Propositions pour un nouveau modèle d'objets dans les langages à prototypes

Daniel Bardou
LIRMM, 161, rue Ada, 34392 Montpellier Cedex 5
(bardou@lirmm.fr)

Christophe Dony
LIRMM, 161, rue Ada, 34392 Montpellier Cedex 5
(dony@lirmm.fr)

Résumé

L'article discute et décrit la première spécification d'un nouveau modèle d'objets pour un langage à prototypes (ou pour tout langage à objets sans classe), où l'on définit un objet comme un ensemble de morceaux.

Ce modèle, tout en préservant ce qui fait l'essence de ces langages (la description d'un système par des objets concrets et le partage de valeurs et de comportements entre objets via les mécanismes de clonage et de délégation), propose des solutions cohérentes aux deux problèmes récurrents suivants :

- la gestion de la factorisation des connaissances communes aux objets d'une même famille de clones (ou problème des traits). Ce problème est résolu en plaçant les comportements communs dans des morceaux partagés,
- la gestion des entités dites "objets morcelés" constituées dans les langages à prototypes existants par un ensemble d'objets reliés par des liens de délégation. Nous faisons de ces ensembles les objets de notre modèle et montrons que leur gestion est intimement liée à la notion de point de vue.

1 Introduction

Les langages à objets sans classes et plus particulièrement les langages dits "à prototypes" sont basés sur la manipulation d'objets concrets par opposition aux langages à classes où la conception et la réalisation des programmes est basée sur la manipulation de concepts.

On trouve sous diverses variantes et avec quelques exceptions les mécanismes fondamentaux de clonage et de délégation dans tous les langages à prototypes. Le mécanisme de délégation permet à un objet B d'être défini comme une extension d'un autre objet A, ce qui revient à faire de A une partie intégrante de l'objet B, les valeurs et les comportements de A devenant alors des valeurs et des comportements de B.

La mise en œuvre de ces mécanismes est encore très diversifiée, chaque système encourage un style de programmation qui lui est propre et il règne un certain flou sémantique sur les "bonnes et mauvaises" utilisations de ces mécanismes. Ainsi deux problèmes sémantiques importants ont été décrits dans [Dony *et al.*, 1992] :

1. Le problème du partage de comportements communs (problème des traits de SELF) est lié au fait que la factorisation (de comportement, de valeurs) est réalisée au niveau des objets concrets et non au niveau des concepts comme dans les langages à classes.

2. Le problème dit des objets morcelés provient de la possibilité, via le mécanisme d'extension précédemment décrit, de créer des entités que l'on ne sait pas adresser, en particulier l'entité constituée par un objet possédant plusieurs extensions.

Nous rappelons plus en détail ces problèmes dans la section 2 de l'article.

Notre problématique a donc été de concevoir un langage intégrant toutes les possibilités d'un langage à prototypes (clonage, délégation, liaison dynamique) mais qui soit sémantiquement cohérent ; i.e. dans lequel d'une part le partage soit réalisé dans tous les cas sans création d'objets incohérents et d'autre part dans lequel les objets morcelés soient pris en compte, ce qui pose plus globalement la question de savoir ce qu'est un objet dans un système à prototypes.

Notre modèle est présenté de façon globale dans la section 3, et le mécanisme d'envoi de message est détaillé dans la section 4. Les travaux connexes sont évoqués dans la section 5.

2 Rappel des mécanismes et des problèmes posés

Nous rappelons brièvement les mécanismes de base des systèmes à prototypes tout en décrivant plus en détail les problèmes évoqués dans l'introduction.

Les langages à prototypes permettent de créer et de manipuler des objets concrets[1], de leur associer des valeurs et des comportements, de leur envoyer des messages. Les objets sont créés soit ex-nihilo (l'utilisateur spécifie totalement les propriétés du nouvel objet) soit en clonant ou en étendant (grâce au lien "parent") des objets existants. Voici des exemples de modèles ou de langages que l'on peut regrouper sous l'appelation "langages à prototypes" : le modèle de Lieberman [Lieberman, 1986] et ObjectLisp [MACL, 1989] l'une de ses implantations, SELF [Chambers *et al.*, 1991; Ungar *et al.*, 1991; Agesen *et al.*, 1993], Kevo [Taivalsaari, 1993] qui ne supporte pas la délégation, Agora [Steyaert, 1994].

Certains langages d'acteurs comme Act1 [Lieberman, 1981] et langages de Frame comme Yafool [Ducournau, 1989] font également partie de cette famille avec des variantes importantes. Dans [Borning, 1986] est présenté une comparaison de l'approche

1. Le terme "objet concret" a été choisi par opposition à celui de "classe". Une classe est la description abstraite d'autres objets (ses instances) alors qu'un objet dans un langage à prototypes contient à la fois la description et les valeurs de ses propriétés.

prototypique avec l'approche adoptée par les langages à classes qui peut également favoriser une meilleure compréhension de la délégation et du clonage.

2.1 Problème de la factorisation des connaissances communes aux objets d'une même famille

Famille de clones et propriétés communes
On manipule fréquemment des objets similaires dans un langage à prototypes. Ces objets sont par exemples des copies d'un même objet initial. Cette copie est réalisée par la primitive de clonage et on parle alors de famille de clones pour désigner l'ensemble des objets ainsi obtenus.

Chacun des clones possède les mêmes propriétés et en particulier le même comportement, mais il n'est pas possible de factoriser ces comportements puisque tous les objets sont autonomes. Rien n'est prévu non plus pour modifier globalement la famille, ajouter une méthode à tous les objets qui la composent par exemple. L'utilisation exclusive du mécanisme de clonage ne suffit donc pas pour gérer le partage "à vie" de propriétés entre objets semblables. Le fait que la délégation soit un mécanisme permettant également de faire du partage entre objets suggère de l'utiliser pour combler cette lacune.

Propriétés de la délégation en rapport avec notre problème
Rappelons tout d'abord brièvement certaines des conséquences de l'existence d'un lien "parent" (support du mécanisme de délégation) entre deux objets, en discutant sur un exemple.

