L'héritage du comportement en programmation logique par objets

Macaire Ngomo **Jean-Pierre Pécuchet** **Abdenbi Drissi-Talbi**

PSI/LIR-INSA, ROUEN
Équipe Heuristique et Informatique Avancée
B.P. 08
76130 MONT SAINT AIGNAN
Tel: (33) 35 52 83 40 Fax: (33) 35 52 83 32
{ngomo,pecuchet,drissi}@lirinsa.insa-rouen.fr

Résumé : L'héritage multiple peut être une source de conflits dans les langages à objets. Généralement, les stratégies par défaut utilisées dans les langages à objets consistent à linéariser le graphe d'héritage. Comme le souligne Masini, les stratégies linéaires ont l'inconvénient majeur de systématiser le traitement de chaque conflit sans tenir compte de la nature du problème à résoudre. Dans ce papier, nous sommes intéressés au problème de l'héritage du comportement en programmation logique par objets et nous proposons une solution non-linéaire basée sur la résolution non-déterministe. Elle permet d'explorer toutes les méthodes possibles et de considérer tous les points de vue d'un objet. Pour amender le comportement par défaut qui n'est pas toujours souhaité, nous proposons une solution basée sur la désignation explicite multiple.

Mots clés : Programmation logique, programmation par objets, héritage du comportement, stratégie de recherche non-déterministe, désignation explicite multiple.

Abstract : Multiple inheritance can be a source of conflicts in the object-oriented programming languages. Generally the default strategies used in the object-oriented programming languages consists of a linearization of the inheritance graph. As Masini underlines, linear strategies have a great inconvenience : they systematize the processing of each conflict without taking into consideration nature of the problem to solve. In this paper we take interest in the inheritance of the inheritance behaviour in object-oriented and logic programming and we propose a non linear solution based on the non determinist resolve. This one enables us to explore all the available methods and still consider all the differents points of view of an object. In order to correct the default behaviour which is not always requested, we suggest a solution based on the multiple explicit designation.

Key words : Logic programming, object-oriented programming, inheritance of the behaviour, non determinist research strategies, multiple explicit designation.

1. Introduction

L'héritage du comportement est une synthèse des conséquences de la relation d'héritage au niveau des méthodes [Royer 91]. Il décrit l'évolution du comportement des classes au travers des liens d'héritage définis par l'utilisateur. Intuitivement, on veut que le comportement des instances d'une classe soit défini par la composition de ses méthodes locales et de celles de ses sur-classes. Le mode de composition doit cependant être défini. Pour ce faire, on fait face à deux choix de conception : la sémantique de l'héritage et l'ordre dans lequel seront considérées les classes. Concernant l'ordre de parcours du graphe d'héritage, les stratégies par défaut utilisées dans les langages à objets sont généralement basées sur la linéarisation du graphe d'héritage. La linéarisation permet de résoudre les conflits d'héritage. Elle est, pour l'instant du moins, le meilleur compromis. De l'avis de beaucoup de chercheurs, elle est actuellement la seule technique acceptable [Ducournau 92]. Cependant, plusieurs raisons nous amènent à proposer, en programmation logique par objets, une solution par défaut non-linéaire. Premièrement, la linéarisation a l'inconvénient majeur de systématiser le traitement de chaque conflit, sans tenir compte de la sémantique de la nature du problème à traiter [Masini 89]. Deuxièmement, il n'existe sans doute pas de stratégie linéaire universelle, idéale, satisfaisante dans tous les cas [Masini 89] [Ducournau 92]. Enfin, le non-déterminisme de la programmation logique permet de résoudre implicitement les conflits d'héritage de manière non-déterministe.

Nous partons du principe que l'héritage multiple sert surtout à représenter des points de vue différents que l'on peut avoir sur un objet [Chouraqui 88] [Dugerdil 86] [Dugerdil 88] [Dugerdil 91] [Ferber 90] et utilisons la possibilité qu'offre Prolog d'explorer, par retour arrière, toutes les alternatives possibles, pour considérer, en cas d'ambiguïtés restantes, toutes les méthodes candidates, c'est-à-dire les méthodes en conflit. Cependant, dans certaines situations, la résolution implicite n'est pas toujours attendue. En effet, il arrive qu'un objet ait plusieurs points de vue et qu'on ne s'intéresse qu'à certains points de vue particulier. La solution par défaut non-déterministe ne permet pas de répondre à cette attente. La résolution implicite peut aussi cacher des erreurs de programmation [Amiel 95]. Pour amender ce comportement par défaut qui n'est pas toujours attendu, cette stratégie sera complétée de la désignation explicite multiple et d'une linéarisation.

Le reste de cet article s'articule autour de trois sections principales. Après une discussion sur la sémantique de l'héritage (section 2) et sur les différentes stratégies par défaut de parcours du graphe utilisés actuellement (section 3), nous présentons (section 4) notre approche non-déterministe d'héritage de méthodes.

