rester facile à écrire ...
  - et facile à lire

Dans le cadre de l'ingénierie des modèles, la précision du langage OCL est nécessaire pour pouvoir traduire automatiquement les contraintes OCL dans un langage de programmation afin de les vérifier pendant l'exécution d'un programme.

### 1.2 Historique

OCL s'inspire de Syntropy [CD94] méthode basée sur une combinaison d'OMT (Object Modeling Technique) [RBE<sup>+</sup>97] et d'un sous-ensemble de Z.

A l'origine, OCL a été développé en 1997 par Jos Warmer (IBM), sur les bases du langage IBEL (Integrated Business Engineering Language). Il a été formellement intégré à UML 1.1 en 1999.

Nous présentons dans ce document OCL tel qu'il est proposé dans la version UML 2.0 [OMG03a]. Vous trouverez dans [MG00] une présentation d'OCL (UML 1.4) qui a servi de base à la réalisation de ce support de cours, et dans [Nyt] quelques exemples complémentaires.

### 1.3 Principe

**La notion de contrainte** Une contrainte est une expression à valeur booléenne que l'on peut attacher à n'importe quel élément UML.

Elle indique en général une restriction ou donne des informations complémentaires sur un modèle.

**Langage déclaratif** Les contraintes ne sont pas opérationnelles.

On ne peut pas invoquer de processus ni d'opérations autres que des requêtes.

On ne décrit pas le comportement à adopter si une contrainte n'est pas respectée.

**Langage sans effet de bord** Les instances ne sont pas modifiées par les contraintes.

### 1.4 Utilisation

Les contraintes servent dans plusieurs situations :

- description d'invariants sur les classes et les types
- pré-conditions et post-conditions sur les opérations et méthodes
- contraintes sur la valeur retournée par une opération ou une méthode
- règles de dérivation des attributs
- description de cibles pour les messages et les actions
- expression des gardes (conditions dans les diagrammes dynamiques)
- invariants de type pour les stéréotypes

Les contraintes servent en particulier à décrire la sémantique d'UML ou de ses diverses extensions, en participant à la définition des profils.

## 2 La notion de contexte

Une contrainte OCL est liée à un *contexte*, qui est le type, l'opération ou l'attribut auquel la contrainte se rapporte.

`context monContexte <stéréotype> :`  
Expression de la contrainte

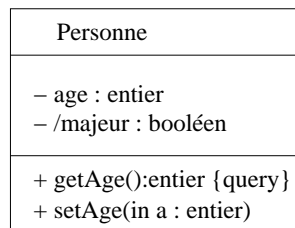


FIG. 1 – Une classe Personne

Le stéréotype peut prendre les valeurs suivantes :

- **inv** invariant de classe
- **pre** précondition
- **post** postcondition
- **body** indique le résultat d'une opération **query**

- `init` indique la valeur initiale d'un attribut
- `derive` indique la valeur dérivée d'un attribut

Par exemple, on peut définir les contraintes suivantes dans le contexte de la classe `Personne` de la figure 1, puis dans le contexte de ses méthodes `setAge`, `getAge` et pour son attribut `age`.

```
context Personne inv :
    (age <= 140) and (age >=0) - - l'âge est compris entre 0 et 140 ans
```

On peut placer des commentaires dans les expressions OCL, ils débutent par deux tirets et se prolongent jusqu'à la fin de la ligne.

```
context Personne::setAge(a :entier)
    pre : (a <= 140) and (a >=0) and (a >= age)
    post : age = a - on peut écrire également a=age
```

```
context Personne::getAge() :entier
    body : age
```

```
context Personne::age :entier
    init : 0
```

```
context Personne::majeur :booléen
    derive : age >= 18
```

**Version visuelle** La figure 2 présente une version alternative, graphique, de la contrainte sur l'âge de la personne.

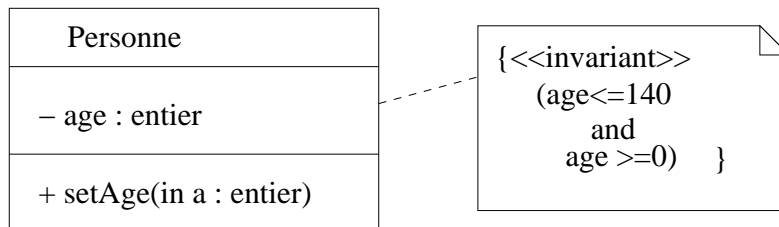


FIG. 2 – Une classe `Personne`

### Nommage de la contrainte

```
context Personne inv ageBorné :
    (age <= 140) and (age >=0) - - l'âge ne peut dépasser 140 ans
```

### Utilisation du mot-clef `self` pour désigner l'objet

Notez le caractère `.` qui offre la navigation (accès) à l'attribut.

```
context Personne inv :
    (self.age <= 140) and (self.age >=0) - - l'âge ne peut dépasser 140 ans
```

## Utilisation d'un nom d'instance formel

```
context p :Personne inv :  
  (p.age <= 140) and (p.age >=0) - - l'age ne peut dépasser 140 ans
```

**Question 2.1** Ajoutez un attribut *mère* de type *Personne* dans la classe *Personne*. Ecrivez une contrainte précisant que la mère d'une personne ne peut être cette personne elle-même et que l'âge de la mère doit être supérieur à celui de la personne.

### Réponse 2.1

```
context Personne inv :  
  self.mère <> self and self.mère.age > self.age
```

## 3 Types OCL

La figure 3 présente la hiérarchie des types en OCL. Nous les décrivons progressivement dans cette section et dans les suivantes.

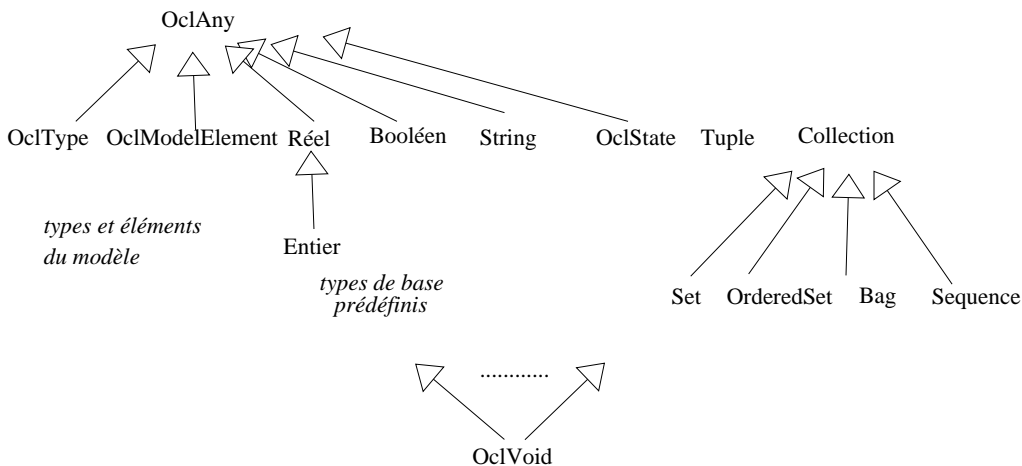


FIG. 3 – La hiérarchie des types en OCL

### 3.1 Types de base

Les types de base prédéfinis sont les suivants.

