Introduction

Il n'est pas évident pour UML d'exprimer précisément ce que fait une opération. Dans le méta-modèle UML, une opération n'est définie que par son nom, ses paramètres et les exceptions qu'elle émet. Le corps de l'opération ne peut donc être défini. Le langage **OCL** (***O****bject* ***C****onstraint* ***L****anguage*) et le standard **AS** (***A****ction* ***S****emantics*) ont été précisément définis par l'OMG pour combler cette lacune et permettre la modélisation du corps des opérations UML.

OCL permet d'exprimer des contraintes sur tous les éléments des modèles UML. Il est notamment utilisé pour exprimer les pré- et post-conditions sur les opérations. Il est, par exemple, possible de dire que l'opération *crédit* d'une classe *CompteBancaire* a pour post-condition que le solde du compte soit incrémenté du montant passé en paramètre de l'opération.

AS peut être vu comme une solution de rechange à OCL du fait des difficultés d'utilisation de ce dernier pour la majorité des utilisateurs UML, plutôt habitués à écrire des suites d'instructions. Grâce à AS, il est possible de modéliser des constructions d'objets, des affectations de variables, des suppressions, des boucles for, des tests if, etc.

I- LANGAGE OCL

En utilisant uniquement UML, il n'est pas possible d'exprimer toutes les contraintes que l'on souhaiterait. Par exemple, il n'est pas possible de dire que l'attribut *solde* d'une classe *CompteBancaire* ne doit pas être inférieur à 1 000 (voir figure 1)

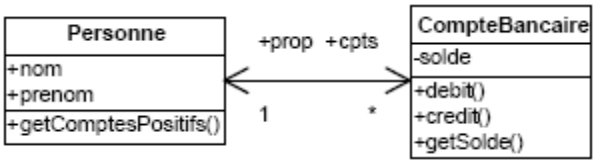

Figure 1 : Exemple de modèle UML

Fort de ce constat, l'OMG a défini formellement le langage textuel de contraintes OCL (Object Constraint Language), qui permet de définir n'importe quelle contrainte sur des modèles UML.

Le concept d'expression est au cœur du langage OCL. Une expression est rattachée à un contexte, qui est un élément de modèle UML, et peut être évaluée afin de retourner une valeur. Par exemple, une expression OCL dont le contexte serait la classe *CompteBancaire* permettrait d'obtenir la valeur de l'attribut *solde*. Une autre expression OCL, dont le contexte serait la classe *Personne*, permettrait de vérifier qu'une personne dispose ou non d'un compte bancaire.

Les expressions OCL ne génèrent aucun effet de bord[1]. L'évaluation d'une expression OCL n'entraîne donc aucun changement d'état dans le modèle auquel elle est rattachée. Une contrainte OCL est une expression dont l'évaluation doit retourner vrai ou faux. L'évaluation d'une contrainte OCL permet de la sorte de savoir si la contrainte est respectée ou non.

Si OCL permet d'exprimer des contraintes, il n'est pas pour autant un langage de programmation.

Il ne permet pas de construire de nouveaux éléments ni même de modifier ou de supprimer des éléments existants.

[1] Une fonction est dite à **effet de bord** si elle modifie un état autre que sa valeur de retour.

Pour modéliser le corps d'une opération UML, les contraintes OCL peuvent être utilisées de deux façons :

- Elles peuvent contraindre l'état des objets avant (pré-condition) et après (post-condition) l'invocation des opérations. Par exemple, pour l'opération *crédit*, il est possible de dire que le solde du compte après invocation est égal au solde du compte avant invocation en ajoutant le montant transmis en paramètre de l'opération.
- Elles peuvent contraindre les données contenues dans les messages échangés lors de l'invocation d'une opération. Il est ainsi possible de dire que le message retourné par un compte bancaire lorsque l'opération *getSolde* est invoquée contient un entier dont la valeur est égale au solde du compte bancaire.

L'objectif du langage OCL est aussi, et surtout, d'être indépendant de toute plateforme d'exécution. Une fois les contraintes OCL spécifiées sur un modèle UML, il est envisageable de générer une application conforme en Java, PHP ou .Net. Pour que l'application soit conforme, il suffit que les contraintes OCL soient respectées. Vérifier cette conformité est toutefois une difficulté qui n'a pas encore rencontré de solution industrielle.

Ajoutons qu'OCL est un langage fortement typé et qu'il est possible de détecter si une expression OCL est évaluable simplement en vérifiant la conformité des types qu'elle utilise.

1- Les expressions OCL

Une expression OCL porte sur un élément de modèle UML. Nous montrons ici la relation qui lie OCL avec UML.

1-1- Relation avec UML

1-1-1. Le contexte

Pour être évaluée, une expression OCL doit être rattachée à un contexte. Ce contexte doit être directement relié à un modèle UML. Même si le contexte d'une contrainte OCL peut être n'importe quel élément de modèle, nous ne nous intéressons ici qu'aux classes UML et aux propriétés des classes (opérations ou attributs).

La signification du contexte pour une contrainte OCL est la suivante :

- Si le contexte est une classe UML, la contrainte OCL est évaluée sur toutes les instances de la classe UML.
- Si le contexte est une propriété (opération ou attribut) d'une classe, la contrainte OCL est évaluée sur toutes les instances de la classe UML mais porte uniquement sur la propriété identifiée.

Dans une contrainte OCL, le mot-clé ***self*** permet de référencer l'élément sur lequel porte l'expression OCL. Grâce à cette référence, l'opérateur de navigation « **.** » permet de naviguer vers les propriétés de l'élément et vers les autres éléments avec lequel l'élément est associé. De proche en proche, il est possible d'évaluer grâce à d'autres expressions OCL les éléments appartenant à l'entourage du contexte initial.

Par exemple, dans l'expression suivante, si nous considérons que le contexte est la classe *CompteBancaire*, nous voyons qu'il est possible d'obtenir le nom du propriétaire du compte :

self.prop.nom

Le contexte peut être explicité à l'aide du mot-clé ***context***. Le mot-clé ***self*** devient dès lors implicite. L'exemple suivant est équivalent à l'exemple précédent :

context *CompteBancaire*

prop.nom

1-1-2. Les invariants

Une contrainte OCL peut exprimer un invariant (mot-clé ***inv***) sur une classe UML. Cela signifie que la contrainte OCL doit être tout le temps vraie pour toutes les instances de la classe.

L'exemple d'invariant suivant exprime que le solde d'un compte bancaire doit être toujours supérieur à – 1 000 :

context *CompteBancaire*

inv*: solde > -1000*

1-1-3. Pré- et postconditions

Une contrainte OCL peut exprimer une pré- et une post-condition, via les mots-clés ***pre*** et ***post***, sur une opération UML. Ces conditions permettent de contraindre les états des objets avant et après l'invocation de l'opération. Le mot-clé ***result*** permet d'identifier la valeur de retour de l'opération.

L'exemple suivant permet de spécifier, pour un compte bancaire, que la valeur du solde doit être positive avant toute invocation de l'opération ***debit*** :

context *CompteBancaire::debit():Integer*

pre*: solde>0*

L'exemple suivant permet de spécifier, pour un compte bancaire, que la valeur de retour de l'opération ***getSolde*** est égale à la valeur du solde du compte :

context *CompteBancaire::getSolde():Integer*

post*:* ***result****=solde*

OCL permet aussi, grâce au mot-clé ***@pre***, de manipuler dans une post-condition la valeur d'un attribut avant invocation de l'opération. L'exemple suivant spécifie que la valeur du solde lors d'un crédit est égale à la valeur du compte avant invocation augmentée de la somme créditée :

context *CompteBancaire::crédit(s:Integer)*

post*: result=solde@pre+s*

1-1-4. Les opérations de sélection

Étant donné qu'OCL permet de définir des requêtes sur des éléments de modèle, il est possible de l'utiliser pour définir le corps d'opérations qui ne font que sélectionner des éléments de modèle. Nous utilisons pour cela le mot-clé ***body***.