2. Sémantique de l'héritage

La définition traditionnelle de l'héritage suppose une sémantique non-monotone dans la composition des différentes classes héritées. Ceci signifie que la redéfinition d'une méthode dans une sous-classe remplace celle déjà donnée dans une surclasse. Ainsi, si une instance de cette classe reçoit un message auquel on doit répondre par application de cette méthode, ce sera la définition de la sous-classe qui sera considérée. En pratique, on fournit souvent un mécanisme pour passer outre cela. C'est par exemple l'envoi de message à *super* en Smalltalk-80 [Goldberg 83], qui désigne explicitement la définition dans les classes au-dessus. La notion d'héritage peut être comprise ici comme un mécanisme de copie virtuelle non monotone [Ferber 90]. En effet, tout se passe comme si toute la surclasse était recopiée dans la sous-classe (*mécanisme de copie*), même si cela n'est pas effectivement implémenté de cette manière (***copie virtuelle***), et la recopie ne s'effectue que pour les informations qui ne sont pas définies au niveau de la sous-classe (***copie non monotone***). De ce fait, la définition du mécanisme d'héritage peut s'exprimer ainsi [Ferber 90] :

- une classe A qui hérite d'une classe B dispose implicitement de tous les attributs et de toutes les méthodes définies dans B ;

- les attributs et les méthodes définies dans A sont prioritaires par rapport aux attributs et méthodes de même nom définis dans B.

Gallaire [Gallaire 86] Léonardi et Mello [Léonardi 88] proposent, en programmation par objets, de remplacer la sémantique non-monotone par une sémantique monotone où, par retour arrière, on explorerait toutes les définitions, des sous-classes vers les surclasses. Pour illustrer cette procédure, considérons la figure 1 ci-dessous. Dans le contexte de cette figure, m étant une méthode définie au niveau des classes C_k, C_j et C_i, le traitement opérationnel du message "O <- m" consiste à exécuter, par retour arrière, des sous-classes vers les surclasses, toutes les définitions possibles de la méthode m.

(1) O <- m/C_k[1]
ou (2) O <- m/C_j
ou (3) O <- m/C_i.

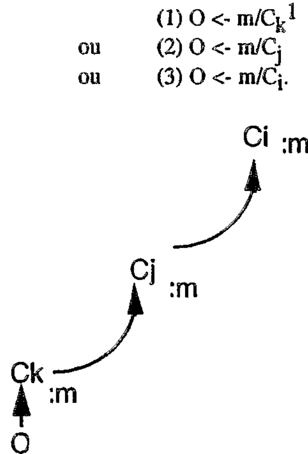

Figure 1 : Sémantique monotonie de l'héritage.

Cette approche, basée sur le non-déterminisme vertical[2], a séduit plus d'un chercheurs. Cependant, elle pose un problème majeur. En effet, si l'héritage est utilisé pour construire sur la base d'une autre classe, ce qui supporte l'idée d'une sémantique monotone, il est aussi utilisé pour *différencier* des comportements. Il arrive souvent que l'on modélise une entité par une classe en se disant : *mes instances seront comme celles de telle classe (**héritage**) sauf pour tel ou tel comportement (**différenciation**)*. Le nouveau comportement introduit par la redéfinition des certaines méthodes dans la sous-classe masque doit dans ce cas masqué l'ancien. Cette dernière interprétation nécessite donc une sémantique non-monotone. Cette nécessité d'avoir une façon de réintroduire une sémantique non-monotone de l'héritage a d'ailleurs conduit certains concepteurs à proposer une nouvelle forme de coupe-choix ("cut" en Prolog) pour empêcher le retour arrière sur les anciennes définitions dans les classes héritées. Il appelle ce coupe-choix "*cut_inheritance*" [Gallaire 86] [Gandilhon 87]. POL [Gallaire 86] et ESP [Chikayama 84] sont deux exemples de langages qui utilisent une sémantique monotone de l'héritage, avec cependant quelques exceptions.
POL est un sur-ensemble de Prolog ne modifiant pas le comportement d'un programme Prolog qui n'utiliserait pas les concepts objets. Il supporte les objets, classes et méthodes avec la possibilité de l'héritage multiple. Les concepts d'héritage et de méthode sont adaptés

[1] m/Ci désigne la méthode m définie dans la classe Ci.

[2] Le graphe d'héritage est parcouru verticalement de bas en haut, des sous-classes vers les classes les plus hautes.

au fonctionnement de Prolog. L'utilisation des variables et du non-déterminisme se retrouve dans la structure objet proposée. POL autorise l'héritage multiple, avec une sémantique monotone, sans proposer aucune solution pour résoudre les conflits. Il offre cependant un moyen explicite permettant de masquer les définitions des méthodes redéfinies dans les sous-classes. Trois opérateurs sont utilisés pour définir les méthodes :

Classe **with**	Nom_de_methode :- Corps_de_methode
Classe **withdefault**	Nom_de_methode :- Corps_de_methode
Classe **withdeterministic**	Nom_de_methode :- Corps_de_methode

Ces opérateur permettent de créer des méthodes qui autorisent ou non des appels déterministes. L'opérateur withdeterministc permet ainsi d'introduire explicitement une sémantique non-monotone de l'héritage. Par contre, la définition d'une méthode dans une sous-classe à l'aide de l'opérateur with ne masque pas les définitions précédentes. L'exécution d'une méthode définie à l'aide de l'opérateur with revient à considérer, du bas vers le haut, toutes ses définitions, de la plus spécialisée à la plus générale. Il n'y a pas masquage.
ESP est un langage développé au Japon dans le cadre du projet "Machines de cinquième génération". Il se dit être à KL0 ce que Flavors [Moon 86] est à LISP [Steele 90]. Il est similaire à Prolog. Il combine la sémantique monotone avec des démons (before, after, etc.). KL0 est le langage machine de la machine séquentielle d'inférence japonaise.

