1 Intérêt d'un langage de contraintes objet comme OCL

1.1 OCL – Introduction

QuesacOCL ?

C'est avec OCL (*Object Constraint Language*) qu'UML formalise l'expression des contraintes . Il s'agit donc d'un langage formel d'expression de contraintes bien adapté aux diagrammes d'UML, et en particulier au diagramme de classes.
OCL existe depuis la version 1.1 d'UML et est une contribution d'IBM. OCL fait partie intégrante de la norme UML depuis la version 1.3 d'UML. Dans le cadre d'UML 2.0, les spécifications du langage OCL figurent dans un document indépendant de la norme d'UML, décrivant en détail la syntaxe formelle et la façon d'utiliser ce langage.
OCL peut s'appliquer sur la plupart des diagrammes d'UML et permet de spécifier des contraintes sur l'état d'un objet ou d'un ensemble d'objets comme :

- des invariants sur des classes ;
- des préconditions et des postconditions à l'exécution d'opérations :
  - les préconditions doivent être vérifiées avant l'exécution,
  - les postconditions doivent être vérifiées après l'exécution ;
- des gardes sur des transitions de diagrammes d'états-transitions ou des messages de diagrammes d'interaction ;
- des ensembles d'objets destinataires pour un envoi de message ;
- des attributs dérivés, etc.

Pourquoi OCL ?

Nous avons dit que les contraintes pouvaient être écrites en langage naturel, alors pourquoi s'embarrasser du langage OCL ? L'intérêt du langage naturel est qu'il est simple à mettre en œuvre et compréhensible par tous. Par contre (et comme toujours), il est ambigu et imprécis, il rend difficile l'expression des contraintes complexes et ne facilite pas les références à d'autres éléments (autres que celui sur lequel porte la contrainte) du modèle.
OCL est un langage formel volontairement simple d'accès. Il possède une grammaire élémentaire (OCL peut être interprété par des outils) que nous décrirons dans les sections 4.3 à 4.6. OCL représente, en fait, un juste milieu entre le langage naturel et un langage très technique (langage mathématique, informatique, …). Il permet ainsi de limiter les ambiguïtés, tout en restant accessible.

1.2 Illustration par l'exemple

Mise en situation

Plaçons-nous dans le contexte d'une application bancaire. Il nous faut donc gérer :

- des comptes bancaires,
- des clients,
- et des banques.

De plus, on aimerait intégrer les contraintes suivantes dans notre modèle :

- un compte doit avoir un solde toujours positif ;
- un client peut posséder plusieurs comptes ;
- une personne peut être cliente de plusieurs banques ;
- un client d'une banque possède au moins un compte dans cette banque ;
- un compte appartient forcément à un client ;
- une banque gère plusieurs comptes ;
- une banque possède plusieurs clients.

Diagramme de classes

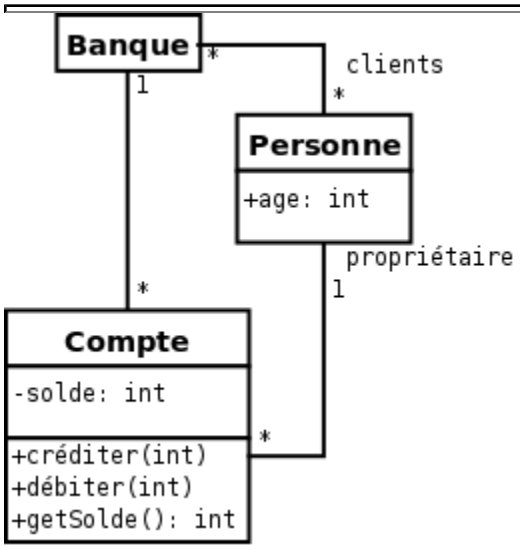


Figure 4.3: Diagramme de classes modélisant une banque, ses clients et leurs comptes.

La figure 4.3 montre un diagramme de classes correspondant à la problématique que nous venons de décrire.

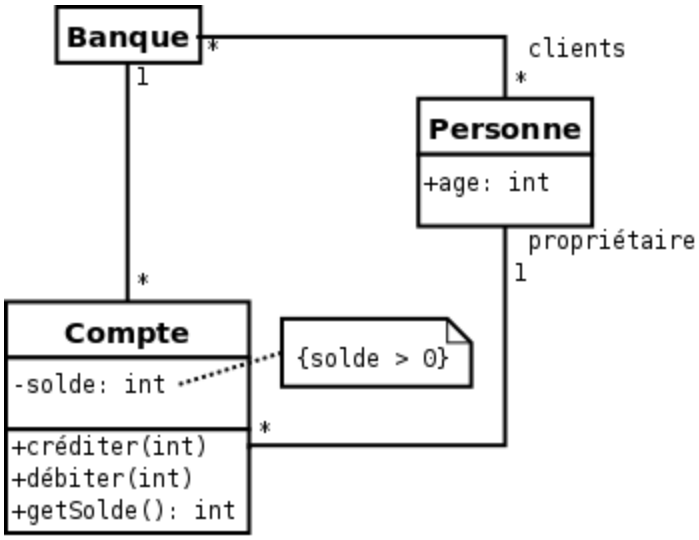


Figure 4.4: Ajout d'une contrainte sur le diagramme de la figure 4.3.

