Hyper-généricité pour les langages à objets : le modèle `OFL`

Adeline Capouillez, Robert Chignoli, Pierre Crescenzo et Philippe Lahire

Laboratoire I3S
CNRS – UNSA
Projet OCL
Les Algorithmes – Bâtiment Euclide B
2000, route des Lucioles
B.P. 121
F-06903 Sophia Antipolis CEDEX
France

{Adeline.Capouillez|Robert.Chignoli|Pierre.Crescenzo|Philippe.Lahire}@unice.fr
http://www.i3s.unice.fr/~ocl/

RÉSUMÉ : *Dans cet article, nous décrivons certains aspects du modèle* `OFL` *qui repose essentiellement sur le paramétrage des classes et des relations entre classes, caractéristique que nous nommons hyper-généricité. Nous souhaitons montrer que le paramétrage des relations entre classes permet de modéliser une grande variété de relations et d'améliorer ainsi la qualité du code produit. Le fait d'utiliser ce système de paramètres permet aussi le plus souvent de s'affranchir d'une lourde étape de méta-programmation. Pour illustrer notre propos, nous définissons deux nouvelles relations* `is-a-view-of` *qui offrent la capacité de créer des points de vue sur les objets. La première prend la forme d'une relation d'utilisation comme la clientèle. La seconde est décrite comme une relation d'importation telle l'héritage.*

ABSTRACT: *In this article, we describe some aspects of the* `OFL` *model, which is essentially based on customisation of classes and relationships between classes. We name this feature hyper-genericity. We want to show that the customisation of relationships between classes allows to model a lot of relationships and to improve software code quality. With this parameters system, we also can free from a huge meta-programming step. To illustrate that, we define two new* `is-a-view-of` *relationships, which provide the ability to create new viewpoints on objects. The first one is a use relationship such as clientele. The second one is described as an importation relationship such as inheritance.*

MOTS-CLÉS : *modèle méta-objet, hyper-généricité, relation entre classes, vue, langage à objets, base de données.*

KEY WORDS: *Meta-Object Model, Hyper-Genericity, Relationship between Classes, View, Object-Oriented Language, Database.*

1. Introduction

« Les relations entre classes dans les langages à objets, et notamment l'héritage, sont des mécanismes de trop bas niveau qu'il serait intéressant de mieux spécifier. » Cette réflexion, issue de nos expériences en programmation orientée objets, est à l'origine de l'élaboration du modèle OFL (*Open Flexible Languages*).

Au fur et à mesure du développement du modèle OFL[1], notre démarche s'est enrichie d'autres motivations que nous présentons dans la section 2.1. OFL n'est donc plus simplement destiné à améliorer l'expressivité de l'héritage. Il modélise les principaux concepts des langages à objets à classes et les met en œuvre pour décrire les langages les plus courants tels Java [ARN 98], C++ [STR 97] ou Eiffel [MEY 97]. Ainsi, le méta-programmeur a la capacité de modifier le comportement de son langage de prédilection pour résoudre les problèmes rencontrés.

OFL fut tout d'abord conçu sous la forme d'un protocole méta-objet tel celui de CLOS [KIC 91]. Cependant, plus ouvert et plus complet que ce dernier, il devint rapidement très difficile et fastidieux d'une part de le programmer, d'autre part de l'utiliser. Pour pallier ce problème, nous nous sommes orientés vers une approche hyper-générique [DES 94]. Plutôt que de permettre de redéfinir les comportements à l'aide d'algorithmes, nous proposons un ensemble de paramètres. Les algorithmes, déjà implantés, prennent en compte les valeurs de ces paramètres pour mettre en œuvre le comportement souhaité. Nous les appelons actions et elles définissent la sémantique opérationnelle.

2. Motivations et principales caractéristiques de l'approche

2.1. *Motivations*

L'objectif initial était donc d'augmenter l'expressivité de la relation d'héritage afin de mieux spécifier les usages qui en sont faits. Aujourd'hui nous avons étendu cette démarche à l'ensemble des relations inter-classes et non plus seulement à l'héritage.

La principale de nos motivations est l'amélioration de la qualité du code. Notre but est donc de spécifier plus précisément l'usage qui est fait des relations entre classes, qu'il s'agisse d'héritage, de clientèle ou d'une autre relation. Nous

[1] La description complète du cœur du modèle OFL fait notamment l'objet d'une thèse de doctorat en cours de rédaction.

souhaitons, par ce biais, augmenter la lisibilité des codes sources et permettre ainsi une meilleure documentation. Plus d'information au niveau des relations, c'est aussi un contrôle plus pertinent par le compilateur ou l'interprète. C'est enfin une meilleure maintenabilité et donc une pérennité prolongée des applications.

