Object Constraint Language (OCL)

Eric Cariou

Université de Bretagne Occidentale
UFR Sciences & Techniques – Département Informatique

Eric.Cariou@univ-brest.fr

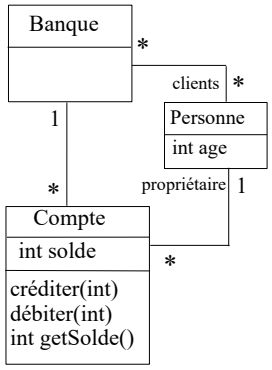
Plan

1. Pourquoi OCL ? Introduction par l'exemple
2. Les principaux concepts d'OCL
3. Exemple d'application sur un autre modèle

Exemple d'application

- Application bancaire
  - Des comptes bancaires
  - Des clients
  - Des banques
- Spécification
  - Un compte doit avoir un solde toujours positif
  - Un client peut posséder plusieurs comptes
  - Un client peut être client de plusieurs banques
  - Un client d'une banque possède au moins un compte dans cette banque
  - Une banque gère plusieurs comptes
  - Une banque possède plusieurs clients

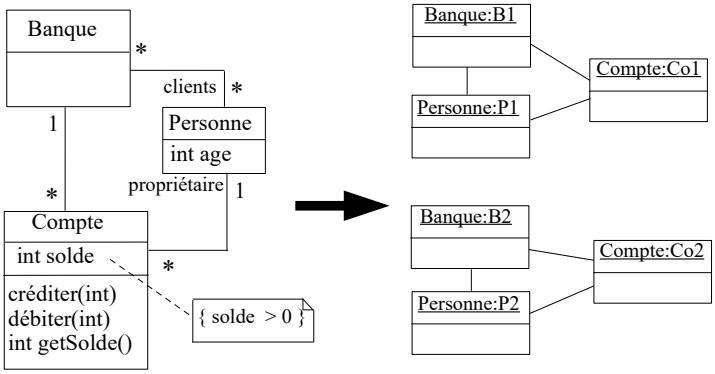
Diagramme de classe



Manque de précision

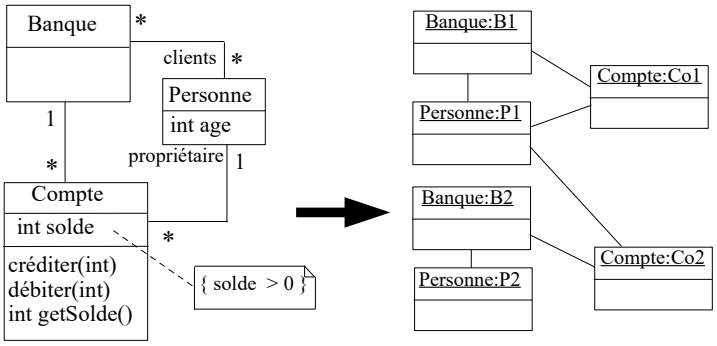
- Le diagramme de classe ne permet pas d'exprimer tout ce qui est défini dans la spécification informelle
- Exemple
  - Le solde d'un compte doit toujours être positif : ajout d'une contrainte sur cet attribut
- Le diagramme de classe permet-il de détailler toutes les contraintes sur les relations entre les classes ?

Diagramme d'instances



- Diagramme d'instances valide vis-à-vis du diagramme de classe et de la spécification attendue

Diagramme d'instances



- Diagramme d'instances valide vis-à-vis du diagramme de classe mais ne respecte pas la spécification attendue
  - Une personne a un compte dans une banque où elle n'est pas cliente
  - Une personne est cliente d'une banque mais sans y avoir de compte

Diagrammes UML insuffisants

- Pour spécifier complètement une application
  - Diagrammes UML seuls sont généralement insuffisants
  - Nécessité de rajouter des contraintes
- Comment exprimer ces contraintes ?
  - Langue naturelle mais manque de précision, compréhension pouvant être ambiguë
  - Langage formel avec sémantique précise : par exemple OCL
- OCL : Object Constraint Language
  - Langage de contraintes orienté-objet
  - Langage formel (mais « simple » à utiliser) avec une syntaxe, une grammaire, une sémantique (manipulable par un outil)
  - S'applique entre autres sur les diagrammes UML

Plan

1. Pourquoi OCL ? Introduction par l'exemple
2. *Les principaux concepts d'OCL*
3. Exemple d'application sur un autre modèle

