Contribution à l'étude de l'association des paradigmes de programmation en logique et programmation par objets

Macaire Ngomo **Jean-Pierre Pécuchet**

Laboratoire d'Informatique de Rouen
INSA de Rouen
Equipe Heuristique et Informatique Avancée

B.P. 08, F-76131 Mont-Saint-Aignan Cedex
Tél. : +(33) 35 52 83 40 ; Fax : +(33) 35 52 83 32
E-mail : {Ngomo,Pécuchet}@lmi.insa-rouen.fr

Résumé

Ce papier présente un nouveau langage de programmation par objets en logique, WorldLog, et, à travers sa réalisation, une nouvelle approche de l'association des paradigmes de programmation en logique et de programmation par objets. WorldLog étend Prolog vers la programmation par objets réflexive et est construit autour des principaux concepts objets adaptés à la programmation en logique: objet logique, classe, instance, métaclasse, clause objet, méthode logique, envoi de message, héritage multiple, héritage partiel. Il est réalisé suivant le modèle ObjVLisp enrichi de notions de variables d'instance privées et de clauses objets privées. Une notion d'objets mutables est introduite. Pour fournir une sémantique logique aux changements d'état des objets, nous étendons la logique du premier ordre, sur laquelle Prolog est basé, à un système modal défini par Warren et nous remplaçons son opérateur modal, *assume*, par un autre opérateur, *given*, qui restreint l'effet de *assume*. L'opérateur *given* définit, comme *assume*, une relation sur les "mondes possibles" (ici les bases de règles logiques). given(BO)@(B) établit un lien dynamique entre un programme P et une base virtuelle d'objets BO pour résoudre le but B. De plus, dans chaque base BO les unifications des variables dans les règles se font en même temps que dans les requêtes. Ces bases d'objets constituent un environnement virtuel permettant la gestion des variables existentielles. Un changement d'état est alors vu comme le passage d'un monde à un autre. Chaque monde représente un aspect de l'univers et un but n'est vrai que relativement à un monde donné. L'exécution d'un programme devient alors la déduction en logique modale, ce qui fait de WorldLog un langage de programmation en logique au sens de Goguen et Meseguer.

Mots clés

Programmation en logique, programmation par objets, Prolog, objets logiques, clauses objets, clauses objets privées, variables privées, méthodes logique, envoi de message, message anonyme, héritage multiple, héritage partiel, réutilisation, unification, quantification des variables, logique modale, ObjVLisp, réflexivité, uniformité.

1. Introduction

Le développement de la recherche en Intelligence Artificielle et la multiplicité des domaines d'applications, ont mis en évidence un réel besoin d'élaboration de langages spécifiques adaptés à ces contextes intelligents. Plus particulièrement, la nécessité de plus en plus forte de manipuler des données de plus en plus complexes et de plus en plus structurées ont permis l'avènement des langages à objets ; la nécessité de manipuler ces données de manière simple et les concepts de la logique formelle ont conduit à la réalisation de langages de programmation en logique dont Prolog est le plus répandu. Prolog est un langage basé sur un seul mécanisme d'évaluation: la résolution logique [Robi65] [Kowa79] [Colm83]. La programmation logique de Prolog offre nombreux avantages: sa sémantique déclarative, son unification, son mécanisme de résolution avec retour en arrière, sa puissance d'expression, sa simplicité, sa base théorique. Cependant, un langage se doit aussi d'être efficace. Le principe de résolution de Prolog permet de trouver toutes les solutions d'un problème: il y a donc non-déterminisme. Ce non-déterminisme est un avantage car il est possible de trouver l'ensemble des solutions d'un problème sans avoir à spécifier comment les trouver. Cependant, la mise en oeuvre de ce mécanisme est très coûteuse à la fois en temps d'exécution et en place mémoire. Les améliorations proposées portent principalement, sur une meilleure structuration et une réutilisation des programmes, sur sa compilation ou la mise en oeuvre de mécanismes de contrôle sur la résolution (backtracking intelligent, ...), sur la résolution parallèle. Dans ce papier, nous nous intéressons à l'extension de Prolog vers la programmation par objets. La logique peut donner des fondements théoriques solides au modèles de calcul à base d'objets et la programmation par objets permet l'application des principes du génie logiciel en apportant à la programmation en logique les fonctionnalités des objets. L'intégration de ces deux paradigmes de programmation au sein d'un même système offre donc plusieurs avantages. Elle permet de bénéficier des avantages de ces deux paradigmes, de corriger les faiblesses de l'un par les avantages de l'autre, et peut conduire à de nouveaux concepts de programmation, pouvant sortir du cadre simple de la programmation par objets ou de la programmation en logique et enrichissant les modèles de départ. Comme le souligne [Masi89], les liens entre Prolog et les langages à objets sont relativement anciens [Zani84] et sont surtout fondés sur des formalismes de représentation complémentaires [NeWa88]. La réalisation d'un langage à objets à partir de Prolog a plusieurs avantages. Elle permet de cumuler les mécanismes d'unification et de résolution avec retour arrière de Prolog et les fonctionnalités des objets [FuHi86], de représenter la connaissance de manière structurée à l'aide des objets et de manière uniforme à l'aide des clauses (l'état des objets et les méthodes sont représentés par des clauses et un envoi de message par un but Prolog). La réalisation de langages à objets en logique est devenue un domaine actif de recherche. Cela peut se justifie par la multiplicité d'approches proposées en une décennie : les Objets en Prolog de [Zani84], ESP [Chik84], Vulcan [KTMB86], LOGIN [AiNa86], POL [Gall86], SPOOL [FuHi86], POLKA [Davi88], les Objets Logiques de [Cone88], CIEL [Gand88], LIFE [AiLi88], ULog [Gloes89], ObjVProlog [Male90], les Objets Linéaires [AnPa90], L&O [MacC92], Modulog [Doui93], etc. Chaque approche est souvent guidée par un but à atteindre, parfois différent de ceux fixés par les autres. Malgré cette panoplie langages proposés, cette intégration pose encore de sérieux problèmes dus en particulier à la recherche d'une sémantique logique, d'une implantation efficace et d'un équilibre entre ces deux. De plus, les langage réalisés n'ont pas encore dépassé le stade de la recherche.