Un second objectif est de réduire le fossé entre les méthodes de conception telles `UML` et les langages de programmation. Une très belle modélisation en `UML` est certes particulièrement satisfaisante mais souvent difficile à implanter directement lorsque nous nous retrouvons face à notre langage de programmation favori. Ou pour éviter cela, et ce n'est pas forcément une bonne solution, nous en venons parfois à n'utiliser qu'une sous-partie des possibilités des méthodes de conception. Nous modélisons en effet en pensant à l'avance à la phase de programmation qui suivra. Aucun de ces deux choix n'est vraiment souhaitable[2]. De notre point de vue, un but à atteindre est donc d'augmenter l'expressivité et la souplesse des langages de programmation, de manière à permettre par exemple, une meilleure prise en compte des problèmes de classification [CAP 00b], de persistance ainsi que d'évolution des applications et des langages eux-mêmes [CAP 00a, CAP 00c].

2.2. *Hyper-généricité*

Le moyen que nous avons choisi pour améliorer la qualité des logiciels produits est basé sur l'hyper-généricité, c'est-à-dire le paramétrage des relations inter-classes afin d'obtenir le comportement souhaité par le programmeur.

Nous avons donc défini un ensemble de paramètres généraux qui représentent les principales caractéristiques des relations entre classes dans les langages à objets courants. Pour mettre en œuvre un tel système, il nous a fallu modéliser la notion de relation. Ces relations sont posées entre des classes. Nous nous sommes donc également intéressés à celles-ci. Enfin, les relations et les classes sont généralement associés à un langage précis. Nous avons donc défini un concept de langage.

Nous appelons *concept-relation* l'entité représentant un type de relation : un concept-relation est donc une méta-relation. Par exemple, nous évoquons parfois les célèbres concept-relations `inherits-of` et `is-client-of`. La description d'un concept-relation représente l'objectif principal de cet article, cependant pour mieux appréhender la totalité du contexte, il est nécessaire de dire quelques mots sur les deux autres concepts importants du modèle : le *concept-description* et le *concept-langage*.

Un concept-description permet de définir la notion de classe telle qu'elle existe dans les langages à objets. Notre objectif est de pouvoir faire coexister plusieurs

[2] Une des approches actuelles pour réduire le fossé entre méthodes de conception et langages de programmation est constituée par les patrons de conceptions (*design patterns*).

concepts-descriptions à la sémantique différente, par exemple pour prendre en compte les langages comme `Java` (où nous décrivons notamment les notions d'interface et de classe comme deux concepts-descriptions). Chaque concept-description définit un ensemble de concepts-relations avec lesquels il est compatible et gère leurs interactions.

Le concept-langage est une notion importante et simple. Il modélise, comme son nom l'indique à l'évidence, un langage. Chaque langage est constitué en particulier d'un ensemble de concepts-descriptions et d'un ensemble de concepts-relations, chacun étant compatible avec au moins un des concepts-descriptions sélectionnés[3].

Pour illustrer cet article, nous avons imaginé une relation entre classes qui n'existe pas actuellement dans les langages à objets courants : la relation `is-a-view-of`. Nous définissons cette relation au fur et à mesure de la présentation des paramètres qui caractérisent l'ensemble des concept-relations, et nous discuterons le choix de leur valeur. Nous présentons cette relation plutôt que les traditionnels héritage et clientèle dans le but de montrer la possibilité de créer de nouvelles relations. Toutefois, le lecteur intéressé par ces relations classiques pourra se référer à [CAP 00a, CAP 00c]. Nous ne discutons pas dans cet article de la sémantique opérationnelle définie dans les actions.

2.3. *Travaux connexes*

De nombreux travaux ont été menés sur les relations entre classes ou entre objets et classes, même s'ils n'utilisent pas tous un niveau méta. On peut remarquer parmi eux des travaux sur la classification [CHE 99], sur la modélisation de la relation d'instanciation [RIV 97], sur des relations de version [TAL 94, OUS 99] ou de vue [COU 99, BEL 99], sur l'héritage [DUC 95, MEY 97], sur la délégation [BAR 98], etc.

Nous présentons ici quelques uns des travaux orientés vers la représentation de méta-information pour la modélisation de concepts objets.

MOF [OMG 97] dont les initiales correspondent à *Meta-Object Facility* a été spécifié par l'OMG pour permettre la gestion d'informations méta. La généralité de l'ensemble de concepts fournis permet d'utiliser MOF pour la description d'autres systèmes de types (exemples : UML, C++, etc.). Par rapport à MOF, nous offrons un cadre plus structuré et complet pour décrire précisément une relation entre classes ; le concept d'*association* qu'il propose est plus rudimentaire.

---

[3] Désormais, pour simplifier le discours nous emploierons le terme habituel de *classe* et non plus celui de concept-description.

Le système FLO [DER 95, DER 96] est basé sur le langage Lisp, il permet de définir des dépendances et des contraintes sémantiques entre les instances de classe qui sont définies par des liens décrits en dehors de la classe. Notre approche est un peu différente puisqu'elle s'appuie sur le fait que toute relation entre objets est décrite par une relation entre classes.