Un premier problème apparaît immédiatement : rien ne spécifie, dans ce diagramme, que le solde du client doit toujours être positif. Pour résoudre le problème, on peut simplement ajouter une note précisant cette contrainte (*{solde > 0}*), comme le montre la figure 4.4.

Figure 4.5: Diagramme d'objets cohérent avec le diagramme de classes de la figure 4.4.

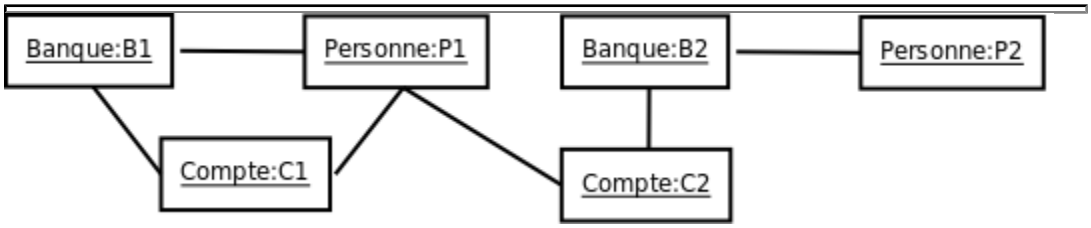
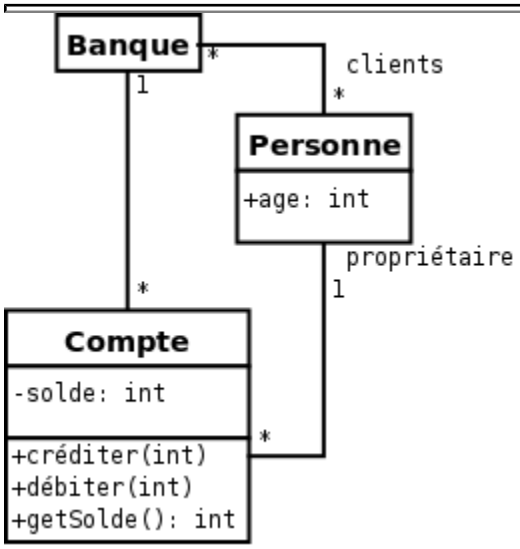


Figure 4.6: Diagramme d'objets cohérent avec le diagramme de classes de la figure 4.4, mais représentant une situation inacceptable.

Cependant, d'autres problèmes subsistent. La figure 4.5 montre un diagramme d'objets valide vis-à-vis du diagramme de classes de la figure 4.4 et également valide vis-à-vis de la spécification du problème. Par contre, la figure 4.6 montre un diagramme d'objets valide vis-à-vis du diagramme de classes de la figure 4.4 mais ne respectant pas la spécification du problème. En effet, ce diagramme d'objets montre une personne (P1) ayant un compte

dans une banque sans en être client. Ce diagramme montre également un client (P2) d'une banque n'y possédant pas de compte.



context Compte
inv : solde > 0

context Compte :: débiter(somme : int)
pre : somme > 0
post : solde = solde@pre - somme

context Compte
inv : banque.clients -> includes (propriétaire)
Figure 4.7: Exemple d'utilisation du langage de contrainte OCL sur l'exemple bancaire.

Le langage OCL est particulièrement adapté à la spécification de ce type de contrainte. La figure 4.7 montre le diagramme de classes de notre application bancaire accompagné des contraintes OCL adaptées à la spécification du problème.

Remarque :

faites bien attention au fait qu'une expression OCL décrit une contrainte à respecter et ne décrit absolument pas l'implémentation d'une méthode.

2 Typologie des contraintes OCL

2.1 Contexte (*context*)

Une contrainte est toujours associée à un élément de modèle. C'est cet élément qui constitue le contexte de la contrainte. Il existe deux manières pour spécifier le contexte d'une contrainte OCL.

Syntaxe

```
context <élément>
```

Exemple

Le contexte est la classe *Compte*:

```
context Compte
```

Le contexte est l'opération *getSolde()* de la classe *Compte*:

```
context Compte::getSolde()
```

2.2 Invariants (*inv*)

Un invariant exprime une contrainte prédicative sur un objet, ou un groupe d'objets, qui doit être respectée en permanence.

Syntaxe

```
inv : <expression_logique>
```

<expression_logique> est une expression logique qui doit toujours être vraie.

Exemple

Le solde d'un compte doit toujours être positif.

context Compte
inv : solde > 0

Les femmes (au sens de l'association) des personnes doivent être des femmes (au sens du genre).

context Personne
inv : femme->forAll(genre=Genre::femme)

3.2 Préconditions et postconditions (*pre*, *post*)

Une précondition (respectivement une postcondition) permet de spécifier une contrainte prédicative qui doit être vérifiée avant (respectivement après) l'appel d'une opération. Dans l'expression de la contrainte de la postcondition, deux éléments particuliers sont utilisables :

- l'attribut result qui désigne la valeur retournée par l'opération,
- et <nom_attribut>@pre qui désigne la valeur de l'attribut <nom_attribut> avant l'appel de l'opération.