- Entier (Integer)
- Réel (Real)
- String
- Booléen (Boolean)

Quelques types spéciaux s'y ajoutent, en particulier :

- OclModelElement (énumération des éléments du modèle)
- OclType (énumération des types du modèle)
- OclAny (tout type autre que Tuple et Collection)
- OclState (pour les diagrammes d'états)
- OclVoid sous-type de tous les types

## Entier

*opérateurs* = <> + - \* / abs div mod max min < > <= >=  
- est unaire ou binaire

## Réel

*opérateurs* = <> + - \* / abs floor round max min < > <= >=  
- est unaire ou binaire

## String

*opérateurs* = size() concat(String) toUpper() toLower()  
substring(Entier, Entier)

Les chaînes de caractères constantes s'écrivent entre deux simples quotes :  
'voici une chaîne'

## Booléen

*opérateurs* = or xor and not

b1 implies b2

if b then expression1 else expression2 endif

Quelques exemples d'utilisation des opérateurs booléens, sur la figure 4.

Personne
- age : entier - majeur : Booléen - marié : Booléen - catégorie : enum {enfant,ado,adulte}

FIG. 4 – La classe Personne précisée

```
context Personne inv :  
  marié implies majeur
```

```
context Personne inv :  
  if age >=18 then majeur=vrai  
  else majeur=faux  
  endif
```

Cette dernière contrainte peut être plus concise (nous l'avons même déjà écrite avec `derive`) :

```
context Personne inv :  
  majeur = age >=18
```

## précédence des opérateurs

. ->
not - unaire
* /
+ -
if then else
< > <= >=
<> =
and or xor
implies

### 3.2 Types énumérés

Leurs valeurs apparaissent précédées de #. Par exemple, pour donner un invariant supplémentaire pour la figure 4, nous pourrions écrire :

```
context Personne inv :
  if age <=12 then catégorie =#enfant
  else if age <=18 then catégorie =#ado
  else catégorie=#adulte
  endif
endif
```

Cet invariant peut aussi s'écrire avec `derive`.

### 3.3 Types des modèles

Les types des modèles utilisables en OCL sont les « classificateurs », notamment les classes, les interfaces et les associations.

On peut écrire en particulier des expressions impliquant des objets dont les types sont reliés par une relation de spécialisation, grâce aux opérateurs :

- `oclAsType(t)` (conversion ascendante ou descendante de type vers t); la conversion ascendante sert pour l'accès à une propriété redéfinie; la conversion descendante sert pour l'accès à une nouvelle propriété.
- `oclIsTypeOf(t)` (vrai si t est supertype direct)
- `oclIsKindOf(t)` (vrai si t est supertype indirect)

Dans le cadre de la figure 5, nous pourrions écrire par exemple les expressions (r est supposé désigner une instance de Rectangle) :

- `p = r`
- `p.oclAsType(Rectangle)`
- `r.oclIsTypeOf(Rectangle)` (vrai)
- `r.oclIsKindOf(Polygone)` (vrai)

**Question 3.1** *En supposant l'existence d'un attribut `hauteur` dans la classe `Rectangle` et d'une méthode `hauteur() :Réel` dans `Polygone`, écrivez une contrainte dans `Polygone` disant que le résultat de `hauteur() :Réel` vaut `hauteur` pour les polygones qui sont des rectangles, sinon 0. Remarquez que c'est un exemple de mauvaise conception objet mais une excellente illustration des opérateurs présentés ci-dessus !*

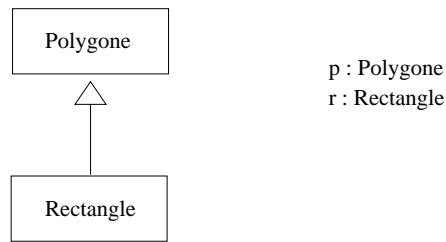


FIG. 5 – Polygones et rectangles

### Réponse 3.1

```

context Polygone::hauteur() post :
  if self.oclIsKindOf(Rectangle)
  then result=self.oclAsType(Rectangle).hauteur
  else result=0
  endif
  
```

## 4 Navigation dans les modèles

### 4.1 Accès aux attributs

L'accès (navigation) vers un attribut s'effectue en mentionnant l'attribut, comme nous l'avons déjà vu, derrière l'opérateur d'accès noté '.'

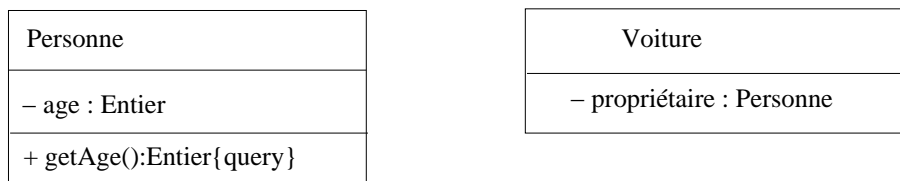


FIG. 6 – Personnes et voitures

Pour la classe **Voiture** de la figure 6, on peut ainsi écrire la contrainte ci-dessous.

```

context Voiture inv propriétaireMajeur :
  self.propiétaire.age >= 18
  
```

**Question 4.1** *Ecrivez pour le diagramme de la figure 7 la contrainte qui caractérise l'attribut dérivé **carteVermeil**. Un voyageur a droit à la carte vermeil si c'est une femme de plus de 60 ans ou un homme de plus de 65 ans.*

### Réponse 4.1

```

context Voyageur inv :
  carteVermeil = ((age >=65) or ((sexe=#féminin) and (age >=60)))
  
```

Cette contrainte peut également s'écrire avec *derive*.