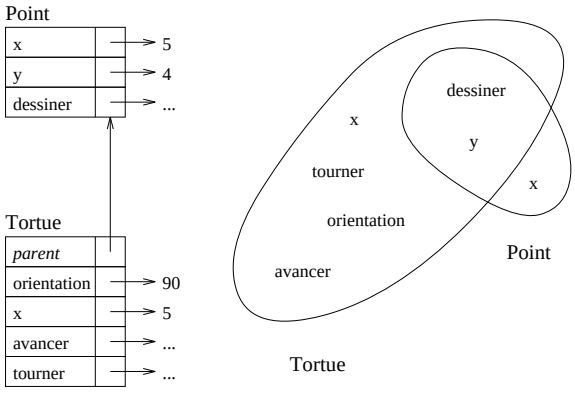

FIG. 1 - *Partage réalisé par la délégation*

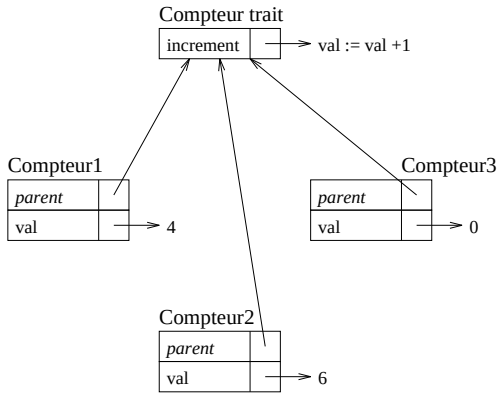

FIG. 2 - *Un "trait" en* SELF

Nous avons représenté un point et une tortue dans un langage à prototypes (Fig. 1). L'objet *Point*, possédant trois propriétés x, y et *dessiner*, est le parent d'un objet *Tortue* pour lequel on a défini quatre propriétés *orientation*, x, *avancer* et *tourner*. Le fait que *Tortue* ait un lien parent vers *Point* permet en fait à *Tortue* de partager les propriétés de *Point* qui n'ont pas été localement redéfinies dans *Tortue*.

Une première interprétation (A) de ce schéma est de considérer que le partage de ces propriétés est total, *Tortue* possède entre autres la propriété y : il peut l'utiliser et la modifier. Si par exemple le message `avancer 10` était envoyé à *Tortue*, étant donné son *orientation* de 90 degrés, la valeur de y (bien que définie en *Point*) serait modifiée en 14.

Une autre interprétation (B), telle que celle adoptée dans Yafool [Ducournau, 1989] par exemple, consiste à interdire la modification d'une propriété de son parent par un objet. Dans notre exemple, cela correspond à ne pas modifier la variable y définie en *Point*, mais à redéfinir y en *Tortue* avec la valeur 14.

La première interprétation (A) est celle qui est la plus généralement adoptée dans les langages à prototypes et notamment SELF. Dans ce cas, l'existence d'un lien parent entre deux objets a pour effet de les rendre totalement dépendants l'un de l'autre au regard des propriétés qu'ils partagent ou pourraient partager, i.e. les propriétés du parent. Cette relation entre fils et parent est une relation "à vie" : elle ne se termine que lorsque l'un des deux objets est détruit ou lorsque le lien parent est supprimé.

Partage de propriétés communes avec la délégation : les traits
Concernant notre problème de factorisation, une solution naturelle consiste donc à regrouper les propriétés devant être partagées par chaque objet d'une famille dans un objet servant en quelque sorte de réservoir partagé : c'est la solution proposée dans le langage SELF dans lequel un tel objet est appelé un "trait". Chaque objet de la famille admet le trait comme parent et hérite donc des propriétés qui y sont définies. Si la factorisation de propriétés est effectivement réalisée, il se pose néammoins les deux problèmes suivants.

- Problème du double statut d'objet concret et de bibliothèque.
  Prenons l'exemple de la représentation de compteurs en SELF (Fig. 2). *Compteur trait* possède une méthode *increment* dont l'exécution modifie une variable *val* non partagée (i.e. non définie dans le trait) de chacun des compteurs. L'envoi du message `increment` à *Compteur trait* provoque donc une exception. Si l'on peut penser à quelques artifices pour éviter le signalement de cette exception, le vrai problème est que *Compteur trait* possède à la fois le statut d'objet concret (puisque c'est un objet du langage) et celui de bibliothèque pour les compteurs. En résumé, un trait est une entité détenant des comportements qui lui sont techniquement applicables (on peut les activer en lui envoyant un message) mais pratiquement inapplicables (exception levée) et par conséquent sémantiquement inappropriés.

- Problème de l'obtention d'un nouveau clone.
  Comme on peut le remarquer dans nos deux précédents exemples, l'une des conséquences de l'existence d'un lien parent entre deux objets est que le parent devient une partie de l'objet délégant : *Tortue* ne serait pas vraiment une tortue sans *Point* comme parent et il en va de même pour *Compteur1* et *Compteur trait*. Pour obtenir une nouvelle tortue, il est judicieux d'effectuer un clonage profond, c'est à dire cloner à la fois *Tortue* et *Point* et de relier les deux objets obtenus par

un lien parent (si l'on ne clonait que *Tortue*, nos deux tortues devraient alors partager le même y). A l'inverse, pour créer un nouveau compteur il ne faut surtout pas cloner *Compteur trait* pour que la factorisation des propriétés qu'il détient reste effective. Une primitive de clonage généraliste ne peut déterminer quelles sont les parties à cloner et celles qui, étant des bibliothèques, ne doivent pas l'être. La seule issue est que l'utilisateur redéfinisse la primitive de clonage pour chaque objet.

2.2 Le problème du statut des objets dans un système à délégation avec partage des valeurs

Le second problème fondamental présent dans les langages à prototypes existants est celui du statut exact d'un objet. Ce problème est implicitement évoqué dans nos deux précédents exemples de la tortue et des compteurs. Un objet du monde réel est en fait représenté par plusieurs objets du langage (une tortue est un ensemble {*Tortue*, *Point*} et *Compteur1* ne serait pas un compteur sans *Compteur trait*). Les objets du langages ne représentent donc dans certains cas que des parties d'objets réels. Les frontières entre objets du langage ne sont pas clairement définies, la tortue peut modifier le point sans que celui-ci soit consulté et inversement. Les objets du monde réel sont en fait éclatés ou morcelés en objets du langage et rien n'existe pour les gérer correctement.