Dans ce papier, nous proposons un nouveau langage de programmation, WorldLog, et, à travers sa réalisation, une nouvelle approche de l'association des paradigmes de programmation en logique et programmation par objets. WorldLog étend Prolog vers la programmation par objets réflexive. et est construit autour des principaux concepts objets adaptés à la programmation en logique : objet logique, classe, instance, métaclasse, clause objet, méthode logique, envoi de message, héritage multiple, héritage partiel d'une méthode. WorldLog est réalisé suivant le modèle ObjVLisp[1] [Coin87] enrichi de notions de variables d'instance privées et de clauses objets privées. Nous donnons, dans la section suivante, une brève description de ce modèle. WorldLog supporte l'héritage multiple avec une sémantique non monotone. Deux stratégies complémentaires sont disponibles pour la gestion de l'héritage et ses conflits: une stratégie par défaut, celui de CLOS [Bobr88], et une stratégie par désignation explicite permettant au programmeur de résoudre explicitement les conflits, en désignant nommément la superclasse de provenance d'une propriété. La désignation explicite d'une classe permet aussi de réduire le coût des méthodes de parcours du graphe d'héritage puisqu'il consiste à effectuer un saut vers la classe désignée[2]. Une méthode logique est représentée par un prédicat, ce qui donne la possibilité pour une méthode d'avoir plusieurs définitions sous forme de clauses. Cette possibilité introduit une autre vision de la notion traditionnelle de méthode et nous a permis, puisque certaines clauses objets peuvent être définies comme privées, d'étendre l'héritage d'une méthode à l'héritage partiel d'une méthode. Une notion d'objets mutables est introduite. Pour fournir unr sémantique logique aux changements d'état des objets, nous étendons la logique du premier ordre, sur laquelle est basé Prolog, à un système modal défini par [Warr84] et nous remplaçons son opérateur modal, *assume* introduit pour remplacer l'*assert* de Prolog et corriger ses défauts, par un autre opérateur modal, *given*, qui restreint l'effet de l'*assume*. L'opérateur *given* définit, comme *assume*, une relation sur l'ensemble des "mondes possibles" (ici les bases de règles logiques). *given(BO)@(B)* établit un lien dynamique entre un programme P et une base virtuelle d'objets BO pour résoudre le but B. Dans chaque base BO les unifications des variables dans les règles se font en même temps que dans les requêtes. Ces bases constituent un environnement virtuel permettant la gestion des variables existentielles. Un changement d'état est alors vu comme le passage d'un monde possible à un autre. Chaque monde représente un aspect de l'*univers* et un but n'est vrai que relativement à une monde donné. L'exécution d'un programme devient alors la déduction en logique modale, ce qui fait de WorldLog un langage de programmation en logique au sens de [GoMe88].

2. Le modèle objet

WorldLog est réalisé suivant le modèle ObjVLisp [Coin87], adapté au cadre de la programmation en logique et enrichi de notions de variables d'instance privées, clauses objets privées. Ce modèle a été choisi pour son uniformité, sa simplicité et sa minimalité. Nous ne donnons ici qu'une description sommaire de ce modèle. Le modèle ObjVLisp est une description uniforme et réflexive des concepts de métaclasse,

---

[1] ObjVLisp n'est pas un langage mais un modèle objet proposé par P. Cointe (1987) pour l'expérimentation des mécanismes objets.

[2] La classe désignée doit appartenir à la hiérarchie locale de l'objet courant.

classe et instance en uniformisant le statut des objets. Il est né des travaux sur Smalltalk-76 [Coin83], avec un souci de minimalité et de simplicité intéressantes. Il réduit considérablement la lourdeur d'un grand système comme Smalltalk-80 [GoRo83]. L'essenciel de ce modèle peut être exprimé en six principes [Coin87]:

Principe 1: Un objet est l'union des données et de procédures.

Principe 2: La seule façon d'activer un objet est la transmission d'un message.

Principe 3: Tout objet est instance d'une classe qui en spécifie les données et le comportement. Les instances d'une classe sont créées dynamiquement et diffèrent par les valeurs de leurs variables d'instance communes.

Principe 4: Une classe est aussi un objet, instance d'une autre classe appelée métaclasse. Par conséquent, d'après le principe 3, toute classe a une métaclasse qui décrit sa structure et son comportement.

Principe 5: Une classe peut être définie comme sous-classe d'autres classes par héritage.

Principe 6: Si les variables d'instance définissent un environnement local à l'objet, il existe aussi des variables de classe qui forment un environnement global à toutes les instance d'une classe. Les variables de classe sont ceux décrites au niveau des métaclasses.

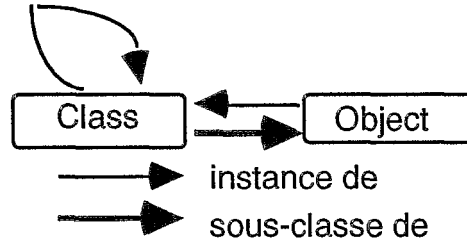

Figure 1: Le noyau d'ObjVLisp.

Comme le montre la figure 1, le noyau d'ObjVLisp repose sur deux classes. La classe ***Class*** est la racine du graphe d'instanciation. Pour éviter la régression à l'infini de la relation d'instanciation, ***Class*** est sa propre instance. Elle est le premier objet du système et c'est à partir d'elle que sont engendrées toutes les autres classes. Elle détient donc le comportement par défaut des classes, en particulier l'unique méthode d'instanciation *new*. La classe ***Object*** est la racine du graphe d'héritage ; toutes les autres classes, y compris ***Class***, héritent de ses propriétés.

3. Les concepts objets du langage

3.1 Objet logique

L'entité de base du langage WorldLog est l'objet logique, c'est-à-dire un objet dont la base de connaissances est formée d'énoncés logiques [Cone88] [Male90]. Un objet logique du langage WorldLog est une base de règles logiques. Il a des slots qui définissent son état interne et des méthodes qui définissent son comportement. Slots et méthodes sont représentés de manière uniforme au moyen des règles logiques. Chaque slot est fourni sous la forme d'un couple (Attr,Val), et peut être rattaché à l'objet Obj au

moyen d'une règle (fait) Prolog R à trois place: R(Objet,Attribut,Valeur). Cette approche nécessite, pour chaque objet, autant de règles que de slots. Ainsi, plutôt que d'utiliser plusieurs règles, tous les slots peuvent être encapsulés dans une même règle R sous la forme: R(Objet,((Attribut$_1$,Valeur$_1$),...,(Attribut$_n$,Valeur$_n$))). Ceci est important étant donné que l'approche directe tend à une explosion combinatoire, en particulier lorsque le nombre de données est important. Nous représentons donc l'état d'un objet par un fait Prolog, ***with/2***, dont les deux paramètres sont le nom de l'objet et la liste de ses slots:

obj(<Nom>) ***with*** [<Attr$_1$>:=<Val$_1$>,...,<Attr$_n$>:=<Val$_n$>].

