THESE DE DOCTORAT DE L'UNIVERSITE PARIS 6

Spécialité :
Informatique
Option :
Intelligence Artificielle

Présentée par
François PACHET

Pour obtenir le titre de **DOCTEUR DE L'UNIVERSITE PARIS 6**

Sujet de la thèse :

Représentation de connaissances par objets et règles : le système NéOpus

Soutenue le 25 Septembre 1992 devant le jury composé de :

M. Jean-François Perrot	*Président*
M. Jean Bézivin	*Rapporteur*
M. Félix Hautin	*Rapporteur*
Mme Marie-Christine Rousset	
M. François Rechenmann	

Résumé

Ce travail consiste à étudier l'intégration d'un mécanisme d'inférence d'ordre un en chaînage avant dans un environnement objet, au sens strict du terme. Cette intégration est étudiée du double point de vue implémentatoire et méthodologique. Le système ainsi conçu, NéOpus, est une reprise et une continuation du système Opus [Atkinson&Laursen 87], à l'origine de nos idées.
Nous énoncons les principes fondamentaux de l'intégration objet/règles (Principe de fermeture, Principe "Tout objet" et principe "Tout expression"). Dans un premier temps nous en étudions les conséquences sur notre technique d'inférence, et en déduisons de nouveaux mécanismes : extensions de la notion de variable de règle (variables locales, locales déclenchantes et objets nommés), notion d'héritage transposé aux bases de règles, architecture procédurale "objet" de contrôle. Dans un deuxième temps, nous mettons notre système à l'épreuve pour en explorer les possibilités et les limites. Nous en déduisons un principe d'utilisation (le principe de modified), et enfin apportons un élément de réponse au problème de la représentation de connaissances *sur* un univers d'objets, en introduisant la notion d'aspect.

Conventions typographiques

Le texte général est en palatino 12.
`Le code Smalltalk est en Courrier 10.`
Les règles NéOpus sont encadrées et en Zapf Chancery 12.

Remerciements

Je tiens à remercier particulièrement Jean-François Perrot pour avoir accepté de diriger cette thèse. Parmi ses nombreuses qualités (de son amour du langage et des langues, à sa patience vis à vis des fautes d'orthographe) j'admire beaucoup sa capacité infinie de pourfendre tous les cadres conceptuels qui passent à sa vue, en trouvant immédiatement des contre-exemples soulignant toujours le caractère hautement prétentieux et microscopique des systèmes d'Intelligence Artificielle.

- Bon, votre système, c'est très bien, mais comment vous faîtes pour dire qu'un triangle équilatéral, c'est un triangle isocèle *pour chacun de ses sommets !*. Hein ?
- Ben, je peux pas le dire....

Je remercie Jean Bézivin d'avoir accepté de rapporter cette thèse. Ses travaux et son enthousiasme pour la programmation par objets ont fortement contribué à rendre Smalltalk populaire.

Je remercie Félix Hautin d'avoir accepté de rapporter cette thèse et de m'avoir donné ainsi son point de vue sur les applications objets d'échelle industrielle.

Je remercie vivement tout la communauté d'utilisateurs de NéOpus, qui ont toujours su souffrir avec bonne volonté sur le système : Thierry Fuhs, Isabelle Alvarez, Sophie Rougegrez, Amel Benhouhou, Sophie Charbonnel, Nicolas Revault, Michel Dojat, les Nantais de l'équipe IAMI : Anne-Marie Forte et Frank Lavaire, et tous ceux qui sont passé entre les mains du système (ou est-ce l'inverse ?) d'une manière ou d'une autre, contribuant ainsi à le faire vivre.

Tous les membres du LAFORIA avec qui j'ai pu avoir avoir des discussions irremplaçables, en particulier Gilles Blain, Michel De Glas, Philippe Laublet.

Je m'agenouille devant mes relectrices et relecteurs : I. Borne, M. Dojat, A.-M. Forte, P. Laublet, N. Revault, E. de Vitry.

Je voudrais aussi remercier l'équipe Smalltalk de la compagnie ACKIA qui m'a appris beaucoup en me faisant profiter de leur grande expérience de la programmation : Philippe Krief, Robert Voyer et Christian Dujardin.

Jacqueline LeBaquer, Patricia Avion et François Dathanat pour leurs patiences et leurs efficacités légendaires.

Enfin je remercie mes parents et amis qui m'ont supporté pendant cette période.

I. Introduction

I.1. Préambule

L' Intelligence Artificielle, à travers toutes ses représentations, ses systèmes, ses tentatives, et ses échecs, peut être perçue, par les informaticiens, comme une vaste manœuvre de *compilation* du langage naturel. Le problème de l'Intelligence Artificielle est bien en effet de "faire rentrer dans la machine" des *connaissances*, utilisables par celle-ci pour "résoudre des problèmes", et donc de trouver des langages, des cadres conceptuels, des environnements de programmation, bref de fabriquer des super-compilateurs, permettant de passer du langage naturel, seule source utilisable de connaissance, à une représentation informatique. Bien entendu, le mot *connaissance* est vague, imprécis, ambigu, à bien des égards allergène, mais hélas pour l'instant indispensable. Nous adopterons comme postulat que la connaissance *existe*, et nous intéresserons à un type particulier de connaissances, les connaissances dites *inférentielles* dont nous ne donnerons qu'une définition fuyante : "une connaissance pouvant produire des connaissances ".
L'inférence, est un mécanisme qui peut être en effet considéré comme le centre de gravité des préoccupations de l'Intelligence Artificielle. D. Kayser propose dans [Vergnaud 91] d'adopter l'inférence comme unité de mesure pour toutes les sciences cognitives (I.A, psychologie, linguistique, neuro-sciences), établissant un parallèle avec la *cellule*, unité commune de toutes les sciences naturelles.
Si la connaissance résiste aux tentatives de classification, l'inférence, elle, se laisse plus facilement classifier. Divers types d'inférences sont reconnus. [Lenat&Feigenbaum 91] affirment même que tous les mécanismes d'inférence sont connus : déduction, abduction, induction, permettent en théorie de représenter toute activité inférentielle.

Mais ce qui nous préoccupe ici est, non pas simplement le *mécanisme* d'inférence, mais la façon dont il *opère* sur les concepts mis en jeu, souvent appelés les "*objets du domaine*". Plus précisément, nous nous intéressons au rapport entre la représentation du monde, et la représentation d'un *discours* sur ce monde. Parmi les nombreux systèmes informatiques inférentiels existants, rares sont ceux qui étudient l'activité inférentielle comme une activité véritablement *opératoire*. Un système comme Prolog, par exemple manipule très bien des énoncés (des clauses) qui énoncent des propriétés du monde, mais ne peut pas manipuler autre chose : les énoncés vivent par eux-mêmes, et le monde modélisé n'existe pas vraiment, ou n'existe qu'à travers les énoncés qui le décrivent. Cette disparition du monde au profit du discours est déjà amorcée par la logique classique, qui d'emblée considère la description du monde ou "langage des objets" [Hottois 89], comme un simple marche-pied obligatoire pour la construction de langages de types supérieurs, qui seuls méritent l'attention.
Les systèmes à règles de production, eux, sont *opératoires* par définition, puisqu'ils proposent d'emblée un découpage naturel entre la représentation du monde (sous

forme de "faits" plus ou moins évolués), et une représentation d'un discours sous forme de règles de productions opérant sur ces faits. Mais cette représentation du monde est le plus souvent très embryonnaire ou ad hoc. Les énoncés qu'ils manipulent se réduisent alors le plus souvent à de simples manipulations d'attributs. Surtout, la représentation du monde est conçue *a posteriori*, et dictée par la nature même des *règles*.
Notre idée est que les langages objets sont une très bonne base pour représenter les "objets du domaine" en offrant une représentation du monde fondée sur la *simulation*, et qu'il est intéressant de partir de cette base pour construire un système inférentiel cohérent, plutôt que d'adopter la démarche inverse.
Notre étude est donc l'intégration d'un mécanisme d'inférence dans un univers objet.

I.2. Notre héritage culturel

I.2.1. Pas d'introduction à l'Intelligence Artificielle

La rédaction de chapitres plus ou moins longs sur l'Intelligence artificielle *en général* semble être un exercice de style prisé de l'école française. De nombreux textes [Ganascia 91], [Vogel 88] [Laurière 87], [Hofstadter 85] débattent du problème du statut, de la définition, de l'existence et des projets de l'I.A. Nous ne nous risquerons pas à ces périlleux exercices. Par pur esprit de contradiction, nous proposons donc ici un non-chapitre sur ce thème, et renvoyons le lecteur intéressé aux références.
En revanche, nous rappellerons la définition du *raisonnement* donnée par Lalande (dictionnaire de Philosophie) :

> Raisonnement :
> Opération discursive par laquelle on conclut qu'une ou plusieurs propositions (*Prémisses*) impliquent la vérité, la probabilité ou la fausseté d'une autre proposition (*Conclusion*).

On voit déjà dans cette définition poindre la notion de règle qui sera au cœur de nos propos. Un des nombreux problèmes soulevés par ce genre de définition est alors, comme le souligne [Grize 90], de comprendre le sens du mot "implication" qui peut prendre, selon les cas, des significations plus ou moins précises (par exemple dans le cas de l'induction ou du raisonnement par analogie). Nous nous bornerons à ne garder de cette *intuition* du raisonnement, que l'idée d'un *mouvement* de gauche à droite, et nous verrons qu'un des problèmes de notre approche vient en particulier de la profonde *dissymétrie* des parties gauches et droites.

I.2.2. Des règles en général

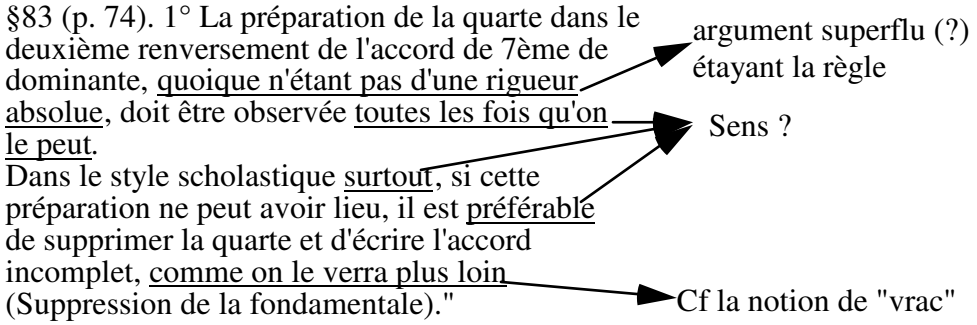
Nous parlerons donc de règles, comme support principal et privilégié de représentation d'un certain type de discours sur le monde. Ici encore, nous esquiverons la définition, et éviterons les problèmes de nature (qu'est-ce qu'une règle ?). Prenons simplement comme postulat que la règle existe. Nos activités d'êtres rationnels sont remplies de règles de natures très diverses. Mais les règles de la vie courante se prêtent mal à la classification. Montrons le par deux exemples.

La grammaire russe, comme celle de la plupart des langues indo-européennes, s'énonce principalement sous forme de règles. Par exemple, celle de la "palatalisation des consonnes" qui indique comment certaines consonnes se transforment au contact d'autres lettres (après un Yod, "k" se transforme en "x", "u" en ";", "[" en "i"). Mais la règle fondamentale de la grammaire est sans doute celle qui décrit les procédés de perfectivation / imperfectivation des verbes. Elle s'énonce comme ceci [Mazon 78] :

> Les verbes *composés* sont soit perfectifs soit imperfectifs, suivant le procédé d'après lequel ils ont été formés. Les uns proviennent de l'apposition d'un préverbe à un verbe simple, et ils sont perfectifs. C'est le préverbe qui dans ces composés détermine l'aspect : ceux-ci sont des *composés préverbaux*.
> Les autres composés sont dérivés directement de verbes eux-mêmes composés, par le moyen de certains suffixes : -a-, -ba-, ou -uba- [...]. Ainsi [...] ces procédés sont complémentaires l'un de l'autre : la *composition préverbale*, qui est un procédé de perfectivation, et la *dérivation suffixale*, qui est un procédé d'imperfectivation.

Ici encore, on voit bien comment un énoncé qui traduit une *intention de régler,* ne se prête pas à la représentation sous forme de règle(s) de production, mais qu'en revanche, l'idée peut s'exprimer par un certain *comportement simulatoire* des objets du discours (en l'occurrence les verbes). En clair, le "procédé d'imperfectivation" s'apparente ici plus à une *méthode*, au sens objet, qu'à un ensemble de règles de production.

L'harmonie classique est un domaine où la règle joue un rôle central. Pratiquement toute la théorie de l'harmonie occidentale (exception faite du solfège) peut s'exprimer sous forme de règles dont voici un exemple tiré du traité d'harmonie de [Dubois 21], une des références en la matière :



Cependant, la saveur de cette règle ne s'apprécie qu'à la lecture de la préface du-dit traité, où il est fourni une sorte de mode d'emploi des règles, sous forme de super méta-règles dans le but de réduire les règles à leur simple aspect didactique :

> ... Notre enseignement, dans son ensemble, n'est pas limité aux règles du *style rigoureux*, bien qu'elles en forment le fond essentiel, mais il s'efforce d'appliquer les combinaisons et les ressources dont l'art musical s'est enrichi peu à peu, et qui constitue ce qu'on est convenu d'appeler le *style libre*. L'élève doit être capable, à la fin de ses études, d'analyser les hardiesses et les licences qui se rencontrent souvent dans les œuvres des plus grands maîtres et qui paraissent en contradiction avec l'enseignement qu'il reçoit.
> Il doit pouvoir se rendre compte de tout et comprendre pourquoi le génie s'est quelquefois affranchi avec bonheur de la rigueur des règles nécessaires aux études classiques.

Nous verrons au chapitre VII.5 une application de NéOpus à un problème d'analyse harmonique qui représente un certain type de règles d'analyse.

Mais la connaissance humaine progresse, même en matière de musique. Boulez dans [Boulez 63] donne une vision très unifiée et générale de la musique occidentale basée sur la notion de "série". Voici la définition qu'en donne Boulez:

> La série est - de façon très générale - le germe d'une hiérarchisation fondée sur certaines propriétés psycho-physiologiques acoustiques, douée d'une plus ou moins grande sélectivité, en vue d'organiser un ensemble FINI de possibilités créatrices liées entre elles par des affinités prédominantes par rapport à un caractère donné ; cet ensemble de possibilités se déduit d'une série initiale par un engendrement FONCTIONNEL (elle n'est pas le déroulement d'un certain nombre d'objets, permutés selon une donnée numérique restrictive). Par conséquent, il suffit, pour instaurer cette hiérarchie, d'une condition nécessaire et suffisante qui assure cohésion du tout et relation entre les parties consécutives ...

Mais cette définition, très générale, est du coup difficilement utilisable. Cet exemple conduit à penser que, pour être vraiment utilisable une règle doit être incomplète, sinon sa trop grande généralité rend sa représentation impossible.
La difficulté du projet de régler est de trouver un compromis entre un énoncé tout instancié et donc inutilisable, et un énoncé trop général, difficile à instancier.

Bref si elle est un objet séduisant, la règle peut aussi écarter de la vérité[3], comme le souligne [Thirouin 89] (p. 69) (à propos du souci omniprésent chez Pascal de *trouver les règles*) :

> Derrière le projet de régler, on croit toujours surprendre chez Pascal le soupir : "faute de mieux!". L'objectif est jugé digne qu'on y emploie ses efforts, mais l'art, à chaque fois, se révèle en dernier ressort inutile, puisqu'il laisse l'homme dans l'ignorance des vérités essentielles.

I.2.3. Des règles de production en particulier

[3] C'est d'ailleurs la véritable étymologie du mot séduire (du latin : seducere écarter du chemin).

Mais dans notre cadre de l'Intelligence Artificielle, nous ne parlons pas de n'importe quel type de règle, et là est bien le problème. Nous parlons d'un type de règle bien particulier qu'est la *règle de production* ou *action conditionnelle* [Pitrat 90]. Une règle de production est en effet un objet *hybride* composé de deux parties. La partie condition *décrit un certain état* du monde, ou plutôt des objets du monde. La partie droite est, en revanche, une *action* sur ce monde, ayant pour effet de modifier les-dits objets (ou d'autres). La partie action de la règle est déclenchée quand la partie condition est satisfaite.
Ainsi, voit-on apparaître le problème principal de la représentation par objets/règles de production. Il s'agit de transformer un *discours* sur le monde en *action* sur ce monde. Ceci se fera bien tant que les règles à représenter seront effectivement des actions conditionnelles (comme par exemple les règles de diagnostic médical, d'analyse, de transformation (Cf. chapitre VII)), beaucoup moins bien dans d'autres cas.

I.2.4. Les langages à objets

I.2.4.1. Un peu d'étymologie

Revenons rapidement sur l'étymologie de mots qui nous entourent. *Objet*, du latin ob-jet, signifie "*jeté devant soi*". Cette étymologie le rapproche de *problème*, du grec προ–βαλειν, qui veut dire sensiblement la même chose (lancé devant). Dans les deux cas, il y a l'idée d'une projection (jeter, lancer) et d'un face à face (pro, ob) entre jeteur et jeté. Or objets et problèmes font effectivement bon ménage, comme si le fait d'exprimer un problème le résolvait déjà d'une certaine manière. En effet, comme le montre [Laurière 87] bon nombre de problèmes compliqués trouvent leur solution dans la *reformulation*. La reformulation dite "objet" est souvent le pas décisif vers la résolution du problème.

I.2.4.2. Une vision des langages objets

Les langages objets nous semblent les plus adaptés à produire une représentation des objets du domaine suffisamment riche pour pouvoir porter et supporter une couche de représentation discursive. Nous nous plaçons dans le cadre des objets "au sens strict", dont on peut trouver un exemple prototypique dans les langages ObjVlisp [Cointe 84], et Smalltalk [Goldberg&Robson 83]. En particulier, nous nous démarquons des systèmes se réclamant objets mais suivant une orientation plus proche de celle des frames.
Parmi les nombreuses qualités des langages à objets, nous retiendrons celles qui effectivement leur confèrent une place de premier plan pour la représentation de connaissances.

I.2.4.2.1. La notion d'abstraction/instanciation

Les objets sont, et sont instances d'une classe. Le mécanisme d'instanciation nous donne une représentation réductrice mais satisfaisante de la notion d'être. Cette représentation est satisfaisante parce qu'elle correspond à un mécanisme de pensée naturelle, qui différencie l'existence de l'essence [Perrot 86], mais réductrice parce que la notion d'être peut cacher des notions plus complexes (Cf. les problèmes posés par la représentation

d'énoncés du type "les étudiants sont jeunes" et les logiques non classiques [LéaSombé 89]).

I.2.4.2.2. Le principe de fermeture

Parmi toutes les qualités reconnues des langages objets, et en particulier de Smalltalk, nous en retiendrons une qui sera déterminante par la suite, et que nous appelerons principe de *fermeture*[4] : c'est l'idée qu'un objet *cache* sa structure aux autres objets, puisqu'il n'est accessible que via son interface (l'ensemble des messages que sa classe ou ses super classes connaissent).

I.2.4.2.3. La notion de réutilisabilité

Smalltalk, comme tous les langages objets, n'échappe pas aux exigences de la réutilisabilité[5]. La notion sera étendue à la celle de la *réutilisabilité de bases de connaissances*. Réutilisable ou simplement utilisable ?

I.2.4.2.4. Héritage et programmation par l'exemple

L'héritage est un mécanisme reconnu comme étant à la fois très utile et très dangereux, en particulier lorsqu'il est utilisé pour représenter des taxonomies naturelles de concepts. Nous ne reviendrons pas sur les multiples problèmes posés par ce mécanisme [Masini&al. 89], [Ducourneau&Habib 89], [Carré 89]. Mais nous garderons l'idée fondamentale à notre sens de *l'héritage comme moyen de programmation par l'exemple*[6]. En effet, l'héritage permet de proposer aux programmeurs des composants à partir desquels il est plus facile de fabriquer les siens propres. Nous retrouvons cette idée dans le cadre de l'acquisition de connaissances : elle est à la base de la méthodologie d'acquisition de connaissances du système Cyc de D. Lenat (système de représentation du sens commun), pour lequel il est dit [Lenat&Guhat 90a] :

"Most Knowledge Entry in Cyc involves finding similar knowledge and copying it and modifying the copy"[7].

---

[4] Les informaticiens parlent souvent de système "transparent" pour désigner un système qui justement ne l'est pas, et cache à l'utilisateur des choses essentielles mais que l'on peut se dispenser de connaître, comme par exemple dans la phrase : "La mémoire virtuelle et/ou le mécanisme de swap sont transparents pour le programmeur".
On retrouve une confusion similaire dans la croyance populaire que le sens du mot russe ukfcyjcnm (*glasnostc)* est *transparence* alors qu'il signifie au contraire *ouverture* .

[5] La réutilisabilité et Smalltalk ont au moins en commun d'être tous les deux de vastes programmes.

[6] Dans un sens exactement opposé au sens traditionnel : ici c'est le système qui donne l'exemple, et non le programmeur.

[7] Ceci est d'ailleurs source de beaucoup d'erreurs, la pratique intensive du copier/coller ayant tendance à laisser des doubles difficiles à repérer. Mais cela peut être un avantage comme il est dit plus loin à propos d'un module chargé de repérer justement des régularités suspectes :

This program roams over the knowledge base, typically at night, looking for unexpected symmetries and asymmetries. These in turn often turn out to be bugs, usually crimes of omission ... In rare cases today and, we expect, more frequently in future years, these turn out to be **genuine little discoveries of useful but hitherto unentered knowledge**

I.2.4.3. Election de Smalltalk

Parmi les langages objets rentrant dans notre catégorie stricte, notre choix de Smalltalk se justifie par plusieurs arguments :

Smalltalk : premier de la classe des langages objets

Smalltalk est la meilleure preuve des qualités des langages objets puisque le langage et l'environnement sont entièrement décrits en Smalltalk (ou presque). L'environnement Smalltalk est sans doute à ce jour le plus gros programme existant écrit dans un langage à objet. D'autre part l'uniformité des concepts et l'absence de langage procédural autre que celui induit par l'envoi de message nous fournissent une plate-forme propre et solide.

Un vaste environnement source d'exemples

Smalltalk nous fournit un terrain d'expérimentation très riche dans lequel on pourra puiser toutes sortes d'exemples non triviaux de représentation (en particulier en ce qui concerne la gestion des objets graphiques, des contrôleurs, de l'environnement). Plus qu'un simple langage, c'est une culture que nous adoptons.

Une programmation par métaclasses

Smalltalk propose un mode de programmation par métaclasse réduit [Cointe 87], mais néanmoins très souple. Une des grandes qualités de Smalltalk est justement de permettre, via les métaclasses, de redéfinir proprement le comportement de certaines classes du système, sans remettre en question celui des autres[8] [Pachet 89].

Smalltalk est donc utilisé dans notre contexte, à la fois comme langage d'implémentation, et comme langage de représentation. En effet, l'implémentation Smalltalk (ou objet) d'un concept sert à la fois comme mécanisme d'implémentation au sens traditionnel, et comme mini-langage de description : l'objet est défini par ses méthodes (son *interface*), à la fois pour la machine mais aussi pour les autres objets et pour l'utilisateur.

I.3. Notre étude

[8] Bien que l'utilisation intensive de ces techniques conduise à d'autres problèmes [Graubé 89b].

Notre étude consiste donc à greffer au dessus de Smalltalk, considéré comme une première couche de représentation de connaissances, une seconde couche de représentation à caractère discursive, basée sur la notion de règle de production.

Chapitre II : Etat de l'Art
Nous présentons notre héritage culturel double : celui des objets au sens strict, plus particulièrement du monde Smalltalk, et celui des inférences en marche avant. Nous situons nos travaux et notre problématique comme une intégration raisonnable de ces deux univers.

Chapitre III : Opus in vivo
Nous y greffons une première couche, appelée Opus, système d'inférence avec variables en marche avant, décrit par [Atkinson&Laursen 87]. Nous commencerons par décrire notre implémentation d'Opus de laquelle émergeront les acteurs essentiels de notre univers, réifiés et dûment dotés de micro-langages : la règle, le conflict set, la base de règles, la règle déclenchable, le token, la liste de tokens, le réseau Rete. Nous dégagerons aussi les principes de base de notre philosophie : parler de tous les objets, utiliser toute expression Smalltalk dans les règles.

Chapitre IV : NéOpus
Nous gravissons alors une marche supplémentaire, qui constituera le système NéOpus. Nous ajoutons au système Opus des notions et mécanismes nouveaux, nés de l'alliance règles/objets, à savoir : l'extension de la notion de variable de règle par l'introduction des notions de variable locale, variable locale déclenchante et d'objet nommé; le mécanisme d'héritage de bases de règles, la notion de typage simple/naturel. Le système ainsi enrichi est baptisé NéOpus.

Chapitre V : Praxis
Puis nous mettons en pratique NéOpus en étudiant l'écriture de bases de règles diverses, pour le mettre à l'épreuve de manière systématique. Nous en dégagerons des principes d'utilisation, et aussi les limites de notre représentation: manque de pouvoir assertionnel, et manque de contrôle sur le contrôle.

Chapitre VI : Le contrôle
Nous présentons ici la double solution NéOpus au problème du contrôle en marche avant. La première, procédurale, est une application des principes de la programmation par objet et métaclasses au problème du contrôle. La seconde, déclarative, est une utilisation de NéOpus lui même pour résoudre le problème du contrôle en NéOpus. Nous introduisons alors la notion d'*assertion*, qui nous permet de *parler* de l'*action* d'une règle.

Chapitre VII : Applications de NéOpus
Nous présentons divers travaux mettant NéOpus à l'épreuve dans des contextes variés. Ce chapitre permet de souligner les caractères importants du système tout en dégageant une certaine philosophie de la programmation par règles et objets.

Chapitre VIII : Vers un schéma triadique
Nous présentons ici une *tentative* de généralisation de notre démarche d'utilisation du système en introduisant la notion abstraite d'*aspect*.

Chapitre IX : Conclusion
Nous concluons en rappelant les nombreux problèmes rencontrés au cours de notre travail, et dégageons de ceux-ci les problèmes de représentation qu'il nous semblent pertinent d'explorer.

I.4. Vocabulaire de base

Nous renvoyons à [Masini&al. 89, Goldberg&Robson 83] pour les notions de bases de la programmation objet au sens strict. On trouvera dans [Wolinski 90] un vocabulaire minimal "de voyage" pour les Smalltalkiens de passage.
Nous rajouterons simplement quelques définitions plus formelles de notions couramment employées : `pointerSur:`, `connait:` et `estConnuDe:`, sous forme de sélecteurs de méthodes (le code importe peu ici) implémentées dans la classe racine `Object` :

```
pointeSur: unAutreObjet
"je pointe sur unAutreObjet, si une de mes variables d'instance a comme valeur cet autreObjet"
```

```
connait: unAutreObjet
"c'est la fermeture transitive de pointerSur:"

"je connais un autreObjet si je pointe dessus, ou bien si une de mes variables d'instance connait unAutreObjet"
```

```
estConnuDe: unObjet
  ^unObjet connait: self
```

II. Etat de l'Art

Avant Propos

Nous faisons dans ce chapitre un tour d'horizon des différents systèmes alliant objets (ou structures de données se réclamant telles) et règles de production, en chaînage avant. Nous concluons sur l'absence de système alliant de *véritables* objets, au sens défini plus haut, et règles de production d'ordre 1 en chaînage avant, ceci justifiant l'originalité du système Opus.

II.1. Systèmes à règles de production

II.1.1. Léger historique

La notion de règle de production connaît un franc succès en Intelligence Artificielle depuis les premiers et fameux systèmes experts (système médical Dendral et chimie organique Mycin [Shortliffe 76]). Ces systèmes introduisent alors la règle de production comme entité représentant un petit savoir granulaire ou "unités de savoir-faire" [Farreny&Ghallab 87]. Ces systèmes donnerons naissance à une fantastique dynastie d'autres systèmes experts qui reprendront l'idée en l'étendant (comme par exemple Mycin avec la notion de coefficient de vraisemblance). Mais l'idée de base est toujours la même : associer dans une seule entité une partie condition et une partie action. Les grands changements interviendront dans la représentation des objets du domaine.

II.1.2. Une histoire d'ordres

Une manière classique de distinguer les capacités des systèmes inférentiels est de comparer le statut des variables apparaissant dans leurs énoncés. Bien qu'il n'y ait pas de réel consensus sur ces appellations, on distingue trois catégories de représentation. Nous prenons l'exemple de la représentation de l'assertion "les étudiants sont jeunes" [Léa Sombé 89] dans les trois types de formalisme.

Ordre 0

Il n'y a pas de variables dans les énoncés. Dans le cas de systèmes à règles de production, les règles peuvent s'apparenter aux implications de la logique des propositions.

Par exemple : Si 'étudiant' alors 'jeune'

Ordre 0 plus

Les variables servent à dénoter les *valeurs d'attributs* pour un objet unique mais imaginaire. Dans le cas des règles de production, on obtient des systèmes plus souples que les précédents, mais difficilement utilisables car tous les attributs sont au même niveau. Les individus n'existent toujours pas vraiment, et l'intérêt des variables est alors discutable.

Par exemple : si (profession = 'étudiant') alors (âge := 'jeune')

Ordre 1

La notion d'ordre 1 provient de la notion logique d'ordre, qui consiste à introduire dans le vocabulaire des *variables* quantifiables universellement, et qui peuvent dénoter tout individu du monde représenté. Le système d'ordre 1 prototypique est la logique des prédicats du premier ordre. Les systèmes comme Snark ou OPS5 suivent aussi la voie de la variabilisation des individus, bien que de manière fort différente, comme on le verra au Chapitre III. Il est important de noter ici que la notion d'ordre un concerne le *statut des variables* apparaissant dans les expressions des règles, et ne concerne pas en particulier la *nature des objets dénotés par ces variable*s. Prolog, Snark, OPS5, mais aussi les systèmes basés sur des mécanismes d'induction ou de généralisation [Vere 80, 87] entrent tout aussi bien dans cette catégorie, ainsi que d'une certaine manière toute la programmation traditionnelle [Perrot 89b].

Exemple :

```
si      x est un Individu
et      la profession de x est 'étudiant'
alors l'âge de x est : 'jeune'
```

D'autres ordres sont alors envisageables. L'ordre 2, en particulier permet de référencer non pas les individus mais les relations elles-mêmes (Snark, Oks). [Perrot 89b] propose même de continuer sur cette lancée avec des ordres supérieurs permettant de parler de groupes d'individus, de groupes de groupes... Nous verrons avec l'exemple des "n reines" (VI.1.1.3.1) une limite concrète de la notion d'ordre un.

II.2. Systèmes hybrides

La notion de système hybride n'est pas non plus très précise. On peut dire de manière générale qu'il y a système hybride à partir du moment où deux formalismes de représentation sont associés. La question est alors de définir cette association : est-ce une simple cohabitation, une agrégation ou une véritable osmose ?
Le premier système hybride est sans doute le système LOOPS [Bobrow&Stefik 83], [Loops 87] qui intègre dans un même environnement plusieurs mécanismes de représentation : fonctionnelle (par le langage lui-même : Inter-Lisp D), par objets, par règles de production, valeurs actives, ainsi qu'une série d'autres mécanismes (comme par exemple la notion de hiérarchie de partie). Son principal défaut est sans doute de n'avoir jamais été accessible sur d'autres machines que les machines Xerox, qui n'ont jamais eu le succès escompté.

II.2.1. Réseaux sémantiques

Les réseaux sémantiques, dont le prototype est sans doute KL-One [Brachman-Shmolze 85] sont des systèmes hybrides par nature, puisqu'ils combinent d'une part

une *représentation structurelle* du monde complexe, sous forme d'un réseau de concepts, incluant des relations de tous ordres (instanciation, héritage, hiérarchies de parties, taxonomies de divers types), avec d'autre part des *représentations assertionnelles* sur les concepts ainsi organisés. Un des principaux défauts des premiers réseaux sémantiques était justement d'autoriser toutes sortes de relations entre concepts [Brachman 77], rendant la consistance du réseau et son interprétation très difficile. Les récents travaux sur les réseaux sémantiques tendent au contraire à contraindre plus fortement ces relations en les dotant d'une sémantique plus précise. Le système ICEO [Bourgeois 90] propose un formalisme de représentation basé sur un réseau sémantique dans lequel simplement les deux relations d'*attribution* et de *subsomption* sont gardées, et montre comment le modèle gagne du coup en clarté sans perdre en puissance d'expression.

II.2.2. Systèmes à base de frames

II.2.2.1. Art

Le système Art [Art 87], [Laurent&al. 87] est une référence en matière de système hybride. Bâti au dessus de Common Lisp, il intègre plusieurs paradigmes puissants de représentation de connaissances, comme les objets, les règles en chaînage avant et arrière, un mécanisme de maintien de la vérité et de gestion d'hypothèses.

Plus précisément, la connaissance déclarative dans Art est représentée par quatre types d'entités : les *faits*, les *schémas*, les *règles* et les *points de vue*.

II.2.2.1.1. Les faits et les schémas Art

Les faits (appelés aussi *propositions*) sont des relations entre objets exprimées sous la forme de n-uplets, dont le premier terme est le nom de la relation, comme par exemple : `(pereDe Mary John)`.
Les schémas sont une structuration de faits, plus commode à manipuler que les faits eux-mêmes. Un schéma est une entité nommée comprenant des *slots* qui sont des couples attribut/valeur.
Par exemple, le schéma suivant `bureauRegional` (tiré de [Art 87]) contient 5 slots:

```
(defschema bureauRegional
  (instanceDe bureau)
  (location        Atlanta)
  (personnel       15)
  (directeur  Miles)
  (brancheDe  BanqueCentrale))
```

Le système Art transforme immédiatement tout schéma en faits, en remplaçant chaque slot du schéma par un fait dont la relation (le premier terme) est le nom du schéma. Les noms des slots sont donc simplement des noms de relation, et il n'y a pas de différence opérationnelle entre les deux.

Ici nous avons donc une équivalence entre le schéma `bureauRegional` et les 5 faits suivants :

```
(instanceDe bureauRegional bureau)
(location   bureauRegional Atlanta)
(personnel  bureauRegional 15)
(directeur bureauRegional Miles)
(brancheDe bureauRegional BanqueCentrale)
```

Art distingue les relations pointant vers des valeurs atomiques (appelées *attribute slots*) des relations pointant vers d'autres schémas (les *relation slots*). Dans notre exemple tous les slots sont des *attribute slots* sauf `instanceDe` qui est un *relation slot* prédéfini. Un certain nombre de *relation slots* sont prédéfinis, comme `instanceOf`, `isA`, `subsetOf`, `prototypeOf`, mais l'utilisateur peut à loisir définir toutes sortes de tels attributs.

II.2.2.1.2. Les règles Art

Les règles Art sont soit des règles en chaînage avant, effectuant des effets de bord sur la base de faits, soit des règles en chaînage arrière, tentant de satisfaire un but donné. Elles utilisent des *patterns* qui sont des descriptions abstraites de faits. Les patterns les plus courants sont les *patterns de restriction*, établissant des contraintes sur les faits/schémas filtrés. Ces restrictions sont des filtres homéomorphes aux faits, mais pouvant comporter des variables pour référencer les valeurs des attributs des faits.
Il existe aussi des restrictions dites *predicate restrictions* permettant d'appeler une expression Lisp pour tester une valeur d'attribut. Le prédicat est supposé ne pas faire d'effets de bord et rendre un résultat `nil` ou `false` (les valeurs logiques vrai et faux en Lisp).

II.2.2.1.3. Critique

Le système Art transforme systématiquement toutes les représentations structurées (schémas) en faits, à partir desquels il offre une panoplie considérable d'opérations, grâce à des filtres très puissants, reposant sur des techniques de compilation de règles éprouvées[7].
Le pari de Art est ainsi de "sacrifier" la représentation des objets du domaine pour la faire coller, via des transformations syntaxiques, à une représentation à base d'attribut/valeur. Cette démarche permet effectivement de bénéficier de tous les travaux basés sur la représentation attribut/valeur et de les faire cohabiter dans un même environnement sans problème conceptuel (compilation Rete, ATMS, mécanismes de gestion d'environnement). Cependant, nous ne pouvons parler ici de langage de représentation objet, dans la mesure où le lien d'instanciation, le lien d'héritage, la notion de méthode/comportement associé à une classe, et les principes

[7] Cette technique est appelée "Destructuration des objets" dans [Voyer 89] p. 131.

qui en découlent (encapsulation, modularité, réutilisabilité) disparaissent, pour faire place à des batteries gigantesques de faits sans structure.

II.2.2.2. Smeci

Smeci est un générateur de systèmes Experts développé par ILOG [Corby 87, Smeci 88]. Il propose une représentation des objets sous forme de trois types d'entités : les *catégories*, les *prototypes* et les *objets*.

II.2.2.2.1. Objets Smeci

Les catégories sont des sortes de classes au sens de la programmation par objets. Elles décrivent les champs de leurs instances (appelés *objets* en Smeci). Les catégories peuvent aussi contenir des *méthodes*, qui sont des fonctions Lisp, locales à une catégorie. Un mécanisme d'héritage permet aux catégories d'hériter des champs et méthodes des super-catégories.
A chaque catégorie est associé un prototype, qui contient les informations communes aux diverses instances de la catégorie. Ces informations sont soit des valeurs partagées par toutes les instances, soit des contraintes pour les valeurs de champs.
Les objets sont des instances de catégories, au sens de la P.P.O. Le mécanisme d'instanciation en Smeci prend en compte la notion de prototype pour initialiser les valeurs des champs. Plusieurs options sont possibles pour récupérer les valeurs prototypiques ou non.

II.2.2.2.2. Règles Smeci

Les règles Smeci comportent classiquement deux parties : condition et action. La partie condition comporte une partie déclaration où les variables sont déclarées en fonction de leur catégorie, puis un ensemble de *prédicats* permettant de restreindre les objets filtrés. Les prédicats peuvent être tout prédicat Le_Lisp portant sur les objets filtrés par la règle. De plus, les quantificateurs universels et existentiels peuvent être utilisés à tout niveau dans un prédicat.
La partie action permet de modifier l'état courant en : créant de nouveaux objets, modifiant des valeurs de champs d'objets, modifiant la catégorie d'un objet, stoppant le raisonnement, ou toute action externe via une fonction Lisp.
De plus les règles comportent un certain nombre d'attributs indicateurs pour le contrôle. Ces indicateurs permettent de définir le mode d'instanciation d'une règle (en mode parallèle, séquentiel, une seule fois ...).

II.2.2.2.3. Tâches Smeci

Le règles Smeci sont organisées autour de la notion de *tâche*, objet à part entière. L'enchaînement des tâches peut être spécifié explicitement grâce à la notion d'*arbre de tâches* ou dynamiquement en manipulant par règles un *agenda des tâches*. A chaque tâche est associée une *base de règles*, contenant elle-même une liste de règles.

Les bases de règles servent à décrire la manière de réaliser une tâche. C'est en fait un ensemble de règles assorti d'indicateurs de contrôle, qui permettent de spécifier la manière dont les règles vont être instanciées. Ces indicateurs permettent de déterminer le critère de choix lors des conflits et de spécifier la modalité de création des *états*.

II.2.2.2.4. Critique de Smeci

Smeci offre un environnement de développement hybride dans lequel les objets du domaine sont représentés de manière très complète, grâce à la notion de prototype. Un mécanisme d'instanciation et d'héritage permet de structurer ces objets d'une manière rationnelle. Cependant, même si ce système a été utilisé dans un certain nombre d'applications [Dieng 89], il offre une représentation très particulière des objets : la notion de classe n'y est pas très claire (les classes ne sont pas de véritables objets), et la dualité classe/prototype, soulignée dans [Lieberman 86a] comme fondamentale est rendue plus confuse.

II.2.2.3. Conclusion

Beaucoup d'autres systèmes proposent une représentation hybride frames/règles : Kee, Harlequin, Knowledge Craft [Carnegie 85], Kool, pour ne citer que les plus connus. Mais à notre connaissance aucun d'entre eux ne part véritablement d'une approche objet, dans le sens strict que nous avons défini. Plutôt, les frames n'y sont vus que comme des extensions plus ou moins sophistiquées de la notion de *fait*. La philosophie qui en découle est donc naturellement *orientée attribut-valeur*: les frames ne servent qu'à représenter des structures ad hoc de représentations pratiques pour écrire des règles de production.

II.2.3. Systèmes à base d'objets

Peu de systèmes hybrides partent d'une véritable représentation objet. Nous décrivons ici des candidats basés sur une représentation par objets. Les points sur lesquels nous effectuons notre étude sont les suivants :

- Nature de la représentation des objets,
- Typage des variables d'objets,
- Typage des variables d'attributs,
- Mélange possible ordre un/ordre zéro
- Intégration objets/règles,
- Existence d'exemples d'utilisations,
- Possibilités de définition du contrôle.

II.2.3.1. Humble

II.2.3.3.1. Présentation

Le langage Humble [Piersol 86] sert de référence incontournable en matière de "moteur essentiel" en Smalltalk, puisque c'est le seul produit distribué officiellement à ce jour par la compagnie ParcPlace qui développe le système Smalltalk. Ce système est néanmoins difficilement utilisable. En effet, Humble réunit à lui seul tous les inconvénients d'un moteur d'inférence :

- un langage d'expression de type O+,
- les objets Humble (appelés *entités*) ne sont pas des objets Smalltalk, mais des constructions particulières, avec un mécanisme rudimentaire de hiérarchie de partie,
- les valeurs des attributs ne peuvent être qu'atomiques (pas de pointeurs entre entités Humble),
- les règles ne sont pas déclenchées par un véritable mécanisme de saturation, mais par un chaînage défini explicitement par l'utilisateur. Ainsi les règles Humble ne sont pas du tout indépendantes les unes des autres.

II.2.3.3.2. Critique

Humble n'est pas à proprement parler un système hybride, puisqu'il n'intègre aucune des caractéristiques des langages à objets. Les entités Humble n'ont en effet qu'une lointaine parenté avec des objets (pas de lien d'instanciation, pas de notion de méthode, pas de dépendance fonctionnelle entre objets) et proposent un lien de hiérarchie de partie qui est difficile à utiliser. De plus, les règles Humble ont une mécanique très particulière qui ne permet pas de les qualifier réellement de règles d'ordre un. Enfin le mécanisme d'appel explicite des règles ôte définitivement à ce système tout intérêt dans le cadre de notre étude.

II.2.3.2. Eloïse

II.2.3.2.1. Présentation

Eloïse [Dixneuf & al. 87] est un système d'inférence en marche avant, développé aux laboratoires de la CGE et intégré au langage objets LORE [Caseau 87, Benoit & al. 86].
Le langage LORE est un langage à objets particulier, fondé sur un modèle ensembliste. Les objets LORE possédant des caractéristiques communes sont regroupés en ensembles, ordonnés suivant une relation d'inclusion. Le langage LORE supporte de nombreuses extensions comme Marie [Caseau 87], système de programmation logique.

Système intégré, le langage de règles Eloïse est écrit en LORE, et permet de manipuler tous les objets LORE. C'est un moteur d'ordre un en chaînage avant, avec une structure de contrôle programmable elle aussi en termes d'objets LORE.
Nous nous intéresserons ici uniquement à la structure des règles. Les règles Eloïse sont bien sûr constituées de prémisses et de conclusion, qui portent sur les objets LORE.

II.2.3.2.1.1. Objets manipulés

Les objets manipulés par les règles Eloïse sont tous les objets LORE.

II.2.3.2.1.2. Les règles Eloïse

Les prémisses Eloïse sont soit des prémisses de typage, par lesquelles on indique le type (c'est à dire, en LORE, l'ensemble) de la variable; soit des prémisses de conditions, qui expriment une contrainte sur les variables de la règle.
Quatre types de conditions existent en LORE :

le *when* **:** exprime une contrainte sur la valeur d'une relation de l'objet LORE, sous forme d'un quadruplet objet, relation, comparateur, valeur.
le *for* **:** permet d'unifier des variables de la règle au résultat de l'évaluation d'un message LORE envoyé à un objet, éventuellement référencé par une autre variable.
Par exemple :

```
(for (?a as JO)
        (?b as [?c best-friend])
```

Ceci constitue, comme on le verra en NéOpus, un premier pas vers l'utilisation des dépendances fonctionnelles des objets.
le *for-each* : est une version itérative du *for*, dans laquelle les variables sont liées aux valeurs successives rendues par l'évaluation d'un message LORE, qui doit donc obligatoirement renvoyer une liste d'objets.
le *check :* est un message LORE *quelconque*, mais dans lequel toutes les variables ont été liées précédemment.

Les actions sont de trois types :

le *deduce* : écriture d'une valeur d'une relation discrète (dans le vocabulaire LORE) d'un objet LORE.
le *create* : création d'un symbole ou d'une nouvelle instance LORE
le *do* : évaluation d'un message LORE

II.2.3.2.2. Critique

Le langage de règles Eloïse permet d'écrire des règles d'ordre un, en privilégiant naturellement la notion de *relation* et de valeur de relation qui est le pendant de la notion d'attribut (ou de variable d'instance) pour les objets, dans le système ensembliste LORE. Nous avons donc ici un équivalent direct du langage Essaim pour le monde Smalltalk, mais comportant en plus la notion d'ensemble telle qu'elle existe dans LORE.

II.2.3.3. Essaim

II.2.3.1.1. Présentation

Essaim [Alizon&Huet 88] est un environnement de programmation de systèmes experts conçu au CNET (Lannion). Le moteur d'inférence d'ordre un fonctionne en marche avant. Le contrôle des inférences repose sur la notion de *modules* et de *problèmes*, gérées par un *agenda*, objet Smalltalk manipulable par l'utilisateur. Signalons ici le système de P. Laublet, ForreEnMat [Laublet 90, 91] de démonstration automatique en mathématiques, construit à partir d'Essaim.

II.2.3.1.1.1. Objets manipulés

Les objets Essaim sont des instances de classes Essaim. Une classe Essaim est une classe Smalltalk héritant d'une classe racine `Fait`[8]. Une classe Essaim définit des *attributs* qui sont des variables d'instances ayant certaines caractéristiques supplémentaires (par rapport aux simples variables d'instances Smalltalk), comme un type ou une valeur par défaut. Cependant les attributs ne sont pas pour autant de véritables objets comme dans le langage PtitLoo [Ferber 89]. Les caractéristiques associées à ces attributs ont un rôle purement utilitaire. Surtout, une classe Essaim contient la liste de ses instances susceptibles d'être filtrées par les règles.