Pour donner une idée des besoins que doit satisfaire cette gestion, considérons un autre exemple réalisé dans un langage à prototypes tel que SELF. On y définit un groupe d'objets représentant chacun un aspect d'une personne, appelons-la Pierre (Fig. 3). L'usage de la délégation qui est fait dans cet exemple est essentiel de par le fait que le caractère permanent du partage qu'elle réalise est pleinement exploité et qu'une telle réalisation est impossible dans un langage à classes. Nous montrons cependant que les langages à prototypes sont à priori inadaptés à la gestion d'un tel ensemble d'objets.

Supposons que l'on ait d'abord voulu représenter Pierre en tant que personne, nous avons d'abord créé l'objet *Personne* avec les propriétés *adresse*, *age*, *nom* et *téléphone*. Puis pour représenter Pierre en tant que sportif, nous avons créé l'objet *Sportif* avec les propriétés *endurance* et *poids* et un lien parent vers *Personne*. Représenter *Pierre-sportif* par ces deux objets nous permet :

- de s'adresser à Pierre en tant que *Pierre-sportif* (possédant *poids*, *endurance*, *adresse*, *age*, *nom* et *téléphone*),
- mais également en tant que *Pierre-personne* (en ignorant ses caractéristiques sportives).

Cette représentation a également pour avantage de permettre l'extension de Pierre selon d'autres critères, comme par exemple en créant l'objet *Cinéphile*, extension de *Personne*, regroupant les caractéristiques *acteurPréféré*, *filmPréféré*, *nombreDeFilms*

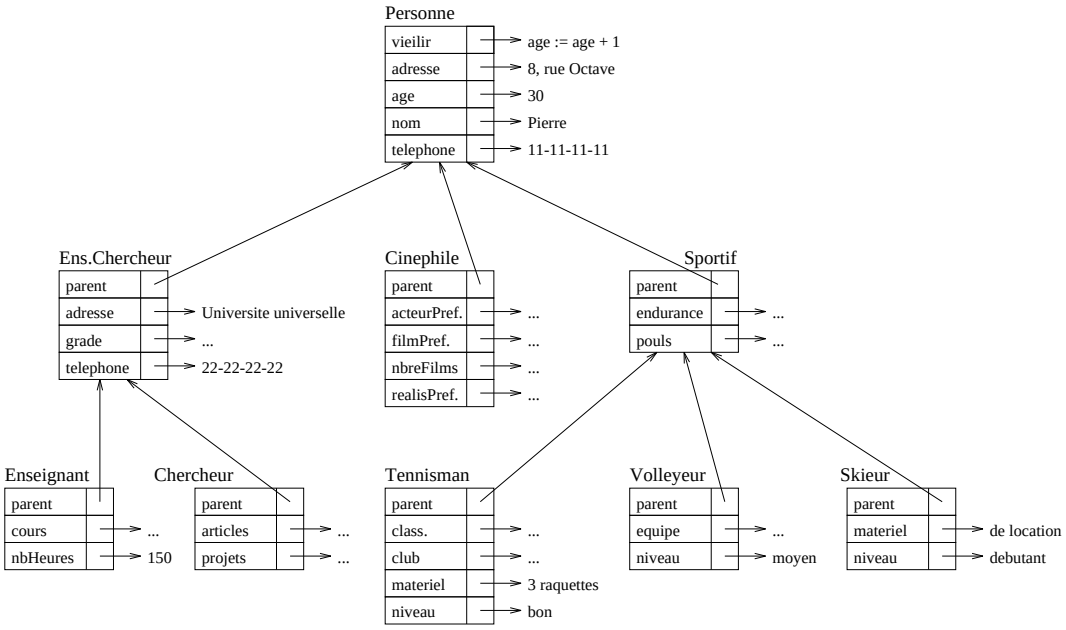

FIG. 3 - *Un objet morcelé en* SELF

et *réalisateurPréféré*. Le fait que les propriétés de Pierre en tant que personne soient partagées par les trois objets nous donne la possibilité de modifier celles-ci en envoyant un message à l'un quelconque de ces trois objets, et cette modification est immédiatement effective non seulement pour *Pierre-personne* mais aussi pour chacune de ses deux extensions. Si par contre nous modifions une propriété caractéristique de l'aspect sportif de Pierre, son *endurance* par exemple, la modification a lieu au niveau de *Sportif* et n'est donc effective que pour *Pierre-sportif*.

Pierre possède alors trois "morceaux" *Personne*, *Sportif* et *Cinéphile* correspondant chacun à un point de vue[2] que l'on peut avoir sur lui, à savoir *Pierre-personne*, *Pierre-sportif* et *Pierre-cinéphile*.

Cet exemple met en évidence le fait que l'ensemble de ces trois "morceaux" représente une entité globale que nous appelons un objet morcelé, et auquel nous ne pouvons pas nous adresser. Comment par exemple demander à Pierre quel est son *acteurPréféré* si l'on ignore que seul *Pierre-cinéphile* détient la réponse? Comment cloner *Pierre-entier* (l'ensemble des trois morceaux)? A qui envoyer le message?

On pourrait penser résoudre le problème en créant un nouvel objet servant de référence pour *Pierre-entier*. Cet objet pourrait être un objet vide de propriétés ayant un lien parent vers chacun des morceaux-feuilles du graphe de délégation. Cette solution ne serait possible que dans un système supportant la délégation multiple et de plus on peut se demander si elle est toujours applicable.

2. Nous utilisons ici "point de vue" au sens commun du terme.

Compliquons quelque peu notre exemple en ajoutant une nouvelle extension à *Personne* redéfinissant les propriétés *adresse* et *téléphone* et ajoutant à Pierre une nouvelle caractéristique *grade*. Nous appelons cette extension *Enseignant-Chercheur* et elle référence *Pierre-enseignant-chercheur*. Pierre est maintenant représenté par quatre morceaux, mais combien a-t-il de composantes, autrement dit de points de vue auquels l'on pourrait s'intéresser?