Syntaxe

- Précondition :

```
pre : <expression_logique>
```

- Postcondition :

```
post : <expression_logique>
```

<expression_logique> est une expression logique qui doit toujours être vraie.

Exemple

Concernant la méthode *débiter* de la classe *Compte*, la somme à débiter doit être positive pour que l'appel de l'opération soit valide et, après l'exécution de l'opération, l'attribut *solde* doit avoir pour valeur la différence de sa valeur avant l'appel et de la somme passée en paramètre.

context Compte::débiter(somme : Real)
pre : somme > 0
post : solde = solde**@pre** - somme

Le résultat de l'appel de l'opération *getSolde* doit être égal à l'attribut *solde*.

context Compte::getSolde() : Real
post : result = solde

4.2 Résultat d'une méthode (*body*)

Ce type de contrainte permet de définir directement le résultat d'une opération.

Syntaxe

```
body : <requête>
```

<requête> est une expression qui retourne un résultat dont le type doit être compatible avec le type du résultat de l'opération désignée par le contexte.

Exemple

context Compte::getSolde() : Real
body : solde

5.2 Définition d'attributs et de méthodes (*def* et *let…in*)

Parfois, une sous-expression est utilisée plusieurs fois dans une expression. *let* permet de déclarer et de définir la valeur (*i.e.* initialiser) d'un attribut qui pourra être utilisé dans l'expression qui suit le *in*.
def est un type de contrainte qui permet de déclarer et de définir la valeur d'attributs comme la séquence *let…in*. *def* permet également de déclarer et de définir la valeur retournée par une opération interne à la contrainte.

Syntaxe de *let…in*

```
let <déclaration> = <requête> in <expression>
```

Un nouvel attribut déclaré dans <déclaration> aura la valeur retournée par l'expression <requête> dans toute l'expression <expression>.
Reportez-vous à la section 4.7 pour un exemple d'utilisation.

Syntaxe de *def*

```
def : <déclaration> = <requête>
```

<déclaration> peut correspondre à la déclaration d'un attribut ou d'une méthode.
<requête> est une expression qui retourne un résultat dont le type doit être compatible avec le type de l'attribut, ou de la méthode, déclaré dans <déclaration>. Dans le cas où il

s'agit d'une méthode, <requête> peut utiliser les paramètres spécifiés dans la déclaration de la méthode.

Exemple

Pour imposer qu'une personne majeure doit avoir de l'argent, on peut écrire indifféremment :

- **context** Personne
  inv : let argent=compte.solde->sum() **in** age>=18 implies argent>0
- **context** Personne
  def : argent : int = compte.solde->sum()
  context Personne
  inv : age>=18 implies argent>0

sum() est décrit section 4.6.2.

6.2 Initialisation (*init*) et évolution des attributs (*derive*)

Le type de contrainte *init* permet de préciser la valeur initiale d'un attribut ou d'une terminaison d'association.
Les diagrammes d'UML définissent parfois des attributs ou des associations dérivées. La valeur de tels éléments est toujours déterminée en fonctions d'autres éléments du diagramme. Le type de contrainte *derive* permet de préciser comment la valeur de ce type d'élément évolue.
Notez bien la différence entre ces deux types de contraintes. La contrainte *derive* impose une contrainte perpétuelle : l'élément dérivé doit toujours avoir la valeur imposé par l'expression de la contrainte *derive*. D'un autre côté, la contrainte *init* ne s'applique qu'au moment de la création d'une instance précisée par le contexte de la contrainte. Ensuite, la valeur de l'élément peut fluctuer indépendamment de la contrainte *init*.

Syntaxe

```
init : <requête>
derive : <requête>
```

Exemple

Quand on crée une personne, la valeur initiale de l'attribut *marié* est faux et la personne ne possède pas d'employeur :

context Personne::marié : Boolean
init : false
context Personne::employeur : Set(Société)
init : Set{}

Les collections (dont *Set* est une instance) sont décrites section 4.4.4. *Set{}* correspond à un ensemble vide.
L'âge d'une personne est la différence entre la date courante et la date de naissance de la personne :

context Personne::age : Integer
derive : date_de_naissance - Date::current()

On suppose ici que le type *Date* possède une méthode de classe permettant de connaître la date courante et que l'opération moins (-) entre deux dates est bien définie et retourne un nombre d'années.

3 Accès aux caractéristiques et aux objets dans les contraintes OCL

Dans une contrainte OCL associée à un objet, il est possible d'accéder aux caractéristiques (attributs, opérations et terminaison d'association) de cet objet, et donc, d'accéder de manière transitive à tous les objets (et leurs caractéristiques) avec qui il est en relation.