Les méthodes de conception doivent permettre la définition de liens sémantiques entre les entités. L'approche proposée par MECANO [GIR 91] propose à l'utilisateur non pas de définir une relation d'héritage entre deux classes mais plutôt de spécifier les usages qu'il compte faire de l'héritage (spécialisation, implémentation, fusion, adaptation, etc.) ou de la délégation (communication, utilisation, etc.). Cependant la liste des types de relation entre entités et celle des types d'entités sont figées dans la méthode.

Les patrons de conception peuvent aussi être considérés comme une approche pour décrire les relations entre classes. En effet, un des effets bénéfiques de cette approche est d'utiliser des combinaisons des mécanismes d'héritage et de clientèle associées à la réalisation de classes *ad hoc* pour décrire des relations plus spécifiques.

3. Paramètres de concept-relation

Nous présentons donc ci-dessous les paramètres valables pour tout concept-relation en en donnant une courte description. Nous préciserons dans la section 4 la valeur qu'ils prendraient pour deux exemples de concept-relation de vue. Nous avons omis quelques paramètres décrits dans le modèle car ceux-ci ne présentaient pas vraiment d'intérêt pour la compréhension de notre approche.

3.1. *Caractéristiques générales*

— `Name` C'est simplement le nom du concept-relation.
— `Kind` Ce paramètre est important, il définit le genre principal de la relation :
- importation inter-classes (`import`) ou
- utilisation inter-classes (`use`) ou
- entre classe(s) et objet(s) (`class-object`) ou
- entre objets (`objects`).

Une relation peut donc être une importation telle l'héritage, mais une relation peut également décrire une utilisation: la clientèle en est un exemple ou une liaison entre une classe et un objet (telle l'instanciation), voire même une liaison entre objets. Précisons que le modèle OFL se focalise sur les relations entre classes, donc beaucoup moins sur les deux dernières formes.

— `Cardinality` Il exprime la cardinalité du concept-relation sous la forme *1-n* qui signifie que les relations issues de ce concept-relation peuvent être créées entre une classe source de la relation et *1* à *n* classes cibles (*n* est ∞ ou un entier naturel supérieur ou égal à *1*). Par exemple, un concept-relation d'héritage simple aura une cardinalité *1-1* (chaque classe ne peut hériter que d'une seule classe) alors que, pour un héritage multiple, elle sera de *1-∞* (chaque classe peut hériter de plusieurs classes). Nous pouvons ainsi limiter la multiplicité d'un héritage (à *1-3* par exemple) ce qui revient par exemple à normaliser la programmation au sein d'une équipe de développement ou d'une entreprise[4].
— `Circularity` Il exprime la possibilité de créer des cycles avec des relations issues de ce concept-relation : `allowed` signifie que des cycles sont permis, `forbidden` qu'ils sont interdits. Par exemple, l'héritage et la clientèle non référencée[5] interdisent les cycles alors que la clientèle les permet.
— `Repetition` Il indique si une répétition de classe est permise dans les classes sources et dans les classes cibles (exemple : héritage répété) des relations issues de ce concept-relation. C'est une paire de valeurs `allowed` ou `forbidden`.
— `Symmetry` Il indique si le concept-relation est symétrique. Un concept-relation est symétrique s'il fournit lui-même une relation sémantique symétrique. Il est possible d'imaginer ainsi un concept-relation `is-a-kind-of` pour lequel la classe-cible et la classe-source sont les termes d'une relation symétrique.
— `Opposite` Il indique, s'il existe, le concept-relation inverse. Un concept-relation est inverse d'un autre (et vice versa), s'ils décrivent une sémantique inverse. Par exemple, un concept-relation de spécialisation est inverse d'un concept-relation de généralisation. Ces définitions entraînent le fait suivant : un concept-relation symétrique est son propre inverse.

3.2. *Gestion du polymorphisme*

— `Polymorphism_direction` (spécifique aux relations d'importation) Il indique si le polymorphisme (la capacité, pour un objet, d'être considéré selon plusieurs classes) est autorisé par le concept-relation et, si c'est le cas, dans quel(s) sens il est possible. Prenons plusieurs exemples : la clientèle interdit le polymorphisme (none), la spécialisation le permet dans le sens source vers cible (up), la généralisation dans le sens inverse (down) et nous pouvons imaginer un concept-relation de version qui l'autorise dans les deux sens (both).
— `Polymorphism` (spécifique aux relations d'importation) Ce paramètre ne prend de sens que si `Polymorphism_direction` est différent de none. Il précise si le polymorphisme sur les méthodes et sur les attributs (c'est donc une paire de valeurs) doit se faire comme un masquage (comme c'est le cas pour les

[4] Nous avons choisi de concevoir tous nos concept-relations avec une cardinalité *1-1* ou *1-n* mais le modèle supporte également les cardinalités de type *m-n*. Cela peut par exemple être utile pour concevoir un concept-relation décrivant l'association en UML.
[5] Clientèle expansée en `Eiffel` ou clientèle primitive en `Java`.