3. Sur les stratégies de recherche des méthodes

Autant l'héritage simple est cohérent, autant l'héritage multiple présente des difficultés. En effet, il existe une possibilité de conflit entre deux surclasses, si elles possèdent des champs ou des méthodes ayant un nom identique. Dans ce cas, quelle est l'information dont doit hériter la sous-classe? A l'heure actuelle, ce problème ne possède pas encore de solution générale, les concepteurs de langages ayant simplement apporté des solutions partielles. Plusieurs solutions ont été proposées pour résoudre ce problème [Ferber 90] :

- Considérer que tout conflit de nom est une erreur ; c'est la voie qu'ont choisie les langages de programmation compilés destinés au génie logiciel [Meyer 90]. Le compilateur prévient qu'il existe un conflit et laisse au programmeur le soin d'apporter les modifications nécessaires.
- Offrir la possibilité de combiner les caractéristiques des champs en conflit, soit au travers d'une théorie de l'héritage, soit en proposant des mécanismes de combinaison. Cette approche n'est utilisée que dans les langages de représentation de connaissances ([Dugerdil 86] [Dugerdil 87] [Dugerdil 88] [Dugerdil 91] [Chouraqui 88], etc.).
- Proposer de définir un ordre de priorité entre les surclasses d'une classe (Bobrown 88] [DeMichiel 87] [Ducournau 87], [Ducournau 89] [Ducournau 91] [[Ducournau 92] [Ducournau 94] [Royer 90a] Royer 90b] [Royer 91a] [Royer 91b], [Masini 89] [Meyer 86] [Meyer 87] [Meyer 90], [Rival 89], etc.). Par exemple, dans le cadre de la figure 5, on dira qu'un appareil électromécanique est plus un appareil mécanique qu'un appareil électronique. C'est la solution adoptée par les langages tels que Eiffel [Meyer 86] [Meyer 87] [Meyer 90], CLOS [Bobrown 88] [DeMichiel 87] et un certains nombre de langages en intelligence artificielle.
- Partir du principe que l'héritage multiple sert surtout à représenter des points de vue différents que l'on peut avoir sur un objet ([Dugerdil 86] [Dugerdil 87] [Dugerdil 88] [Dugerdil 91] [Chouraqui 88], etc.). Par exemple, on peut dire qu'un appareil électromécanique possède un aspect électronique et un aspect mécanique (cf. Figure 2).

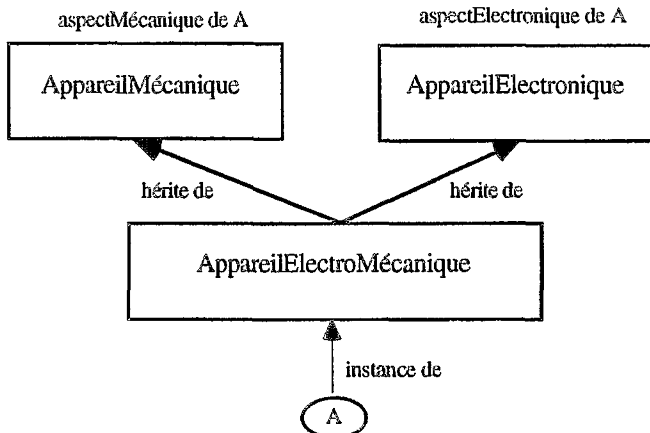

Figure 2 : Différents aspects d'un appareil électromécanique.

Dans cette section, nous nous intéressons aux solutions par défaut proposées pour parcourir le graphe d'héritage. Dans beaucoup de langages, la solution consiste à définir un ordre dans lequel ces surclasses seront examinées pour trouver la définition de la méthode qui sera utilisée pour répondre à un message. Ceci revient classiquement à la définition d'un ordre total ou partiel dans le graphe d'héritage ou dans le sous-graphe dont la source est la classe courante. Plusieurs ordres ont été proposés. Nous pouvons citer :

La stratégie des Flavors: Elle repose sur deux règles [Moon 86] [Royer 91] [Masini 89] :

(1) La classe est prioritaire sur toutes ses surclasses.
(2) L'ordre des surclasses est compatible avec l'ordre de précédente locale.

L'algorithme[3] est le suivant : on construit une première liste débutant par la classe et contenant les ordres obtenus pour chaque surclasse directe de la classe. Les ordres des surclasses directes sont unis en respectant la multiplicité locale. On utilise une dernière règle pour éliminer les occurrences multiples des surclasses. Cette règle est : on ne garde que l'occurrence la plus proche de la classe et qui respecte (1) et (2). Une variante de cet algorithme est utilisée par LOOPS (Lisp Object and data Oriented Programming System) [Bobrown 83], la dernière règle est de prendre l'occurrence la plus proche de Object.