Par exemple, il est possible de spécifier en OCL le corps de l'opération permettant de sélectionner tous les comptes bancaires positifs d'une personne :

context *Personne* **: :** *getComptePositif():Set*

pre: *self.cpts.notEmpty()*

body*: self.cpts->select(c | c.solde>0)*

1-2- Types et valeurs

Le système de types du langage OCL est légèrement différent de celui d'UML.

1-2-1. Les types de base

Les types de base du langage OCL sont les booléens (*Boolean*), les entiers (*Integer*), les réels (*Real*) et les chaînes de caractères (*String*).

OCL fournit les opérations de base sur ces types, telles que et, ou, addition, soustraction, concaténation, etc.

1-2-2. Les types UML

Nous avons déjà vu que le contexte d'une contrainte OCL pouvait être une classe UML ou n'importe laquelle de ses propriétés.

Les classes UML sont considérées comme des types par OCL. Les objets instances de ces classes sont donc considérés comme les valeurs de ces types.

1-2-3. Les variables

Il est possible de définir des variables à l'aide du mot-clé ***let***. Cependant, OCL n'étant pas un langage à effet de bord, les variables OCL ont une valeur fixe, qui ne peut évoluer.

Elles sont, en quelque sorte, des raccourcis qui peuvent être utilisés dans d'autres expressions OCL.

1-2-4. Les opérations ajoutées

Il est possible de définir de nouvelles opérations à l'aide du mot-clé def afin de les utiliser dans d'autres expressions OCL. Les opérations ajoutées sont relativement semblables aux opérations existantes sur les classes UML, si ce n'est qu'elles ne peuvent être que des opérations de sélection et qu'elles doivent être entièrement spécifiées en OCL.

1-2-5. Conformité des types

Pour OCL, un type ***T1*** est conforme à un type ***T2*** si une instance de ***T1*** peut être substituée à toute place où une instance de T2 est demandée.

Pour les types de base, seul le type entier est conforme au type réel. Pour les types UML, c'est l'héritage entre classes qui indique la conformité entre types.

La conformité de types est transitive. Si ***T1*** est conforme à ***T2*** et que ***T2*** soit conforme à ***T3***, ***T1*** est conforme à ***T3***.

Une expression OCL est dite valide si tous ses types sont en conformité. En OCL, une expression qui ferait une addition entre une chaîne de caractères et un entier serait non valide, *String* et *Integer* n'étant pas conformes.

1-3- Propriétés des classes UML

1-3-1. Les attributs

Nous avons déjà vu qu'il était possible avec OCL d'obtenir la valeur d'un attribut d'une classe. Plus précisément, il est possible d'obtenir la valeur d'un attribut portée par un objet instance d'une classe.

Le type de la valeur correspond au type de l'attribut tel qu'il est défini dans la classe. Si, par exemple, l'attribut a pour type String, dans OCL, le type de la valeur sera String. Par contre, si l'attribut a pour type *String[]* (tableau de chaîne de caractères), dans OCL, le type de la valeur sera *Sequence* (collection de valeurs).

En UML, certains attributs sont statiques. Dans ce cas, la valeur de l'attribut est portée par la classe et non par les objets. Il est aussi possible en OCL d'obtenir la valeur de ces attributs. Pour ce faire, il faut utiliser non pas le mot-clé *self* mais directement le nom de la classe.

Rappelons qu'OCL étant un langage sans effet de bord, il n'est pas possible de modifier la valeur d'un attribut.

1-3-2. Les opérations

En OCL, il est possible d'invoquer des opérations sur des objets UML. En fait, seules les opérations de sélection sont invocables car elles ne modifient pas l'état d'un objet (elles ne modifient pas les valeurs des attributs de l'objet). On peut aussi faire des opérations produisant un résultat de test.

Les opérations qui ne sont pas des opérations de sélection ne peuvent être invoquées. Cela violerait le fait qu'OCL est un langage sans effet de bord.

Nous avons déjà vu qu'en OCL il était possible de définir de nouvelles opérations de sélection. Celles-ci doivent être entièrement définies (leur corps doit être spécifié en OCL) et être rattachées à une classe UML.

1-3-3. Les associations

En OCL, il est possible de naviguer *via* les associations UML. Les noms de rôles des associations sont utilisés pour définir comment se fait la navigation par défaut.

La navigation peut se faire de proche en proche. Il est en ce cas possible de chaîner les navigations et ainsi de parcourir une chaîne d'objets.

La navigation *via* les associations ayant une multiplicité « multiple », c'est-à-dire un à plusieurs, elle s'effectue grâce aux collections d'éléments. Les collections sont des types OCL. Elles offrent des fonctions permettant d'itérer sur les éléments de la collection.

1-4- Les collections

Les collections sont des types OCL décrivant des ensembles d'éléments. OCL fournit plusieurs fonctions permettant la manipulation des collections.

1-4-1. Les différentes sortes de collections

Pour manipuler les ensembles d'éléments, OCL propose différentes sortes de collections, appelées *Bag, Set, OrderedSet et Sequence*. Toutes ces collections sont des ensembles d'éléments de même type.

- ***Bag :*** Collection d'éléments dans laquelle les éléments ne sont pas ordonnés et où les doublons sont possibles (un même élément peut apparaître plusieurs fois dans la collection).
- ***Set*** : Collection d'éléments dans laquelle les éléments ne sont pas ordonnés et où les doublons ne sont pas possibles (un même élément ne peut apparaître plusieurs fois dans la collection).
- ***OrderedSet*** : Collection d'éléments dans laquelle les éléments sont ordonnés et où les doublons ne sont pas possibles.
- ***Sequence*** : Collection d'éléments dans laquelle les éléments sont ordonnés et où les doublons sont possibles.

1-4-2. Manipulation des collections

OCL propose différentes fonctions pour manipuler les collections, quel que soit leur type.

La fonction *select()* permet de sélectionner un sous-ensemble d'éléments d'une collection en fonction d'une condition. Nous pouvons obtenir le contraire de la fonction *select()* grâce à la fonction *reject().*

L'exemple suivant sélectionne les comptes bancaires dont le solde est positif :

context *Personne*

self.cpts->select(c | c.solde>0)

La fonction *forAll()* permet de vérifier si tous les éléments d'une collection respectent une expression OCL. L'expression OCL à respecter est passée en paramètre.

L'exemple suivant permet de s'assurer que tous les comptes bancaires ont un solde positif :

context *Personne*

self.cpts->forAll(c | c.solde>0)

La fonction *exist()* permet de vérifier si au moins un élément respectant une expression OCL existe dans la collection. L'expression OCL à respecter est passée en paramètre de la fonction.

L'exemple suivant permet de s'assurer qu'il existe au moins un compte bancaire avec un solde positif :

context *Personne*

self.cpts->exist(c | c.solde>0)

1-5- Les messages

Lors d'invocations d'opérations, UML considère que des messages sont échangés entre les objets. En OCL, il est possible de contraindre ces messages. Ce mécanisme permet de spécifier les interactions entre plusieurs objets.

L'exemple suivant illustre une interaction entre deux objets, un sujet (*Subject*) et un observateur (*Observer*), spécifiant que lorsqu'on invoque l'opération *hasChanged* sur un sujet, le sujet invoque l'opération *update* sur un observateur, avec *12* et *14* comme valeurs des paramètres de cette opération :

context *Subject* ***: :*** *hasChanged()*

post*: Observer^update(12,14)*

2- Le méta-modèle OCL2.0

Avant la version 2.0, OCL n'était spécifié qu'en langage naturel (anglais). Cela présentait l'inconvénient de rendre difficile l'alignement avec le standard UML. De plus, les expressions OCL étaient peu pérennes et peu productives.

L'objectif de la version 2.0 est principalement de spécifier OCL grâce à un méta-modèle afin que celui-ci soit facilement intégrable dans MDA. Depuis OCL2.0, les expressions OCL sont des modèles pérennes, productifs et explicitement liés aux modèles UML. Cela offre un gain significatif d'expressivité aux modèles UML.