attributs en `Java` : `hiding`) ou comme une redéfinition (cas des méthodes, toujours en `Java` : `overriding`).
— `Primitive_variance` (spécifique aux relations d'importation) Ce paramètre signale le type de variance de la relation par un triplet de valeurs, la première pour les paramètres de méthode, la seconde pour les résultats de fonction, la dernière pour les attributs. Chaque membre du triplet peut donc être :

- `covariant` : Le type des paramètres de méthode (ou résultats de fonction ou attributs, cette parenthèse étant valable pour les autres explications) de la classe-source doit être identique à ou être une source (pour le même concept-relation) du type des paramètres de méthode correspondants dans la classe-cible.
- `contravariant` : Le type des paramètres de méthode de la classe-source doit être identique à ou être une cible (pour le même concept-relation) du type des paramètres de méthode correspondants dans la classe-cible.
- `nonvariant` : Le type des paramètres de méthode de la classe-source doit être identique à celui des paramètres de méthode correspondants dans la classe-cible.
- `non_applicable` : Aucune contrainte ne s'applique entre le type des paramètres de méthode de la classe-source et celui des attributs correspondants dans la classe-cible.

— `Assertion_variance` (spécifique aux relations d'importation) Ce paramètre signale le type de variance de la relation pour les assertions. Ils est en fait composé de trois valeurs, une première pour les invariants, une deuxième pour les préconditions et une dernière pour les postconditions. Chacune des valeurs peut être :

- `weakened` : L'assertion de la classe-source doit être identique à ou plus large que celle de la classe-cible. C'est-à-dire que la première doit être impliquée par la seconde.
- `strengthened` : L'assertion de la classe-source doit être identique à ou plus restreinte que celle de la classe-cible. C'est-à-dire que la première doit impliquer la seconde.
- `unchanged` : L'assertion de la classe-source doit être identique à celle de la classe-cible. C'est-à-dire qu'elles doivent être équivalentes.
- `non_applicable` : Aucune contrainte ne s'applique entre l'assertion de la classe-source et celle de la classe-cible.

3.3. *Gestion de l'accès et du partage*

— `Direct_access` Ce paramètre permet de préciser si la relation offre un accès direct aux primitives de la classe cible[6] dans la classe source. Si un accès direct est présent pour une primitive et que cet accès ne présente aucune ambiguïté (cas de la présence d'un autre accès direct ou d'une primitive locale homonyme), alors

[6] La classe source est celle qui déclare la relation, la classe cible celle qui est utilisée ou importée. Dans l'héritage, l'héritière est la source alors que l'héritée est la cible.

l'utilisation de la primitive distante peut se faire comme si elle était locale. Naturellement, les relations d'importation privilégient souvent les accès directs. Les valeurs possibles pour ce paramètre sont `obligatory`, `allowed` et `forbidden`.
— `Indirect_access` Ce paramètre est similaire au précédent mais pour la gestion des accès indirects. Par accès indirect, nous entendons la nécessité de nommer explicitement la classe-cible ou l'une de ses instances pour accéder à la primitive. Par exemple, si `Direct_access` et `Indirect_access` valent `allowed`, deux types d'accès sont possibles. S'ils sont tous deux à `forbidden`, la primitive de la classe-cible est inaccessible. Les relations d'utilisation, par définition, ont plutôt tendance à mettre en œuvre des accès indirects.
— `Dependence` (spécifique aux relations d'utilisation) Ce paramètre signale si les instances de la classe-cible sont dépendantes (`dependent`) ou non (`independent`) de celle de la classe-source. Le fait d'être dépendant signifie que l'instance de la classe-cible n'est accessible qu'au travers de l'instance de la classe-source et ne peut survivre à cette dernière.
— `Sharing` (spécifique aux relations d'utilisation) Il indique si les instances de la classe-cible sont spécifiques (`specific`) à une instance de la classe-source ou si elles sont partagées (`shared`) entre toutes les instances de la classe-source (c'est l'idée de primitive de classe).
— `Read_accessor` (spécifique aux relations d'utilisation) Ce paramètre spécifie si l'accès en lecture aux attributs peut se faire directement (`optional`) ou s'il est nécessaire de passer par des accesseurs (`obligatory`).
— `Write_accessor` (spécifique aux relations d'utilisation) C'est le pendant du précédent, pour la modification des attributs : l'affectation d'un attribut est-elle possible directement (`optional`) ou exclusivement par un accesseur (`obligatory`) ?

3.4. *Gestion de l'adaptation des primitives*

— `Removing` Ce paramètre indique si la suppression de primitive est permise (`allowed`) ou interdite (`forbidden`) au travers de cette relation.
— `Renaming` Ce paramètre précise si le renommage de primitive est autorisé (`allowed`) ou interdit (`forbidden`) au travers de cette relation.
— `Redefining` (plutôt adapté aux relations d'importation) Ce paramètre indique si la redéfinition de primitive est autorisée ou interdite au travers de cette relation. Au contraire des précédents, ce paramètre est multi-valué. On peut donner la valeur `allowed` ou `forbidden` pour autoriser ou interdire la redéfinition des assertions, de la signature, du corps et des qualifieurs (visibilité, protection, constance, …) de la primitive.
— `Hiding` Ce paramètre est présent pour notifier la possibilité ou l'interdiction de masquer une primitive — et non supprimer (cf. `Removing`), elle peut être démasquée (cf. `Showing`) par une autre relation.