Parcours en profondeur avec inversion : L'algorithme est un parcours en profondeur d'abord, effectué sur le graphe de multiplicité inverse, avec récupération de l'ordre inverse de sortie de pile. Cet algorithme est nommé depth-first out [Royer 91]. Il existe un autre algorithme de ce type dans [Ducournau 89], appelé P1, et dont une version simplifiée est décrite dans [Masini 89].

Le tri topologique de CLOS (ou Pclos) **:** Cette stratégie se base sur l'extraction d'un ordre total à l'aide d'un tri topologique sur le graphe d'héritage auquel on a ajouté les relations de précédences. Trois cas peuvent se produire [Royer 91] :

- Un seul ordre.

---

[3] Les informations de [Moon 86] sont incomplètes à ce sujet, la version présente est extraite de [Masini 89] [Royer 91].

- Plusieurs ordres sont possibles, auquel cas on en choisit un suivant une règle qui consiste à remonter le plus possible sur la branche en cours de parcours.
- Pas d'ordre possible, le graphe des précédences a un circuit, il y a génération d'une erreur.

Une autre variante de cette approche existe dans [Ducournau 89].

En largeur d'abord : La technique est cette fois de faire un parcours en largeur d'abord. A chaque niveau l'ordre des noeuds est celui défini par la multiplicité des noeuds du niveau inférieur. Ce parcours détermine en fait le plus court chemin d'héritage pour la méthode. Le parcours en largeur appartient à une classe de stratégies dont les propriétés ne sont pas bien définies.

Il existe d'autres stratégies de recherche que nous ne décrivons pas dans ce papier, par manque de place et surtout parce que c'est pas l'objet de notre étude (l'algorithme Pami[4] [Ducournau 92] qui est une sorte de synthèse en P1 et Pclos ; la stratégie statique développée dans le modèle ObjScheme [Royer 89] : elle est équivalente à un parcours en profondeur d'abord respectant la multiplicité locale ; etc.). Leur importance est de taille dans la résolution des conflits d'héritage. La linéarisation est, pour l'instant du moins, le meilleur compromis. Pour certains, elle est actuellement la seule technique acceptable [Ducournau 92]. Cependant, plusieurs raisons nous amènent à proposer, pour la programmation logique par objets, une approche par défaut non-linéaire.

D'une part, il n'est sans doute pas possible de construire une stratégie linéaire universelle de parcours du graphe d'héritage satisfaisante dans tous les cas. L'exemple suivant, tiré de [Ducournau 92][5] (figure 3), distingue deux algorithmes P1 [Ducournau 89] et Pclos [Bobrown 88] qui produisent respectivement {C7, C5, C3, C6, C2, C4, C1} et {C7, C5, C6, C2, C4, C3, C1} comme listes de précédence.

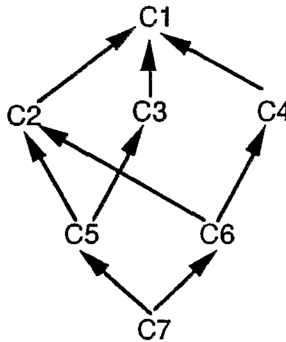

Figure 3 : un exemple discriminant.

D'autre part, les stratégies linéaires ont l'inconvénient majeur de systématiser le traitement de chaque conflit, sans tenir compte de la nature du problème à résoudre. Comme le souligne [Masini 89], la résolution des conflits ne peut être fiable que si elle prend en compte les connaissances liées à l'application. Seul le concepteur, ou un expert du domaine, possède la compétence requise. Dans ce cas, appliquer systématiquement une solution par défaut ne peut jamais régler correctement chaque cas. Partant de cette analyse,

---

[4] Pami est une synthèse de P1 et Pclos qui représente une avancée significative par rapport à P1 et Pclos.

[5] L'exemple est bien tiré de [Ducourneau 92]. Nous avons seulement renommé les classes.

nous adoptons une stratégie non-linéaire basée sur le non-déterminisme de la programmation logique.

4. Une approche non-linéaire de l'héritage du comportement

4.1 Stratégie par défaut non-déterministe

La stratégie par défaut que nous présentons retient la sémantique non-monotone de l'héritage et, afin d'utiliser la multiplicité des différents cas de figures, utilise un ordre partiel avec retour arrière, dans le cas des ambiguïtés restantes. Elle est basée sur la possibilité qu'offre Prolog d'explorer, par retour arrière, les différentes alternatives. Dans les approches classiques, un choix est fait, sans possibilité de retour arrière. Dans cette approche, le retour arrière permet l'application de toutes les méthodes en conflits, en supposant que chacun méthode décrit le comportement d'un des points de vue de objet. Le non-déterminisme est ici utilisé, non pas verticalement (figure 4a), mais horizontalement (figure 4b).