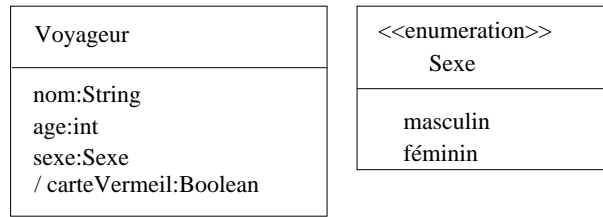


FIG. 7 – Voyageur

## 4.2 Accès aux opérations query

Il se fait avec une notation identique (utilisation du '.').

Pour la classe `Voiture` de la figure 6, on peut ainsi réécrire la contrainte `propriétaireMajeur`.

```

context Voiture inv :
    self.propriétaire.getAge() >= 18
  
```

## 4.3 Accès aux extrémités d'associations

La navigation le long des liens se fait en utilisant :

- soit les noms de rôles
- soit les noms des classes extrémités en mettant leur première lettre en minuscule, à condition qu'il n'y ait pas ambiguïté

Pour la classe `Voiture` de la figure 8, on peut ainsi réécrire la contrainte `propriétaireMajeur`.

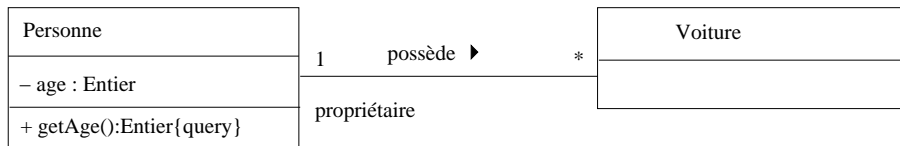


FIG. 8 – Personnes et voitures (avec association)

```

context Voiture inv :
    self.propriétaire.age >= 18
  
```

```

context Voiture inv :
    self.personne.age >= 18
  
```

**Question 4.2** *Donnez un exemple où l'utilisation du nom de classe conduit à une ambiguïté.*

**Réponse 4.2** *Deux associations entre Voiture et Personne (possède et aConduit)*

**Question 4.3 (Exercice tiré de [Nyt])** *Soient deux contraintes associées au diagramme de la figure 9 :*

```

context C1 inv :
    c2.attr2=c2.c3.attr3
  
```

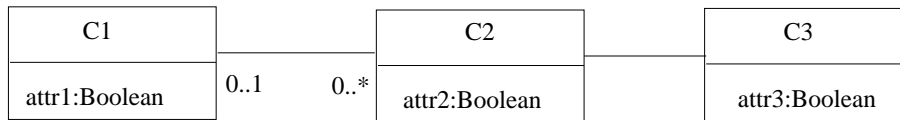


FIG. 9 – Contraintes ...

*context C2 inv :*  
*attr2=c3.attr3*

*Pensez-vous que les deux contraintes soient équivalentes ?*

**Réponse 4.3** *La première dit que pour des instances de C2 et C3 liées avec une instance de C1 les attributs attr2 et attr3 sont égaux, mais n'impose rien à des instances de C2 et C3 non liées à une instance de C1 (et il y en a à cause de la multiplicité).*

*La deuxième dit que pour tout couple d'instances de C2 et C3 liées, ces deux attributs sont égaux.*

#### 4.4 Navigation vers les classes association

Pour naviguer vers une classe association, on utilise le nom de la classe (en mettant le premier caractère en minuscule). Par exemple dans le cas du diagramme de la figure 10<sup>1</sup> :

*context p :Personne inv :*  
*p.contrat.salaire >= 0*

Ou encore, on précise le nom de rôle opposé (c'est même obligatoire pour une association réflexive) :

*context p :Personne inv :*  
*p.contrat[employeur].salaire >= 0*

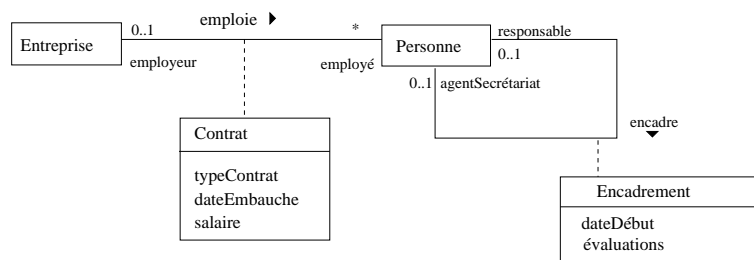


FIG. 10 – Emploi

Pour naviguer depuis une classe association, on utilise les noms des rôles ou de classes. La navigation depuis une classe association vers un rôle ne peut donner qu'un objet. Dans l'exemple ci-dessous, *c* est un lien, qui associe par définition une seule entreprise à un seul employé.

*context c :Contrat inv :*  
*c.employé.age >= 16*

<sup>1</sup>en UML 2, contrairement à UML 1.xx, le nom de la classe d'association et le nom de l'association devraient être identiques car il s'agit d'un même élément UML ; le diagramme est donc ici écrit en UML 1.xx

**Question 4.4** Dans la même figure 10, exprimez par une contrainte :

- le fait que le salaire d'un agent de secrétariat est inférieur à celui de son responsable ;
- un agent de secrétariat a un type de contrat "agentAdministratif" ;
- un agent de secrétariat a une date d'embauche antérieure à la date de début de l'encadrement (on suppose que les dates sont des entiers) ;
- écrire la dernière contrainte dans le contexte *Personne*.

**Réponse 4.4** Le salaire d'un agent de secrétariat est inférieur à celui de son responsable.

```
context e :encadrement inv :
    e.responsable.contrat.salaire >= e.agentSecrétariat.contrat.salaire
```

Un agent de secrétariat a un type de contrat "agentAdministratif".

```
context e :encadrement inv :
    e.agentSecrétariat.contrat.typeContrat='agentAdministratif'
```

Un agent de secrétariat a une date d'embauche antérieure à la date de début de l'encadrement (on suppose que les dates sont des entiers).

```
context e :encadrement inv :
    e.agentSecrétariat.contrat.dateEmbauche <= e.dateDebut
```

L'écrire dans le contexte *Personne* ...

```
context p :Personne inv :
    p.agentSecrétariat.contrat.dateEmbauche
    <= p.encadrement[agentSecrétariat].dateDebut
```

Dans ce dernier cas, pour être rigoureux il faudrait vérifier de manière préliminaire que *p.agentSecrétariat* est évalué, mais nous ne verrons que plus loin dans le cours comment faire.

## 5 Eléments du langage

### 5.1 Désignation de la valeur antérieure

Dans une post-condition, il est parfois utile de pouvoir désigner la valeur d'une propriété avant l'exécution de l'opération. Ceci se fait en suffixant le nom de la propriété avec `@pre`.