3.1 Accès aux attributs et aux opérations (*self*)

Pour faire référence à un attribut ou une opération de l'objet désigné par le contexte, il suffit d'utiliser le nom de cet élément. L'objet désigné par le contexte est également accessible par l'expression *self*. On peut donc également utiliser la notation pointée : self.<propriété>.
Une opération peut avoir des paramètres, il faut alors les préciser entre les parenthèses de l'opération.
Lorsque la multiplicité d'un attribut, de type *T*, n'est pas 1 (donc s'il s'agit d'un tableau), la référence à cet attribut est du type *ensemble* (*i.e. Set*(*T*)).
Par exemple, dans le contexte de la classe *Compte*, on peut utiliser les expressions suivantes :

- solde
- **self.**solde
- getSolde()
- **self.**getSolde()
- débiter(1000)
- **self.**débiter(1000)

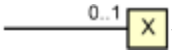
Dans l'exemple précédent, le résultat de l'expression ***self.****débiter(1000)* est un singleton du type *Real*. Mais une opération peut comporter des paramètres définis en sortie ou en entrée/sortie. Dans ce cas, le résultat sera un tuple contenant tous les paramètres définis en sortie ou en entrée/sortie. Par exemple, imaginons une opération dont la déclaration serait *operation(out param_out : Integer):Real* possédant un paramètre défini en sortie *param_out*. Dans ce cas, le résultat de l'expression *operation(paramètre)* est un tuple de la forme *(param_out : Integer, result : Real)*. On peut accéder aux valeurs de ce tuple de la façon suivante :

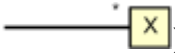
- operation(paramètre).param_out
- operation(paramètre).result

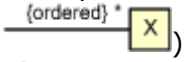
3.2 Navigation via une association

Pour faire référence à un objet, ou un groupe d'objets, en association avec l'objet désigné par le contexte, il suffit d'utiliser le nom de la classe associée (en minuscule) ou le nom du

rôle d'association du coté de cette classe. Quand c'est possible, il est préférable d'utiliser le nom de rôle de l'association du côté de l'objet auquel on désire faire référence. C'est indispensable s'il existe plusieurs associations entre l'objet désigné par le contexte et l'objet auquel on désire accéder, ou si l'association empruntée est réflexive.
Le type du résultat dépend de la propriété structurelle empruntée pour accéder à l'objet référencé, et plus précisément de la multiplicité du côté de l'objet référencé, et du type de l'objet référencé proprement dit. Si on appelle *X* la classe de l'objet référencé, dans le cas d'une multiplicité de :

- 1, le type du résultat est *X* (ex : 

) ;
- * ou 0..*n*, …, le type du résultat est *Set*(*X*) (ex : 

) ;
- * ou 0..*n*, …, et s'il y a en plus une contrainte *{ordered}*, le type du résultat est *OrderedSet*(*X*) (ex : 

).

Emprunter une seule propriété structurelle peut produire un résultat du type *Set* (ou *OrderedSet*). Emprunter plusieurs propriétés structurelles peut produire un résultat du type *Bag* (ou *Sequence*).
Par exemple, dans le contexte de la classe *Société* (**context** Société) :

- directeur désigne le directeur de la société (résultat de type *Personne*) ;
- employé désigne l'ensemble des employés de la société (résultat de type *Set*(*Personne*)) ;
- employé.compte désigne l'ensemble des comptes de tous les employés de la société (résultat de type *Bag*(*Compte*)) ;
- employé.date_de_naissance désigne l'ensemble des dates de naissance des employés de la société (résultat de type *Bag*(*Date*)).

3.3 Navigation via une association qualifiée

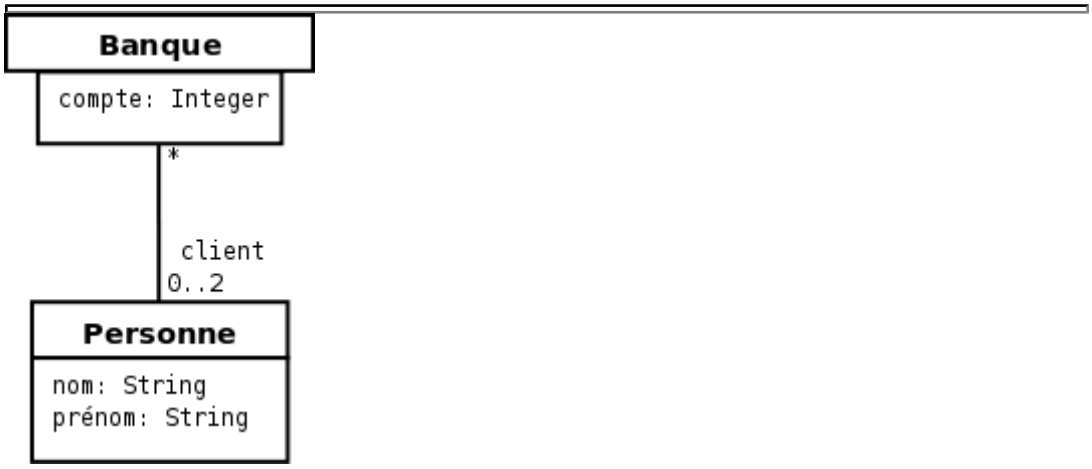


Figure 4.10: Diagramme illustrant une association qualifiée entre une classe *Banque* et une classe *Personne*.