Le langage OCL

- Norme de l'OMG
  - Version courante : 2.4 (2014)
  - https://www.omg.org/spec/OCL
  - Peut s'appliquer sur tout type de modèle, indépendant d'un langage de modélisation donné
- OCL permet principalement d'exprimer deux types de contraintes sur l'état d'un ou plusieurs objets
  - Des invariants qui doivent être respectés en permanence
  - Des pré et post-conditions pour une opération
    - Précondition : doit être vérifiée avant l'exécution
    - Postcondition : doit être vérifiée après l'exécution
- Attention
  - Une expression OCL décrit une contrainte à respecter et non pas le « code » d'une méthode

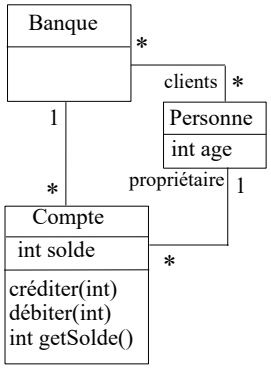
Utilisation d'OCL dans le cadre d'UML

- OCL peut s'appliquer sur la plupart des diagrammes UML
- Il sert, entre autres, à spécifier des
  - Invariants sur des classes
  - Pré et postconditions sur des opérations
  - Gardes sur transitions de diagrammes d'états ou de messages de diagrammes de séquence/collaboration
  - Des ensembles d'objets destinataires pour un envoi de message
  - Des attributs dérivés
  - Des stéréotypes
  - ...

Utilisation d'OCL en IDM

- Définition de méta-modèles en Ecore/EMF
  - *Well-formedness rules*
    - Règles de bonne formation définies par des invariants OCL pour compléter la structure du méta-modèle
  - Permet d'assurer que les modèles sont bien formés
- Utilisé dans beaucoup d'outils IDM
  - Permet de faire des requêtes pour filtrer et récupérer des éléments d'un modèle
  - Dans le cadre de l'UE « programmation générative »
    - Langage de transformation ATL
    - Génération de code avec Acceleo

Exemple OCL sur l'application bancaire



context Compte
inv: solde > 0

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

context Compte
inv: banque.clients -> includes (propriétaire)

- On rajoute les invariants et les pré/post-conditions spécifiant les contraintes non exprimables par le diagramme de classe seul

Contexte

- Une expression OCL est toujours définie dans un contexte
  - Ce contexte est une classe
- Mot-clé : **context**
- Exemple
  - **context** Compte
  - L'expression OCL s'applique à la classe Compte, c'est-à-dire à toutes les instances de cette classe

Invariants

- Un invariant exprime une contrainte sur un objet ou un groupe d'objets qui doit être respectée en permanence
- Mot-clé : **inv**
- Exemple
  - **context** Compte
    inv: solde > 0
  - Pour toutes les instances de la classe Compte, l'attribut solde doit toujours être positif

Pré et postconditions

- Pour spécifier une opération
  - Précondition : état qui doit être respecté avant l'appel de l'opération
  - Postcondition : état qui doit être respecté après l'appel de l'opération
  - Mots-clés : **pre** et **post**
- Dans la postcondition, deux éléments particuliers sont utilisables
  - Pseudo-attribut **result** : référence la valeur retournée par l'opération
  - mon_attribut**@pre** : référence la valeur de *mon_attribut* avant l'appel de l'opération
- Syntaxe pour préciser la signature de l'opération
  - context ma_classe::mon_op(liste_param) : type_retour

Pré et postconditions

- Exemples
  - **context** Compte::débiter(somme : Integer)
    pre: somme > 0
    post: solde = solde@pre – somme
    - La somme à débiter doit être positive pour que l'appel de l'opération soit valide
    - Après l'exécution de l'opération, l'attribut solde *doit* avoir pour valeur sa valeur avant l'appel à laquelle a été soustrait la somme passée en paramètre
  - **context** Compte::getSolde() : Integer
    post: result = solde
    - Le résultat retourné doit être le solde courant
- Attention
  - On ne décrit pas comment l'opération est réalisée mais des contraintes sur l'état avant et après son exécution

Conception par contrat pour opérations

- Pre et postconditions permettent de faire une conception par contrat
  - Contrat passé entre l'appelant d'une opération et l'appelé (celui qui exécute l'opération)
  - Si l'appelant respecte les contraintes de la précondition alors l'appelé s'engage à respecter la post-condition
    - Si l'appelant ne respecte pas la précondition, alors le résultat de l'appel est indéfini
  - Pour exemple précédent
    - Si l'appelant de l'opération *débiter* passe une somme positive en paramètre, alors le compte est bien débité de cette somme