Par exemple, pour décrire une nouvelle méthode de la classe *Personne* :

```
context Personne::feteAnniversaire()
    pré : age < 140
    post : age = age@pre + 1
```

## 5.2 Définition de variables et d'opérations

On peut définir des variables pour simplifier l'écriture de certaines expressions. On utilise pour cela la syntaxe :

```
let variable : type = expression1 in
expression2
```

Par exemple, avec l'attribut dérivé `impot` dans `Personne`, on pourrait écrire :

```
context Personne inv :
let montantImposable : Réel = contrat.salaire*0.8 in
if (montantImposable >= 100000)
then impot = montantImposable*45/100
else if (montantImposable >= 50000)
  then impot = montantImposable*30/100
  else impot = montantImposable*10/100
endif
endif
```

Là encore, il serait plus correct de vérifier tout d'abord que `contrat` est valué.

Si on veut définir une variable utilisable dans plusieurs contraintes de la classe, on peut utiliser la construction `def`.

```
context Personne
def : montantImposable : Réel = contrat.salaire*0.8
```

Lorsqu'il est utile de définir de nouvelles opérations, on peut procéder avec la même construction `def`.

```
context Personne
def : ageCorrect(a :Réel) :Booléen = a>=0 and a<=140
```

Une telle définition permet de réécrire la pré-condition de l'opération `setAge` et l'invariant sur l'âge d'une personne de manière plus courte.

```
context Personne::setAge(a :entier)
pre : ageCorrect(a) and (a >= age)
```

```
context Personne inv :
ageCorrect(age) - - l'âge ne peut dépasser 140 ans
```

## 5.3 Retour de méthode

L'objet retourné par une opération est désigné par `result`.

```
context Personne::getAge()
post : result=age
```

Lorsque la postcondition se résume à décrire la valeur du résultat c'est donc équivalent à l'utilisation de `body`.

**Question 5.1** Proposez une classe *Etudiant*, disposant de 3 notes et munie d'une opération *mention* qui retourne la mention de l'étudiant sous forme d'une chaîne de caractères. Ecrivez les contraintes et en particulier utilisez le *let* pour écrire la postcondition de *mention*.

**Réponse 5.1**

```

context Etudiant::mention() post :
let moyenne : Real =(note1+note2+note3)/3 in
if (moyenne >= 16)
then result='très bien'
else if (moyenne >= 14)
then result='bien'
else result='moins bien'
endif
endif

```

## 6 Collections

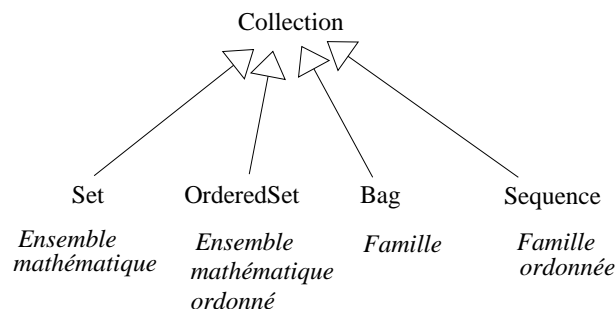


FIG. 11 – Les types de collections

### 6.1 Types de collections

La figure 11 présente les cinq types de collections existant en OCL :

- **Collection** est un type abstrait
- **Set** correspond à la notion mathématique d'ensemble
- **OrderedSet** correspond à la notion mathématique d'ensemble ordonné
- **Bag** correspond à la notion mathématique de famille (un élément peut y apparaître plusieurs fois)
- **Sequence** correspond à la notion mathématique de famille, et les éléments sont, de plus, ordonnés.

On peut définir des collections par des littéraux de la manière suivante :

- Set { 2, 4, 6, 8 }
- OrderedSet { 2, 4, 6, 8 }
- Bag { 2, 4, 4, 6, 6, 8 }
- Sequence { 'le', 'chat', 'boit', 'le', 'lait' }
- Sequence { 1..10 } spécification d'un intervalle d'entiers

Les collections peuvent avoir plusieurs niveaux, il est possible de définir des collections de collections, par exemple : Set { 2, 4, Set {6, 8 } }

## 6.2 Les collections comme résultats de navigation

**Set** On obtient naturellement un **Set** en naviguant le long d'une association ordinaire (ne mentionnant pas **bag** ou **seq** à l'une de ses extrémités). Par exemple, pour la figure 10, l'expression `self.employé` dans le contexte `Entreprise` est un **Set**.

**Bag** La figure 12 illustre une navigation permettant d'obtenir un **Bag** : c'est le résultat de l'expression `self.occurrence.mot` dans le contexte `Texte`. Cette expression est, par ailleurs, un raccourci pour une notation que nous verrons ci-après.



FIG. 12 – Occurrences de mots dans un texte

**OrderedSet** On peut obtenir un ensemble ordonné en naviguant vers une extrémité d'association munie de la contrainte **ordonné**, par exemple avec l'expression `self.personne` dans le contexte `FileAttente`.

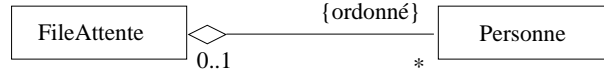


FIG. 13 – File d'attente

**Sequence** On peut obtenir une séquence en naviguant vers une extrémité d'association munie de la contrainte **seq**, par exemple avec l'expression `self.mot` dans le contexte `Texte` de la figure 14.



FIG. 14 – Une autre représentation des textes

**Question 6.1** *Qu'obtient-on en accédant à un attribut multi-valué ?*

**Réponse 6.1 (Question 6.1)** *On obtient un bag si une valeur peut être répétée plusieurs fois ; une séquence si les valeurs sont de plus ordonnées ; un set s'il n'y a jamais de doublon dans les valeurs ; un orderedSet si les valeurs sont de plus ordonnées.*

## 6.3 Opérations de Collection

Notez que les langages de transformation de modèles qui voient le jour actuellement disposent d'opérateurs assez semblables à ceux proposés par OCL, à des fins de parcours des modèles.

### 6.3.1 Remarques générales

Dans la suite, nous considérons des collections d'éléments de type T.