Fournir le nom d'un l'objet comme un paramètre d'une règle apporte beaucoup de souplesse dans la manipulation des objets. La notation obj(<Nom>) est utilisée pour distinguer, parmi les termes utilisés, les identificateurs d'objets des autres termes. Ainsi, *attr$_1$:=obj(ma_voiture)* et *attr$_2$:=ma_voiture* signifient que la valeur de l'attribut *attr$_1$* est un objet alors que celle de l'attribut *attr$_2$* est la constante *ma_voiture*. Le programme P1 ci-dessous est un exemple de représentation de l'état d'un objet.

obj(ma_voiture) ***with*** [class:=obj(voiture),proprietaire:=obj(paul),couleur:=blanche].
Programme P1.

3.2 Métaclasse, classe et instance

D'après les principes du modèle ObjVLisp décrits dans la section 2., tout objet est instance d'une classe. Une classe est aussi un objet appartenant à une autre classe appelée métaclasse. La première métaclasse est la classe **obj(class)**, racine de l'arbre d'instanciation. Une classe a la même structure que celle définie ci-dessus, avec un slots privilégié, *supers*, désignant la liste de ses superclasses. La racine du graphe d'héritage est la classe **obj(object)**. Un exemple de déclaration d'une classe est donné par le programme P2..

obj(point) **with** [class:=obj(meta_point),supers:=[obj(object)],x:=0,y:=0].
Programme P2.

3.3 Clause objet

3.3.1 Syntaxe d'une clause objet

Une clause de Horn a la forme suivante A :- A$_1$,...,A$_n$. Une clause objet est une clause particulière préfixée par sa classe d'appartenance. Sa syntaxe est la suivante:

<Classe> & <BO> @ <Sélecteur>(<Arg$_1$>,...,<Arg$_n$>) :- <Corps>.

où <Classe> est sa classe d'appartenance, <Sélecteur> est la constante prédicative, les <Arg$_i$> sont ses paramètres, <Corps> est le corps de la clause. L'opérateur, &/2 est vu ici comme un opérateur d'attachement et permet de rattacher une clause à sa classe d'appartenance. Nous verrons le rôle de l'opérateur @/2 dans la suite. Le paramètre <BO> est une variable non instanciée qui s'unifiera avec une base d'objets lors de l'activation de la clause. Nous reviendrons sur le rôle de ce paramètre. Voici un exemple de clause objet.

obj(point) & BO @ getx(X) :- ...
Programme P3.

3.3.2 Lecture d'une clause objet

Une clause de Horn A :- $A_1,...,A_n$. peut être lue de la manière suivante: "si $A_1,...,A_n$ sont vrais, alors A est vrai". A est alors une conséquence logique de $A_1,...,A_n$. La lecture d'une clause objet C & BO @ A :- $A_1,...,A_n$. ne se fait que relativement à sa classe d'appartenance C: "dans C, si $A_1,...,A_n$ sont vrais, alors A est vrai".

3.4 Méthode logique

Dans les langages à objets traditionnels, comme C++ [Stro89] ou Smalltalk-80 [GoRo83], une méthode sont représentée par une procédure. Il n'existe pas comme tel la notion de procédure en programmation en logique. Cependant, il existe une correspondance naturelle entre une procédure en programmation traditionnelle et un prédicat défini à l'aide des clauses. Il est donc naturel de représenter une méthode logique par un prédicat logique. Ce choix apporte aux méthodes logiques les avantages des règles logiques [LeMe88]: les méthodes logiques sont définies déclarativement et utilisées dans un processus de déduction; les paramètres en entrée et en sortie d'une méthode logique ne sont pas fixés statiquement mais plutôt déterminés à l'appel de la méthode; la même méthode logique peut avoir plusieurs définitions, sous forme de clauses, qui seront considérées dans la déduction. Ainsi, une méthode logique en WorldLog est représentée par un prédicat défini à l'aide des clauses objets.

3.5 Envoi de message

Etant donné qu'une méthode logique est un prédicat, un message est interprété comme la délégation d'un but à résoudre pour l'objet expéditeur et comme une requête de démonstrastion d'un but pour l'objet receveur, en utilisant les règles de sa base [Male90]. Dans cette interprétation, la sémantique appel-retour de la programmation procédurale est abandonnée au profit de la sémantique succès-échec de la programmation en logique [Gall86] [LeMe88]. Ceci implique que plusieurs solutions au même appel peuvent être trouvées, de la même façon qu'un but Prolog peut avoir plusieurs solutions. En WorldLog, un envoi de message a l'une des formes suivantes:

O :: BO @ B envoie le but B à l'objet O, la recherche de la méthode commence au niveau de sa classe d'instanciation et B est résolu en utilisant la base d'objets BO.

O as C :: BO @ B envoie le but B à l'objet O, la recherche de la méthode commence au niveau de la classe C et B est résolu en utilisant la base d'objets BO.

self :: (BO,B) envoie le but B à l'objet lui-même, la recherche de la méthode commence au niveau de sa classe d'instanciation et B est résolu en utilisant la base d'objets BO.

self as C :: BO @ B envoie le but B à l'objet lui-même, la recherche de la méthode commence au niveau de la classe C et B est résolu en utilisant la base d'objets BO.

super :: BO @ B envoie le but B à l'objet lui-même, la recherche de la méthode commence au niveau des superclasses de sa classe d'instanciation et B est résolu en utilisant la base d'objets BO.