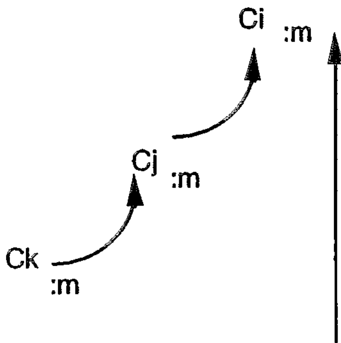

Figure 4.a: backtracking vertical

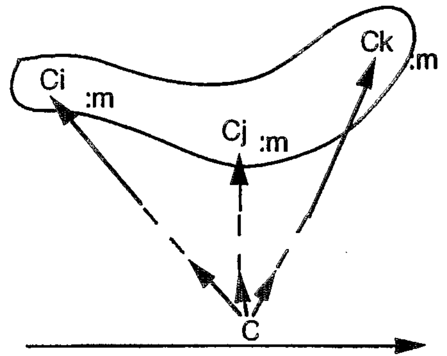

Figure 4.b: backtracking horizontal

Nous définissons un C-sous-graphe d'un graphe G, le sous-graphe de G construit à partir de C en remontant le graphe G et qui contient tous les chemins de G qui partent de C. Une classe S est un majorant minimal (ou une borne inférieure) pour une propriété P dans le C-sous-graphe si elle ne possède aucune sous-classe dans le C-sous-graphe du graphe d'héritage qui définit la propriété P. L'ensemble de ces majorants minimaux est alors appelé la borne inférieure dans le C-sous-graphe relative à M. Par exemple, dans le contexte du graphe de la figure 4.b, Ci, Cj et Ck sont des majorants minimaux pour la méthode m.
Ainsi, notre stratégie repose sur un principe simple : pour exécuter une méthode M à partir d'une classe C, seules les méthodes de même nom associées aux majorants minimaux pour M dans le C-sous-graphe seront appliquées. Lors d'un envoi de message, l'algorithme détermine l'ensemble des minorants maximaux correspondant, et déclenche, de manière non-déterministe, toutes les méthodes associées à la borne inférieure de C relative à la méthode considérée, en "backtrackant" horizontalement sur l'ensemble des éléments de cette borne inférieure. Illustrons ceci par quelques exemples concrets. Considérons pour cela l'exemple classique du graphe ci-dessous (cf. figure 5). Dans cet exemple, la méthode :departement permet de déterminer selon le cas le département d'étude ou de travail d'une personne. La méthode :calendrier_de_vacances retourne quant à elle le calendrier des vacances scolaires d'un étudiant (dans la classe Etudiant), le calendrier annuel des congés d'un salarié (dans la classe Salarié).
Dans la figure ci-dessus, Jacques est un ATER. Il pourra être interrogé à propos de son département ou de son calendrier de vacances.

Jacques <- departement(D)
Jacques <- calendrier_de_vacances(P)

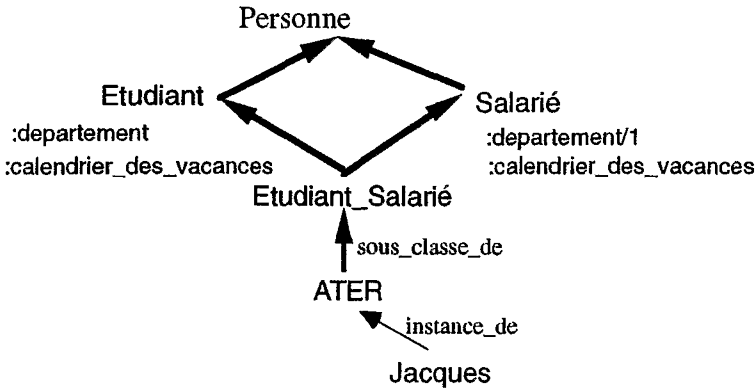

Figure 5: Etudiant et Salarié : quel définition de *departement* Jacques utilise-t-il au niveau de ATER?

Avec une approche linéaire on une solution par défaut correspondant à la classe la plus prioritaire (Etudiant ou Salarié, selon la stratégie utilisée). Dans l'approche que nous proposons, étant donné le caractère ambigu de ces questions, le système répondra en proposant toutes les solutions disponibles :

(1) le département d'étude et
le calendrier des vacances scolaires de Jacques en tant qu'Etudiant.
(2) le département de travail et
le calendrier des vacances de Jacques en tant que Salarié.