**Appel** Les opérations sur une collection sont généralement mentionnées avec ->

**Itérateurs** Les opérations qui prennent une expression comme paramètre peuvent déclarer optionnellement un itérateur, par exemple toute opération de la forme `operation(expression)` existe également sous deux formes plus complexes.

```
operation(v | expression-contenant-v)
operation(v : T | expression-contenant-v)
```

### 6.3.2 Opérations sur tous les types de collections

**isEmpty() :Boolean**

**notEmpty() :Boolean**

Permettent de tester si la collection est vide ou non. Ils servent en particulier à tester l'existence d'un lien dans une association dont la multiplicité inclut 0. Pour revenir sur l'exemple des employés (dernière question) :

```
context p :Personne inv :
  p.agentSecretariat->notEmpty() implies - p est un responsable
  p.agentSecrétariat.contrat.dateEmbauche
  <= p.encadrement[agentSecrétariat].dateDebut
```

**size() :Entier**

Donne la taille (nombre d'éléments).

**count(unObjet :T) :Entier**

Donne le nombre d'occurrences de unObjet.

**sum() :T**

Addition de tous les éléments de la collection (qui doivent supporter une opération + associative)

Nous pouvons par exemple définir une variable représentant l'âge moyen des employés d'une entreprise.

```
context e :Entreprise
  def : ageMoyen : Real = e.employé.age->sum() / e.employé.age->size()
```

Ou encore, pour une classe **Etudiant** généralisée, disposant d'un attribut **notes :Real[\*]**, on peut définir la variable moyenne.

```
context e :Etudiant
  def : moyenne : Real = e.notes->sum() / e.notes->size()
```

**includes(o :T) :Boolean**

**excludes(o :T) :Boolean**

Retourne vrai (resp. faux) si et seulement si o est (resp. n'est pas) un élément de la collection.

Toujours pour la classe **Etudiant** généralisée, disposant d'un attribut **notes :Real[\*]**, on peut définir l'opération **estEliminé() :Boolean**. Un étudiant est éliminé s'il a obtenu un zéro.

```
context e :Etudiant
def : estEliminé() : Boolean = e.notes->includes(0)
```

**includesAll(c :Collection(T)) :Boolean**

**excludesAll(c :Collection(T)) :Boolean**

Retourne vrai (resp. faux) si et seulement si la collection contient tous les (resp. ne contient aucun des) éléments de c.

**forAll(uneExpression :Boolean) :Boolean**

Vaut vrai si et seulement si **uneExpression** est vraie pour tous les éléments de la collection.

Dans une entreprise, une contrainte est que tous les employés aient plus de 16 ans.

```
context e :Entreprise inv :
e.employe->forAll(emp :Personne | emp.age >= 16)
```

Cette opération a une variante permettant d'itérer avec plusieurs itérateurs sur la même collection. On peut ainsi parcourir des ensembles produits de la même collection.

```
forAll(t1,t2 :T | expression-contenant-t1-et-t2)
```

En supposant que la classe **Personne** dispose d'un attribut **nom**, on peut admettre que deux employés différents n'ont jamais le même nom.

```
context e :Entreprise inv :
e.employe->forAll(e1,e2 :Personne | e1 <> e2 implies e1.nom <> e2.nom)
```

**exists(uneExpression :Boolean)**

Vrai si et seulement si au moins un élément de la collection satisfait **uneExpression**.

Dans toute entreprise, il y a au moins une personne dont le contrat est de type "agent administratif".

```
context e :Entreprise inv :
e.contrat->exists(c :Contrat | c.typeContrat='agent administratif')
```

**iterate(i :T; acc :Type = uneExpression |**

**ExpressionAvecietacc) :**

Cette opération permet de généraliser et d'écrire la plupart des autres.

*i* est un itérateur sur la collection

*acc* est un accumulateur initialisé avec *uneExpression*

L'expression *ExpressionAvecietacc* est évaluée pour chaque *i* et son résultat est affecté dans *acc*. Le résultat de *iterate* est *acc*.

Cette opération permet dans l'exemple ci-dessous de caractériser le résultat d'une hypothétique opération **masseSalariale** d'une entreprise.

```

context Entreprise::masseSalariale() :Real post :
result = employé->iterate(p :Personne; ms :Real=0 | ms+p.contrat.salaire)

```

**Question 6.2** *Ecrire `size` et `forall` dans le contexte de la classe `Collection`.*

**Réponse 6.2 (Question 6.2)** *Voici comment ces méthodes sont spécifiées dans [OMG01].*

```

context Collection : :size() post :
result = self->iterate(elem; acc :Integer=0| acc+1)

```

```

context Collection : :forall(expr) post :
result = self->iterate(elem; acc :Boolean=true| acc and expr)

```

**isUnique(uneExpression :BooleanExpression) :Boolean**

Vrai si et seulement si `uneExpression` s'évalue avec une valeur distincte pour chaque élément de la collection.

**sortedBy(uneExpression) :Sequence**

Retourne une séquence contenant les éléments de la collection triés par ordre croissant suivant le critère `uneExpression`. L'évaluation de `uneExpression` doit donc supporter l'opération `<`.

**Question 6.3** *Ecrire la `post` condition d'une méthode `salariésTriés` dans le contexte de la classe `Entreprise`. Cette méthode retourne les employés ordonnés suivant leur salaire.*

**Réponse 6.3 (Question 6.3)**

```

context Entreprise : :salariesTries() :OrderedSet(Personne) post :
result = self.employe->sortedBy(p | p.contrat.salaire)

```

**any(uneExpression :OclExpression) :T**

Retourne n'importe quel élément de la collection qui vérifie `uneExpression`.

**one(uneExpression :OclExpression) :Booléen**

Vrai si et seulement si un et un seul élément de la collection vérifie `uneExpression`.

### 6.3.3 Opérations sur tous les types de collections, définies spécifiquement sur chaque type

Le lecteur doit être averti du fait que la présentation des opérateurs proposée ci-dessous privilégie la concision de l'exposé. Certaines opérations ont des signatures plus précises que celles mentionnées ici. Par exemple `select`, commune aux trois types de collection concrets, n'est pas définie dans `Collection`, mais seulement dans les types concrets avec des signatures spécifiques :

```
select(expr :BooleanExpression) : Set(T)
select(expr :BooleanExpression) : OrderedSet(T)
select(expr :BooleanExpression) : Bag(T)
select(expr :BooleanExpression) : Sequence(T)
```

Dans un tel cas, nous présentons une généralisation de ces opérations  
`select(expr :BooleanExpression) : Collection(T)`  
Le lecteur est renvoyé à la documentation OCL [OMG03a] pour les signatures précises.

```
select(expr :BooleanExpression) : Collection(T)
reject(expr :BooleanExpression) : Collection(T)
```

Retourne une collection du même type construite par sélection des éléments vérifiant (resp. ne vérifiant pas) `expr`.