Nous pouvons d'ores et déjà compter autant de composantes que de morceaux : *Pierre-personne*, *Pierre-sportif*, *Pierre-cinéphile* et *Pierre-enseignant-chercheur*. Nous pouvons encore ajouter à ce compte *Pierre-entier* (maintenant composé de quatre morceaux), mais remarquons qu'en le faisant nous considérons un ensemble de composantes déjà comptées. Il convient donc de compter autant de composantes qu'il y a de sous-ensembles dans l'ensemble des morceaux, car par exemple *Pierre-enseignant-chercheur* et *Pierre-personne-et-enseignant-chercheur* (l'union de *Pierre-personne* et *Pierre-enseignant-chercheur*) sont deux composantes différentes. Pour s'en convaincre, notons que demander son adresse à *Pierre-enseignant-chercheur* est une question sans ambiguïté tandis qu'elle ne l'est pas si on la pose à *Pierre-personne-et-enseignant-chercheur*.

Parce que quatre morceaux composent notre objet morcelé, Pierre a donc $2^4 - 1$ soit 15 composantes. Si l'on considère maintenant notre exemple complété par les cinq autres morceaux *Enseignant*, *Chercheur*, *Tennisman*, *Volleyeur* et *Skieur*, Pierre a donc alors 511 composantes ($2^9 - 1$). Nous en concluons que disposer d'un objet pour référencer chacune des composantes d'un objet morcelé n'est pas une solution raisonnable quant à la place mémoire alors nécessitée.

Les systèmes à prototypes existants semblent donc insuffisants pour permettre de manipuler l'entité unique Pierre, un ensemble d'objets reliés par des liens de délégation, de telle sorte que l'on puisse référencer l'une quelconque de ses composantes.

3 Nouveau modèle : description générale

Nous proposons dans cette section une première spécification d'un modèle d'objets permettant toutes les manipulations usuelles dans un langage à prototypes mais résolvant les deux problèmes que nous venons de présenter.

3.1 Un modèle d'objets morcelés

Parce qu'**un** objet morcelé est la représentation "éclatée" en **plusieurs** morceaux d'**une** entité du monde réel, il devient **un** objet de notre modèle. Les objets classiques (ceux de nos paragraphes précédents) deviennent des morceaux d'objets auxquels on ne peut pas envoyer de message directement.

Tout objet est donc défini comme étant un ensemble de morceaux (on peut donc le voir comme un dictionnaire de morceaux, les noms des morceaux étant locaux à l'objet) et chaque morceau détient une partie des propriétés de l'objet.

Création
Les objets sont créés par clonage d'un objet existant. La création d'un objet ex-nihilo est réalisée par clonage de l'objet vide (vide de morceaux) puis ajout de morceaux. L'objet vide est un objet prédéfini. La primitive de clonage est présentée plus précisément en § 3.2.

Organisation
Les morceaux sont organisés dans une hiérarchie de délégation, formant un graphe connexe et sans circuit, admettant une racine. Une primitive permet à l'utilisateur de spécifier et modifier les liens parents des morceaux, cette primitive est activée par envoi de message à l'objet qui veille au maintien de la connexité et de l'absence de circuits dans le graphe de délégation[3].

Evolution
Toute interaction avec les morceaux s'effectue par l'intermédiaire de l'objet, i.e. par envoi de message à l'objet, celui-ci a le contrôle de l'évolution de ses morceaux. Les méthodes de manipulation des morceaux sont des primitives du langage. On trouve des primitives pour :

- ajouter ou supprimer un morceau au sein de l'objet,
- ajouter, supprimer ou modifier une propriété détenue par un morceau.

Dans une syntaxe très informelle de type `receveur selecteur (arg1, ..., argN)`, le pseudo-code ci-dessous correspond à la création ex-nihilo de notre exemple Pierre (voir Fig.3) et de ses morceaux *Personne* et *Sportif* (nous considérons dans la suite que l'objet morcelé représentant Pierre a totalement été créé). Une propriété de type variable détient une valeur, une propriété de type méthode est représentée comme un bloc à la Smalltalk (`[param1, ..., paramN | corps]`). Les commentaires sont en italiques.

```
Pierre := EmptyObject clone(). création de Pierre par clonage de l'objet vide
Pierre addPart (#Personne, nil). ajout du morceau #Personne à Pierre avec un lien parent à
    nil
Pierre addProperty (#adresse, #Personne, "8, rue Octave"). ajout de la propriété #adresse
    à Pierre dans le morceau #Personne avec la valeur "8, rue Octave"
Pierre addProperty (#age, #Personne, 30).
Pierre addProperty (#nom, #Personne, "Pierre").
Pierre addProperty (#telephone, #Personne, 11-11-11-11).
Pierre addProperty (#vieillir, #Personne, [ | age := age + 1]).
Pierre addPart (#Sportif, #Personne). ajout du morceau #Sportif à Pierre avec un lien parent
    vers le morceau#Personne
Pierre addProperty (#endurance, #Sportif, 10).
Pierre addProperty (#poids, #Sportif, 80).
```

3. Une vérification du même type est effectuée à chaque ajout ou suppression de morceaux.

Activation de propriétés
Les propriétés de nos objets (définies au niveau des morceaux) peuvent être activées, comme dans tout langage de programmation par objets, par envois de messages.

1. L'envoi de message de base s'adresse à l'objet dans sa globalité, on peut par exemple demander à notre personnage Pierre (voir. section 2.2) quel est son age (accès à une variable) ou lui demander de calculer son nombre d'heures d'enseignement effectuées dans l'année (activation d'une méthode). On peut également demander à Pierre quel est son numéro de téléphone, message plus complexe à interpréter (le message peut être ambigü, voir § 4) car plusieurs réponses sont possibles, la propriété étant surchargée dans divers morceaux.

2. Nous avons montré que dans un objet morcelé, chaque morceau peut être considéré comme un point de vue particulier sur l'entité globale qu'est l'ensemble des morceaux. A cette caractéristique des objets morcelés, nous associons assez naturellement la possibilité d'envoyer un message à un objet selon un point de vue (en spécifiant un nom de morceau). Il s'agit, comme dans les systèmes à objets intégrant la notion de points de vue (voir § 5), d'effectuer une recherche de propriété en restreignant l'espace de recherche, on peut par exemple demander à Pierre en tant qu'enseignant-chercheur quel est son téléphone (on attend 22-22-22-22 comme réponse).