II.2.3.1.1.2. Les règles Essaim

Les règles Essaim sont constituées d'une partie *contexte* (vocable préféré à *partie condition*) et d'une partie *conclusion*. Les parties contexte sont constituées de *motifs* définis comme des "*représentations abstraites de faits*". Ces motifs sont unifiés aux objets Essaim, et peuvent comporter des contraintes sur les valeurs d'attributs de ces objets.
Les contraintes sont exprimées par des "*blocs de paires*". Un bloc de paires est une suite éventuellement vide de paires *attribut-filtre,* permettant de tester les valeurs des attributs. Le filtre est soit un test d'égalité, soit une contrainte, soit une disjonction ou une conjonction de valeurs.

[8] Essaim entretient lui-aussi dans son vocabulaire la confusion fait/objet.

Comme en OPS5, il existe deux types de variables : les variables dénotant les objets eux-mêmes, utilisées principalement par les parties action (pour signaler des modifications), et les variables dénotant les valeurs d'attributs, utilisées dans les filtres (Cf. la règle ci-dessous).
Les actions Essaim sont les pendants des blocs de filtres appelés *blocs d'effet*. Ce sont des actions qui permettent de modifier la "base de faits", l'agenda courant et plus généralement de provoquer tout effet de bord sur l'environnement. Certaines actions spécifiques à Essaim (appelées *directives*) permettent d'effectuer les opérations les plus courantes avec une syntaxe simplifiée, comme la modification d'attributs d'un objet. Par ailleurs, toute expression Smalltalk peut être utilisée dans une partie action de règle.

Voici par exemple une règle typique Essaim, tirée de l'exemple du singe et des bananes :

```
SI:
Motif Singe: ?singe
    position = ?p
    hauteur = ?hs
Motif Caisse
    position = ?p
    hauteur = ?hc
ALORS
?singe modifier :
    hauteur <- (?hs + ?hc)
```

Une règle Essaim

II.2.3.1.2. Critique

Le système Essaim est une adaptation directe du système OPS5 en Smalltalk, sans la compilation Rete. Il repose en effet entièrement sur la notion d'attribut-valeur. Le pouvoir d'expression est donc similaire à celui d'OPS5, augmenté de fonctionnalités permises par Smalltalk. En particulier les ressemblances avec OPS5 sont :

- la notion de filtre attribut/valeur,
- le double usage de variables (variables référençant les objets / variables de valeurs d'attributs).

Les possibilités supplémentaires de Essaim par rapport à OPS5 viennent de :

- Typage des attributs des objets.

On peut spécifier en Essaim le type des attributs. Ce type, facultatif, n'est utilisé que pour des vérifications, et non pas pour des optimisations. Aucun "calcul" sur ces types n'est effectué.

- La prise en compte de l'héritage.

Essaim permet de déclarer pour chaque variable de règle si le typage de la variable doit prendre en compte l'héritage de classe ou non (en mettant entre parenthèse le mot-clé *héritage*). Nous proposerons aussi un tel mécanisme mais moins complet en introduisant le typage naturel (Cf. IV.1.6.3.).

- La possibilité d'utiliser des messages Smalltalk quelconques.

Essaim comporte cependant quelques limitations sévères comme l'absence de compilation des règles (les règles sont interprétées au sens le plus strict : aucune optimisation n'est effectuée pour limiter les dégâts de l'explosion combinatoire). D'autre part les objets Essaim filtrés par les règles ne peuvent être tout objet Smalltalk, mais doivent hériter d'une classe racine, comme nous l'avons dit plus haut. Enfin, l'environnement de travail ne permet pas les fonctionnalités traditionnelles de l'environnement Smalltalk, comme le copier/coller : l'écriture/modification d'une règle ne peut se faire qu'à travers une série de browsers spécialisés.

II.2.3.4. Oks

Oks est un système d'inférence conçu par Robert Voyer [Voyer 89a, 89c, 89d, 89e], qui propose un nouveau mécanisme de compilation des règles en marche avant, appelé "compilation réflexe" plus efficace que les modes de compilation traditionnels (par mémorisation aussi bien que par propagation de contraintes).

II.2.3.4.1. Présentation

Les règles Oks sont constituées d'une *partie déclaration*, d'une *partie condition*, elle-même constituée de *motifs*, et d'une *partie action*.
Les objets filtrés par les règles Oks sont des objets Smalltalk, avec une contrainte sur la racine d'héritage : tous les objets doivent être instance d'une sous-classe de la classe racine Oks.
Ici encore deux types de variables sont utilisés dans les règles : les variables référençant les objets eux-mêmes, qui sont déclarées dans une partie déclaration, et les variables servant à référencer les valeurs d'attributs, ou les attributs eux-mêmes (permettant ainsi des inférences dites "à l'ordre deux").
Les motifs sont des triplets Snarkiens <objet attribut valeur>, dans lesquels objets, attributs et valeurs peuvent être variables. La partie action est constituée de tout

expression Smalltalk, pouvant éventuellement modifier les objets filtrés dans la règle, et utiliser les variables d'attributs.
Voici par exemple la règle du "singe et des bananes" dans le système Oks :

```
exemple
| Singe ?s. Caisse ?c |
   <?singe position  ?p>
   <?singe heuteur  ?hs>
   <?c position ?p>
   <?c hauteur ?hc>
actions
   ?s hauteur: (?hs + ?hc)
```

II.2.3.4.2. Critique d'Oks

Le système Oks existe sous forme prototypique et ne constitue donc pas un système véritablement opérationnel. Nous pouvons cependant souligner ici les points essentiels de ce système en tant que système hybride.

II.2.3.4.2.1. Avantages

Prise en compte efficace des modifications des objets

Les modifications des objets par effets de bord (partie action des règles) sont prises en compte automatiquement par le système (via les démons), sans que le programmeur ait à signaler explicitement les objets modifiés. Ceci marche bien entendu uniquement pour les modifications *directes* (Cf. à ce sujet la notion de d-modification au Chapitre V.3.2.1).

Grande efficacité des inférences

Les inférences sont (réputées) efficaces, même pour un nombre élevé d'objets et de règles, contrairement aux systèmes basés sur une architecture classique (Treat, Rete ou Snark). Ceci est dû au mode de compilation très astucieux des règles sous forme de démons [Voyer 89a, Bouaud 89a, 89b]. En particulier, un des points originaux du système est qu'il reste insensible au nombre d'objets qui ne filtrent aucune règle. Ainsi, les performances du système sont indépendantes du nombre d'objets "parasites", contrairement aux systèmes traditionnels (Nous aurons en revanche ce problème en NéOpus).

II.2.3.4.2.2. Contraintes

Malheureusement, la mécanique sophistiquée d'Oks comporte aussi un certain nombre de contraintes.

A. La classe des objets filtrés est contrainte (classe racine `Oks`)

Cette contrainte est difficilement supprimable dans la mesure où toute l'architecture de la compilation Oks repose sur la mise en place de démons, associés à chaque objet pour chaque variable d'instance. Ces démons sont représentés par autant de variables d'instances supplémentaires, qui sont générées automatiquement et "cachées" à l'utilisateur. Les classes système étant non recompilables, elles échappent à la compilation réflexe, du moins telle qu'elle est proposée ici.

B. Attitude stricte envers les méthodes d'accès aux variables d'instance

Le bon fonctionnement du système repose sur le principe que tous les accès en lecture ou écriture aux variables d'instance d'un objet se font par envoi de message, et non par accès direct, comme c'est souvent le cas en Smalltalk. En effet, la gestion des démons se traduit par des méthodes d'accès (en écriture notamment) générées automatiquement par le système, qui se chargent de propager les modifications d'un objet aux objets concernés. Un accès direct à une variable d'instance court-circuite donc ce mécanisme et peut provoquer des incohérences.
Il s'agit ici d'une attitude par ailleurs tout à fait recommandable en théorie, comme le démontre [Wirfs-Brock&Wilkerson 89], mais qui suppose une pratique de programmation "sans variable" non supportée par l'environnement Smalltalk en l'état actuel.

C. Mauvaise prise en compte de l'héritage dans les variables de règles

L'architecture de compilation Oks conduit à un comportement étrange vis-à-vis de la prise en compte de l'héritage dans les règles. En effet, Oks distingue les *démons d'instance*, stockés dans une variable d'instance de l'objet, et donc spécifique à chaque objet, des *démons de classe*, qui, eux, sont spécifiques à une classe, et représentés sous forme de variable de classe. Mais ce choix de représentation rend incohérent l'usage de l'héritage, puisque les variables de classe sont partagées par toutes les sous-classes, "dans les deux sens".
Si l'on suppose par exemple la classe `Processus` définie par :

```
Object subclass: #Processus
  instanceVariableNames: 'etat pere'
```

Oks générera quatre listes de démons : `demonsInstPere demonsInstEtat`, comme des variables d'instance de `Processus`, et `DemonsClassPere` et `DemonClassEtat`, représentés sous forme de variables de classe de `Processus`.
La classe `Processus` sera donc en réalité définie comme :

```
Object subclass: #Processus
 instanceVariableNames: 'etat pere demonsInstPere demonsInstEtat'
 classVariableNames:    'DemonsClassPere DemonsClassEtat'
```

Ces deux derniers étant des variables de classe, ils sont donc accessibles par les sous-classes éventuelles de `Processus`. Soit, par exemple, une telle sous-classe `Processus2`. Le problème est alors que tous les démons de classes installés par des variables déclarées par `Processus2` vont mettre à jour les variables de classe définies au niveau de `Processus`, et donc rendre incohérentes les déclarations de variables de règles.
Par exemple, la règle suivante, destinée à suspendre toutes les instances de `Processus2` va filtrer aussi bien les instances de `Processus2` que les instances de `Processus` :

```
tuerProcessus2
| Processus2 ?p |
   <?p ?x ?y>>
actions
   ?p suspend
```

D. Contrôle opportuniste

La politique de déclenchement opportuniste de Oks rend très difficile l'écriture de règles dont les parties *action* sont complexes. En effet, les règles se déclenchant de manière réflexe à toute modification des objets, on ne peut garantir la séquentialité des opérations spécifiées dans la partie action d'une règle, ce qui peut conduire à des résultats incohérents (Cf. l'exemple du renversement d'une chaîne de caractères dans [Voyer 89a]). Un mécanisme de gel resterait à mettre en place pour permettre de contrôler dynamiquement le déclenchement des réflexes.

Oks est cependant très efficace quand on sait spécifier les attributs "sensibles" des objets. Oks doit donc être vu avant tout comme une implémentation provisoire d'un mécanisme de compilation efficace, plus que comme un véritable langage de représentation, comme le dit son auteur qui le qualifie de *langage de représentation de langage de représentation de connaissances* [Voyer 89c], bien que la réalisation d'un tel langage de représentation à l'aide d'Oks soit encore à faire.

Oks ne contribue donc pas à intégrer les règles et les objets, mais fournit plutôt un mécanisme original et très efficace de "compilation traditionnelle".

II.3. Synthèse

Notre rapide tour d'horizon nous permet de désigner Opus comme unique dans le sens où il "part" d'une représentation du monde sous forme d'objets au sens strict, sans limitation. D'autre part, les systèmes rencontrés, s'ils proposent des outils intéressants, ne proposent aucun support méthodologique. Les objets ne sont vus

que comme des entités pratiques pour écrire des règles. Nous allons renverser ce point de vue en considérant les objets a priori comme représentation du monde (avec bien sûr ses limitations), *sur lesquels* nous allons prononcer des discours, via une mécanique d'inférence adaptée.

III.Opus in vivo

Avant-Propos

Dans ce chapitre, nous présentons notre implémentation du système Opus, réalisée à partir de notre interprétation de la description de [Atkinson&Laursen 87].
Après avoir rapidement résumé le style de programmation OPS5, nous montrons comment le plongement de règles OPS5 dans l'univers Smalltalk conduit naturellement aux principes de base du système Opus.

Nous décrivons ensuite plus en détail l'implémentation de chacun des éléments du système et son utilisation. Enfin nous soulignons les points litigieux les plus importants découlant de cette implémentation.

Cette partie a pour but de mettre en place les différents objets essentiels de l'environnement Opus, à savoir les règles, les bases de règles, les conflict sets, les règles déclenchables, les réseaux et les nœuds Rete. Nous décrivons succinctement leur implémentation, en donnant pour chacun d'entre eux, leur définition Smalltalk et les méthodes essentielles permettant d'y accéder, de les modifier, d'en *parler*. Cette partie est donc assez technique, et nécessite une connaissance sommaire du langage Smalltalk.

Nous signalons les points sur lesquels nous nous éloignons de la description des auteurs, ou sur lesquels ceux-ci ne donnent aucun détail d'implémentation chaque fois qu'il est nécessaire. Les méthodes les plus importantes sont écrites en mettant entre parenthèses les passages non essentiels (comme la gestion de l'interface, ou certaines optimisations). Pour plus de détails, se reporter au code commenté, en annexe.

III.1. Rappel sur le système OPS5

III.1.1. Des faits aux objets

Les architectures d'inférence traditionnelles, comme Snark, proposent une dichotomie entre la *base de faits*, qui contient les structures de données manipulées par le système et la *base de règles*, qui contient des règles de production permettant de manipuler les faits, de les modifier, de les supprimer et d'en créer de nouveaux.
Dans Snark, les faits sont des structures de données simples, matérialisées par des triplets (objet, attribut, valeur). Ces triplets ont un caractère intrinsèquement booléen : leur présence affirme leur vérité.
Par exemple, l'énoncé "*la couleur de la voiture est bleue*" se représente en Snark par le triplet :

```
(voiture couleur bleue)
```

et s'interprète naturellement de manière booléenne (avec le sous-entendu "*il est vrai que ...* ").

Les parties actions des règles correspondent du coup à des assertions logiques : "asserter" un fait est équivalent à l'ajouter dans la base de faits. Il y a donc dans ce cas une parfaite adéquation entre le savoir et l'action.

Le système OPS5 [Forgy 81] propose lui aussi une architecture classique d'inférence, basée sur la même dichotomie, mais où les faits subissent une extension notable. On parle déjà d'*instance*, en OPS5, et non de *fait*, dans la mesure où les structures de données manipulées ne sont plus de simples relations binaires mais des relations n-aires typées. Ces instances perdent du coup leur caractère booléen. Une *base de faits OPS5* contient donc des *objets OPS5*.

En OPS5, les types des objets (ou classes), doivent être déclarés avant l'écriture des règles (contrainte que l'on retrouvera en Opus).
Les classes sont déclarées de la façon suivante, à l'aide de la commande *literalize* :

```
(literalize nomClasse attribut1 attribut2 .. attributn)
```

Par exemple, pour créer la classe des personnes ayant les attributs *nom*, *pere*, *mere*, *sexe* :

```
(literalize personne nom pere mere sexe)
```

Tout objet de la base de faits est une instance d'une de ces classes, ayant des valeurs particulières pour les attributs. Les valeurs des attributs peuvent être soit atomiques (nombre, chaîne de caractère) soit vectorielles (une suite de valeurs atomiques). Elles ne peuvent pas être des pré-objets à proprement parler.

Une fois les classes créées par la commande *literalize*, les instances[9] sont créées en utilisant la commande *make*, de la façon suivante, en utilisant le caractère "^" comme index; les attributs non mentionnés (ici l'attribut `sexe`) étant initialisés par défaut à nil :

```
(make personne ^nom Isaac    ^pere Abraham    ^mere Sara)
(make personne ^nom Esau     ^pere Isaac      ^mere Rebecca)
(make personne ^nom Jacob    ^pere Isaac ^mere Rebecca)
```

III.1.2. Un langage pré-objet

La typologie des objets OPS5 n'a qu'une lointaine parenté avec celle des langages à objets, en particulier avec Smalltalk. En effet, seul le processus *d'instanciation* est présent, avec beaucoup de restrictions : type et nombre des attributs, pas de lien entre les classes (pas d'héritage), pas de notion de comportement (méthodes) associé aux objets. Enfin, les attributs étant des atomes (c'est-à-dire des chaînes de caractères, des nombres, ou des vecteurs de ces derniers), ils ne pointent pas vers les objets eux-mêmes. Ainsi, dans notre exemple, le père de Jacob (la valeur de l'attribut `pere` de l'instance de la classe `Personne`, de nom 'Jacob') est rerpésenté par la chaîne de caractère 'Isaac', et non Isaac lui-même (l'instance de `Personne` de nom 'Isaac').
Nous verrons que cette approche conduit à une indirection systématique, qui empêchera notamment la propagation des contraintes dans les prémisses des règles. Cette indirection disparaîtra en Opus, où les dépendances fonctionnelles des objets seront naturellement prises en compte.

Les objets OPS5 sont plus des *records* à la Pascal que de véritables classes au sens des langages à objets. Nous pouvons parler de *pré-objets* par opposition aux véritables objets Smalltalk.

Nous avons donc la progression suivante de la notion de fait, des symboles aux objets :

Logique des propositions	symbole
Ordre zéro-plus	attribut valué
Snark	triplet
OPS5	pré-objet
Opus	objet

La notion de fait dans les systèmes de règles de production

III.1.4. Des règles en marche avant

[9] On parlera donc d'*instances* , ou d'*objets* en OPS5 et non plus de fait. On différenciera, par la suite les *objets OPS5* des *objets Smalltalk*.

Les règles sont de type classique condition/action, où les conditions sont des *filtres* établissant des contraintes sur les pré-objets présents dans la base de faits, et la partie action une séquence d'opérations de mise à jour de la base de faits ou de contrôle sur le moteur d'inférence.
Les filtres sont exprimés par des contraintes simples (comparaison, égalité) entre les valeurs des attributs des pré-objets. Ces valeurs peuvent être dénotées par des variables indiquées entre crochets.

Les opérations de mise à jour de la base de faits sont essentiellement les trois opérations `make` (création d'un nouvel objet), `remove` (destruction d'un objet) et `modify` (raccourci de la séquence `remove` + `make`).

La syntaxe des règles est la suivante :

```
(p nomDeRegle
        (unePremisse)
        (unePremisse) ....
->(une action)
        (une action) ... )
```

Deux types d'utilisation des variables existent en OPS5 :

- les variables référençant la valeur d'un attribut pour un objet (notées entre crochets),
- les variables référençant les pré-objets eux-mêmes (variables apparaissant en exergue de la prémisse).

Les rôles joués par ces variables sont distincts : les variables de valeur d'attributs servent à établir les contraintes entre les pré-objets (ainsi tester que deux attributs ont même valeur se traduit implicitement en utilisant des variables de même nom), tandis que les variables référençant les pré-objets ne sont qu'une commodité syntaxique utile permettant de référencer les pré-objets en partie conclusion, pour les modifier par exemple.
Nous verrons ultérieurement que les rôles des variables sont intervertis en Opus.

III.1.5. Exemple : le calcul d'ancêtres

Voici l'exemple d'une base de règles qui calcule les ancêtres d'une personne (tiré de [Brownston&al. 85]) de façon récursive en parcourant l'arbre généalogique par les parents. On crée un certain nombre d'instances de la classe `Personne`, avec différents noms. Le parcours de l'arbre généalogique s'effectue en manipulant des instances de la classe `Request`, définie comme suit :

```
(literalize Request target)
```

où `target` est le nom de la personne dont on recherche les ancêtres.

La règle suivante décrit alors ce parcours, en créant de nouvelles instances de `Request` pour la mère et le père de l'instance de `Personne` filtrée :

```
(p findAncestors

  (Request ^target (<aName> <> nil) )
  (Person ^name <aName> ^mother <mother-name> ^father <father-name> )
-->

  (remove <request1>)
  (write <mother-name> and <father-name> are ancestors via <aName> )
  (make Request ^target <mother-name> )
  (make Request ^target <father-name> ) )
```

III.1.6. Un moteur d'inférence figé

Le moteur d'inférence OPS est basé sur un cycle standard d'évaluation, qui découle directement de l'algorithme Rete :

Détermination par filtrage de toutes les règles déclenchables
Tant que cet ensemble est non vide :
- Choix d'une règle selon une stratégie fixée
- Exécution du membre droit de la règle

Il faut remarquer qu'il n'y pas dans cet algorithme de re-calcul explicite du conflict set : cette modification repose uniquement sur l'*effet des modifications de la base de faits* opérées en partie droite des règles (essentiellement à l'aide des actions `make`, `modify` et `remove`). On parle alors de déclenchement dirigé par les données (data-driven) ou de programmation p*ar effet de bord*.

Les deux stratégies de choix proposées par OPS5 : LEX et MEA, reposent sur la notion de *fraîcheur* des pré-objets : les pré-objets sont numérotés à leur création, renumérotés à leur modification, et le choix de la règle à déclencher tiendra compte de ces numéros.

LEX (pour lexicographique) : les règles sont ordonnées lexicographiquement selon les numéros d'ordre des faits les constituant, en privilégiant les règles ayant le plus de conditions[10].
MEA (pour Mean-ends Analysis) fait la même chose mais en rajoutant une étape qui consiste à considérer le numéro de fait filtré par la première condition de la règle comme critère de tri principal.

[10] Pour plus de détails, voir les commentaires de [Farreny & Ghallab 87].

III.1.7. Rappel sur l'algorithme Rete

Le cycle d'inférence utilise une adaptation du réseau Rete [Forgy 82] de façon à minimiser les tests sur les jeux d'instanciation. Les combinaisons ne sont ainsi testées qu'une seule fois au cours d'une exécution, les jeux d'instanciation étant mémorisés au fur et à mesure qu'ils sont construits dans les nœuds du réseau.
Sans donner une description complète et détaillée de cet algorithme (la description originale par l'auteur [Forgy 82] est très détaillée), nous pouvons rapidement en rappeler le mécanisme.

Le réseau Rete est en fait constitué de deux réseaux : le réseau de discrimination et le réseau de jointure. Le réseau de discrimination est constitué de nœuds à une seule entrée qui font des tests sur une seule instance. Le deuxième réseau est constitué de nœuds à deux entrées qui font des tests sur deux éléments, réalisant une jointure entre les n-1 prémisses précédentes et la prémisse courante.
Chaque nœud de jointure du réseau Rete est constitué d'une mémoire de gauche et d'une mémoire de droite. La mémoire de gauche contient tous les faits filtrés par le motif de la prémisse, et la mémoire de droite contient tous les jeux d'instanciation ayant passé avec succès les jointures précédentes. Le rôle du nœud est donc de fabriquer de nouveaux jeux d'instanciation en fonction de sa mémoire de gauche (les nouveaux faits) et droite (les jeux d'instanciation partiels déjà créés et validés), et de les propager au nœud suivant (dans sa mémoire de droite).

Un bon exemple valant mieux qu'un long discours, nous donnons ici le réseau Rete pour l'exemple suivant, exprimé en OPS5 :

Soient les classes suivantes, représentant les incontournables objets `Singe`, `ObjetPhysique` et `But` :

```
(literalize But status nomActeur nomObjet typeAction)
(literalize Singe nom position nomObjet)
(literalize ObjetPhysique nom position nomObjetSur)
```

et une règle commençant par :

```
(p testSinge
  (But ^status active ^nomActeur <a> ^nomObjet <o>)
  (Singe ^nom <a> ^position <p>)
  (ObjetPhysique ^nom <o> ^position <p>)
-> ...)
```

Cette règle de trois prémisses va produire le réseau Rete suivant (Figure 1) :

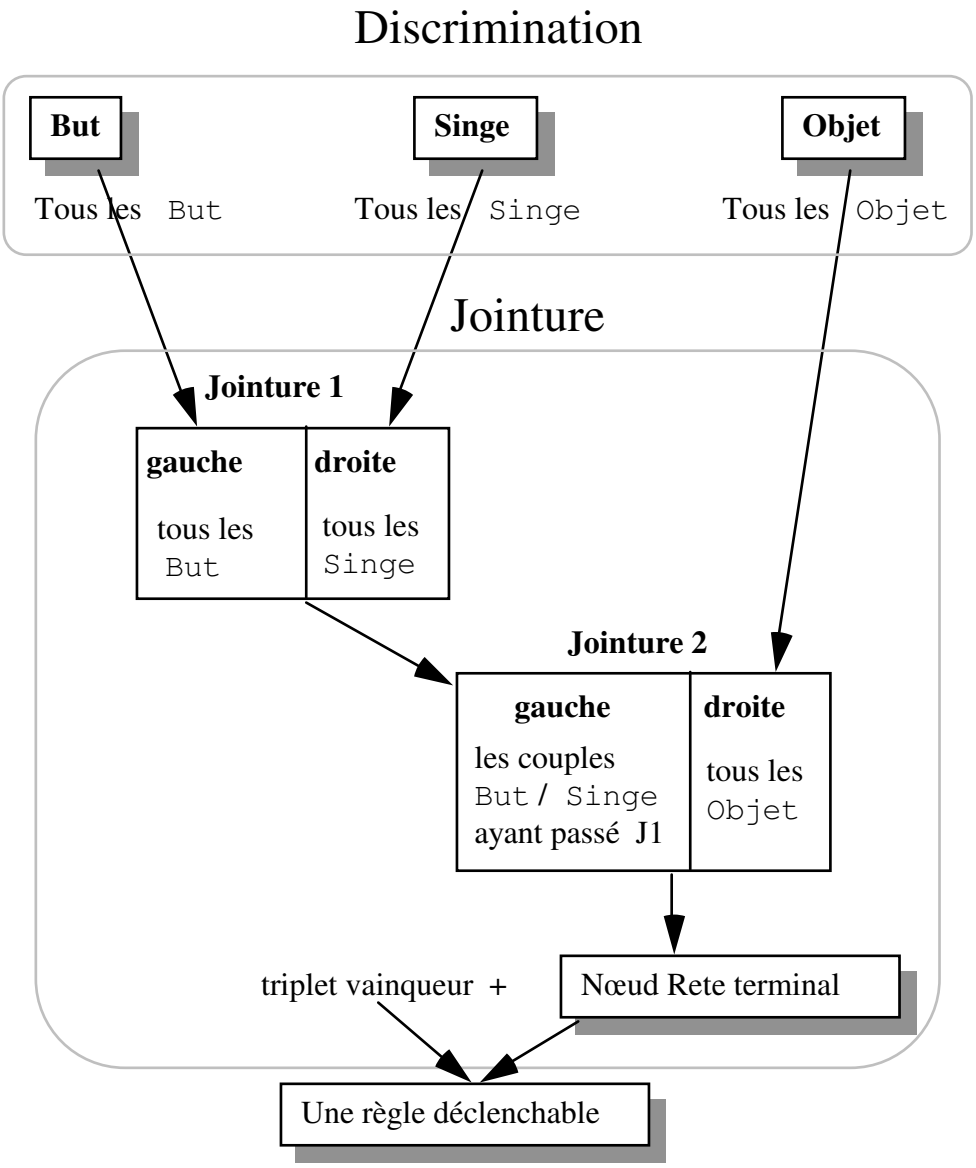


Figure 1. Le réseau Rete pour la règle `testSinge`.

Notons plusieurs points importants dans cet algorithme :

- L'efficacité de l'algorithme (par rapport à un algorithme "naïf", qui parcourrait toutes les prémisses à chaque exécution) vient de la mémorisation des jeux d'instanciation partiels au niveau de chaque nœud, qui ne sont donc calculés qu'une seule fois.
- L'inefficacité vient du manque de propagation des contraintes entre variables. Ici par exemple, étant donné un objet `But`, il n'y a qu'un seul objet `Singe` dont le nom est égal à celui spécifié dans l'objet `But`.
Cependant, le nœud `J1` va essayer toutes les combinaisons entre cet objet `But` et tous les objets `Singe`, afin de trouver celui qui vérifie la contrainte (nom du singe = nom de l'acteur du but). On dit alors qu'il n'y a pas de propagation des

contraintes. Le système Snark [Vialatte 85], saura à l'inverse propager les contraintes entre prémisses, mais ne fera pas de mémorisation.
- L'algorithme Rete est un algorithme efficace dans le cas d'univers peu évolutifs. Cela est dû au coût élevé des opérations de modifications de faits. En effet, modifier un fait consiste à le retirer puis à le rajouter. Or, l'opération de retrait est aussi coûteuse que celle d'ajout.
- Le réseau de propagation Rete présente en outre une particularité intéressante : celle de pouvoir factoriser les nœuds correspondant à des prémisses identiques de différentes règles. Néanmoins cette factorisation reste limitée car elle est dépendante de l'ordre des conditions.

De nombreux travaux ont été effectués pour optimiser certaines phases de l'algorithme [Cordier&al. 86]. [Shor & al. 86] présente quelques optimisations, en particulier pour le modified. Treat [Miranker 90, Savéant 90] remédie à l'inefficacité induite par les modifications d'objets, en supprimant les redondances dans les mémorisations partielles. Certains systèmes [Parchemal 88] tentent d'adapter dynamiquement le mode de compilation en fonction des données. L'utilisation d'un *arbre d'unification* [Ghallab 80, 88] réduit encore les coûts de l'opération de pattern-matching. Une analyse fine des lacunes de l'algorithme Rete, notamment de l'absence de propagation des contraintes entre variables, est faite dans [Voyer 89a], et conduit à un nouveau mode de compilation dit "réflexe" et au système Oks [Voyer 89a, 89c, 89d, 89e].

En ce qui concerne Opus, nous verrons comment l'algorithme Rete a été transposé dans l'univers Smalltalk, ce qui a conduit à un certain nombre de modifications substantielles. Le problème de l'absence de propagation des contraintes entre variables y prendra un éclairage nouveau, résolu en partie par la dépendance fonctionnelle des objets entre eux : l'objet `But` connaissant (voir lexique) son objet acteur (le singe), l'accès direct à l'objet `Singe` est possible de manière fonctionnelle sans avoir à faire de filtrage.

III.2. D' OPS5 à Opus

Opus est un outil de programmation par règles, intégré à l'environnement Smalltalk-80, et a été l'objet d'une seule publication par ses auteurs [Atkinson&Laursen 87]. Dans cet article les auteurs présentent successivement les objectifs de leur système, sa description et des éléments d'implémentation, et quelques problèmes liés à leur approche. Nous résumons dans ce chapitre notre lecture de cette description, en expliquant comment le principe d'intégration du langage dans l'environnement Smalltalk permet le passage d'OPS5 à Opus.

III.2.1. Objectifs des auteurs

Le système Opus est né de l'envie des auteurs de rassembler en un seul système deux paradigmes puissants de représentation : celui des objets, tel qu'il existe en Smalltalk, et celui des règles de production [Brownston&al. 85, Hayes-Roth&al. 83, Perrot89b] sur le modèle de OPS5 [Forgy 81].
Plus généralement, les objectifs des auteurs s'inscrivent dans la tradition du développement des langages à objets, à savoir :

- Pouvoir réunir plusieurs paradigmes de programmation en un seul environnement. La volonté de fabriquer des systèmes coopérants (*collaborative systems*) sera, on le verra, fortement facilitée par les propriétés hautement réutilisables des applications Smalltalk.

- Avoir un système facilement modifiable et adaptable. Le besoin de particulariser les systèmes et donc de produire des systèmes pervertissables est un souci omniprésent en programmation par objets.

Deux exemples d'applications sont simplement évoqués, à savoir : utiliser les règles pour faire de la classification d'information et créer des "user-specific communication assistants".
Il s'agit donc plus d'une "expérience paradigmatique" que d'un véritable besoin informatique ou industriel, bien que les auteurs soulignent que, grâce au caractère branchable des composants du système, il serait envisageable, à terme, de substituer progressivement à chacun de ces composants, des composants plus efficaces, et moins souples, en vue d'une utilisation en vraie grandeur.

III.2.2. Guides de conception

La conception du système est guidée par un certain nombre de considérations relativement générales, mais décisives, telles que :

Intégration maximale du système, du langage et de l'environnement.
Liberté maximale d'expression du langage : pouvoir utiliser le langage, non pas comme une porte de sortie (trap-door) comme dans OPS5 ou même Humble, mais donner la possibilité d'écrire toute expression Smalltalk dans les prémisses comme dans la partie action des règles.
Disposer des fonctionnalités de **navigation** (browsing) standard de Smalltalk pour l'écriture des règles.
Avoir un système **modulaire** avec la notion d'éléments *branchables* : la base de faits, l'interprète (avec ou sans debugger), les outils de trace, les éléments d'interface.

III.2.3. Description du système par les auteurs

L'architecture du système comprend quatre objets principaux :

La mémoire de travail
: comprend l'ensemble des *catégories* définies par l'utilisateur. Ces catégories sont des ensembles d'objets quelconques Smalltalk. Les catégories ne correspondent pas aux classes Smalltalk. Les auteurs donnent comme exemple les catégories `ButAtteint` et `ButNonAtteint`, toutes deux composées d'instances de la classe `But`. Nous reviendrons sur cette notion au Chapitre IV.1.6.

Les Bases de règles
: Celles-ci sont des classes dont les méthodes sont les règles Opus. Ces classes bénéficient de toutes les fonctionnalités de Smalltalk de gestion des classes.

Les règles
: Les règles sont des textes constitués d'un identificateur, d'une déclaration de variables, d'une liste de prémisses, et d'une partie action[11]. La syntaxe de ces règles est profondément différente de celle des règles OPS5. Nous montrons au paragraphe suivant comment cette nouvelle syntaxe se déduit de la syntaxe originale.

L'interprète
: Un interprète gère l'exécution de la base de règles. Il est chargé notamment de choisir à chaque cycle une ou plusieurs règles, en invoquant une stratégie de choix particulière.

De plus, conformément à la tradition Smalltalk, le système comprend un interface utilisateur sophistiqué comportant une vue de la mémoire de travail, du conflict set, du réseau Rete, et de l'historique de l'exécution des règles.
Nous décrivons en détail les structures de ces différents objets au chapitre III.3.

III.2.4. Implications sur l'expression des règles

L'introduction dans le monde Smalltalk de règles de type OPS5, conduit de façon assez naturelle à la syntaxe d'Opus proposée par les auteurs.

1. Les faits de OPS5 vont devenir les objets Smalltalk

Plutôt que de reproduire la structure des faits OPS5 (par exemple en créant une classe abstraite `ObjetOPS5`), il est naturel d'étendre ces faits aux objets Smalltalk

[11] Nous donnerons la syntaxe complète des règles Opus par la suite, en y incluant nos ajouts et modifications.

eux-mêmes. Les attributs des faits OPS5 deviennent ainsi les variables d'instances des objets Smalltalk.
Le principe d'intégration postule par ailleurs que les règles pourront utiliser *tout objet Smalltalk*[12] (*utiliser* sera ici synonyme de "filtrer" ou de "parler de"). Nous érigeons cette volonté en principe :

Principe "Tout objet"
Tout objet de l'environnement Smalltalk pourra être utilisé dans une règle

2 Les règles Opus établissent des *contraintes sur des objets* Smalltalk

Les variables n'ont donc plus le même rôle : en OPS5, les filtres établissent des contraintes simples entre les valeurs des attributs des faits filtrés. Or, en Smalltalk, les objets ne sont accessibles que par message, y compris pour accéder aux valeurs des variables d'instances. Les prémisses des règles vont donc être constituées de messages Smalltalk établissant des contraintes entre objets Smalltalk. Pour écrire ces contraintes apparaît tout de suite la nécessité d'utiliser des *variables* pour référencer les objets filtrés (il faut bien envoyer les *messages* à des *objets*).
Ainsi par exemple, la prémisse OPS5 suivante (filtrant toutes les instances de `Personne` dont le `pere` est non nil) :

```
(personne ^pere (<p>  <> nil))
```

va-t-elle se transformer en la transmission booléenne Smalltalk suivante, où `unePersonne` représente une instance de la classe Smalltalk `Personne`, pour laquelle le message d'accès `pere` a été défini (le message `notNil` étant un message prédéfini) :

```
unePersonne pere notNil.
```

Le typage des variables disparaissant dans la transmission, il faut ajouter aux règles une *partie déclaration* dans laquelle les classes des variables seront spécifiées. Ici, cette déclaration prendra la forme suivante (en suivant une syntaxe où le nom de la classe précède le nom des variables de cette classe, et où le tout est entre barres verticales) :

```
| Personne unePersonne|
```

On assiste alors à la disparition des variables qui référencent les valeurs d'attributs : en effet le besoin d'avoir une variable pour référencer la valeur d'un attribut (<p> <> nil en OPS5) disparaît puisque la contrainte s'exprime directement en terme de message Smalltalk. En revanche la variable `unePersonne` référençant l'objet lui-

[12] Smalltalk est un langage hautement redéfinissable. Néanmoins, les classes du noyau (`Object`, `Class`, `Behavior`) ne sont pas recompilables (l'expérience a été tentée). Ainsi la contrainte de pouvoir manipuler tout objet Smalltalk sous-entend-elle "sans recompiler les classes du noyau".

même devient ici fondamentale. C'est donc un renversement du rôle des variables qui s'opère ici : les objets prennent le dessus sur les attributs.

3. Une extension naturelle

A partir du moment où les contraintes entre objets sont exprimées par des transmissions Smalltalk, le rôle des attributs (ou variables d'instances) paraît tout de suite limitatif et très contraignant : pourquoi limiter les transmissions à des messages d'accès ? Les prémisses peuvent donc être définies comme "toute transmission Smalltalk de type[13] booléen.
On définit un deuxième principe, qui, on le verra, sera lourd de conséquences :

Principe "Toute expression"
Une prémisse des règles peut être toute expression booléenne Smalltalk

Cette extension, on le verra, s'avérera être très puissante, puis qu'elle permettra de considérer les objets filtrés par les règles indépendamment de leur implémentation, ce qui est impossible en OPS5, les objets n'y *existant* que par et à travers leur structure.

4. Le rôle des atomes s'estompe

Les liens entre les faits OPS5 reposent essentiellement sur le partage d'attributs atomiques. Ainsi des faits de la classe `Personne` seront liés par le lien de parenté de la façon suivante :

```
(make personne ^nom Isaac  ^pere Abraham )
(make personne ^nom Jacob  ^pere Isaac)
```

Une prémisse testant qu'une personne (Abraham) est le père d'une autre (Isaac) s'écrira donc, en utilisant deux fois la même variable (ici `<p>`) pour indiquer l'égalité des valeurs des noms :

```
  (personne ^pere <p> ... )          la personne B
  (personne ^nom <p> ... )           la personne A
```

Le lien entre Jacob et Isaac est le partage d'une chaîne de caractères (Cf. Figure 2) :

[13] Smalltalk n'étant pas un langage typé, la notion de type est ici prise au sens intuitif du terme : une transmission est de type booléen si on (le programmeur) est sûr que le résultat rendu par la méthode racine de l'arbre syntaxique rend, dans tous les cas où elle est utilisée, un booléen (true ou false en Smalltalk). En particulier, il n'existe aucun moyen automatique de vérifier qu'une transmission Smalltalk quelconque est de type booléen.

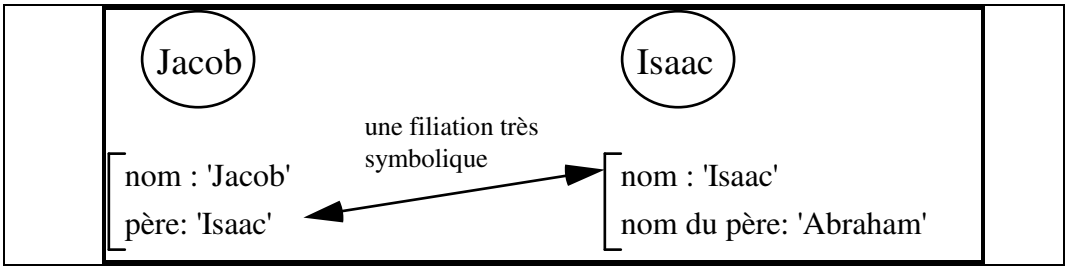


Figure 2. Lien symbolique en OPS5

En Opus, cette indirection est inutile et maladroite, le réseau formé par les objets Smalltalk étant alors inutilisé. Les objets Smalltalk seront avantageusement reliés de manière fonctionnelle directement : le père de Jacob sera effectivement Isaac, et non le nom d'Isaac ! La méthode suivante crée deux instances de la classe `Personne`, en les reliant fonctionnellement :

```
!'Personne class methodsFor: 'exemple'!

exemple
   |x y|
   x <- Personne new nom: #Isaac.
   y <- Personne new nom: #Jacob.
   y pere: x.
```

Le lien entre Jacob et Isaac est alors un lien de pointeurs (Cf. Figure 3).

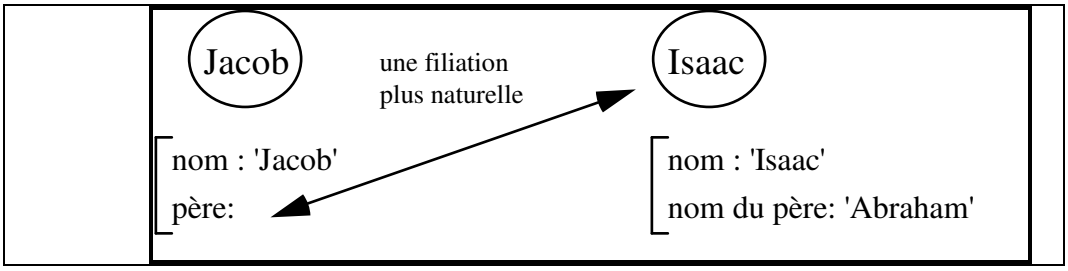


Figure 3. Lien naturel en Opus

Les prémisses précédentes s'écrivent alors sans utiliser de variables intermédiaires et en déclarant la classe des variables :

```
|Personne a b|
a pere == b.
```

Pour clore le débat, comparons l'écriture des prémisses nécessaires à tester que le père et la mère d'une certaine personne ont le même prénom dans les deux systèmes :

A/ En OPS5, on aura à filtrer trois variables, respectivement pour la personne considérée, son père et sa mère (on suppose que la classe `Personne` est enrichie de l'attribut `prenom`) :

```
(Personne ^pere <p> ^mere <m>)            la personne
(Personne ^nom <p> ^prenom <lePrenom>)    son père
(Personne ^nom <m> ^prenom <lePrenom>)    sa mère
```

B/ En Opus, une seule variable est nécessaire, le test étant réalisé par une seule transmission Smalltalk :

```
| Personne p |
p pere prenom = p mere prenom.
```

La règle OPS5 `findAncestors` s'écrit donc en Opus :

```
!PersonneRules methodsFor: 'une regle'!

findAncestors
   | Personne p. RequestOPS r |
   r target = p.
actions
   |r1 r2|
   r1 := Request new target: p pere.
   r2 := Request new target: p mere.
   r1 go. r2 go
```

III.2.5. Différences entre OPS5 et Opus (une première synthèse)

A ce niveau, nous pouvons déjà souligner les différences majeures entre OPUS et OPS5. Notons que si Smalltalk ne permettait que les messages d'accès (en lecture et en écriture) aux attributs, et les quelques messages de comparaison standards (=, <, >, ...), on retrouverait exactement le système OPS5, mais avec une syntaxe un peu différente.
La richesse du système Opus par rapport à OPS5 vient donc, dans un premier temps, de ces deux différences :

- extension des prémisses à tout message Smalltalk,
- utilisation des liens fonctionnels entre objets.

Nous verrons que cet apport changera profondément la rédaction des règles. Nous reviendrons plus loin (Chapitre V.3, le `modified`) sur les conséquences de l'utilisation des dépendances fonctionnelles entre objets dans les règles.

III.3. Opus in vivo

Nous décrivons maintenant notre implémentation effective du système Opus, basée sur notre compréhension de la description des auteurs, et sur notre expérience. Les

acteurs d'Opus sont liés entre eux de façons trop multiples pour pouvoir les étudier séparément et en séquence. Nous donnons en premier lieu une vision d'ensemble de l'anatomie d'Opus, puis reviendrons en détail sur chaque élément dans les parties suivantes.

III.3.1. Structure générale

On peut distinguer deux activités majeures d' Opus : la *compilation* des règles en réseau Rete, et l'*activation* de ce réseau. Les objets importants dans les deux paragraphes suivants sont signalés en gras.

A - Compiler les règles en réseau

Les **règles** Opus sont organisées en **bases de règles**. Les bases de règles sont des classes Smalltalk (plus précisément des classes abstraites), et les règles apparaissent sous forme de méthodes de cette classe.
A chaque base de règles est associée une autre classe Smalltalk, appelée **classe dynamique**, dans laquelle sont compilées les batteries de méthodes Smalltalk correspondant aux règles Opus.
L'idée de la compilation Rete en Opus est de représenter chaque prémisse d'une règle par une méthode particulière, à valeur booléenne. L'opération de vérification de la prémisse se traduit par l'envoi du message correspondant à un objet appelé **token** (jeton). De même la partie conclusion de la règle est représentée par une méthode Smalltalk ad hoc, et son exécution est gérée par l'envoi du message correspondant à un token terminal. Les représentations des prémisses et des conclusions sont organisées, selon la philosophie Rete, en un réseau dont les prémisses sont les nœuds et les conclusions les sorties (ou sommets terminaux).
La classe dynamique est la classe génératrice des tokens (Cf. §B.).
La compilation d'une règle a un double effet :

- la génération et la compilation de méthodes Smalltalk correspondant à chacune des **prémisses** et à la partie conclusion dans la classe dynamique.
- la mise à jour d'un **réseau** Rete, dans lequel pour chaque prémisse et pour la partie conclusion sont créés des **nœuds** Rete, auxquels sont associés les sélecteurs des méthodes créées dans la classe dynamique.

B - Activer ce réseau

Une fois les règles compilées en réseau, l'activation de ce réseau repose sur la notion de token. Un token représente un jeu d'instanciation, ou combinaison d'objets Smalltalk vérifiant les *n* premières prémisses d'une règle (conformément à la stratégie de compilation OPS5). Les tokens sont des instances de la classe dynamique associée à la base de règles activée.
Activer le réseau consiste à fabriquer ces tokens, à les envoyer aux nœuds d'entrée du réseau, et à les propager jusqu'à ce qu'ils atteignent éventuellement un nœud

terminal (conclusion d'une règle) : dans ce cas la règle correspondante est dite **déclenchable**. Elle est alors ajoutée à un **conflict set**, ainsi que le token qui a traversé le réseau.
Une fois tous les tokens propagés, le système choisit une des règles déclenchables et évalue sa partie action. L'évaluation de cette partie action aura le plus souvent pour effet de modifier un ou plusieurs objets, et donc de changer l'état d'instanciation des règles de la base.
Afin d'informer le système de cette modification, le message `modified` doit être explicitement envoyé à l'objet venant d'être modifié afin de mettre à jour le réseau Rete et le conflict set.

Les objets importants d'Opus sont donc les suivants (Cf Figure 4) :

Les bases de règles
: Elles servent de support textuel pour les règles, et contiennent aussi les méthodes d'activation.