Accès aux objets, navigation

- Dans une contrainte OCL associée à un objet, on peut
  - Accéder à l'état interne de cet objet (ses attributs)
  - Naviguer dans le diagramme : accéder de manière transitive à tous les objets (et leur état) avec qui il est en relation
- Nommage des éléments pour y accéder
  - Attributs ou paramètres d'une opération : utilise leur nom directement
  - Objet(s) en association : on utilise au choix
    - Le nom de la classe associée (avec la première lettre en minuscule)
    - Le nom de l'association si elle nommée
    - Le nom du rôle d'association du coté de la classe vers laquelle on navigue s'il est nommé
- La navigation retourne
  - Si cardinalité de 1 pour une association : un objet
  - Si cardinalité > 1 : une collection d'objets

Accès aux objets, navigation

- Pseudo-attribut particulier
  - self : référence l'objet de départ, d'où part la navigation
- Exemples, dans contexte de la classe Compte
  - solde : attribut référencé directement
  - banque : objet de la classe Banque (référence via le nom de la classe) associé au compte
  - propriétaire : objet de la classe Personne (référence via le nom de rôle d'association) associée au compte
  - banque.clients : ensemble des clients de la banque associée au compte (référence par transitivité)
  - banque.clients.age : ensemble des âges de tous les clients de la banque associée au compte
- Le propriétaire d'un compte doit avoir plus de 18 ans
  - **context** Compte
    inv: self.propriétaire.age >= 18

Opérations sur objets et collections

- OCL propose un ensemble de primitives utilisables sur les collections
  - size() : retourne le nombre d'éléments de la collection
  - isEmpty() : retourne vrai si la collection est vide
  - notEmpty() : retourne vrai si la collection n'est pas vide
  - count(obj) : le nombre d'occurrences de l'objet obj dans la collection
  - includes(obj) : vrai si la collection inclut l'objet *obj*
  - excludes(obj) : vrai si la collection n'inclut pas l'objet *obj*
  - including(obj) : la collection référencée doit être cette collection en incluant l'objet *obj*
  - excluding(obj) : idem mais en excluant l'objet *obj*
  - includesAll(col) : la collection contient tous les éléments de la collection *col*
  - excludesAll(col) : la collection ne contient aucun des éléments de la collection *col*
- Syntaxe d'utilisation : *objetOuCollection -> primitive*

Opérations sur objets et collections

- Exemples, invariants dans le contexte de la classe Compte
  - propriétaire -> size() = 1 : le nombre d'objets Personne associés à un compte est de 1
    - Vrai par principe à cause de la cardinalité de 1 qui doit être respectée
    - On manipule ici un objet (cardinalité de 1) comme une collection contenant l'objet
  - banque.clients -> size() >= 1 : une banque a au moins un client
  - banque.clients -> includes(self.propriétaire) : l'ensemble des clients de la banque associée au compte contient le propriétaire du compte
  - banque.clients.compte -> includes(self) : le compte appartient à un des clients de sa banque

Opérations sur objets et collections

- Autre exemple
  - Un nouveau compte est créé pour une personne. La banque doit gérer ce nouveau compte. Le client passé en paramètre doit posséder ce compte. Le nouveau compte est retourné par l'opération.
  - **context** Banque::creerCompte(p : Personne) : Compte
    post: result.oclIsNew() **and**
    compte = compte@pre -> including(result) **and**
    p.compte = p.compte@pre -> including(result)
- Il n'est pas utile de rajouter les contraintes

  result.proprietaire = p **and** result.banque = self
  - En effet, les associations ici sont bidirectionnelles
    - Si un compte appartient à un ensemble de comptes d'une banque, ce compte est associé par principe à cette banque
    - Si un compte appartient à un ensemble de comptes d'une personne, ce compte a par principe cette personne comme propriétaire

Opérations sur objets et collections

- oclIsNew()
  - Primitive indiquant qu'un objet doit être créé pendant l'appel de l'opération
  - Ne peut être utilisé que dans une postcondition
- and
  - « et logique » : l'invariant, pré ou postcondition est vrai si toutes les expressions reliées par le « and » sont vraies
  - Il existe en OCL les autres opérateurs logiques classiques que l'on combine comme on veut : or, not, xor
  - Possibilité de parenthèser pour changer les priorités ou éviter des ambiguïtés