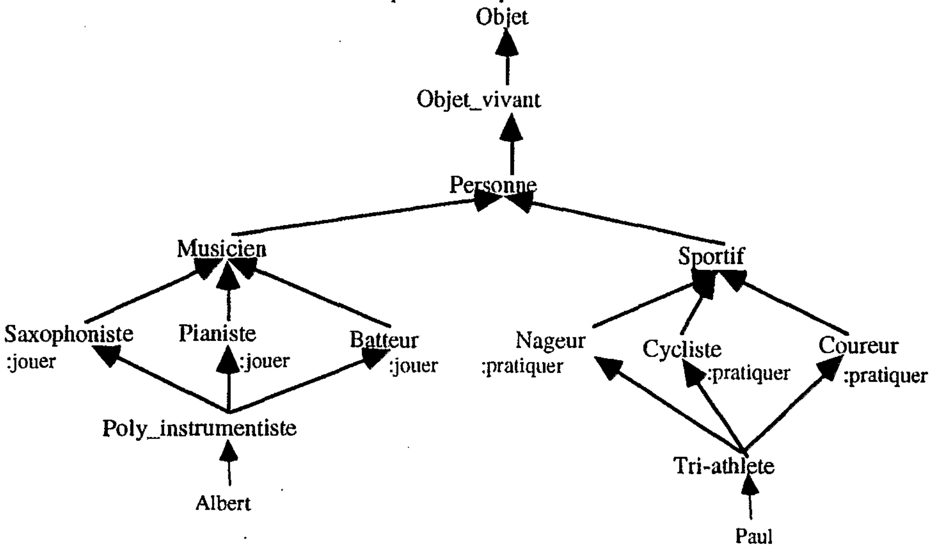

Figure 6 : Les différents points de vue de Albert et de Paul.

On peut multiplier des exemples de ce genre. De tels cas de figures sont couramment rencontrés dans notre vie quotidienne. Par exemple, dans le contexte de la figure 6, Albert est un Tri-instrumentiste (Violoniste, Pianiste, Guitariste), et Paul un Tri-athlèthe. Albert pourra être appelé à "jouer une partition" de musique et Paul à "pratiquer un type d'athlétisme".

Albert <- jouer.
Paul <- pratiquer

Par défaut, un envoi de message active toutes les méthodes possibles, en profitant des retours arrière effectués par l'interprète Prolog dans sa recherche exhaustive des solutions à une requête.

4.2 La désignation explicite multiple

Dans certaines situations, la résolution par défaut n'est pas toujours attendue. En particulier, elle ne permet pas de fixer dynamiquement certains points de objets. Par exemple, si nous ne sommes intéressés que par la vie estudiantine de Jacques.

"Jacques, quel est ton département d'étude?"

Avec la seul stratégie par défaut présentée en 4.1, le système ne peut pas répondre à de telles attentes. Pour amender ce comportement par défaut qui n'est pas toujours attendu, nous proposons de compléter la stratégie non-déterministe par une désignation explicite multiple. La désignation explicite est un mécanisme utilisé dans les langages à objets, en complément de la stratégie par défaut. Elle consiste à désigner explicitement une classe d'appartenance d'une propriété. Elle ne peut difficilement servir de stratégie par défaut d'un langage dans la mesure où elle n'intervient souvent que dans des cas exceptionnels. Une utilisation abusive de ce mécanisme ne pourrait être supportable pour l'utilisateur [Ducournau 92]. Ce mécanisme permet de lever certains conflits à la main, en désignant explicitement, en cas d'ambiguïté, une des classes en conflits, supposée être la plus adaptée pour une situation donnée. Nous l'introduisons ici par le qualificatif *as* dont la syntaxe est la suivante :

OBJ as CLASS.

Sémantiquement, elle consiste à voir un objet comme une instance *directe* de la classe désignée. Grâce à ce mécanisme, nous pouvons maintenant préciser nos questions :

Jacques as Etudiant <- departement(D).
Paul as Cycliste <- pratiquer.
Albert as Violoniste <- jouer

Désignation explicite multiple

La désignation explicite telle que nous venons de l'introduire ne permet pas d'effectuer des choix multiples. La désignation explicite multiple permet d'effectuer de tels choix, en désignant plusieurs classes d'un objet, tout en respectant les mêmes règles que dans le cas d'une désignation simple. La syntaxe devient alors la suivante :

OBJ as [$CLASS_1$, ..., $CLASS_n$].

Ceci permet d'écrire par exemple :

Albert as [Saxophoniste,Pianiste] <- jouer.

Comme le montre ces exemples, la désignation explicite apporte au modèle de traitement d'héritage plus de souplesse et permet au programmeur un plus grand contrôle sur le mécanisme d'héritage multiple.

Désignation explicite et principe du masquage

La désignation explicite est un mécanisme efficace mais sa sémantique opérationnelle n'est toujours pas claire dans certains langages à objets. En particulier, que ce passe-t-il si la méthode recherchée est redéfinie dans l'une des sous-classes de la classe désignée? Par exemple, dans le contexte de la figure 1, que se passe-t-il si un programmeur écrire :

O as Cj <- m?

Pour un traitement plus rigoureux, un mécanisme de contrôle doit permettre à la fois de vérifier l'appartenance de l'objet courant à la classe désignée et le respect du principe de masquage.

Désignation explicite et parcours du graphe d'héritage

La désignation explicite est aussi un moyen permettant de réduire la complexité des méthodes de parcours du graphe d'héritage, puisqu'elle consiste à effectuer un saut vers la classe désignée et par conséquent à réduire le graphe de recherche de la méthode, ce qui évite de visiter inutilement toutes les classes intermédiaires.

Désignation incomplète

Une désignation peut être incomplète. Par exemple,

Jacques as EtudiantSalarié <- departement(D)

Etant donné que cette méthode ne possède pas la méthode departement, la recherche continuera au niveau de ses surclasses en utilisant la stratégie pas défaut. Un exemple de désignation complète serait :

Jacques as Etudiant <- departement(D).