BO désigne une base virtuelle d'objets qui contient l'état de l'objet activé. Le but B est résolu en considérant la base de règles obtenue en liant dynamiquement la base BO et le programme (une autre base de règles logiques). Le traitement d'un message défini à l'aide du prédicat ::/2 nécessite deux étapes: la recherche de la méthode puis son activation. Cependant, il arrive que l'on n'ait besoin que d'interroger un objet s'il sait faire quelque chose. Pour répondre à un tel besoin, nous avons introduit un autre prédicat, ?:/2, qui limite le traitement d'un message à la recherche de la méthode. La syntaxe d'envoi d'un message avec ce nouveau prédicat devient:

O ?: BO @ B	O as C ?: BO @ B	self ?: BO @ B
self as C ?: (BO,B)		super ?: BO @ B

3.6 Héritage multiple

WorldLog supporte l'héritage multiple avec une sémantique non monotone. L'héritage multiple est géré par deux stratégies complémentaires. La première est la stratégie par défaut du système. Elle consiste à linéariser le graphe d'héritage en définissant un ordre total entre les classes de la hiérarchie local d'un objet. L'algorithme utilisé ici est celui de CLOS [Bobr88]. La seconde est la stratégie par désignation explicite. Elle permet de résoudre explicitement les conflits en désignant nommément la classe de provenance d'une propriété, lorsqu'il y a ambiguïté ou lorsque l'on désire utiliser une propriété particulière. La désignation explicite permet au programmeur d'avoir le plus grand contrôle sur le mécanisme d'héritage. C'est aussi un moyen permettant de déduire la complexité des méthodes de parcours du graphe d'héritage, puisqu'elle consiste à effectuer un saut vers la classe désignée[3], ce qui évite de visiter inutilement toutes les classes intermédiaires.

4. Modélisation du changement d'état des objets logiques

Nous abordons maintenant un problème central en programmation par objets en logique: le changement d'état des objets logiques. Traditionnellement, l'état d'un objet est représenté par les valeurs affectées à ses variables d'instance et cet état peut être modifié par affectation d'une nouvelle valeur à une de ces variables. En programmation en logique, la situation est fort différente, et en particulier en Prolog qui sert de base pour WorldLog. Prolog pur implante un sous ensemble de la logique du premier ordre. Or, la logique du premier ordre décrit un ensemble de relations et de propriétés statiques entre les entités. Aucune procédure de déduction existe, en logique du premier ordre, lorsque l'ensemble d'assertions utilisé peut changer pendant la déduction. Pour fournir une sémantique aux changements d'état des objets, nous avons choisi d'étendre la logique du premier ordre, sur laquelle Prolog est basé, à un système modal défini par [Warr84] comme base sémantique des mis à jours. Rappelons au passage la définition de Goguen et Meseguer: "un langage de programmation en logique est celui dont les programmes

[3]La classe désignée doit être un élément de la hiérarchie local de l'objet courant.

sont faits de phrases d'un système logique bien défini et dont la sémantique est la déduction dans ce système logique" [GoMe86]. Warren propose de remplacer l'*assert* de Prolog, dont la sémantique est impétative, par son opétateur modal, *assume*, et a défini un système modal pour donner un sens à l'*assume*. L'opérateur *assume* fait passer la base de règles d'un monde possible à un autre dans lequelle le reste de la déduction s'exécutera [Warr84] [MaWa88]. Ceci implique une application de l'opérateur modale sur une suite de buts. Illustrons cela par un exemple adapté de [Warr84]:

```
p(a).
:- p(X),assume(p(b))@(p(Y)).
```
Programme P5.

Ici, le sous but p(X) est déduit uniquement à partir de la règle définie dans le programme, p(a), alors que p(Y) est déduit à partir de la règle définie dans le programme et celle fournie comme paramètre de l'opérateur assume, c'est-à-dire p(b). L'opérateur assume s'applique donc au but p(Y) en ajoutant la règle p(b). au programme. Les solutions pour cette requête sont alors {X=a,Y=a} et {X=a,Y=b} alors qu'avec l'assert la variable X serait unifiée aussi à la valeur b. On désigne explicitement sur quels buts s'applique la modification à l'aide de l'opérateur @/2 dont la syntaxe est <opérateur modal> @ <suite de buts> et la sémantique est "applique l'opérateur modal à gauche en ajoutant au programme les règles fournies comme paramètres de l'opérateur modal pour la résoudre la suite de buts donnée à droite". Les buts qui ne sont pas dans la liste de buts sur laquelle s'applique la modification ne sont pas affectés par cette modification. Ils sont entièrement résolus dans la version initiale du programme. Corollairement, les seuls buts qui sont affectés par une modification sont ceux visés par l'opérateur @/2. Si la requête échoue, aucune modification n'est faite et on retrouve le programme initial. Par exemple, la requête assume(p(b))@(p(Y)),p(X) le système répondra {Y=a,X=a} et {Y=b,X=a}. On peut aussi imbriquer les opérateurs modaux avec la même interprétation: assume(p(b))@(assume(p(c))@(p(X))). Le système modal qui donne une sémantique à l'*assume* est défini dans [Warr84]. Ceci étant, pour appliquer ce système à la programmation par objets en logique, nous avons rempacé l'*assume* par un autre opérateur modal, given, qui restreint l'effet de l'*assume*: given(BO)@(B). La restriction consiste surtout à ne pas permettre des imbrications, ce que *assume* autorise. D'autre part, dans la formule modale given(BO)@(B), BO est une base d'objets (un monde possible), constituée de règles qui définissent l'état des objets, et le but B est déduit à partir des règles du programme et celles contenues dans BO. Les règles du programme sont celles qui définissent les méthodes logiques et éventuellement des règles ordinaires Prolog. En programmation par objets, le seul moyen d'activer un objet c'est de lui envoyer un message. Nous allons donc étendre les formules modales de la forme given(BO)@(B) au cadre objet en spécifiant l'objet receveur du message:

```
O :: given(BO) @ (B)
O ?: given(BO) @ (B)
```

signifiant "O résoud le but B dans la base BO". BO est alors une base virtuelle qui contient l'état de l'objet O. Ainsi, si l'on considère deux bases distinctes OB1 et OB2, O::given(BO1)@(B) et O::given(BO2)@(B) peuvent conduire à des résultats différents, car l'état de O n'est pas forcément le même dans les deux bases. Conformément à la théorie de Warren, nous avons défini les clauses modales de la manière suivante:

given(OB) @ (B) :- <Corps>.

et une extension au cadre objet nous a conduit, en choisissant de préfixer chaque clause de ce type par sa classe d'appartenance, à la syntaxe suivante:

C & given(OB) @ (B) :- <Corps>.

où C est la classe d'appartenance de la clause. Pour obtenir les syntaxes données dans les section 3.3 et 3.5, il suffit alors de rendre implicite l'existence de l'opérateur modal given. Ce qui donne O :: BO @ B et O :: BO @ B pour un envoi de message et C&BO@B:-<Corps> pour une clause objet.