Relations ensemblistes entre deux collections

- union : l'union des deux collections
- intersection : l'intersection des deux collections
- - : la collection en y retirant les éléments qui se trouvaient aussi dans l'autre collection
- symmetricDifference : la collection qui contient les éléments n'existant que dans une des deux collections
- Exemples
  - (col1 -> intersection(col2)) -> isEmpty()
    - Renvoie vrai si les collections *col1* et *col2* n'ont pas d'élément en commun
  - col1 = col2 -> union(col3)
    - La collection *col1* doit être l'union des éléments de *col2* et de *col3*

Filtre de collections

- A partir d'une collection, on peut en récupérer une sous-partie ou une autre collection
- Primitives offrant ces services et s'appliquant sur une collection *col*
  - select : retourne le sous-ensemble de la collection *col* dont les éléments respectent la contrainte spécifiée
  - reject : idem mais ne garde que les éléments ne respectant pas la contrainte
  - collect : retourne une collection (de taille identique) construite à partir des éléments de *col*. Le type des éléments contenus dans la nouvelle collection peut être différent de celui des éléments de col.
  - collectNested : idem que collect sauf qu'en cas de collections imbriquées retournées, collect fait une mise à plat et pas collectNested
  - sortedBy : retourne la même collection mais avec les éléments triés selon l'expression passée en paramètre
  - any : retourne un objet de la collection qui respecte l'expression passée en paramètre

Contraintes sur éléments d'une collection

- OCL permet de vérifier des contraintes sur les éléments d'une collection
- Primitives offrant ces services et s'appliquant sur une collection *col*
  - exists : retourne vrai si au moins un élément de *col* respecte la contrainte spécifiée et faux sinon
  - forAll : retourne vrai si tous les éléments de *col* respectent la contrainte spécifiée (pouvant impliquer à la fois plusieurs éléments de la collection)
  - one : retourne vrai si un et un seul des éléments de *col* respecte la contrainte spécifiée
  - isUnique : réalise un collect puis retourne vrai si tous les éléments de la nouvelle collection sont différents

Opérations sur éléments d'une collection

- Syntaxe des opérations citées : 3 usages
  - collection -> primitive(expression)
    - La primitive s'applique aux éléments de la collection et pour chacun d'entre eux, l'expression *expression* est vérifiée. On accède implicitement aux attributs/relations d'un élément.
  - collection -> primitive(elt : type | expression)
    - On fait explicitement apparaître le type des éléments de la collection (ici *type*). On accède aux attributs/relations de l'élément courant en utilisant *elt* (c'est la référence sur l'élément courant)
  - collection -> primitive(elt | expression)
    - On nomme l'attribut courant (*elt*) mais sans préciser son type

Opérations sur éléments d'une collection

- Dans le contexte de la classe Banque
  - compte -> select(c | c.solde > 1000)
    - Retourne une collection contenant tous les comptes bancaires dont le solde est supérieur à 1000 €
  - compte -> reject(solde > 1000)
    - Retourne une collection contenant tous les comptes bancaires dont le solde n'est pas supérieur à 1000 €
  - compte -> collect(c : Compte | c.solde)
    - Retourne une collection contenant l'ensemble des soldes de tous les comptes
  - compte -> select(solde > 1000) -> collect(c | c.solde)
    - Retourne une collection contenant tous les soldes des comptes dont le solde est supérieur à 1000 €

Opérations sur éléments d'une collection

- **context** Banque
  inv: **not** (clients -> exists (age < 18))
  - Il n'existe pas de clients de la banque dont l'age est inférieur à 18 ans
  - Peut aussi s'écrire :
    context Banque **inv**: clients -> forAll (c | c.age >= 18)
- **context** Personne **inv**:
  Personne.allInstances() -> forAll(p1, p2 |
  p1 <> p2 **implies** p1.nom <> p2.nom)
  - Le forAll à deux variables va former tous les combinaisons possibles de 2 éléments de la collection et vérifier la contrainte pour chaque couple
    - Deux personnes différentes ont un nom différent
  - allInstances()
    - Primitive s'appliquant sur une classe (et non pas un objet) et retournant toutes les instances de la classe référencée (ici la classe Personne)