Une association qualifiée (cf. section3.3.6) utilise un ou plusieurs qualificatifs pour sélectionner des instances de la classe cible de l'association. Pour emprunter une telle association, il est possible de spécifier les valeurs, ou les instances, des qualificatifs en utilisant des crochets ([]).

Plaçons-nous dans le cadre du diagramme de la figure 4.10. Dans le contexte de banque (**context** Banque), pour faire référence au nom des clients dont le compte porte le numéro 19503800 il faut écrire :

```
self.client[19503800].nom
```

Dans le cas où il y a plusieurs qualificatifs, il faut séparer chacune des valeurs par une virgule en respectant l'ordre des qualificatifs du diagramme UML. Il n'est pas possible de ne préciser la valeur que de certains qualificatifs en en laissant d'autres non définis. Par contre, il est possible de ne préciser aucune valeur de qualificatif :

```
self.client.nom
```

Dans ce cas, le résultat sera l'ensemble des noms de tous les clients de la banque.
Ainsi, si on ne précise pas la valeur des qualificatifs en empruntant une association qualifiée, tout se passe comme si l'association n'était pas qualifiée. Dans ce cas, faites attention à la cardinalité de la cible qui change quand l'association n'est plus qualifiée (cf. section3.3.6).

3.4 Navigation vers une classe association

Pour naviguer vers une classe association, il faut utiliser la notation pointée classique en précisant le nom de la classe association en minuscule. Par exemple, dans le contexte de la classe *Société* (**context** Société), pour accéder au salaire de tous les employés, il faut écrire :

```
self.poste.salaire
```

Cependant, dans le cas où l'association est réflexive (c'est le cas de la classe association *Mariage*), il faut en plus préciser par quelle extrémité il faut emprunter l'association. Pour cela, on précise le nom de rôle de l'une des extrémités de l'association entre crochets ([]) derrière le nom de la classe association. Par exemple, dans le contexte de la classe *Personne* (**context** Personne), pour accéder à la date de mariage de toutes les femmes, il faut écrire :

```
self.mariage[femme].date
```

4.5 Navigation depuis une classe association

Il est tout-à-fait possible de naviguer directement depuis une classe association vers une classe participante.
Exemple :

```
context Poste
inv : self.employé.age > 21
```

Par définition même d'une classe association, naviguer depuis une classe association vers une classe participante produit toujours comme résultat un objet unique. Par exemple, l'expression *self.employé.age* de l'exemple précédant produit bien un singleton.

5.6 Accéder à une caractéristique redéfinie (*oclAsType()*)

Quand une caractéristique définie dans une classe parente est redéfinie dans une sous-classe associée, la caractéristique de la classe parente reste accessible dans la sous-classe en utilisant l'expression *oclAsType()*.
Supposons une classe *B* héritant d'une classe *A* et une propriété *p1* définie dans les deux classes. Dans le contexte de la classe *B* (**context** B), pour accéder à la propriété *p1* de *B*, on écrit simplement :

```
self.p1
```

et pour accéder à la propriété *p1* de *A* (toujours dans le contexte de *B*), il faut écrire :

```
self.oclAsType(A).p1
```

3.7 Opérations prédéfinies sur tous les objets

L'opération *oclAsType*, que nous venons de décrire (section 4.5.6), est une opération prédéfinie dans le langage OCL qui peut être appliquée à tout objet. Le langage OCL en propose plusieurs :

- oclIsTypeOf (t : OclType) : Boolean
- oclIsKindOf (t : OclType) : Boolean
- oclInState (s : OclState) : Boolean
- oclIsNew () : Boolean
- oclAsType (t : OclType) : instance of OclType

Opération *oclIsTypeOf*

oclIsTypeOf retourne *vrai* si le type de l'objet au titre duquel cette opération est invoqué est exactement le même que le type *t* passé en paramètre.
Par exemple, dans le contexte de *Société*, l'expression *directeur.oclIsTypeOf(Personne)* est vraie tandis que l'expression *self.oclIsTypeOf(Personne)* est fausse.

Opération *oclIsKindOf*

oclIsKindOf permet de déterminer si le type *t* passé en paramètre correspond exactement au type ou à un type parent du type de l'objet au titre duquel cette opération est invoqué.
Par exemple, supposons une classe *B* héritant d'une classe *A* :

- dans le contexte de *B*, l'expression *self.oclIsKindOf(B)* est vraie ;
- toujours dans le contexte de *B*, l'expression *self.oclIsKindOf(A)* est vraie ;
- mais dans le contexte de *A*, l'expression *self.oclIsKindOf(B)* est fausse.

Opération *oclIsNew*

L'opération *oclIsNew* doit être utilisée dans une postcondition. Elle est vraie quand l'objet au titre duquel elle est invoqué est créé pendant l'opération (*i.e.* l'objet n'existait pas au moment des préconditions).

Opération *oclInState*

Cette opération est utilisée dans un diagramme d'états-transitions (cf. section 5). Elle est vraie si l'objet décrit par le diagramme d'états-transitions est dans l'état *s* passé en paramètre. Les valeurs possibles du paramètre *s* sont les noms des états du diagramme d'états-transitions. On peut faire référence à un état imbriqué en utilisant des «::» (par exemple, pour faire référence à un état B imbriqué dans un état A, on écrit : A::B).