3. Il semble également naturel de s'adresser à un objet selon plusieurs points de vue. On peut par exemple demander à Pierre en tant qu'enseignant et sportif quelles sont ses heures de cours.

Remarquons que ces trois sortes d'envoi de message ne sont que les variantes de l'envoi de message selon plusieurs points de vue (3). Envoyer un message à l'objet dans sa globalité (1) peut s'interpréter comme envoyer un message à l'objet selon tous ses points de vue. Quand à l'envoi selon un seul point de vue (2), il s'agit d'une simple restriction du mécanisme.

Il est donc possible de référencer l'objet dans sa globalité afin d'en modifier ses morceaux et l'organisation de ceux-ci mais également pour activer l'une de ses propriétés. S'adresser à un objet selon un ou plusieurs points de vue permet à l'utilisateur de s'adresser à l'une quelconque de ses composantes sans que l'on ait eu besoin de créer un objet pour la référencer.

3.2 Factorisation de propriétés

Nous poursuivons la description du nouveau modèle et montrons ici comment gérer les bibliothèques de comportements communs aux objets d'une même famille, sans pour autant introduire des objets incohérents.

Les morceaux bibliothèques
Dans les morceaux d'un objet, on distingue les morceaux dits bibliothèques des autres

morceaux, appelés morceaux constituants. Les morceaux constituants sont ceux que nous avons vus dans la section précédente. Les morceaux bibliothèques jouent dans notre modèle le rôle des traits de SELF, i.e. ils servent de réservoirs de comportement partagés. La différence fondamentale est qu'ils n'ont pas le statut d'objet, mais celui de morceau : on ne peut donc pas leur envoyer de message, les propriétés qu'ils détiennent en sont utilisables que via le mécanisme de délégation par les morceaux constituants qui leur sont descendants dans la hiérarchie d'un objet.

Un morceau bibliothèque peut être racine d'un objet mais tout objet doit admettre une racine constituante, c'est à dire un morceau constituant ascendant dans la hiérarchie de tous les autres morceaux constituants.

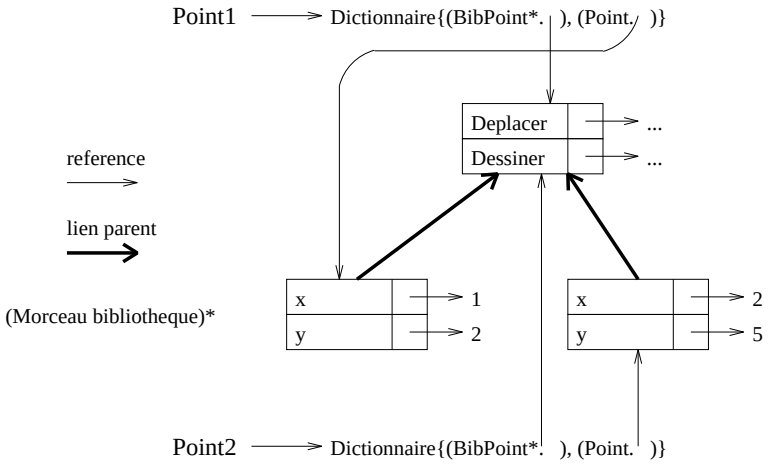

FIG. 4 - *Factorisation de propriétés communes à deux points*

Il est par exemple possible de factoriser deux méthodes sur une famille de points comme nous le montrons dans la figure 4. Le pseudo-code suivant montre la création de *Point1* . *BibPoint* est un morceau bibliothèque.

```
Point1 := EmptyObject clone().
Point1 addBib (#BibPoint, nil).
Point1 addProperty (#deplacer, #BibPoint, [ | <code de deplacer>]).
Point1 addProperty (#dessiner, #BibPoint, [ | <code de dessiner>]).
Point1 addPart (#Point, #BibPoint).
Point1 addProperty (#x, #Point, 1).
Point1 addProperty (#y, #Point, 2).
```

Les morceaux bibliothèques ne peuvent pas être désignés comme points de vue lors de l'envoi d'un message. L'objet veille à la réception de tout message au respect de cette règle. `Point1 as (#BibPoint, #dessiner)` provoque une erreur, `#BibPoint` n'est pas un morceau constituant de l'objet.

Clonage

Lors du clonage d'un objet, un clonage profond a lieu sur les morceaux constituants mais pas sur les morceaux bibliothèques. Les morceaux constituants sont donc réellement dupliqués alors que seules les références de l'objet vers les morceaux bibliothèques

sont copiées en ce qui les concerne. Ainsi lorsque nous clonons `Point1` pour définir un nouveau point `Point2`, le morceau `#BibPoint` appartient aux deux points.

```
Point2 := Point1 clone ().
Point2 changeProperty (#x, #Point, 2).
Point2 changeProperty (#y, #Point, 5).
```

Statut des bibliothèques
Le statut particulier des morceaux bibliothèques nous permet de réaliser la factorisation de propriétés comme nous l'illustrons sur notre exemple. La modification d'une méthode du morceau `#BibPoint` par l'intermédiaire de l'un ou l'autre des deux objets étant effective pour les deux points, les méthodes sont bien factorisées.

Les morceaux bibliothèques ne pouvant être sélectionnés en tant que point de vue lors des envois de messages, l'activation d'une méthode ne peut se faire que dans un contexte correct : tout ce qui était possible et indésirable avec les traits ne l'est plus (comme envoyer directement le message `deplacer` à `#BibPoint`), mais par contre les possibilités de factorisation existent toujours.

Les morceaux bibliothèques jouent le rôle de réservoir de propriétés joué par les classes dans les langages à classes mais la comparaison s'arrête là. Ce ne sont ni des descripteurs ni des créateurs d'instances.

3.3 Conclusion de la présentation générale

En définissant le concept d'objet dans notre modèle comme étant un ensemble de morceaux, nous permettons à l'utilisateur d'appréhender un objet morcelé dans son ensemble. Nos objets sont entiers et constituent une référence unique, certes restrictible par points de vue, pour l'utilisateur. Le fait que tous les messages soient traités au niveau de l'objet transforme nos objets en gestionnaires des morceaux et il est donc possible d'avoir un contrôle adéquat pour veiller au respect des règles de cohérence que nous avons définies : graphe de délégation connexe et sans circuit, "anonymat" des morceaux bibliothèques quant à l'activation de propriétés. La possibilité d'envoyer un message selon un point de vue, plusieurs points de vue, ou sans point de vue évite de créer un nombre exponentiel d'objets pour référencer toutes les composantes de l'objet.