2-1- OCL et UML

Les expressions OCL portent toutes sur des éléments de modèles UML. Il y a donc un fort lien entre le méta-modèle OCL et le méta-modèle UML.

La figure 4.2 illustre les méta-classes participant à ce lien.

Dans le méta-modèle UML, la méta-classe *Constraint* représente n'importe quelle contrainte. Cette méta-classe est reliée à la méta-classe *ModelElement*, qui joue le rôle de *constrainedElement*. Il est ainsi possible d'attacher une contrainte à n'importe quel élément du modèle. La méta-classe *Constraint* est aussi reliée à la méta-classe *Expression*, qui joue le rôle de *body*. Le corps des contraintes UML est de la sorte spécifié à l'aide d'une expression. UML laisse le choix du langage dans lequel les expressions doivent être écrites.

Le méta-modèle OCL définit quant à lui la méta-classe *ExpressionInOcl*, qui hérite de la Méta-classe *Expression.* Grâce à cet héritage, cette méta-classe représente des contraintes UML écrites avec OCL. Cette méta-classe est reliée à la méta-classe *OclExpression*, qui joue le rôle de *bodyExpression*. C'est cette dernière méta-classe qui représente réellement la spécification d'une contrainte en OCL.

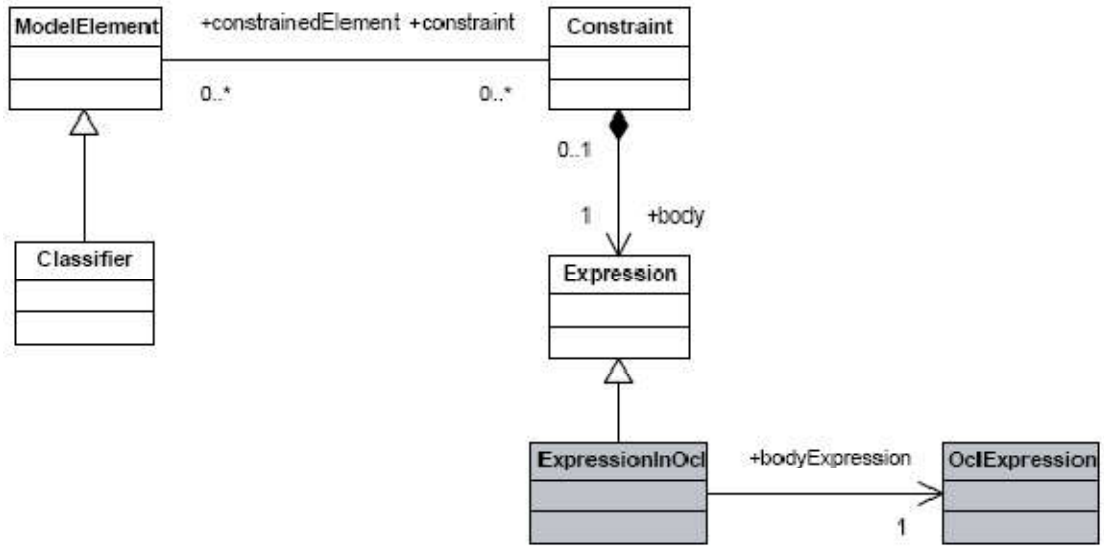

Figure 2 : Liens entre les méta-modèles UML et OCL

2-2- Le package Expressions

Le package *Expressions* du méta-modèle OCL, illustré en partie dans les figures suivantes, contient toutes les méta-classes permettant de représenter les expressions OCL sous forme de modèles.

Nous allons plus particulièrement présenter les méta-classes *OclExpression*, *VariableExp* et *ModelPropertyCallExp*, qui sont les trois classes de base permettant de modéliser une expression OCL.

La figure 3 illustre la partie principale du package Expressions.

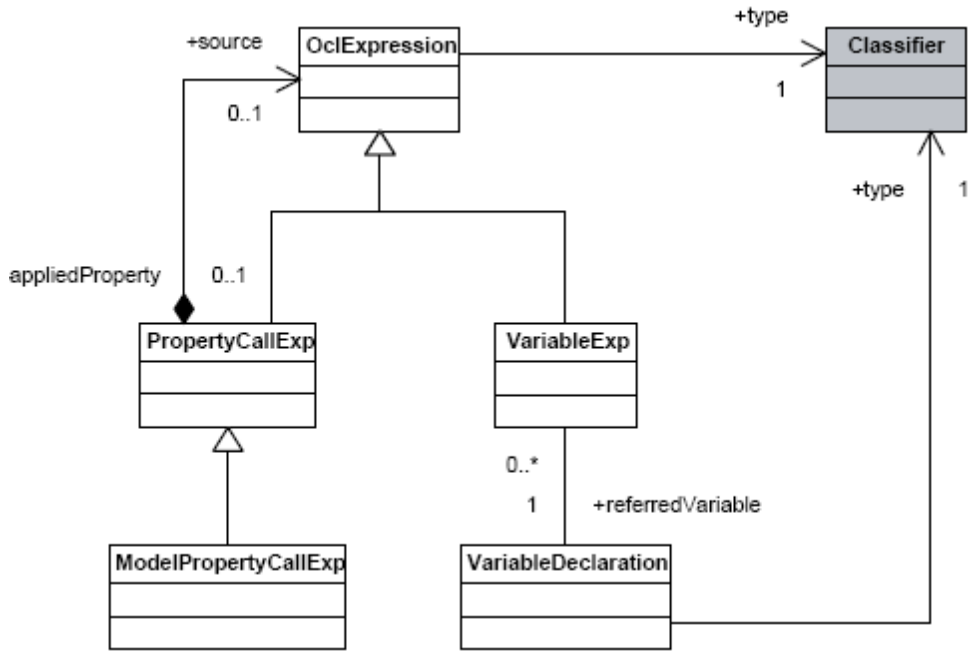

Figure 3 : La partie *OclExpression* du package *Expressions*

La méta-classe la plus importante est *OclExpression*. Héritée par toutes les autres méta-classes de ce package, elle représente n'importe quelle expression OCL.

Cette méta-classe est reliée à la méta-classe *Classifier* du méta-modèle UML, qui joue dans cette association le rôle de type. Cela signifie qu'une expression OCL a un type qui est un classifier, c'est-à-dire n'importe quel type OCL (voir le package *Types* à la section suivante). Si le type d'une expression OCL est booléen, l'expression peut être considérée comme une

contrainte. C'est d'ailleurs pourquoi la méta-classe *Constraint* n'apparaît pas dans ce package.

La méta-classe *VariableExp* représente une variable. Cette méta-classe est reliée à la méta-classe *VariableDeclaration*, qui représente une déclaration de variable. C'est grâce à cette méta-classe, par exemple, que la variable *self* est déclarée. La méta-classe *VariableDeclaration* est reliée à la méta-classe *Classifier*, qui joue le rôle de type. Cette association a pour but de définir le type de la variable ou le type du contexte si la variable est *self*.

La méta-classe *PropertyCallExp* représente un appel de propriétés. Cette méta-classe est héritée par la méta-classe *ModelPropertyCallExp*, qui représente un appel de propriétés d'une classe UML. Dans OCL, une propriété d'une classe UML peut aussi bien être un attribut, une opération ou une référence.

La figure 4 représente les méta-classes permettant de modéliser les expressions OCL qui sont des appels aux propriétés des classes UML (*ModelPropertyCallExp*). Parmi elles, nous ne présentons que la méta-classe *AttributeCallExp*, qui représente un appel à un attribut d'une classe. Cette méta-classe est reliée à la méta-classe *Attribut* du méta-modèle UML afin d'identifier l'attribut appelé.