3.8 Opération sur les classes

Toutes les opérations que nous avons décrites jusqu'ici s'appliquaient sur des instances de classe. Cependant, OCL permet également d'accéder à des caractéristiques de classe (celles qui sont soulignées dans un diagramme de classes). Pour cela, on utilise le nom qualifié de la classe suivi d'un point puis du nom de la propriété ou de l'opération : *<nom_qualifié>.<propriété>*.
Le langage OCL dispose également d'une opération prédéfinie sur les classes, les interfaces et les énumérations (*allInstances*) qui retourne l'ensemble (*Set*) de toutes les instances du type au titre duquel elle est invoquée, au moment où l'expression est évaluée. Par exemple, pour désigner l'ensemble des instances de la classe personne (type *set*(*Personne*)) on écrit :

```
Personne.allInstances()
```

Opérations sur les collections

4.1 Introduction : «.», «->», «::» et *self*

Comme nous l'avons vu dans la section précédente (4.5), pour accéder aux caractéristiques (attributs, terminaisons d'associations, opérations) d'un objet, OCL utilise la notation pointée : *<objet>.<propriété>*. Cependant, de nombreuses expressions ne produisent pas comme résultat un objet, mais une collection. Le langage OCL propose plusieurs opérations de base sur les collections. Pour accéder ce type d'opération, il faut, utiliser non pas un point mais une flèche : *<collection>->><opération>*. Enfin, rappelons que pour désigner un élément dans un élément englobant on utilise les «::» (cf. section 4.3.2 et 4.5.7 par exemple). En résumé :
«::» – permet de désigner un élément (comme une opération) dans un élément englobant (comme un classeur ou un paquetage) ; **«.» –** permet d'accéder à une caractéristique (attributs, terminaisons d'associations, opérations) d'un objet ; **«->» –** permet d'accéder à une caractéristiques d'une collection.

Nous avons dit dans la section 4.5.1 que l'objet désigné par le contexte est également accessible par l'expression *self*. *self* n'est pas uniquement utilisé pour désigner le contexte d'une contrainte dans une expression mais également pour désigner le contexte d'une sous-expression dans le texte (en langage naturel). Ainsi, lorsque l'on utilise *self* pour une opération <*opération*>, c'est pour désigner l'objet (comme une collection par exemple) sur lequel porte l'opération. Cette objet peut être le résultat d'une opération intermédiaire comme l'évaluation de l'expression <*expression*> précédant l'opération <*opération*> dans l'expression complète : <*expression*>.<*opération*>.

4.2 Opérations de base sur les collections

Nous ne décrirons pas toutes les opérations sur les collections et ses sous-types (ensemble, …) dans cette section. Référez vous à la documentation officielle [19] pour plus d'exhaustivité.

Opérations de base sur les collections

Nous décrivons ici quelques opérations de base sur les collections que propose le langage OCL.
size():Integer – retourne le nombre d'éléments (la cardinalité) de *self*.
includes(objet:T):Boolean – vrai si *self* contient l'objet *objet*.
excludes(objet:T):Boolean – vrai si *self* ne contient pas l'objet *objet*.
count(objet:T):Integer – retourne le nombre d'occurrences de *objet* dans *self*.
includesAll(c:Collection(T)):Boolean – vrai si *self* contient tous les éléments de la collection *c*. **excludesAll(c:Collection(T)):Boolean –** vrai si *self* ne contient aucun élément de la collection *c*. **isEmpty() –** vrai si *self* est vide. **notEmpty() –** vrai si *self* n'est pas vide. **sum():T** retourne la somme des éléments de *self*. Les éléments de *self* doivent supporter l'opérateur *somme* (+) et le type du résultat dépend du type des éléments.
product(c2:Collection(T2)):Set(Tuple(first:T,second:T2)) – le résultat est la collection de Tuple correspondant au produit cartésien de *self* (de type Collection(T)) par *c2*.

Opérations de base sur les ensembles (*Set*)

Nous décrivons ici quelques opérations de base sur les ensembles (type *Set*) que propose le langage OCL.
union(set:Set(T)):Set(T) – retourne l'union de *self* et *set*. **union(bag:Bag(T)):Bag(T) –** retourne l'union de *self* et *bag*. **=(set:Set(T)):Boolean –** vrai si *self* et *set* contiennent les mêmes éléments. **intersection(set:Set(T)):Set(T) –** intersection entre *self* et *set*.
intersection(bag:Bag(T)):Set(T) – intersection entre *self* et *bag*.
[1]**including(objet:T):Set(T) –** Le résultat contient tous les éléments de *self* plus l'objet *objet*. **excluding(objet:T):Set(T) –** Le résultat contient tous les éléments de *self* sans l'objet *objet*. **-(set:Set(T)):Set(T) –** Le résultat contient tous les éléments de *self* sans ceux de *set*. **asOrderedSet():OrderedSet(T) –** permet de convertir *self* du type *Set(T)* en *OrderedSet(T)*. **asSequence():Sequence(T) –** permet de convertir *self* du type *Set(T)* en *Sequence(T)*. **asBag():Bag(T) –** permet de convertir *self* du type *Set(T)* en *Bag(T)*.