Les règles et les prémisses
: Les règles sont analysées par un parser spécialisé, qui les décompose en prémisses + partie action en associant un **type** aux prémisses. Le type renseigne en particulier sur la nature des variables dans la prémisse (libre ou liée, locale, globale).

Les classes dynamiques
: sont les classes dans lesquelles sont implémentées les méthodes pour chaque prémisse. Leurs instances sont les tokens qui sont propagés dans le réseau Rete de la base de règles.

Les tokens
: représentent les jeux d'instanciation. Ils sont instances de la classe dynamique de la base de règles.

Le réseau Rete
: est associé de manière univoque à chaque base de règles.

Les nœuds Rete
: constituent le réseau et servent à produire, tester et propager les tokens.

Les règles déclenchables
: Une règle est dite déclenchable s'il existe un jeu d'instanciation (un token) vérifiant toutes ses prémisses. Par abus de langage, on appelle, *règle déclenchable*, l'ensemble {règle + jeu d'instanciation satisfaisant les prémisses de la règle}.

les conflict sets

représentent à tout moment l'ensemble des règles déclenchables.

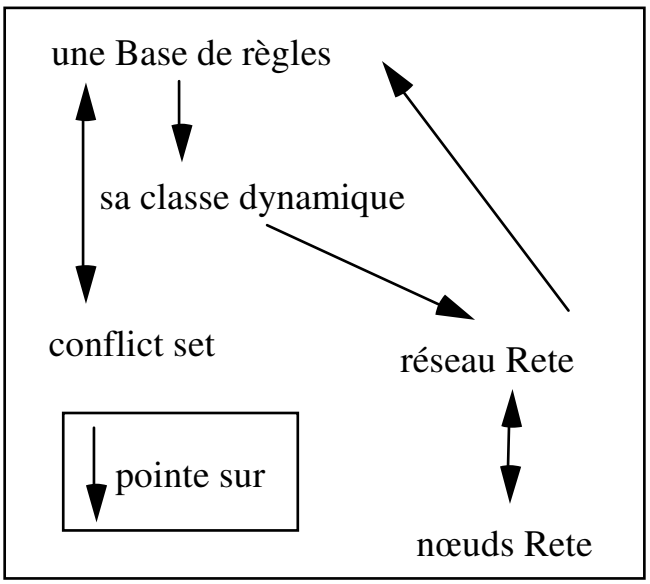


Figure 4. Les liens entre les agents Opus

D'une certaine manière, toute la richesse d'Opus repose sur la réification des tokens, et la notion de classe dynamique, qui permettent une simplification considérable des calculs de tests. Cependant nous nous éloignons déjà beaucoup de la description des auteurs sur plusieurs points :

- Il n' y pas d'interprète à proprement parler. Cet interprète sera délocalisé au niveau de chacune des bases de règles. Nous verrons comment de manière générale Opus répond au problème de l'évaluation et du contrôle d'une base de règles (Cf. chapitre VI).
- Il n' y a pas de mémoire de travail. Nous considérons en effet que la notion de mémoire de travail ne justifie pas la définition d'une classe à part entière. L'évaluation des bases de règles, telle que nous la décrirons, permettra de prendre en compte suffisamment d'informations pour gérer l'ensemble des objets devant être considérés par les règles.
- Nous faisons intervenir des objets nouveaux (par rapport bien sûr à la description des auteurs) : les règles déclenchables et les prémisses.

III.3.2. Les règles

La nature textuelle des règles s'accommode bien de leur parenté avec les méthodes Smalltalk. Elles partagent, en particulier, le même `Browser`, et les mêmes fonctionnalités de l'environnement. Les règles, à l'instar des méthodes Smalltalk, mais au contraire des règles OPS5 (qui sont anonymes), ont un nom (un symbole), mais ne sont jamais utilisées comme réponse à un envoi de message. Le nom des

règles joue uniquement un rôle dans l'organisation des règles en bases de règles (il jouera aussi un rôle pour l'héritage des bases de règles, Cf. IV.1.7).

III.3.2.1. Syntaxe sommaire des règles Opus

Nous donnons ici la syntaxe réduite des règles. Nous y avons ajouté d'autres constructions, en particulier :

- les variables locales en partie prémisse,
- les variables locales déclenchantes,
- les variables globales,
- les prémisses récursives.

Voir le chapitre XI pour une description de ces extensions, et la syntaxe complète.

<règle Opus> ::==
    <nom>
    <déclaration de variables>
    <prémisses>
    actions
    <partieAction>

```
<nom>              ::==  un symbole alphanumérique
<déclaration de variables> ::== | <declaration>  { . <declaration> } *|
<declaration>      ::==  <type> <nomDeVariable> {<nomDeVariable> }*
<type>             ::==  <nom de classe Smalltalk> <mot-clé>
<prémisses>        ::==  <prémisse>
                   ou    <prémisse>  <prémisses>
<prémisse>  ::==  <prémisse positive>
                   ou    <prémisse négative>
<prémisse positive> ::== <expression Smalltalk>.
<prémisse  négative>::==  NOT  "{"  <declaration  de  variables>  <code
Smalltalk>" }".
<partieAction>     := <code Smalltalk>
<code Smalltalk>  ::== <expression Smalltalk> { . <expression Smalltalk>}*.
```

Donnons un exemple typique de règle Opus. Nous disons dans la règle suivante que : "*si p est une porte fermée à clef, alors on introduit la clef dans la serrure, on la tourne et on ouvre la porte.*"
Nous verrons plus loin comment cette règle peut être réécrite, en utilisant les variables locales et locales déclenchantes.

Exemple (1) :

```
ouvrirPorte
"si la porte p est fermee a clef, on entre la clef dans la serrure, on ouvre la serrure,
et on ouvre la porte"
    | Porte p. Serrure s. Clef c |
    p estFermee.
    s == p serrure.
    c ==  s clef.
actions
    c entreDans: s.
    c tourne.
    p ouvre.
    p modified.
```

Nous verrons que les variables apparaissant dans les règles peuvent être de natures diverses, dépendant du type spécifié en partie déclaration.
Nous allons ici donner une première définition des variables d'ordre un, les plus fréquemment rencontrées dans les règles.

Définition des variables d'ordre un
Une variable de règle est dite *d'ordre un* si son type (spécifié en partie déclaration) est une classe Smalltalk[14]. Elle est alors considérée universellement.

III.3.2.2. Analyse syntaxique des règles

La nature des prémisses des règles étant très riche, nous avons introduit la classe `OpusPremisse` pour structurer toutes les informations les concernant :

Le type des variables apparaissant dans la prémisse :
Les variables utilisées dans les règles peuvent être de nature très différentes. Une variable est dite libre si elle apparaît pour la première fois dans la règle, liée sinon. Nous étendrons la notion de type de variable plus loin, en y incorporant la notion de variable locale, globale et locale déclenchante.

Le type de la prémisse
De façon à constituer le réseau Rete, la notion de type est introduite pour les prémisses. Le type de la prémisse dépend des types des variables de la prémisse. Ce type servira à choisir la bonne classe de nœud Rete lors de la mise à jour du réseau. Par exemple, une prémisse avec une seule variable libre aura le type `#oneVar`, une prémisse sans variable libre le type `#noVar`, etc.. Voir en annexe la liste complète des types possibles.

Le nom des variables

[14] Classe système ou utilisateur, et non pas un mot-clé.

Les variables de la règle vont être renommées (de i1 à iN)[15], de façon à n'utiliser qu'une seule classe pour les tokens (la classe dynamique) d'une base de règles. L'objet prémisse va donc garder la trace d'un dictionnaire indiquant pour chaque nom de variable utilisé dans le texte original, le nom de la variable une fois renommée. Ce dictionnaire servira d'interface entre l'utilisateur et le système, par exemple dans le mode *pas à pas*, pour inspecter une règle déclenchable en cours d'exécution.

La classe des variables libres
Elle est déterminée par la partie déclaration et sert à produire les liens de la classe vers les nœuds Rete et réciproquement.

La définition de la classe représentant les prémisses est donc :

```
Object subclass: #OpusPremisse
  instanceVariableNames: 'texte dicoOriginalVars dicoTypeVars
dicoClassVars type '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'OPUS-kernel'
```

où :

`texte` est le texte de la prémisse une fois renommée,
`dicoOriginalVars` un dictionnaire nom original/nom renommé,
`dicoTypeVars` un dictionnaire nom renommé/type (libre ou lié),
`dicoClassVars` un dictionnaire nom renommé/classe,
`type` le type de la prémisse (Voir annexe pour la liste complète).

Les règles sont analysées syntaxiquement par un analyseur spécialisé (`OpusParser`), qui produit pour chaque prémisse de la règle une instance de `OpusPrémisse`, correctement initialisée. Cet objet sera utilisé ensuite pour la création du réseau Rete. La partie conclusion, elle, ne donne lieu à aucun traitement particulier, et sera compilée directement dans la classe dynamique.
Le parser une fois son travail accompli (méthode `parse`), renvoie comme résultat une instance d'`OpusRule`, qui contient les informations relatives à la règle elle-même :

```
Object subclass: #OpusRule
  instanceVariableNames: 'name actions varDeclarations
actionSelector filtersSelectors originalVars premisses ruleBase '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'OPUS-kernel'
```

où `name` est le nom de la règle,
`actions` le texte de la partie actions,

[15] n = 8 a été suffisant pour toutes les applications développées avec NéOpus.

`varDeclarations` un dictionnaire variable/classe,
`actionSelector` le sélecteur de la méthode correspondant à la partie action,
`filterSelectors` la liste des sélecteurs des méthodes correspondant aux prémisses,
`premisses` les prémisses,
`ruleBase` la base de règles contenant la règle.

Le parser Opus est donc structuré de manière parallèle aux règles Opus :

```
Scanner[16] subclass: #OpusRuleParser
 instanceVariableNames: 'name theFilters theActions theVarDeclaration
dicoVars originalVars ruleBaseName thePremisses instSize '
 classVariableNames: 'ActionTypeTable '
 poolDictionaries: ''
 category: 'OPUS-parsing'
```

Il comprend une seule méthode "publique" : `parse`, qui analyse le texte et renvoie une instance d'`OpusRule`, (à l'instar du parser Smalltalk renvoyant un arbre syntaxique) :

```
parse
  self parseName; parseDeclaration; parseFilters; parseAction.
  ^OpusRule   name: name
              varDeclar: theVarDeclaration
              actions: theActions contents
              originalVars: originalVars
              correspondance: dicoVars
              premisses: thePremisses
```

Sans rentrer dans les détails de l'implémentation du parser, nous pouvons dissocier deux fonctions importantes de celui-ci dans notre architecture.

- Renommage des variables
Le renommage des variables est une pure commodité d'implémentation, mais nous permet de n'avoir qu'une seule classe de token pour une base de règles donnée. Ne pas renommer les variables aurait conduit à avoir une classe de token par règle, à l'instar du système Oks (bien qu'utilisant un mode de compilation différent). Décidant que cela aurait considérablement alourdi la mécanique de création/propagation des tokens, nous avons préféré complexifier le parser pour simplifier la gestion de la propagation.

- Macro-expansion
Un certain nombre de messages utilisés dans les règles Smalltalk doivent être expansés pour être utilisés correctement. C'est le cas par exemple des messages de modification (comme le `modified`). Une phase de macro-expansion va permettre de modifier le texte de la règle, pour y ajouter les paramètres nécessaires. Nous

[16] `OpusParser` est sous-classe de `Scanner` et non de `Parser` (elle-même sous-classe de `Scanner`) car aucune des fonctionnalités de `Parser` (pas assez général car trop orienté vers l'analyse de code Smalltalk) n'est réutilisable ici.

indiquerons lorsque nécessaire la nature des macro-expansions réalisées par le parser.

Exemple :

Reprenons notre règle précédente, compilée dans la base de règles `ReglesPorte`:

```
ouvrirPorte
"si la porte p est fermee a clef, on entre la clef dans la serrure, on ouvre la serrure,
et on ouvre la porte"
   | Porte p. Serrure s. Clef c |
   p estFermee.
   s == p serrure.
   c ==  s clef.
actions
   c entreDans: s.
   c tourne.
   p ouvre.
   p modified.
```

Cette règle va générer les méthodes suivantes, qui seront compilées dans la classe dynamique `ReglesPorteDynamic` :

```
!ReglesPorteDynamic methodsFor: 'ouvrir'!

P101
    ^(i1 estFermee)

P102
    ^(i2 == i1 serrure)

P103
    ^(i3 == i2 clef)

ouvrirPorte
  i3 entreDans: i2.
  i3 tourne.
  i1 ouvre.
  i1 modifiedFor: self ruleBase.
```

Voir le §IV.6.3 pour la nature exacte des macro-expansions pour le `modified`.

III.3.3. Les bases de règles

III.3.3.1. Une triple abstraction

Les bases de règles sont définies comme des sous-classes d'une classe abstraite `OpusRuleSet`. Les règles sont écrites comme des méthodes pour cette classe (comprendre des méthodes d'instances). Comme nous l'avons vu plus haut, ces textes sont analysés par un parser particulier, qui produit une instance d'`OpusRule`

dont les différents éléments sont ensuite compilés, de manière standard, dans la classe dynamique.
La compilation des règles consiste à produire autant de méthodes (Smalltalk) que la règle comporte de prémisses, ainsi qu'une méthode (Smalltalk) correspondant à la partie action de la règle. Ces méthodes seront utilisées par les nœuds du réseau Rete, pour faire les tests des jeux d'instanciation, et pour exécuter la partie action. Les variables utilisées dans la règle sont renommées et dénotent les *variables d'instances* des tokens propagés dans le réseau Rete.

Les bases de règles, en tant que classes Smalltalk servent ainsi de support d'organisation aux règles. Les règles apparaissent comme des méthodes d'instance de ces classes, mais sont compilées uniquement pour pouvoir bénéficier des fonctionnalités du browser Smalltalk.
Ces classes d'un point de vue objet sont des classes abstraites : aucune instance n'est supposée en être créée. Elles ne définissent donc aucune structure (variable d'instance), ni aucune méthode d'instance.
La racine de ces classes est la classe `OpusRuleSet`, sous-classe d'`Object` :

```
Object subclass: #OpusRuleSet
  instanceVariableNames: ''
  classVariableNames: ''
```

`OpusRuleSet` joue le rôle d'une classe abstraite à trois titres :

- Au sens traditionnel des langages Objets : aucune instance n'est sensée être créée d'`OpusRuleSet`.
- Si les classes abstraites en programmation par objets servent à définir des comportements généraux spécialisés dans les sous-classes[17], (qui elles ne sont plus abstraites), ce n'est pas le cas ici puisque les sous-classes (les bases de règles) sont elles-mêmes *toutes* abstraites.
- Enfin, `OpusRuleSet` est abstraite dans le sens où elle ne représente pas une base de règles elle-même : aucune règle ne peut être écrite de façon générale, et applicable à tous les domaines d'expertise[18] ! Cela se traduira dans le code par un certain nombre de tests et d'indirections dans les méthodes d'`OpusRuleSet class`. (Cf. 3.3.4. à propos de ces indirections).

En revanche, toute la structure de ces bases de règles se situe au niveau de leur métaclasse. Deux attributs sont pour l'instant nécessaires : la *classe dynamique*, dans laquelle seront compilées les diverses méthodes Smalltalk correspondant aux prémisses et à la partie action des règles, et le *conflict set* qui servira lors de l'activation.

---

[17] Comme par exemple la classe `Collection`, qui définit les méthodes communes à toutes ses sous-classes.
[18] A notre connaissance.

```
OpusRuleSet class
  instanceVariableNames: 'dynamicClass conflictSet'
```

L'initialisation d'une base de règles va donc consister à créer la classe dynamique ainsi qu'une instance de `ConflictSet`. Le réseau Rete est aussi créé pour la circonstance, pointé par la classe dynamique.
Notons ici que la classe Smalltalk `OpusRuleSet` n'étant pas une base de règles une indirection est prévue pour initialiser cette dernière (méthode `initOpusRuleSetOnly`).

noter l'indirection →

```
!OpusRuleSet class methodsFor: 'initializations'!

initialize
 "on cree une classe dynamique et un reseau Rete.
 Attention : OpusRuleSet itself n'est pas une base de regles"

 self == OpusRuleSet ifTrue: [^self initOpusRuleSetOnly].
 self initConflictSet.
 dynamicClass <- superclass dynamicClass newDynamicClass: name.
 dynamicClass network: (OpusNetwork newWithBase: self).
 self initReteNodes.
 self initContext.
 self initTypage

initOpusRuleSetOnly
 "initialisation de OpusRuleSet, mais pas de ses sous classes"

 dynamicClass <- OpusToken.
 (NewColl <- OrderedCollection new.)
 self setSimpleTypage

initConflictSet
 conflictSet <- self defaultConflictSetClass new.
```

Il faut remarquer que l'environnement de programmation utilisé pour compiler ces règles - à savoir le browser, les mécanismes de références croisées et la gestion des fichiers textes (`fileIn` et `fileOut`) - n'a pas besoin d'être modifié et marche gratuitement.

III.3.3.2. Création d'une base de règles

Les bases de règles se créent canoniquement en fabriquant une sous-classe de la classe `OpusRuleSet`[19]. Aucune structure supplémentaire n'est à prévoir, mais nous

[19] Ou d'une base de règles déjà existante, si l'on utilise l'héritage de bases de règles, Cf Chapitre IV.1.7.

laissons cependant la possibilité de définir des variables de classes[20]. Ainsi le message de création raccourci suivant est-il défini de la manière suivante :

```
OpusRuleSet subclass: #unNomDeBaseDeRegles
  classVariableNames: 'des variables de classe'
  category: 'une categorie'
```

Le message de création standard est redéfini de manière à initialiser la nouvelle base de règles à l'aide de la méthode définie plus haut :

```
!OpusRuleSet class methodsFor: 'subclassing'!

subclass: t1 instanceVariableNames: t2 classVariableNames: t3
poolDictionaries: t4 category: t5
  "intercepte pour initialiser la base de regles correctement"

  | newOne |
  self checkForExistingDynamicClass: t1.
  newOne <- super subclass: t1 instanceVariableNames: t2
              classVariableNames: t3 poolDictionaries: t4 category: t5.
"on recompile (statiquement) les regles eventuellement heritees"
  newOne initialize; compileAllFromSuperClass.
  ^newOne
```

La création de bases de règles étant confondue avec la création de sous-classes, nous confondrons, par la suite, les termes *sous-classe* et *sous-base*, pour les bases de règles.

III.3.3.3. Compilation des règles

Pour bénéficier de l'environnement de programmation gratuitement, il faut "leurrer" le système, en lui faisant croire que les règles que l'on écrit sont des méthodes Smalltalk. Ceci se fait simplement en redéfinissant la méthode de compilation standard Smalltalk pour les bases de règles, de manière à compiler dans la base de règles une méthode dont le code source est le texte de la règle et le code Smalltalk est vide (ou, plus exactement, n'est que le nom de la règle) :

[20] Nous verrons plus loin que l'on peut utiliser les variables de classe des bases de règles comme variables d'ordre zéro. Cf. la notion d'objets nommés au IV.1.5.

```
!OpusRuleSet class methodsFor: 'compiling'!

compile: code classified: heading notifying: requestor
   | opusRule  |
   self == OpusRuleSet ifTrue:
      [^super compile: code classified: heading notifying: requestor].
   opusRule := (OpusRuleParser new) initOn: code ruleBaseName: self name.
   opusRule parse; ruleBaseName: self name.

"on compile la regle dans la classe dynamique"
   dynamicClass compileRule: opusRule classified: heading
notifying: requestor.

"on compile un leurre dans la base de règle"
^self compileRule: code name: opusRule name classified: heading notifying:
requestor
```

La compilation du leurre consiste simplement à passer comme argument pour le texte à compiler le nom de la règle, tandis que le code source est le texte de la règle (ce qui permet de garder les fonctionnalités de browsing) :

```
compileRule: code name: leNom classified: heading
    notifying: requestor
  "ne compile que le nom de la regle dans la classe"

  | selector |
  selector _ self compile: leNom  notifying: requestor ifFail: [^nil].
  (methodDict at: selector) putSource: code class: self category: heading
  inFile: 2.
  self organization classify: selector under: heading.
  ^selector
```

le leurre

III.3.3.4. Une métaclasse cachée ?

L'indirection systématique dans les méthodes de *OpusRuleSet class* est symptomatique de la présence cachée d'une métaclasse. En effet, les bases de règles sont définies uniquement en termes d'héritage (comme sous-classes de `OpusRuleSet`), mais pas en termes d'instanciation : nous ne pouvons pas en Smalltalk définir la *classe* des bases de règles, puisque celles-ci étant déjà des classes, il nous faudrait définir une métaclasse générale, ce qui pose problème.
En effet, bien que les classes en Smalltalk soient des objets à part entière, leurs métaclasses n'ont pas un statut de première classe, comme en ObjVlisp [Cointe 87]. En particulier elles :

- sont créées automatiquement par le système à la création d'une classe,
- n'ont qu'une seule instance,
- sont anonymes.

"Programmer par métaclasses" n'est donc possible qu'en termes de spécification d'attributs et de méthodes pour une classe qui est fabriquée par le système.

L'architecture de métaclasses de Smalltalk ne permet pas toutes les manipulations souhaitables sur les métaclasses.
Une autre architecture conviendrait mieux à la nature des bases de règles, si l'on disposait d'un système avec métaclasses à part entière (comme l'environnement ClassTalk [Cointe&Briot 89], qui permet de créer soi-même ses propres métaclasses, dans la tradition ObjVlisp [Cointe 87]).

Ce point se fait cruellement sentir lorsqu'on désire utiliser concurremment une même base de règles pour deux utilisations (ou "instances" dans le sens courant - i.e. non objet - du terme) différentes. En effet, le réseau Rete et le conflict set sont *uniques* pour une même base de règles. On ne peut les "instancier" pour des utilisations particulières de la base.
C'est le cas par exemple pour les systèmes d'acteurs (Cf. l'extension de NéOpus vers les acteurs au Chapitre VII.7.7, ou le système de bridge au Chapitre VI-II.7.5, ou encore la non-réflexivité des méta-bases au chapitre VI.7.1). Nous énonçons ce problème comme le problème de la métaclasse cachée :

Problème de la métaclasse cachée
La nature même des bases de règles (représentées par des *classes*) empêche l'utilisation simultanée de deux "instances" distinctes d'une même base de règles

Une solution consisterait à considérer `OpusRuleSet` non pas comme la racine du graphe d'héritage pour les bases de règles mais comme leur métaclasse. On créerait alors la classe `MetaOpusRuleSet` comme racine du graphe d'instanciation des bases de règles. `MetaOpusRuleSet` contiendrait ainsi toutes les méthodes d'initialisation sans aucune indirection (Cf. Figure 5). Les bases de règles seraient alors créées comme sous-classe de `Object`, et *instances* de `MetaOpusRuleSet`.

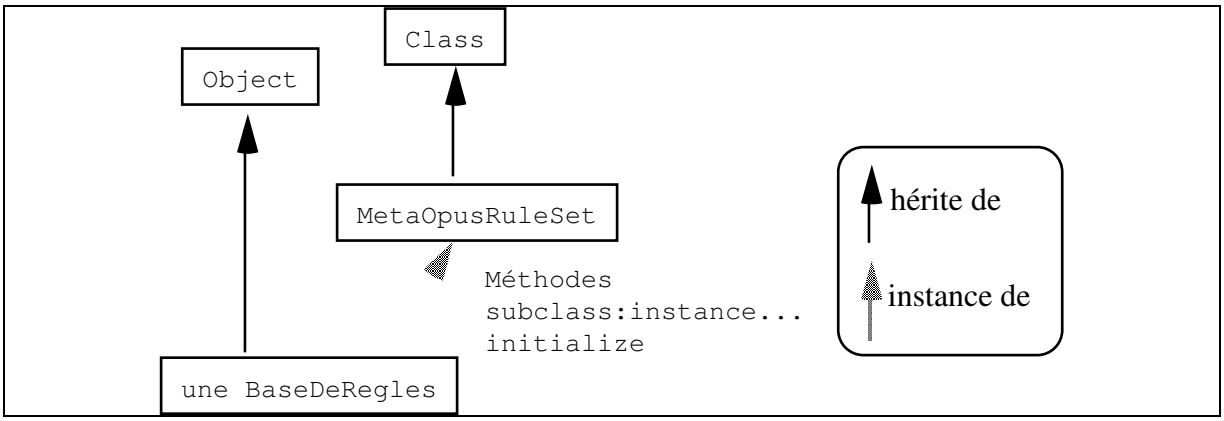


Figure 5. Une autre implémentation de `OpusRuleSet`

Mais l'architecture de métaclasses de Smalltalk, bien que théoriquement moins puissante que celle d'ObjVlisp (ou de ClassTalk) possède une particularité très utile : celle de proposer une programmation par métaclasse systématique (chaque classe ayant sa propre métaclasse, on programme ainsi systématiquement à deux niveaux), que ne propose pas, en standard ObjVlisp. Or nous utiliserons extensivement cette

programmation à deux niveaux pour les bases de règles. Nous gardons donc notre architecture, qui bien que conceptuellement moins satisfaisante nous permettra à moindre frais d'implémenter un mécanisme d'héritage pour les bases de règles. Notons enfin que ce problème de métaclasse cachée sera repris au Chapitre V.1.7.7 où nous verrons comment l'utilisation de l'héritage de bases de règles permettra de résoudre notre problème.

III.3.4. Les classes dynamiques pour les tokens

III.3.4.1. Nature des tokens

Les tests correspondant aux prémisses sont implémentés comme des méthodes Smalltalk. Cette solution a été choisie en faveur de celle consistant à utiliser des blocks (des instances de la classe Smalltalk `BlockContext`) comme c'est le cas dans le système Humble [Piersol 86]. Pour des raisons d'efficacité, liées au coût du passage de paramètres et de leur indexation, l'utilisation de méthodes compilées a été préférée. La classe dans laquelle sont compilées ces méthodes est appelée *classe dynamique*, et sera associée à la base de règles correspondante. Elle représente de facto les fameux tokens : tester une prémisse sur un token revient alors simplement à lui envoyer le message correspondant. Cette solution a d'énormes avantages sur le plan de la gestion des variables de règles : les variables d'une règle sont représentées directement par des variables d'instances des tokens. On s'affranchit alors des lourdeurs de la gestion d'environnements de liaison traditionnellement attachée à la construction de moteurs d'inférence [Voyer 87]. Ainsi, chaque réseau Rete connaît sa classe dynamique, à partir de laquelle les tokens seront créés, testés et propagés.

III.3.4.2. Une (deuxième) incursion dans ObjVlisp

L'architecture proposée par les auteurs pour l'implémentation des classes dynamiques est assez particulière. Ces classes sont créées à partir d'une métaclasse spéciale : `OpusTokenBehavior`, dont elles sont des instances. `OpusTokenBehavior`, sous classe directe de `Behavior` (qui contient le minimum d'informations nécessaires pour créer des classes), contient aussi certaines informations permettant aux tokens de communiquer avec le reste du système.
`OpusTokenBehavior` est donc une métaclasse à plusieurs instances, ce qui est inhabituel en Smalltalk [Cointe&Briot 89, Pachet 89] (Cf. Figure 6). Chaque réseau Rete instancie cette métaclasse pour créer sa propre classe de tokens.

Après avoir essayé d'implémenter cette architecture, ce qui demande un certain remaniement de la structure profonde du système Smalltalk, notamment au niveau de la classe `Behavior`, nous avons abandonné cette solution (nous n'avions pas à l'époque le système ClassTalk).

Il est relativement facile en effet de pervertir le système Smalltalk de façon à pouvoir créer des classes et des métaclasses "à la ObjVlisp", c'est à dire, en spécifiant à la fois sa superclasse et sa métaclasse, afin de briser le parallèle soigneusement entretenu par le système Smalltalk entre la hiérarchie des classes et celle de leurs métaclasses. Cela a été fait aussi dans le système ICEO [Bourgeois 90], de manière à créer des classes ayant la structure d'ensembles. En revanche, produire un environnement de programmation assurant la compatibilité entre ces classes "à la ObjVlisp" et le reste du système est moins simple, et a fait l'objet de développements particuliers [Cointe&Briot 89].

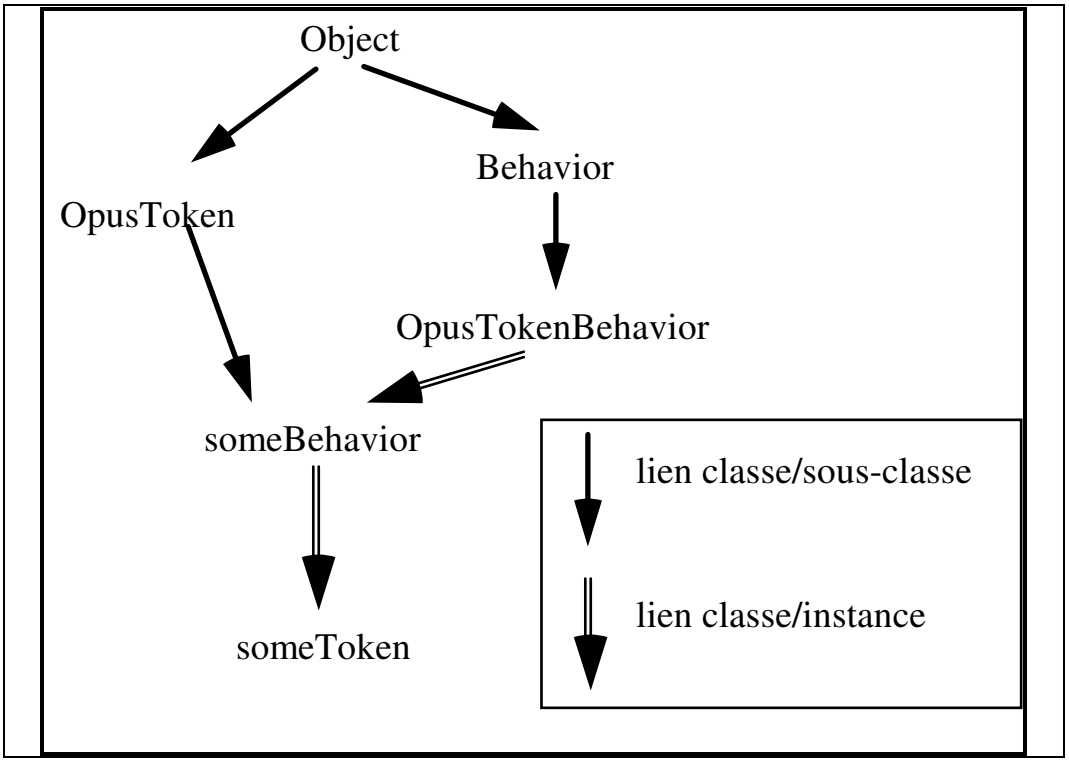


Figure 6. Architecture proposée par les auteurs

Or, dans notre cas il est parfaitement possible d'implémenter les classes dynamiques sans utiliser de métaclasse spécialisée, en utilisant simplement les variables de métaclasse.
Nous avons donc choisi de garder une architecture plus classique, en produisant la classe `OpusToken` comme sous-classe d' `Object`, et en enrichissant sa métaclasse :

```
Object subclass: #OpusToken
  instanceVariableNames: 'i1 i2 i3 i4 i5 i6 i7 i8 '
  classVariableNames: 'Index '
  poolDictionaries: ''
  category: 'OPUS-kernel'
```

La classe `OpusToken`, racine de toutes les classes dynamiques, définit un certain nombre de variables d'instances, représentant les variables utilisées dans les règles. Les variables des règles sont renommées (voir l'analyse syntaxique) de i1 à iN (N = 8 par défaut). Si une règle contient plus de huit variables, la classe dynamique de la

base de règles est recompilée de façon à ajouter une ou plusieurs variables d'instances supplémentaires.
`OpusToken` implémente d'une part des méthodes d'accès permettant de connaître la valeur d'une variable d'instance par son index (utilisées par les nœuds Rete à l'évaluation du réseau) et d'autre part des méthodes permettant de tester si le token pointe sur un objet donné :

```
!OpusToken methodsFor: 'access'!

field: n
"rend la valeur de la n ieme var. d'instance"
^self instVarAt: n!

setField: n to: aValue
"modifie la valeur de la n ieme var. d'instance"
self instVarAt: n put: aValue

!OpusToken methodsFor: 'testing'!

contains: x
  1 to: self class instSize do:
        [:i | (self instVarAt: i) == x ifTrue: [^true]].
  ^false
```

La métaclasse d'`OpusToken` (`OpusToken class`) va pointer sur un réseau Rete et sera chargée de la compilation d'une règle. Cette compilation aura pour effet de :

- Compiler les textes des prémisses et de la partie action sous forme de méthodes Smalltalk.
  Pour les prémisses, le texte de la méthode compilée sera composé d'un sélecteur calculé (`'P'` suivi d'un entier incrémenté automatiquement), suivi du texte de la prémisse (renommée et expansée lors de l'analyse syntaxique).
  Pour la partie conclusion, le texte sera un sélecteur égal au nom de la règle, suivi du texte de la partie action (lui-aussi préalablement renommé et expansé).
- Mettre à jour le réseau Rete.

La méthode de compilation est représentée Figure 7.

```
!OpusToken class methodsFor: 'compiling'!

compileRule: aRule classified: prot notifying: t3
  | newSelector code listeSelecteursPremisses unTexte |
"on enleve la règle si elle existait deja"
 self removeRule: aRule name.
"on compile la partie action de façon standard"
 aRule actionSelector: (super
   compile: aRule name , aRule actions
   classified: prot notifying: nil).
"on compile les prémisses"
 listeSelecteursPremisses  <- OrderedCollection new.
 aRule premisses do:
   [:p |
   unTexte <- p texte.
   newSelector <- self newSelector.
   code <- newSelector , ' ^(' , unTexte  , ')'.
   listeSelecteursPremisses  add: newSelector .
   super compile: code classified: prot notifying: nil].
 aRule filtersSelectors: listeSelecteursPremisses.
"mise à jour du réseau Rete"
 network addNodesForRule: aRule
```

le texte de la méthode implémentant la partie action

le texte de la méthode implémentant une prémisse

Figure 7. La méthode de compilation d'une règle dans la classe dynamique.

III.3.5. Le réseau Rete

III.3.5.1. Adaptation de Rete aux objets Smalltalk.

Les différences de nature entre les règles OPS5 et les règles Opus ont de grandes conséquences sur le réseau Rete de compilation Opus. En fait nous n'allons garder de Rete que l'idée de nœuds de mémorisation à deux mémoires.
Rappelons les différences importantes de nature entre OPS5 et Opus (Cf Chap. III.2.5):

- renversement du rôle des variables,
- extension des contraintes à toute expression Smalltalk,
- possibilité de filtrer plusieurs objets dans une seule prémisse.

Ces points amènent à une autre vision du réseau Rete :

- Il ne peut y avoir de réseau de discrimination. Celui-ci est remplacé implicitement par le lien d'instanciation des objets vers leur classe.
- Le réseau doit s'accommoder de la disparition des relations n-aires : les prémisses peuvent être tout message Smalltalk. Il n'est ainsi plus possible d'établir un réseau de discrimination.
- Les nœuds doivent pouvoir filtrer plus d'un nouvel objet par prémisse (nœuds multi-entrée).

L'exemple suivant montre une prémisse multi-entrée, filtrant trois objets à la fois :

```
troisEntiersSuccessifs
    | Integer a b c|
a = (b + 1) and: [b = c + 1].
    actions
Transcript show: 'il y a trois entiers consécutifs';cr.
```

Ce nouveau réseau Rete est bien différent de l'ancien. Il sera en particulier difficile de factoriser les nœuds pour plusieurs règles ayant une prémisse en commun. Nous allons maintenant étudier sa structure précise.

III.3.5.2. Un réseau en Smalltalk

De nombreuses classes existent en Smalltalk, qui implémentent toutes sortes de structures de données collectives : listes ordonnée (`OrderedCollection`), listes triées (`SortedCollection`), sacs (`Bag`), ensembles (`Set`), tableaux (`Array`), dictionnaires (`Dictionary`), ...
Il n'existe cependant aucune classe pour représenter les structures arborescentes comme les réseaux, composée de nœuds et de liens entre ces nœuds. Mais la nature associative des objets Smalltalk, qui fait que l'environnement Smalltalk peut être envisagé comme un vaste réseau de pointeurs, rend très facile l'implémentation des structures de réseau.

La classe **`OpusNetwork`** implémente les fonctionnalités requises pour la création d'une structure de réseau Rete (un graphe orienté sans cycle) et pour son évaluation. La définition du réseau Rete est donc :

```
Object subclass: #OpusNetwork
 instanceVariableNames: 'ruleBase dispensers dicoRules '
 classVariableNames: ''
 poolDictionaries: ''
 category: 'OPUS-network'!
```

où :

`ruleBase` : est la base de règles associée,
`dispensers` : la liste des classes apparaissant en partie déclaration d'au moins une des règles de la base de règles. Ces classes sont appelées dispenser, pour la base de règles en question,
`dicoRules` : un dictionnaire associant pour chaque instance de règle (`OpusRule`) de la base de règles, la liste des nœuds Rete correspondant à ses prémisses et à sa conclusion. Ce dictionnaire sert à maintenir la gestion des nœuds, notamment lors de suppression ou redéfinition de règles.

III.3.5.3. Création du réseau Rete

Le réseau est modifié à chaque fois qu'une règle est compilée ou supprimée. La méthode principale est la méthode `addNodeForRule:` appelée par la classe dynamique lors de la compilation de la règle pour créer les nœuds. Cette méthode crée les nœuds Rete correspondant aux prémisses analysées par le parser de la façon suivante :

- pour chaque prémisse de la règle, on crée un nœud Rete correspondant au type de la prémisse,
- on initialise le nœud en lui donnant le sélecteur de la méthode à envoyer aux tokens qui passeront par le nœud,
- on relie ce nœud au nœud précédent (s'il n'est pas le premier)

Pour chaque variable libre de la prémisse, on relie la classe de la variable au nœud. Ce lien est double : d'une part du nœud vers la classe (voir les nœuds) et d'autre part de la classe vers le nœud (voir les dispensers),

```
addNodesForRule: aRule

| newNode precedingNode selectors finalNode
listeNodes i |

 self removeNodesForSelector: aRule name.
 selectors _ aRule filtersSelectors.
 listeNodes _ OrderedCollection new.
 i _ 0.
"pour chaque prémisse"
 aRule premisses do:
  [:unePremisse |
  i _ i + 1.
  newNode _ self newNodeFor: unePremisse.
  self connectDispensersToNode: newNode
   fromPremisse: unePremisse
   inRule: aRule.
  precedingNode notNil
   ifTrue: [newNode precedingNode: precedingNode.
      precedingNode followingNode: newNode].
  precedingNode _ newNode.
  newNode selectorToCall: (selectors at: i).
  newNode initVat;dummyLimit: aRule dummyLimit.
  listeNodes add: newNode].
 listeNodes add: (finalNode _ self newNodeFinal).
 precedingNode followingNode: finalNode.
 finalNode precedingNode: precedingNode;
    selectorToCall: aRule actionSelector;
dummyLimit: aRule dummyLimit.
 dicoRules at: aRule put: listeNodes.
```

on commence par enlever les nœuds de la règle si elle existait déjà

prenons un nouveau nœud correspondant au type de la prémisse

on connecte les dispensers au noeud

un nœud particulier pour la partie action

on relie les nœuds entre eux

on met à jour le dictionnaire dicoRules

La connexion des dispensers aux nœuds consiste à établir les liens entre les dispensers et les nœuds :

```
connectDispensersToNode: aNode fromPremisse: unePremisse inRule: aRule
| dispName disp |
unePremisse goodVars do:
      [:v |
      dispName _ aRule dispenserOf: v.
      disp _ Smalltalk at: dispName.
      aNode addDispenser: dispName toVar: (self numberOf: v).
      self ruleBase addNode: aNode for: disp.
      self addDispenser: disp
```

La création des nœuds dépend du type de la prémisse (calculé par le parser). La méthode adéquate est donc une grande conditionnelle qui ne comporte que des tests de type (que l'héritage de classe, pour une fois, ne permet pas de faire disparaître !) :

```
!OpusNetwork methodsFor: 'creating nodes'!

newNodeFor: aPremisse
 "cree une instance de nœud pour le type de la premisse"

 | type |
 type _ aPremisse type.
 type = #general ifTrue: [^OpusNode newFrom: self].
 type = #noVar ifTrue: [^OpusNodeNoDisp newFrom: self].
 type = #oneVar ifTrue: [^OpusNodeOneVar newFrom: self].
 type = #localVar0 ifTrue: [^OpusNodeLocalVar newFrom: self].
 type = #localVarGeneral ifTrue: [^OpusNodeLocalVar newFrom: self].
 type = #localVarOne ifTrue: [^OpusNodeLocalVar newFrom: self].
 type = #localVarDec0 ifTrue: [^(OpusNodeAssign0 newFrom: self) assign:
(self numberOf: aPremisse variable); yourself].
 type = #negativeOneVar ifTrue: [^OpusNodeOneVarNegative newFrom: self].
 type = #negativeGeneral ifTrue: [^OpusNodeNegative newFrom: self].
 type = #createGoalNoVar ifTrue: [^OpusNodeCreateGoalNoVar newFrom: self].
 type = #createGoalVar ifTrue: [^OpusNodeCreateGoalVar newFrom: self].
 ^self error: 'type de premisse non encore implemente : ' , type
```

III.3.5.3.1. Création des tokens

Les tokens sont créés à partir de la classe dynamique. Ils sont soit vides soit copies d'un autre token (en utilisant alors la copie superficielle définie par le message `shallowCopy`) :

```
!OpusNetwork class methodsFor: 'token creation'!

newToken
  ^self dynamicClass new

newTokenCopyOf: t
  ^t shallowCopy
```

III.3.5.3.2. Gestion de l'organisation des règles/nœuds

La classe `OpusNetwork` est chargée de maintenir un dictionnaire règle/liste de nœuds (`dicoRules`), de manière à assurer la cohésion du système, notamment lors de redéfinition ou suppression de règles.
Ce dictionnaire permet par exemple de calculer la liste des nœuds du réseau :

```
listOfNodes
|o|
o _OrderedCollection new.
dicoRules keys asOrderedCollection do: [:k| o addAll: (dicoRules at: k ) ].
^o
```

Un certain nombre de méthodes à caractère utilitaire permettent de gérer cette liste, pour retrouver les instances d'`OpusRule` par leur nom ou par un sélecteur de méthode de la classe dynamique :

```
opusRuleNamed: aName
"pour chercher l'instance d'OpusRule de nom aName"
  dicoRules keys do: [:k | (k name = aName) ifTrue: [^k]].
  ^nil

opusRuleOf: aSelector
"pour chercher l'OpusRule qui a un sélecteur de nom aSelector"
  dicoRules keys do: [:r| (r includesSelector: aSelector) ifTrue: [^r]].
  ^self error: 'pas de telle regle'
```

III.3.5.4. Classification des nœuds

Les auteurs distinguent quatre types de nœuds pour le réseau Rete :

Les *nœuds dispensers*
Ils représentent les racines du réseau, à partir desquelles sont effectuées les opérations d'ajout et de retrait d'objets. A chaque classe présente en partie déclaration d'une règle (les *dispensers*) correspond un "nœud dispenser" dans le réseau.

Les *nœuds positifs*
Ce sont les nœuds correspondant aux prémisses positives des règles.

Les *nœuds négatifs*
Ce sont les nœuds représentant les prémisses négatives.

Les *nœuds terminaux*
Ce sont les nœuds représentant les parties action de règles.

Nous avons modifié et étendu cette classification, pour mieux tenir compte de la diversité des nœuds, et pour optimiser certaines opérations de combinaison lors de l'évaluation. Nous avons par ailleurs supprimé la notion de *nœud dispenser* en ajoutant les fonctionnalités nécessaires à l'envoi des objets dans le réseau au niveau des *classes* Smalltalk elles-mêmes.
La classe racine des nœuds est la classe `OpusNode`, qui définit le comportement général des nœuds Rete. Des spécialisations seront ensuite introduites pour tenir compte des cas particuliers (nœud final, nœud sans variable libre, nœud avec variable locale, nœud négatif ...).

Un nœud Rete est structuré de manière à permettre la fabrication, le test et la propagation des tokens. Sa définition est la suivante :

```
Object subclass: #OpusNode
    instanceVariableNames: 'precedingNode followingNode selectorToCall
leftMemory newTokens inputDispensers network reverseDico'
    classVariableNames: ''
    category: 'OPUS-network'
```

où :
`precedingNode` est le nœud précédent et `followingNode` le nœud suivant.
`selectorToCall` le sélecteur de la méthode à invoquer pour passer le test.
`leftMemory` la mémoire de gauche (elle contiendra la liste des tokens arrivant du nœud précédent).
`newTokens` la mémoire de droite généralisée. La mémoire de droite est ici un dictionnaire, car plusieurs variables peuvent être filtrées en même temps. Ce dictionnaire contient pour chaque variable libre (clé) la liste des nouveaux tokens.
`inputDispensers` est un dictionnaire classe/variable libre; `network` le réseau.
`reverseDico` un dictionnaire inverse (variable libre/classe), utilisé pour raisons d'optimisation.

III.3.5.5. Liens entre classes Smalltalk et nœuds Rete : le problème du dictionnaire de dictionnaires

Afin de lier les classes Smalltalk dont les instances sont propagées dans le réseau (les dispensers), aux nœuds Rete ayant une entrée de cette classe, il faut être capable, pour chaque classe Smalltalk, de fournir la liste de ces nœuds Rete, et ce pour chaque base de règles.

Il y a a priori trois solutions (et demi) à ce problème, que l'on appelle problème du "dictionnaire de dictionnaire".
En notant un dictionnaire entre accolades {clé -> valeur}, voici ces solutions :

1. Implémenter un dictionnaire de dictionnaires, dans une classe "neutre", (comme la classe `Opus`), ayant en clé les bases de règles, et en valeurs des dictionnaires {classe -> liste de nœuds},

1 Bis. Ou bien le contraire : un dictionnaire de dictionnaires :
{classe -> {Base de règles -> liste de nœuds}}.

2. Implémenter un dictionnaire simple au niveau de chaque classe Smalltalk, ayant en clé une base de règles et en valeurs la liste des nœuds :
{base de règles -> liste de nœuds}.

3. Symétriquement, implémenter au niveau des bases de règles, un dictionnaire {classe -> liste de nœuds).

Le choix est alors simple : la solution 1 (et 1 bis) est la plus lourde et la moins "objet" puisqu'elle consiste à représenter l'information en dehors du monde considéré. La solution 2 implique de modifier la structure des classes[21], puisque l'on veut être capable d'utiliser *toute classe* du système.