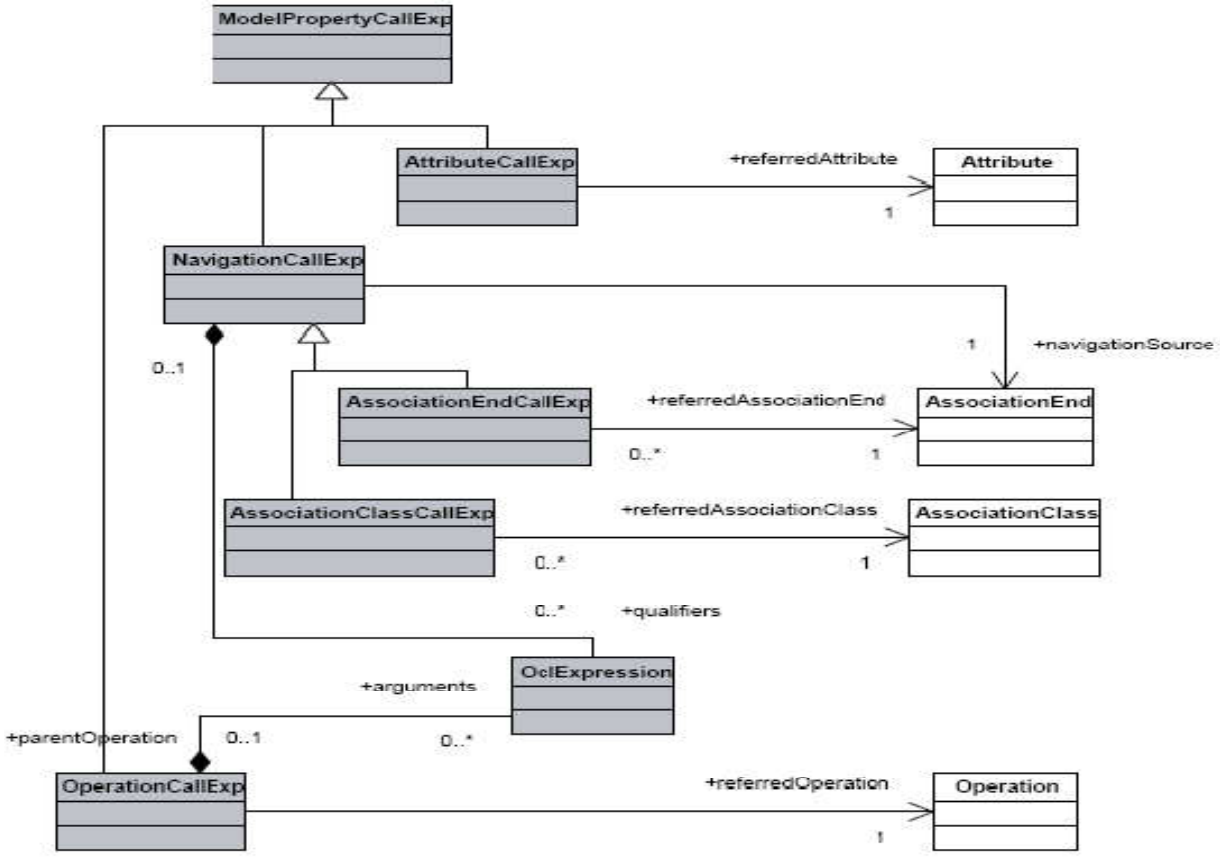

Figure 4 : La partie *ModelPropertyCallExp* du package *Expressions*

2-3- Le package Types

Nous avons déjà indiqué que toute expression OCL était typée. Dans le méta-modèle OCL, c'est le package Types, illustré à la figure 5, qui définit tous les types OCL. Il contient donc toutes les méta-classes nécessaires à l'élaboration des types OCL.

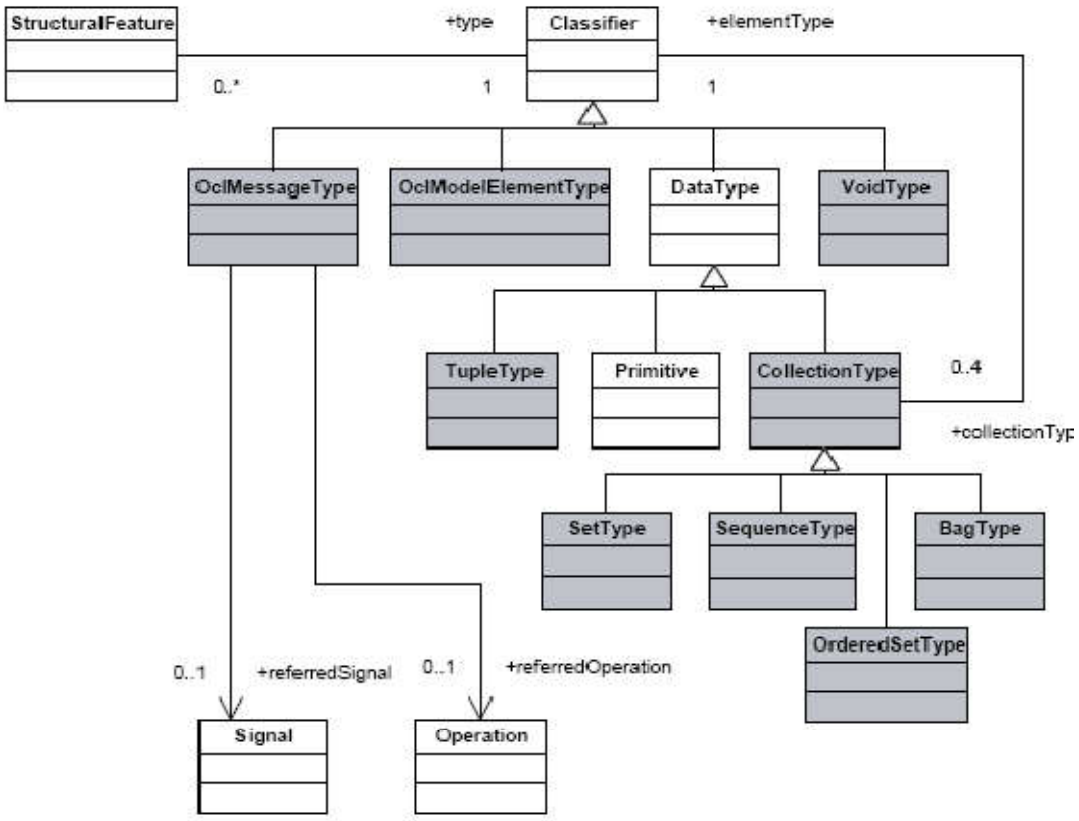

Figure 5 : Le package *Types* d'OCL2.0

Ce package référence la méta-classe *Classifier,* qui vient du package core d'UML1.4 (semblable à la méta-classe *Classifier* du package kernel d'UML2.0 Superstructure). Toutes les méta-classes qui représentent des types OCL héritent de cette méta-classe. Cela signifie que tous les types OCL sont des classifiers UML, c'est-à-dire des types UML.

Le package *Types* contient la méta-classe *VoidType*, qui permet de représenter le type *void.* Il contient aussi la méta-classe *OclModelElementType*, qui permet de représenter dans OCL les types UML, ceux-ci étant parfois manipulés dans les expressions OCL. Ce package contient enfin la méta-classe *OclMessageType*, qui permet de typer les messages OCL. Cette méta-classe étant reliée aux méta-classes *Signal* et *Operation* d'UML, les messages sont liés soit à un signal, soit à une opération.

Le package *Types* référence aussi la méta-classe *DataType*, qui vient du package core d'UML1.4. Les méta-classes qui représentent les collections (*CollectionType, SetType, SequenceType, OrderedSetType et BagType*) héritent de cette méta-classe. Cela signifie que les collections UML sont des types de données UML. Ce package référence aussi la méta-

classe *Primitive*, qui représente les types de base (*Integer, Boolean, etc.*). Ce package contient enfin la méta-classe *TupleType*, qui permet de mettre dans une même structure différents types OCL.