Types OCL : types de base

- Types de base et exemples d'opérations associées
  - Integer
    - 1, -2, 145
    - *, +, -, /, abs()
  - Real
    - 1.5, -123.4
    - *, +, -, /, floor()
  - String
    - 'bonjour'
    - concat(), size(), substring()
  - Boolean
    - true, false
    - and, or, not, xor, implies, if-then-else
    - La plupart des expressions OCL sont de types Boolean
    - Notamment les expressions formant les inv, pre et post

Types OCL : types de collection

- 4 types de collections
  - Set : ensemble au sens mathématique, pas de doublons, pas d'ordre
  - OrderedSet : idem mais avec ordre (les éléments ont une position dans l'ensemble)
  - Bag : comme un Set mais avec possibilité de doublons
  - Sequence : un Bag dont les éléments sont ordonnés
- Exemples :
  - { 1, 4, 3, 5 } : Set(Integer)
  - { 1, 4, 1, 3, 5, 4 } : Bag(Integer)
- Notes
  - Un collect renvoie un Bag, un sortebBy un OrderedSet
  - Possibilité de transformer un type de collection en un autre type de collection avec opérations OCL dédiées

Opérations sur collections ordonnées

- first() : le premier élément de la collection
- last() : le dernier élément de la collection
- at(index : Integer) : l'élément de la collection se trouvant en position *index*
- indexOf(elt) : la position de l'élément *elt* dans la collection
- append(elt) : la collection augmentée de l'élément *elt* placé à la fin
- prepend(elt) : la collection augmentée de l'élément *elt* placé au début
- insertAt(index : Integer, elt) : la collection augmentée de l'élément *elt* placé à la position index
- subOrderedSet(lower : Integer, upper : Integer) : l'ordered set contenant les éléments de la position *lower* à *upper* à partir d'un ordered set
- subSequence(lower : Integer, upper : Integer) : la séquence contenant les éléments de la position *lower* à *upper* à partir d'une séquence
- reverse() : la même collection mais avec les éléments inversés en position

Types OCL : types de collection

- Collections imbriquées
  - Via navigation, on peut récupérer des collections ayant pour éléments d'autres collections
  - Deux modes de manipulation
    - Explicitement comme une collection de collections [de collections ...]
    - Collection unique : on « aplatit » le contenu de toutes les collections imbriquées en une seule à un seul niveau
      - Opération flatten() pour aplatir une collection de collections
  - Tuples/n-uplet
    - Données contenant plusieurs champs
      - Ex: Tuple { nom:String = 'toto', age:Integer = 21 }
    - Peut manipuler ce type de données en OCL

Types OCL : conformance de types

- Conformance de type
  - Prise en compte des spécialisations entre classes du modèle
  - Opérations OCL dédiées à la gestion des types
    - oclIsTypeOf(type) : vrai si l'objet est du type *type*
    - oclIsKindOf(type) : vrai si l'objet est du type *type* ou un de ses sous-types
    - oclAsType(type) : l'objet est « casté » en type *type*
- Types internes à OCL
  - Conformance entre les types de collection
    - Collection est le super-type de Set, Bag et Sequence
    - Conformance entre collection et types des objets contenus
      - Set(T1) est conforme à Collection(T2) si T1 est sous-type de T2 ...
    - Integer est un sous-type de Real

Conditionnelles

- Certaines contraintes sont dépendantes d'autres contraintes
- Deux formes pour gérer cela
  - **if** expr1 **then** expr2 **else** expr3 **endif**
    - Si l'expression *expr1* est vraie alors *expr2* doit être vraie sinon *expr3* doit être vraie
  - expr1 **implies** expr2
    - Si l'expression *expr1* est vraie, alors *expr2* doit être vraie également.
    - Si *expr1* est fausse, alors l'expression complète est vraie
  - Il n'existe pas de if ... then sans la branche else
    - Il faut utiliser le implies pour cela

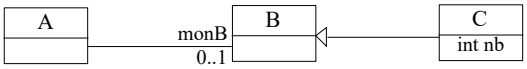
Conditionnelles

- **context** Personne **inv**:
  if age < 18
  then compte -> isEmpty()
  else compte -> notEmpty()
  endif
  - Une personne de moins de 18 ans n'a pas de compte bancaire alors qu'une personne de plus de 18 ans possède au moins un compte

- **context** Personne **inv**:
  compte -> notEmpty() **implies** banque -> notEmpty()
  - Si une personne possède au moins un compte bancaire, alors elle est cliente d'au moins une banque