Nous avons donc choisi de gérer un dictionnaire {base de règles -> nœuds Rete} pour chaque classe Smalltalk, qui sera implémenté dans la base de règles en question. Ce dictionnaire permet pour chaque classe Smalltalk d'accéder à la liste des nœuds Rete de la base de règles, dans lesquels il faudra envoyer ses instances.

La structure des bases de règles est donc enrichie d'une variable d'instance (`reteNodes`) :

```
OpusRuleSet class
  instanceVariableNames: 'dynamicClass conflictSet reteNodes'
```

L'initialisation des bases de règles va prendre en compte cette variable d'instance supplémentaire en l'affectant à un dictionnaire vide (Cf. méthode `initReteNodes`).

Par exemple, dans la base de règles `MonkeyRule` il y aura après compilation des règles un dictionnaire comme celui-ci :

Clés		**Valeurs**
Singe	->	une liste de nœuds
Objet	->	une liste de nœuds
ButSinge	->	une liste de nœuds

Les messages d'accès à ce dictionnaire sont définis dans la classe `OpusRuleSet`. Ils seront utilisés par les bases de règles, pour connaître la liste des nœuds auxquels envoyer les objets au début de leur activation.

[21] En l'occurrence il faudrait rajouter une variable d'instance supplémentaire pour la métaclasse `Class`, et donc recompiler tout le système. Nous nous le sommes déjà interdit !

```
!OpusRuleSet class methodsFor: 'rete nodes'!

initReteNodes
  reteNodes _ Dictionary new

nodesFor: aClass
"la liste des nœuds Rete pour aClass"
  ^reteNodes at: aClass name ifAbsent: [OrderedCollection new]

addNode: aNode for: aClass
"ajouter un nœud a la liste pour aClass"
  reteNodes at: aClass name ifAbsent: [reteNodes at: aClass name
put: OrderedCollection new].
  (reteNodes at: aClass name) add: aNode

removeNode: aNode for: aClass
  "ibid"

hasDispenser: aClass
  ^reteNodes includesKey: aClass name
```

Il faut noter que le problème du "dictionnaire de dictionnaires" est récurrent en programmation objet. En particulier le système Systalk [Wolinski 90] [Perrot&Wolinski 91, 92] s'est confronté à ce problème dans la représentation d'objet multi-facettes.

III.3.5.6. Exemple

Prenons l'exemple de la règle improvisée suivante, tirée du mythique exemple du singe et des bananes. Nous aurons l'occasion de revenir sur le bien fondé de l'écriture de telles règles, seule la création du réseau Rete nous intéresse ici :

```
!SingeEtBananes methodsFor: 'prendre'!

prendreCaisse
|But b. Singe s. Caisse c|
    b actif.
    b acteur = s.
    s neTientRien.
    b action= #tenir.
    b objet = c.
    s position = c position.
actions
    s position: c position.
    s prends: c.
    s modified.
    b statut: #reussi.
    b modified.
```

Le réseau Rete généré sera alors le suivant (Figure 8) :

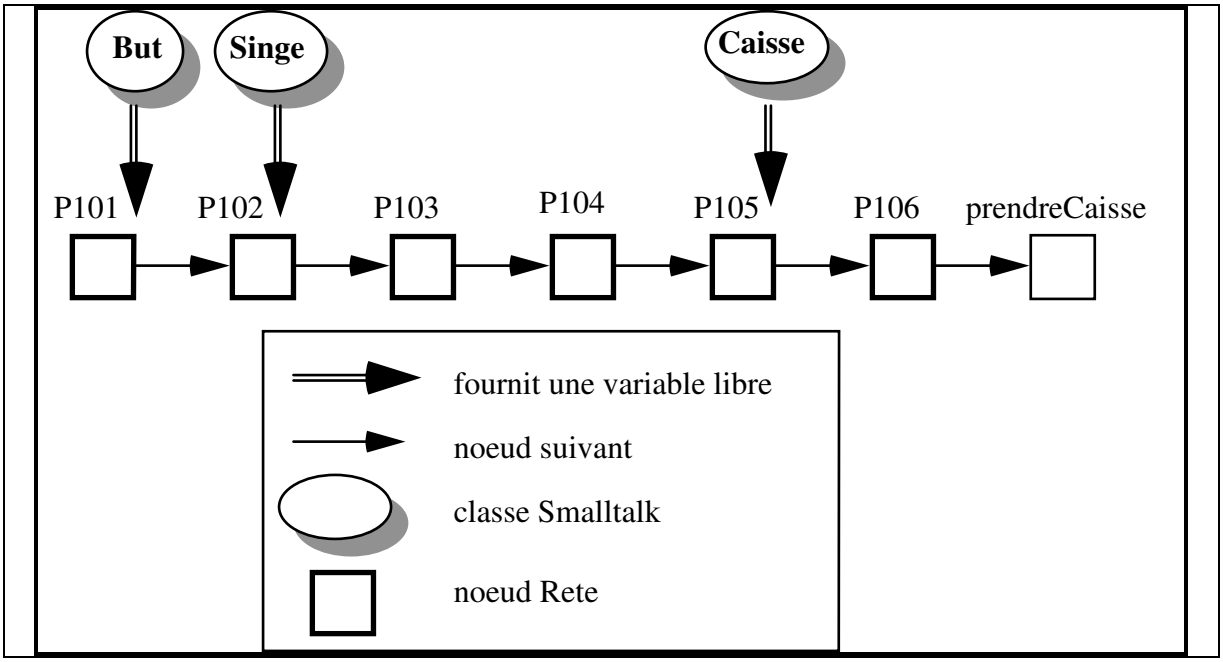


Figure 8. Le réseau Rete généré pour la règle `prendreCaisse`

III.3.5.7. Une optimisation impossible

L'optimisation consistant à factoriser les nœuds correspondant à des prémisses identiques est ici rendue extrêmement difficile pour plusieurs raisons :

1. Une optimisation similaire à celle pratiquée en OPS5, c'est à dire consistant à repérer les prémisses exactement identiques serait possible, bien que plus compliquée qu'en OPS5, à cause de la présence de variables (qu'il faudrait renommer), et de la nécessité de prendre en compte leur type (libre ou liée) dans la recherche.
2. Mais cette optimisation est très maladroite en Opus.
En effet, le principe "Toute expression" induit, à l'inverse d'OPS5 un très grand nombre de manières différentes d'exprimer la même chose. Or la détection de *prémisses synonymes* en Smalltalk est impossible :

- La présence de variables référençant les objets induit un problème d'ordre de celles-ci dans la prémisse. En particulier, il faut être capable de détecter les messages commutatifs (`p1 age = p2 age` est synonyme de : `p2 age = p1 age`), ou ayant un inverse (`p1 age > p2 age` est synonyme de : `p2 age >= p1 age`). Tout un programme !

- Des *messages* Smalltalk peuvent être synonymes (par exemple `p pere notNil` est synonyme de `p pere = nil`).
- Enfin l'utilisation des variables multiplie encore le nombre de combinaisons. Par exemple, comment repérer les identités suivantes à la compilation ? :

```
|Person p1 p2|

p1 age = p2 age.
p2 age > 50. ou  alors : p1 age > 50 ??
```

Nous ne bénéficions donc pas de cette optimisation, pourtant terriblement efficace en OPS5 (où beaucoup de prémisses sont partagées). C'est un des nombreux effets (ici négatif) du principe "Toute expression".

III.3.6. Cycle d'inférence et évaluation

Les nœuds Rete étant reliés entre eux en respectant l'ordre d'écriture des règles, l'activation du réseau consiste à envoyer des tokens (ou jetons) représentant un jeu d'instanciation à partir des classes Smalltalk concernées, et à les propager dans le réseau, à travers les nœuds positifs ou négatifs, jusqu'aux nœuds terminaux.
Les nœuds positifs effectuent le test correspondant, et propagent le token si le test réussit. Les nœuds négatifs suivent un algorithme plus compliqué (décrit en détail dans [Charbonnel 90]). Lorsqu'un token arrive à un nœud terminal, cela signifie que le jeu d'instanciation rend la règle correspondante déclenchable. Le token, ainsi que le nœud terminal, forment alors une instance de règle déclenchable (`OpusFireableRule`), qui est ajoutée au conflict set de la base de règles.

Le cycle d'inférence consiste alors à :

```
Envoyer les instances concernées dans le réseau Rete
Tant que le conflict set n'est pas vide
    choisir une règle dans le conflict set et l'appliquer
Terminer
```

Le cycle d'inférence étant propre à une base de règles, nous l'implémentons au niveau d' `OpusRuleSet` comme méthode de métaclasse.

III.3.6.1. Un interprète délocalisé

Comme nous l'avons signalé plus haut, notre implémentation d'Opus supprime la notion d'interprète comme objet à part entière et la remplace par une utilisation "plus objet" des bases de règles : les bases de règles sont responsables de leur propre évaluation, ce qui permettra de moduler celle-ci de manière locale, simplement en utilisant l'héritage.
Le lancement d'une évaluation se fait en envoyant le message `execute` à la base de règles concernée. Par exemple, pour lancer l'exécution de la base de règles `SingeEtBananes`, on utilisera la transmission :

```
SingeEtBananes execute
```

La méthode `execute` étant implémentée comme :

```
execute
  ^self sendInstances; sature; return
```

avec :

```
sendInstances
"envoie toutes les instances concernées dans mon reseau"
  self dispensers do: [:d| d sendInstancesInto: name].

sature
  [conflictSet isEmpty] whileFalse: [self applyRule]

applyRule
  self conflictSet applyRule

return
"rien par défaut"
```

La décomposition du processus d'évaluation d'une base de règles est reprise en détail au chapitre suivant. Notons simplement ici que l'héritage nous permettra de redéfinir localement pour chaque base de règles certaines méthodes participant à l'évaluation.

III.3.6.2. Propagation des tokens dans le réseau

La propagation des tokens dans le réseau est gérée au niveau des nœuds Rete eux-mêmes. Les méthodes les plus importantes réalisent la propagation. Il y a deux cas principaux :

- L'arrivée d'un objet (envoyé par sa classe par les méthodes `goFor:`, `modifiedFor:` ou `removeFor:`) filtrant une des variables libres de la prémisse. C'est la méthode `acceptNewObject:`
- L'arrivée d'un token provenant des prémisses précédentes (et ayant donc réussi les tests correspondants). C'est la méthode `acceptNewToken:`.

III.3.6.2.1. Arrivée d'un nouvel objet

Lorsqu'un nouvel objet arrive dans le nœud, celui-ci correspond à une variable libre de la prémisse associée au nœud. Un nouveau token (ici `newToken`) est alors fabriqué. L'objet arrivant est affecté à la bonne variable d'instance du token (cette variable est fournie par le dictionnaire `reverseDico`).
Le token frais est mémorisé dans la liste `newTokens`, à la clé correspondant à la classe de la variable libre correspondante. Le token subit ensuite le test, par la

méthode `performTestWithNewToken:field:`, qui se charge de faire les combinaisons nécessaires avec les autres tokens déjà présents.

```
!OpusNode methodsFor: 'propagation'!

acceptNewObject: anObject
  | newToken  |
  (reverseDico at: (anObject class)) do:
        [:k |
        newToken _ network newToken.
        newToken setField: k to: anObject.
        (newTokens at: k) add: newToken.
        self performTestWithNewToken: newToken field: k]
```

Quand un objet est enlevé du réseau, il faut l'enlever des mémoires de droite et de gauche :

```
!OpusNode methodsFor: 'propagation'!

removeObject: anObject
(reverseDico at: (anObject class)) do:
  [:k | (newTokens at: k ) removeAllSuchThat: [:t | t contains:
anObject]].
self removeObjectFromLeftMemory: anObject.

removeObjectFromLeftMemory: anObject
leftMemory removeAllSuchThat: [:t| t contains: anObject].
followingNode removeObjectFromLeftMemory: anObject
```

III.3.6.2.2. Arrivée d'un nouveau token

Lorsqu'un token arrive des nœuds précédents, il faut le mémoriser dans la mémoire de gauche, puis fabriquer toutes les combinaisons avec les objets en attente dans `newTokens` (par la méthode `computeTokensAt:with:`). Les tokens une fois créés sont prêts pour passer le test (méthode `performTest:`).

```
!OpusNode methodsFor: 'propagation'!

acceptNewToken: aToken
 | f newList oldList keys |
 leftMemory add: aToken.
 keys _ inputDispensers keys asSortedCollection.
 newTokens do: [ :each | each isEmpty ifTrue: [^nil] ].
 f _ keys first.
 newList _ self computeTokensAt: f with: aToken.
 keys remove: f.
 keys do: [:k |
       oldList _ newList.
       newList _ OrderedCollection new.
       oldList do: [:oT |
             newList addAll: (self computeTokensAt: k with: oT) ]].
 newList do: [:t | self performTest: t].
 ^ newList
```

La méthode pour fabriquer toutes les combinaisons d'objets avec le token `aToken` et les tokens mémorisés à la clé `k` dans la mémoire de droite, consiste à calculer une

liste de copies du nouveau token en leur affectant successivement toutes les valeurs des tokens de la mémoire de droite pour la variable d'instance `k`. C'est plus difficile à dire qu'à faire :

```
computeTokensAt: k with: aToken
  ^(newTokens at: k) collect:
        [:t |
        (network newTokenCopyOf: aToken)
              setField: k to: (t field: k)]
```

Grâce à la notion de classe dynamique, on effectue le test d'un token comme un simple envoi de `perform:` (méthode `evalPremisseWith:`). Si le test réussit, on propage le token au nœud suivant :

```
performTest: aToken
(self evalPremisseWith: aToken)
  ifTrue:
        [followingNode acceptNewToken: aToken]
```

La méthode de test d'une prémisse se réduit enfin à l'envoi de message dont le sélecteur est pointé par le nœud (c'est là tout la finesse de la manœuvre !). On s'attend évidemment à ce que le résultat de cette évaluation rende un booléen :

```
evalPremisseWith: aToken
"doit rendre un booléen !"
  ^aToken perform: selectorToCall
```

Un nouveau token étant créé, il faut faire toutes les combinaisons avec les tokens en attente (`leftMemory`), et éventuellement les autres objets filtrés par la prémisse (`newTokens`) :

```
performTestWithNewToken: aToken field: f
 | newList oldList keys n |
 newList _ OrderedCollection new.
 precedingNode isNil
       ifTrue: [newList add: aToken]
       ifFalse:
             [leftMemory isEmpty ifTrue: [^nil].
             newList _ leftMemory collect:
                   [:t |
                   n _ network newTokenCopyOf: t.
                   n setField: f to: (aToken field: f)].
 keys _ inputDispensers keys.
 keys remove: f ifAbsent: [^self error: 'Erreur dans la propagation'].
 keys do: [:k |
       oldList _ newList.
       newList _ OrderedCollection new.
       oldList do: [:oT | newList addAll:
             (self computeTokensAt: k with: oT) ]].
 newList do: [:t | self performTest: t]
```

Nous présentons dans les paragraphes suivants la description des nœuds Rete les plus courants. Grâce à la décomposition en méthodes élémentaires, la spécialisation

des nœuds Rete demandera un minimum de code, puisque seules les méthodes sensibles seront redéfinies.

III.3.6.2.3. Nœuds sans variable libre

Certaines prémisses peuvent ne pas avoir de variable libre. Ceci est spécifique à Opus, et n'aurait aucun sens en OPS5, puisque *chaque prémisse OPS5 introduit un nouvel objet*. Dans Opus, une prémisse peut très bien effectuer des tests sur des objets qui sont liés dans des prémisses précédentes.
Dans ce cas, la propagation est réduite au strict minimum. Afin d'éviter des tests coûteux, nous avons défini une classe particulière pour ces nœuds (`OpusNodeNoDisp`), qui redéfinit les méthodes de propagation de manière à tenir compte de l'absence de variable :

```
OpusNode subclass: #OpusNodeNoDisp
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'OPUS-network'
```

En particulier la méthode `acceptNewToken:` ne vas pas créer de combinaisons et va faire le test directement :

```
acceptNewToken: aToken
"Pour recuperer le node en question"
self performTest: aToken
```

Le retrait d'un objet ne va pas, lui non plus, provoquer de calcul :

```
removeObjectFromLeftMemory: x
followingNode removeObjectFromLeftMemory: x
```

III.3.6.2.4. Nœuds négatifs

Les nœuds négatifs permettent de tester l'absence d'objets vérifiant une certaine condition (absence bien sûr au moment du test : le système ne gère pas la non-monotonie). C'est la négation par l'absence, à ne pas confondre avec la négation Smalltalk, réalisée avec le message Smalltalk `not`. La syntaxe de ces prémisses est particulière :

NOT { <Declaration de variables> | texte Smalltalk }.

Un exemple significatif de cet opérateur est le tri d'élèves suivant leurs notes, en une seule règle qui s'énonce ainsi :
"*Si x a une moins bonne note que y, et qu'il n'existe aucun élève z ayant une note entre celle de x et celle de y, alors x est classé après y*".

Cette règle s'écrit ici :

```
suit
   | Eleve x y|
   x note < y note.
   NOT |Eleve z | (z ~~x & (z~~y)) and: [z note > x note and: [z note > y note]].
actions
   x place: y place + 1.
   x modified
```

Pour une description détaillée de l'implémentation des nœuds négatifs, et une discussion sur leur efficacité, se reporter à [Charbonnel 90].

III.3.6.2.5. Nœuds terminaux

Lorsqu'un token arrive dans un nœud terminal, celui-ci est ajouté au conflict set. La définition des nœuds terminaux se fait très simplement par héritage de `OpusNode`, et par la redéfinition de quelques méthodes de propagation :

```
OpusNode subclass: #OpusNodeFinal
  instanceVariableNames: ''
  classVariableNames: ''
```

L'arrivée d'un nouveau token dans ce nœud terminal a pour effet de l'envoyer au conflict set, accompagné du nœud en question, grâce à la méthode `newRule:token:` implémentée dans `OpusConflictSet` (voir chapitre suivant) :

```
acceptNewToken: aToken
self conflictSet newRule: self token: aToken
```

Le retrait d'un objet ne provoque rien puisque les propagations sont toutes terminées :

```
removeObjectFromLeftMemory: x
  ^self
```

Le conflict set est connu du nœud via le réseau :

```
conflictSet
^network conflictSet
```

III.3.6.3. Le triplet `modified/go/remove`

La modification d'un objet doit être signalée au système, par l'envoi du message `modified` à l'objet[22]. De même, un objet pouvant être créé en partie action d'une règle, il faut signaler au système sa prise en compte. Cela se fait en lui envoyant le message `go`. Le message `remove`, lui, permettra de supprimer l'objet du réseau.
Ces trois messages doivent être compris par tous les objets Smalltalk, puisqu'il n'y a pas de restriction sur les classes filtrées par les règles. Ces messages sont donc implémentés au niveau de la classe racine `Object`, et prennent tous la base de règles en argument.
Ainsi, les messages `modified`, `go` et `remove` ne sont pas véritablement des messages Smalltalk. Pour être correctement interprétés, ces messages, lorsqu'ils sont rencontrés en partie action par l'analyseur Opus, sont expansés (à l'instar des macros écrasantes en Lisp) par leur équivalent avec passage d'argument :

`go` est expansé en `goFor: self ruleBase`,
`modified` en `modifiedFor: self ruleBase`,
`remove` en : `removeFor: self ruleBase`.

où `self` est le token ayant rendu la règle déclenchable. La méthode `ruleBase` est donc implémentée dans la classe `OpusToken` et rend la base de règles correspondant à la classe du token :

```
!OpusToken methodsFor: 'rule base access'!
ruleBase
        ^self class ruleBase
```

Ces trois méthodes s'écrivent dans la classe `Object`, en renvoyant la tâche à la base de règles.
Pour envoyer un objet dans la base de règles `aRuleBase` :

```
goFor: aRuleBase
  aRuleBase acceptObject: self
```

Pour modifier un objet :

```
modifiedFor: aRuleBase
  aRuleBase modifyObject: self
```

Pour enlever un objet :

```
removeFor: aRuleBase
  aRuleBase removeObject: self
```

[22] Il est paradoxal de signaler *au système* la modification d'un objet en envoyant un message à *l'objet lui-même* (et non au système qui n'a d'ailleurs pas d'existence, à proprement parler). C'est un effet du caractère extrêmement délocalisé de notre type de programmation.

Ces méthodes retransmettant les messages à la base de règles, les messages correspondant sont donc implémentés au niveau d'`OpusRuleSet`, et vont utiliser les messages d'accès aux listes de nœuds Rete (Cf III.3.5.3.2.).

Ces messages sont les suivants (voir plus haut pour la gestion des nœuds Rete, et la méthode `nodesFor:`). Notons ici que le `modified` n'est rien de plus qu'un `remove` suivi d'un `go` comme dans tous les sytèmes Rete de base :

```
!OpusRuleSet class methodsFor: 'rete propagation'!

acceptObject: x
  (self nodesFor: x class)
        do: [:n | n acceptNewObject: x]

removeObject: x
  (self nodesFor: x class)
        do: [:n | n removeObject: x].
  conflictSet removeObject: x

acceptModifiedObject: x
  (self nodesFor: x class)
        do: [:n | n acceptNewObjectModified: x]

modifyObject: x
  self removeObject: x; acceptModifiedObject: x
```

III.3.6.4. Broadcast du `modified`

On définit sur notre lancée un "modified broadcast" pour les collections, `areModified`, permettant de signaler toute une collection d'objets comme modifiés d'un seul coup. Ce message est lui aussi expansé en `areModifiedFor:` pour pouvoir prendre la base de règles en argument. Le message `areModifiedFor:` est implémenté trivialement dans la classe `Collection` pour effectuer un broadcast :

```
!Collection methodsFor: 'broadcast'!

areModifiedFor: aRuleBase
  self do: [:x | x modifiedFor: aRuleBase]
```

Il sera utilisé en particulier dans l'exemple des réseaux de Petri (Chapitre V.1.2.8). On trouvera en annexe (XI.2) un récapitulatif des pseudo-méthodes expansées par l'analyseur Opus.

III.3.6.5. Mécanique du `modified`

Revenons un instant sur le `modified`. Envoyer le message `modified` à un objet a pour conséquence le re-essai[23] des prémisses dans lesquelles l'objet apparaît comme variable libre. L'utilisation de cette méta-action joue un rôle primordial dans

[23] On pourrait dire plus prosaïquement le re-titillement.

l'écriture de règles Opus. Nous reviendrons sur ce point au chapitre suivant, mais érigeons d'abord en principe l'*action du modified* :

Principe de l'action du modified
Déclarer un objet comme `modified` aura pour conséquence le re-essai de toutes les prémisses de toutes les règles de la base de règles dans lesquelles cet objet apparaît comme variable libre d'ordre un.

III.3.7. Le conflict set

III.3.7.1. Les règles déclenchables

Le conflict set va permettre de tenir à jour la liste des règles déclenchables. Les règles étant déclenchées une fois la propagation terminée, il faut aussi garder le token qui a traversé le réseau. Ce couple {nœud terminal/token} est représenté par une instance de `FireableRule`, qui implémentera en particulier la méthode de déclenchement (évaluation de la partie action de la règle) :

```
Object subclass: #OpusFireableRule
  instanceVariableNames: 'node token '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'OPUS-kernel'
```

La méthode de déclenchement s'écrit alors en envoyant le message `perform:` au token :

```
!OpusConflictSet methodsFor: 'rule firing'!
apply
  ^token perform: node selectorToCall
```

III.3.7.2. Parler des règles déclenchables

Un certain nombre de messages d'accès au contenu du conflict set sont alors définis. Ces méthodes constituent véritablement un vocabulaire (ou plutôt un micro-langage) qui servira à parler des règles déclenchables.

Pour tester si la règle parle d'un certain objet, on renvoie la question au token qui possède toutes les informations nécessaires (en l'occurrence ses propres variables d'instance) :

```
parleDe: anObject
  ^token parleDe: unObject
```

```
!OpusConflictSet methodsFor: 'rule access'!

regleAyantLePlusRecent: aClass
  ^self firstWithSortBlock:
        [:a :b | (a objetDeClasse: aClass) timeTag >
              (b objetDeClasse: aClass) timeTag]

regleAyantLePlusRecentObjet
  ^self firstWithSortBlock: [:a :b | a highestTimeTag > b
highestTimeTag]

regleParlantDe: anObject
  ^rules detect: [:as | as parleDe: anObject]
        ifNone: [nil]

reglesDeProtocole: p
  ^rules select: [:r | r protocole = p]

first
  ^rules first

firstWithSortBlock: unBlock
  ^(rules asSortedCollection: unBlock) first
```

Nous verrons au Chapitre VI.4.5.2 l'explication et la justification de la méthode `regleAyantLePlusRecentObjet`, liée à la notion de fraîcheur, introduite plus tard.

III.3.7.3. Définition du conflict set

Le conflict set est une instance de la classe `OpusConflictSet` définie comme suit:

```
Object subclass: #OpusConflictSet
 instanceVariableNames: 'rules'
 classVariableNames: ''
 poolDictionaries: ''
```

où `rules` est une liste de règles déclenchables (instances d' `OpusFireableRule`).

Quand une règle devient déclenchable la méthode suivante est envoyée au conflict set afin de mettre à jour sa liste de règles déclenchables :

```
newRule: aNode token: aToken
rules addFirst: (OpusFireableRule node: aNode token: aToken).
(self changed: #rule)
```

Notons que par défaut la règle déclenchable est ajoutée en début de liste (méthode `addFirst:`). Si, comme c'est le cas ici, la première règle est systématiquement choisie, cela induira un parcours en profondeur du graphe d'état. De même, choisir la dernière règle aurait pour conséquence un parcours en largeur.

Quand un objet est retiré du réseau (à la suite d'un `remove` ou d'un `modified`), il est aussi retiré du conflict set. La méthode est la suivante :

```
removeObject: x
 rules removeAllSuchThat:
       [:uneRegleDeclenchable | (uneRegleDeclenchable contains: x)]
```

Le conflict set a aussi à charge l'exécution et le choix de la règle, via la méthode `declencheDefault`. Par défaut, la première règle est choisie et déclenchée comme suit :

```
!OpusConflictSet methodsFor: 'declenchement'!

declencheDefault
  self declenche: rules first

declenche: uneRegle
  rules remove: uneRegle.
  self changed: #rule.
  uneRegle apply
```

Nous reparlerons du conflict set au chapitre VI (sur le contrôle) et au chapitre IV.1.7.4, à propos de l'interprétation de la notion d'héritage de bases de règles.

III.4. Points sensibles soulevés par les auteurs

Nous présentons ici les remarques conclusives des auteurs quant à l'utilisation effective du système de base. Les trois points qui suivent ne seront pas repris par la suite, nous présentons simplement leurs énoncés. Ce sont donc toujours des problèmes ouverts.

III.4.1. Le `modified`

Le souci de généralité maximum dans l'expression des règles a pour conséquence que les objets testés par les prémisses peuvent être tout objet Smalltalk; à savoir des objets de la mémoire de travail; des objets référencés par ces derniers, ou des objets accessibles par variable globale (comme les classes).
Le fait que toute expression puisse être écrite en partie action, et que donc tout objet Smalltalk puisse être modifié de façon non contrôlable peut entraîner une incohérence entre l'état de mémorisation des tokens sur les nœuds Rete et l'état réel des objets ayant été filtrés.
Pour pallier à cet inconvénient, les auteurs proposent de signaler au système Opus les objets ayant pu être changés à la suite d'une partie action d'une règle. C'est le fameux problème du `modified`.
En OPS5, le `modified` ne pose pas de problème dans la mesure où toutes les modifications des objets se font par l'intermédiaire d'un `modified` (ou, ce qui revient au même, d'un `remove` et d'un `add`), qui est une des seules primitives du langage OPS5. Le problème consistant à signaler au système une modification "faite

de l'extérieur" ne se pose donc pas. En revanche, en Opus, le `modified` apparaît réellement comme une redondance dans le codage.
De plus, il brise subtilement le principe de fermeture (l'idée que l'auteur des règles n'a pas besoin de connaître la manière dont sont implémentées les méthodes pour écrire effectivement ses règles). L'auteur d'une règle doit savoir quels objets doivent être reconsidérés après l'application de la partie `action` de la règle.
La solution adoptée est considérée comme partielle par les auteurs, qui considèrent que le programmeur a la liberté de signaler ou non au système les objets qui ont été modifiés. Plus précisément, toute modification de l'objet n'a pas à être signalée, mais seulement les modifications qui remettent en cause l'état de mémorisation des nœuds.
Le programmeur a donc la lourde responsabilité de *décider* si un objet a été modifié et si, en outre, cette modification a changé l'état d'instanciation des prémisses. Nous verrons au chapitre V.3 que ceci est à la fois une contrainte et une souplesse du langage, dans la mesure où cela donne justement la possibilité de contrôler, de manière indirecte, la mécanique d'instanciation des nœuds Rete.

III.4.2. Problème du Undo

La complexité de l'environnement Smalltalk rend les opérations de sauvegarde d'environnement difficiles. L'opération dite de `Undo` pose ici particulièrement problème. La structure de réseau de l'environnement Smalltalk rend les problèmes de copie d'objet compliqués, et nécessite des algorithmes non triviaux adaptés (voir, pour un algorithme relativement simple [Vegdahl 86], sa mise en œuvre à l'aide de métaclasses [Pachet 89] et la solution plus générale mais aussi plus complexe adoptée par ParcPlace (le BOSS) dans [ParcPlace 88]). Aucune solution n'est proposée pour permettre effectivement de revenir en arrière sur le déclenchement d'une règle par exemple.
Nous avons pour notre part implémenté un mécanisme de retour arrière permettant d'ajouter un mécanisme de backtrack à NéOpus (Cf Chapitre VII.6.4). Ce mécanisme n'étant pas générique (i.e. il ne s'applique pas à tous les objets Smalltalk), nous ne l'avons pas intégré au système Opus de base.

III.4.3. Debugging

Smalltalk-80 possède un debugger extrêmement puissant, qui permet d'inspecter le flux d'exécution (la pile des messages), de la stopper, et même de recompiler à la volée une méthode et de relancer l'exécution [ParcPlace 88, Lalonde&Pugh 90].
Adapter le debugger Smalltalk à Opus signifierait définir préalablement ce qu'est un *flux d'exécution* Opus : ce n'est certainement pas la pile Smalltalk, qui contient simplement les messages implémentant les diverses actions Opus, ainsi que les messages Smalltalk apparaissant dans les règles. Cela impliquerait de définir une machine virtuelle Opus. En attendant, le debugger Smalltalk est disponible, mais

donne évidemment une vision très implémentatoire du processus de debugging de bases de règles.

III.5. Conclusion

Nous avons présenté dans ce chapitre notre implémentation d'Opus, réalisée à partir de notre lecture de la description des auteurs, et aussi de notre propre expérience du système.
Comme on peut le remarquer en parcourant le code, l'essentiel de l'implémentation d'Opus repose sur trois méthodes (mis à part les méthodes du parser) :

- La méthode de mise à jour du réseau Rete à la compilation d'une règle (`addNodesForRule:` dans la classe `OpusNetwork`),
- Les deux méthodes principales de propagation des tokens dans ce réseau (`acceptNewObject:` et `acceptNewToken:` dans `OpusNode` et ses sous-classes).

Le reste, d'une certaine manière n'est que fioriture, méthodes d'accès plus ou moins triviales et ne présente pas de difficulté particulière, grâce aux hautes qualités d'organisation de Smalltalk.

V. Praxis

Avant-propos

Dans cette partie, nous présentons notre pratique du système NéOpus, tel qu'il a été décrit dans le chapitre précédent. Nous explorons les possibilités et les faiblesses du mode de raisonnement induit par notre système. En particulier, nous classons les diverses utilisations de NéOpus en trois catégories, du plus simple au plus complexe. Nous présentons aussi une application moins triviale de NéOpus à un système de raisonnement géométrique.
Nous tirons de ces expériences un *principe d'utilisation* (le principe du `modified`) et soulignons deux limites principales du système, liées au manque de représentation assertionnelle, et au manque de contrôle sur le raisonnement.

V.1. Utilisations de NéOpus par l'exemple

Nous décrivons ici l'utilisation de NéOpus par degrés croissants de complexité, en illustrant nos propos par de petits exemples.

V.1.1. Première utilisation : comme pattern-matcher

L'environnement Smalltalk peut être vu comme une vaste base de données relationnelle [Codd 70]. Les objets Smalltalk sont alors vus comme des entités de la base de données. Les relations entre entités sont représentées par les liens de variables d'instances entre objets.
On peut alors utiliser NéOpus comme système de requêtes sur cette base de données. L'intérêt de cette approche est de bénéficier ainsi d'un système de requêtes ayant un pouvoir d'expression maximal. En effet, les langages d'interrogation relationnels classiques (comme SQL [Oracle 88] ou dBase) permettent d'effectuer des requêtes logiques portant sur des contraintes entre valeurs d'attributs [Bodin&Pichat 90]. NéOpus nous permet d'aller plus loin et de chercher les objets satisfaisant toute contrainte exprimable sous forme d'expression Smalltalk. Le réseau Rete est alors utilisé comme mécanisme de discrimination/jointure, classique en bases de données [Copeland-Maier 84], [Maier-Stein 90].
Les règles que nous écrivons dans ce chapitre ne comportent aucune méta-action (`go`, `remove` ou `modified`).

V.1.1.1. Exemple sur une mini base de données

Supposant les classes `Etudiant` (`pere mere sexe age coursSuivis couleurCheveux nom`) (éventuellement sous classe de la classe `Personne` déjà nommée) et `Cours` (`intitule lieu` ..), alors on peut facilement transcrire toute requête sur une base de données comportant de tels objets.
Par exemple, trouver toutes les étudiantes de moins de 25 ans, blondes, nobles, suivant un cours de Smalltalk avancé et de cuisine tropicale, et de père fortuné s'écrira simplement sous la forme d'une règle NéOpus du type de `idealZero`.

```
idealZero
| Etudiante e. Cours c1 c2|
e age < 25.
e couleurCheveux = #blond.
e pere profession salaire > 30000.
e nomFamille commencePar: 'de '.
e suitCours: c1.   c1 intitule = 'Smalltalk avance'.
e suitCours: c2.   c2 intitule = 'Cuisine tropicale'.
actions
Transcript show: 'Une instance d'idéal féminin : ',e nom;cr.
```

Une règle de requête sur une mini base de données

Utilisation

Une telle base de règles pourra être utilisée de la manière suivante, en créant une méthode de lancement dans la métaclasse `IdealRules` class, qui créera les instances correspondantes :

```
exemple
      |e1 e2 e3 e4 e5 e6 e7|
"des etudiants"
e1 <- Etudiante new cheveux: #brun; nom: #Lydia; nomFamille: 'DuVilloge'.
e2 <- Etudiante new cheveux: #auburn; nom: #Colette; nomFamille:
'MaVallée'.
e3 <- Etudiante new cheveux: #roux; nom: #Berthe; nomFamille: 'MaVallée'.
e4 <- Etudiante new cheveux: #blond; nom: #Berthe; nomFamille: 'de la
RocheFoucault'.
"des cours"
c1 <- Cours new intitule: 'Cuisine tropicale'; lieu: 'salle 111'; ...
c2 <- Cours new intitule: 'Smalltalk debutant'; lieu: 'salle 09; ...
"des personnes"
p1 <- Personne new nom: #Paul; profession: #professeur; salaire: 30000; ...
"des liens entre les objets"
e4 pere: p1. e4 suitCours: c1; suitCours: c2 ...
"lancement de la base de règles sur tous les objets existant"
self executeWithAllObjects
```

Evidemment, l'intérêt d'un tel exemple vient du fait que la notion de cours suivi est représentée par une *classe* à part entière, et non par un symbole.

V.1.1.2. Requêtes sur l'environnement Smalltalk lui-même

On peut aussi utiliser NéOpus comme système de requêtes sur l'environnement Smalltalk lui-même, considéré comme une base de données à part entière.
On peut alors simuler et étendre les mécanismes de références (Cf. notamment les exemples du "System Workspace") existant en Smalltalk et permettant de faire certaines requêtes sur les objets préexistant du système. La mécanique NéOpus nous permet alors de "browser" l'environnement par des spécifications très succinctes.

Exemple

Voici une base de règles NéOpus effectuant une requête sur l'environnement Smalltalk. Il s'agit de compter les instances existantes de fenêtres actives, et de les faire flasher.
Les fenêtres en Smalltalk sont représentées par la classe `View` [Goldberg&Robson 83, Lalonde&Pugh90]. Mais cette classe a elle-même plusieurs sous-classes (44 en version 2.5), représentant diverses variations de la notion de fenêtre (comme les classes `StandardSystemView`, `BrowserView`, `FileListView`, `FileListView` ...). Pour quérir toutes les fenêtres actives de toutes ces sous-classes, on utilise le

typage naturel, et on écrit une prémisse testant simplement la non-nilité de son contrôleur (pour éviter les instances de fenêtres fantômes non encore récupérées par le garbage-collecteur).
On utilise de plus le principe d'existence (Cf. Chap IV.1.3) en s'épargnant le test `notNil` :

```
requeteFenetres
| View v. Global CompteurFenetres|
v controller.          "pour éviter de filtrer les fenêtres mortes mais non enterrées"
actions
v flash.
CompteurFenetres incr.
```

On déclare pour ce faire un objet nommé `CompteurFenetres` dans la base de règles :

```
OpusRuleSet subbase: #Requetes
  globalObjects: 'CompteurFenetres'
```

On peut alors lancer la base de règles par une méthode comme la suivante, en initialisant le Compteur à une instance de `Compteur` (et non pas à 0 !!) :

```
!Requetes class methodsFor: 'exemple'!

requetes
  CompteurFenetres := Compteur new.
  self executeWithAllObjects.
  Transcript show: CompteurFenetres valeur printString.
```

Ce genre de requêtes peut être multiplié à l'infini, sur toutes les classes du système.

V.1.1.2.1. Discussion

Remarquons que l'utilisation de NéOpus comme requêteur est paradoxale, dans le sens où une requête est, au sens de NéOpus, *une règle qui ne fait rien* (dont la partie action est vide). Cette utilisation s'accompagne donc le plus souvent d'utilisation d'objets nommés, comme les compteurs, qui permettent de *récupérer le résultat des inférences.*

V.1.1.3. Utilisation comme combinateur

Le mécanisme de saturation permet de coder simplement des calculs de combinaisons *respectant* certaines contraintes. Mais il faut noter qu'alors *le réseau Rete n'apporte aucune optimisation* par rapport à l'algorithme de recherche naïf. En effet, toutes les combinaisons sont testées, puisqu'aucune *propagation* de contrainte n'est faite.

Cependant, une écriture soignée des prémisses permet tout de même de "limiter les dégâts", en assurant "à la main" une propagation des contraintes.

V.1.1.3.1. Exemple : les n reines

Ce problème célèbre [Laurière 87] consiste à trouver une configuration de n reines sur un échiquier de n*n cases telle que les reines ne soient pas en prise deux à deux.

On définit alors les objets du domaine : `Case`, définie par un point, `Echiquier`, définie comme un ensemble de cases, et `Solution`, définie comme un ensemble de cases sur lesquelles sont les reines[24], toutes sous-classes d'`Object` :

```
Case instanceVariableNames:                 'point'
Echiquier instanceVariableNames:            'cases'
SolutionReines instanceVariableNames:       'cases'
```

Une méthode de la classe `Case` permet de tester qu'une case est en prise avec une autre :

```
!Case methodsFor: 'testing'!

enPriseAvec: uneCase
  point x = uneCase point x ifTrue: [^true].
  point y = uneCase point y ifTrue: [^true].
  v <- point - uneCase point.
  ^v x abs = v y abs

nonEnPriseAvec: uneCase
"la contraposee"
  ^(self enPriseAvec: uneCase) not
```

On crée alors toutes les cases de l'échiquier par une initialisation appropriée de l'échiquier :

```
!Echiquier methodsFor: 'initialisation'!

initialize: taille
  cases <- OrderedCollection new.
  1 to: taille do: [:ligne |
        1 to: taille do: [:colonne |
              cases add: (Case new point: (colonne @ ligne))]]
```

La classe `SolutionReines` sert juste d' "entrepôt" et ne comporte qu'une méthode de création (`avec:et:et:et:`) et d'affichage (`printOn:`).

V.1.1.3.2. Première version

[24] Notons qu'il y n'y a alors pas de différence structurelle entre un échiquier et une solution.

Si l'on fixe le nombre de reines on peut en une seule règle spécifier les contraintes du problème. Pour chaque n-uplet de cases, on teste que ces cases ne sont pas en prise deux à deux. On crée alors un objet `Solution` comportant les n cases.
Par exemple, si n = 4, cela donne :

```
quatreReines
   | Case c1 c2 c3 c4]
   c1 nonEnPriseAvec: c2. c1 nonEnPriseAvec: c3. c1 nonEnPriseAvec: c4.
   c2 nonEnPriseAvec: c3. c2 nonEnPriseAvec: c4.
   c3 nonEnPriseAvec: c4.
actions
   Transcript show: (SolutionReines avec: c1 et: c2 et: c3 et: c4) printString.
```

V.1.1.3.3. Discussion

Ordre des prémisses
l'ordre d'écriture des prémisses influe énormément sur la vitesse d'exécution.
Dans la première solution par exemple, la troisième prémisse est très coûteuse:

```
c1 nonEnPriseAvec: c4.
```

En effet, on a filtré dans les deux prémisses précédente deux cases (référencées par `c1` et `c3`) qui sont peut-être en prise. Il vaudrait mieux alors s'assurer qu'elles ne le sont pas, *avant de filtrer un nouvel objet*. Un ordre différent des prémisses permet de prendre ce critère en compte.

Efficacité
La règle reste très peu efficace (toutes les combinaisons sont testées), surtout comparée à un système sachant propager les contraintes (comme le système Alice [Laurière 78] par exemple).

Consistance
Il y a beaucoup trop de solutions.
Pour n = 4, soit 16 cases, on génère 16 puissance 4 = 65536 combinaisons. Beaucoup de ces combinaisons sont identiques aux permutations de reines près. Ceci fait ressortir le besoin de filtrer d'avantage les objets : Il faudrait pouvoir restreindre les cases au moment de leur déclaration.

Manque de généralité
Le nombre de reines doit être fixé une fois pour toute pour une règle donnée. Il faut donc écrire autant de règles que de valeurs de n désirées. NéOpus n'est pas capable de prendre ce genre de critère de généralité en compte. Ceci est lié à la notion d'ordre un, qui ne permet de dénoter qu'un seul objet à la fois, par définition. Il faudrait ici un passage à un ordre supérieur, permettant de filtrer un nombre variable d'objets.
En somme, quelque chose comme :

```
nReines
   |        Case [i] |
   Pour tout i, j tels que :
            i <> j.
            case [i] nonEnPriseAvec: c [j].
   actions
   (SolutionReines avec: Case [i])...
```

Mais nous ne savons pas, en l'état actuel des connaissances "passer" à cet ordre supérieur.

V.1.1.3.4. Deuxième version

Voici une deuxième version qui permet de prendre en compte les trois premiers points, en rajoutant plus de contraintes, en particulier le fait qu'il n'y a qu'une reine par ligne, et en changeant l'ordre des test pour minimiser le nombre d'échecs. De plus, on rajoute quatre prémisses qui permettent de prendre en compte la contrainte qu'il n'y a qu'une reine par colonne et de ne garder que les n-uplets qui satisfont à cette contrainte.
Ceci va diminuer notablement le temps de calcul (0.5 secondes sur Macintosh FX, soit 10 fois plus rapide que la version précédente) et le nombre de solutions (2 solutions pour 4 cases) :

```
quatreReines
   | Case c1 c2 c3 c4 |
   c1 point x = 1. c2 point x = 2. c3 point x = 3. c4 point x = 4.

   c1 nonEnPriseAvec: c2. c1 nonEnPriseAvec: c3.
   c1 nonEnPriseAvec: c4. c2 nonEnPriseAvec: c3.
   c2 nonEnPriseAvec: c4. c3 nonEnPriseAvec: c4.
actions
   Transcript show: (SolutionReines avec: c1 et: c2 et: c3 et: c4) printString.
```

V.1.2. Seconde utilisation : avec méta-actions, sans stratégie de contrôle

Nous allons maintenant utiliser les trois messages singuliers `go`, `modified` et `remove` et étudier leur effets. Ces trois messages peuvent être considérés comme des méta-actions, puisque leur application va avoir un effet direct sur le mécanisme d'évaluation de la base de règles.
Plus précisment, l'utilisation des trois méta-actions dans les règles va permettre de :

- `modified` : prendre en compte les *changements* des objets pour déclencher des nouvelles règles,
- `go` : prendre en compte des objets *nouveaux*, créés en partie action des règles,
- `remove` : supprimer des objets (du réseau Rete)[25].

V.1.2.1. Fonction récursive : le calcul de Fibonacci

Une première utilisation du `modified` est de permettre l'écriture sous forme de règles de *mécanismes récursifs*. Le `modified` joue alors le rôle de remontée de pile, implicite dans l'écriture récursive. L'exemple le plus populaire d'une telle fonction est celui du calcul de la série de Fibonacci. L'idée [Voyer 87, 89a] est de calculer la valeur de la suite de Fibonacci en aplatissant complètement le mécanisme de récursion. Chaque demande de calcul de Fibonacci est représenté par une instance de la classe `Fibo` définie comme suit :

```
Object subclass: #Fibo
  instanceVariableNames: 'arg val fils1 fils2'
```

Pour chaque calcul de Fibonacci, on crée deux fils, qui représentent les calculs engendrés habituellement par les appels récursifs. Trois règles permettent alors de représenter le calcul : une règle d'arrêt (`fibFin`), une règle de génération de fils (`creerFils`) et une règle de retour, pour calculer la valeur quand celles des deux fils le sont (`retour`) :

fibFin
" si arg est >= 2 alors val est 1"
|Fibo f|
f arg >= 2.
f val doesNotExist.
actions
f val: 1.
f modified

creerFils
"on cree les deux fils pour f"
|Fibo f|
f arg > 2.
f val doesNotExist.
f fils1 doesNotExist.
actions
| f1 f2|
f 1 := Fibo new arg: f arg - 1.
f2 := Fibo new arg: f arg - 2.
f fils1 : f1; fils2: f2.
f1 go. f2 go.
f modified "inutile"

resultat
"on additionne les valeurs des deux fils"
|Fibo f f1 f2|
f arg > 2.
f val doesNotExist.
f1 <- f fils1.
f1 val exists.
f2 <- f fils2.
f2 val exists.
actions
f val: f1 val + f2 val.
f modified

Un exemple d'utilisation est le suivant : on exécute la base de règles avec un seul objet `Fibo` crée pour la circonstance (avec par exemple un `arg` à 10 et une `val` à nil),

[25] Si la notion de *création* d'objet est parfaitement fondée (via le mécanisme d'instanciation), celle de *suppression* n'a en revanche aucun sens en programmation par objet. Un objet existe tant qu'il est référencé. Quand il ne l'est plus il n'existe plus : c'est la mort par quarantaine.

et on lance la base de règles avec cet objet unique comme contexte, via la méthode `executeWithSingleObject:`:

```
!FiboRules class methodsFor: 'exemple'!

exemple
  self executeWithObject: (Fibo new arg: 10)
```

V.1.2.2. Discussion

Résultat des inférences

Le résultat de l'évaluation de cette base de règles est à chercher dans la valeur de l'attribut `arg` de l'objet créé dans la méthode exemple (Cf §V.1.4 pour les benchmarks de cette base de règles).