Remarque :

les sacs (type *Bag*) disposent d'opérations analogues.

Exemples

1. Une société a au moins un employé :

   context Société **inv :** self.employé->notEmpty()

2. Une société possède exactement un directeur :

   context Société **inv :** self.directeur->size()=1

3. Le directeur est également un employé :

   context Société **inv :** self.employé->includes(self.directeur)

4.3 Opération sur les éléments d'une collection

Syntaxe générale

La syntaxe d'une opération portant sur les éléments d'une collection est la suivante :
`<collection> -> <opération>( <expression> )`
Dans tous les cas, l'expression <*expression*> est évaluée pour chacun des éléments de la collection <*collection*>. L'expression <*expression*> porte sur les caractéristiques des éléments en les citant directement par leur nom. Le résultat dépend de l'opération <*opération*>.
Parfois, dans l'expression <*expression*>, il est préférable de faire référence aux caractéristiques de l'élément courant en utilisant la notation pointée : <*élément*>.<*propriété*>. Pour cela, on doit utiliser la syntaxe suivante :
`<collection> -> <opération>( <élément> | <expression> )`
<*élément*> joue alors un rôle d'itérateur et sert de référence à l'élément courant dans l'expression <*expression*>.
Il est également possible, afin d'être plus explicite, de préciser le type de cet élément :
`<collection> -> <opération>( <élément> : <Type> | <expression> )`
La syntaxe générale d'une opération portant sur les éléments d'une collection est donc la suivante :
`<collection> -> <opération>( [ <élément> [ : <Type> ] | ] <expression> )`

Opération *select* et *reject*

Ces deux opérations permettent de générer une sous-collection en filtrant les éléments de la collection *self*. Leur syntaxe est la suivante :
`select( [ <élément> [ : <Type> ] | ] <expression_logique> )`
`reject( [ <élément> [ : <Type> ] | ] <expression_logique> )`
select permet de générer une sous-collection de *self* ne contenant que des éléments qui satisfont l'expression logique <*expression_logique*>. **reject** permet de générer une sous-collection contenant tous les éléments de *self* excepté ceux qui satisfont l'expression logique <*expression_logique*>.

Par exemple, pour écrire une contrainte imposant que toute société doit posséder, parmi ses employés, au moins une personne de plus de 50 ans, on peut écrire indifféremment :

1. **context** Société
   inv: self.employé->select(age > 50)->notEmpty()
2. **context** Société
   inv: self.employé->select(individu | individu.age > 50)->notEmpty()
3. **context** Société
   inv: self.employé->select(individu : Personne | individu.age > 50)->notEmpty()

Opération *forAll* et *exists*

Ces deux opérations permettent de représenter le quantificateur universel (∀) et le quantificateur existentiel (∃). Le résultat de ces opérations est donc du type *Boolean*. Leur syntaxe est la suivante :

```
forAll( [ <élément> [ : <Type> ] | ] <expression_logique> )
exists( [ <élément> [ : <Type> ] | ] <expression_logique> )
```

forAll permet d'écrire une expression logique vraie si l'expression *<expression_logique>* est vraie pour tous les éléments de *self*. **exists** permet d'écrire une expression logique vraie si l'expression *<expression_logique>* est vraie pour au moins un élément de *self*.
Par exemple, pour écrire une contrainte imposant que toute société doit posséder, parmi ses employés, au moins une personne de plus de 50 ans, on peut écrire :

> **context** Société
> **inv:** self.employé->exists(age > 50)

L'opération *forAll* possède une variante étendue possédant plus d'un itérateur. Dans ce cas, chacun des itérateurs parcourra l'ensemble de la collection. Concrètement, une opération *forAll* comportant deux itérateurs est équivalente à une opération *forAll* n'en comportant qu'un, mais réalisée sur le produit cartésien de *self* par lui-même.
Par exemple, imposer qu'il n'existe pas deux instances de la classe *Personne* pour lesquelles l'attribut *nom* a la même valeur, c'est à dire pour imposer que deux personnes différentes ont un nom différent, on peut écrire indifféremment :

1. **context** Personne
   inv: Personne.allInstances()->forAll(p1, p2 | p1 <> p2 implies p1.nom <> p2.nom)
2. **context** Personne
   inv: (Personne.allInstances().product(Personne.allInstances()))
   ->forAll(tuple | tuple.first <> tuple.second implies tuple.first.nom <>
   tuple.second.nom)

Opération *collect*

Cette opération permet de construire une nouvelle collection en utilisant la collection *self*. La nouvelle collection construite possède le même nombre d'éléments que la collection *self*, mais le type de ces éléments est généralement différent. La syntaxe de l'opérateur *collect* est la suivante :

```
collect( [ <élément> [ : <Type> ] | ] <expression> )
```

Pour chaque élément de la collection *self*, l'opérateur *collect* évalue l'expression *<expression>* sur cet élément et ajoute le résultat dans la collection générée.
Par exemple, pour définir la collection des dates de naissance des employés d'une société, il faut écrire, dans le contexte de la classe *Société* :

```
self.employé->collect(date_de_naissance)
```

Puisque, toujours dans le contexte de la classe *Société*, l'expression *self.employé->collect(date_de_naissance)->size() = self.employé->size()* est toujours vraie, il faut en conclure que le résultat d'une opération *collect* sur une collection du type *Set* n'est pas du type *Set* mais du type *Bag*. En effet, dans le cadre de notre exemple, il y aura certainement des doublons dans les dates de naissance.