Le statut particulier des morceaux bibliothèques, morceaux partagés et anonymes à l'envoi de messages, nous laisse toute liberté pour réaliser une factorisation de propriétés de manière sûre.

4 Nouveau modèle : envoi de message

Nous détaillons dans cette section la recherche de propriété effectuée lors d'un envoi de message. La forme la plus générale (mais pas nécessairement la plus utilisée) d'envoi de message dans notre système est l'envoi de message selon plusieurs points de vue. Les deux autres formes (un seul point de vue, globalité) en sont des cas particuliers.

Rappels et définitions préliminaires
Nous rappelons que si nous parlons de points de vue à l'envoi de messages, c'est parce que nous avons établi une relation entre les morceaux et les points de vue d'un objet. Nous avons appelé **morceau constituant** tout morceau n'étant pas morceau bibliothèque. A chaque morceau constituant d'un objet correspond un point de vue de l'objet comme nous l'avons montré au travers de notre exemple de représentation de la personne Pierre (voir § 2.2).

De plus, organiser les morceaux d'un objet dans une hiérarchie de délégation, c'est établir une relation de partage de propriétés entre morceaux. Plus une propriété est partagée, plus elle est générale, c'est à dire commune à un grand nombre de morceaux, donc de points de vue. Nous disons qu'un point de vue G est **plus général** (et inversement **plus spécifique**) qu'un autre point de vue S si le morceau auquel G correspond est un ascendant du morceau auquel correspond S. Nous rappelons encore que nous avons exigé que tout objet admette une **racine constituante**, c'est à dire un morceau constituant ascendant de tout autre morceau constituant de l'objet dans la hiérarchie de délégation. Tout objet admet donc un **point de vue le plus général**, c'est celui qui correspond à sa racine constituante.

Enfin, nous définissons comme étant le **plus petit ancêtre commun** (**PPAC**) d'un ensemble E de morceaux d'un même objet, le morceau correspondant au point de vue le plus spécifique, se trouvant sur tout chemin partant d'un élément de E vers la racine constituante. L'existence d'une racine constituante dans tout objet nous assure l'existence de PPAC pour tout ensemble de morceaux dans un objet.

Recherche de propriété
Nous pouvons maintenant présenter les différentes étapes de la recherche de propriété effectuée lors de l'envoi d'un message. Il se peut que la propriété soit définie dans plusieurs morceaux du receveur du message. Il nous a paru logique dans ce cas d'activer la propriété telle qu'elle a été définie pour le point de vue le plus spécifique, plus général que le plus grand nombre des points de vue spécifiés à l'envoi du message.

Soit le message O `as (`E`,` P`)` où O est l'objet receveur, E un ensemble de noms de morceaux et P le sélecteur de la propriété à activer. La recherche de P dans O selon les points de vue E est effectuée de la manière suivante :

1. vérification de l'appartenance à O des morceaux spécifiés dans E et de leur statut de morceau constituant (une exception est levée en cas d'échec),
2. restriction du graphe de délégation de O au sous-graphe ayant pour sommets les morceaux spécifiés dans E et leurs ancêtres,
3. recherche dans le graphe réduit du PPAC des morceaux spécifiés dans E,
4. recherche de P dans le graphe réduit "vers le haut" depuis PPAC,

5. (si échec au point précédent) recherche de P dans le graphe réduit "vers le bas".

Le point 1 est une simple vérification, tous les noms spécifiés dans E doivent être des noms de morceaux constituants de O.

Au point 2 est restreint le graphe de délégation de O de telle sorte que P ne soit pas recherché dans les morceaux où il ne devrait pas l'être, notamment lors de la recherche "vers le bas" effectuée au point 5. Dans notre exemple de la personne Pierre (voir Fig. 3 § 2.2) le graphe obtenu en ce point dans l'envoi du message `Pierre as ((#Personne, #Chercheur, #Volleyeur), niveau)` a pour ensemble de sommets {`#Personne`, `#Chercheur`, `#EnseignantChercheur`, `#Sportif`, `#Volleyeur`}.
Cette réduction du graphe de délégation n'est bien entendu effective que pendant l'évaluation du message : il ne s'agit pas de supprimer définitivement des morceaux dans un objet en demandant l'activation de l'une de ses propriétés.

Le point 3 consiste à déterminer le PPAC des morceaux spécifiés dans E en tant que points de vue de O. Le PPAC est le morceau auquel correspond le point de vue le plus spécifique, plus général que tous ceux spécifiés lors de l'envoi du message. Par exemple, dans notre représentation de Pierre, `#Sportif` est le PPAC de `#Tennisman`, `#Volleyeur` et `#Skieur` et *Pierre-Sportif* est le point de vue le plus spécifique plus général que les points de vue *Pierre-Tennisman*, *Pierre-Volleyeur* et *Pierre-Skieur*.
C'est donc à partir du PPAC qu'est effectuée une première recherche "vers le haut" au point 4. Cette recherche correspond au "lookup" classique du mécanisme de délégation. Le sélecteur est recherché dans le PPAC puis dans ses ascendants (parents, puis "grand-parents", etc...). S'il y a succès, la recherche est terminée et la propriété est activée. Par exemple, si l'on considère le message `Pierre as ((#EnseignantChercheur, #Personne, #Cinephile), adresse)`, le PPAC est le morceau `#Personne` et c'est la valeur de `adresse` définie en `#Personne`, soit `"8, rue Octave"`, qui est retournée.

Le point 5 est détaillé dans le paragraphe suivant.

Recherche vers le bas
Il s'agit de la partie la plus complexe dans la recherche du sélecteur. Elle n'est appliquée que si la propriété recherchée n'a pas été définie pour le point de vue le plus spécifique, plus général que tous les points de vue sélectionnés. Dans ce cas, le graphe réduit est parcouru vers le bas (dans le sens inverse de celui des liens de délégation) en largeur d'abord. Le sélecteur P est recherché dans chacun des morceaux ainsi rencontrés jusqu'à ce qu'il soit trouvé, ou qu'il n'y ait plus de morceaux à examiner.