Avec cette approche, un changement d'état est, comme nous l'avons déjà dit, interprété comme le passage d'une base à une autre. Chaque base représente un aspect de l'univers (un monde possible) et un but n'est donc vrai que relativement à une base donnée. Les principales méthodes de changement du langage WorldLog sont:

obj(class) & BO1 @ new(O,E,BO2) :- ..une classe de la base BO1 recevant ce message crée une instance O avec l'état E dans la base BO2.

obj(object)& BO1 @ setv(Attr,Val,BO2):-... affecte à l'attribut Attr de l'objet receveur de la base BO1 la valeur Val en créant dynamiquement une autre base BO2 qui contiendra le nouvel état de l'objet.

obj(object) & BO @ getv(Attr,Val):- ... c'est la méthode d'accès à la valeur d'un attribut, mais peut être aussi utilisée pour affecter à une valeur à un attribut, cependant l'opération d'affectation correspond ici à une unification.

Cette approche à l'avantage d'être basée sur une logique bien définie. Elle est aussi remarquable dans sa façon de traiter le problème du changement de la quantification des variables [Warr84]. En effet, une variable logique dans une requête est quantifiée existentiellement alors que dans un programme les variables sont quantifiées universellement. Par conséquent, lorsque par exemple une règle contenant des variables est l'argument d'un *assert*, son ajout au programme fait passer ces variables d'un statut à un autre. Illustrons ceci par un exemple simple.

(1) :- X=2,assert(point(X,3)),point(Z,3). => {X=2,Z=2}
(2) :- assert(point(X,3)),X=2,point(Z,3). => {X=2,Z=_123}
Programme P6.

Commentaires:

(1): X prend la valeur 2 et on ajoute le fait fermé point(2,3) dans la base. Et à la question ∃Z tel que point(Z,3)? le système répond {Z=2} par unification de point(Z,3) et point(2,3).

(2): on commence par ajouter le fait point(X_copy,3) dans la base (X_copy étant obtenu par renommage de X) puis X prend la valeur 2 (mais pas X_copy). On passe donc de la quantification existentielle (∃X) à la quantification universelle (∀X_copy). Et à la question ∃Z tel que point(Z,3)? le système répond {Z=_123} par unification de point(Z,3) et point(_123,3).

Trois solutions ont été proposées pour résoudre ce problème:

1) ne permettre l'ajout que des règles ne contenant pas des variables [Warr84] [MaWa88] ;
2) forcer le programmeur à quantifier explicitement les variables dans les règles à ajouter au programme [BoWe85] [Bowe85] ;
3) permettre l'ajout des règles avec des variables mais gérer les unifications à ces variables de telle façon que, lorsqu'une variable (dans une règge dans le programme) est unifiée dans la requête, la règle dans la base soit aussi modifiée en remplaçant la variable par le terme auquel elle a été unifiée.

La première approche est très restrictive. La seconde, utilisée dans ObjVProlog [Male90], est déjà plus élaborée mais elle est à notre avis très contraingante pour le programmeur qui ne voit pas ses efforts de programmation diminuer. La troisième approche, bien que difficile à implanter, puisqu'elle pose le problème de la gestion des variables existentielles dans la base, nous a beaucoup plus séduit et nous l'avons adoptée pour l'implantation de WorldLog. Pour contourner la difficulté de gérer des variables existentielles dans le programme, nous avons introduit une base virtuelle et temporaire qui ne contient que les règles définissant l'état des objets. Les autres règles restent dans la base réel (le programme). Lors de la résolution, même si certains objets peuvent être définis dans le programme, l'état des objets n'est accessible qu'à travers cet environnement virtuel et dans lequel les unifications des variables dans les règles se font en même temps que celles des variables dans les requêtes. Les variables étant quantifiées existentiellement dans cet environnement, les liens entre les variables utilisées dans les requêtes sont faits naturellement. Pour reprendre l'exemple de l'*assert* ci-dessus, nous avons:

```
(1)' :- X=2,O::BO1@setx(X,BO2),O::BO2@getx(Z).      => {X=2,Z=2}
(2)' :- O::BO1@setx(X,BO2),X=2,O::BO2@getx(Z).      => {X=2,Z=2}
```
Programme P7.

Ce qui nous conduit au même résultat. Cette procédure de gestion des variables existentielles est illustrée par la figure 2. Contrairement aux approches déterministes, impératives (ESP [Chik84], [Cone88], ULog [Gloes90], L&O [MacC92], Modulog [Doui93], ...) ou quasi-impératives, celles qui fournissent des mécanismes pragmatiques en donnant une sémantique opérationnelle et en essayant de ne pas trop s'éloigner de la logique (ObjVProlog [Male90],...), cette approche préserve la sémantique logique, le prix à payer étant la prolifération des variables et demander au programmeur d'être explicite dans ses intentions en l'obligeant de préciser, à chaque délégation d'un but, le monde dans lequel doit être résolu ce but. En ce qui concerne la prolifération des variables, notons que le même problème se pose, avec plus d'acuité d'ailleurs, dans certaines approches utilisant des structures incomplètes [Doui93] ou des processus comme l'approche des Objets Linéaires d'Andréoli et Pareschi [AnP91] qui utilise l'approche des langages de programmation logique à la Concurrent Prolog. De plus, dans l'approche de WorldLog, la gestion des bases d'objets est laissée à la charge du programmeur qui a la liberté de passer, dans un protocole d'envoi de message, la base d'objets de son choix. Contrairement à certaines approches comme celle de CIEL [Gand88], le programmeur ne manipule pas la structure de l'objet, mais son identificateur. Il est donc possible de partager un objet entre plusieurs portions d'un programme. La manipulation d'un identificateur au lieu d'une structure apporte plus de souplesse dans la manipulation des objets et permet d'éviter la verbosité [KTMB86].