— `Showing` C'est l'inverse de `Hiding`. Il exprime s'il est possible (`allowed`) ou non (`forbidden`) de rendre de nouveau visible une primitive préalablement masquée. S'il est possible de masquer une primitive, il est intéressant de pourvoir la démasquer.
— `Abstracting` (plutôt adapté aux relations d'importation) Ce paramètre permet de signifier s'il est possible de rendre une primitive abstraite au travers de cette relation (alors qu'elle est concrète dans la classe-source).
— `Effecting` (plutôt adapté aux relations d'importation) C'est le pendant d'`Abstracting`, il décrit l'autorisation (`allowed`) ou non (`forbidden`) de rendre concrète une primitive (qui est abstraite dans la classe-source).

3.5. *Interdépendance des paramètres*

Il est à noter que la valeur d'un paramètre peut être liée à la valeur d'un autre. Par exemple, il nous semble normal qu'`Opposite` soit dépendant de `Symmetry` puisqu'une relation symétrique est son propre opposé. De la même façon, il existe des combinaisons de valeurs de paramètres auxquelles nous n'avons pas trouvé de sens. Cependant nous n'avons pas souhaité mettre en place un mécanisme de contrôle de la cohérence du système de paramètres pour laisser toute liberté au méta-programmeur. Celui-ci pourra donc implanter, au prix de la redéfinition des actions (sémantique opérationnelle) associées aux paramètres, des relations que nous n'avons pas prévues. Il est toutefois envisageable qu'un tel contrôle soit ajouté au modèle `OFL` et activé à la demande du méta-programmeur.

4. Concepts-relations de vue

La relation de vue permet de définir une vue des instances d'une classe, dans un sens proche des vues en base de données. Une vue permet de présenter des données sous une forme différente de celle proposée par leur classe d'origine. On peut par exemple cacher certaines parties des objets ou en présenter d'autres sous un nouvel angle. Nous discutons dans cette section 4 l'influence de la valeur des paramètres de concept-relation sur l'usage du lien.

Une relation de vue peut être considérée de deux manières, soit comme une relation d'utilisation, soit comme une relation d'importation. Le paramètre `Name` aura alors comme valeur la chaîne de caractères `"`*`Is-a-uview-of`*`"`, pour l'utilisation, et `"`*`Is-a-iview-of`*`"` pour l'importation.

4.1. *Choix du type de relation*

Nous commençons à discuter ici de trois paramètres : `Kind`, `Direct_access` et `Indirect_access`. Pour expliquer l'intérêt de ces paramètres nous décrivons, comme nous l'avons vu précédemment, deux concepts-relations de vue. Nous allons mettre en évidence leurs différences en montrant comment nous les utilisons. Pour le premier, *`Is-a-uview-of`*, le paramètre `Kind` a la valeur `use` tandis que pour le second, *`Is-a-iview-of`*, la valeur est `import` (cf. Figure 1).

Le choix d'une relation d'utilisation a pour effet d'introduire dans la classe source une référence à une instance de la classe cible (par exemple à travers un attribut), ce qui n'est pas le cas dans une relation d'importation. Par ailleurs chacune des sortes de relation implique des paramètres spécifiques qui mettent en évidence de nouvelles différences.

Nous avons positionné le paramètre `Direct_access` du concept-relation `Is-a-uview-of` à `allowed`, ce qui est inhabituel pour une relation d'utilisation. La procédure `test1` de la classe `GARAGE` décrite dans la Figure 2 montre l'intérêt d'une telle démarche avec le deuxième appel à `print`. On pourrait même rendre l'utilisation de `CAR-VIEW-USE` similaire à celle de `CAR-VIEW-IMPORT` en cachant l'attribut `aCar`, par une directive `private`[7] par exemple, ce qui aurait pour effet d'invalider le premier appel à `print`.

```
class CAR-VIEW-USE
aCar : Is-a-uview-of CAR
     rename maximal-price as price
     remove cost-price
end Is-a-uview-of

function promotional-price : REAL {
     return 0.9 * price()
}
end class
```

```
class CAR-VIEW-IMPORT
Is-a-iview-of CAR
     rename maximal-price as
                    car-price
     remove cost-price
end Is-a-iview-of

function promotional-price : REAL {
     return 0.9 * car-price()
}
end class
```

```
class GARAGE
car1 : client-of CAR-VIEW-USE
car2 : client-of CAR-VIEW-IMPORT
oneCar : client-of CAR

procedure test1() {…}
procedure test2() {…}
procedure test3() {…}

end class
```

Figure 1. Illustration de relations d'utilisation et importation[8]

---

[7] Comme en `Java`.
[8] *client-of* est la relation de clientèle classique que l'on trouve dans les langages à objets.

```
procedure test1 {
    print(car1.aCar.maximal-price);
    // appel de la primitive maximal-price de l'attribut aCar de l'objet car1
    print(car1.price);
    // appel de la primitive price de l'objet car1 :
    // intérêt du paramètre Direct_access qui permet d'appeler maximal-price
    // avec le nom price et sans passer par aCar
    print(car2.car-price);
    // maximal-price a été importé sous le nom car-price
    // (effet similaire à l'héritage avec renommage)
}
```

Figure 2. *Illustration du paramètre* `Direct_access`

De manière plus traditionnelle, `Direct_access` est à `allowed` et `Indirect_access` est à `forbidden` pour le concept-relation `Is-a-iview-of` et `Indirect_access` est à `allowed` dans le concept-relation `Is-a-uview-of`.