4.4 Règles de précédence des opérateurs

Ordre de précédence pour les opérateurs par ordre de priorité décroissante :

1. *@pre*
2. «.» et «->»
3. *not* et «-» (opérateur unaire)
4. «*» et «/»
5. «+» et «-»(opérateur binaire)
6. *if-then-else-endif*
7. «<», «>», «<=» et «>=»
8. «=» et «<>»
9. *and*, *or* et *xor*
10. *implies*

Les parenthèses, «(» et «)», permettent de changer cet ordre.

5 Exemples de contraintes

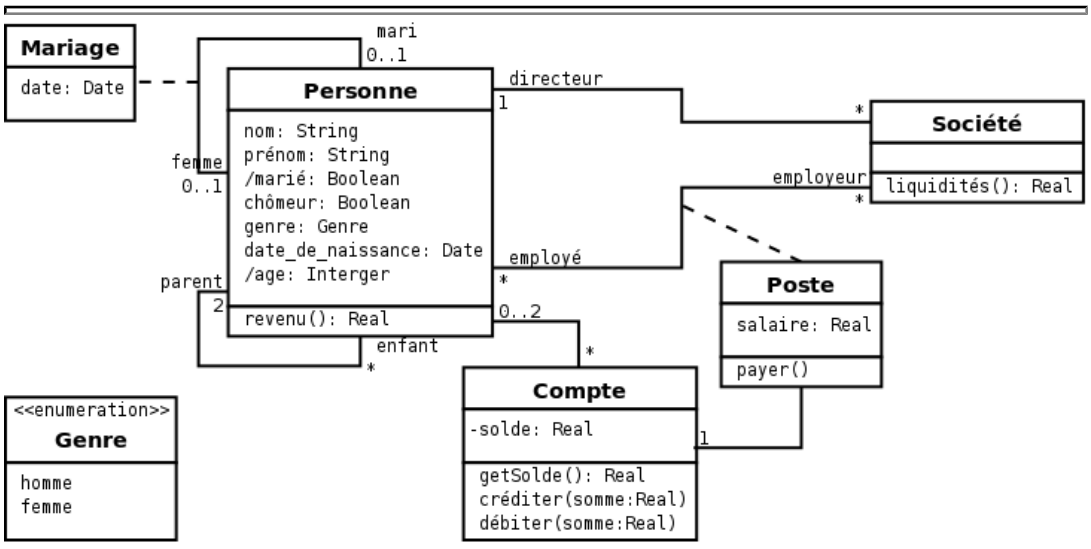


Figure 4.11: Reprise du diagramme de la figure 4.8.

Dans cette section, nous allons illustrer par quelques exemples l'utilisation du langage OCL. Nous restons toujours sur le diagramme de classes de la figure 4.8 représenté à nouveau sur la figure 4.11 pour des raisons de proximité.

1. Dans une société, le directeur est un employé, n'est pas un chômeur et doit avoir plus de 40 ans. De plus, une société possède exactement un directeur et au moins un employé.

```
context Société
inv :
self.directeur->size()=1 and
not(self.directeur.chômeur) and
self.directeur.age > 40 and
self.employé->includes(self.directeur)
```

2. Une personne considérée comme au chômage ne doit pas avoir des revenus supérieurs à 100 €.

```
context Personne
inv :
let revenus : Real = self.poste.salaire->sum() in
if chômeur then
revenus < 100
else
revenus >= 100
endif
```

3. Une personne possède au plus 2 parents (référencés).

```
context Personne
inv : parent->size()<=2
```

4. Si une personne possède deux parents, l'un est une femme et l'autre un homme.

```
context Personne
inv :
parent->size()=2 implies
( parent->exists(genre=Genre::homme) and
parent->exists(genre=Genre::femme) )
```

5. Tous les enfants d'une personne ont bien cette personne comme parent et inversement.

```
context Personne
inv :
enfant->notEmpty() implies
```

```
enfant->forAll( p : Personne | p.parents->includes(self))

context Personne
inv :
parent->notEmpty() implies
parent->forAll ( p : Personne | p.enfant->includes (self))
```

6. Pour être marié, il faut avoir une femme ou un mari.

```
context Personne::marié
derive : self.femme->notEmpty() or self.mari->notEmpty()
```

7. Pour être marié, il faut avoir plus de 18 ans. Un homme est marié avec exactement une femme et une femme avec exactement un homme.

```
context Personne
inv :
self.marié implies
self.genre=Genre::homme implies (
self.femme->size()=1 and
self.femme.genre=Genre::femme)
and self.genre=Genre::femme implies (
self.mari->size()=1 and
self.mari.genre=Genre::homme)
and self.age >=18
```