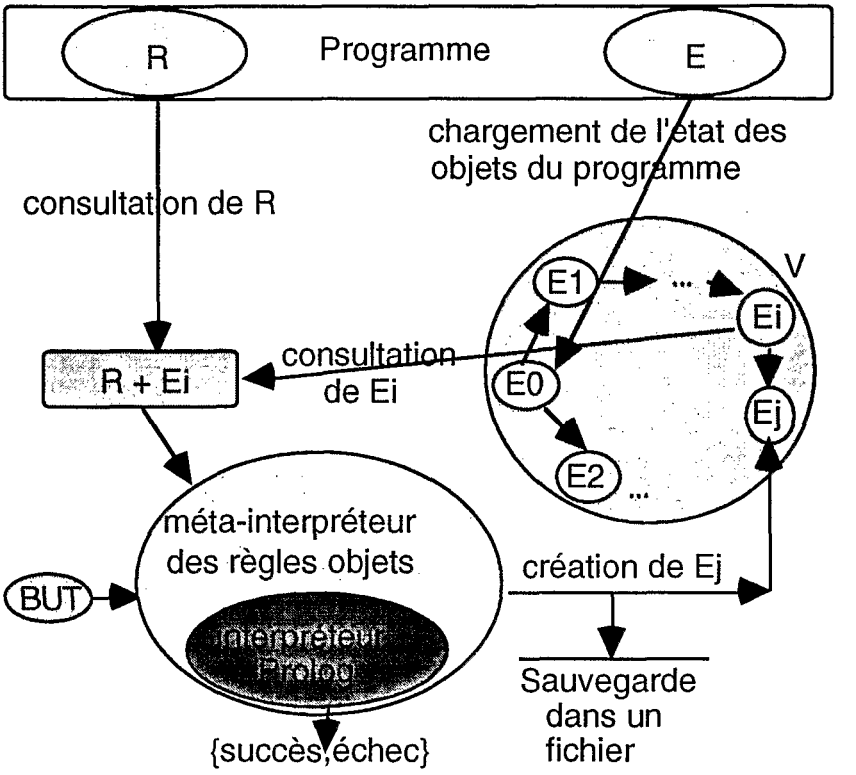

Figure 2: Architecturede l'environnement WorldLog.
R : règles objets + règles Prolog ordinaires ; E : règles définissant d'état des objets définis dans le programme
Ei : base virtuelle d'objets ; BUT : but à résoudre ; V : environnement virtuel.

5. Autres caractéristiques du langage

5.1 Variables d'instance privées et variables d'instance publiques

En WorldLog, une variable d'instance peut être définie comme privée ou publique. Alors que la déclaration d'une variable d'instance publique se fait de façon naturelle, <Var>:=<Val>, celle d'une variable d'instance privée est précédée du symbole " #- ": #- <Var>:=<Val>.

5.2 Clauses objets privées et clauses objets publiques

Comme il existe des variables d'instance privées et des variables d'instance publiques, il existe aussi des causes objets privées et des clauses objets publiques. Afin d'introduire ces deux notions, nous avons enrichi la syntaxe d'une clause objet en ajoutant un paramètre spécifiant le type de la clause. Ce qui donne la forme générale définitive suivante:

(<Classe>,<Type>) & <BO> @ <Sélecteur>(<Arg_1>,...,<Arg_n>) :- <Corps>.

où <Type> vaut #+ si la clause est publique et #- si la clause est privée. Par exemple, la propriété de VOLER des OISEAUX peut être vue comme privée aux AUTRUCHES.

5.3 Héritage partiel d'une méthode

Etant donné qu'une méthode logique est représentée par un prédicat et que une clause objet peut être privée ou publique, l'ensemble des clauses objets définissant une méthode logique peut contenir à la fois des clauses objets privées et des clauses objets publiques. Dans ce cas, seules les clauses publiques seront accessibles par héritage. Une telle méthode est dite partiellement héritable. Cette approche permet de modéliser un phénomène réel et fort intéressant dans la pratique, celui de l'héritage partiel d'un comportement. Dans les langages procéduraux, il n'y a aucun moyen de le faire puisqu'une méthode ne possède qu'une seule définition sous forme d'une procédure. Considérons par exemple les classes OISEAU et sa sous classes AUTRUCHE où OISEAU définit les propriétés suivantes:

```
#- mode_de_deplacement(X,voler) :- oiseau(X).        %les oiveaux volent
#+ mode_de_deplacement(X,marcher) :- oiseau(X).     % les oiseaux marchent
oiseau(o).                                          % o est un oiseau
oiseau(X):- autruche(X).                % les autruches sont des oiseaux
autruche(a).                                        % a est une autruche
```

Programme P8.

La propriété mode_de_deplacement(X,Y) de la classe OISEAU a deux clauses dont la première ne peut être accessible aux instances de la classe AUTRUCHE. Ainsi, au but autruche(X),mode_de_deplacement(X,Y)? le système répondra {X=a,Y=marcher} alors qu'au but oiseau(X),mode_de_deplacement(X,Y)? le système répondra {X=o,Y=voler} et {X=o,Y=marcher}.

5.4 Parcours de la hiérarchie d'héritage

La recherche d'une propriété commence dans la classe de départ[4]. Si la la propriété n'a pas été trouvée au niveau local[5], la recherche continue au niveau des superclasses de la classe de départ, mais en visitant uniquement les parties publiques de celles-ci. Un échec est déclenché si, jusqu'au niveau de la plus haute classe, la classe **obj(objetct)**, la propriété n'a pas été trouvée. Cette procédure est illustrée par la figure 3 ci-dessous.

---

[4]La classe de départ est soit celle désignée explicitement dans le protocole d'envoi de message (O as Une_Classe::BO@B), soit, par défaut, la classe d'instanciation de l'objet activé.

[5]Au niveau local, si la classe de départ correspond à la classe d'instanciation de l'objet courant, les deux parties (publique et privée) sont accessibles à celui-ci.

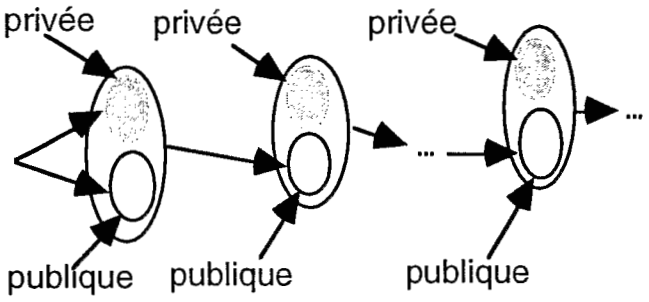

Figure 3: Procédure de parcours du graphe d'héritage.

6. Exemple simple d'un programme en WorldLog

La programmation en WorldLog consiste à définir des classes (éventuellement leurs instances) et à envoyer des requêtes aux objets créés dans le programmes ou dynamiquement. Illustrons cela par un exemple simple.