Nous concevons nos deux relations de vue comme des relations simples, la valeur de `Cardinality` est donc *1-1*. Notamment dans le cas d'une relation d'importation (`Is-a-iview-of`), il pourrait être intéressant de modéliser une relation multiple. Cela permettrait entre autres de simuler l'opération de jointure des bases de données relationnelles. Par contre, il faudrait alors gérer tous les problèmes habituels des relations multiples, tels que les conflits de noms. Pour les relations d'utilisation (*`Is-a-uview-of`*), on préférera plusieurs relations simples (plusieurs attributs) à une unique relation multiple (un seul attribut avec plusieurs relations !).

Quel que soit le cas, simple ou multiple, le paramètre `Repetition` prend la valeur `<forbidden, forbidden>`. Il est à notre avis inutile qu'une même classe soit deux fois la cible d'une relation de vue. Cela n'empêche évidemment pas d'établir une autre relation d'importation ou d'utilisation vers une classe déjà cible d'une relation de vue si les paramètres de cette autre relation le permettent.

Une classe ne peut être une vue d'une de ses vues, donc le paramètre `Circularity` a la valeur `forbidden`. De plus, nos concepts-relations de vue ne sont pas symétriques (`Symmetry` est donc `false`) et n'ont pas d'opposé (`Opposite` est égal à `none`).

4.2. *Paramètres spécifiques à l'importation*

Pour le concept-relation `Is-a-iview-of` et pour lui seulement, les paramètres suivants prennent un sens.

Si Polymorphism_direction prend la valeur up, cela signifiera que toutes les instances de la vue sont aussi instances de la classe cible. S'il est égal à down, c'est que l'on souhaite que toute instance de la classe cible soit également instance de la vue. Si l'on choisit la valeur both, les deux comportements sont cumulés, ce qui signifie que les extensions de la classe cible et de la vue sont identiques. Enfin la valeur none indiquerait que les deux extensions sont indépendantes. Indépendamment de la sémantique qu'on veut donner à la vue, si celle-ci ne fait que supprimer des primitives, il est alors acceptable de proposer la valeur down. Si on ne fait qu'en ajouter, *up* est plus adapté. Si on enlève et n'ajoute rien alors both est raisonnable (ce dernier cas n'a souvent d'intérêt que dans une relation multiple). Dans le contexte des applications de base de données relationnelles, il n'est pas possible d'ajouter un attribut[9] dans une vue. On peut par contre en supprimer, donc la valeur la plus raisonnable dans ce contexte serait down. Précisons dans ce cas que l'ajout de primitives dans la vue est généralement interdit. S'il est autorisé, sa gestion est dévolue au méta-programmeur. On remarque enfin que l'usage de l'héritage pour simuler un mécanisme de vue dans les langages à objets présente le net inconvénient d'avoir un polymorphisme up, ce qui est inadapté pour une relation de vue. Nous choisissons pour notre concept-relation Is-a-iview-of la valeur down.

Pour le paramètre Polymorphism le choix de l'une ou l'autre des valeurs permet seulement d'adopter une politique de programmation différente, c'est pourquoi, dans un souci d'orthogonalité, nous avons choisi la valeur <*overriding*, *overriding*>.

Puisque la vue contient un sous-ensemble des primitives de sa classe cible (à l'inverse de l'héritage), il semble intéressant de choisir la contra-variance pour les paramètres de méthode, les résultats de fonction et les attributs. La valeur du paramètre Primitive_variance serait donc <contravariant, contravariant, contravariant>.

Pour le paramètre Assertion_variance, compte tenu de la possibilité de suppression de primitives, une solution simple consiste à choisir la valeur <non_applicable, non_applicable, non_applicable>. Une autre solution possible consiste à tenir compte du polymorphisme inverse par rapport à l'héritage et donc à choisir la valeur <strengthened, weakened, weakened>[10].

Le contenu de la procédure test2 de la classe GARAGE de la Figure 3 contient un certain nombre d'expressions valides et invalides.