Si P est trouvé dans un premier morceau M_1, la recherche est continuée en ignorant les descendants de M_1. Si alors le sélecteur n'est plus rencontré, la valeur associée à P dans M_1 est activée.
Par exemple, pour le message `Pierre as ((#Sportif, #Cinephile), endurance)`, la recherche a échoué au point 4 depuis le PPAC `#Personne` et `endurance` n'est ensuite trouvé que dans `#Sportif`.

Dans le cas contraire, si P est également trouvé dans un autre morceau parcouru M_2,une exception signalant un message ambigü[4] est levée. Cela signifie en effet que, comme le parcours est effectué en largeur d'abord et que les descendants de M_1 ne sont pas parcourus, il n'y a pas de relation d'ascendance ou de descendance entre M_1 et M_2. En d'autres termes les points de vue correspondant à M_1 et M_2 sont incomparables selon la relation de généralité/spécificité que nous avons défini sur les points de vue d'un objet. Il est donc alors impossible de déterminer quelle est la propriété la plus spécifique commune au plus grand nombre de points de vue spécifiés.
Par exemple, pour le message `Pierre as ((#Cinephile, #Tennisman, #Skieur), niveau)` (la recherche ayant échoué au point 4 depuis le PPAC `#Personne`), `niveau` n'est pas trouvé dans `#Cinephile` ni dans `#Sportif`, mais est ensuite trouvé une première fois dans `#Tennisman` et une deuxième fois dans `#Skieur`. Le point de vue *Pierre-Skieur* n'étant pas ni plus spécifique, ni plus général que le point de vue *Pierre-Tennisman*, quelle valeur devrait-on renvoyer pour `niveau` (`bon` ou `moyen`)?

Cas particuliers
A l'envoi d'un message, l'utilisateur spécifie les points de vue selon lesquels il désire s'adresser à l'objet en donnant l'ensemble des morceaux E auxquels ces points de vue correspondent. Si E est un singleton, cela signifie que l'utilisateur s'adresse à l'objet selon un seul point de vue. C'est un cas particulier de l'envoi de messages.
Dans ce cas, au point 3 de la recherche de propriété, le PPAC est facilement déterminé puisqu'il s'agit du morceau spécifié dans E. La recherche est alors effectuée vers le haut conformément au point 4. En cas d'échec aucune recherche vers le bas (point 5) n'est effectuée puisque tous les descendants du PPAC, autrement dit de l'unique morceau spécifié dans E, sont absents du graphe réduit.

Un autre cas particulier est celui dans lequel le message est adressé à l'objet vu dans sa globalité, c'est à dire selon tous ses points de vue à la fois. Pour envoyer un tel message on peut, bien sûr, spécifier un ensemble E dont tout morceau constituant de l'objet est élément, mais nous admettons qu'à un tel message correspond un message sous une forme simplifiée $O\ P$ où O est le receveur du message et P le sélecteur de la propriété à activer. Ainsi, par exemple pour notre Pierre, `Pierre as ((#Personne, #Enseignant-Chercheur, ..., #Volleyeur, #Skieur), adresse)` peut s'écrire également `Pierre adresse`. On peut remarquer que pour un tel envoi de message, le point 2 de la recherche de propriété n'a pour seul effet que l'élimination dans le graphe réduit d'éventuels morceaux bibliothèques étant feuilles de la hiérarchie (l'existence de tels morceaux est peu probable et ne pourrait s'expliquer que par le fait que l'objet doit être ultérieurement étendu par d'autres morceaux constituants). Le PPAC déterminé au point 3 est alors le morceau auquel correspond le point de vue le plus général de O, soit sa racine constituante.

4. Les ambiguïtés ne sont pour l'instant que signalées dans notre modèle, mais celles-ci admettent une forte ressemblance avec les ambigüités dûes à l'héritage multiple et il est fort probable que l'une des nombreuses solutions (linéarisation, priorités sur les liens parents [Chambers *et al.*, 1991], redirection explicite) [Ducournau *et al.*, 1992] jusqu'à lors proposées à ce problème puisse être adaptée dans notre modèle.

5 Travaux connexes

La spécification d'un objet comme un ensemble de morceaux est également présentée dans [Malenfant, 1995]. Le problème posé est le même que celui qui nous préoccupe, à savoir l'identité des objets dans les systèmes à délégation, et la solution proposée est à la base assez proche de la nôtre. L'accent y est surtout mis sur l'identité et l'encapsulation et le modèle est présenté de façon formelle. L'envoi de message aux objets dans leur globalité ainsi que le problème des bibliothèques ne sont pas abordés. Ce travail est tout à fait connexe au nôtre et fera l'objet de développements communs.

Le langage Kevo [Taivalsaari, 1993] propose une solution au problème de la factorisation de propriétés basée sur la gestion automatique des familles de clones. Cette solution est implémentée et fonctionne correctement avec cependant un certain nombre de restrictions. En particulier, Kevo ne supporte pas la délégation dans toute sa généralité et cette solution ne semble pas applicable dans le cas le plus général qui nous intéresse.

Notre tentative pour présenter un langage à protoypes sémantiquement cohérent nous a conduit à manipuler la notion d'envoi de message avec point de vue. Les points de vues ont été largement étudiés dans les langages à objets, citons par exemple des travaux tels que LOOPS [Bobrow *et al.*, 1983], ROME [Carré, 1989], FROME [Dekker, 1994], [Ferber, 1989]. A notre connaissance, toutes les études portent sur des langages à classes. Elles décrivent des systèmes permettant de spécifier un concept suivant différents points de vue et proposant des mécanismes pour qu'un même objet concret puisse être un représentant de plusieurs points de vue du concept.