L'expression représentant dans le contexte d'une entreprise les employés âgés de plus de 60 ans serait la suivante :

```
self.employé->select(p :Personne | p.age >= 60)
```

```
collect(expr :BooleanExpression) : Collection(T)
```

Retourne une collection composée des résultats successifs de l'application de `expr` à chaque élément de la collection.

La manière standard d'atteindre les mots d'un texte s'écrit ainsi dans le contexte d'un texte (Figure 12) :

```
self.occurrence->collect(mot)
self.occurrence->collect(o | o.mot)
self.occurrence->collect(o :Occurrence | o.mot)
```

Comme nous l'avons évoqué précédemment, cette notation admet un raccourci :

```
self.occurrence.mot
```

```
including(unObjet :T) :Collection(T)
excluding(unObjet :T) :Collection(T)
```

Retourne une collection résultant de l'ajout (resp. du retrait) de `unObjet` à la collection.

On peut ainsi décrire la post-condition d'une opération `embauche(p :Personne)` de la classe `Entreprise`.

```
context Entreprise : :embauche(p :Personne) post :
    self.employé = self.employé@pre->including(p)
```

**opération =**

```
union(c :Collection) :Collection
```

## 6.4 Opérations de conversion

Chaque sorte de collection (`set`, `orderedSet`, `sequence`, `bag`) peut être transformée dans n'importe quelle autre sorte.

Par exemple un `Bag` peut être transformé en :

- `sequence` par l'opération `asSequence() : Sequence(T)` ; l'ordre résultant est indéterminé
- `ensemble` par l'opération `asSet() : Set(T)` ; les doublons sont éliminés
- `ensemble ordonné` par l'opération `asOrderedSet() : OrderedSet(T)` ; les doublons sont éliminés ; l'ordre résultant est indéterminé.

A titre d'illustration, si nous voulons exprimer le fait que le nombre de mots différents d'un texte est plus grand que deux (figure 12) :

```
context Texte inv :
self.occurrence->collect(mot)->asSet()->size() >= 2
```

## 6.5 Opérations propres à Set et Bag

- `intersection`

## 6.6 Opérations propres à OrderedSet et Sequence

- ajout d'un objet à la fin
  - `append(unObjet : T) : OrderedSet(T)` pour les ensembles ordonnés
  - `append(unObjet : T) : Sequence(T)` pour les séquences
- ajout d'un objet au début
  - `prepend(unObjet : T) : OrderedSet(T)` pour les ensembles ordonnés
  - `prepend(unObjet : T) : Sequence(T)` pour les séquences
- objets d'un intervalle
  - `subOrderedSet(inf :Entier, sup :Entier) : OrderedSet(T)` pour les ensembles ordonnés
  - `subSequence(inf :Entier, sup :Entier) : Sequence(T)` pour les séquences
- `at(inf :Entier) : T`
- `first() : T`
- `last() : T`
- insertion d'un objet à un certain indice
  - `insertAt(i :Integer, o :T) : OrderedSet(T)` pour les ensembles ordonnés
  - `insertAt(i :Integer, o :T) : Sequence(T)` pour les séquences
- `indexOf(o :T) : Integer`

**Question 6.4** *Ecrire la post-condition de `sortedBy`*

## 6.7 Relation de « conformance de type »

La figure 15 présente les relations de conformance de type prévues en OCL.

Si la classe `Assiette` est sous-classe de `Vaisselle`, on peut en déduire :

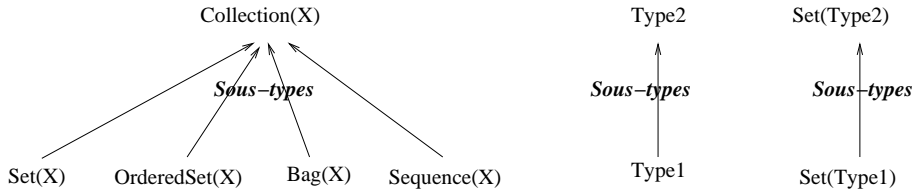


FIG. 15 – Sous-typage de types paramétrés

- `Set(Assiette)` se conforme à `Collection(Assiette)`
- `Set(Assiette)` se conforme à `Set(Vaisselle)`
- `Set(Assiette)` se conforme à `Collection(Vaisselle)`

OCL parle même de relation de sous-typage. Nous étudions la question de la substituabilité sur un bref exemple qui peut être généralisé et qui illustre le fait que, les collections étant constantes, la conformance rejoint ici la substituabilité.

Prenons le cas d'un `Set(Assiette)`. Considéré comme un `Set(Vaisselle)`, on peut lui appliquer l'opération `including(object :Vaisselle) :Set(Vaisselle)` où `object` peut être tout à fait autre chose qu'une assiette. Ce faisant on crée et retourne un `Set(Vaisselle)` (et non un `Set(Assiette)`), ce qui n'est pas problématique (il n'y a pas d'erreur de type, on n'a pas introduit un verre dans un ensemble d'assiettes).

## 7 Utilisation d'OCL pour les gardes et les actions

De manière générale, OCL est un langage de navigation qui peut être utilisé dans différents points des diagrammes UML. Nous allons illustrer cet aspect d'OCL dans le cadre des diagrammes d'états, pour exprimer des gardes et préciser des actions.

### 7.1 Gardes

Les gardes sont des expressions booléennes utilisées dans le cadre des diagrammes UML basés sur les automates d'état finis : diagrammes d'états et diagrammes d'activités.

Elles conditionnent le déclenchement d'une transition lors de l'occurrence d'un événement. En UML elles sont notées entre crochets.

Les diagrammes d'états modélisent le comportement d'éléments UML (le plus souvent le comportement d'une instance d'une classe donnée). Ils se composent d'*états* et de *transitions*.

Un *état* est un moment de la vie d'une instance déterminé par le fait qu'elle vérifie une condition particulière, est en train d'effectuer une opération ou attend un événement. Il est représenté en UML par une boîte aux angles arrondis.

Une *transition* représente le passage d'un état à un autre. Ce passage est déclenché par un *événement* : une modification (une condition sur l'instance devient vraie), l'appel d'une opération, la réception d'un signal asynchrone, la fin d'une période de temps. Les gardes sont associées aux transitions, elles sont évaluées lorsqu'un événement survient.