Associations en cardinalité 0..1

- Avec une cardinalité 0..1, il y a soit au bout de l'association un objet ou rien
  - oclIsUndefined() : retourne vrai si l'objet n'existe pas, faux sinon
  - oclIsDefined() n'existe pas !
- Exemple
  - Si un objet de type *A* a sa référence vers un objet de type *B* positionnée, on vérifie si ce *B* est un *C* que son attribut *nb* est positif



context A **inv**:
not self.monB.oclIsUndefined() **implies**
(self.monB.oclIsTypeOf(C) **implies**
self.monB.oclAsType(C).nb > 0)

Commentaires et nommage de contraintes

- Commentaire en OCL : utilisation de --
  - Exemple

```
context Personne inv:
if age < 18                    -- vérifie age de la personne
then compte -> isEmpty()       -- pas majeur : pas de compte
else compte -> notEmpty()      -- majeur : doit avoir au moins un compte
endif
```

- On peut nommer des contraintes
  - Exemple
    - **context** Compte
      inv soldePositif: solde > 0
    - **context** Compte::débiter(somme : Integer)
      pre sommePositive: somme > 0
      post sommeDébitée: solde = solde@pre - somme

Variables

- Pour faciliter l'utilisation de certains attributs ou calculs de valeurs on peut définir des variables
- Dans une contrainte OCL : let ... in ...
  - **context** Personne
    inv: **let** argent = compte.solde -> sum() **in**
    age >= 18 **implies** argent > 0
    - Une personne majeure doit avoir de l'argent
  - sum() : fait la somme de tous les objets de la collection
- Pour l'utiliser partout : def
  - **context** Personne
    def: argent : Integer = compte.solde -> sum()
  - **context** Personne
    inv: age >= 18 **implies** self.argent > 0

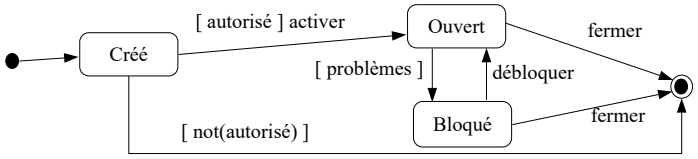
Définition de fonctions de type query

- Opérateur « def » permet également de définir des fonctions d'interrogation du modèle (query)
  - Avec paramètres et type de retour au besoin
  - Pour faciliter la navigation et la sélection d'éléments sur le modèle
- Exemple: les comptes dont le solde est supérieur à une certaine somme *val*
  - **context** Banque **def**: soldesSup(val : Integer) : Set(Compte) = self.comptes -> select (c | c.solde > val)
  - S'utilise ensuite comme une opération de la classe Banque dont on se sert pour écrire une contrainte
  - **context** Banque
    inv: self.soldesSup(1000) -> notEmpty()

Gestion attributs/opérations d'une classe

- Peut utiliser une opération d'une classe dans une contrainte
  - Si pas d'effets de bords (de type « query »)
    - Car une contrainte OCL exprime une contrainte sur un état mais ne précise pas qu'une action a été effectuée
  - Exemple
    - **context** Banque
      inv: compte -> forAll(c | c.getSolde() > 0)
    - getSolde() est une opération de la classe Compte. Elle calcule une valeur mais sans modifier l'état d'un compte
    - On définit alors en OCL la valeur retournée par cette opération
      context Banque::getSolde() : Integer
      body: self.solde
- Pour les attributs
  - Peut définir leur valeur initiale (init) ou un attribut dérivé (derive)
  - Exemple : 50€ offerts par la banque à la création d'un compte
    context Compte::solde
    init: 50

Liens avec diagrammes d'états



- Possibilité de référencer un état d'un diagramme d'états associé à l'objet
  - oclInState(etat) : vrai si l'objet est dans l'état *etat*
  - Pour sous-états : *etat1::etat2* si *etat2* est un état interne de *etat1*
- Exemple
  - **context** Compte::débiter(somme : Integer)
    pre: somme > 0 **and** self.oclInState(Ouvert)
  - L'opération débiter ne peut être appelée que si le compte est dans l'état ouvert

Liens avec diagrammes d'états

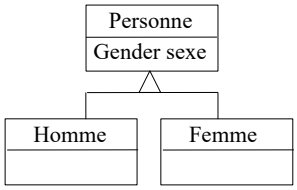
- On ne peut pas avoir plus de 5 comptes ouverts dans une même banque

