Mémento du Langage OCL (Object Constraint Language v1.4)

v.0.8.1 © Philippe Collet - 2007

Ce document est un mémento du langage OCL. Il ne représente pas une référence exacte du langage, ni une description précise de sa sémantique. Il reflète uniquement l'interprétation de l'auteur d'après la spécification de la version 1.4, très proche de la version 2.0.

Utilisation

Au sein d'UML, OCL sert à spécifier des invariants de classe, des pré et postconditions d'opérations, et des corps de requêtes (fonctions sans effet de bord), ainsi que des gardes dans les diagrammes d'états. Toute expression OCL est sans effet de bord.

Contexte

Le contexte définit la portée d'une expression :

```
package packageP
      context ClassC
            inv ...
      context ClassC::operationO(i: Integer)
            pre ...
endpackage
```

Le contexte permet d'associer une contrainte à n'importe quel élément du modèle : package, classe, interface, composant. Il peut aussi dénoter un sous-élément comme une opération ou un attribut. Dans un contexte, le mot-clé `self` dénote l'objet courant.

Contraintes

On peut définir des contraintes comme des invariants (états des objets), des préconditions ou postconditions sur des opérations ou des définitions de fonctions (c'est-à-dire le corps d'une méthode « accesseur »).

Exemple d'invariants :

```
      context ClassA
            inv: attrA > 10
      context ClassB
            inv deuxiemeInvariant: attrB < 4
```

On peut avoir autant de déclarations `inv` que nécessaire. Chacune peut éventuellement être nommée par une étiquette. L'invariant résultant est la conjonction des déclarations (`i1 and i2 and`…).

Exemple de pré et postconditions :

```
      context ClassC::operationO(i1: Integer) : Integer
            pre valeurAssezGrande: i1 >= 10
            post: attrC >= attrC@pre + 2 and result >= (attrc+i1)/10
```

La notation `@pre` permet de dénoter l'état « avant » et `result` est un mot réservé qui représente le résultat de l'opération (si celle-ci rend bien un résultat). On peut avoir autant de déclarations `pre` et `post` que nécessaire, la pré/postcondition résultante est la conjonction des déclarations.

Définition de fonctions « accesseur » :

Les fonctions d'accès ne modifient pas l'état de l'objet et les pré/postconditions ne sont pas forcément utilisées pour les définir. Lorsque c'est possible, elles sont définies par une déclaration `body` :

```
      context ClassD::fonctionD(i2: Integer) : Integer
            body: attrD + 10 + i2
```

Types de base

Les types de base suivants font partie du langage :

- `Integer, Real: =, <>, <=, >=, +, -, *, /, x.mod(y), x.div(y)` (`div` est la division entière)
- `String: s.concat(t), s.size(), s.toLower(), s.toUpper(), s.subString(start, end)`
- `Boolean: and, or, not, xor, =, <>, implies, "if b1 then ... else ...endif"`

Collections

Les types suivants sont fournies pour les collections, tous sont des sous-types d'une classe abstraite `Collection`.

- `Set` : pas de doublons, pas d'ordre.
- `Bag` : doublons possibles, pas d'ordre.
- `OrderedSet` : pas de doublons, ordonné.
- `Sequence` : doublons possibles, ordonné.

Dans les expressions de navigation (a.b), le type du résultat dépend de la cardinalité de l'association :

- Cardinalités simples : 0 ou 1, rend un objet.
- Cardinalités multiples, sans ordre, rend un Set.
- Cardinalités multiples, avec ordre, rend une Sequence.

Exemple d'expressions sur des collections :

```
Type{initialiser}            Set{1, 2, 3}           Bag{'one', 'two', 'three', 'two'}
OrderedSet{true, false}      Sequence{1..30} -- Cas particulier des entiers
```

Opérations de manipulation des collections :

Les opérations de collection utilisent l'opérateur `->` (tous les indices commencent à 1) :

- `= <>` Collections identiques ou différentes
- `append(obj)` Rend la valeur de la collection ordonnée avec obj ajouté à la fin
- `asSet(), asOrderedSet(), asSequence()` Conversion de type entre collections
- `at(idx)` Rend l'objet à l'indice idx dans une collection ordonnée
- `count(obj)` Nombre d'apparition de obj dans une collection
- `excludes(obj)` Rend count(obj) = 0 ?
- `excludesAll(coll)` Est-ce que tous les éléments de coll satisfont count(obj) = 0 ?
- `excluding(obj)` Rend la valeur de la collection sans l'objet obj
- `first()` Rend le premier élément d'une collection ordonnée
- `includes(obj)` Rend count(obj) > 0 ?
- `includesAll(coll)` Est-ce que tous les éléments de coll satisfont count(obj) > 0 ?
- `including(obj)` Rend la valeur de la collection avec l'objet obj ajouté
- `indexOf(obj)` Rend l'indice de la 1ère occurrence de obj dans une séquence
- `insertAt(idx,obj)` Rend la valeur de la collection avec obj ajouté à l'indice idx
- `intersection(coll)` Intersection des collections non ordonnées self et coll
- `isEmpty()` Rend size() = 0 ?
- `last()` Rend le dernier élément d'une collection ordonnée
- `notEmpty()` Rend size() > 0 ?
- `prepend(obj)` Rend la valeur de la collection ordonnée avec obj ajouté en tête
- `size()` Nombre d'éléments dans la collection
- `subOrderedSet(start,end)` Sous-partie d'un ensemble ordonné entre les indices
- `subSequence(start,end)` Sous partie d'une séquence extraite entre les indices
- `sum()` Somme des éléments d'une collection (Integer ou Real)
- `union(coll)` Union des collections self et coll

Opérations collectives :

Les opérations collectives sont des opérations de parcours de collection, comme select(), collect() et forAll(). Elles sont utilisées pour appliquer des prédicats aux collections.

- `collect(expr)` Rend un bag contenant la valeur de exp appliquée à chaque élément de la collection
  `maCollection.valeur` est équivalent à `maCollection->collect(valeur)`
- `exists(expr)` Est-ce l'expression expr est satisfaite pour l'un des éléments de la collection ?
- `forAll(expr)` Est-ce l'expression expr est satisfaite pour tous les éléments de la collection ?
- `isUnique(expr)` Rend vrai si expr donne une valeur différente pour chaque élément de la collection
- `iterate(i : Type; a : Type | expr)`
  Opération à partir de laquelle les autres sont définies. La variable « i » est l'itérateur et « a » est l'accumulateur, qui récupère la valeur de expr après chaque évaluation
- `one(expr)` Rend l'expression `coll->select(expr)->size()=1`
- `reject(expr)` Rend la sous-collection des éléments pour lesquelles expr n'est pas satisfaite
- `select(expr)` Rend la sous-collection des éléments pour lesquelles expr est satisfaite
- `sortedBy(expr)` Rend la séquence contenant tous les éléments de la collection, ordonnés par la valeur expr (le type des éléments doit posséder un opérateur « < »)

Les opérations qui utilisent une condition peuvent être utilisées avec trois syntaxes équivalentes :

- `coll->select(attr >= 100)`
- `coll->select(e | e.attr >= 100)`
- `coll->select(e : typeElement | e.attr >= 100)`

Expression Let

Cette expression est utilisée pour définir des attributs ou des opérations temporaires afin qu'ils puissent être utilisés dans des contraintes pour simplifier l'expression et éviter les répétitions :

```
context ClassA inv labelA1:
      let val1:Boolean = attrA1 >= 1 and attrA2 <= 140
      let majeur(age :Integer):Boolean = age >= 18
      in val1 and (attrA3=30 or majeur(attrA4))
```