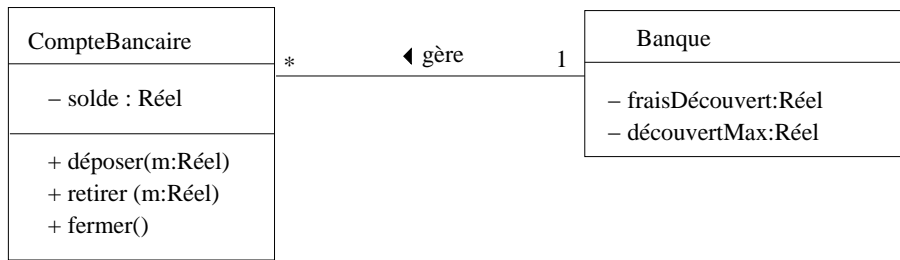


FIG. 16 – Comptes bancaires

La figure 17, tirée de [OMG00], présente un diagramme d'états simple pour un compte bancaire dont le diagramme de classes est décrit sur la figure 16. Un compte bancaire peut avoir deux états, *ouvert* ou *fermé*. Il passe de l'état *ouvert* à l'état *fermé* lorsque l'opération *fermer()* est appelée sur l'objet. Il reste dans l'état *ouvert* lorsque deux opérations, respectivement *déposer* et *retirer* sont appelées. La garde qui suit l'opération *retirer* indique que cette dernière n'a d'effet que si le montant retiré est positif et inférieur au solde du compte auquel on ajoute une somme représentant le découvert autorisé.

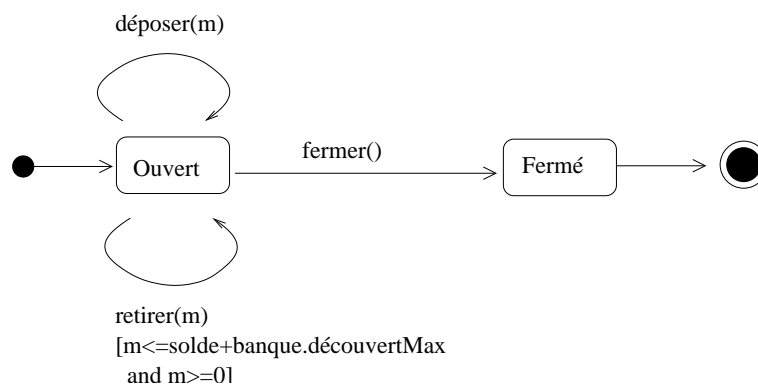


FIG. 17 – Etats principaux d'un compte bancaire

## 7.2 Actions

OCL peut être utilisé également dans la description d'actions qui étiquettent les transitions. Ces actions prennent place dans l'étiquette derrière le symbole /. Dans ce cas on se sert d'OCL comme d'un langage de navigation, mais on n'écrit pas forcément une expression booléenne.

La figure 18 décrit ainsi les sous-états de *Ouvert*. Un compte ouvert est débiteur ou créditeur. Tout dépôt sur un solde débiteur entraîne le retrait d'une somme forfaitaire pour frais de gestion des découverts. Le retrait est impossible sur un compte débiteur. Dans la situation où le compte est créditeur, un retrait d'une somme supérieure au solde sans dépassement du découvert autorisé fait passer le compte dans l'état débiteur. Inversement, lorsque le compte est débiteur, le dépôt d'une somme supérieure au débit (augmenté des frais de découvert) fait passer le compte dans l'état créditeur.

Les actions sont exprimées à l'aide du symbole := et d'expressions de navigation en OCL, comme par exemple l'accès à l'attribut `fraisDécouvert`.

Le diagramme d'états étant naturellement décrit pour une instance d'une classe (compte bancaire), ces expressions de navigation sont écrites comme si nous nous trouvions dans `contexte CompteBancaire`.

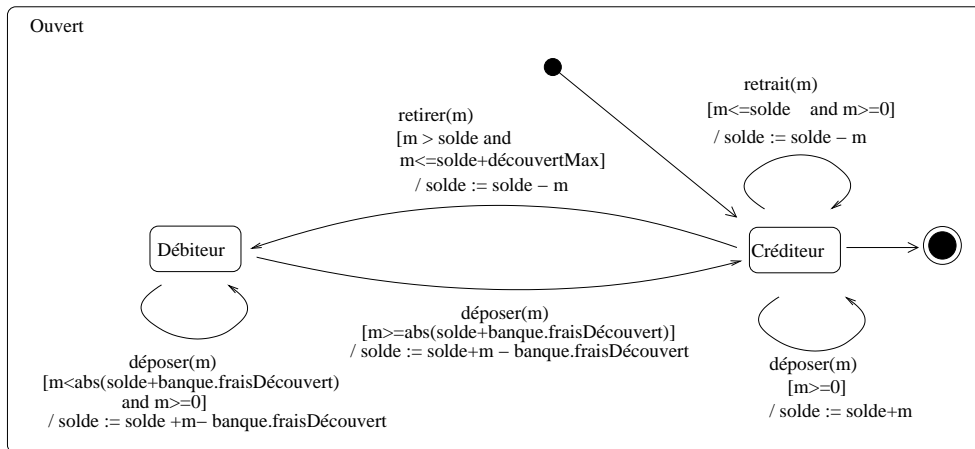


FIG. 18 – Etat ouvert d'un compte bancaire

Alternativement ou parallèlement, une partie des informations portées sur le diagramme d'états peut apparaître dans le diagramme de classes. La figure 19 présente ainsi des précisions sur le diagramme de la figure 16. Les contraintes OCL sont présentées de manière graphique dans des notes. L'état apparaît comme un attribut (`étatSolde`) de la classe `CompteBancaire`. Les gardes du diagramme d'états correspondent dans le diagramme de classes à des préconditions ou à des conditions dans la post-condition, tandis que les actions sont associées à l'écriture de la post-condition. Cet exemple doit aussi nous inviter à réfléchir au problème de la cohérence entre diagrammes. En effet, les informations que portent le diagramme d'états et le diagramme de classes se doivent d'être en accord.

## 8 OCL dans le méta-modèle UML

Dans la première partie du cours, nous avons abordé le méta-modèle UML qui est une description du langage d'expression des modèles. La description du méta-modèle utilise le même formalisme que celle des modèles. Les éléments du méta-modèle sont les méta-classes décrivant les classes, les attributs, les associations, les opérations, les composants : `Class`, `Attribute`, `Operation`, etc. Les éléments d'un modèle sont des instances de ces méta-classes.

Dans la documentation qui décrit le méta-modèle UML, les contraintes sont utilisées pour préciser des invariants sur les méta-classes et notamment pour décrire des extensions du méta-modèle à l'aide de stéréotypes. Notez cependant que ces règles ne décrivent pas *toute* la sémantique UML. Une partie de cette sémantique reste écrite en langue naturelle, et une partie n'est pas décrite du tout.