Contrôle

A un moment donné du cycle d'exécution, plusieurs règles pourront être déclenchables, mais le choix de la règle n'influencera pas le résultat final. Tous les chemins partant de l'état initial conduisent au même état final. L'ordre de déclenchement des règles n'a donc ici aucune influence sur le résultat[26].

Modified

Notons ici que dans la règle `creerFils`, l'objet dénoté par `f` est effectivement modifié après la création de ses deux fils. Mais sa déclaration comme `modified` est inutile. En effet, aucune autre règle de la base de règles ne peut *changer d'état d'instanciation* après une telle modification de f. Seule la règle `resultat`, qui calcule effectivement la valeur de `f` justifie une déclaration `modified`. Nous reviendrons sur le problème du `modified` plus en détail au Chapitre VI.3.2.

V.1.2.3. Variation immédiate : calcul de la factorielle

Dans le même genre d'idées, on peut écrire les règles pour le calcul de la factorielle. Ici un seul sous-calcul est généré. On définit la classe `Fac`, version simplifiée de la classe `Fibo` n'ayant qu'un fils :

```
Object subclass: #Fac
  instanceVariableNames: 'arg val fils'
```

et trois règles très similaires aux précédentes avec la génération d'un seul fils :

[26] Il en a en revanche sur l'efficacité. Un parcours en profondeur s'avère ici plus efficace qu'en largeur (ou que toute autre stratégie).

facDe1
"fac de 1 = 1"
| Fac f |
f arg = 1. f val doesNotExist.
actions
f val: 1. f modified

rFacDesAutresDepart
"on cree un fils avec n - 1"
| Fac f |
f arg > 1.
f val doesNotExist.
f fils doesNotExist.
actions
|nouveauFac|
nouveauFac := Fac new arg: (f arg - 1).
f fils: nouveauFac.
nouveauFac go.
f modified "inutile"

rFacDesAutresRetour
"on recupere la valeur du fils une fois calculee et on multiplie par n"
| Fac f f1|
f arg > 1.
f val doesNotExist.
f1 := f fils.
f1 val exists.
actions
f val: (f1 val * f arg).
f modified

Un exemple d'utilisation similaire au précédent :

```
!FacRules class methodsFor: 'exemple'!

exemple
  self executeWithSingleObject: (Fac new  arg: 10)
```

Nous allons profiter de la ressemblance entre les deux bases de règles pour généraliser et produire une base de règles générale de calcul de fonctions récursives, au §V.2.2.5. Auparavant, étudions une dernière fonction récursive un peu plus complexe :

V.1.2.4. Le problème des tours de Hanoï

Le problème des tours de Hanoï peut se décrire aussi de la même manière. En supposant la classe `HanoiPb` définie comme suit, ainsi que ses méthodes d'accès :

```
HanoiPb (n fils1 fils2 depart arrivee intermediaire termine)
```

où `n` est le nombre de plaques, `fils1` et `fils2` les deux sous-problèmes générés et `termine` un booléen indiquant si le problème a été résolu ou non (initialisé à `false`). On utilise la décomposition récursive du problème pour écrire les règles :

Déplacer n tours de `départ` à `arrivee` en utilisant `intermediaire`, c'est :

(règle `r1`) :
si n = 1, et que le problème n'est pas résolu, simplement afficher le déplacement d'une plaque de `depart` à `arrivee`.
(règle `r2`) :
si n > 1, et que le premier sous-problème n'as pas encore été créé, alors on crée un premier sous-problème (appelé fils1) ayant n - 1 plaques, de `depart` à `intermediaire` en passant par `arrivee`.

(règle `r3`) :
si le premier sous-problème est résolu, et que le deuxième n'est pas encore créé, on fait deux choses :

on déplace une plaque de `depart` à `arrivee`,

on crée le deuxième sous-problème (fils2) : un `HanoiPb` avec `n - 1` plaques de `intermediaire` à `arrivee` en passant par `depart`.

(règle `r4`) :
si le deuxième sous-problème est résolu, alors le problème (père) est résolu et signalé comme modifié.

r1	**r2**	**r3**	**r4**
"si n = 1, on deplace la plaque"	"si n > 1, on cree un sous-probleme"	"on deplace la derniere plaque, et on crée le second sous-problème"	"les deux sous-pb sont termines"
| HanoiPb p |	| HanoiPb p |	| HanoiPb p p1|	|HanoiPb p p1 p2|
p n = 1.	p n > 1.	p n > 1.	p termine not.
p termine not.	p termine not.	p1 <- p fils1.	p1 <- p fils1.
	p fils1 isNil.	p termine not.	p2 <-
actions		p fils1 termine.	p fils2.
Transcript show:	**actions**	p fils2 isNil.	p fils1 termine.
'Je deplace de ',		**actions**	p fils2 termine.
p depart,' a ',	|p1 |	|p2|	
p arrivee;cr.	p1 := HanoiPb new.	Transcript show:	**actions**
p termine: true.	p1 n: p n - 1.	'Je deplace de ',	p termine: true.
p modified	p1 depart: p depart;	p depart,' a ', p arrivee;cr.	p modified
	arrivee: p	p2 := HanoiPb new.	
	intermediaire;	p2 n: p n -1.	
	intermediaire: p	p2 depart: p	
	arrivee.	intermediaire;	
	p fils1: p1.	arrivee: p arrivee;	
	p1 go. p modified	intermediaire: p depart.	
		p fils2: p2.	
		p2 go. p modified	

L'exemple d'utilisation est alors trivial :

```
!HanoiRules class methodsFor: 'exemple'!
exemple
   self executeWithSingleObject:
      (HanoiPb new  n: 5; depart: #T1; intermediaire: #T2; arrivee: #T3)
```

V.1.2.5. Discussion

Ordonnancement dans la structure
On voit qu'ici apparait déjà un problème d'ordonnancement qui n'existait pas dans les exemples de `Fibo` et `Fac` : il faut spécifier d'une manière ou d'une autre que le

deuxième sous-problème doit être exécuté *après* le premier, ici en utilisant la variable booléenne `termine`.

Apparition de prémisses supplémentaires
Cette base de règles montre d'autre part la nécessité de tester systématiquement en partie prémisse que l'évaluation de la partie action de la règle ne rendra pas la règle à nouveau déclenchable. Ici, par exemple, omettre le test `(p termine not)` dans la règle `r2` pourrait conduire à un bouclage infini, la règle `r2` étant toujours déclenchable.
Ce test est inévitable en NéOpus, et est fortement lié à la mécanique de Rete, dans la mesure où il est impossible de connaître le résultat de l'application de la règle a priori. Dans certains systèmes (Snark, Andromac [Voyer 87]), les règles ne sont déclenchées que si elles ajoutent un fait nouveau dans la base de faits. Ici il est évidemment impossible de prévoir à l'avance un tel effet.

V.1.2.6. Généralisation : dérécursivation de fonctions

Dans les exemples précédents, la dérécursivation des fonctions (`Fibo`, `Fac`, `Hanoi`) suit toujours le même principe de réification :

- La *fonction* devient l'objet principal et est transformée en classe.
- Le résultat de la fonction et ses arguments deviennent des variables d'instances pour cette classe.
- Les appels récursifs intermédiaires sont représentés explicitement par des objets, et stockés aussi sous forme de variables d'instance "fils" au niveau du nœud père.

Ces objets/fonctions représentent donc à la fois :

- les données initiales du problème à résoudre (la fonction et ses arguments),
- son résultat une fois calculé (`val`)
- les calculs intermédiaires (les nœuds dans l'arbre du calcul).

La transformation fonction récursive / base de règles rend alors le parcours de l'arbre explicite. Dans le cas de `Fac` comme de `Fibo`, trois règles sont nécessaires pour représenter :

- la condition d'arrêt de la récursion,
- la création d'une branche de l'arbre,
- le retour au nœud appelant.

Nous allons généraliser ce procédé, en créant des classes abstraites de fonctions et des bases de règles associées. Une hiérarchie de classes représentant les différents types de fonctions récursives, et de bases de règles permet alors de transformer un certain nombre de fonctions récursives sous formes de règles en marche avant, en redéfinissant par héritage les trois méthodes décrites ci-dessous.

Une classe générique `FonctionRecursive` est donc définie, comme une classe abstraite représentant toutes les fonctions récursives, et ayant simplement une variable d'instance `val` représentant le résultat de la fonction :

```
Object subclass: #FonctionRecursive
  instanceVariableNames: 'val '
```

Une base de règles, elle aussi générale, `FonctionRecursiveRules` est alors définie, ne contenant qu'une seule règle, celle du test de fin, qui sera commune à toutes les sous-bases :

```
OpusRuleSet subbase: #FonctionRecursiveRules
  globalObjects: ''
  category: 'OPUS-rules-fonctions'!


!FonctionRecursiveRules methodsFor: 'arret'!

arret
| FonctionRecursive f|
   f val isNil.
   f testFin.
actions
   f val: f valeurFin.
   f modified! !
```

Cette base de règles filtre des objets très généraux (`FonctionRecursive`). Il nous faudra donc utiliser alors le typage naturel, pour pouvoir prendre en compte les instances des sous-classes de `FonctionRecursive`.

La règle `arret` est donc générale à deux titres :

- Généralité au sens du typage : son interprétation sera différente suivant les sous-classes de `FonctionRecursive`, qui redéfiniront les méthodes utilisées en prémisse et en partie action.
- Généralité au sens de l'héritage de bases de règles : elle sera ré-utilisée par toutes les sous-bases de `FonctionRecursiveRules`.

Nous allons maintenant utiliser nos représentations abstraites (la classe abstraite `FonctionRecursive`, et la base de règles abstraite `FonctionRecursiveRules`) en en créant des sous-classes et des sous-bases pour représenter divers types de fonctions récursives.

V.1.2.6.1. Fonctions récursives à un appel

Nous allons ici généraliser aux fonctions récursives à *un seul* appel. Dans le cas d'une fonction `Fonc`, avec un argument, `arg`, et un seul appel récursif, la définition Smalltalk la plus générale serait :

```
fonc: argument
^self testFin
  ifTrue: [self valeurFinale: argument]
  ifFalse: [self calculValeur: (self initFils: argument)]
```

On définit alors les fonctions récursives à un argument et un seul appel récursif par héritage, en rajoutant les variables d'instance `arg` et `fils` :

```
FonctionRecursive subclass: #FonctionRecursiveUnArgumentUnAppel
 instanceVariableNames: 'arg fils '
```

Puis on écrit les quatres méthodes représentant chaque calcul de la fonction récursive (en plus des méthodes d'accès standard) :

```
testFin
la fonction de test d'arrêt

valeurFinale
rend la valeur pour le cas final

initFils: unFils
initialisation des valeurs pour le fils

calculeValeur
calcul de la valeur au retour de l'appel récursif
```

Conformément à la politique Smalltalk concernant les classes abstraites, ces quatre méthodes sont définies comme rendant `self subclassResponsibility`. Elles sont en effet destinées à être redéfinies dans les sous-classes.

On définit ensuite une sous-base de `FonctionsRecursivesRules` qui va prendre en compte le cas des fonctions à un seul appel :

```
FonctionRecursiveRules subbase: #FonctionRecursiveUnAppelRules
 globalObjects: ''
```

Les deux règles baptisées `appel` et `retour` sont alors les suivantes :

```
appel
"si le testFin n'est pas vérifié, on crée un nouveau fils, et on l'initialise"
    | FonctionRecursive f|
    f val isNil.
    f testFin not.
    f fils isNil.
actions
    | newFonc |
    newFonc _ f class new.
    f fils: newFonc.
    f initFils.
    newFonc go.

retour
"le calcul de fils est terminé, on dépile et on calcule la valeur pour le pere"
    | FonctionRecursive f f1|
    f val isNil.
    f1 _ f fils.
    f fils val notNil.
actions
    f calculeValeur.
    f modified.
```

Une nouvelle version de `Fac` peut maintenant être écrite, où `Fac` est sous-classe de `FonctionRecursiveUnAppel`, et *redéfinit* les quatres méthodes utilisées par les règles :

```
FonctionRecursiveUnArgumentUnAppel        subclass:        #TestFin
  isntanceVariableNames: ''

testFin
  ^arg = 1

valeurFinale
  ^1

initFils: unFils
  unFils arg: arg - 1.

calculeValeur
  val <- arg * fils val
```

On utilise alors la base de règles `FonctionRecursiveUnArgumentUnAppel`, en initialisant son contexte avec une instance de `Fac`.
Un exemple d'utilisation est alors :

```
'Fac methodsFor 'exemple'!

exemple
  FonctionRecursiveUnArgumentUnAppel
          executeWithSingleObject: (Fac new arg: 10)
```

De même, donnons un autre exemple de fonction récursive à un appel et un argument[27] : celui du calcul de la fonction d'inversion d'une collection.
Nous définissons alors simplement la classe : `ReverseList`, comme sous classe de `FonctionRecursiveUnAppelUnArgument`.
Les méthodes redéfinies sont alors :

```
FonctionRecursiveUnArgumentUnAppel subbase: #ReverseList
  globalObjects: ''
  category: 'OPUS-exemples-fonctions'!


!ReverseList methodsFor: 'methodes'!

calculeValeur
  val _   fils val, (Array with: arg first)!

initFils
  fils arg: (arg copyFrom: 2 to: arg size)!

testFin
  ^arg size = 1!

valeurFinale
  ^arg
```

La méthode d'exemple est alors simplement :

```
!ReverseList class methodsFor: 'exemple'!

exemple
FonctionRecursiveUnAppelRules executeWithSingleObject:
   (self new arg: #(1 2 3))
```

V.1.2.6.2. Fonctions récursives à deux appels

On peut maintenant généraliser à d'autres types de fonctions récursives, plus complexes, comme les fonctions à un argument et deux appels. Dans le cas d'une fonction `Fonc`, avec un argument `arg1`, et deux appels récursifs, la définition Smalltalk serait :

```
fonc: argument
^self testFin
  ifTrue: [self valeurFinale: argument]
  ifFalse: [self calculValeur: (self initFils1: argument)
                           et:   (self initFils2: argument)]
```

On a alors la définition de la classe correspondante suivante, avec les variables d'instance `arg`, `fils1` et `fils2` :

[27] Pour exemplifier un mécanisme générique, il faut en effet au moins deux exemples !

```
FonctionRecursive subclass: #FonctionRecursiveUnArgumentDeuxAppels
    instanceVariableNames: 'arg fils1 fils2 '
```

et les quatres méthodes utilisées dans le calcul de la fonction (en plus des méthodes d'accès standard) :

```
testFin
la fonction de test d'arrêt

valeurFinale
rend la valeur pour le cas final

initFils1
initialisation des valeurs pour le fils1

initFils2
initialisation des valeurs pour le fils2

calculeValeur
calcul de la valeur au retour de l'appel récursif. Les fils ne
sont pas passés en arguments puisqu'ils sont accessibles
directement sous forme de variable d'instance
```

On définit maintenant une sous-base de la base racine `FonctionRecursiveRules`, qui va définir les deux règles nécessaires au calcul :

```
FonctionRecursiveRules subbase: #FonctionRecursiveDeuxAppelsRules
  globalObjects: ''
  category: 'OPUS-rules-fonctions'!
```

Puis les deux règles `appel` et `retour` sont définies :

```
appel
| FonctionRecursive f|
      f val isNil.
      f testFin not.
      f fils1 isNil.
actions
      | newFonc1 newFonc2|
      newFonc1 _ f class new.
      f fils1: newFonc1.
      f initFils1.
      newFonc2 _ f class new.
      f fils2: newFonc2.
      f initFils2.
      newFonc1 go. newFonc2 go.
```

```
retour
  | FonctionRecursive f f1 f2|
  f val isNil.
  f1 _ f fils1.
  f2 _ f fils2.
  f fils1 val exists.
  f fils2 val exists.
actions
  f calculeValeur.
  f modified.
```

Une nouvelle version de `Fibo` peut maintenant être écrite, où `Fibo` est sous-classe de `FonctionRecursiveUnArgumentDeuxAppels`, et redéfinit les quatres méthodes utilisées par les règles :

```
FonctionRecursiveUnArgumentDeuxAppels subclass: #Fibo
  instanceVariableNames: ''
```

```
calculeValeur
  val _ fils1 val + fils2 val

initFils1
  fils1 arg: arg - 1

initFils2
  fils2 arg: arg - 2

testFin
  ^arg = 1 or: [arg = 2]

valeurFin
  ^1
```

Un exemple d'utilisation est alors :

```
'Fibo methodsFor 'exemple'!
exemple
   FonctionRecursiveUnArgumentDeuxAppels
         executeWithSingleObject: (self new arg: 10).
```

Cette transformation est relativement généralisable. Pour un nombre quelconque d'appels récursifs, si les appels peuvent être exécutés en parallèle (comme dans `Fibo`), la généralisation est immédiate. En revanche si les appels doivent être exécutés en séquence, alors il faut les synchroniser : c'est le cas de `Hanoi`.

Dressons un bilan graphique de notre travail (Cf. Figures 20 et 21) :

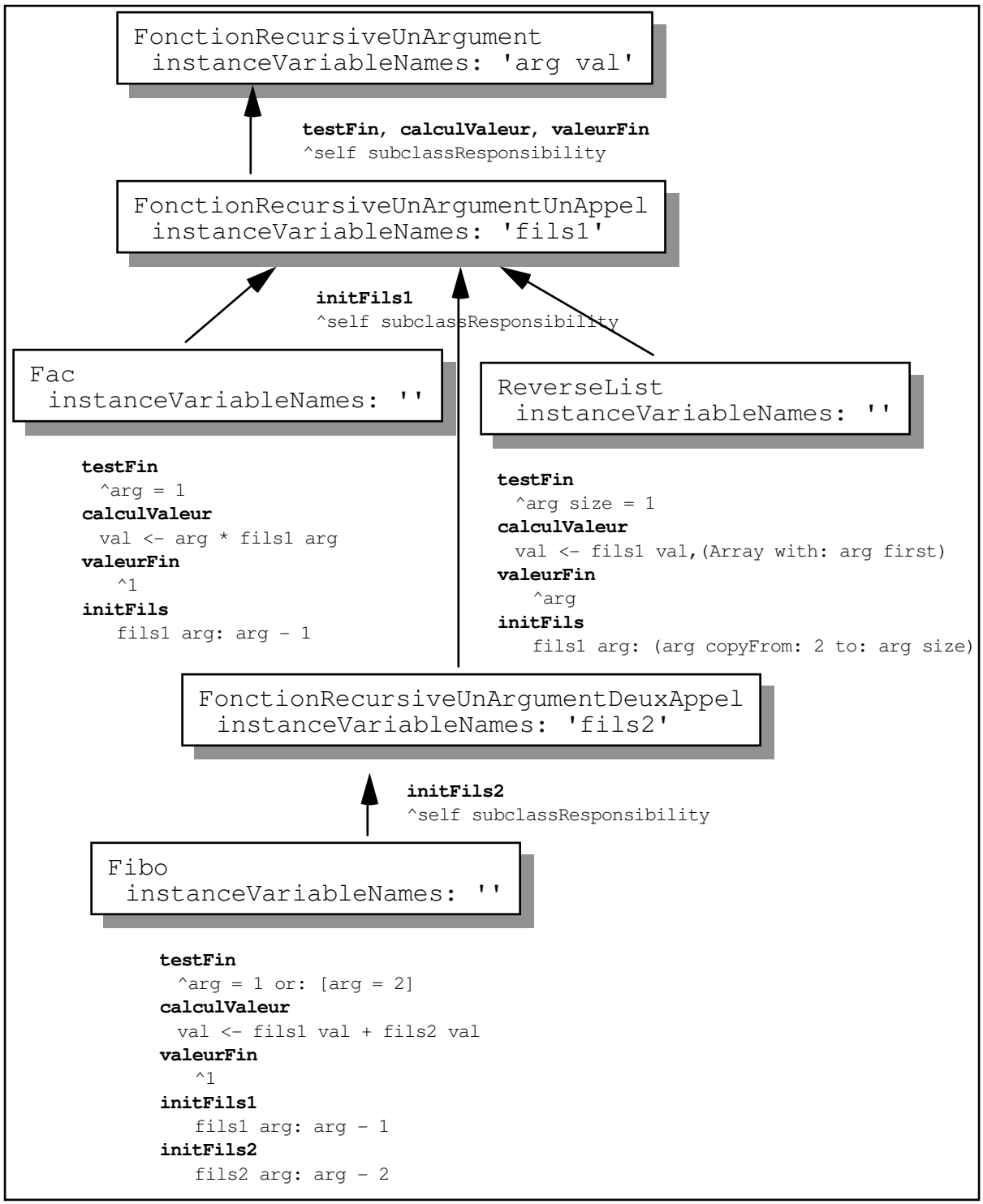


Figure 20. Le graphe d'héritage pour les classes représentant les fonctions récursives

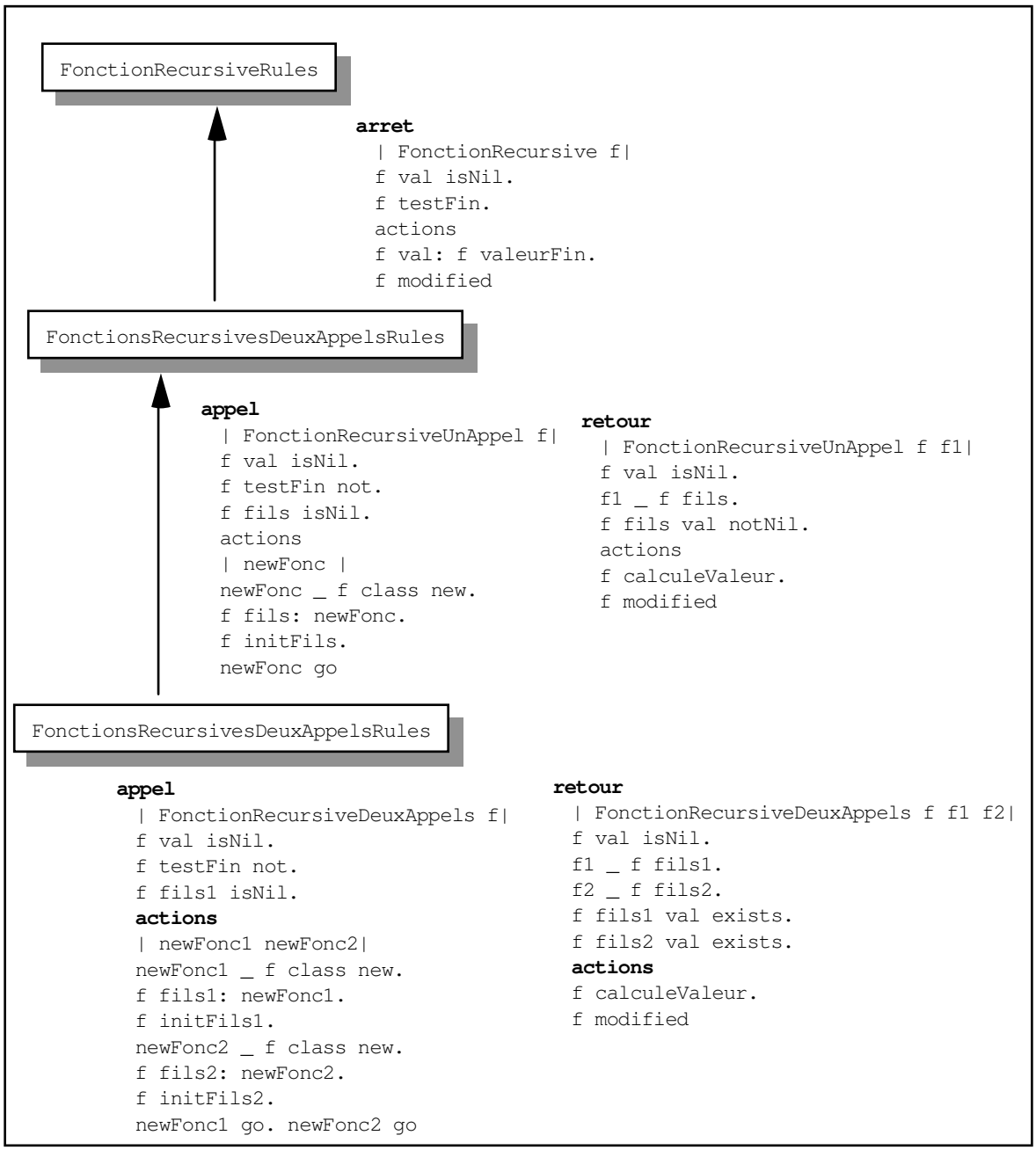


Figure 21. Le graphe d'héritage pour les bases de règles de calcul de fonctions récursives

V.1.2.6.3. Discussion

Mise en œuvre de l'héritage de bases de règles

Cet exemple tout à fait scolaire (les fonctions récursives se représentent très bien dans un langage récursif, comme Smalltalk) permet tout de même d'illustrer la notion d'héritage de base de règles.

V.1.2.7. Utilisation des objets nommés : La date du lendemain

Problème
Ce problème a déjà été utilisé comme exemple pour le système Oks [Voyer 89a]. Il s'agit de calculer la date du lendemain, en tenant compte des années bissextiles pour le mois de février, et du changement d'année le 31 Décembre. Nous reprenons ce problème pour le traiter en respectant le plus fidèlement possible la démarche "humaine", c'est à dire en mimant l'*égrenage* des mois sur les phalanges, et en traitant différemment les cas simples indépendants de la longueur du mois, des cas plus compliqués où la connaissance de la longueur du mois est nécessaire.

Objets
Les objets du domaine sont ici :

les *Quantièmes*, assimilés à des entiers,
les `Mois` (représentés par un `nom`, un `quantième` et une `longueur`). Une méthode d'initialisation de classe permet de créer les 12 instances canoniques de `Mois`. Les longueurs sont initialisées à 0 et sont calculées par les règles.
les `Années` (représentées par un nombre), et répondant au message `bissextile` par un booléen approprié,
les `DateOpus`[28] représentées par un `quantième`, un `mois` et une `année`, instances des classes précédentes.

Le calcul du lendemain consiste alors à créer une nouvelle instance de `DateOpus` à partir d'une date donnée, représentant son lendemain défini par les conventions internationales.

Règles
Notre idée ici est de déclarer le résultat du calcul, en l'occurrence la date du lendemain comme un objet nommé (`Demain`) de la base de règles :

```
OpusRuleSet subbase: #DateDuLendemain
  globalObjects: 'Demain '
  category: 'OPUS-rules'!
```

On écrit alors un jeu de règles permettant de traiter les différents cas. Les règles filtrent une instance de `DateOpus`, représentant la date d'aujourd'hui, une instance de `Mois` représentant le mois de la date, et l'objet nommé `Demain`, représentant la date du lendemain.

[28] La classe `Date` existe déjà en Smalltalk mais n'est pas réutilisable ici car elle est représentée de manière trop particulière.

Voici alors les règles traitant les cas "simples" c'est à dire pour lesquels la longueur du mois est inutile :

```
!DateDuLendemain methodsFor: 'pas besoin de la longueur du mois'!

dernierDuMois
"si on est le dernier jour du mois alors demain est le premier du mois suivant
On gere aussi le passage a l'annee suivante. On pourrait aussi ecrire deux regles"
   | DateOpus d. Mois m. Global Demain|
   d quantieme >= 28.
   m _ d mois.
   m longueur exists.
   d quantieme = m longueur.
actions
   Demain quantieme: 1; mois: m suivant; annee: d annee.
"si on est en decembre alors on passe a l'annee suivante"
   d mois = Decembre ifTrue: [Demain annee incremente].!

loinDeLaFin
"pour un quantieme inferieur a 28, la date du lendemain est immediate"
   | DateOpus d. Global Demain|
   d quantieme < 28.
actions
   Demain quantieme: (d quantieme + 1); mois: d mois; annee: d annee!

loinDeLaFinSupA28
"redondant avec la regle loinDeLaFin, mais makes sense anyway"
   | DateOpus d. Mois m. Global Demain|
   d quantieme >= 28.
   m _ d mois.
   m longueur exists.
   (d quantieme = m longueur) not.
actions
   Demain quantieme: (d quantieme + 1); mois: d mois; annee: d annee!
```

Notons ici que la variable `m` est utilisée comme variable locale déclenchante : elle est fonctionnellement accessible par l'objet `DateOpus`, mais ses modifications peuvent remettre en cause l'état d'instanciation de la règle.

Pour les cas plus compliqués, où l'on doit calculer la longueur d'un mois, on se dote alors explicitement une paire de mains, qui va nous permettre d'égrener les mois, selon la pratique populaire.
La classe `DeuxMains` est créée pour la circonstance pour représenter la théorie de l'égrenage des phalanges :

```
Object subclass: #DeuxMains
  instanceVariableNames: 'phalanges position '
```

Un jeu de méthodes permettent de faire marcher cette main :

```
!DeuxMains methodsFor: 'egrenage'!

avanceUnCran
  position _ position + 1!

bosse
  ^(phalanges at: position) = #b!

creux
  ^(phalanges at: position) = #c!

partDeJanvier
  position _ 1! !

!DeuxMains methodsFor: 'initialize'!

initialize
  "b = bosse, c = creux"

  phalanges _ #(b c b c b c b b c b c b c b).
  position _ 1
```

Avec une sempiternelle initialisation automatique :

```
!DeuxMains class methodsFor: 'creation'!

new
  ^super new initialize
```

Cette classe est instanciée dans la règle `trouverLongueur` ...

```
trouverLongueur
"si on doit connaitre la longueur, on se cree une paire de mains"
   | DateOpus d. Mois m |
   d quantieme >= 28.
   m <- d mois.
   m longueur exists not.
actions
   DeuxMains new go
```

... et utilisée dans la règle `longueurDesAutresMois` :

```
longueurDesAutresMois
"on egrenne les mois sur les deux mains"
   | Mois m. DeuxMains d|
   (m nom = #Fevrier) not.
   m longueur exists not.
    d exists.
actions
   d partDeJanvier.
   Mois tousLesMois do: [:unMois |
   (unMois = m) ifTrue:
            [d bosse ifTrue: [unMois longueur: 31]
                        ifFalse: [unMois longueur: 30].
            m longueur: unMois longueur. ^m modified].
            d avanceUnCran]
```

Le mois de Février mérite une règle à lui tout seul, pour traiter le cas des années bissextiles. Il faut en effet dans ce cas connaître l'année, donc filtrer la date :

```
longueurFevrier
"la longueur de Fevrier depend de l'annee"
   | DateOpus d. Mois m |
   m nom = #Fevrier.
   m <- d mois.
   m longueur exists not.
actions
   m longueur: (d annee bissextile ifTrue: [29] ifFalse: [28]).
   m modified.
```

Une méthode de lancement peut être la suivante : il faut initialiser l'objet nommé `Demain` avec une nouvelle instance de `DateOpus`, et lancer la base avec comme contexte la date à calculer, et les douze mois :

```
'DateDuLendemain class methodsFor: 'exemples'!

exemple
  Mois initialize.
  self lanceAvecDate: ((DateOpus new) quantieme: 4; mois: Decembre; annee:
(Annee new n: 1991))

lanceAvecDate: uneDate
  Demain _ DateOpus new.
  self executeWithObjects: (Mois tousLesMois add: uneDate; yourself)
```

Discussion

Séquencement des règles par instanciation

Le séquencement des règles est opéré par l'*instanciation* de l'objet `DeuxMains`. Ainsi les règles de calcul de mois ne sont déclenchables que quand une instance de `DeuxMains` *existe*.

Les mois n'ont pas à être tous présents

Seul le mois de la date à calculer peut être déclaré dans le contexte. On peut écrire alors une méthode de lancement qui rend la base de règles beaucoup plus efficace:

```
lanceAvecDate: uneDate
  Demain _ DateOpus new.
  self executeWithObjects: (Array with: uneDate mois with: uneDate).
```

Ceci nous montre comment l'initialisation du contexte d'activation d'une base de règles contient aussi des connaissances sur le mode de fonctionnement de la base en question.

Un seul demain, plusieurs aujourd'hui

Le fait que le lendemain soit déclaré comme objet nommé rend impossible l'utilisation de la base *pour plusieurs dates* simultanément. Si on veut se permettre cette possibilité, il faut alors déclarer la date de demain comme variable d'ordre un standard. Mais on perdra du coup la possibilité de *séquencement par instanciation* vue plus haut, puisque la création d'une paire de mains est dépendante du calcul d'une date particulière. La base de règles telle qu'elle est décrite ici implique très fortement l'utilisation d'*un seul* objet `DateOpus`.

Parler du mois de Février

Les règles `longueurFevrier` et `longueurDesAutresMois` sont maladroites à cause du test sur le nom : `m nom = #Fevrier.`
Puisque la règle a besoin de parler du mois de Février en particulier, il faudrait le déclarer comme deuxième objet nommé de la base de règles :

```
OpusRuleSet subbase: #DateDuLendemain
  globalObjects: 'Demain Fevrier'
  category: 'OPUS-rules'!
```

Mais cette solution n'est toujours pas satisfaisante, car il faudrait tout de même filtrer l'instance courante de `Mois`. En effet, la règle `longueurFevrier` s'écrirait alors (ainsi que sa sœur `longueurDesAutresMois`) :

```
longueurFevrier
"la longueur de Fevrier depend de l'annee"
   | DateOpus d. Mois m. Global Fevrier |
   m == Fevrier.
   m longueur exists not.
actions
   m longueur: (d annee bissextile ifTrue: [29] ifFalse: [28]).
   m modified.
```

L'écriture reste lourde, puisque on ne peut confondre à la déclaration, *l'instance de `Mois` dont on parle*, et qui a un caractère intrinsèquement d'ordre 1 et ***le*** *mois de Février*, objet nommé et fixe. On est obligé de filtrer les deux représentations de ce même objet et de tester leur égalité.

V.1.2.8. Utilisation perverse du `modified`

Le `modified` peut être aussi utilisé de manière indirecte, en modifiant des objets *qui ne sont pas filtrés explicitement dans les règles*. Cette écriture est possible mais entraîne une utilisation parfois redondante de variables dans la partie déclaration.

V.1.2.8.1. Exemple : évaluation d'un réseau de Petri

Les réseaux de Petri ont une mécanique d'évaluation qui se décrit naturellement sous forme d'action conditionnelle. Nous allons la représenter sous forme de règle NéOpus. Un réseau de Petri est constitué de *places*, de *transitions* et de *messages*. Une transition de Petri est *valide* si le nombre de jetons de ses places aval est supérieur aux poids des arcs joignant ces places, et si les messages en entrée de la transition sont tous valides. Les messages sont des booléens : un message est valide s'il est vrai. On a donc les définitions Smalltalk des `TransitionPetri`, `PlacePetri`, `ArcPetri` suivantes :

```
Object subclass: TransitionPetri
  instanceVariableNames: 'arcsAval arcsAmonts'

valide
  rend un booléen

declenche
  arcsAval do: [:a decrementeNbJetonsPlace].
  arcsAmont do: [:a a incremente nbJetonsPlace].
  messagesAval do: [:m | m update].

Object subclass: PlacePetri
  instanceVariableNames: 'nombreDeJetons'

Object subclass: ArcPetri
  instanceVariableNames: 'place transition poids'
```

A chaque cycle de l'évaluation, une transition est choisie et déclenchée. Le déclenchement d'une transition a pour effet de modifier le nombre de jetons des

places aval et amont, et de modifier les messages avals de la transition. Mais il est important de noter que *la transition elle-même n'est pas modifiée* !

Une première écriture d'une règle décrivant le déclenchement d'une transition est donc la suivante :

```
declencherTransition
"si une transition est valide, on la déclenche"
| Transition t |
   t valide.
actions
   t declenche.
   t places areModified.
   t messagesAval areModified.
```

Mais ici, le fait de signaler les places et les messages comme modifiés ne va pas remettre à jour l'état d'instanciation de la règle (avec éventuellement d'autres transitions devenues valides, après modifications des nombres de jetons et des messages), car seul les objets `Transition` sont filtrés.
Il faut alors permettre à cette règle d'être réessayée pour toute modification d'une des places ou d'un des messages de la transition. Cela se fera en déclarant ces objets comme variables de la règle :

```
declencherTransition
|Transition t. Place p. Message m|
   p transition = t.
   t aCommeMessage: m.
   t valide.
actions
   t declenche.
   t places areModified.
   t messagesAval areModified.
```

V.1.2.8.2. Discussion

Cette écriture a une conséquence importante et imprévue : l'ajout de variables fonctionnellement dépendantes (ici les places et les messages sont *connus*[29] de l'objet transition) entraîne une inflation des prémisses qui devront tester que ces objets sont effectivement reliés fonctionnellement entre eux (ici "`p transition = t`" et "`t aCommeMessage: m`").

[29] Cf le Chapitre I pour une définition de `connaitre`.

On peut certes limiter cette inflation en utilisant une variable locale déclenchante pour `t`, qui est accessible fonctionnellement à partir de la place, mais pas pour `m`, qui ne l'est pas (il faut saturer sur *tous* les messages) :

```
declencherTransition
|Transition t. Place p. Message m|
    p exists.
    t <- p transition.
    t aCommeMessage: m.
    t valide.
actions
    t declenche.
    t places areModified.
    t messagesAval areModified.
```

D'autre part, pour une transition valide, il y aura autant de règles déclenchables que de couples (place/message) pour cette transition, par le jeu de la saturation. Cependant, encore une fois, le mécanisme du `modified` sera tel qu'à chaque déclenchement de la règle, toutes les autres règles déclenchables par cette transition seront ré-essayées.
Bref, cet exemple d'apparence simple se révèle beaucoup plus difficile que prévu, alors même que la définition du mécanisme de déclenchement des transitions semble bien se prêter au jeu de la règle de production.

V.1.3. Troisième utilisation : avec stratégies de contrôle

Une utilisation plus générale consiste à utiliser des stratégies de déclenchement particulières, adaptées au problème à résoudre. Les règles sont alors écrites en fonction d'une certaine stratégie, ce qui leur ôte leur caractère modulaire.

V.1.3.1. Les bases de règles OPS5

V.1.3.1.1. Traduction directe

On peut bien sûr en NéOpus réécrire toutes les bases de règles écrites en OPS5. Il est même aisé de construire un traducteur automatique transformant les faits OPS5 en classes Smalltalk, et les règles OPS5 en règles NéOpus.
Cet algorithme doit remplacer chaque filtre d'attribut par une prémisse NéOpus, utilisant un message d'accès correspondant.

V.1.3.1.2. Traduction intelligente

Cependant une traduction directe n'est pas à conseiller. Il s'agit bien en NéOpus, d'une *reconception* des classes utilisées, de manière à :

- Prendre en compte l'héritage,
- Prendre en compte les dépendances fonctionnelles. Ainsi l'attribut `pere` de la classe `Personne` va pointer sur des instances de la classe `Personne`, et non sur des symboles représentant leur nom,
- Repérer les attributs OPS5 utilisés comme intermédiaires de calcul, et qui se représentent mieux sous formes de méthodes Smalltalk.

Bref, une traduction intelligente, elle, ne peut se faire de manière automatique, et doit être entreprise au coup par coup.

Un bon exemple d'utilisation de traduction indirecte est celui de la base de règles du singe et des bananes [Brownston 85, Vialatte 85, Rousset 88]. Celle-ci peut être reprise en NéOpus de manière indirecte, en transposant simplement les structures OPS5 en classes Smalltalk, et en traduisant syntaxiquement les règles une par une. Nous reprenons plus en détail cette base de règles au Chapitre VIII.2 et dans [Pachet 91c].
Notons ici simplement qu'il existe un problème de *compatibilité* lié au mode de contrôle OPS5. En effet, cette base de règles suppose (prérequis implicite dans la description originale) un déclenchement des règles respectant la stratégie OPS5 MEA : les sous buts créés dynamiquement doivent être considérés en *profondeur d'abord*, sous peine de voir le système errer et faire des inférences illicites. Ceci est réglé en choisissant la stratégie de contrôle OPSMEA pour la base de règles.
Mais cette stratégie pour fonctionner correctement implique que les objets filtrés par les règles aient une théorie de la fraîcheur convenable. Nous introduirons la notion de fraîcheur au Chapitre VII.4.5, en définissant la classe `OPS5Compatibility`. Il faut alors déclarer les objets dont la fraîcheur est significative, comme sous-classe de la classe `OPS5Compatibility`.
En l'occurrence, seule la classe `But` est déclarée comme sous-classe de `OPSCompatibility`, puisque la stratégie de contrôle doit préférer les règles filtrées par les buts les plus récemment créés.

V.1.4. Quelques benchmarks

Bien que la question de l'efficacité ne fasse pas partie de nos préoccupations principales, il est intéressant ici de donner quelques indications sur l'efficacité des inférences en NéOpus.
Nous donnerons deux exemples simples, qui sont facilement transposables dans tout système de règles d'ordre un : les règles de Fibonacci, et un exemple de transitivité de l'égalité.

V.1.4.1. Benchmark pour Fibonacci

La base de règles `FiboRules`, contenant trois règles, et décrite plus haut donne les résultats suivants. (les temps ne sont pas toujours identiques à cause du système de caching des méthodes de l'interprète Smalltalk).

Fib (10) (création de 109 objets, 163 déclenchements de règle)
sur Macintosh FX : 1,2 secondes maximum.
Sur Sparc 1 : 0.8 s.

Fib(15) (création de 1219 objets, 1828 déclenchements de règle).
Sur Macintosh FX : 57 sec.
Sur Sparc 1 : 45 sec

Ce temps est à comparer avec celui du système Oks (7 secondes sur Mac II). Aucun autre système écrit en Smalltalk n'offre de temps comparable (Essaim : 48 secondes pour Fib 10. Humble : impossible).

V.1.4.2. Benchmark pour Egalité

Nous proposons la base de règles singleton suivante, très coûteuse en saturation, proposée par Philippe Laublet pour son système ForreEnMat [Laublet 90] pour décrire la transitivité de l'égalité. On Fabrique la classe `Egalite`, ayant un membre droit et un membre gauche (tous deux des symboles), et une validité (un booléen) :

```
Object subclass: Egalite
   instanceVariableNames: 'droit gauche valide'
```

La base de règles `EgaliteRules` contient une seule règle dans laquelle on pourra ou non inclure un `modified` :

```
abc
"si (a = b) et (b = c) alors (a = c)"
   | Egalite e1 e2 e3 |
   e1 valide. e2 valide. e3 valide not.
   e1 droit = e2 gauche.
   e3 gauche = e1 gauche.
   e3 droit = e2 droit.
actions
   e3 valide: true. (e3 modified)
```

Une méthode d'exemple est alors écrite comme suit, en créant dix instances de chaque égalité :

```
!EgaliteRules class methodsFor: 'exemple'!

exemple
self executeWithObjects:
  (1 to: 10) collect: [:i | Egalite new gauche: #A droit: #B valide: true],
  (1 to: 10) collect: [:i | Egalite new gauche: #B droit: #C valide: true],
  (1 to: 10) collect: [:i | Egalite new gauche: #A droit: #C valide: false]
```

Suivant que l'on mette ou non le `modified`, on obtiendra, dans le cas de notre exemple, à peu près les mêmes résultats :

sans le modified : 12 secondes pour 1000 tirages.
avec le modified : 13,5 secondes, pour 10 tirages.

V.1.4.3. Tableau récapitulatif des résultats

Voici un tableau récapitulant les résultats. N'ayant pu réaliser les tests sur la même machine, nous mettons entre parenthèse la machine sur laquelle est effectuée le test. A titre indicatif nous pouvons proposer des équations approximatives sur les vitesses respectives pour comparer des temps sur des machines différentes :

Mac FX est 0.75 fois plus rapide que Sparc 1+.
Mac FX est 0.65 fois plus rapide que Sparc 2.

Systèmes **Bases de règles**	**NéOpus** (Mac FX)	**Essaim** (Mac FX)	**Clips** (Sparc 2)	**Oks**[30] (Mac FX Stk v. 2.3)
Fib (5)	0.1 s.	0.5 s.	négligeable	0.16 s.
Fib (10)	1,2 s.	48 s.	négligeable	1.95 s.
Fib (15)	55 s.	non testé	3 s.	explose
Egalite (10)	12 s.	non testé	non testé	explose

V.1.4.4. Conclusion sur l'efficacité

On peut vérifier ici le facteur exponentiel lié au nombre d'objets. Pour Fib (10), le temps moyen pour un déclenchement est de (1.2 / 163) = 0.0074 sec. Pour Fib (15), on obtient un temps pratiquement 10 fois moins bon : (57 / 1828 = 0.03 sec). Ceci est en accord avec les calculs d'efficacité des algorithmes à mémorisation [Ghallab 88, Voyer 89a]. Notre algorithme Rete revisité est en effet moins efficace que des

[30] Nous avons testé Oks (version non-monotone) sur la même machine que NéOpus, mais avec une version de Smalltalk plus ancienne (2.3), dont le compilateur génère un code légèrement moins efficace que celui de la version 2.5.

algorithmes de compilation plus sophistiqués du style de Treat [Miranker 90] ou Oks [Voyer 89a].
Cependant, les résultats sont bons pour un système basé sur un réseau de compilation Rete, et non industriel. Cela est du en grande partie à la facilité donnée par l'héritage à optimiser de manière très locale certaines des méthodes les plus fréquemment employées, *sans remettre en cause l'intégrité du système*.
En particulier, les points suivants ont pu être optimisés sans perturber l'intégrité du système :

. Fabrication d'un *graphe d'héritage extensif* pour les nœuds Rete, avec spécialisation des méthodes de propagation,
. Utilisation de *structures de données collectives adaptées*.
NéOpus passe la majeure partie de son temps à manipuler non pas des tokens en tant que tels, mais des *listes de tokens*. Nous avons donc créé une classe représentant une telle liste de token (la classe `TokenCollection`), qui optimise certaines méthodes de `Collection` spécifiques aux tokens (comme les retraits conditionnels de tokens d'une telle liste).
Le même principe a été appliqué pour les règles déclenchables : la classe `FireableRuleList`, optimise aussi certaines méthodes de `Collection` spécifiquement utilisées par les conflict sets.
. Techniques de cache pour certaines méthodes appelées très souvent (`ruleName`, `implementingRuleBase`). Les mises à jours des "attributs sensibles" sont effectuées de manière paresseuse.
. *Optimisation des méthodes sensibles* (i.e. appelées le plus souvent) comme la méthode `contains:` dans `OpusToken`, détectées grâce aux facilités d'espionnage de Smalltalk (`MessageTally`).