En ce qui concerne la représentation des objets, notre système ne propose aucune structure de données spécifique pour représenter des points de vues, nous avons montré que leur expression découle de la nature du lien parent et du mécanisme de délégation. Il existe néanmoins certainement une connexion forte entre un graphe de morceaux tel que le nôtre et un système de multi-instanciation tel que celui de ROME ; nous le remarquons en constatant que l'implantation de ROME en Smalltalk nécessite la mise en place d'un graphe de morceaux d'objets liés par des liens de délégation [Vanwormhoudt, 1994; Bourdin, 1994]. L'étude comparative entre prototypes et systèmes de gestion de points de vue reste à faire.

6 Conclusion

Notre but était d'étudier la spécification d'un langage à prototypes sémantiquement cohérent et dans lequel seraient résolus deux problèmes relevés dans les langages existants : le problème du statut des objets et le problème du partage de propriétés communes à un ensemble d'objets similaires.

Nous avons présenté une première spécification d'un modèle à prototypes basé sur la représentation d'un objet par un ensemble de morceaux reliés par des liens de délégation.

Nous avons montré comment cette description permet de gérer de manière correcte, en les représentant comme des morceaux d'objets, les bibliothèques communes à des ensembles d'objets.

Nous avons expliqué pourquoi il était cohérent dans un système avec délégation. Les objets de plein droit étant des ensembles de morceaux reliés par des liens "parent", les morceaux peuvent être considérés comme des points de vue sur l'objet global et adressables comme tels.

Certains problèmes restent à travailler comme le partage de morceaux constituants entre objets ou la possibilité de définir des points de vue comme entité de première classe. Les prochaines étapes de notre travail seront également de faire le lien avec les systèmes de points de vues des langages à classes, avec la spécification formelle des objets morcelés puis de proposer une implantation valide de notre spécification. A ce sujet les principaux problèmes résident certainement dans la représentation du graphe des morceaux et dans l'optimisation de la recherche de propriétés, l'utilisation des techniques connues de linéarisation et de caches de propriétés seront certainement indispensables.

Remerciements
Ce travail fait suite à celui initié avec Jacques Malenfant et Pierre Cointe et a bénéficié de nos nombreuses discussions sur l'analyse des langages à prototypes. Nous remercions également Roland Ducournau pour ses suggestions relatives au lookup.

Références

[Agesen *et al.*, 1993] O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, et M. Wolczko. *The* SELF *3.0 Programmer's Reference Manual.* Sun Microsystems Inc, Stanford University, 1993.

[Borning, 1986] A.H. Borning. Classes versus Prototypes in Object-Oriented Languages. Dans *Proceedings of ACM/IEEE Fall Joint Computer Conference*, pages 36–40, 1986.

[Bobrow *et al.*, 1983] D. Bobrow, M. Stefik. *The LOOPS Manual.* Memo KB-VLSI-81-13, Xerox Parc, 1983.

[Bourdin, 1994] C. Bourdin. *Points de vue et représentation multiple et évolutive dans les langages à objets.* Mémoire de DEA informatique, Université des Sciences et Techniques du Languedoc, Montpellier 2, 1994.

[Carré, 1989] B. Carré. *Méthodologie orientée objet pour la représentation des connaissances, concepts de points de vue, de représentation multiple et évolutive d'objet.* Thèse d'informatique, Université de Lille, 1989.

[Chambers *et al.*, 1991] C. Chambers, D. Ungar, B.-W. Chang, et U. Hölzle. Parents are Shared Parts of Objects : Inheritance and Encapsulation in Self. *Lisp and Symbolic Computation (4)*, pages 207–222, 1991.

[Dekker, 1994] L. Dekker. *FROME: Représentation multiple et classification d'objets avec points de vue.* Thèse d'informatique, Université des Sciences et Technologies de Lille, 1989.

[Dony *et al.*, 1992] C. Dony, J. Malenfant, et P. Cointe. Prototype-Based Languages : From a New Taxonomy to Constructive Proposals and Their Validation. Dans *Proceedings of the 7th OOPSLA, Vancouver, British Columbia, ACM SIGPLAN Notices (27)10*, pages 201–217, 1992.

[Ducournau, 1989] R. Ducournau. Y3/YAFOOL *: Le langage à objets.* Sema Group 1989.

[Ducournau *et al.*, 1992] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, et A. Napoli. *L'héritage multiple dans tous ses états.* Rapport de recherche 92-021, Laboratoire d'Informatique Robotique et Micro-électronique de Montpellier, 1992.

[Ferber, 1989] J. Ferber. *Objets et agents : une étude des structures de représentation et de communications en Intelligence Artificielle.* Thèse d'informatique, Université Pierre et Marie Curie, Paris 6, 1989.

[Lieberman, 1981] H. Lieberman. *A preview of Act1.* AI memo No 625, Massachusets Institute of Technology, 1981.

[Lieberman, 1986] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Dans *Proceedings of the 1st OOPSLA, Portland, Oregon, ACM SIGPLAN Notices, 21(11)*, pages 214–223, 1986.

[MACL, 1989] *Macintosh Allegro Common-Lisp Reference Manual.* Version 1.3, 1989.

[Malenfant, 1995] J. Malenfant. *Split Objects : Taming Value Sharing in Object-Oriented Languages.* Rapport de recherche IRO-968, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1995.

[Mariño, 1993] O. Mariño. *TROPES.* Thèse d'informatique, Université Joseph Fourier, Grenoble 1, 1993.

[Myers *et al.*, 1992] B.A. Myers, D.A. Giuse, et B. Van der Zanden. Declarative Programming in a Prototype-Instance System : Object-Oriented Programming Without Writing Methods. Dans *Proceedings of 7th OOPSLA, Vancouver, British Columbia, ACM SIGPLAN Notices (27)10*, pages 184–200, 1992.

[Steyaert, 1994] P. Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks.* Thèse d'informatique, Vrije Universiteit Brussel, Belgique, 1994.

[Taivalsaari, 1993] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming.* Thèse d'informatique, Université de Jyväskylä, No 23, Finlande, 1993.

[Ungar *et al.*, 1991] D. Ungar, C. Chambers, B.-W. Chang, et U. Hölzle. Organizing Programs without Classes. *Lisp and Symbolic Computation (4)*, pages 223–242, 1991.

[Vanwormhoudt, 1994] G. Vanwormhoudt. *Points de vue, représentation multiple et évolutive en SmallTalk.* Note interne au Laboratoire d'Informatique Fondamentale de Lille, 1994.