Dans cette partie nous allons en donner quelques exemples liés aux propriétés, à la relation de spécialisation/généralisation, aux associations et enfin aux contraintes.

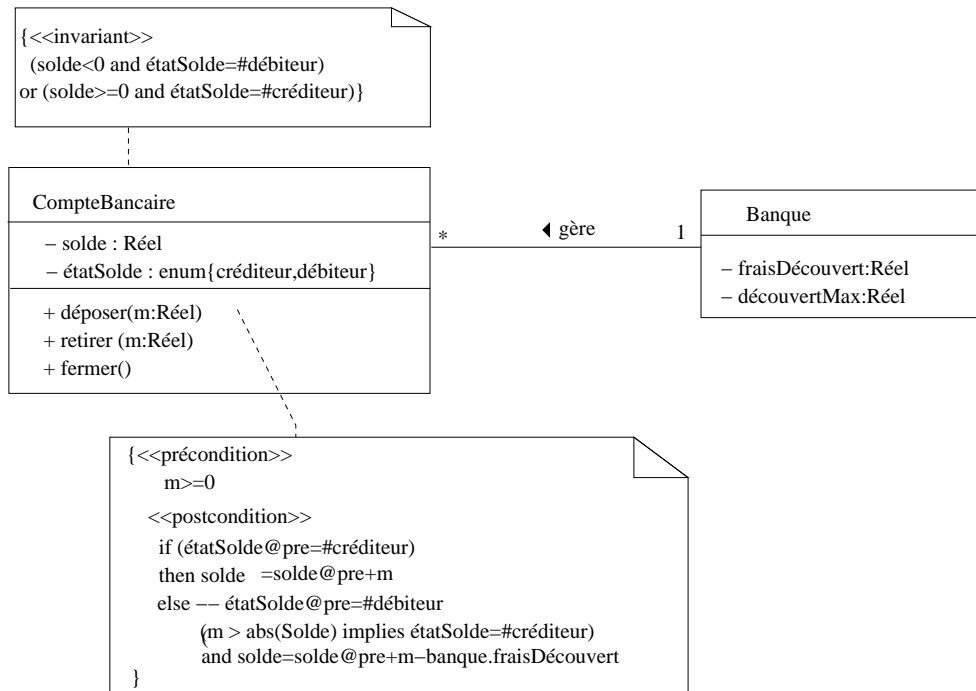


FIG. 19 – Comptes bancaires détaillés

Le document utilisé est [OMG03b]. Nous ajoutons la partie `context` pour plus de clarté.

## 8.1 Contraintes sur les propriétés

### Règle 8.1

Quelques contraintes établissent des règles sur la multiplicité (Figure 10 du méta-modèle [OMG03b]).

```

context MultiplicityElement inv :
lower = lowerBound()
lowerBound()->notEmpty() implies lowerBound() >= 0
lowerBound()->notEmpty() and upperBound()->notEmpty()
implies upperBound() >= lowerBound()
  
```

```

context MultiplicityElement : lowerBound() : Integer inv :
result=if lowerValue->isEmpty() then 1
else lowerValue().integerValue() endif
  
```

*upperBound()* est définie de manière similaire.

## 8.2 Contraintes sur la spécialisation/généralisation

Nous montrons tout d'abord comment les super-classes directes d'une classe sont obtenues (figure 22 dans [OMG03b]).

```

context Classifier::parents() :Set(Classifier) post :
result = self.generalization.parent
  
```

Cette opération se généralise facilement pour obtenir toutes les super-classes d'une classe.

```
context Classifier::allParents() :Set(Classifier) post :  
result = self.parents()->union(self.parents()->collect(p | p.allParents()))
```

### Règle 8.2

*La relation de spécialisation/généralisation ne peut induire un circuit.*

```
context Classifier inv :  
not self.allParents() -> includes(self)
```

### 8.3 Contraintes sur les associations

Les associations sont décrites dans la figure 30 du document [OMG03b].

#### Règle 8.3

*endType est dérivé à partir des types des memberEnd*

```
context Association inv :  
self.endType = self.memberEnd -> collect(e | e.type)
```

*La multiplicité dans une composition ne peut dépasser 1.*

```
context Property inv :  
isComposite implies (upperBound()->isEmpty() or upperBound() <= 1)
```

### 8.4 Contraintes sur les contraintes

D'après le diagramme 14 du méta-modèle UML [OMG03b].

#### Règle 8.4

*Une contrainte ne peut pas s'appliquer à elle-même.*

```
context Constraint inv :  
not constrainedElement->include(self)
```

Sur ce diagramme, nous voyons apparaître une limitation d'OCL dans l'expression des contraintes. La sémantique voudrait notamment que :

- la spécification de valeur pour une contrainte soit de type booléen ;
- l'évaluation d'une contrainte ne devrait pas avoir d'effet de bord.

D'après la documentation, ces deux contraintes ne peuvent être exprimées en OCL.

## Références

- [CD94] Steve Cook and John Daniels. *Designing Object Systems : Object-Oriented Modelling with Syntropy*. Prentice Hall, [http ://www.syntropy.co.uk/syntropy/](http://www.syntropy.co.uk/syntropy/), 1994. ISBN 0-13-203860-9.
- [MG00] Pierre-Alain Muller and Nathalie Gaertner. *Modélisation objet avec UML*. Eyrolles, 2000.
- [Nyt] Jan Pettersen Nytnun. OCL, The Object Constraint Language. Slides. [http :// fag.grm.hia.no/ ikt2340/ year2002/ themes/ executableUML/ executableUML.html](http://fag.grm.hia.no/ikt2340/year2002/themes/executableUML/executableUML.html).
- [OMG00] OMG. Unified modeling language v1.3. Technical report, Object Management Group, 2000. UML Semantics.
- [OMG01] OMG. Unified modeling language v1.4. Technical report, Object Management Group, 2001. chap. 6 - Object Constraint Language Specification.
- [OMG03a] OMG. Uml 2.0 ocl specification. Technical report, Object Management Group, 2003. ptc/03-10-14.
- [OMG03b] OMG. Uml 2.0 superstructure specification. Technical report, Object Management Group, 2003. ptc/03-08-02.
- [RBE<sup>+</sup>97] James Rumbaugh, Michel Blaha, Frederick Eddy, William Premerlani, and W. Lorensen. *OMT, Modélisation et conception orientées objet*. MASSON, 1997. Edition française.