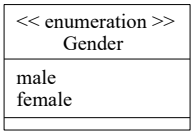
context Compte::activer()
pre: self.oclInState(Créé) **and**
proprietaire.compte -> select(c |
self.banque = c.banque) -> size() < 5
post: self.oclInState(Ouvert)

- On peut aussi exprimer la garde [autorisé] en OCL
  - **context** Compte
    def: autorisé : Boolean = proprietaire.compte -> select(c |
    self.banque = c.banque) -> size() < 5

Énumération

- Utilisation d'une valeur d'une énumération
  - *NomEnum::valeur*
  - Ancienne notation : *#valeur*





  - **context** Homme
    inv: sexe = Gender::male
  - **context** Femme
    inv: sexe = #female

Propriétés

- De manière générale en OCL, une propriété est un élément pouvant être
  - Un attribut
  - Un bout d'association
  - Une opération ou méthode de type requête
- Accède à la propriété d'un objet avec « . »
  - Exemples
    - **context** Compte **inv**: self.solde > 0
    - **context** Compte **inv**: self.getSolde() > 0
- Accède à la propriété d'une collection avec « -> »
  - On peut utiliser « -> » également dans le cas d'un objet (= collection d'1 objet)

Accès aux attributs pour les collections

- Accès à un attribut sur une collection
  - Exemple dans contexte de Banque : compte.solde
  - Renvoie l'ensemble des soldes de tous les comptes
- Forme raccourcie et simplifiée de
  - compte -> collect (solde)

Propriétés prédéfinies en OCL

- Pour objets
  - oclIsTypeOf(type) : l'objet est du type *type*
  - oclIsKindOf(type) : l'objet est du type *type* ou un de ses sous-types
  - oclInState(etat) : l'objet est dans l'état état
  - oclIsNew() : l'objet est créé pendant l'opération
  - oclAsType(type) : l'objet est « casté » en type *type*
  - oclIsUndefined() : la propriété (association par exemple) n'a pas été initialisée (équivalent d'un « null »)
- Pour collections
  - isEmpty(), notEmpty(), size(), sum()
  - includes(), excludes(), includingAll() ...
  - ...

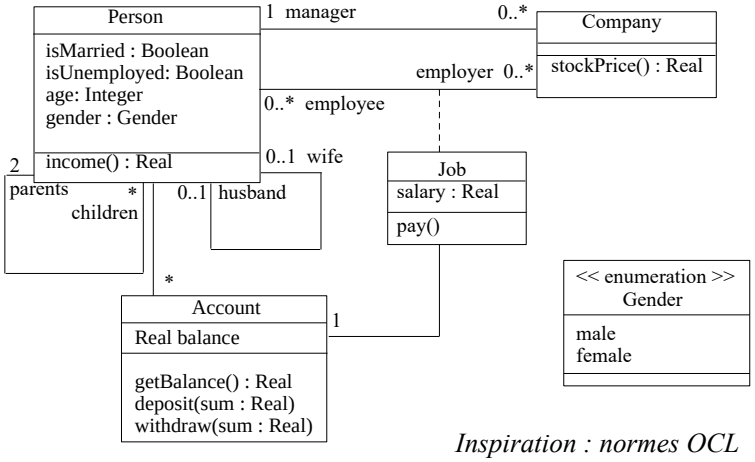
Règles de précédence

- Ordre de précédence pour les opérateurs/primitives du plus au moins prioritaire
  - @pre
  - . et ->
  - not et -
  - * et /
  - + et -
  - if then else endif
  - >, <, <= et >=
  - = et <>
  - and, or et xor
  - implies
- Parenthèses permettent de changer cet ordre

Plan

1. Pourquoi OCL ? Introduction par l'exemple
2. Les principaux concepts d'OCL
3. *Exemple d'application sur un autre modèle*

Diagramme de classe



Inspiration : normes OCL

Contraintes sur employés d'une compagnie

- Dans une compagnie, un manager doit travailler et avoir plus de 40 ans
- Le nombre d'employé d'une compagnie est non nul

context Company
inv:
self.manager.isUnemployed = false **and**
self.manager.age > 40 **and**
self.employee -> notEmpty()

Lien salaire/chomage pour une personne

- Une personne considérée comme au chômage ne doit pas avoir des revenus supérieurs à 500 €

context Person **inv**:
let money : Real = self.job.salary->sum() **in**
if isUnemployed **then**
money < 500
else
money >= 500
endif