En association avec chacune de ces méta-classes, OCL définit un ensemble d'opérations. Ces dernières sont celles que l'on peut faire classiquement sur les valeurs de ces types. Nous y retrouvons les opérations d'addition, de soustraction et de comparaison d'entiers, d'égalité de booléens, de manipulation de caractères, etc. Elles sont appelées dans les expressions OCL lorsque nous voulons, par exemple, exprimer une égalité ou une somme.

2-4- Modèle versus texte

Le méta-modèle OCL permet de représenter n'importe quelle expression OCL sous forme de modèle. Cette représentation permet de rendre les expressions OCL pérennes et productives. De plus, le lien fort qui unit les modèles UML et les expressions OCL représentées sous forme de modèles permet d'exploiter pleinement les modèles UML dans une approche MDA. OCL contribue de la sorte à faire des modèles UML des PIM de MDA.

Pour des raisons évidentes de simplicité d'utilisation, OCL reste avant tout un langage textuel. Les contraintes OCL sont donc toujours représentées textuellement, avec la syntaxe que nous avons brièvement présentée en début de chapitre.

Un effort important a été fourni pour définir la façon de passer automatiquement de la forme textuelle à la forme modèle. Cette traduction entre le format textuel et le format modèle n'est pas triviale. Nous allons l'illustrer au travers d'un exemple que nous présentons d'abord sous un format textuel puis sous un format modèle.

L'exemple est le suivant :

context *CompteBancaire*

inv: *solde > -1000*

Nous avons déjà présenté cet exemple en début de chapitre. Il s'agit d'une contrainte OCL attachée à la classe UML *CompteBancaire*, qui permet d'exprimer le fait que le solde du compte bancaire ne doit pas être inférieur à – 1000 .

La figure 6 illustre une partie du modèle de cette contrainte OCL. Nous voyons que la contrainte est une contrainte UML (*Constraint*) et qu'elle est attachée à une classe UML, en l'occurrence un élément de modèle UML. Cette contrainte contient une expression, *ExpressionInOcl,* qui est spécifiée à l'aide de l'expression OCL *OclExpression*.

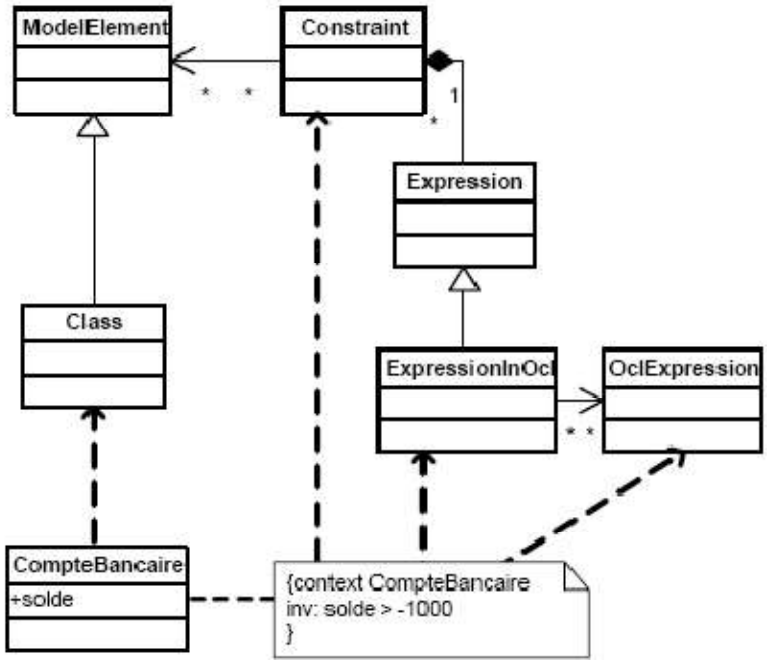

Figure 6 : Contrainte OCL sous forme de modèle (1/2)

La figure 7 détaille cette expression OCL sous forme de modèle. L'expression OCL commence par un appel à opération (*PropertyCallExp*). L'opération appelée est >, de la méta-classe *Integer*. Un des paramètres de cette opération est la valeur littérale -1000.

L'autre paramètre de l'opération est un appel à l'attribut *solde* de la classe *CompteBancaire (ModelPropertyCallExp).*

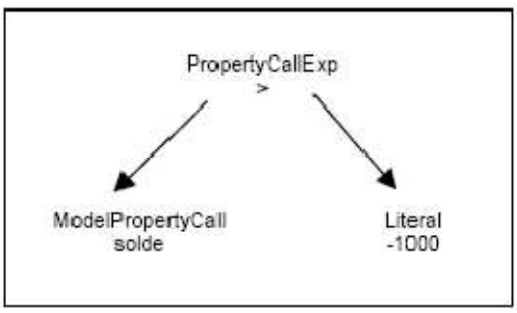

Figure 7 : Contrainte OCL sous forme de modèle (2/2)

Cet exemple illustre le fait qu'il est possible de traduire automatiquement les expressions OCL textuelles vers des modèles, même si cette traduction est très difficile à implémenter pour le constructeur d'outils de modélisation. Grâce à cette traduction, les expressions OCL peuvent être facilement saisies par les utilisateurs des outils de modélisation (sous forme textuelle) tout en restant pérennes et productives car transformées automatiquement en modèles.

3- En résumé

Nous avons vu que le langage OCL permettait de spécifier des contraintes sur les modèles UML. Grâce à ces contraintes, il est possible de spécifier le corps des opérations des classes. Les contraintes OCL sont indépendantes des langages de programmation. La traduction d'une contrainte OCL en un jeu d'instructions dans un langage de programmation est d'ailleurs un problème qui n'est pas encore résolu dans MDA.

OCL permet de préciser les modèles UML tout en faisant en sorte qu'ils soient toujours indépendants des plates-formes de programmation. Il est principalement utilisé dans MDA pour l'élaboration des PIM (Platform Independent Model). Afin de clarifier l'intégration du standard OCL dans MDA, l'OMG a décidé de le standardiser sous forme de méta-modèle. Grâce au méta-modèle OCL, les contraintes OCL sont désormais représentées sous forme de modèles.

Le méta-modèle OCL est relié au méta-modèle UML de façon à exprimer la dépendance entre UML et OCL. Ce lien est un lien interne vers le méta-modèle OCL. Il est donc intégralement défini dans le méta-modèle OCL. Ce lien est actuellement établi pour le méta-modèle UML1.4. La version OCL2.1 spécifiera le lien vers UML2.0 Superstructure. L'évolution du lien vers UML2.0 Superstructure ne devrait pas poser trop de problème étant donné le rapprochement conceptuel entre les packages core d'UML1.4 et kernel d'UML2.0 Superstructure.

Grâce à ce lien, il est possible de naviguer de manière informatique de la contrainte OCL vers le modèle UML. Il est de la sorte envisageable d'automatiser l'évaluation des contraintes OCL. Au finale, on peut dire que le méta-modèle OCL a permis de rendre les contraintes OCL productives.

II- LE LANGAGE AS

Jusqu'à sa version 1.4, UML était très critiqué parce qu'il ne permettait pas de spécifier des créations, des suppressions ou des modifications d'éléments de modèles. Ces actions ne pouvant pas non plus être spécifiées à l'aide du langage OCL, puisque celui-ci est sans effet de bord, il était nécessaire de standardiser un nouveau langage. C'est ce qui a donné naissance au langage AS (Action Semantics).

L'objectif d'AS est de permettre la spécification d'actions. Une action, au sens AS du terme, est une opération sur un modèle qui fait changer l'état du modèle. Grâce aux actions, il est possible de modifier les valeurs des attributs, de créer ou de supprimer des objets, de créer de nouveaux liens entre les objets, etc. Le concept d'action permet de spécifier pleinement le corps des opérations UML.

AS a tout d'abord été standardisé comme un langage à part entière, lié au standard UML, avant d'être inclus dans la version 1.5 d'UML. Dans UML2.0, il est enfin totalement intégré au méta-modèle. Cette section présente AS tel que défini dans le standard UML2.0 Superstructure.