Dans ce cas, la propriété recherche ne peut pas être atteinte directement. Il faut la classe désignée ne permet pas d'atteindre directement la propriété recherchée. Dans ce cas, la recherche continue au niveau des surclasses de la classe désignée en utilisant la stratégie par défaut.

5. Conclusion

Nous venons d'aborder un aspect de l'héritage qui est celui de l'héritage du comportement. Au lieu d'une "bonne linéarisation" comme stratégie par défaut du système, comme dans beaucoup de langages à objets ou dans le "portrait robot de l'héritage multiple idéal" proposé dans [Ducournau 92], nous proposons, pour la programmation logique par objets, une stratégie par défaut non-déterministe. Notre choix est motivé par plusieurs raisons. Premièrement, il n'existe sans doute pas une stratégie linéaire universelle, idéale, satisfaisante dans tous les cas. Deuxièmement, la linéarisation a l'inconvénient majeur de systématiser le traitement de chaque conflit, sans tenir compte de la nature du problème à traiter. Enfin, la possibilité qu'offre Prolog d'explorer, par retour arrière, toutes les alternatives possibles, permet, en cas d'ambiguïtés, de considérer tous les points de vue d'un objet (évitant ainsi de linéariser le graphe d'héritage). Le but est surtout de permettre au programmeur de pouvoir exprimer clairement ses intentions, au lieu qu'une solution par

défaut lui soit imposée par le système. Cette approche retient la sémantique non-monotone, la plus courante en programmation par objets. L'ordre d'exploration des classes en conflit n'est pas fixé statiquement, mais déterminé à l'appel d'une méthode. La désignation explicie multiple permet d'amender le comportement par défaut, lorsque celui-ci n'est pas souhaité. Cette solution a été expérimentée dans le langage ObjTL [Ngomo 95a] [Ngomo 95b] [Ngomo 95c]. ObjTL étend Prolog vers la programmation par objets. Il est développée en sur-couche de Delphia-Prolog [Delphia 92] et tourne sur une station UNIX. Nous avons déjà expérimenté en ObjTL, la possibilité d'introduire, en plus des stratégies existantes, une stratégie linéaire (algorithme P1 par exemple). Cependant, elle est utilisée explicitement en introduisant une notation supplémentaire, grâce à la définition d'un nouvel opérateur :

Jacques :: departement(D).

Nos travaux dans ce sens évoluent vers la définition par l'utilisateur de ses propres stratégies. Cela pourra se faire par exemple en ajoutant un paramètre supplémentaire dans le protocole d'envoi de message et en prévoyant une primitive système qui prendra en entrée le nom de la stratégie choisie lors de l'envoi d'un message. Le protocole d'envoi de message contiendra alors, en plus des éléments habituels (receveur, sélecteur, arguments), le nom de la stratégie choisie. Par exemple,

Jacques :: ('Pclos')@departement(D).

Bibliographie

[Amiel 95] E. Amiel et E. Dujardin "Un outil d'aide à la résolution explicite des ambiguïtés des multi-méthodes". LMO'95, Nancy 1995, pp.131-152.

[Bobrow 88] Bobrow D.G., DeMichiel L.G., Gabriel R.P., Keene S.E., Kiczales G., & Moon. D.A. "Special Issue, Common Lisp Object System Specification, X3J13 Document 88-002R". ACM SIGPLAN Notices, 23, Sep. 1988.

[Chicayama 84] Chicayama T. "Unique Features of ESP." Pro. Int'l Conf. on Fifth Generation Computer Syst.", (1984), pp.292-298.

[Chouraqui 88] Chouraqui E., Dugerdil P. "Conflict solving in a frame-like multiple inheritance system". In Proc. of the 8th ECAL. Munich, Germany, p.226-231, 1988.

[DeMichiel 87] DeMichiel L. G., Gabriel R. P. "The Common Lisp Object System : an overview", ECOOP'87.

[Ducournau 87] Ducournau R., Habib M. "On some algorithms for multiple inheritance in Object oriented programming". ECOOP'87.

[Ducournau 89] Ducournau R. , Habib M. "La multiplicité de l'héritage dans les langages à objets". TSI, vol 8 n°1, janv. 1989.

[Ducournau 91] Ducournau R., Habib M. "Masking and conflicts, or to inheritance is not own", in [Viaregio 91].

[Ducournau 92] Ducournau R., Habib M., Huchard M., Mugnier M. & Napoli A. "L'héritage multiple dans tous ses états". Rapport Technique, LIRMM N° 92-021, Montrouge, Juil. 1992.

[Ducournau 94] Ducournau R., Habib M., Huchard M., Mugnier M.L. "Proposal for a monotonic multiple inheritance linearization". In Proc. OOPSLA, 1994.

[Dugerdil 86] Dugerdil P. "A propos des mécanismes d'héritage dans les langages orientés objets.", Actes du 2e CIIA (Marseille, France), pp.67-77., 1986.

[Dugerdil 87] Dugerdil P. "Les mécanismes d'héritage d'OBJLOG : vertical et sélectif multiple avec point de vue.", Actes du 6ème CARFIA, pp. 259-273, Antibes, 1987.