[9] Sauf s'il est calculé (et donc non modifiable).
[10] Naturellement les deux dernières valeurs du triplet n'ont d'effet que si la redéfinition des pré et postconditions est possible (cf. section 4.4).

```
procedure test2 {
    oneCar ← car2; // non valide : le polymorphisme est descendant
    car2 ← oneCar; // valide
    // l'instance de CAR est vue comme une CAR-VIEW-IMPORT
    print(car2.car-price); print(oneCar.maximal-price);
    // la primitive maximal-price a été renommée dans CAR-VIEW-IMPORT
    print(oneCar.cost-price); print(car2.cost-price);
    // seul le premier appel est valide : cost-price a été supprimée dans
    // CAR-VIEW-IMPORT
}
```

Figure 3. Illustration du sens du polymorphisme

4.3. *Paramètres spécifiques à l'utilisation*

Dans le cas d'un concept-relation comme `is-a-uview-of`, le paramètre `Dependence` doit être fixé à `independent` car il n'y a naturellement aucune raison à ce que l'instance de `CAR` disparaisse dans le cas ou sa vue est supprimée. La valeur à associer au paramètre `Sharing` doit être `specific` car chaque instance de la vue concerne en général une instance et une seule de la classe cible. Pour les paramètres concernant les accesseurs, le fait de permettre ou non l'accès direct en lecture ou en modification à l'attribut est tout à fait corrélé à la philosophie du langage que l'on veut réaliser. Dans notre cas on l'autorise en lecture et la valeur du paramètre `Read_accessor` est donc `optional`, mais on l'interdit en modification et la valeur du paramètre `Write_accessor` est `obligatory`.

```
procedure test3 {
    car1.price ← 100000; // non valide : les mises à jour doivent se faire à travers un accesseur
    print(car1.price); // la consultation d'attribut ne nécessite pas le passage par un accesseur
}
```

Figure 4. *Illustration des paramètres concernant les accesseurs*

4.4. *Clauses d'adaptation*

Une clause d'adaptation est un mécanisme permettant de modifier (redéfinir, renommer, etc.) une primitive importée. Dans `OFL`, nous modélisons, toujours à l'aide de paramètres, le fait que l'on peut ou non faire usage d'une telle clause.

Les clauses d'adaptation représentent un aspect important de la définition d'un concept-relation de vue, comme ceux présentés dans la section 4.1. D'elles vont dépendre la valeur d'un certain nombre de paramètres des sections 4.1, 4.2 et 4.3. Le choix des clauses d'adaptation influence directement l'usage que l'on va pouvoir faire des relations. Par exemple, si on souhaite construire des vues qui représentent des projections (au sens des bases de données relationnelles), on va naturellement autoriser la suppression de primitives et le paramètre Removing aura la valeur allowed (voir Figure 1). De même on autorise le renommage des primitives afin d'adapter au plus juste la signification de la primitive ou de résoudre les conflits[11] ; la valeur de Renaming est donc allowed.

Pour le paramètre Redefining, il n'y a pas plus d'obligation de choisir l'une ou l'autre des valeurs, par contre l'autorisation de redéfinir des primitives va donner un sens aux valeurs choisies pour des paramètres comme Primitive_variance ou *Assertion_variance* pour un concept-relation d'importation. Nos concepts-relations autorisent la redéfinition et la valeur de Redefining est <allowed, allowed, allowed, allowed>.

Les paramètres Hiding et Showing sont particulièrement intéressants lorsqu'on introduit la possibilité d'avoir des vues de vues et donc d'avoir un arbre de relations Is-a-iview-of ou Is-a-uview-of. Dans notre cas, nous autorisons le masquage et le démasquage et les paramètres Hiding et Showing sont tous les deux à allowed.

Enfin, nous choisissons d'interdire l'abstraction et la réalisation de primitive. Les paramètres Abstracting et Effecting prennent la valeur forbidden.

5. Bénéfice de notre approche

Nous permettons au méta-programmeur d'enrichir un langage de programmation par de nouvelles relations plus précises. Dans notre exemple, nous proposons deux relations de vues spécifiques (Is-a-iview-of et Is-a-uview-of) qui pourront, par la suite, être utilisées durant la phase de programmation. Désormais, il n'est donc plus nécessaire de détourner l'usage de l'héritage ou de la clientèle afin de simuler une vue. Bien que le modèle ne bride pas le méta-programmeur, il faut éviter la multiplication de liens trop proches. Ainsi, pour profiter de notre approche sans ajouter de confusion, chaque langage gérant des vues ne devrait contenir qu'un seul des deux concepts-relations mentionnés ci-dessus.

S'il est directement visible qu'une relation de vue est posée quand Is-a-iview-of est utilisée, il est souvent difficile et parfois impossible de déterminer que le mécanisme d'héritage est mis en œuvre dans le but d'implanter une relation de vue. Ainsi, grâce à l'association d'une sémantique spécifique adaptée à l'usage de chaque relation (par exemple, une vue, un sous-type, une version, etc.), le code obtenu est nettement plus précis et donc plus lisible. Cela permet aussi d'activer des contrôles et des actions automatiques qui facilitent la maintenance des applications.

[11] L'autorisation du renommage est particulièrement intéressante pour un concept-relation multiple ou dans le cas où il y aurait d'autres concepts-relations d'importation.