V.2. Pratique : prospection

Dans cette deuxième partie, nous allons décrire un exemple plus complet de raisonnement géométrique [Pachet 90]. Cet exemple va nous permettre d'explorer les capacités du système à représenter un certain type de connaissances moins triviales. Nous conclurons ensuite sur les défauts qui nous semblent inhérents à notre système.
Le but de cette partie est de montrer que les objets "à eux seuls" ne suffisent pas à représenter toute l'information que l'on a *sur* eux. On utilise alors un mécanisme de délégation pour résoudre, partiellement, le problème.
Nous proposons plus tard (chapitre VIII) une généralisation de ce mécanisme pour la représentation de connaissances à l'aide de règles de production.

V.2.1. Exemple de la géométrie (1)

V.2.1.1. Objectifs

L'objectif de ce système est de "faire du raisonnement" sur des objets géométriques, de la géométrie euclidienne à deux dimensions, comme les polygones, les cercles, les droites. L'idée est d'utiliser un domaine où :

L'expertise est largement connue de tout le monde et indiscutée.
La représentation des objets du domaine est non triviale.
Les connaissances mises en jeu sont compliquées.

Bref, un domaine idéal pour mettre à l'épreuve nos idées sur la représentation de connaissances[31]. Mais notre but n'est pas de fabriquer un démonstrateur de théorèmes au sens de [Pastre 84, 90, Laublet 90, Braun 88, Dufourd 90], encore moins de proposer un cadre formel de raisonnement dans un tel univers. Ce qui nous intéresse ici est la représentation des objets et des connaissances mises en jeu, plus que la recherche d'un système général de raisonnement mathématique. Par exemple, nous ne faisons pas ici la distinction entre axiome, définition, et théorème[32].

[31] Ce système a été proposé par J.-F. Perrot, et avait déjà été l'objet d'une petite base de règles d'ordre 0 en Prolog. Cette base détermine les types de figures géométriques simples, à partir de certaines définitions ou théorèmes simples de géométrie (comme : 'un losange dont les diagonales se coupent à angle droit est un carré').

[32] un axiome étant ici considéré comme un énoncé générateur d'autres énoncés. En géométrie la nature des énoncés (définition, axiome ou théorème) varie en fonction des problèmes. Plusieurs définitions peuvent coexister pour un même type d'objet. Considérer l'une d'entre elles comme définition confère aux autres le statut de théorème et inversement.

V.2.1.2. L'expertise

L'expertise que nous cherchons à représenter ne fait pas intervenir de calculs réels et donc approchés, (comme les calculs de longueurs de segments obliques), et utilise la représentation existante en Smalltalk des points. Ceux-ci seront représentés par des instances de la classe `Point` Smalltalk, définie par ses deux coordonnées `x` et `y`, supposées[33] entières, et représentant un point sur l'écran (un pixel).
L'expertise mise en jeu fait intervenir des *objets intermédiaires*, créés pour les besoins de la cause lors des inférences. Ces objets doivent être pris en compte par le système dynamiquement. D'autre part, les connaissances géométriques feront rapidement apparaître des problèmes de contrôle, qui nécessiteront, en autres, une représentation adéquate de la notion de but.

Deux problèmes vont donc se présenter successivement : la représentation des objets de bases du domaine, la représentation des connaissances mises en jeu (les définitions ou théorèmes).

V.2.1.3. Représentation des objets géométriques

V.2.1.3.1. Insuffisance de l'héritage des langages à taxonomie de classe

Afin de représenter les différents types géométriques classiques (Les sous-types des polygones, comme les carrés, rectangles, losanges, triangles), l'héritage de classe tel qu'il existe dans les langages à taxonomie de classes, qu'il soit simple ou multiple, ne convient pas pour deux raisons :

1. Si ces différents types géométriques forment bien une taxonomie naturelle, l'héritage ne peut prendre en compte les relations de contrainte entre sous-types (par exemple : *un carré est un rectangle dont deux côtés adjacents sont de longueurs égales*). Les sous-types ne peuvent être définis simplement en termes d'ajout d'attributs ou de redéfinition de méthode.

2. L'héritage de classe, même multiple, oblige à *choisir* une définition parmi plusieurs possible. Un classe est définie de manière unique, et il n'est pas possible de dire par exemple, qu'*un rectangle est un parallélogramme qui est aussi un Trapèze rectangle* ou bien, que *c'est un quadrilatère dont deux angles opposés sont droits*. Une seule définition doit être choisie, l'autre devenant du coup un théorème.

V.2.1.3.2. Solutions pour représenter ces types géométriques

[33] simplement supposées, car Smalltalk n'est pas typé

Plusieurs solutions peuvent être envisagées pour représenter une taxonomie de concepts qui échappe à l'héritage de classe. Soit on étend la notion d'héritage, soit on fabrique de toutes pièces un autre mécanisme de classification.

V.2.1.3.2.1. Etendre l'héritage

La notion d'héritage de classe peut être étendue ou modifiée pour prendre en compte les contraintes définissant les sous-classes.

V.2.1.3.2.1.1. Le système Géophile

Un héritage par restriction, tel que celui proposé par [Braun 88] avec le système *Géophile*, permet de prendre en compte ces contraintes, en introduisant un nouveau type d'héritage. Dans ce système, deux types d'héritage existent : l'héritage par "ajout de champ", héritage traditionnel, et par "restriction". L'héritage par "restriction" consiste à définir une nouvelle classe par des *prédicats de classe,* qui définissent des contraintes imposées aux entités de la classe plus générale. Les deux types d'héritage sont cumulables.
La notion de prédicat de classe à l'avantage d'être simple à mettre en œuvre, mais présente un certain nombre de problème quant à son intégration dans un univers complètement objet, notamment par rapport à la notion d'instanciation.
En effet, le système géophile propose un mécanisme d'instanciation très différent du mécanisme orthodoxe, puisque la classe d'un objet est déterminée a posteriori par le système de manière automatique, à l'aide d'un algorithme de classification, et non par l'utilisateur[34].
Si l'on définit par exemple la classe `Matrice`, et plusieurs sous-classes représentant les matrices bijectives (déterminant non nul), isométriques, etc. L'utilisateur peut créer une instance de la classe `Matrice` sans se préoccuper de ses propriétés particulières, le système déterminera automatiquement la bonne sous-classe (la plus spécifique dans l'arbre d'héritage), qui sera, in fine, la véritable classe de l'objet créé.

Les deux principales objections à cette architecture dans notre cadre sont :

- Mutation des objets.
  Un objet pouvant être modifié, l'évaluation des prédicats de classes peut donc changer. Cela rend ces prédicats très peu maniables : à chaque modification de l'objet, un parcours de l'ensemble des prédicats des superclasses est nécessaire. D'autre part, que devient alors l'objet ? Le système Smalltalk n'est pas construit pour permettre la mutation d'objets. Celle-ci doit être réalisée par l'utilisateur.

[34] Nous ne pouvons donc pas parler de langage objet au sens strict, qui repose de manière fondamentale sur le mécanisme d'instanciation : un objet est créé à partir de sa classe (unique) et ne peut en changer [Cointe 84].

- Unicité des définitions.
Une définition doit être préférée parmi les possibles. Par exemple, Géophile définit le `Trapèze`, comme sous classe de la classe `Quadrilatère`, avec un prédicat de classe testant que les deux côtés opposés sont parallèles. Ce mécanisme fournit une manière unique de définir et de créer un trapèze. Nous voulons être capable de *découvrir* qu' un objet est un trapèze, et ce, de manière quelconque : soit par ce que deux côtés opposés sont parallèles, soit parce qu'un théorème, après construction trouve que la figure est un trapèze, ou soit par ce que l'on décide *arbitrairement* que cette figure est un trapèze.

V.2.1.3.2.1.2. Une solution en Oks

Un autre solution consiste à définir les prédicats de classes comme des réflexes, au sens de [Voyer 89a, Bouaud 89a, 89b] (pour une présentation critique du système Oks, Cf. II.2.3.4). On trouve alors une utilisation particulièrement judicieuse du mécanisme de réflexe.
Un réflexe est une procédure qui se déclenche dès qu'un certain état du monde est atteint, de manière opportuniste (au contraire des déclenchements des systèmes à base de règles traditionnels, qui eux, nécessitent un appel explicite au moteur d'inférence).
Ainsi, on peut définir par exemple la classe `Carre` comme simple sous-classe de la classe `Rectangle`, sans ajout de champs. Un réflexe est alors écrit, qui va se charger de dire qu'un rectangle dont deux côtés adjacents sont de longueurs égales est un carré (on utilise la méthode `r` de la classe `Point` rendant la longueur du vecteur correspondant) :

```
rectangleDevientCarre
|Rectangle r|
<r origine ?o>
<r corner ?c>
(r  topLeft - r topRight) r = (r topLeft - r bottomLeft) r.
alors
r devientUn: Carre
```

La méthode `devientUn:` peut être définie par défaut dans la classe `Object`, de manière à opérer la mutation de l'objet dans un nouvel objet de la classe passée en paramètre. Il faut alors faire un certain nombre de suppositions sur la nouvelle classe concernant la présence/position des valeurs des variables d'instances : que faire par exemple des variables d'instances non présentes dans la nouvelle classe ? On retrouve ici l'idée de l'instanciation multiple d'objets, telle qu'elle existe dans le langage *Rome* [Carré 90] par exemple.

Une solution "objet" consiste à déléguer l'opération de mutation au niveau des classes elles-mêmes, en laissant le problème entre les mains du programmeur.

V.2.1.3.2.1.3. Critiques

Le mode de programmation réflexe est particulièrement bien adapté à l'idée des prédicats de classes, grâce en particulier au mode de déclenchement opportuniste. L'algorithme de classification est "délocalisé" et remplacé par un ensemble de réflexes, qui peuvent être définis en nombre quelconques. Mais cette solution a aussi des inconvénients :

- Pas de maintenance de la vérité

De manière parallèle, il faut définir un réflexe dans la classe `Carre`, pour les mutations inverses. Par exemple :

```
carreDevientRectangle
|Carre r|
<r origine ?o>
<r corner ?c>
((r  topLeft - r topRight) r = (r topLeft - r bottomLeft) r) not.
alors
r devientUn: Rectangle
```

- Limites dues à Oks

Les limites d'Oks sont décrites au chapitre II. Donnons ici un simple exemple : si l'on modifie non pas l'*objet* lui-même (Cf la notion de *d-modification*), mais une de ses variables d'instances (Cf la notion de *i-modification*), les déclenchements n'ont alors pas lieu. Par exemple, si l'on crée un rectangle et que l'on modifie l'origine du rectangle (le point *origin* directement), sans pour autant *modifier* cette origine (le rectangle pointe toujours sur le même point origine), alors aucun déclenchement n'a lieu, comme le montre cette micro-session :

Exemple de session avec Oks :

Expressions Smalltalk	Résultats et commentaires

`| r |`
`r <- Rectangle new.`
`r origin: 0@0`
`corner: 20@10.`	

	"création d'un rectangle"
`r class`	
	"le rectangle n'est pas carré"
`r corner: 20@20`	
	Rectangle
`r class`	
	"déclenchement du reflexe rectangleDevientCarre"
`r origin: 1@2`	
	Carre
r class	
	"déclenchement du reflexe carreDevientRectangle"
r origin translateBy: (-1@-2)	
	Rectangle de nouveau
r	
	"On ne change pas l'origine, mais on la modifie :
r class	aie : aucun déclenchement !"
	00 corner: 20@20
	Rectangle "helas"

V.2.1.3.2.2. Représenter la typologie par un mécanisme ad hoc

Notre vision de la typologie géométrique se traduit beaucoup mieux en termes de délégués [Lieberman 86a]. Les objets géométriques sont représentés par une classe Smalltalk (`Figure`)dépourvue de typologie a priori. Les différents types possibles *déduits* par le système de cette figure seront représentés par autant d'objets, liés par un mécanisme de délégation à l'objet principal.
Dans notre système, les figures seront exclusivement des polygones. La classe `Figure` implémente donc uniquement la structure essentielle des polygones, à savoir une liste de points, et une liste de délégués :

```
Object subclass: #Figure
        instanceVariableNames: 'points delegues'
```

V.2.1.3.2.3. Racine des types géométriques

Les différents types géométriques seront des instances de classes Smalltalk reliées entre elles de manière à refléter la taxonomie naturelle. Ces classes sont toutes sous

classes de la classe racine `TypeGeometrique`, définie comme suit, ave un pointeur vers la figure (appelé `modele`) :

```
Object subclass: #TypeGeometrique
  instanceVariableNames: 'modele'
```

Le lien entre les types est un lien multiple : un type peut avoir plusieurs super-types. Ceci se représente par une variable d'instance de métaclasse *superTypes* définie dans la métaclasse `TypeGeometrique class` :

```
TypeGeometrique class
  instanceVariableNames: 'superTypes'
```

Une méthode de création de sous-types adaptée est définie, permettant de créer un nouveau type en spécifiant, en plus des variables d'instance éventuelles, la liste de ses `superTypes` :

```
!Type class methodsFor: 'sub type creation'!

declare: aSymbol sousTypeDe: aListe instanceVariableNames: ivnString
category: c

    | new |
"Creation de la classe Smalltalk"
new <- self subclass: aSymbol instanceVariableNames: ivnString
classVariableNames: '' poolDictionaries: '' category: c.

"on lui affecte la liste des supertypes une fois scannee"
new superTypes:
    ((Scanner new scanFieldNames: aListe)
          collect: [:s| Smalltalk at: s asSymbol]).
^new
```

Les types géométriques sont ainsi de véritables classes Smalltalk, qui sont toutes sous-classes directes de la classe `TypeGeometrique`, le lien naturel de sous-type étant géré "à la main" (c'est à dire sans utiliser l'héritage de classe) comme une variable d'instance de métaclasse.

V.2.1.3.2.4. Définition de types

Ces types sont donc de véritables objets, avec structures et comportement. Par exemple, le type `TriangleIsocele` est définit comme ayant une base et un sommet.

```
TypeGeometrique declare: #TriangleIsocele
  superTypes: 'Triangle'
  instanceVariableNames: 'base sommet'
```

La base et le sommet sont des informations redondantes si l'objet est un triangle. On peut donc implémenter ici des méthodes de "maintien de cohérence", qui trouvent

l'un en fonction de l'autre, en utilisant bien sûr le modèle (instance de la classe `Figure`, qui contient la liste des points).
Par exemple, on peut trouver le sommet connaissant la base ou l'inverse :

```
sommet
  sommet isNil ifTrue:
   [base isNil ifFalse:
        [sommet <- modele autrePointQue: base. ^sommet]].
  ^nil

base
  base isNil ifTrue:
   [sommet isNil ifFalse:
        base <- modele coteOpposeA: sommet. ^base.
  ^nil
```

Nous n'envisageons pas ici de procédure automatique de calcul, qui déborderait trop de nos préoccupations initiales. Voici alors dans notre nouvelle syntaxe les définitions Smalltalk de quelques uns des types géométriques (Cf. Figure 22):

```
TypeGeometrique                declare:
#Quadrilatère
  superTypes: 'TypeGeometrique'
  instanceVariableNames: ''

TypeGeometrique declare: #Trapeze
  superTypes: 'Quadrilatere'
  instanceVariableNames: ''
```

```
TypeGeometrique                        declare:
#TriangleIsocele
  superTypes: 'Triangle'
  instanceVariableNames: 'base sommet'

TypeGeometrique declare: #Carre
  superTypes: 'Losange Rectangle'
  instanceVariableNames: ''
```

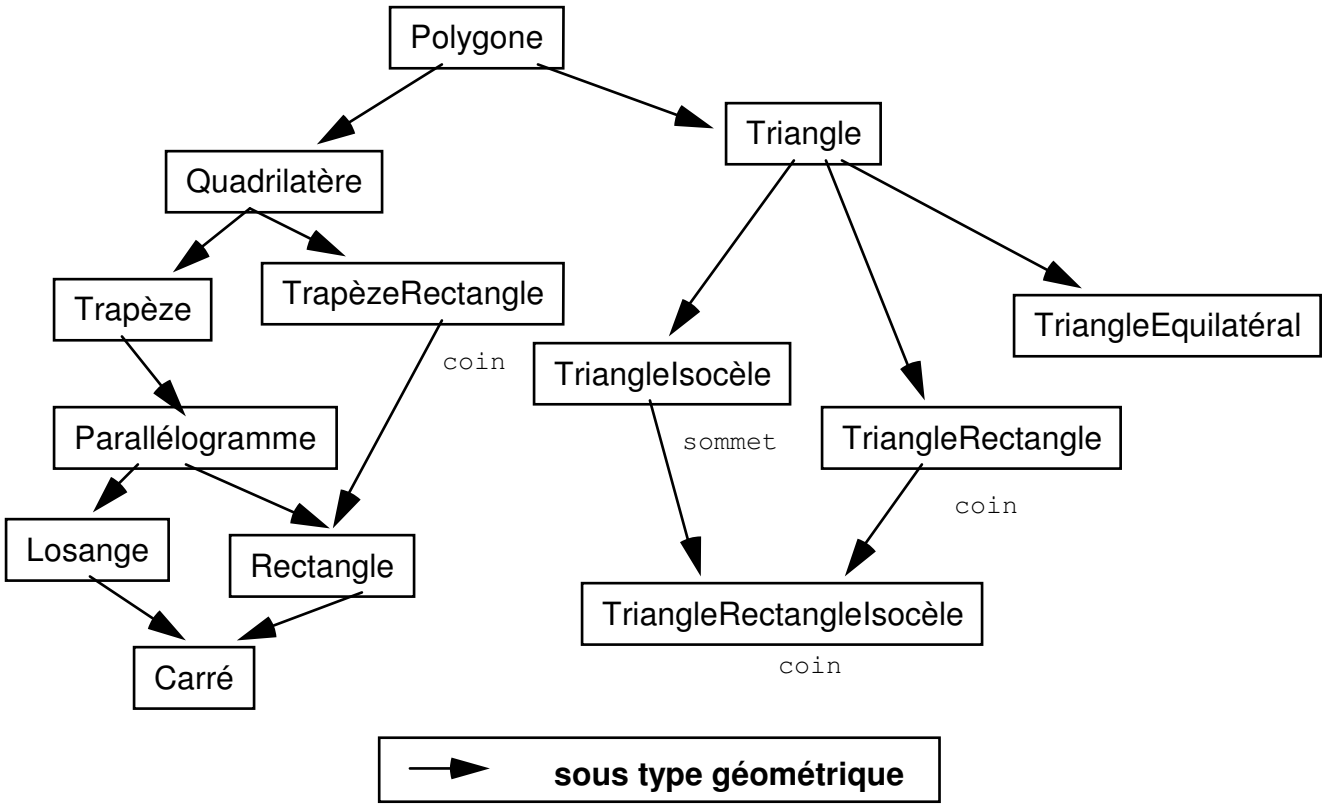


Figure 22. Les types géométriques

V.2.1.3.2.5. Création d'une instance de Type géométrique

La création d'une instance d'un type se fait alors par une méthode d'instanciation appropriée `newFrom:` qui prend en argument le modèle.
Par exemple, pour créer un type `TriangleIsocèle` pour le modèle `aModele` :

```
TriangleIsocele newFrom: aModele
```

V.2.1.3.2.6. Les méthodes d'accès aux types

Une batterie de méthodes d'accès génériques aux délégués d'un objet sont définies, de manière à tester et modifier les délégués (`addType: estUn:`), et à réaliser l'acte de délégation explicite standard (`as:`). Ces méthodes sont les suivantes :

1/ Méthodes de test de présence d'un délégué d'un type donné

```
!Figure methodsFor: 'tests sur les delegues'!

estUn: unType
"teste si l'objet possede un delegue de type unType"
 ^delegues values detect: unType

nEstPasUn: unType
 ^(self estUn: unType) not
```

2/ Méthode d'accès et création de délégué

```
!Figure methodsFor: 'acces aux delegues'!

addType: unType
 delegues at: unType class put: unType newFrom: self

as: unType
"renvoie le delegue de type unType"
 delegues at: unType ifAbsent:
       [self error: 'pas de delegue de type ', unType printString]
```

Ainsi, par exemple, pour tester qu'une figure est un `TriangleIsocele` :

```
aFigure estUn: TriangleIsocele
```

Pour dire qu'une figure est un `TriangleIsocele` :

```
aFigure addType: TriangleIsocele
```

Pour parler au délégué de type `TriangleIsocele` :

```
aFigure as: TriangleIsocele
```

V.2.1.4. Mise en route du système

Une figure initiale est créée graphiquement par l'utilisateur, qui spécifie la liste des points à la souris. Cette figure est une instance de la classe `Figure`, initialement créée avec une collection vide de délégués.
Puis la base de règles est activée (par clic dans un bouton de la fenêtre). Il n'y a donc pas, comme en Oks, de déclenchement opportuniste : le lancement de la base de règles est explicite. Le résultat de l'activation de la base est de mettre à jour la liste des délégués pour la figure créée par l'utilisateur.

Munis de notre représentation des objets du domaine (figures et types), nous pouvons d'ores et déjà écrire quelques règles simples. Une base de règles `ReglesDeFigures` est définie, contenant toutes les règles de notre système, qui sont réparties en différents protocoles, correspondant aux types des figures traités (carrés, triangles, losanges).

V.2.1.4.1. Règles de définition de types

Les règles qui sont des définitions de types peuvent ici trouver leur place. Définir un type consiste à ajouter un délégué du type correspondant, si les conditions nécessaires sont réunies. Le formalisme de règle de production est bien adapté ici puisque :

Il ne favorise pas une définition plutôt qu'une autre.
Il représente de manière satisfaisante la notion de contrainte évoquée plus haut.

Voici, par exemple la définition de quelques types simples dans notre système :

Le quadrilatère, définit par un polygone de quatre points :

```
quadrilatere
| Figure f |
f nEstPas: Quadrilatere.
f points size = 4.
actions
f addType: Quadrilatere. f modified.
```

Le carré, définit comme étant un `Losange` et un `Rectangle` :

```
carre
| Figure f |
f estUn: Losange.
f estUn: Rectangle.
f nEstPas: Carre.
actions
Transcript show: 'un carre';cr.
f addType: Carre.
f modified.
```

Mais d'autres définitions du carré sont possibles comme celle-ci :

"*un carré est un parallélogramme dont les diagonales se coupent à angle droit et sont de longueur égales*"

Nous nous permettons d'appliquer cette règle uniquement dans le cas où les diagonales ne sont pas obliques, pour éviter les calculs approchés de longueur. Il nous faut alors récupérer les diagonales de notre figure, via la variable locale `diagonales` et la méthode `lesDeuxDiagonales` :

```
carre2
| Figure f. Local diagonales |
f estUn: Quadrilatere.
f nEstPas: Carre.
diagonales <- f lesDeuxDiagonales.
diagonales first estVertical | diags first estHorizontal.
diagonales last estVertical | diags last estHorizontal.
diagonales first longueur = diags last longueur.
diagonales first milieu = diags last milieu.
actions
f addType: Carre. f modified
```

Ou bien encore, toujours pour le carré :

"*Un carré est un rectangle dont deux côtés adjacents sont de longueurs calculables et égales*"

```
carre3
| Figure f |
f estUn: Rectangle.
f nEstPas: Carre.
f aUnCoteHorizontal.
f aUnCoteVertical.
f coteHorizontal longueur = f coteVertical longueur.
actions
f addType: Carre. f modified
```

V.2.1.4.2. Entre définition et théorème

D'autres connaissances considérées traditionnellement comme des "petits" théorèmes, peuvent être représentées de cette manière. Voici par exemple une définition particulière d'un triangle rectangle :

"*Un triangle rectangle est isocèle si les deux longueurs calculables sont égales*"

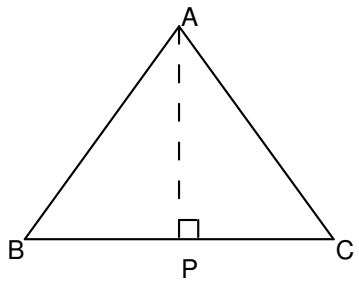
On a alors la transcription directe en règle NéOpus :

```
triangleIsoceleRectangle
| Figure f |
f estUn: TriangleRectangle.
f nEstPas: TriangleRectangleIsocele.
f aUnSegmentHorizontal.
f aUnSegmentVertical.
f coteVertical longueur = (f coteHorizontal longueur).
actions
f addType: TriangleRectangleIsocele.f modified
```

V.2.1.4.3. Calculs moins simples

Certains théorèmes simples trouvent ici une représentation directe. Par exemple le célèbre théorème sur le triangle isocèle :

Théorème (1)
Soit ABC un triangle qui a un côté horizontal. Soit A le sommet opposé à ce côté. Si sa projection sur BC est égale au milieu de BC alors le triangle est isocèle.




```
triangleIsocele
| Figure f. Local BC A P|
f nEstPasUn: TriangleIsocele.
f aUnCoteHorizontal.
BC <- f coteHorizontal..
A <- f pointOpposeA: BC.
P <- A projeteSur: BC.
P = BC milieu.
actions
f addType: TriangleIsocele. f modified.
```

Les théorèmes plus complexes nécessitent l'introduction dans le discours d'objets intermédiaires.

V.2.1.5. Objets intermédiaires

V.2.1.5.1. Idée

La notion d'objets intermédiaires, construits au cours du raisonnement est fondamentale dans le raisonnement géométrique. Dans nombre de théorèmes, des propriétés sur des objets *construits à la volée* permettent d'inférer des propriétés sur les objets initiaux du discours. Par exemple le théorème suivant considère deux objets intermédiaires :

Théorème (2) :

SOIT un Parallélogramme ABCD,
SOIT alors P le point construit par : BP = 2 BC,
CONSIDERONS alors le triangle BDP,
SI nous pouvons prouver que BDP est à angle droit en D,
ALORS notre parallélogramme est un lozange.

Mais les mathématiciens ont une théorie de l'*existence* bien particulière et très généreuse : il suffit de *parler* d'un objet pour le faire naître. Dans notre univers moins abstrait, et plus humain, les objets naissent, vivent, et meurent. Ainsi, avant d'en parler il faut les créer : si le parallélogramme ABCD existe effectivement a priori (tout en étant dénoté par une variable d'ordre un), il nous faut créer le point P et le triangle BDP de toutes pièces !

V.2.1.5.2. Des objets intermédiaires en NéOpus

Considérons notre théorème (2)[35]. Ce qui nous intéresse ici n'est pas d'amener le système à découvrir un tel théorème, mais bien de représenter ce théorème tel quel, et de le faire utiliser par le système.
L'idée essentielle de notre démarche est de remarquer la profonde différence de nature qu'il existe entre les variables d'un tel énoncé.
Le point `P` est un simple objet défini pour rendre le texte plus clair, et sur lequel aucun raisonnement n'est effectué. Mais le triangle ABP est d'un autre ordre. Il s'agit d'une sorte d'appel récursif au moteur, pour trouver des propriétés sur lui, et nécessitant *in fine* un retour arrière (retour au parallélogramme), en fin de raisonnement.
Si la création en partie prémisse d'un nouvel objet se résout directement par l'utilisation de *variables locales*, ce qui nous intéresse plus particulièrement est l'appel au moteur pour résoudre un sous-problème.

Plus précisément, cet appel comporte deux arguments :

- le nouvel objet sur lequel porter les inférences,
- une condition d'arrêt : ici la propriété d'être un triangle droit en `D`.

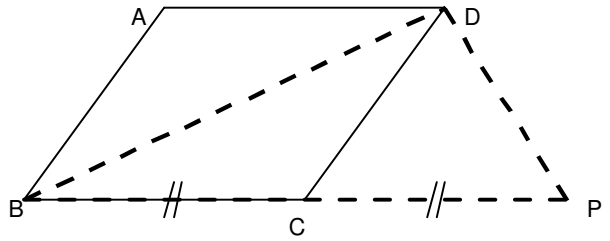
Nous allons pour ce faire introduire le message `lance:jusquA:`, qui permet de réaliser cette opération. Ce message est envoyé à la base de règles (via la pseudo-

[35] La démonstration mathématique de ce théorème peut être :

BC = CP donc, comme ABCD est parallélogramme, on a : AD = BC = CP.
Donc ADPC est un parallélogramme, donc AC est parallèle à DP
Donc, comme DP est à angle droit avec BD, on en conclut que AC est à angle droit avec BD, et donc que les deux diagonales de ABCD se coupent à angle droit.
Donc ABCD est un losange, cqfd

variable `selfBase`), et provoque un appel récursif au moteur. Nous détaillons au paragraphe suivant sa sémantique.

La règle s'écrit alors :



Théorème 1

losangeParTriangleDroit
| Figure f.
Local deuxCotes A B C P newObject|
f estUn: Parallelogramme.
f isNotA: Losange.
deuxCotes <- f cotesHorizontaux.
A <- deuxCotes plusBas gauche.
B <- deuxCotes plusHaut droit.
C <- deuxCotes plusBas droit.
P <- C + C - A.
newObject <- Triangle p1: A p2: B p3: P.
selfBase
 lance: (Figure new addType: newObject)
 jusquA:
[(newObject estUn: TriangleDroit)
 and:
[(newObject as: TriangleDroit) coin = D]].
actions
f addType: Losange. f modified

V.2.1.5.3. Sémantique du `lance:jusquA:`

La méthode `lance:JusquA:` réalise un appel récursif au moteur avec en argument un objet sur lequel vont porter les inférences, et une condition d'arrêt, représentée par un bloc Smalltalk. Le moteur va raisonner en chaînage avant, et s'arrêtera quand la condition d'arrêt sera atteinte ou qu'aucune règle ne sera applicable. Le message rend donc comme résultat un booléen suivant que la condition d'arrêt à été atteinte ou non. Il y a alors reprise des activités du moteur (i.e. propagation aux prémisses suivantes de la règle). Bien sûr ce mécanisme est récursif : il peut y avoir, pendant une exécution un appel récursif au moteur qui va lui même provoquer un appel récursif, etc ...
C'est donc une sorte de chaînage avant avec une notion réduite de but à atteindre.

Une première implémentation[36]

[36] Nous verrons au chapitre VI une représentation plus propre de ce mécanisme

L'idée est tout simplement de réaliser cette opération par un empilement d'appels au moteur suivis de la restauration du contexte initial. Le contexte est dans notre cas déterminé par *l'état du conflict set*. Le résultat renvoyé par la méthode est simplement le résultat de l'évaluation du bloc en fin d'exécution. La méthode s'écrit donc tout simplement comme suit :

```
!OpusRuleSet methodsFor: 'appels récursifs'!

lance: unObjet jusquA: unBloc
   |sauveConflictSet|
   sauveConflictSet := conflictSet fireableRules.
   conflictSet initRules.
   self executeWithSingleObject: unObjet jusquA: unBloc.
   conflictSet fireableRules: sauveConflictSet.
   ^unBloc value
```

Remarque

Il faut noter ici que cette méthode est lancée *en cours* de propagation des tokens dans le réseau Rete puisque elle est le résultat de l'évaluation d'une prémisse. Le réseau étant en cours de propagation, les tokens n'ont (éventuellement) pas encore fini de se propager. Mais, comme le montre notre implémentation du `lance:JusquA:`, l'état du réseau Rete *n'est pas* sauvegardé lors de l'appel récursif. Il y a donc dans ce réseau *co-existence* des jeux d'instanciations de l'exécution appelante avec les jeux d'instanciation de l'exécution appelée (et ainsi de suite).
Ceci est justifié par un argument assez subtil. En effet, l'appel récursif considère un *nouvel objet*, créé pour la circonstance. Il n'y a donc pas de jeux d'instanciation *communs* à deux exécutions !
Donc, toutes les opérations d'ajout, de retrait ou de modification d'objet ne vont pas remettre en cause l'état du réseau Rete des exécutions précédentes. On peut donc accumuler tous ces jeux dans les nœuds Rete sans avoir à gérer de sauvegarde/restauration pour les nœuds, ce qui rendrait rhédibitoirement inefficace notre approche.

V.2.1.5.4. Un deuxième exemple qui fait boucler le système

Voici un deuxième exemple du même type, mais qui introduit un nouveau problème. Soit à représenter le théorème suivant (3) :

Théorème (3)[37] : Un triangle ABC est isocèle en B, si le parallélogramme ABCP, tel que P = A + BC, est un losange

triangleIsoceleParLeLlosange |Figure f.
Local BC A B C P newObject|
f estUn: Triangle.
f nEstPasUn: TriangleIsocele.
f aUnCoteHorizontal.
BC <- f coteHorizontal.
A <- f pointOpposeA: BC.
B <- BC pointGauche.
C <- BC pointDroit.
P <- A + C - B.
newObject <- Parallelogramme new with: A with: B with: C with: P.
selfBase lance: (Figure new addType: newObject)
jusquA: [newObject estUn: Losange].
actions
f addType: TriangleIsocele.
(f as: TriangleIsocele) base: BC.
f modified.

B

Théorème 2

Ce théorème s'énonce en NéOpus de la même manière que le précédent. Néanmoins, on voit tout de suite apparaître un problème crucial de contrôle.
Supposons un raisonnement effectué sur un parallélogramme initial. La règle `losangeParTriangleDroit` va provoquer la création d'un triangle et d'un appel récursif. Celui-ci va à son tour créer un nouveau parallélogramme et un appel récursif. Ce deuxième appel va provoquer la création d'un nouveau triangle et ainsi de suite. L'application *non raisonnée* de ces règles peut conduire le système à créer de nouveaux objets sans fin (Cf. Figure 23).

[37] La démonstration mathématique de ce théorème peut être:

ABCP est un losange donc AC coupe BP en son milieu, et à angle droit. Or le milieu de BP n'est autre que la projection de B sur AC. Donc ABC est isocèle en B, de base AC.

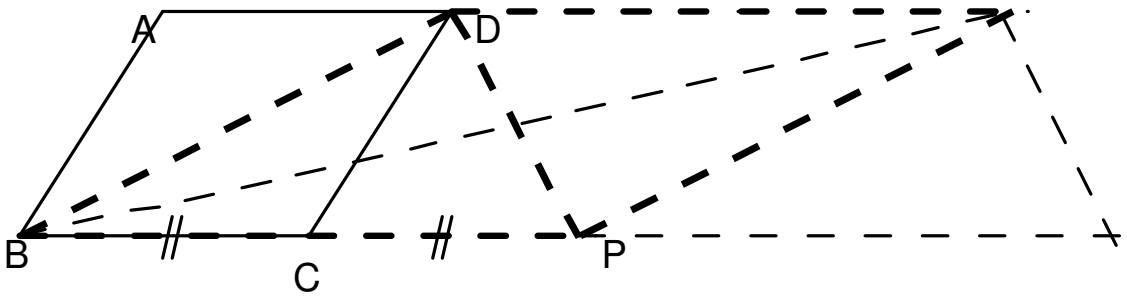


Figure 23. Bouclage du système avec les deux règles `losangeParTriangleDroit` et `triangleIsoceleParLeLosange`

V.2.1.6. Synthèse

Comme nous l'avons vu dans ce système de raisonnement géométrique, certains énoncés se laissent représenter facilement grâce à cette architecture. D'autres ne se laissent pas représenter, ou moins facilement. Voici deux sources générales de nos difficultés.

V.2.1.6.1. Une dissymétrie profonde

On voit clairement ici apparaître la profonde différence de nature entre les expressions utilisées en prémisses et celles utilisées en partie action. Par exemple, si l'on considère l'énoncé suivant :

Enoncé : "les deux segments AB et AC ont même longueur"

Cet énoncé se laisse facilement représenter en partie prémisse, en supposant que les deux segments répondent au message `longueur` par l'expression Smalltalk :

```
Prémisse :    AB longueur = AC longueur
```

où `AB` et `AC` sont instanciés par deux instances de la classe `Segment`.

En revanche, il est beaucoup moins facile, et même, dans notre cadre *totalement impossible* de représenter cet énoncé en partie conclusion : comment en effet représenter sous forme d'une *action* sur l'environnement Smalltalk, si riche qu'il soit, le fait que les longueurs de deux segments sont égales ?

Ainsi, s'il est simple de représenter le théorème simple sur le triangle isocèle (théorème 1), on ne peut pas en revanche représenter le théorème dual :

Théorème 1 bis si un triangle `ABC` est isocèle en `A` alors les longueurs des segments `AB` et `AC` sont égales

V.2.1.6.2. Une contrainte d'évaluabilité

Les prémisses étant des expressions Smalltalk, qui sont, au moment opportun, évaluées, celles-ci doivent être, c'est tautologique, évaluables. Mais cela ne correspond pas toujours à l'intuition. Si l'on reprend l'énoncé précédent, celui-ci perdra de sa précision dans sa transcription Smalltalk. En effet, dire que deux segments ont même longueur ne présuppose rien sur ces longueurs, en particulier sur leur calculabilité et la connaissance par le système de cette longueur : lorsqu'un triangle est isocèle, les deux longueurs des segments (autres que la base) *sont égales*, même si elles ne sont pas calculables !
Or l'expression Smalltalk, elle, devant être évaluable, on testera en fait l'égalité *des deux valeurs* des longueurs.

V.3. Le problème du modified revisité

Nous revenons ici sur le problème du modified, et plus généralement de l'usage des variables dans les règles NéOpus. Ce problème est une retombée directe du célèbre *frame-problem* [McCarthy 69], à savoir : "quels sont les objets concernés par la modification d'un objet". Le frame-problem a de nombreuses apparences. Citons en particulier celle qu'il revêt dans le système à base de frames Shirka [Rechenmann 85], sous forme de problème de caching de méthodes [Euzenat 90]. Il est alors résolu à l'aide d'un mécanisme de TMS adapté à cet environnement de frames.
Plus précisément, son instanciation NéOpus soulève deux questions :

Quels sont les objets à déclarer comme modifiés dans une règle ?

avec comme question corollaire :

Quels sont les objets à déclarer comme variable d'ordre un ?

En effet, le principe PPremisse (l'extension des prémisses NéOpus à toute expression Smalltalk) a un effet pervers que nous allons montrer ici : il oblige à **choisir** entre (1) l'abandon du principe de Fermeture (Cf Chap I.2.4.2.2.), ou (2) une utilisation *non intuitive* du `modified`.

V.3.1. Exemple

Prenons un exemple simple composé de deux règles que nous énoncerons en français :

r1 : "*toute personne dont la caution est inférieure à 20000 FF ne peut bénéficier de l'emprunt A.*

r2 : "*toute personne d'age égal à l'age de la retraite, devient à la retraite. Son salaire est alors diminué de 25%*"

Ces deux règles sont artificielles, mais se représentent bien en termes de règles de production en NéOpus.
Nous définissons la classe `Personne` de manière canonique, comme ayant un `pere`, éventuellement des `fils` (une liste), un `age`, et un `salaire` :

```
Object subclass: #Personne
        instanceVariableNames: 'age pere fils salaire'
```

Nous introduisons aussi la classe `Salaire`, de telle sorte que le salaire d'une personne soit une instance de cette classe (et non directement un nombre) :

```
Object subclass: #Salaire
        instanceVariableNames: 'montant'
```

Les méthodes d'accès standard sont alors définies, ainsi que la méthode suivante rendant la *caution* d'une personne, définie comme le montant du salaire du père:

```
!Personne methodsFor: 'caution'!

caution
"le montant du salaire du père"
  ^pere salaire montant
```

Nous allons maintenant étudier la représentation de ces deux règles en NéOpus, qui va s'avérer moins simple que prévu.

V.3.2. Définitions de la modification

La notion de *modification d'un objet* est une notion subtile. Avant d'examiner le problème du modified, nous devons définir clairement la notion de modification pour notre système.
Cette notion n'est pas des plus claires, comme l'illustre l'exemple suivant. Si l'on évalue l'expression suivante:

```
unePersonne age: 10.
```

Est-ce alors la personne qui est modifiée ou son age ?

V.3.2.1. Modification directe

Pour répondre au problème, nous définissons la notion de modification directe, liée aux variables d'instances d'un objet :

Définition de la modification directe
Un objet est dit d-modifié (directement modifié) par une expression `E`, si une de ses variables d'instances pointe sur un *objet différent* après évaluation de `E`

Exemple :

Après l'évaluation de l'expression :

```
  unePersonne salaire: (Salaire montant: 20000)
```

l'objet dénoté par `unePersonne` est d-modifié (la variable d'instance `salaire` pointe sur un objet nouveau, donc forcément différent de l'ancien, quel qu'il fût).

V.3.2.2. Modification indirecte

Nous définissons maintenant la i-modification comme la fermeture transitive de la d-modification :

Définition de la modification indirecte
Un objet est dit i-modifié (indirectement modifié) par une expression `E`, s'il est d-modifié, **ou** si l'objet dénoté par une de ses variables d'instance est i-modifié après évaluation de `E`.

Exemple :
Après l'évaluation de l'expression :

```
unePersonne pere salaire montant: 20000
```

l'objet dénoté par `unePersonne` est i-modifié, mais non d-modifié.

V.3.2.3. Modification pour une base de règles

La i-modification est une notion très large, qui inclut toutes les d-modifications sur tous les objets accessibles fonctionnellement. Nous allons restreindre cette notion en en définissant une troisième, qui limite la i-modification à une base de règles donnée :

Définition de la modification pour une base de règles
Un objet est dit r-modifié (règle-modifié), pour une base de règles donnée, par une expression `E`, s'il existe une prémisse de cette base de règles *concernée* par cet objet et `E`.

Il faut alors définir la notion de *concerné par* :

Définition d'une prémisse concernée par un objet et une expression
Une prémisse de règle est dite concernée par un objet, si sa valeur de vérité peut changer après évaluation de `E` sur cet objet

Par exemple, la prémisse suivante :

```
p pere salaire montant > 20000
```

est concernée par les objets (dénotés par les expressions) suivants : `p; p pere; p pere salaire`, avec les expressions :

```
p pere: Personne new.
p pere salaire: Salaire new,
p pere salaire montant: 1000, ...
```

En revanche la prémisse suivante n'est concernée que l'objet dénoté par `p` :

```
p age < 25
```

V.3.3. Le problème du `modified` revisité

Ecrivons maintenant nos deux règles en NéOpus. La première règle s'écrit alors naturellement en profitant du principe "Toute expression", c'est à dire en utilisant la méthode `caution` :

```
r1
"si la caution est inferieure a 2000 ..."
    | Personne p |
    p caution < 2000.
actions
    p nePeutPasBeneficierDuPlan ... "peu importe"
```

La règle `r2` s'écrira alors en suivant la même démarche :

```
r2
"si l'age est superieur a l'age de la retraite alors le salaire diminue"
    | Personne p |
    p age > p profession ageRetraite.
actions
    p salaire montant: (p salaire * 0.75).
   p modified.
```

Mais dans ce cas, il est clair que notre base de règles, hélas, ne produira pas les effets escomptés. En effet, la modification du salaire d'une personne (par l'évaluation de la partie action de la règle `r2`) ne va pas produire de re-essai sur la prémisse de la règle `r1`, car dans cette règle, seule une modification de l'objet dénoté par `p` peut provoquer un tel re-test !
Or l'application de la règle `r2` va modifier le salaire d'une personne, qui éventuellement pourrait rendre `r1` déclenchable.

Deux solutions existent pour rendre cette base conforme à l'idée que nous avons de la caution :

1 - changer la déclaration des objets modifiés,
2 - changer la déclaration des variables.

Nous allons dans les paragraphes suivants tenter d'apporter une vision claire sur ce problème, baptisé *problème du modified*.

Le problème s'énonce ainsi :

Problème du modified :
Dans une règle NéOpus, quels sont les objets à déclarer comme variables d'ordre un, et quels sont les objets à déclarer comme modifiés ?

Rappelons ici le principe (Cf Chapitre III.3.6.5), qui définit *l'action* exacte du `modified` sur les règles de la base de règles :

Principe d'action du modified
Déclarer un objet comme `modified` aura pour conséquence le re-essai de toutes les prémisses de toutes les règles de la base de règles dans lesquelles cet objet apparait comme variable libre d'ordre un.

Nous allons donner deux réponses : une réponse sage mais conformiste, et une autre moins sage mais plus intéressante et que nous adopterons par la suite.

V.3.4. Une solution sage

La seule solution sage consiste à s'interdire le principe "Toute expression", et à n'autoriser que les prémisses plates dans les règles. Une prémisse plate n'est autre qu'une expression plate (i.e. envoi de message dont le receveur et les arguments sont des variables, et non des expressions) ne comportant que des *messages d'accès* !
D'ou la définition suivante :

Définition d'une prémisse plate
Une prémisse est plate si tous les objets la concernant sont déclarés comme variables d'ordre un

Dans cette optique, les objets à déclarer comme variables d'ordre un sont simplement tous les objets intervenant dans les calculs mis en jeu.

Du coup, les objets à déclarer comme modifiés doivent être tous les objets d-modifiés dans la partie action. On est alors assuré d'une bonne marche du mécanisme de propagation, puisque *le système est alors strictement équivalent au système OPS5*. On a donc une première solution au problème du modified :

Solution 1 au problème du modified

- On déclare comme variable d'ordre un tous les objets concernés par toutes les prémisses.
- On déclare comme modifié tout objet d-modifié

Application

Dans notre exemple, il faut alors déclarer tous les objets intermédiaires concernant le message `caution` comme des variables d'ordre un. La règle `r1` devient alors la suivante, (la règle `r2` restant, elle, inchangée) :

```
r1
    | Personne p p2. Salaire s|
    p pere == p2.
    p2 salaire = s.
    s montant > 2000.
actions
    p peutBeneficierDuPlan ...
```

Critiques

Les défauts de cette approche sont ceux d'OPS : Tous les objets intermédiaires nécessaires à exprimer une contrainte doivent être exprimés explicitement. Cela porte un coup fatal au *principe de fermeture*. Dans notre cas, la méthode `caution` ne peut être utilisée dans la règle, on est obligé de l'éclater en composants plats. Bref les objets ne sont plus que des natures mortes. C'est donc véritablement un retour arrière vers le paradigme de représentation attribut-valeur, puisque les méthodes sont finalement limitées aux méthodes d'accès, et aux méthodes de tests (=,<,>, ...). Plus exactement, cela implique que les méthodes qui ne sont pas de simples méthodes d'accès (c'est à dire les plus intéressantes, d'un point de vue objet) *ne peuvent être utilisées dans une règle*.