1- Le méta-modèle AS

AS n'est standardisé que sous forme de méta-modèle, et aucune syntaxe concrète n'est définie, contrairement à OCL. Ce standard laisse donc la liberté à chacun d'utiliser la syntaxe qu'il souhaite. Le fait qu'il n'existe aucune syntaxe de référence est toutefois une lacune assez gênante puisqu'il est impossible de trouver des exemples présentant de manière textuelle des ensembles d'actions.

Dans UML2.0, AS constitue le socle de définition de la dynamique des modèles. Les modèles dynamiques UML (activité, séquence et machine à états) s'appuient dessus. AS assure l'unification et la cohérence de l'ensemble des modèles dynamiques.

Même si l'objectif premier d'AS était de permettre la spécification des corps des opérations UML et leur transformation vers des langages de programmation, il apparaît aujourd'hui que d'autres utilisations sont possibles.

AS peut être utilisé pour des transformations de modèles. Il est possible d'exprimer des transformations de modèles UML avec AS car celui-ci propose des opérations de création, de suppression et de modification d'éléments de modèles.

Une autre utilisation possible d'AS est l'exécution de modèles UML. Il est envisageable de construire des machines virtuelles UML permettant l'exécution des actions directement sur les modèles. Cette utilisation relève encore de la recherche mais semble prometteuse selon l'OMG puisqu'elle est en cours de standardisation.

1-1- Les packages constituant AS

Le méta-modèle UML2.0 Superstructure intègre totalement le méta-modèle AS. Plus précisément, le méta-modèle AS se retrouve dans les packages *StructuredActivities, CompleteActions et IntermediateActions* du méta-modèle UML2.0 Superstructure.

Le package *IntermediateActions* définit les méta-classes représentant les actions classiques de création, de suppression et de modification d'éléments de modèle. Ce package merge le package *StructuredActivities*, lequel contient les méta-classes nécessaires à la représentation d'activités.

Le package *CompleteActions* définit les méta-classes représentant des actions très spécifiques, telles les actions permettant de modifier les liens d'instanciation entre des objets et leur classe. Nous n'utilisons pas ce package ni ce genre d'action . Notons simplement que ce package merge le package *IntermediateActions.*

La figure 8 illustre la découpe des relations entre les packages du méta-modèle UML2.0 Superstructure qui incarnent les concepts d'AS.

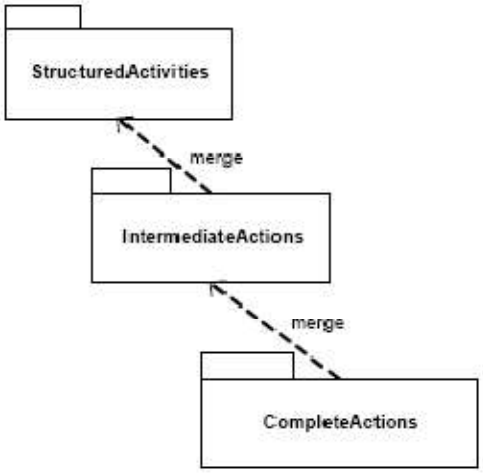

Figure 8 : Les packages AS dans UML2.0

1-2- Le package StructuredActivities

Le package *StructuredActivities* définit principalement le concept d'activité comme étant un enchaînement d'actions (voir figure 9).

Il contient la méta-classe *Activity*, qui représente la notion d'activité, et la méta-classe *ActivityNode*, qui représente une étape, ou nœud, de l'activité. La méta-classe *Activity* est reliée à la méta-classe *ActivityNode* afin d'exprimer le fait que l'activité est constituée d'un ensemble d'étapes.

Ce package contient aussi la méta-classe *Action*, qui hérite de la méta-classe *ActivityNode*. Cela signifie qu'une étape d'une activité peut être une action. D'autres méta-classes héritent de la méta-classe *ActivityNode,* mais nous ne nous attardons pas sur elles.

Le package *StructuredActivities* contient la méta-classe *ActivityEdge*, qui représente un lien entre deux étapes d'une même activité. Cette méta-classe est reliée à la méta-classe *ActivityNode* par deux méta-associations. Cela permet de structurer les activités sous forme de graphes d'actions.

Le package *StucturedActivities* définit un peu plus précisément la notion d'action comme étant quelque chose d'exécutable, qui accepte en entrée et rend en sortie des données (voir figure 10). Ce package contient en effet la méta-classe *Pin*, qui représente la notion de donnée. Un pin comporte une valeur. Les méta-classes *OutputPin* et *InputPin* représentent

les données en entrée et en sortie d'une action. La méta-classe *Action* est donc reliée aux méta-classes *OutputPin* et *InputPin*. Grâce à ces méta-classes, il est possible de spécifier que les données en sortie d'une action sont consommées en entrée d'une autre action. Dit autrement, cela permet de spécifier des flots de données entre les actions.

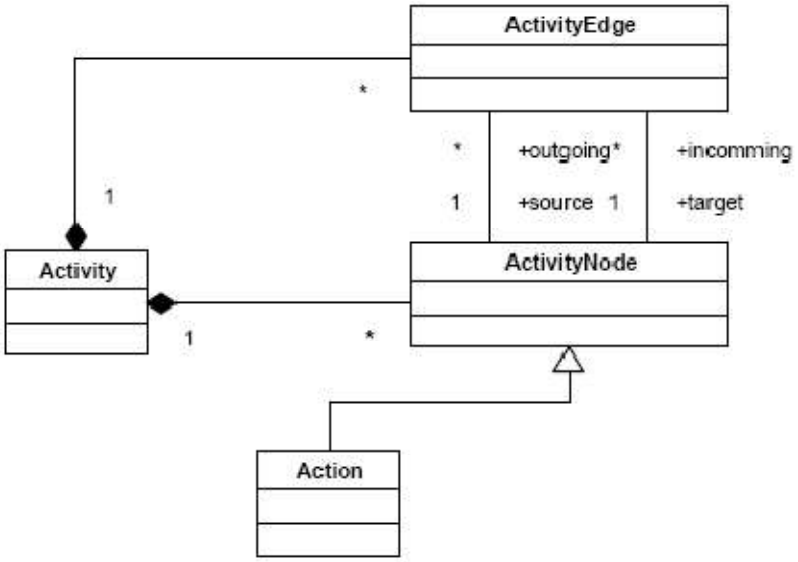

Figure 9 : La méta-classe *Activity* dans le package *StructuredActivities*

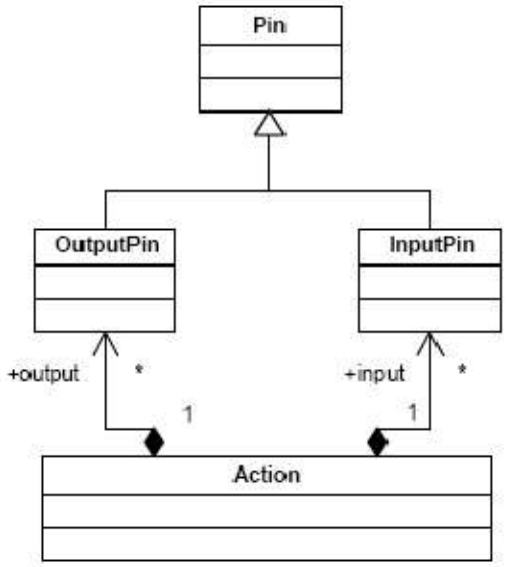

Figure 10 : La méta-classe *Action* dans le package *StructureActivities*

1-3- Le package IntermediateActions

Le package *IntermediateActions* contient toutes les méta-classes permettant de représenter les actions classiques de création, de suppression et de modification d'éléments de modèle. Nous ne détaillons pas ici la totalité de ces méta-classes et ne nous concentrons que sur certaines d'entre elles afin de bien faire comprendre comment élaborer une suite d'instructions en utilisant AS. Nous renvoyons le lecteur curieux au standard UML2.0 Superstructure pour obtenir la liste complète des méta-classes représentant des actions AS.