Le fait d'ajouter des relations à un langage augmente l'expressivité de ce dernier et permet ainsi de réduire le fossé qui est généralement présent entre la modélisation et l'implantation d'une application. En effet, les méthodes de conception actuelles utilisent souvent des relations inter-classes qui n'existent pas dans les langages de programmation. Notre approche permet d'implanter ces relations dans le langage.

6. Perspectives et conclusion

Dans cet article, nous avons voulu mettre en évidence l'expressivité des paramètres qui permettent de mettre en œuvre différents concepts-relations afin d'améliorer la qualité des logiciels. La diversité des usages des vues, notamment dans les bases de données, nous a permis de montrer que le jeu de paramètres proposé est utile pour décrire les relations et pour ne pas demander trop de travail au méta-programmeur.

Nous travaillons actuellement dans deux directions : d'une part, la validation du modèle, et, éventuellement, son extension. L'objectif est de prendre en compte, par la définition de bibliothèques de concepts-relations, la diversité des besoins en matière de gestion de versions et de vues. D'autre part, nous sommes en train d'étudier `MOF`, `XMI` ou `XML-Schema` afin de représenter notre modèle à partir de ces standards et ainsi de rendre plus facile l'échange de nos spécifications et leur implantation dans un prototype.

Une définition du modèle `OFL` en `UML` et une implantation de cette définition en `Java` sont également en cours de réalisation.

7. Références

[ARN 98] ARNOLD K. and GOSLING J. The Java Programming Language. 2nd Edition, Sun Microsystems, The Java Series.

[BAR 98] BARDOU D. Étude des langages à prototypes, du mécanisme de délégation, et de son rapport à la notion de point de vue. Thèse de Doctorat de l'Université Montpellier II.

[BEL 99] BELLAHSENE Z. Updates and Object-Generating Views in `OBDS`. Rapport de Recherche LIRMM n° 99130.

[CAP 00a] CAPOUILLEZ A., CHIGNOLI R., CRESCENZO P. et LAHIRE P. Gestion des objets persistants grâce aux liens entre classe. Conférence OCM'2000 (Objets, Composants, Modèles), Nantes, France.

[CAP 00b] CAPOUILLEZ A., CHIGNOLI R., CRESCENZO P. and LAHIRE P. Towards a More Suitable Class Hierarchy for Persistent Object Management. International Workshop *Objects and Classification: a Natural Convergence*, ECOOP'2000 (14st European Conference on Object-Oriented Programming), Sophia Antipolis and Cannes, France.

[CAP 00c] CAPOUILLEZ A., CHIGNOLI R., CRESCENZO P. and LAHIRE P. How to Improve Persistent-Object Management using Relationship Information? WOON'2000 (4st International Conference *The White Object Oriented Nights*), Saint-Petersburg, Russia.

[CHE 99] Chevalier N., Dao M., Dony C., Huchard M., Leblanc H. and Libourel T. An Environment for Building and Maintaining Class Hierarchies. ECOOP'99: Workshop *Object-Oriented Architectural Evolution*, Lisbonne, Portugal.

[COU 99] COULONDRE S. and LIBOUREL T. Viewpoints Handling in an Object Model with Criterium-Based Classes. DEXA'99 (10th International Conference on Database and Expert Systems Applications), Florence, Italy.

[DER 95] DERY A.-M., DUCASSE S. and FORNARINO M. A Reflective Model for First Class Dependencies. OOPSLA'95 (10th Object-Oriented Programming Systems Languages and Applications), Austin.

[DER 96] DERY A.-M., DUCASSE S. and FORNARINO M. Object and Dependency Oriented Programming in FLO. ISMIS'96 (9th International Symposium on Methodologies for Intelligent Systems).

[DES 94] DESFRAY P. Object Engineering, the Fourth Dimension. Addison-Wesley Publishing Co.

[DUC 95] DUCOURNAU R., HABIB M., HUCHARD M., MUGNIER M.-L. et NAPOLI A. Le point sur l'héritage multiple. Technique et Science Informatiques.

[GIR 91] GIROD X. MECANO : une Méthode et un Environnement de Construction d'ApplicatioNs par Objets. Thèse de Doctorat de l'Université de Grenoble 1.

[KIC 91] KICZALES G., DES RIVIÈRES J. and BOBROW D. The Art of the Metaobject Protocol. MIT Press.

[MEY 97] MEYER B. Object-Oriented Software Construction. 2nd Edition. Prentice Hall.

[OMG 97] OMG. MOF Specifications: Joint Revised Submission OMG Document ad/97-08-14.

[OUS 99] OUSSALAH C., editor. Génie objet : analyse et conception de l'évolution. Hermès.

[RIV 97] RIVARD F. Évolution du comportement des objets dans les langages à classes réflexifs. Thèse de Doctorat de l'Université de Nantes.

[STR 97] STROUSTRUP B. The C++ Programming Language. Addison-Wesley Publishing Co.

[TAL 94] TALENS G. Gestion des objets simples et composites. Thèse de Doctorat de l'Université Montpellier II.