% Déclaration des objets (classes):

```
obj(meta_entreprise) with [class:=obj(class),supers:=[obj(object)]].

obj(entreprise_peugeot) with [class:=obj(meta_entreprise),supers:=[obj(object)]].
(obj(entreprise_peugeot),#+)&BO1@vend_voiture(V,P,BO2):V@BO1::change_proprietaire(P,BO),
obj(P)::BO@setv(voiture,obj(V),BO2).

obj(meta_voiture) with [class := obj(class),supers:=[obj(object)],couleur:=blanche].
(obj(meta_voiture),#+)&BO1@new(O,E,BO2):-self as class @ BO1 ::new(O,E,BO2).

obj(voiture_peugeot)with[class:=obj(meta_voiture),supers:=[obj(object)],
proprietaire:=obj(entreprise_peugeot)].
(obj(voiture_peugeot),#+) &BO@ton_proprietaire(P):-self@BO::getv(proprietaire,P).
(obj(voiture_peugeot),#+) &BO1@change_proprietaire(P,BO2):-self@BO1::setv(prop,P,BO2).
```

Programme P9.

% exemple de requête:

```
?- obj(voiture)::BO1@new(O,[],BO2),obj(entreprise_peugeot)::BO2@:vend_voiture(O,X,BO3),
X=obj(paul),O::BO2@ton_proprietaire(P_avant),O::BO3@ton_proprietaire(P_apres).
{O=obj(voiture0),X=obj(paul),P_avant=obj(entreprise_peugeot),P_apres=obj(paul)}
succes
```

7. Conclusion

Le but de ce papier était de présenter le langage de programmation par objets en logique, WorldLog, et, à travers sa réalisation, de proposer une autre approche de l'association des paradigmes de programmation en logique et de programmation par objets. Après avoir défini la syntaxe et les concepts objets du langage adaptés à la programmation en logique, nous nous sommes intéressés à la sémantique du langage. Nous avons défini une sémantique logique du langage basée sur un système modal défini par Warren. Contrairement aux approches impératives ou quasi-impératives, WorldLog a l'avantage d'avoir comme base sémantique un système logique bien défini, ce qui fait de lui un langage de programmation en logique au sens de Goguen et Meseguer. L'approche

utilisée ne consiste pas à privilégier un style de programmation, mais à tirer les avantages de chacun d'eux. Nous avons aussi évoqué quelques problèmes de réalisation dûs en particulier à la recherche d'un équilibre entre une sémantique logique et une implantation efficacité. Dans ce papier, nous nous sommes plus préoccupés de la sémantique logique du langage et nous avons peu insisté sur des aspects liés à l'efficacité du langage. Ainsi, en vue d'améliorer les performances du langage, nos travaux futurs porteront en particulier sur l'efficacité du langage. Nous avons également vu au cours de cette étude que l'association des deux paradigmes de programmation peut conduire à des modèles plus riches que les modèles de départ. WorldLog est écrit en Delphia-Prolog. Une implantation minimale tourne sur une machine UNIX, sous l'environnement Delphia-Prolog 2.5. Et, comme tout langage construit à partir d'un langage hôte, il n'est pas complètement unifié: les objets du langage WorldLog côtoient les entités classiques de Prolog qui gardent leur comportement habituel. Ceci est dû à notre volonté de permettre à l'utilisateur de représenter une entité par une structure Prolog ou par un objet de WorldLog. Le langage est donc accessible à la fois aux utilisateurs de la programmation par objets et à ceux de la programmation en logique.

Bibliographie

[AiLi88] Ait-Kaci, H. & Lincoln, P. "LIFE, A Natural Language for Natural Language". MCC Technical Report Number ACA-ST-074-88, Austin, Feb. 1988.

[AiNa86] Aït-Kaci, H. & Nasr, R. "LOGIN: A Logic Programming Langage with Built-in Inheritance". J. of Logic Programming 3, 3 (Oct. 1986), pp. 185-215.

[AiPo91] Ait-Kaci, H. & Podelski, A. "Towards a Meaning of LIFE". Proc. of the Thirsd Int'l Conf. on Programming Language Implementation and Logic Programmaing, Lectures Notes in Comp. Sciences, Passaü, Aug. 1991.

[Anje86] Anjewierden, J-M.. "How about Prolog Object?" Actes des 3è JLOO, Bigre + Globule N° 48, pp. 167-176, Paris, Janv. 1986.

[AnPa90] Andréoli, J-M. & Pareschi, R. "Linear Objects: Logical Processes with Built-in Inheritance". In 9th Conf. on Logic Programming, Jérusalem, Israel, 1990.

[AnPa91] Andréoli, A. & Pareschi, R. "Linear Objects: Logical processes built-in inheritance". New Gen. Comp., 1991.

[Brio84] Briot, J-P. "Instanciation et Héritage dans les langages orientés objets". Thèse de Doctorat, Paris VI, 1984.

[Brio85] Briot, J-P. "Les métaclasses dans les langages orientés objets". Actes du 5è CARFIA, pp. 755-764, Grenoble, 1985.

[Bobr88] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, & D.A. Moon. "Special Issue, Common Lisp Object System Specification, X3J13 Document 88-002R". ACM SIGPLAN Notices, 23, Sep. 1988.

[Bowe85] Bowen, K.A. "Meta-Level Programming and Knowledge Representation". New Gen. Comp. 3 (1985), pp. 359-383.

[BoWe85] Bowen, K.A. & Weinberg, T. "A Meta-Level Extension of Prolog". IEEE Int'l Symp. on Logic Programming'85 (1985), pp. 48-53.

[Chik84] Chikayama, T. "Unique Features of ESP". Proc. Int'l Conf. on Fifth Gen. Comp. Sys. (1984), pp. 292-298.

[ChWa88] Chen, W. & Warren, D.S. "Objects as Intensions". Actes Fifth Int'l Conf. on Logic Programming, pp. 404-419, 1988.

[Coin83] Cointe, P. "A VLisp Implementation of Smalltalk-76". In P. Degano and E. Sandewall, editors, Integrated Interactive Computing Systems, pp. 89-102, North-Holland, New York, 1983.

[Coin87a] Cointe, P. "Metaclasses are First Class: the ObjVLisp Model". in Proc. of OOPSLA'87, ACM SIGPLAN Notices 22, pp. 156-167, Orlando, Florida, Dec. 1987.

[Coin87b] Cointe, P. "The ObjVLisp Kernel: A Reflexive Lisp Architecture to define a Uniform Object-Oriented System". in P. Maes and D. Nardi, editors, Meta-Level Architectures and Reflection, pp. 155-176, North-Holland, Amsterdam, 1987.

[Cone88] Conery, J.S. "Logical Objects". Proc. of Fifth Int'l on Logic Prog., pp. 420-434, 1988.

[Davi88] Davison, A. "Polka: a Parlog Object-Oriented Language". Internal Report, Dept. of Comp., Imperial College, London, 1988.