De plus, le nombre de variables déclarées étant plus grand, cela induit une plus grande inefficacité vis à vis de la propagation Rete.

Cette solution est donc à la fois sage, parce que l'on connait parfaitement la mécanique d'OPS5, mais aussi archaïque, puisqu'elle supprime l'intérêt de l'extension des prémisse des règles au langage Smalltalk. Bref le système perd singulièrement de son attrait.

V.3.5. Une solution moins sage

Dans cette seconde optique on s'autorise l'usage de toute prémisse Smalltalk dans les prémisses. Il faut alors reconsidérer le problème du `modified`, puisqu'il ne suffit plus de déclarer comme modifiés les objets d-modifiés, comme nous l'avons montré sur notre mini-exemple, mais les objets dont la modification peut remettre en cause l'état d'instanciation d'une règle de la base, soit les objets r-modifiés.

On a alors une deuxième solution au problème du modified :

Solution 2 au problème du modified

- On déclare comme variable d'ordre un *les objets du discours*
- On déclare comme modifié tout objet r-modifié

Dans ce cas, le `modified` représente non pas la modification directe, ou indirecte, mais bien la r-modification.

En revanche, on peut se contenter de ne déclarer comme variable d'ordre un que les objets "importants", racines du discours.

Exemple

Reprenons notre règle telle que nous l'avons énoncée naturellement au §VI.3.2.3, c'est à dire en déclarant comme variable d'ordre 1 uniquement l'objet principal du discours (`p`).
Il faut alors informer la règle `r1` de la r-modification, non pas de p, mais *des fils de p !* :

```
r2
   | Personne p |
   p age > p profession ageRetraite.
actions
   p salaire montant: (p salaire * 0.75).
  p fils areModified.
```

Ce n'est en effet pas l'objet `p` qui est modifié dans cette optique, mais bien les fils, au regard du problème de la caution parentale (Le `modified` est ici un `r-modified`).

Critique

Dans cette optique, on doit représenter explicitement toutes les dépendances fonctionnelles existant entre les objets et les messages utilisés dans les prémisses. Mais cela n'est pas toujours possible, car il s'agit bien d'un renversement dans la représentation, et il nécessite l'existence de liens inverses.
Dans notre exemple, il faut en effet opposer :

A/ Dans la première écriture, la sur-population des prémisses de `r1`, avec disparition de la méthode `caution` :

```
p pere == p2.
p2 salaire = s.
s montant > 2000.
```

avec :

B/ dans la deuxième, la réapparition de la méthode `caution`, mais l'apparition du `modified` pour le lien inverse (`fils`) :

```
r1
   p caution > 2000
   ...

r2
   ...
   actions
   p fils do: [:f | f modified]
```

qui n'est possible que parce que le lien inverse du `pere` existe (`fils`). Si ce lien n'existait pas, la deuxième écriture serait impossible!
Nous allons maintenant explorer cette voie de programmation.

V.3.6. Utilisation intelligente des dépendances fonctionnelles

Certains objets Smalltalk sont liés entre eux de manière fonctionnelle, c'est à dire par un lien du type `connaitre:` (Cf Chapitre I.4).
Les règles NéOpus permettent de filtrer (déclarer l'objet comme une variable) un objet, et d'écrire des prémisses portant sur tout objet accessible à partir de l'objet filtré. La question se pose alors de savoir si ces autres objets doivent être aussi filtrés.

mauvais exemple d'utilisation de dépendances fonctionnelles

Prenons l'exemple d'une règle disant que le nom de famille *d'une* personne est celui de son père.
Une écriture possible serait : '*si p et p2 sont des personnes, telles que p2 soit le père de p, que p n'ait pas de nom, et que p2 en ait un, alors le nom de p est celui de p2 (son père)*'.

```
heritageNomFamille
   | Personne p p2 |
   p pere == p2.
   p2 nomFamille exists.
   p nomFamille exists not.
actions
   p nomFamille: p2 nomFamille.
   p modified
```

héritage du nom (1)

Mais ici, `p` et `p2` sont liés de manière fonctionnelle : `p` connait son père. Ainsi l'écriture suivante, dans laquelle seul `p` est filtré, serait-elle plus naturelle :

```
heritageNomFamille
   | Personne p |
   p nomFamille exists not.
   p pere nomFamille exists.
actions
   p nomFamille: p pere nomFamille.
   p modified
```

héritage du nom (2)

Mais cette seconde écriture, bien que plus concise, sera incorrecte du point de vue de la mécanique de NéOpus, car le père de `p` n'étant pas déclaré comme variable de la règle, toute modification de celui-ci ne sera pas prise en compte. Ainsi cette règle ne s'appliquera que si les *conditions initiales* vérifient les prémisses. Si l'objet père, initialement dépourvu de nom de famille, vient à en avoir un ultérieurement, (ou bien change de nom de famille) la règle ne sera pas réessayée.

bon exemple d'utilisation de dépendances fonctionnelles

En revanche, si les objets accessibles fonctionnellement n'ont pas (à la connaissance du programmeur) lieu d'être modifiés, ou si leur modification ne remet pas en cause l'état d'instanciation de la base de règles, alors la dépendance fonctionnelle peut être utilisée, et va grandement réduire les temps de filtrage.
Prenons l'exemple souvent cité (par exemple dans [Voyer 89a]) comme prototype de règle d'ordre un, le cas de la règle exprimant le fait que pour prendre un objet situé sous un autre, il faut générer un sous but de prendre l'objet qui est dessus.
L'écriture conventionnelle conduit à filtrer trois objets : le but, l'objet dessous (sur lequel pointe le but) et l'objet dessus :

```
rPrendreDessous
   | But b. Objet oDessous oDessus |
b isActive.
b tenir = oDessous.
oDessous sous = oDessus.
actions
Generer un sous but de prendre oDessus ...
```

L'écriture lispienne prédicative donnerait [Voyer 89] :

```
prendreDessous
   (?b isActive)
   (but tenir = ?o)
   (?o sous = ?o1)
alors
   (generer ...)
```

Mais ici, le filtrage de `oDessous` est inutile, puisque cet objet est accessible fonctionnellement à partir de b, et que sa modification ne va pas remettre en question l'état d'instanciation de la règle :

. la prémisse "`b tenir = oDessous.`" ne peut être, dans notre exemple, remise en cause (on suppose que les buts ne changent pas leur attribut `tenir`),

. la prémisse "`oDessous sous = oDessus`", elle, ne peut être remise en cause que par une modification de `oDessus`, et non de `oDessous` (le but de toute cette opération étant justement de représenter le fait que pour prendre un objet qui est sous un autre, il faut prendre celui qui est dessus ...).

L'écriture suivante, où `oDessous` est une simple variable locale (déclarée comme `Local`) est donc à la fois plus naturelle et plus efficace :

```
rPrendreDessous
   | But b. Objet oDessus. Local oDessous  |
b isActive.
oDessous <- b tenir.
oDessous sous = oDessus.
actions:
Generer un sous but de prendre oDessus ...
```

On peut même alors utiliser une variable locale déclenchante (Cf. chapitre IV.1.4) pour la variable `oDessus`, et ainsi court-circuiter la référence à `oDessous`, devenue inutile, de la manière suivante :

```
rPrendreDessous

  | But b. Objet oDessus|

b isActive.
oDessus <- b tenir sous.

actions:
Generer un sous but de prendre oDessus ...
```

Le règle obtenue est alors parfaitement correcte du point de vue de la mécanique du `modified`, et ne comporte aucun objet superflu. Il faut noter qu'alors on ne filtre plus que deux objets, au lieu des trois initiaux, ce qui réduit d'autant les propagations Rete et donc les temps de calcul.

V.3.7. Les objets à déclarer comme modifiés ne sont pas nécessairement les objets filtrés

Il faut noter que les objets déclarés comme modifiés ne sont pas nécessairement des objets filtrés. Ainsi on pourrait imaginer la règle suivante, où l'objet `pere` est filtré mais non modifié alors que ses fils sont modifiés mais non filtrés :

```
heritageNomFamille
  | Personne p |

  p nomFamille exists.

actions
  p fils do: [:unFils | unFils nomFamille: p nomFamille].
  p fils areModified.
```

V.3.8. Conclusion sur le `modified`

Nous voyons ici que l'écriture du type OPS5 est très coûteuse parce qu'elle oblige à filtrer des objets accessibles fonctionnellement. NéOpus permet, simplement en utilisant les dépendances fonctionnelles entre objets, à l'aide d'un jeu de principes d'utilisation adéquats, de réduire le nombre d'objets filtrés, et donc de gagner en rapidité d'exécution, et en concision.
Ce problème à déjà été soulevé par [Ghallab 88], et a donné lieu à des optimisations de l'algorithme Rete par la création de graphes explicitant les dépendances entre objets, de manière en à tirer profit pour limiter le nombre de jointures Rete. Ici l'algorithme Rete n'est plus mis en jeu, mais l'écriture des règles NéOpus permet de tirer parti naturellement de ces dépendances.
Cependant, comme nous l'avons vu plus haut, l'utilisation de ces dépendances est limitée au deux cas suivant :

Principe de dépendance fonctionnelle

On pourra utiliser le principe "Toute expression", ainsi que la dépendance fonctionnelle dans une règle si :

- soit les objets dépendant fonctionnellement sont réputés ne pas pouvoir être modifiés par une règle de la base.
- soit si l'on peut assurer, par l'intermédiaire d'un `modified` (interprété comme une r-modification) que les modifications d'objets sont bien propagées vers les prémisses concernées

V.4. Conclusion provisoire

V.4.1. Méthodologie sommaire

Nous recensons ici certaines caractéristiques d'utilisation du système importantes.

V.4.1.1. Le "mode démon"

Le mécanisme d'inférence de NéOpus ne contient pas de notion d'attachement procédural, ou de démon, comme beaucoup de systèmes d'inférences (Loops, Smeci, Art, PtitLoo [Ferber 89]). Nous estimons en effet que cette notion est inconsistante dans un univers objet au sens propre, puisque tous les accès aux variables d'instance s'effectuent nécessairement par envoi de message, et ne sont pas différenciés des autres envois de messages. D'une certaine manière il n'y a que des attachements procéduraux, ou il n'y en a pas, c'est selon.
Nous pouvons simplement parler d'un "mode démon" lorsqu'un message d'accès effectue un calcul non trivial, au lien de lire ou de modifier simplement une variable d'instance.
Nous appelerons en particulier mode démon les bases de règles activées par une méthode d'accès, comme c'est le cas dans divers exemples exposés ici (Cf VII. 5, VII. 7.4).[38]

V.4.1.2. Comment récupérer le résultat des inférences

Le problème de la récupération du résultat des inférences est réglé implicitement par effets de bord sur les objets. Il n'y a donc pas de mécanisme particulier pour récupérer le résultat de l'activation d'une base de règles. Une des raisons principale est que l'action d'une base de règles peut être de modifier *plusieurs* objets. Dans ce cas il est difficile d'envisager un protocole unifié d'échange.

V.4.1.3. Inflation des prémisses

La mécanique Rete implique une inflation des prémisses pour éviter les bouclages. Nous ne proposons pas de remède à ce problème (bien que certains systèmes proposent des modifications à Rete pour le résoudre [Shor & al. 86]).

V.4.1.4. Le modified et ses conséquences

[38] Pour faciliter cet emploi de NéOpus, nous utilisons la méthode `ruleBaseNamed:` définie dans `Opus` qui permet de récupérer une base de règles d'après son nom, et ce afin d'éviter les problèmes de compilation.

L'utilisation du `modified` est réglementée par deux principes incompatibles d'utilisation du modified (V.3.4 ou V.3.5).

V.4.1.5. Utilisation des métaclasses des bases de règles

La programmation par métaclasse induite par Smalltalk est transposée aux bases de règles. Les métaclasses de celles-ci sont utilisées extensivement pour écrire des méthodes de lancement, et pour porter toutes sortes d'informations de contrôle (Cf. Chapitre VI, comme les classes de conflict set, les classes d'évaluateurs par défaut, les conditions d'arrêt), et diverses méthodes du ressort des bases de règles.

V.4.1.6. L'héritage de bases de règles

Les bases de règles peuvent être construites comme raffinement de base de règles préexistantes grâce à la notion d'HBR.

V.4.1.7. Le typage naturel

Le typage naturel permet de donner aux règles un caractère abstrait et de limiter leur nombre. Utilisé conjointement à l'héritage de bases de règles, il induit une programmation multi-niveaux, supporté par un environnement de programmation adapté.

VI. Le Contrôle

Avant propos

Nous abordons dans ce chapitre le problème du contrôle en NéOpus. Après avoir décrit l'architecture procédurale du contrôle, nous en montrons les limites. Nous introduisons alors une seconde architecture qualifiée de déclarative, qui consiste à spécifier le contrôle à l'aide de règles NéOpus. Pour ce faire, un nouvel objet appelé Evaluateur est introduit, qui sert de support d'expression de méta-règles.
Les deux architectures sont complémentaires, toute base de règles pouvant être contrôlée soit procéduralement, soit déclarativement, à l'aide d'une autre base de règles.
L'architecture procédurale de contrôle est une application directe de la programmation par objets à la spécification procédurale du contrôle, l'architecture déclarative, elle, est une application de notre procédé de représentation, consistant à réifier les objets en vue d'un discours par règles.

VI.1. Le contrôle procédural en marche avant

Les méthodes de raisonnement en marche avant font apparaître rapidement des problèmes de contrôle, inhérents au mécanisme lui-même : à un moment donné du cycle d'inférence, plusieurs règles peuvent être applicables, et le choix de l'une d'entre elles conditionne toute la suite du raisonnement. La majorité des systèmes actuels utilise un mode de contrôle procédural, c'est à dire décrit par l'implémentation du moteur d'inférence. Le problème du contrôle n'est d'ailleurs pas limité aux seuls systèmes à base de règles et est commun à beaucoup de systèmes d'Intelligence Artificielle. [Hayes-Roth 85] le définit de manière générale comme :

"which of its potential actions should an AI system perform at each point in the problem-solving process".

[Laurent 83] propose une présentation abstraite du problème du contrôle d'un moteur d'inférence et une typologie déterminée par la manière dont le moteur effectue ses choix. En particulier il distingue trois catégories de critères de choix : basés sur les *objets* filtrant les règles (O), sur les *actions* des règles (A), et sur les deux (O-A). Il existe en effet de nombreux critères de choix possibles pour sélectionner une règle déclenchable parmi un ensemble de possibilités. Notre expérience de programmation de bases de règles nous a convaincu qu'il ne peut y avoir de stratégie "fixe" et générale, et que toutes les bases de connaissances nécessitent un type de contrôle particulier. En bref, un moteur d'inférence "moderne" doit être capable de prendre en compte tous les types de critères raisonnables, et ne pas se limiter à un jeu de stratégies fixe (comme dans OPS5 par exemple).
Nous allons dans un premier temps nous placer dans ce cadre de la définition d'une structure de contrôle procédurale la plus générale possible, en suivant bien sûr notre orientation objet, qui va nous conduire à une solution basée sur l'héritage des métaclasses.

VI.2. Une architecture procédurale et objet de contrôle

VI.2.1. Les bases de règles sont des classes

Comme nous l'avons vu au chapitre III, les bases de règles sont représentées en Opus par des classes, donc des objets Smalltalk. Ces classes, abstraites, portent toutes les informations nécessaires au filtrage des objets par les règles, et à la gestion (compilation, héritage) de ces règles. Notamment, les bases de règles connaissent (au sens Smalltalk) :

- le conflict set (les règles déclenchables avec leur jeux d'instanciation),
- le réseau et les nœuds Rete, contenant tous les filtrages partiels pour chacune des prémisses,
- un contexte : l'ensemble des objets initiaux devant être considérés par la base de règles à son évaluation,
- une classe dynamique, génératrice des tokens propagés dans le réseau.

L'activation d'une base de règles est une procédure propre à celle-ci, qui contient toutes les informations nécessaires à cette activation. La programmation par objet nous conduit donc à la représenter comme une méthode de la base de règles *en tant qu'objet*. Les méthodes de contrôle sont donc implémentées au niveau des métaclasses des bases de règles, et donc, grâce à l'héritage parallèle des métaclasses, au niveau de `OpusRuleSet class`.

VI.2.2. L'évaluation d'une bases de règles est un processus décomposable

VI.2.2.1. Trois étapes standard pour l'évaluation

Le lancement de l'évaluation d'une base de règles est décrit en Opus sous forme de méthodes, écrite dans la métaclasse `OpusRuleSet class`. La méthode de lancement `execute` est décomposée en trois appels successifs aux méthodes : `sendInstances`, `sature` et `return`, qui décrivent le cycle standard d'évaluation d'une base de règles :

envoi des instances initiales dans le réseau Rete,
boucle tant que conflict set n'est pas vide,
arrêt par retour éventuel d'un résultat.

```
!OpusRuleSet class methodsFor: 'controle'!

execute
  ^self sendInstances; sature; return

sendInstances
"envoie toutes les instances concernées dans mon reseau"
  self dispensers do: [:d| d sendInstancesInto: name].

sature
  [self stopCondition] whileFalse: [self applyRule]

stopCondition
  ^self basicStopCondition or: [self publicStopCondition]

basicStopCondition
  ^conflictSet isEmpty

publicStopCondition
  ^false

applyRule
  self conflictSet declencheDefault

return
"rien par défaut"

defaultConflictSetClass
  ^OpusConflictSet
```

Cette décomposition en trois étapes est nécessaire pour permettre grâce à l'héritage de classe la redéfinition par les sous-classes (sous-bases) d'une ou plusieurs de ces trois étapes.

VI.2.2.1.1. Une décomposition héritable du contrôle

On peut de cette manière traduire différentes stratégies de contrôle en utilisant l'héritage dans les sous-classes d'`OpusRuleSet` et de `OpusConflictSet`.

On suit en cela de très près le mécanisme des contrôleurs de fenêtres Smalltalk, à la base du système de programmation d'interface MVC [Goldberg 84, Krasner-Pope 88]. En effet, la classe Smalltalk `Controller` définit de façon générique le comportement de tous les contrôleurs du système de la manière suivante (envoi du message `startUp` au contrôleur) :

```
!Controller class methodsFor: 'control'!

startUp
  self controlInitialize.
  self controlLoop.
  self controlTerminate

controlLoop
  [self isControlActive] whileTrue: [self controlActivity]

controlActivity
  self controlToNextLevel

controlToNextLevel
  | aView |
  aView _ view subViewWantingControl.
  aView ~~ nil ifTrue: [aView controller startUp]

isControlActive
^(view containsPoint: sensor cursorPoint) &
  sensor blueButtonPressed not

controlInitialize
"rien par defaut"

controlTerminate
"rien par defaut"
```

Les différentes sous classes de `Controller` (comme `StandardSystemController`, `ScrollController`, `NoController`), redéfinissent alors uniquement la ou les méthodes nécessaires.
La décomposition de la méthode `startUp` est qualifiée d'*héritable*. Sa validité est démontrée par l'existence d'une librairie de classes de contrôleurs, et le bon fonctionnement du paradigme MVC, lui même démontré par l'existence de l'environnement Smalltalk-80, entièrement basé sur ce principe [Goldberg 83].

A la manière de la décomposition des contrôleurs MVC, notre décomposition est, elle aussi, héritable : chacune des trois méthodes va pouvoir être redéfinie par des sous-classes d'`OpusRuleSet`, pour implémenter des comportements particuliers.

VI.2.2.1.2. Exemples de redéfinitions

On peut ainsi fabriquer autant de sous classes d'`OpusRuleSet` que l'on veut, associées à des conflict set(s) différents, pour définir des stratégies de déclenchement spécifiques.
Les méthodes implémentant le cycle de base peuvent donc être localement redéfinies, afin de modifier la méthode de saturation, la condition d'arrêt, la méthode de retour, ou la classe de conflict set.

A/Exemple de redéfinition de la méthode `stopCondition`

La méthode définissant la condition d'arrêt comporte deux tests : un test "basic", définit par la méthode `basicStopCondition` consistant à tester que le conflit set

n'est pas vide. Ce test est obligatoire, quelle que soit la stratégie adoptée : si aucune règle n'est déclenchable, le moteur doit s'arrêter. Mais la condition d'arrêt peut être rafinée, dans une sous-base, en redéfinissant la méthode `publicStopCondition`, qui par défaut rend faux.
Par exemple, on peut vouloir arrêter un déclenchement par clic de la souris. Dans ce cas, il suffit de redéfinir la méthode `publicStopCondition` dans la métaclasse de la base de règles :

```
!MaBaseDeRegles class methodsFor: 'arret'!

publicStopCondition
  ^Sensor anyButtonPressed
```

B/ Exemple de redéfinition de conflict set

Par défaut, la classe `OpusConflictSet` déclenche la <u>première</u> règle déclenchable (et donc parcourt le graphe d'état en profondeur d'abord, puisque les règles sont ajoutées, par défaut, au début (Cf. III.3.7.3)). Définir une stratégie de déclenchement différente peut donc se faire en définissant une nouvelle sous-classe de `OpusConflictSet` qui redéfinit la méthode de déclenchement `declencheDefault`.
Par exemple, pour déclencher une règle tirée au hasard, en utilisant un générateur de nombre aléatoire, on pourra définir la classe `OpusConflictSetRandom` comme suit :

```
OpusConflictSet subclass: #OpusConflictSetRandom
  classVariableName: 'RandomGenerator'

!OpusConflictSetRandom class methods for: 'initialization'!

initialize
  RandomGenerator <- Random new

!OpusConflictSetRandom methods for: 'declenchement'!

applyRule
  "on declenche effectivement au hasard"
  | uneRegle |
  uneRegle <- dicoRules at: ((RandomGenerator next) * dicoRules
size) rounded.
  self declenche: uneRegle
```

Et on associera cette nouvelle classe de conflict set à la base de règles considérée, via la méthode `defaultConflictClass`.

```
OpusConflictSet subclass: #MaBaseDeRegles
  category: 'bases de regles'

!MaBaseDeRegles class methods for: 'conflict set'!

defaultConflictSetClass
  ^OpusConflictSetRandom
```

Une série de conflict sets sont facilement implémentés de cette manière, comme `ConflictSetFIFO`, `ConflictSetLIFO`, dont les noms parlent d'eux-mêmes. Ces conflict sets permettent de définir des stratégies de déclenchement simples. Des stratégies plus complexes seront décrites par l'architecture déclarative de contrôle.
Notons ici que redéfinir la méthode `defaultConflictSetClass` ne suffit pas à changer la stratégie de déclenchement d'une base de règles : il faut aussi effectivement recréer son conflict set. Ceci se fait par l'envoi du message `initConflictSet` à la base de règles.

C/ Changer la méthode `sature`

La méthode de saturation effectue une boucle tant que la condition d'arrêt n'est pas remplie. Cette méthode peut être redéfinie pour raffiner cette boucle, par exemple pour enregister un historique, déclencher les règles suivant un mode

D/ Changer la méthode `sendInstances`

Cette méthode peut être redéfinie pour considérer d'autres types de contextes, par exemple pour les classes mémorisant leurs instances d'une manière particulière (les mémo-classes).

VI.2.3. Critique de l'architecture procédurale

VI.2.3.1. Critique de notre solution

Cette architecture est très pratique pour redéfinir localement de *petites variations* autour du cycle de base.
Mais la décomposition du contrôle entre les base de règles/ conflict set / réseau Rete rend ces objets fortement interconnectés. La création de classes génériques de conflict set(s) est ainsi très limitée. Par exemple, la gestion d'un agenda de paquets de règles ou de priorités y est très maladroite.

- trop grande interconnectivité entre conflict set et base de règles (à la différence des contrôleurs MVC qui ont toute l'information),
- héritage de classe mal adapté
- pas de possibilité de définir des contrôle de manière générale.

VI.2.3.2. Solution originale

Atkinson&Laursen proposent initialement d'ajouter un troisième objet, l'*interprète*, chargé de toutes les opérations d'interprétation de la base de règles. Bien que cette architecture n'est pas été explorée, il est facile de lui porter la même critique concernant la généricité des contrôles. Dans notre cadre, on peut dire que le contrôle est un processus qui est :

- trop compliqué pour pouvoir être modélisé par un seul objet (la base de règles ou interprète à part entière).
- qui s'accomode mal de la diversité des objets mis en jeu, et d'un déroulement séquentiel des opérations.
- doit pouvoir être suffisamment souple pour permettre simplement des redéfinitions *locales* du processus

VI.3. L'architecture déclarative

Ayant reconnu que le contrôle d'une base de règles en marche avant était un processus crucial[39], non trivial, et pouvant nécessiter des connaissances à la fois générales (stratégies générales de choix d'une règle) et spécifiques au domaine d'application de la base de règles (comme la notion de but en OPS5), nous en sommes naturellement parvenu à tenter de donner au contrôle un statut de première classe.

L'idée que l'évaluation d'une base de règles est un processus qui, lui aussi, utilise des connaissances (ou des métaconnaissances [Pitrat 90]), nous conduit alors chercher une représentation déclarative du contrôle.
De plus le caractère profondément uniforme de l'environnement Smalltalk (tout est objet, y compris les classes), allié à la possibilité d'écrire des règles Opus sur tout objet de l'environnement, conduisent naturellement à envisager le problème du contrôle de manière déclarative : en écrivant des règles (appelées méta-règles) portant sur les objets particuliers que sont les bases de règles.
Enfin, et c'est là notre objectif majeur, à la suite des échecs de l'architecture procédurale, une architecture déclarative semble être la seule, d'un point de vue strictement "objet", à pouvoir offrir la possibilité de définir des contrôles de manière générale.

Faire un état de l'Art sur le *contrôle déclaratif* des bases de connaissances est une tâche ardue. Un certain nombre de travaux mettent en lumière la *nécessité* de disposer de moyens déclaratifs de contrôle [Batali 88], [Clancey 83b], [Pitrat 90]; d'autres

[39] crucial du latin crux, crucis : la croix, pris ici au sens "croisée de chemins". Littéralement, le choix d' une règle détermine de façon irréversible le chemin dans le graphe d'états.

proposent des cadres généraux de représentation [Georgeff 82], comme les blackboards [Engelmore & al. 88], [Hayes-Roth 85], [Bachimont 90]. Mais le problème de la représentation déclarative du contrôle est encore jeune en Intelligence Artificielle, et la plupart des systèmes proposent des solutions adaptées au problème à résoudre [Davis 80], [Pinson 87]. L'école française en la matière, initiée par J. Pitrat, se penche sur des problèmes de *réflexivité* (comment appliquer des connaissances à elles-mêmes [Pitrat90], [Dormoy 86, 87], [Fouet 86, 87] ; des problèmes de planification [Hilario 88], d'explication intelligente [Jimenez 90] ; d'espionnage de bases de connaissances [Kornmann 90], [Parchemal 88]).

Nous proposons donc une deuxième architecture, qualifiée de déclarative, qui permet de définir le contrôle à l'aide de règles NéOpus. Nous introduisons un objet intermédiaire, appelé évaluateur, qui nous permettra d'écrire des règles manipulant les objets particuliers que sont les bases de règles. Ces objets évaluateurs sont ensuite déclinés pour prendre en compte divers types de contrôles.

Nous allons pour cela mettre en applications les principes obtenus lors de notre expérience pratique de NéOpus, en écrivant des connaissances portant sur les bases de règles.

VI.3.1. Une architecture par substitution

Nous distinguons dans les systèmes proposant une architecture déclarative de contrôle, plusieurs types d'architecture qui se différencient par la place des méta-règles dans le cycle d'évaluation.

par priorité

Dans Snark, les méta règles sont au même niveau que les règles, mais sont considérées comme prioritaires par rapport aux autres. Ainsi il n'y a pas véritablement d'*architecture*, puisque le cycle d'évaluation reste inchangé.

par insertion

Une architecture classique (notamment celle proposée par [Atkinson&Laursen87]) consiste à greffer dans le cycle standard d'évaluation, au moment du choix de la règle à déclencher, une phase explicite d'appel à une base particulière de méta-règles. Le contrôle est ainsi toujours défini de manière procédurale, mais inclut une partie déclarative.

par substitution

C'est la voie que nous choisissons. Elle consiste à substituer totalement à la procédure de contrôle l'activation d'une base de méta-règles, qui est chargée de *tout* le cycle, et pas seulement une partie.

VI.3.2. Les bases de règles ne portent pas assez d'informations pour représenter leur état d'évaluation

Un première tentative d'écriture de règles décrivant l'évaluation d'une (autre) base de règles[40] se heurte tout de suite à un problème de représentation : comment représenter la notion de statut, ou d'état d'évaluation (où en est-on à un moment donné), et la séquencialité des trois opérations ?.
Il est clair que les bases de règles ne portent pas assez d'informations à ce sujet. Dans une implémentation procédurale, la séquencialité des opérations est naturellement traduite par la séquencialité du langage Smalltalk (la méthode `execute`), quant à la notion de statut, elle n'est pas nécessaire dans un cadre procédural : le programme n'a pas besoin de savoir où il en est.

VI.3.2.1. Intérêt de la réification du contrôle

L'intérêt de représenter le contrôle comme un objet est multiple :

- La représentation du contrôle est dissociée de la base de règles, ce qui permettra de donner à celui-ci un caractère "branchable", et donc d'écrire des bases de méta-règles indépendantes et générales; et d'autre part d'écrire des bases de règles indépendamment de leurs processus d'évaluation,

- Il est possible d'envisager *plusieurs* évaluateurs concurrents pour une même exécution. La gestion de ces divers évaluateurs va alors être définie au niveau de la base de méta-règles, et permettra entre autres de représenter des raisonnements complexes, faisant intervenir des objets intermédiaires créés dynamiquement (Cf Chapitre VI.5),

- Des objets de contrôle de plus en plus complexes peuvent être définis, simplement en utilisant l'héritage de classes. En particulier les notions d'agendas, de buts récursifs en partie prémisses ou de stratégies trouveront ici une véritable définition en termes d'objet.

VI.3.2.2. Les évaluateurs

Afin de représenter toutes les informations relatives à l'évaluation d'une base de règles, nous introduisons la notion d'Evaluateur, objet représentant un état d'évaluation d'une base de règles.
Cet objet a comme structure minimale la base de règles dont il représente l'évaluation, un statut (un symbole) représentant l'étape courante, et un contexte,

[40] ou, comme dirait Pitrat, une *déprocéduralisation* du contrôle

représentant le contexte d'initialisation de la base de règles, et une condition d'arrêt représentée dans un premier temps par un bloc Smalltalk.

```
Object subclass: Evaluateur
  instanceVariableNames: 'ruleBase status contexte stopCondition'
```

VI.3.2.2.1. Le statut d'un évaluateur

On définit bien sûr les méthodes d'accès aux variables d'instances qui seront utilisées dans les prémisses des méta-règles. On peut même s'offrir le luxe de méthodes plus génériques, (copiées sur le comportement des processus) telles que `resume` ou `terminate`:

```
statut
  ^statut

statut: unSymbole
  statut <- unSymbole

resume
  self statut: #actif

terminate
  self statut: #inactif

isActive
  ^(statut = #inactive) not
```

VI.3.2.2.2. La condition d'arrêt

```
!Evaluateur methodsFor: 'evaluation'!

reussi
"si le but est un bloc, on lui envoie le message value, sinon on l'evalue
symboliquement.
Les messages inconnus ou provoquant une erreur se traduisent par un
resultat false, pour pouvoir prendre en compte les assertions non
Smalltalk"

  stopCondition isNil
          ifTrue: [stopCondition _ [ruleBase conflictSet isEmpty]].
  (stopCondition isKindOf: Assertion)
          ifTrue: [^self errorSignal
                       handle: [:ex | ex returnWith: false]
                       do: [stopCondition evaluate]].
  ^stopCondition value!

nonReussi
  ^self reussi not!
```

VI.3.2.2.4. Création des évaluateurs

VI.3.2.2.4.1. Création standard

La création d'un évaluateur se fait par la méthode générique `makeEvaluateurWithContext:stopCondition:` envoyée à la base de règles.
Cette méthode renvoie une instance d'évaluateur correctement initialisée, et prêt à fonctionner en :

- créant une nouvelle instance d'évaluateur, d'après la méthode `defaultEvaluatorClass`,
- initialisant cet évaluateur avec la condition d'arrêt spécifiée, qui doit être un bloc Smalltalk,
- initialisant cet évaluateur avec le contexte spécifié (un dictionnaire),
- initialisant l'évaluateur avec un statut à `#init`.

```
!OpusRuleSet class methodsFor: 'evaluateurs'!

makeEvaluateurWithContext: aContext stop: aBlockOrAnAssertion
    ^self defaultEvaluatorClass
          fromRuleBase: self
          status: #init
          stopCondition: aBlockOrAnAssertion
          context: aContext!
```

VI.3.2.2.4.2. Création simplifiée

Des messages de création simplifiés permettent la création d'évaluateurs canoniques. La méthode `makeEvaluator` rend un évaluateur dont la condition d'arrêt est la vacuité du conflictSet et le contexte, le contexte de la base de règles :

```
makeEvaluateur
  ^self makeEvaluateurWithMessage: conflictSet isEmpty]

makeEvaluateurWithMessage: aMessage
  ^self makeEvaluateurWithMessage: aMessage context: context
```

VI.3.2.2.4.3. La CRE est determinée par la méta-base

Il faut noter que la classe d'évaluateur dépend uniquement du *processus d'évaluation*, et non pas de la *base de règles* évaluée. Ainsi, la méthode `defaultEvaluatorClass` va chercher cette information non pas au niveau de la base de règles, mais au niveau de la méta-base, si elle existe, par l'intermédiaire de la méthode `requiredEvaluatorClass`.
Nous introduisons donc la notion de classe d'évaluateur requise pour une base de méta-règles :

Définition de la CRE
La classe d'évaluateur requise d'une méta-base (notée CRE) est la classe d'évaluateur déclarée par ses méta-règles. Elle est déclarée explicitement par la méthode `requiredEvaluatorClass`

Ainsi, on définira la méthode `requiredEvaluatorClass` dans chaque base de méta-règles, qui sera responsable d'une déclaration conforme aux déclarations de variables de ses méta-règles.

Par défaut, la classe d'évaluateur pour la base de méta-règles standard `DefaultMeta` est `Evaluateur`. Les sous-classes de `DefaultMeta` redéfiniront chacune leur propre classe d'évaluateur.

```
!OpusRuleSet class methodsFor: 'evaluateurs'!

defaultEvaluatorClass
  "on va chercher cette information dans la metabase si elle existe"

  metaBase isNil ifTrue: [^Evaluateur].
  ^metaBase requiredEvaluatorClass!

requiredEvaluatorClass
  "le classe d'evaluateurs utilise dans la base de regles. redefinit
dans les sous classes de DefaultMeta"

  ^Evaluateur
```

VI.3.2.2.4.4. Exemple

Créer un évaluateur pour la base de règles `FiboRules` se fait par la transmission suivante :

```
  FiboRules makeEvaluateur.
```

On peut aussi créer un évaluateur avec un contete particulier et/ou une condition d'arrêt spécifique, par exemple :

```
|e|
  e <- FiboRules makeEvaluateur.
  e stopCondition: [Sensor anyButtonPressed].
  e context: (Fibo new arg: 10).
```

VI.3.2.2.5. Un parallèle entre évaluateurs et processus

Le parallèle entre évaluation (méthode `execute`) et contrôle au sens MVC (méthode `startUp`) évoqué plus haut prend ici un éclairage nouveau : une fois réifiée, la notion d'évaluateur se rapproche de celle de processus, en particulier, par le caractère indépendant de l'évaluateur (objet à part entière), et concurrent (de la même manière que plusieurs processus cohabitent dans l'environnement, ici plusieurs évaluateurs différents pourront coexister pour une même base de règles).

VI.3.2.3. Méta-règles et Méta-bases

Une fois la notion d'évaluateur introduite, il ne reste plus qu'a définir la manière dont bases de règles, évaluateurs, et méta-règles vont interagir.

VI.3.2.3.1. Lien structurel entre bases et méta-bases

Afin de structurer l'interaction entre bases de règles et bases de méta-règles, on ajoute une variable d'instance supplémentaire pour les bases de règles, qui pointera sur une base de méta-règles chargée de son évaluation (méta-base) :

```
OpusRuleSet class
  instanceVariableNames: 'dynamicClass conflictSet context metaBase'
```

L 'association entre une base de règles et une base de méta-règles ne requiert alors qu'une modification de cette variable d'instance (simple clic dans le Tableau de bord Opus).

En revanche, aucun lien inverse n'est défini : une base de méta-règles ne connait pas directement la base de règles qu'elle contrôle.
En effet, établir un lien structurel inverse de la méta-base vers la base ne suffit plus puisque les informations sont maintenant contenues dans les objets évaluateurs et non pas dans la base elle-même. Le lien inverse est défini de manière opératoire :

VI.3.2.3.2. Lien opératoire

Le lien entre méta base et base va se faire par l'intermédiaire du contexte de la méta-base. Informer une méta base qu'elle contrôle une base se traduira par la présence, dans le contexte de la méta-base, d'un évaluateur de la base. Cette initialisation du contexte se fera au moment de l'exécution de la base.

VI.3.2.3.3. Nouveau protocole d'évaluation

L'évaluation d'une base de règles est donc réécrite, de manière à ce que l'évaluation de la base de règles soit entièrement gérée par la méta-base, si elle existe, et par défaut de manière procédurale.

VI.3.2.3.3.1. Redéfinition de la méthode `execute`

Lé méthode `execute` est donc redéfinie, et, suivant l'existence de la méta-base dirige l'action vers le contrôle procédural ou le contrôle déclaratif.
Nous ne commentons pas ici l'escalade des méthodes `execute`, `executeUntil:`, `executeWithContext:until:` , traditionnelle, qui permet de factoriser le code de manière efficace.
Notons simplement que le contrôle est assuré par la méthode `executeWithEvaluateur:`, recevant comme argument un évaluateur correctement instancié, grâce aux méthodes de création d'évaluateur définies ci-dessus.

```
execute
  ^self executeWithContext: context!

executeUntil: unBlock
  ^self executeWithContext: context until: unBlock!

executeWithAssertion: anAssertion
  ^self executeWithContext: context until: anAssertion!

executeWithContext: aDictionary
  ^self executeWithContext: aDictionary until: [conflictSet
isEmpty]!

executeWithContext: aContext until: aBlockOrAnAssertion
  ^self executeWithEvaluateur:
        (self makeEvaluateurWithContext: aContext
              stop: aBlockOrAnAssertion)
```

VI.3.2.3.3.2. Exécution déclarative

La méthode d'exécution va donc opérer comme suit :

```
executeWithEvaluateur: unEvaluateur
  metaBase isNil ifTrue:
        [^self basicExecuteWithEvaluateur: unEvaluateur].
  context _ unEvaluateur context.
  metaBase emptyAllNilClassesInContext.
  metaBase addInContext: unEvaluateur; addInContext: conflictSet.
  metaBase execute.
  ^self return!
```

C'est à dire :

- redirection vers un contrôle procédural en cas d'absence de méta-base par appel à la méthode `basicExecuteWithEvaluateur:`.
Cette méthode n'est autre que l'ancienne méthode `execute` rebaptisée pour l'occasion, et prenant un évaluateur en argument, par souci d'homogénéité.

```
basicExecuteWithEvaluateur: unEvaluateur
  context _ unEvaluateur context.
  self sendInstances.
  self satureUntil: unEvaluateur stopCondition.
  "self efface: s at: p."
  ^self return!
!OpusRuleSet class methodsFor: 'executing'!
```

- dans le cas d'un contrôle par méta-base :

1- mise à jour du contexte de la base de règles par le contexte défini par l'évaluateur.
2- Exécution de la méta-base :
- Etablissement du *lien opératoire* :
Initialisation du contexte de la méta-base. L'évaluateur est ajouté à ce contexte, ainsi que le conflict set de la base de règles.

- Exécution proprement dite de la méta-base par envoi du message `execute`.

VI.3.2.4. Une base de méta règles standard : `DefaultMeta`

Maintenant que les bases de règles ont leurs évaluateurs, il est possible d'écrire une première base de méta-règles, chargée de l'évaluation standard d'une base de règles. Cette base, appelée `DefaultMeta`, comprends cinq règles, et reproduit le même comportement que la méthode `execute`, mais bien sûr de façon déclarative. Les règles sont organisées en trois protocoles, reproduisant les trois étapes de la méthode `execute` :.
Les deux objets manipulés par cette base de règles sont l'évaluateur et le conflict set de la base de règles.

VI.3.2.4.1. Définition de `DefaultMeta`

`DefaultMeta` est une simple sous-classe de `OpusRuleSet` :

```
OpusRuleSet subbase: #DefaultMeta
  globalObjects: ''
  category: 'OPUS-rules'!
```

Mais la métaclasse `DefaultMeta class` différencie celle-ci des bases de règles standard :

- en définissant une classe d'évaluateur requis (`requiredEvaluatorClass`),
- en se proclamant méta (`isMetaBase`),
- en anonçant délibérément un typage naturel (`initTypage`).

```
!DefaultMeta class methodsFor: 'evaluateur'!

requiredEvaluatorClass
  ^Evaluateur

!DefaultMeta class methodsFor: 'testing'!

isMetaBase
  ^true

!DefaultMeta class methodsFor: 'typage des variables'!

initTypage
  typing _ true
```

VI.3.2.4.2. Les règles

Les trois étapes sont gérées par l'intermédiaire de la variable d'instance `status` de l'évaluateur, qui passe par les trois valeurs : `#init`, `#loop`, `#end`. L'évaluateur est initialement crée avec un status à `#init`.

VI.3.2.4.2.1. Initialisation

L'initialisation consiste à envoyer dans le réseau Rete les objets spécifiés par le contexte.
Deux règles prennent en charge l'initialisation d'une base de règles, suivant la présence ou pas de contexte pour l'évaluateur :

```
!DefaultMeta methodsFor: 'init'!

initNoObjects
"un evaluateur qui n'a pas d'objets nouveaux, passe directement a loop"
  | Evaluateur e |
    e status = #init.
    e context exists not.
actions
    e status: #loop.
    e modified!

initObjects
"on envoie les nouveaux objets dans le reseau Rete de la base de regles.
l'evaluateur change de status, et le conflict set est modifie (en general))"
  | Evaluateur e |
    e status = #init.
    e context exists.
actions
    e sendContext.
    e status: #loop.
    e modified.
    e conflictSet modified.
"car sendContext a pu rendre des regles declenchables"
```

Il faut remarquer ici l'usage de la r-modification : l'évaluation du message `sendContext` envoyé à l'évaluateur, aura (certainement) pour effet d'ajouter des règles dans le conflict set, et donc de r-modifier celui-ci.

VI.3.2.4.2.2. Saturation

La saturation consiste à appliquer une règle quand le conflict set n'est pas vide, sinon à s'arréter. Deux règles prennent en charge la saturation d'une base de règles de la manière suivante :

```
!DefaultMeta methodsFor: 'loop'!

loop1
"si l'evaluateur n'est pas reussi, on declenche la premiere regle du conflict set"
  | Evaluateur e. OpusConflictSet c |
    e status = #loop.
    c _ e ruleBase conflictSet.
    e reussi not.
    c notEmpty.
actions
    c declencheDefault.
    c modified.
"car l'application d'une regle peut rendre la condition d'arret vraie"
    e modified!

loopEnd
"l'evaluateur a reussi, on change de status"
  | Evaluateur e |
    e status = #loop.
    e reussi.
actions
    e status: #end.
    e modified.
```

VI.3.2.4.2.3. Arrêt

```
!DefaultMeta methodsFor: 'end'!

end
"c'est fini"
  | Evaluateur e |
    e status = #end.
actions
    e suspend.
```

Exemple : évaluation de la base `FiboRules` avec `DefaultMeta` :

. Assignation de la metaBase de `FiboRules` à `DefaultMeta` dans le browser.
. La méthode de lancement est strictement la même que sans la méta-base.

VI.3.2.4.3. Une autre écriture, utilisant la notion de r-modification

Dans la base de règles `DefaultMeta`, le conflict set est déclaré une fois, dans la règle `loop1`. Le déclenchement d'une règle a pour effet de modifier celui-ci.

On pourrait aussi écrire cette base de règles d'une autre manière, en supprimant simplement les conflict set du discours, en ne gardant que les évaluateurs, et en utilisant la notion de r-modification :

```
!DefaultMeta methodsFor: 'loop'!

loop1
"si l'evaluateur n'est pas reussi, on declenche la premiere regle du conflict set"
  | Evaluateur e. Local c|
    e status = #loop.
    c <- e ruleBase conflictSet.
    e reussi not.
    c notEmpty.
actions
    c declencheDefault.
    e modified
```

Dans ce cas, le lien opératoire entre méta-base et base se simplifie puisque seul l'évaluateur doit être ajouté au contexte de la méta-base. C'est une voie que nous n'avons pas explorée, bien qu'elle est paraisse intéressante, puisqu'elle réduit le nombre d'objets filtrés, et donc uniformise le discours.

VI.4. Déclinaisons des évaluations

Le mécanisme de contrôle présenté plus haut peut maintenant se décliner pour prendre en compte des évaluations moins triviales que celle définie par `DefaultMeta`. Ces déclinaisons vont consister à écrire de nouvelles bases de méta-règles, et à enrichir la structure des évaluateurs; ces deux mouvement étant n'étant pas toujours simultanés.
On différenciera par ailleurs deux types de déclinaisons : abstraite et non abstraite (?) :

Déclinaison abstraite: Fabrication de bases de règles et d'évaluateurs à usage général, implémentant un mécanisme de contrôle général (ex: trace, évaluateurs récursifs) : Base de règles abstraite (Cf. les trois dimensions d'abstraction pour les bases de règles).

Déclinaison par spécialisation: Fabrication d'une base de méta-règles associée à une base de règles particulière : On parlera alors de programmation multi-niveau. Représentation des méta connaissances liées à un domaine particulier (par exemple la géométrie, les singes)

Ces déclinaisons seront décrites par référence à la base de règles `DefaultMeta`, comprenant les cinq règles précédemment décrites.

VI.4.1. Un critère de compatibilité

La notion de classe d'évaluateur requise (Cf plus haut VI.3.2.2.4.3) permet d'exprimer pour une méta-base une contrainte sur la classe d'évaluateur utilisée. Les méta-bases étant définies en utilisant extensivement le typage naturel et l'héritage de bases de règles, on retrouve alors le problème de compatibilité énoncé au (IV.1.7.7). Ici ce problème s'énonce plus simplement puisque les méta-règles ne filtrent qu'un seul type d'objets (les évaluateurs). La contrainte de compatibilité peut s'énoncer comme suit :

Critère de compatibilité de CRE

la CRE d'une méta-base doit être une sous-classe de la CRE de sa super base, et ce, récursivement jusqu'à `DefaultMeta`.