La figure 11 illustre la partie du package *IntermediateAction* contenant la méta-classe *CallOperationAction*. Cette méta-classe représente une action qui est un appel à une opération d'une classe UML. Cette méta-classe est reliée à la méta-classe *Operation* du méta-modèle UML afin d'identifier l'opération à appeler. Cette méta-classe est aussi reliée à la méta-classe *InputPin* afin d'identifier l'objet sur lequel sera appelée l'opération.

La méta-classe *InvocationAction* représente une action qui est une invocation. Cette méta-classe est reliée à la méta-classe *InputPin* afin d'identifier les arguments de l'invocation et à la méta-classe *OutputPin* afin d'identifier le résultat de l'invocation.

La méta-classe *CallOperationAction* hérite de la méta-classe *CallAction*, qui hérite à son tour de la méta-classe *InvocationAction*. La méta-classe *CallOperationAction* a donc elle aussi un résultat et des paramètres.

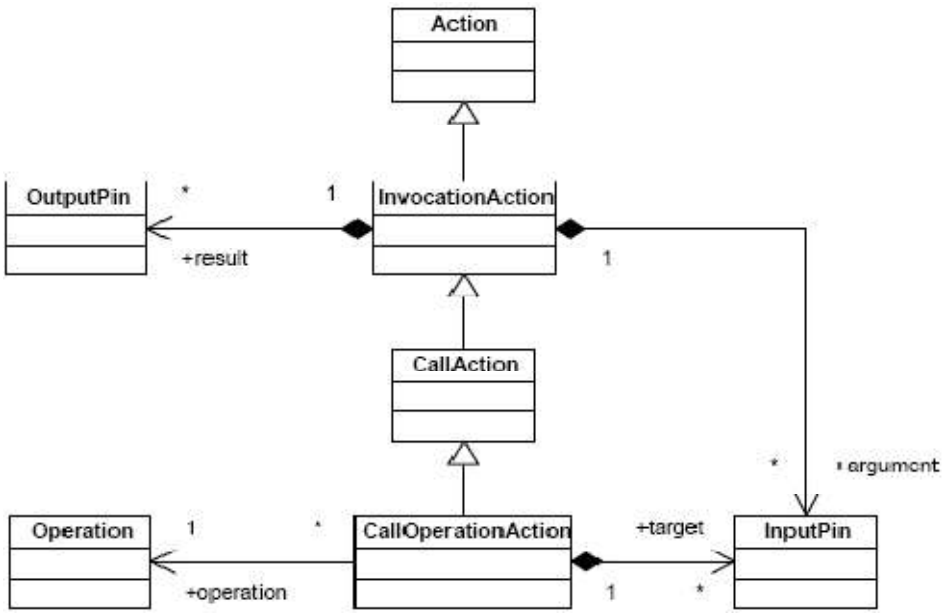

Figure 11 : La méta-classe *CallOperationAction*

La figure 12 illustre une autre partie du package *IntermediateActions* contenant les méta-classes *CreateObjectAction* et *DestroyObjectAction*. Ces méta-classes représentent respectivement des actions de création et de destruction d'objet.

La méta-classe *CreateObjectAction* est reliée à la méta-classe *Classifier* afin d'identifier la classe de l'objet à construire. Elle est aussi reliée à la méta-classe *OutputPin* afin d'identifier l'objet qui sera créé après l'exécution de l'action.

La méta-classe *DestroyObjectAction* est reliée à la méta-classe *InputPin* afin d'identifier l'objet à détruire.

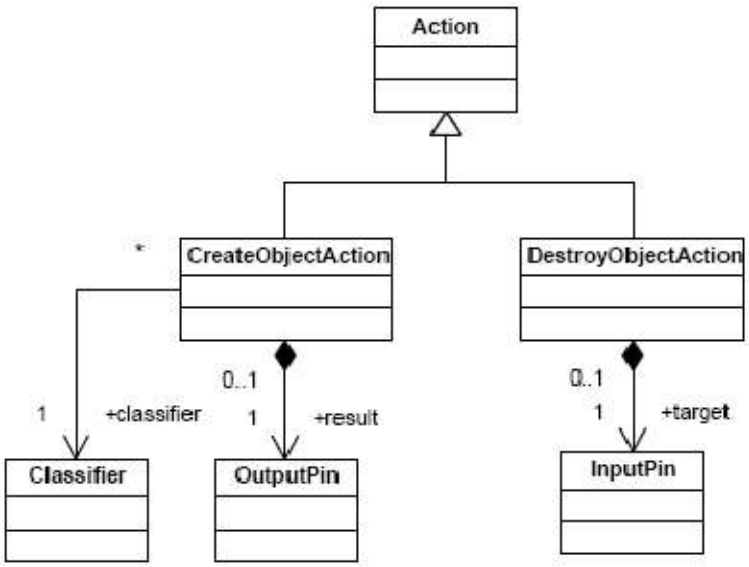

Figure 12 : Les méta-classes *CreateObjectAction* et *DestroyObjectAction*

La figure 13 illustre la partie du package *IntermediateActions* contenant la méta-classe *StructuralFeature*. Cette méta-classe représente une action d'accès à une propriété (attribut ou référence, par exemple). Elle est donc reliée à la méta-classe *StructuralFeature* du méta-modèle UML afin d'identifier la propriété qui sera accédée.

La méta-classe *ReadStructuralFeatureAction*, qui hérite de la méta-classe *StructuralFeature Action*, représente l'action de lecture de la valeur d'une propriété. Cette méta-classe est reliée à la méta-classe *OutputPin* afin d'identifier cette valeur retournée.

La méta-classe *WriteStructuralFeatureAction,* qui hérite aussi de la méta-classe *Structural FeatureAction*, représente une action qui écrit une nouvelle valeur pour une propriété. Cette

méta-classe est reliée à la méta-classe *InputPin*, qui représente la nouvelle valeur de la propriété.

La méta-classe *AddStructuralFeatureValueAction,* qui hérite de la méta-classe *WriteStructural FeatureAction,* représente l'action d'écriture d'une nouvelle valeur pour une propriété qui est un tableau. Cette méta-classe est reliée à la méta-classe *InputPin* jouant le rôle d'*insertAt* afin de définir l'index de la case du tableau dans laquelle se fera l'écriture.

Le package *IntermediateActions* définit d'autres méta-classes représentant toutes les autres actions exécutables sur des éléments de modèle. Il existe, par exemple, des actions pour ajouter des liens entre les objets, émettre des messages, naviguer parmi les instances d'une classe, etc. Le package contient en tout une cinquantaine de méta-classes. Pour une liste complète de ces méta-classes, voir directement le standard UML2.0 Superstructure.