[Dugerdil 88] Dugerdil P. "Contribution à l'étude de la représentation des connaissances fondée sur les objets. Le langage OBJLOG". Thèse de l'Université d'Aix-Marseille II, 1988.

[Dugerdil 91] Dugerdil P. "Inheritance mechanisms in the OBJLOG langage: multiple selective and multiple vertical with points of view" in [Viaregio 91].

[Ferber 90] Ferber J. "Conception et Programmation par Objets". Hermes 1990.

[Gallaire 86] Gallaire H. "Merging Objects and Logic Programming: Relational Semantics, Performance and Standarization". In Proc. AAAI'86, pp. 754-758, Philadelphia, Pensylrania, 1986.
[Gandilhon 87] Gandilhon T. "Proposition d'une extension objet minimale pour Prolog.", Actes du séminaire de Programmation en Logique, Trégastel (mai 1987), pp. 483-506.
[Goldberg 83] Goldberg A. & Robson D. "Smalltalk-80: The language and its implementation". Addison-Wesley, 1983.
[Léonardi 88] Leonardi, L. & Mello, P. "Combining Logic and Object-Oriented Programming Language Paradigms". Actes 21 st Hawaii Int'l Conf. on Sys. Sc., pp. 376-385, 1988
[Malenfant 90] Malenfant J. "Conception et Implantation d'un langage de programmation intégrant trois paradigmes: la programmation logique, la programmation par objets et la programmation répartie". Thèse de PhD, Univ. de Montréal, 1990.
[Malenfant 92] Malenfant J. "Architecture méta-réflexives en programmation logique par objets". JFPL 92, pp. 253-267, 1992.
[Masini 89] Masini G., Napoli A., Colnet D., Léonard D. & Tombre K. "Les langages à objets". InterEditions, Paris, 1989.
[Meyer 86] Meyer B. "Eiffel : un langage et une méthodologie pour le génie logiciel.", Interactive Software Engineering, Inc., 1986.
[Meyer 87] Meyer B. "Eiffel : Programming for reusability and extendibility.", ACM SIGPLAN Notices, 22(2):85-94, 1987.
[Meyer 90] Meyer B. Conception et programmation par objets, pour le génie logiciel de qualité.", InterEditions, Paris 1990.
[Moon 86] Moon D. "Object-Oriented Programming with Flavors.", In Proc. of the 1st OOPSLA, pp.1-8, Portland, Oregon, 1986.
[Ngomo 94a] Ngomo M. & Pécuchet J-P. "Contribution à l'étude de l'association des paradigmes de programmation logique et programmation par objets". Poster RJC-IA'94, p. 314, sept. 1994 Marseille, France.
[Ngomo 94b] Ngomo M. & Pécuchet J-P. "Contribution à l'étude de l'association des paradigmes de programmation logique et programmation par objets". Actes du 2ème Colloque Africain sur la Recherche en Informatique, CARI'94, pp.879-893, Ouagadougou, Burkina Faso, oct. 1994.
[Ngomo 95a] Ngomo M., Pécuchet J-P. & Drissi-Talbi A. "Une approche déclarative et non-déterministe de la programmation logique par objets mutables". Actes des 4èmes Journées Francophones de Programmation Logique et Journées d'étude Programmation par contraintes et applications industrielles, Prototype JFPLC'95, pp.391-396, Dijon, 1995, France.
[Ngomo 95b] Ngomo M. , Pécuchet J-P. & Drissi-Talbi A. "La gestion de l'héritage multiple en ObjTL". RPO'95 dans les Actes des 15èmes Journées Internationales IA'95, pp.261-270, Montpellier 1995, France.
[Ngomo 95c] Ngomo M., Pécuchet J-P., Drissi-Talbi A. "Intégration des paradigmes de programmation logique et de programmation par objets : une approche déclarative et non-déterministe". Actes du 2ème Congrès bienal de l'Association Française des Sciences et Technologies de l'Information et des Systèmes, AFCET - Technologie Objet - 95, pp.85-94, Toulouse 1995, France.
[Rival 89] Rival I. Ed "Algorithms and order", Nato ASI serie C, volume 225, Kluwer Acad. Publ. (1989).
[Royer 89] Royer J-C. "Un modèle de programmation par objets en SCHEME - application à la synthèse d'images 2D, Thèse d'université, Janv. 1989 Bordeaux I.
[Royer 90a] Royer J-C. Extensions Orientées Objets de SCHEME.", Journées des Langages Applicatifs, La Rochelle, 18-19 Janv. 1990.
[Royer 90b] Royer C. "Un modèle pour l'héritage multiple.", Journées Groplan, 24-26 Janv 1990 Nice.
[Royer 91a] Royer J-C. "A propos des concepts de CLOS". Journées Francophones des Langages Applicatifs, janv. 1991, France.
[Royer 91b] Royer J-C. "Une étude de l'héritage dans le cadre d'une utilisation plus stricte en programmation". Rapport de recherche LIST 91-09, 27 Juin 1991.
[Stroustrup 89] Stroustrup B. "Le Langage C++". InterEditions 1989.
[Steele 90] Steele G. L. "Common Lisp the language" second edition, Digital Press, 1990.