Embauche d'un nouvel employé

- Un employé qui est embauché n'appartenait pas déjà à la compagnie

context Company::hireEmployee(p : Person)
post:
employee = employee@pre -> including(p) **and**
employee@pre -> excludes(p) **and**
stockPrice() = stockPrice()@pre + 10

- Équivalent (ou presque) à

context Company::hireEmployee(p : Person)
pre: employee -> excludes(p)
post:
employee -> includes(p) **and**
stockPrice() = stockPrice()@pre + 10

Revenus selon l'age

- Selon l'age de la personne, ses revenus sont
  - 1% des revenus des parents quand elle est mineure (argent de poche)
  - Ses salaires quand elle est majeure

```
context Person::income() : Real
body:
if age < 18 then
   (parents.job.salary -> sum()) * 1%
else
   self.job.salary -> sum()
endif
```


Versement salaire

- Salaire payé :
  context Job::pay()
  post: account.balance = account.balance@pre + salary
- Depuis OCL 2.0 : peut aussi préciser que l'opération *deposit* doit être appelée
  - **context** Job::pay()
    post: account^deposit(salary)
  - *objet^operation(param1, ...)* : renvoie vrai si un message *operation* est envoyé à *objet* avec la liste de paramètres précisée (si pas de valeur particulière : utilise « ? : type »)
  - Note
    - On s'éloigne des principes d'OCL (langage de contraintes et pas d'actions) et généralement exprimable en UML avec diagrammes d'interactions (séquence, collaboration)
    - Néanmoins, permet ici de retrouver le principe d'encapsulation en objet : l'attribut *balance* d'un compte n'est normalement pas accessible directement

Contraintes sur les parents/enfants

- Un enfant a un père et une mère biologiques

```
context Person
def: parent1 = parents -> asSequence() -> at(0)
def: parent2 = parents -> asSequence() -> at(1)

context Person
inv:
if parent1.gender = #male
then          -- parent1 est un homme
   parent2.gender = #female
else          -- parent1 est une femme
   parent2.gender = #male
endif
```


Contraintes de mariage

- Mariage hétérosexuel seulement ici : on vérifie qu'un homme a une épouse de sexe féminin et inversement
- Pour être marié, il faut avoir plus de 18 ans

```
context Person def: conjoint : Person =
if self.husband.oclIsUndefined()
then self.wife
else self.husband
endif

context Person inv:
(self.isMarried implies self.age >= 18 and not self.conjoint.oclIsUndefined())
and (not self.wife.oclIsUndefined() implies
   self.wife.gender = #female and
   self.gender = #male and
   self.wife.isMarried and
   self.wife.husband = self) )
and (not self.husband.oclIsUndefined() implies
   self.husband.gender = #male and
   self.gender = #female and
   self.husband.isMarried and
   self.husband.wife = self )
```


Contraintes de descendance

- On ne peut pas se retrouver dans ses descendants (enfants, petits-enfants, arrière-petits-enfants …)
- Utilise la fermeture transitive : crée une collection en suivant récursivement une association, ici, celle des enfants
  context Person **inv**:
  self -> closure(children) -> excludes(self)
- Autre façon de faire : écrire une fonction OCL récursive

```
context Person def: pasDansDescendants(p : Person) : Boolean =
if self.children -> isEmpty() then true
else
   if self.children -> includes(p) then false
   else self.children -> forAll ( c | c.pasDansDescendants(p) )
   endif
endif

context Person inv:
self.pasDansDescendants(self)
```


Contraintes de descendance

- Les enfants sont plus jeunes que leurs parents
  context Person **inv**:
  self.children -> forAll(c | c.age < self.age)
- Cette simple contrainte fait que la précédente n'est plus utile car on ne pourra pas se retrouver dans ses descendants à cause de cette contrainte d'âge
- Ça n'est pas un problème d'avoir des contraintes se recoupant tant qu'elles ne sont pas contradictoires
  - Ici de plus, la contrainte de non appartenance à ses descendants est implicite par rapport à la contrainte d'âge
  - Il n'est donc pas inutile de la définir explicitement ou au moins de préciser que la contrainte d'âge implique la contrainte de descendance
- La difficulté principale avec OCL est d'arriver à trouver toutes les contraintes requises pour avoir une spécification complètement définie