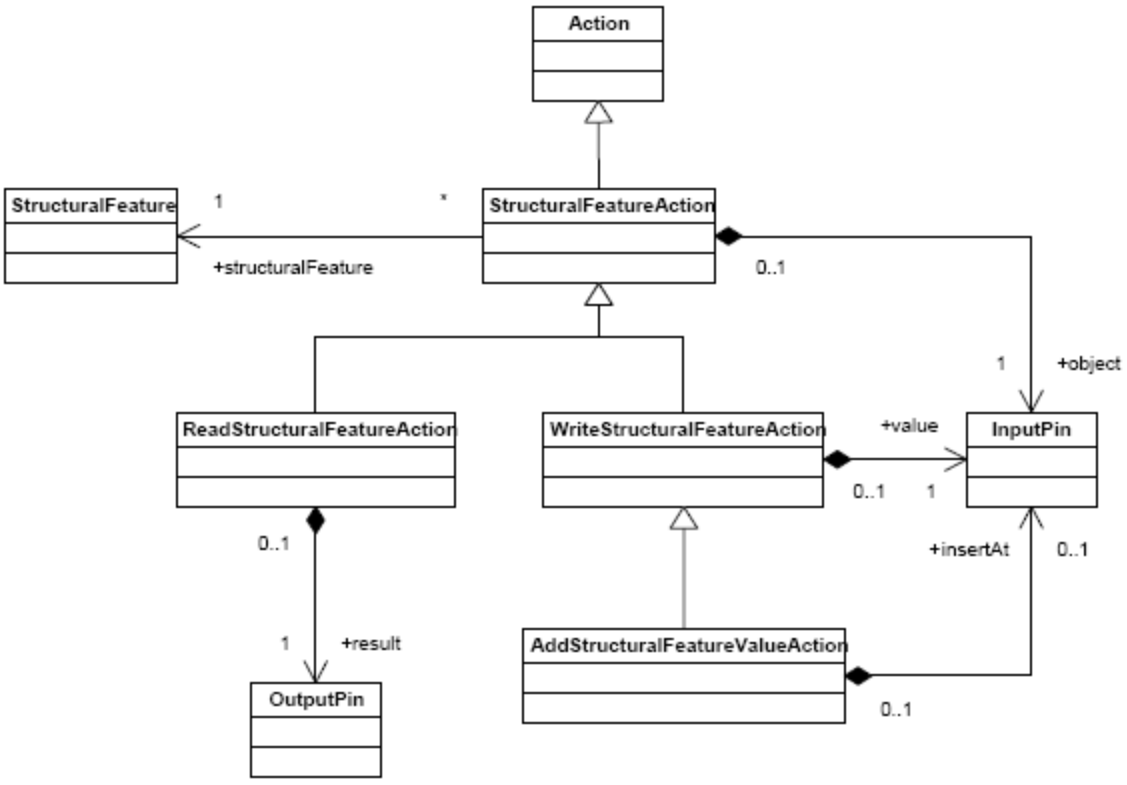

Figure 13 : Les méta-classes *ReadStructuralFeatureAction* et *WriteStructurealFeatureAction*

1-4- Lien avec la syntaxe concrète

Comme expliqué précédemment, AS n'est défini que sous forme de méta-modèle et ne propose pas de format concret. Il n'est donc pas possible de spécifier des activités sous un format concret.

Le standard précise cependant que plusieurs formats concrets peuvent être utilisés et qu'il est du ressort du concepteur d'outil de modélisation de fournir un format concret et d'expliquer comment ce dernier peut se traduire en un format de modèles. Ce travail étant particulièrement difficile à réaliser, aucune proposition de format concret n'a encore réellement séduit les utilisateurs. C'est la raison pour laquelle AS n'est pas encore pleinement exploité, contrairement à OCL.

Afin de mieux faire comprendre l'importance d'un format concret, nous allons détailler un exemple simple d'activité. Cette activité contient trois actions : la création d'un compte bancaire, l'affectation d'un montant au solde et l'appel à une opération de débit. Nous proposons d'exprimer cette activité en utilisant une syntaxe proche du langage Java.

Cette activité se représenterait de la façon suivante :

*CompteBancaire cb = **new** CompteBancaire() ;*

cb.solde=100 ;

cb.debit(10) ;

La figure 14 donne une représentation de cet exemple sous forme de modèle. Nous avons choisi d'illustrer les actions sous forme de cercle et les flots de données sous forme de flèche. La figure fait apparaître les trois actions et les flots de données. Pour que le modèle soit complet, il faudrait faire apparaître l'activité qui définirait la séquence entre les actions. Cet exemple montre combien il est difficile de passer d'un format concret (ici textuel) à un format de modèle.

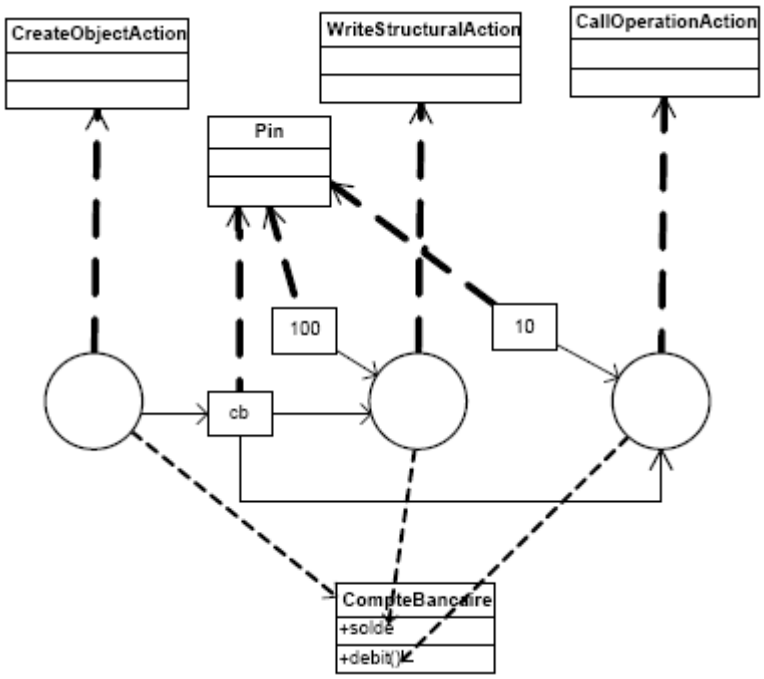

Figure 14 : Les actions sous forme de modèle

2- En résumé

Nous avons vu que le langage AS permettait de spécifier des actions sur les modèles UML. Il est ainsi possible de spécifier le corps des opérations des classes à l'aide d'une suite d'actions (une activité). Les actions AS sont indépendantes des langages de programmation, l'objectif étant de traduire une suite d'actions AS vers un langage de programmation.

AS permet d'élaborer des modèles UML très précis, quasiment exécutables. Comme ces modèles UML sont toujours indépendants des plates-formes d'exécution, AS est utilisé pour l'élaboration des PIM.

Nous avons vu qu'AS était défini à l'aide d'un méta-modèle. Ce méta-modèle est entièrement intégré au méta-modèle UML2.0. De ce fait, les actions AS sont fortement liées aux modèles UML, ce qui permet de rendre les modèles UML beaucoup plus pérennes et productifs.

Nous avons vu aussi qu'AS ne proposait pas de format concret de représentation. Il s'agit là d'une lacune assez gênante puisqu'il n'est pas possible pour un utilisateur de saisir facilement une suite d'instructions.

Tout comme OCL, AS est utilisé pour élaborer des transformations de modèles.

CONCLUSION

Même si leurs approches sont différentes, OCL et AS sont des standards relativement proches. Ils permettent de modéliser les corps des opérations UML.

L'approche OCL consiste à définir des contraintes sur les opérations afin de préciser ce que doit être la sortie d'une opération en fonction de son entrée. Plus proche des langages de programmation, l'approche AS consiste à définir une suite d'actions modifiant l'état d'un modèle. Les deux standards permettent d'élaborer des modèles UML beaucoup plus précis en définissant dans le détail les comportements des objets.

OCL et AS sont entièrement indépendants des plates-formes d'exécution. Grâce à ces standards, les modèles UML peuvent représenter intégralement la logique métier d'une application, et ce indépendamment des plates-formes d'exécution. On peut donc dire qu'ils offrent un gain significatif de pérennité aux modèles UML. Ils sont essentiellement utilisés pour l'élaboration des PIM dans MDA.

Les standards OCL et AS sont définis entièrement sous forme de méta-modèles. Ces derniers étant intégralement liés au méta-modèle UML, il est possible d'automatiser leurs traitements. Ils sont à ce titre pleinement inscrits dans MDA, même s'il est vrai qu'actuellement peu de produits commerciaux les supportent complètement.

Bibliographie:

- Ingénierie logicielle guidée par les modèles, X a v i e r B l a n c, 2005.
- Object Constraint Language (OCL), Master 2 IMS Informatique-PRO - Module UMIN346, Marianne Huchard, 2006.
- **OCL:** *Object Constraint Language* Le langage de contraintes d'UML, Eric Cariou, 2003.