[Colm83] Colmerauer, A., Kanoui, H. & Van Caneghem, M. "Prolog, bases théoriques et développement sactuels". Techniques et Sciences Informatiques, 2(4): 271-311, 1983.

[Delp92] Delphia-Prolog "Manuel Utilisateur, version 2.5". SLIGOS Agence DELPHIA, 1992.
[Doma86] Doma, A. "Object-Prolog: Dynamic Object-Oriented Representation of Knowledge". SzKi Comp. Research and Inn. Center (1986), 14 p.
[Doui93] Douin, J-M. "Objets, Programmation en Logique & Implantation Parallèle". Thèse de Doctorat, Conservatoire National des Arts et Métiers, Paris, Fév. 1993.
[DuHa89] R. Ducournau & M. Habib "La multiplicité de l'héritage dans les langages à objets". Techniques et Sciences Informatiques, 8(1): 41-62, 1989.
[DuHa92] R. Ducournau, M. Habib, M. Huchard, M. Mugnier & A. Napoli "L'héritage multiple dans tous ses états". Rapport Technique, LIRMM N° 92-021, Montrouge, Juil. 1992.
[Ferb87] Ferber, J. "Approches réflexives en informatique". Actes du Congrès COGNITIVA 87, pp. 402-407, Paris, La Villette, 1987.
[Ferb90] Ferber, J. "Conception et Programmation par Objets". Hermes 1990.
[FTKY84] Fukayama, K., Takeuchi, A., Kunifuji, S., Yasukawa, H., Ohki, M. & al. "Mandala: A Logic-based Knowledge Programming System". Actes Fifth Gen. Comp. Sys. Conf.'84 (1984), pp. 613-622.
[FuHi86] Fukunaga, K. & Hirose, S. "An Experience with a Prolog-based Oriented-Object Language". In Proc. of the 1th OOPSLA, pp. 224-231, Portland, Oregon, 1986.
[Gall86] Gallaire, H. "Merging Objects and Logic Programming: Relational Semantics, Performance and Standarization". In Proc. AAAI'86, pp. 754-758, Philadelphia, Pensylrania, 1986.
[GaMa89] Gandriau, M., & Massoutie, C. "Classes et Types: Aides à la Programmation Logique". Actes du 8è Séminaire de Programmation en Logique, CNET, pp. 57-69, Mai 1989.
[Gand88] Gandriau, M. "CIEL: classes et instances en logique". Thèse de Doctorat, ENSEEIHT 1988, 151 p.
[Gloes89] Gloess, P.Y. "ULog, Aspect Formels et Pratiques d'un Interface entre Programmation Logique et Objets". Actes du 8è Séminaire de Programmation en Logique, pp. 71-96, Mai 1989.
[Gloes90] Gloess, P.Y. "Contribution à l'optimisation de mécanisme de raisonnement dans des structures spécialiées de représentation de connaissances". Thèse d'état, Univ. de TechnWorldLogie de Compiègne, Janv. 1990.
[GoMe86] Goguen, J.A. & Meseguer, J. "EQLOG: Aquality, Types and Generic Modules for Logic Programming". Logic Programming (DeGroot & Lindstrom), pp. 295-363, 1986.
[GoRo83] Goldberg, A. & Robson, D. "Smalltalk-80: The language and its implementation". Addison-Wesley, 1983.
[GoSS92] Goldberg, Y., Silverman, W. & Shapiro, E. "Logic Programs with Inheritance". In Proc. of Int'l Conf. on Fifth Gen. Comp. Sys., ICOT 1992.
[Hato91] Haton, J-P., Bouzid, N., Charpillet, F., Haton, M-C., Lâasri, B., Lâasri, H., Marquis, P., Mondot, T., & Napoli, A. "Le raisonnement en intelligence artificielle". InterEditions, Paris 1991.
[HoMi90] Hodas, J.S. & Miller, D. "Representing Objects in a Logic Programming Language with Scoping Constructs". In Proc. of the seventh Int'l Conf. of Logic Programming, Jerusalem, Israel, 1990.
[KTMB86] Kahn, K., Tribble, E.D., Miller, M.S. & Bobrow, D.G. "Objects in Concurrent Logic Programming Languages". Actes de OOLPSA'86, ACM Sigplan Notices 21, 11(Nov. 1986), pp. 242-257.
[Kowa79] Kowalski, R. "Logic for problem solving". North-Holland, Amsterdam, 1979.
[LeMe88] 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
[MacC92] MacCabe, G.F. "Logic & Object". Prentice-Hall International, 1992.
[Male90] 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, Mars 1990.
[Male92] Malenfant, J. "Architecture méta-réflexives en programmation logique par objets". JFPL 92, pp. 253-267, 1992.
[Masi89] Masini, G. Napoli, A., Colnet, D., Léonard, D. & Tombre, K. "Les langages à objets". InterEditions, Paris, 1989.
[MaWa88] Manchanda, S. & Warren, D.S. "A Logi-based Language for Database Updates". Actes W. on Foun. of Ded. Db. and Logic Programming, pp. 363-394, 1988.
[NeWa88] Newton, M. & Watkins, J. "The Combination of Logic and Objects for Knowledge Representation". J. of Object-Oriented Programming, 1(4): 7-10, 1988.
[Robi65] Robinson, J.A. "A Machine-Oriented Logic Based On the Resolution Principle". J. ACM 12, pp. 23-41, Janv. 1965.
[Shap86] Shapiro, E. "Concurrent Prolog: A progress report". IEEE Computer 19, pp. 44-58, Aug. 1986.
[ShTa87] Shapiro, E. & Takeuchi, A. "Object-Oriented Programming in Concurrent Prolog". Collected Papers, Vol 2, Chapter 29, MIT Press, 1987.
[Stro89] Stroustrup, B. "Le Langage C++". InterEditions 1989.
[StSh90] Sterling, L. & Shapiro, E. "L'Art de Prolog". MASSON 1990.
[Wegn92] Wegner, P. "Object-Based Versus Logic Programming". In Proc. of the Int'l Conf. on Fifth Gen. Comp. Sys., ICOT 1992.
[Warr84] Warren, D.S. "Databases Updates in Pure Prolog". Actes of the Int'l Conf. on FGCS, pp. 244-253, 1984.
[Zanio84] Zaniolo, C. "Object-Oriented Programming in Prolog". In Proc. of the IEEE Internatial Symposium on Logic Programming, pp. 265-270, Atlantic City, New Jersey, 1984.