Il faut rappeler que cette de contrainte de compatibilité n'est en rien *obligatoire*. Aucune vérification n'est faite par le système est toute classe est autorisée. Elle exprime simplement un critère raisonnable d'utilisation, en dehors duquel rien n'est plus garanti.

VI.4.2. Trace

Le problème de la trace d'un raisonnement par règles est un problème complexe et largement étudié dans le monde de l'Intelligence Artificielle. Sans plonger dans cet univers, nous pouvons simplement implémenter un contrôle standard fournissant une trace du déclenchement des règles, en créant une méta-base particulière, `TraceMeta`, qui se chargera, au moment du déclenchement d'une règle, de le signaler explicitement d'une manière ou d'une autre (affichage dans une fenêtre spécialisée, ou dans un fichier par exemple).

La seule règle à modifier par rapport à `DefaultMeta` est donc la règle qui a pour effet de déclencher une règle, c'est à dire la règle `loop1`. Cette règle doit maintenant inclure un message de notification, par exemple sous forme d'affichage dans la fenêtre `Transcript` :

```
!TraceMeta methodsFor: 'loop'!

loop1
"si l'evaluateur n'est pas reussi, on declenche la premiere regle du conflict set"
  | Evaluateur e. OpusConflictSet c |
    e status = #loop.
    c == e ruleBase conflictSet.
    e reussi not.
    c notEmpty.
actions [41]
    | premiere |
    premiere <- c premiereRegle.
    Transcript show: 'je declenche la règle ',premiere  nom;cr.
    c declenche: premiere .
    c modified.
```

Le problème de la trace du raisonnement, et plus généralement de son explication a été traité dans [Alvarez 91] dans l'environnement NéOpus, à l'aide de bases de méta-règles plus complexes.

VI.4.3. Divers

Une application intéressante et amusante consiste à écrire une base de méta règles qui demande systématiquement à l'utilisateur la règle à déclencher au moment du choix. Cela permet de contrôler le raisonnement, lors de debugging par exemple.
Il suffit pour cela d'écrire une base de règles ne redéfinissant que la méta-règle `loop1`, par exemple de la manière suivante :

```
!MetaUser methodsFor: 'loop'!

loop1
"on demande a l'utilisateur la regle a declencher"
| Evaluateur e. OpusConflictSet c |
    e status = #loop.
    c == e ruleBase conflictSet.
    e reussi not.
actions
    | uneRegle |
    uneRegle _c askForRegle.
    c declenche: uneRegle.
    c modified.
```

VI.4.4. Notions de paquets de règles et d'Agenda

[41] on est alors obligé d'utiliser une variable locale, ici `premiere`, pour pouvoir afficher son nom, avant son déclenchement

L'organisation des règles en paquets de règles à l'intérieur d'une base de règles peut se gérer aussi par la création d'une base de méta-règles particulière. Une extension pratique consiste à utiliser les protocoles du Browser Smalltalk comme noms de paquets, de façon à pouvoir très facilement modifier le contenu des paquets en utilisant l'interface Smalltalk standard, sans avoir à recompiler quoi que ce soit.
Encore une fois, il suffit alors de complexifier l'Evaluateur, en lui ajoutant une structure supplémentaire, qui est l'agenda des paquets de règles à considérer, dans un certain ordre[42].

```
Evaluateur subclass: #EvaluateurAvecAgenda
  instanceVariableNames: 'agenda'
```

L'agenda est lui même un objet, de la classe Agenda, qui permet de gérer la liste des paquets et l'index du paquet courant :

```
Object subclass: #Agenda
  instanceVariableNames: 'paquets indexPaquetCourant'
```

Les méthodes d'accès suivantes sont définies ainsi que la méthode qui passe au paquet suivant (en testant que l'on est pas arrivé à la fin):

```
paquetCourant
^paquets at: indexPaquetCourant

passeAuPaquetSuivant
indexPaquetCourant = paquets size ifTrue: [^nil].
indexPaquetCourant := indexPaquetCourant + 1
```

La méta-base `MetaAgenda` représente ce type d'évaluation. Elle ne déclenchera les règles que si elles appartiennent à l'agenda courant. Cela se traduira par une méta règle du type suivant, où l'on déclenche une règle que si elle appartient au paquet courant :

[42] La notion d'agenda dans les systèmes de règles est utilisée avec plusieurs sens différents (Snark, Essaim, OPS5). Nous prenons ici une définition courante, mais les autres définitions peuvent aussi être représentées dans la même architecture.

```
!MetaAgenda methodsFor: 'loop'!

loop1
  | Evaluateur e. OpusConflictSet c |

    e status = #loop.
    c == e conflictSet.
    e agenda notNil.
    (c reglesDeProtocole: e agenda paquetCourant) exists.

actions
    c declenche: (c reglesDeProtocole: e agenda paquetCourant) first.
    c modified.
```

La gestion des paquets de fait alors par deux méta règles : une qui passera au paquet suivant, par exemple, lorsque plus aucune règle de ce paquet n'est déclenchable, et l'autre qui changera l'évaluateur de statut :

```
finAgenda
  | Evaluateur e. OpusConflictSet c |
    e status = #loop.
    c == e conflictSet.
    e agenda notNil.
    (c reglesDeProtocole: e agenda paquetCourant) exists not.
    e agenda fini.
actions
    e status: #end. e modified

nextPaquet
  | Evaluateur e. OpusConflictSet c |
    e status = #loop.
    c == e conflictSet.
    e agenda notNil.
    (c reglesDeProtocole: e agenda paquetCourant) exists not.
    e agenda nonFini.
actions
    e agenda passeAuPaquetSuivant.
    e modified
```

Exemple : les systèmes NéoSecsi et NéoGanesh (Cf. Chapitre VII).

VI.4.5. La stratégie MEA et la théorie de la fraîcheur locale

La stratégie MEA offerte originellement par OPS5 peut ici se décrire aussi très simplement grâce à une méta-règle particulière, qui choisira de déclencher la règle dont les faits sont les plus frais (pour simplifier). Pour cela on introduit la notion de

fraîcheur pour les objets Smalltalk. Puis nous allons écrire une méta-règle permettant de prendre en compte cette notion.

VI.4.5.1. La fraîcheur et la classe `OPS5Compatibility`

Comme tous les objets Smalltalk sont susceptibles d'être filtrés par les règles, et qu'il n'est pas possible de recompiler la classe `Object` (pour ajouter en l'occurrence un attribut `timeTag`[43]), nous avons choisi une solution intermédiaire qui consiste simplement à introduire une classe Smalltalk `OPS5Compatibility`, sachant gérer la notion de fraîcheur en fonction des avatars de ses instances (conformément à la politique choisie par OPS5) tout en décrivant un comportement par défaut dans la classe `Object` pour les autres classes du système[44].
Le cahier des charges pour cette classe consiste simplement à :

Affecter une fraîcheur à tout objet crée par incrémentation d'un compteur global à tous les objets,
Modifier cette fraîcheur à tout modification de l'objet.

Cette classe se définit donc simplement comme suit, en interceptant les méthodes des méta-actions (expansées pour prendre en argument la base de règles, Cf. Chapitre III.3.6.3 et IV.2.2) :

```
Object subclass: OPS5Compatibility
  instanceVariableNames: 'timeTag'

initializeTimeTag
  fraicheur <- self class newTimeTag

modifiedFor: aRuleBase
  self initializeTimeTag.
  super modifiedFor: aRuleBase

removeFor: aRuleBase
  self initializeTimeTag.
  super removeFor: aRuleBase

goFor: aRuleBase
  self initializeTimeTag.
  super goFor: aRuleBase
```

Et, bien sûr, en initialisant automatiquement tout objet créé, via une méthode dans la métaclasse :

[43] timeTag représentant la date de création ou de modification de l' objet.

[44] Nous suivons la solution Smalltalk avec la double gestion des dépendances pour les modèles des trilogies MVC, à la fois dans la classe `Object` "par défaut" et dans la classe `Model`, de manière plus propre [Goldberg&Robson 83, Lalonde&Pugh 90]; ce qui permet d'utiliser toute classe du système comme modèle, tout en proposant une "meilleure" solution pour les classes recompilables.

```
OPS5Compatibility class
  instanceVariableNames: ''

!OPS5Compatibility class methodsFor: 'instance creation'!

new
  ^super new initializeTimeTag
```

Il faut enfin s'assurer que les objets non compatibles-OPS5 ont un comportement par défaut vis à vis de la notion de fraîcheur, pour permettre la généricité des tests effectués par OPSMEA. Ceci se fait tout simplement en définissant dans la classe `Object` la méthode `timeTag`, comme rendant constamment 0 :

```
!Object methodsFor: 'OPS5 compatibility'!

timeTag
     ^0
```

VI.4.5.2. La règle loop1 de OPSMEA

L'utilisateur a cependant tout loisir de redéfinir la notion de fraîcheur pour les classes qu'il souhaite. Par exemple, il est intéressant de redéfinir cette notion pour les *évaluateurs*, dont la date de création peut jouer un rôle dans le choix de la règle. En fait la stratégie MEA est principalement construite pour provoquer un parcours en profondeur d'abord sur les objets Buts crées par les règles. Il est donc naturel de définir une notion de fraîcheur locale à chaque type d'objet.

On ajoute alors au vocabulaire de `OpusConflictSet` (Cf. Chapitre III.3.7.2) une méthode permettant de récupérer la règle filtrant les plus récents objets, en utilisant notre notion de fraîcheur :

```
!OpusConflictSet methodsFor: 'rule access'!

regleAyantLePlusRecentObjet
  ^self firstWithSortBlock: [:a :b | a highestTimeTag > b
highestTimeTag]
```

La base de méta règles OPSMEA peut alors s'écrire simplement de la manière suivante, en ne redéfinissant que la méta-règle `loop1` :

```
!OPSMEA methodsFor: 'loop'!

loop1
"on choisit de declencher une regle ayant l'evaluateur le plus recent
pour provoquer le parcours en profondeur"
| Evaluateur e. OpusConflictSet o |
    e status = #loop.
    o == e ruleBase conflictSet.
    o notEmpty.
    e reussi not.
actions
    | uneRegle |
    uneRegle _ o regleAyantLePlusRecentObject.
    Transcript show: uneRegle name;cr.
    o declenche: uneRegle.
    o modified
```

VI.4.5.2.1. Fraîcheur locale : Utilisation

Il faut noter que notre notion de fraîcheur locale permet de raffiner la stratégie de contrôle MEA, telle qu'elle existe en OPS5. En effet, en OPS5 la fraîcheur étant une notion globale, tous les objets filtrés par une règle sont considérés lors de la recherche, ce qui est un peu grossier. En particulier, dans l'exemple du singe et des bananes, l'idée est de préférer les règles filtrées par un objet But le plus récent. Les autres objets (singe, objets physiques) sont non significatifs dans cette recherche. Au contraire de OPS5, nous pouvons prendre en compte cette différence, en déclarant la classe `But` comme sous-classe d'`OPS5Compatibility`, mais pas les autres.

VI.5. Une application particulière : l'appel récursif en partie prémisse

VI.5.1. Problème

Une application plus complexe du mécanisme des évaluateurs consiste à manipuler *plusieurs évaluateurs concurremment* pour une même base de règles.
Cette idée est née du besoin de représenter des théorèmes de géométrie faisant intervenir des objets intermédiaires, construits au cours du raisonnement et sur lesquels un "sous-raisonnement" doit avoir lieu avant de pouvoir affirmer la conclusion du théorème (Cf Chapitre V.2.1.4).

Rappelons le théorème (3) (Chapitre V.2.1.4.4) :

> Théorème (3) :
> Un triangle ABC est isocèle en B, si le parallélogramme ABCP, tel que P = A + BC, est un losange

Nous avons proposé une première solution au Chapitre VI.2.1.4.4), basé sur une opération particulière (le lance:jusquA:). Nous reprenons ici cet exemple, en l'intégrant dans notre cadre déclaratif. Notre idée principale est que :

(Idée) Appel récursif = nouvel évaluateur
Un appel récursif au moteur se décrit simplement comme la création d'un évaluateur *supplémentaire* pour la base de règles.

Notre mécanique s'adapte très bien à ce type de connaissance, à condition d'*autoriser* l'usage d'évaluateurs multiples, déjà annoncée au Chapitre VII.3.2.
L'évaluateur crée sera alors géré par une base de méta-règles appropriée.

VI.5.2. Evaluateurs récursifs

Nous avons besoin alors de définir une nouvelle classe d'évaluateurs, qui peuvent mémoriser leur origine, et en cas de réussite, reprendre le cours de propagation de tokens qui s'était interrompu.
Ceci se fait par héritage, en définissant la classe `EvaluateurRecursif`, ayant comme variable d'instance le nœud Rete géniteur et le token parvenu à ce nœud:

```
Evaluateur subclass: #EvaluateurRecursif
  instanceVariableNames: 'token node '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'OPUS-evaluation'!
```

La méthode principale de cette nouvelle espèce est alors celle qui permet de reprendre le cours de propagation des tokens dans le réseau. Elle s'écrit donc naturellement comme l'envoi du message `acceptNewToken:` au nœud Rete suivant le nœud géniteur, avec en argument le token qui avait réussi les tests précédents :

```
!EvaluateurRecursif methodsFor: 'retriggering'!

retrigger
   Transcript show:
      'Je reessaie une premisse de la regle : ' , node ruleName; cr.
   node followingNode acceptNewToken: token
```

Les évaluateurs récursifs sont donc initialisés à leur création avec le nœud Rete et le token responsable de leur création. Ceci est effectué par la méthode de création standard décrite au paragraphe suivant.

VI.5.3. Ecriture

Nous proposons en outre une écriture raccourcie, permettant de créer cet évaluateur de manière pratique et standard. Cette écriture consiste à s'affranchir de la création explicite de l'évaluateur, et de sa prise en compte par la méta-base, pour n'écrire que

le code décrivant la condition d'arrêt du nouvel évaluateur. Ce code est écrit entre accolades. Considérons notre nouvelle règle une fois écrite :

triangleIsoceleParLeLosange
|Figure f. Local BC A B C P newObject|

f estUn: Triangle.
f nEstPasUn: TriangleIsocele.
f aUnCoteHorizontal.
BC <- f coteHorizontal.
A <- f pointOpposeA: BC.
B <- BC pointGauche.
C <- BC pointDroit.
P <- A + C - B.
newObject <- Parallelogramme new with: A with: B with: C with: P.
{newObject estUn: Losange}.
actions
f addType: TriangleIsocele.
(f as: TriangleIsocele) base: BC.
f modified.

Le Parser transforme alors la dixième prémisse en l'expression suivante dans la classe dynamique :

```
!ReglesDeFiguresDynamic methodsFor: 'triangles'!

P9573Node: aNode
  | newEvaluator |
  (newEvaluator <- self ruleBase defaultEvaluatorClass new
ruleBase: self ruleBase) addNewObjet: i2.
  newEvaluator stopCondition: [i2 estUn:Losange].
  newEvaluator token: self; node: t2.
  newEvaluator status: #init.
  ^self ruleBase newGoal: newEvaluator
```

Cette méthode prend comme argument le nœud Rete représentant la prémisse. Lors du test canonique effectué par le nœud, un nouvel évaluateur est créé, avec comme condition d'arrêt le bloc défini en prémisse, puis envoyé à la base de règles, par le message `newGoal:`.

Le message `newGoal:` va se charger de transmettre ce nouvel objet à la méta-base, si elle existe, par le message `goFor:` (expansion du go). Elle va en outre tester le bloc condition d'arrêt, au cas où celui-ci est déjà vérifié :

```
!OpusRuleSet class methodsFor: 'recursive cycles'!

newGoal: unEvaluateur
   unEvaluateur reussi ifTrue: [^true].
   metaBase isNil
    ifFalse:
          [Transcript show: 'un nouveau but declencheur est ne'; cr.
          unEvaluateur goFor: metaBase].
   ^false
```

VI.5.4. Une base de méta-règles gérant les évaluateurs récursifs

Une fois le nouvel évaluateur créé (ou les nouveaux par extensions), il faut être capable de gérer ces multiples évaluateurs par une base de méta-règles spécifiques. C'est ce que nous faisons maintenant, en introduisant la méta-base `MetaButsRecursifs`. Cette base est en fait très simple et n'ajoute qu'une méta-règle par rapport à `DefaultMeta`. Il suffit en effet de prendre en compte les évaluateurs récursifs *réussis*, et alors de *re-tester* la règle à l'origine de la création de cet évaluateur. Ceci est réalisé grâce au fait que les évaluateurs récursifs connaissent le nœud Rete qui est à leur origine.

```
DefaultMeta subbase: #MetaButsRecursifs
   globalObjects: ''
   category: 'OPUS-rules'
```

Cette méta-base *requiert* bien sûr un `EvaluateurRecursif`, et nous le déclarons :

```
!GoalGenerationMetaRules class methodsFor: 'evaluateur'!

requiredEvaluatorClass
   ^EvaluateurRecursif
```

La méta-règle de redéclenchement s'écrit alors :

```
!GoalGenerationMetaRules methodsFor: 'loop'!

loop1RetriggerGoal
"si l'evaluateur est reussi, on redeclenche la premisse suivante"
 | EvaluateurRecursif e. OpusConflictSet c |
    e status = #loop.
    c <- e conflictSet.
    e recursive.
    e reussi.
actions
    e retrigger; suspend; modified. c modified
```

VI.5.5. Une librairie de méta-bases réutilisables

Les méta-bases que nous avons décrites sont organisées en un graphe d'héritage et forment un ensemble de composants réutilisables (Cf Figure 24) Notre idée est d'offrir un *point de départ* à caractère exemplaire pour la spécification de contrôle sophistiqués. La spécification du contrôle s'effectue, comme nous l'avons vu par un double raffinage : raffinage de la notion d'évaluateur d'une part avec l'héritage de classe, et raffinage d'une base de méta-règles préexistante, via l'héritage de bases de règles.

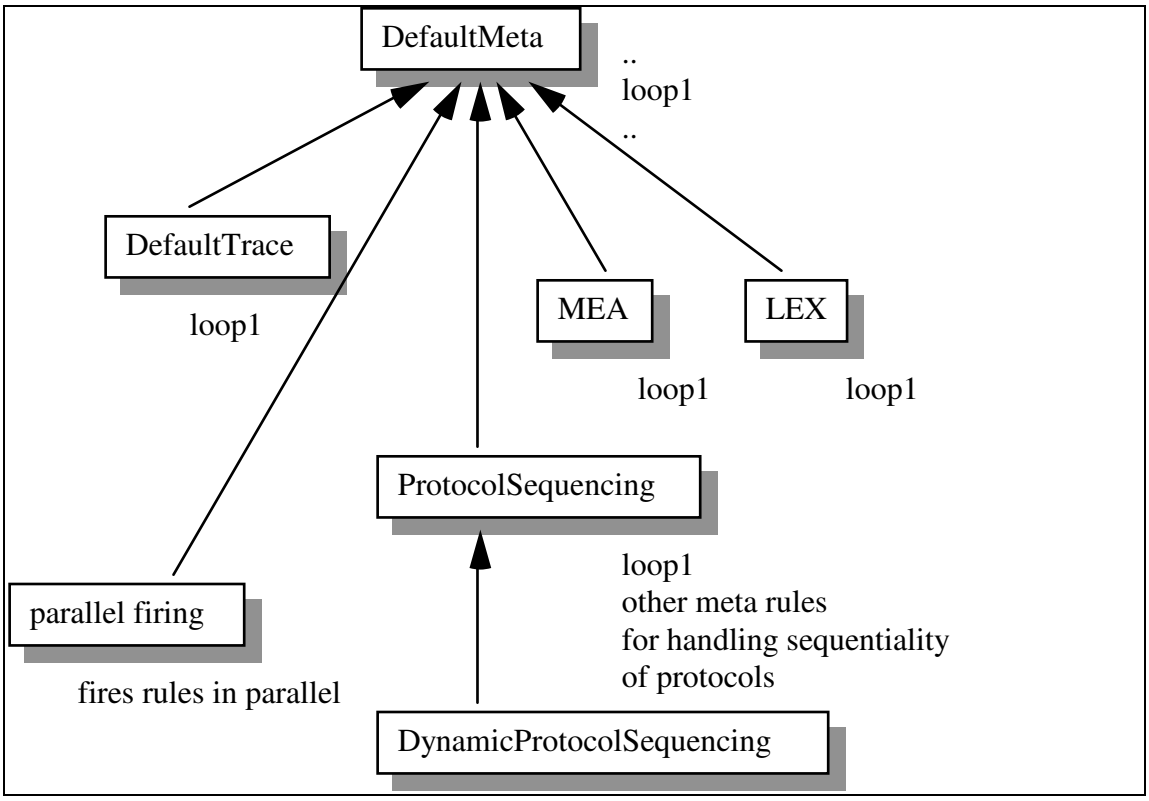


Figure 24. Une hiérarchie de méta-bases.

VI.5.6. Un exemple complet : le contrôle dans le système NéoGanesh

Le système NéoGanesh (décrit plus en détail an Chapitre VIII) contient une représentation du contrôle qui utilise extensivement notre architecture. Le problème du contrôle y est très particulier[45] : il faut d'un part représenter la séquencement des différentes phases du diagnostic, d'autre part être capable de modifier cette séquence dynamiquement en fonction du contexte, et enfin de représenter la notion d'alarmes, i.e. d'actions à effectuer en priorité maximale, quels que soient les autres actions ou raisonnements en cours.

[45] Comme dans toute base de connaissances réelle.

Ceci a abouti à une base de méta-règles structurées en cinq niveaux d'héritage, chaque niveau étant responsable d'une fonction de contrôle particulière.
Mais il est important de noter ici que le découpage en niveaux distincts de la base de contrôle s'est fait progressivement : les cinq niveaux en question sont apparut petit à petit. L'héritage nous a en particulier aidé à structurer le contrôle d'une manière *réutilisable*, puisque deux méta-bases des niveaux supérieures sont maintenant intégrées au système de base pour être utilisées par d'autres applications.

VI.6. La notion de but revisitée

Ce chapitre décrit une extension des règles NéOpus permettant de mieux en parler dans les méta-règles.

VI.6.1. Parler des règles (2)

L'architecture de NéOpus permet de parler des règles déclenchables, via le conflit set de manière assez réductrice. On peut parler des *objets* filtrés par la règle, de son nom, du protocole dans lequel elle est déclarée, mais on ne peut pas parler de son *action* sur l'environnement. Nous proposons ici d'introduire la notion d'assertion pour représenter la partie action sous la forme d'un véritable objet.

VI.6.2. La notion d'Assertion : un méta-langage au dessus de Smalltalk

VI.6.2.1. Motivations

Les expressions Smalltalk existent sous deux formes : la chaîne de caractère, et la méthode compilée. Ces deux formes sont pratiquement inutilisables dans le cadre d'un système discursif. On ne sais pas parler d'une chaîne de caractères ni d'une méthode compilée. Nous introduisons donc la notion d'`Assertion`, comme objet représentant une expression Smalltalk quelconque, de manière utilisable, en particulier par les méta-règles.

VI.6.2.2. Définition et représentation

Les assertions sont représentées par une classe Smalltalk `Assertion`, qui s'apparente à un arbre syntaxique instancié. La classe abstraite `Assertion` est une classe abstraite. Les différent types d'expression Smalltalk sont représentés par autant de sous-classe d'`Assertion` : les expressions sans argument (`Message0`), avec un argument (`Message1`) et avec deux (`Message2`).

Représentation syntaxique
Les assertions se représentent syntaxiquement comme des expressions Smalltalk, entourées d'accolades.

Par exemple, l'écriture suivante dans laquelle `aSinge` et `aBut` dénotent respectivement deux instances des classes `Singe` et `But` :

```
{aSinge position = aBut positionToReach}
```

est équivalente à celle-ci où les assertions sont crées explicitement :

```
Message2 new
  objet : aSinge
  selecteur : tient
  argument:   (Message1 new
                    objet: unBut
                    selecteur: positionToReach)
```

Les assertions sont capables de répondre au message `evaluate`, qui renvoie la valeur de l'expression Smalltalk. La méthode `evaluate` consiste tout simplement à parcourir l'arbre en profondeur d'abord, et est implémentée dans chacune des sous classes d'`Assertion`. La classe `Message0` rend l'objet qu'elle dénote, la classe `Message1` rend le résultat de l'envoi du message sans argument à son objet, et ainsi de suite.

VI.6.2.3. La partie `finalState` des règles

Les règles sont alors dotées d'un appendice supplémentaire permettant de décrire, sous la forme d'une assertion l'état de l'environnement après application de la règle. Par exemple, après application de la règle `holdObjectNotCeiling`, l'état du système sera tel que `{s isHolding: o}`, puisque le singe dénoté par `s` aura pris `o` (`(s take: o)` en partie action).

```
'MonkeyRules methodsFor: 'holding'!

holdObjectNotCeiling
 | Monkey s. ObjectOPS o |
   o weight = #light.
   o isNotOn: #ceiling.
   s isOn: #floor.
   s holdsNothing.
   s isAt: o at.
actions
   s take: o.
   o modified. s modified.
finalState
   {s isHolding: o}
```

Il est intéressant de regarder ce que génère le parser à partir de cette définition. Dans la classe dynamique de la base de règles la méthode

holdObjectNotCeilingFinalState est générée, en plus des méthodes nécessaires à la création du réseau Rete :

```
!MonkeyRulesDynamic methodsFor: 'holding'!

holdObjectNotCeilingFinalState
  ^Message2
    objet: i1
    selector: #isHolding:
    argument: i2
```

Cette méthode rendra une instance de Message2, qui représente les expressions Smalltalk à un argument, dont l'objet est lui même une instance de Message0 (représentant les variables; ici s qui donne, une fois renommée i1), le selecteur est #isHolding:, et l'argument une autre instance de Message0.
Les variables apparaissant dans la méthode ne sont autres que les variables de la règle renommées (Cf. pour plus de détails la compilation des règles en réseau Rete au Chapitre III).
On voit ici que l'état obtenu après déclenchement de la règle holdObjectNotCeiling peut être consulté avant que celle-ci ne soit déclenchée. Cet état est obtenu en envoyant le message holdObjectNotCeilingFinalState au token arrivant au bout du réseau Rete (le nœud et le token sont tous les deux contenus dans l'instance de règle déclenchable qui est alors créée).

VI.6.2.4. Encapsulation par les évaluateurs

Ces assertions vont nous servir à représenter plus finement la notion de but à atteindre pour les évaluateurs. Jusqu'à présent cette notion était représentée par un bloc Smalltalk. Nous pouvons maintenant définir un nouveau type d'évaluateurs ayant comme condition d'arrêt une assertion. Ceci va nous permettre ensuite de parler de la condition d'arrêt d'un évaluateur de manière beaucoup plus fine.
En fait, il n'est pas besoin pour ce faire d'introduire de nouvelle classe Smalltalk, nous allons simplement autoriser l'utilisation indifférente de blocs ou d'assertions pour représenter les conditions d'arrêt. Ceci est justifié par la remarque que les évaluateurs doivent pouvoir répondre au message reussi. Nous allons donc redéfinir la méthode reussi dans la classe Evaluateur, de manière à prendre en compte le cas où la condition d'arrêt est non pas un bloc Smalltalk mais une assertion, comme suit :

```
!Evaluateur methodsFor: 'evaluation'!

reussi
"si le but est un bloc, on lui envoie le message value, sinon on
l'evalue symboliquement.
Les messages inconnus ou provoquant une erreur se traduisent par un
resultat false"

   (stopCondition isKindOf: Assertion)
    ifTrue: [^self errorSignal handle: [:ex | ex returnWith: false]
                do: [stopCondition evaluate]].
   ^stopCondition value
```

Notons ici aussi que la méthode reussi ne déclenche pas d'erreur au cas où l'évaluation de l'assertion déclenche une erreur. Ceci nous permet à moindre frais de considérer les assertions comportant des sélecteurs non Smalltalk comme s'évaluant systématiquement à faux.

VI.6.2.5. Utilisation dans une méta-règle

Maintenant que nous avons un moyen de décrire l'action d'une règle sur l'environnement, nous allons utiliser ce moyen pour parler des règles. L'idée est d'utiliser les assertions comme condition d'arrêt pour les évaluateurs, au lieu des blocs Smalltalk, qui ne sont pas aussi facilement utilisables.
Le message `stopCondition` rend la condition d'arrêt d'un évaluateur. Il faut aussi un message rendant la partie finalState d'une règle déclenchable. Ce message est le message `finalState`.
Mais pour être facilement utilisable, il nous faut en particulier un message envoyé au conflict set permettant d'accéder aux règles *satisfaisant une assertion donnée*.
Un règle satisfait une assertion si sa partie finalState est unifiable avec l'assertion. Ce mécanisme d'unification est réalisé dans la classe des assertions. Le message `reglesSatisfaisant: uneAssertion` permet de récupérer cette liste.

VI.6.3. une base de méta-règles adaptée : `DefaultMetaAssertions`

Nous pouvons maintenant utiliser cette notion pour écrire une base de méta-règles reproduisant, en partie, le comportement de l'agent rationnel, tel qu'il est décrit dans [Ganascia 91]. L'idée est que nous allons déclencher en priorité une règle dont le finalState satisfait effectivement la condition d'arrêt d'un évaluateur.
Cela s'écrira simplement en redéfinissant la règle de bouclage `loop1`, de manière à choisir la règle vérifiant la condition d'arrêt d'un évaluateur dans une base de règles baptisée `DefaultMetaAsertions` :

```
!DefaultMetaAssertions methodsFor: 'loop'!

satisfyGoal
"on declenche une regle satisfiant le but "
| Evaluateur e. OpusConflictSet c. Local regles|
    e status = #loop.
    c _ e conflictSet.
    e reussi not.
    regles _ c reglesSatisfaisant: e stopCondition.
actions
    c declenche: regles first.
    e status: #end.
    c modified.
    e modified.
    e father notNil ifTrue: [e father modified].
finalState
    {e reussi}
```

VI.6.3.1. Parler d'un évaluateur

Les assertions sont des objets dont il est difficile de parler de manière concise. Nous avons introduit un mécanisme supplémentaire, qui n'ajoute rien au mécanisme de l'assertion en général, mais qui permet simplement de parler d'une assertion de manière concise. Par exemple, on aimerait savoir si la condition d'arrêt d'un évaluateur s'unifie avec une condition d'arrêt passée en paramètre. Un certain nombre de constructions syntaxiques permettent de parler d'une assertion, essentiellement via des mécanismes d'unification/filtrage. On pourra trouver des exemples complets dans la deuxième version du singe et des bananes présentée au chapitre VIII.

VI.6.4. Exemples

La notion d'assertion est encore récente et n'a pas été utilisée dans beaucoup d'exemples.

VI.6.4.1. Le singe et les bananes revisited

VI.6.4.2. La géométrie revisited

VI.6.5. Une autre interprétation des règles NéOpus

La notion d'assertion nous permet de donner une représentation manipulable de l'action d'une règle, et donc de représenter son *intention*. Elle donne par là même une

autre sémantique aux règles de production. Celles-ci, munies de leur appendice finalState représentent alors des *potentialités*.

Cette notion nous parait extrêmement intéressante mais nous n'avons pas eu le loisir de l'approfondir.

VI.7. Méthodologie pour la programmation multi-niveaux

VI.7.1. Non réflexivité et bases abstraites

Cf le problème de la métaclasse cachée (énoncé au III.3.3.4, résolu au IV.1.7.7).

VI.7.2. Environnement de programmation

Les éléments d'interface spécialisés : MultipleBrowserView, et MultipleConflictSetView.

VII. Applications de NéOpus

Avant-Propos

Nous présentons dans ce chapitre quelques applications (entendre bases de règles ou systèmes experts) réalisées à l'aide de NéOpus dans des cadres divers. Nous indiquons simplement pour chacune d'elles leurs traits caractéristiques et renvoyons éventuellement aux publications correspondantes. Ces applications font ressortir certains traits intéressants de NéOpus, d'autres soulèvent des problèmes partiellement ou non résolus. Nous avons choisi de présenter cette liste parce qu'elle constitue à notre avis une source non négligeable de problèmes qui restent encore à résoudre, tout en restant dans le cadre de nos possibilités en représentation de connaissances.
Ce chapitre est donc un prolongement du Chapitre VI (Praxis) et constitue une mise à l'épreuve "en vrai grandeur" des mécanismes que nous avons développés.

VII.1. Le singe et les bananes : première version

VII.1.1. Problème, contexte

L'idée est de réécrire la base de règles existante de [Brownston], en NéOpus pour étudier sur un exemple connu les apports de NéOpus par rapport au système OPS5 originel. Cette réécriture est décrite en détail dans [Pachet 91c]. Deux versions de cette base de règles ont été écrites. La première est une transposition directe de la version originale, utilisant les possibilités de NéOpus en particulier l'intégration de l'héritage de classe dans les règles. La deuxième est une tentative pour dénouer l'imbrication forte entre la gestion des objets du domaine d'une part et la gestion des buts d'autre part, en utilisant l'architecture de contrôle de NéOpus. Cette base de règles suppose (c'est un prérequis implicite dans la description originale) un déclenchement des règles respectant la stratégie MEA : en effet, les sous buts créés dynamiquement doivent être considérés avant leurs buts père, sous peine de voir le système s'arrêter inopinément.

VII.1.2. La base de règles originale

La base de règles originale utilise les quatre classes suivantes (définies par la commande OPS `literalize`):

```
     PhysicalObject
name         : une chaine de caractères
at           : la position horizontale. un nombre
weight       : le poids parmi (#light #heavy)
on           : le nom d'un PhysicalObject ou #floor ou #ceiling

     Monkey
at           : la position. un nombre
on           : la position verticale. le nom d'un PhysicalObject ou #floor
ou               #ceiling
holds        : nil ou le nom d'un PhysicalObject de poids #light.

     Goal
status       : parmi #active ou #satisfied
type  : parmi #holds #on ou #at.
object-name: le nom de l'objet du but if any. Varie suivant les types.
to           : nil ou les types #on et #holds. sinon l'endroit ou amener un
                    objet.

     Testcase
type :       parmi #general #holds #at #to.
name :       unique pour chaque instance, indique quelles regles sont   à
             tester.
```

La base de règles contient 26 règles, réparties en plusieurs paquets correspondant aux divers types d'actions entreprises par le singe et un pour terminer la session. Les paquets sont eux-mêmes divisés en plusieurs sous-paquets, suivant l'action envisagée (attraper un objet, sauter à terre).

Ces règles peuvent par ailleurs être réparties en trois groupes : les règles générant des sous-buts, les règles satisfaisant un but en effectuant une modification des objets, et les règles satisfaisant un but sans modifier les objets.

Exemples

Nous donnons ici quelques exemples de règles OPS que nous reprendrons comme repères pour la comparaison des deux systèmes :

Une règle de génération de sous-but

"*Si il y a un but de tenir un objet qui est au plafond, et que l'échelle est à terre ailleurs, alors on génère un sous-but pour le singe d'amener l'échelle sous l'objet.*"

```
(p Holds::Object-ceiling:At-Object

  (goal ^status active ^type holds ^object-name <o1>)
  (phys-object ^name <o1> ^weight light ^at <p> ^on <ceiling>)
  (phys-object ^name ladder ^at <> <p>)
-->
  (make goal ^status active ^type at ^object-name ladder ^to <p>))
```

Une règle de satisfaction de sous-but par exécution d'une action

"*Si il existe un but d'amener un objet à un endroit, et que le singe tient l'objet, mais à un autre endroit, alors le singe va à l'endroit du but, et on modifie les trois objets* `singe`, `objet` *et* `but` *en conséquence*"

```
(p At::Object

  {(goal ^status active ^type at ^object-name <o1> ^to <p>)      <goal>}
  {(monkey ^at <> <p> ^holds <o1> ^on floor)                     <monkey>}
  {(phys-object ^name <o1>)                                  <object1>}

-->
  (write Move <o1> to <p>)
  (modify <object1> ^at <p>)
  (modify <monkey> ^at <p>)
  (modify <goal> ^status satisfied))
```

Une règle de satisfaction de but sans action

"*Si il existe un but de ne rien tenir, et que le singe ne tient effectivement rien alors le but est satisfait.*"

```
(p Holds::nil:Satisfied

  {(goal ^status active ^type holds ^object-name nil)            <goal>}
  {(monkey  ^holds nil ^at <p> ^on <q>)                       <monkey>}

-->
  (write Monkey is holding nothing)
  (modify <goal> ^status satisfied) )
```

Enfin, deux règles étranges gèrent la fin de la session :

Terminaison

```
(p congratulations

  (testcase)
  (goal ^status satisfied)
- (goal ^status active)
-->
  (write congratulations
         all goals satisfied)
  (halt))
```

```
(p impossible

(goal ^status satisfied ^type <g>)

-->

(write impossible
        <g> cannot be satisfied))
```

Ces deux règles doivent être appliquées en fin de session. La première contient une prémisse négative, qui teste qu'il n'existe aucun but de status active. Cette prémisse s'écrira de la même manière en NéOpus. La deuxième ne prend de sens qu'avec la stratégie de contrôle particulière d'OPS5. En effet, cette règle *sera toujours applicable*, mais ne sera déclenchée que lorsqu'aucune autre règle sera déclenchable, car elle sera toujours *la moins spécifique* (une seule prémisse très peu contrainte). Nous verrons comment donner une interprétation propre en NéOpus à cette deuxième règle, en lui rendant son caractère intrinsèquement "méta".

VII.1.3. Les classes

La fabrication des classes Smalltalk correspondant aux structures de données OPS est relativement directe. Nous pouvons cependant utiliser au mieux les possibilités de Smalltalk et de NéOpus en remarquant que :

. les classes `Monkey` et `PhysicalObject` partagent des structures (`at` et `on`). Nous allons utiliser l'héritage Smalltalk pour représenter cette ressemblance. Plus généralement, tous les objets intervenant dans notre univers vont partager une superclasse commune.

. Les valeurs possibles des attributs OPS sont atomiques. Nous allons faire pointer directement les objets les uns sur les autres, et ainsi nous affranchir des gestions maladroites des *noms* d'objets et des indirections correspondantes.

. Les prémisses OPS sont constituées de tests simples (=, <>) sur les valeurs des attributs des instances. Nous allons nous affranchir de cette programmation en définissant un certain nombre de méthodes d'accès plus lisibles, qui rendront

l'implémentation de ces objets transparente[46]. De plus ces méthodes pourront factoriser certaines d'actions (comme les affichages), qui permettront d'alléger la partie action des règles.

. La classe `TestCase` ne nous intéresse pas ici, nous effectuerons les tests en définissant des méthodes Smalltalk, bien mieux adaptées.

Nous définissons donc quatre classes, en écrivant un certain nombre de méthodes d'accès réparties en deux protocoles : les accès utilisés en partie prémisse (lecture) et ceux utilisés en partie conclusion (écriture).
Noter que les messages en *lecture* utilisent la troisième personne du singulier (comme `isOn:`, `holdsSomething`), ceux en *écriture* utilisent la deuxième personne (comme `goTo:`, `bring:to:`)[47].

Les objets physiques

Ceux ci définissent trois variables d'instance : `on`, `at`, et `weight`. Le nom est supprimé, rendu inutile par le typage des variables.

```
Object subclass: #ObjectOPS
  instanceVariableNames: 'at on weight'
```

Quatre méthodes d'accès en lecture nous suffisent :

```
!ObjectOPS methodsFor: 'rule testing'!

isAt: aPosition
  ^at = aPosition

isNotAt: aPosition
  ^self isAt: aPosition not

isOn: aPosition
  ^on = aPosition

isNotOn: aPosition
  ^(self isON: aPosition) not
```

Trois sous-classes de `ObjectOPS` vont être créées, pour représenter les divers objets mis en jeu : `Couch`, `Blanket` et `Banana`. Ces classes ne redéfinissent aucune variable d'instance ni méthode, tout étant hérité d'`ObjectOPS` :

[46] Ou plutôt opaque, Cf.la note du paragraphe I.2.4.2.2.

[47] Smalltalk suit assez bien cette règle (Cf les messages standards comme `become:`, `at:put` (sans `s`) en écriture et `isNumber` ou `isKindOf:` en lecture). Seul le message Smalltalk `yourself` (rendant l'objet receveur, donc accès en lecture) ne suit pas cette règle : on devrait dire `itself`.

```
ObjectOPS subclass: #Blanket
  instanceVariableNames: ''

ObjectOPS subclass: #Couch
  instanceVariableNames: ''

ObjectOPS subclass: #Banana
  instanceVariableNames: ''
```

Les singes

Nous définissons aussi la classe `Monkey` comme sous-classe de `ObjectOPS`, en rajoutant simplement une variable d'instance représentant l'objet tenu.

```
ObjectOPS subclass: #Monkey
  instanceVariableNames: 'objectHeld '
```

Les méthodes suivantes nous permettent de faire effectuer au singe les actions canoniques relatives à sa situation: bouger, amener un objet à un certain endroit, lâcher un objet, grimper sur un objet, sauter sur un autre. Notons que l'écriture de ces méthodes permet en autres d'encapsuler la gestion des affichages, et ainsi d'alléger les parties action des règles.
Certaines des méthodes en écriture effectuent des effets de bords sur le singe uniquement (comme `goTo:`) d'autres effectuent des effets de bords à la fois sur le singe et sur l'objet passé en paramètre (comme `bring:to:` et `drop`). Nous retrouverons ici le classique *frame-problem*. Celui-ci se traduira par les déclarations appropriées d'objets modifiés (les `modified`) en partie conclusion des règles.

```
!Monkey methodsFor: 'rule testing'!

holdsSomething
 ^objectHeld exists

isHolding: anObject
 ^objectHeld == anObject

!Monkey methodsFor: 'rule actions'!

bring: anObject to: aPosition
 Transcript show: 'le singe amene ',anObject printString,
                  ' a ', aPosition printString;cr.
 at _ aPosition.   anObject at: aPosition

climbOn: anObject
 Transcript show: 'le singe grimpe sur ' , anObject printString; cr.
 on _ anObject

drop
 Transcript show: 'le singe lache ' , holds printString; cr.
 objectHeld on: #floor.  objectHeld _ nil

...
```

Les échelles

Nous fabriquons une sous classe particulière pour les échelles, afin de supprimer la gestion maladroite des noms. Ainsi, on ne testera plus qu'un objet est une échelle en testant son nom (en OPS : `nom = #ladder`), mais simplement en déclarant la variable comme étant de la classe `Ladder`. Cette classe ne définit aucune variable d'instance ni méthode, tout étant hérité de la classe `ObjectOPS` :

```
ObjectOPS subclass: #Ladder
  instanceVariableNames: ''
```

Les buts

Enfin, les buts sont définis comme sous-classes d'`Object`, dans cette première version, avec les mêmes attributs que dans la version OPS5 :

```
Object subclass: #MonkeyGoal
  instanceVariableNames: 'status objet to type '
```

Trois méthodes d'accès évoluées sont définies pour accéder au `status` :

```
!MonkeyGoal methodsFor: 'rule action'!

becomeSatisfied
  status _ #satisfied!

!MonkeyGoal methodsFor: 'rule testing'!

isActive
  ^status == #active!

isSatisfied
  ^status == #satisfied
```

Les méthodes d'accès standard sont définies pour les trois autres variables d'instance.

VII.1.4. La base de règles[48]

Reprenons les exemples de règles décrit précédemment. Notre base de règles se définit comme sous-base de la base racine `OpusRuleSet` :

```
OpusRuleSet subbase: #MonkeyRules
  globalObjects: ''
  category: 'OPUS-rules'
```

Les règles sont organisées en cinq protocoles : `at`, `hold`, `on`, `climb`, `termination`, correspondant aux paquets de la base originale. De même, comme dans la base de règles originale, nous avons trois types de règles : génération de sous-buts, satisfaction de buts et terminaison. Donnons ici leur transcription NéOpus :

[48] Se reporter à [Pachet 91c] pour la base de règles complète.

La règle de génération de sous-buts
(ex `Holds::Object-ceiling:At-Object`)

On peut remarquer ici que :

- L'objet `o` est fonctionnellement accessible à partir de l'objet but `b`. Cela se traduira par la prémisse d'affectation (`o <- b objet`), utilisant la notion de *variable locale déclenchante*, au lieu de la prémisse (`o name = b objet name`), implicitement utilisée par OPS5.

- Noter aussi l'usage de la classe `Ladder`, qui évite le test sur le nom.

On obtient alors 7 prémisses pour la règle NéOpus, contre quatre prémisses et 9 tests pour OPS5 :

```
holdObjectCeilingAtObj
"si le but est que le singe tienne un objet qui est au plafond et que
l'echelle n'est pas au meme endroit alors generer un but d'amener
l'achelle au bon endroit"
  | MonkeyGoal b. ObjectOPS o. Ladder l|

 b isActive.
 b type = #hold.
 o _ b objet.
 o weight = #light.
 o isOn: #ceiling.
 l isOn: #floor.
 l isNotAt: o at.

actions
 (MonkeyGoal new status: #active; type: #at; objet: l; to: o at) go
```

La règle de satisfaction de sous-but par exécution d'une action
(ex `At::Object`)

Remarquons ici l'utilisation de la méthode `bring:to:` qui a pour effet de modifier à la fois le singe et l'objet transporté. Cela se traduit par la déclaration `modified` à la fois pour `s` et pour `o` (l'ordre des déclarations est indifférent) :

```
atObject
  | MonkeyGoal b. Monkey s. ObjectOPS o|

  b isActive.
  b type = #at.
  o _ b objet.
  s isNotAt: b to.
  s isHolding: o.
  s isOn: #floor.
actions
  s bring: o to: b to.
  b becomeSatisfied.
  b modified. o modified. s modified.
```

La règle de satisfaction de but sans action
(ex `Holds::nil:Satisfied`)

Remarquons ici simplement que notre tactique d'encapsulation des messages (Cf. VII.1.3) ne marche plus dans ce cas, le message ne peut être écrit ailleurs que dans la règle, puisque la partie action n'envoi aucun message au singe !:

```
holdNilSatisfied
  | MonkeyGoal b. Monkey s |

  b isActive.
  b type = #hold.
  b objet isNil.
  s holdsNothing.
actions
  b becomeSatisfied.
  Transcript show: 'le singe ne tiens rien ';cr.
  b modified.
```

Terminaison
(ex `congratulations` et `impossible`)

La première règle utilise naturellement une prémisse négative, avec l'opérateur NéOpus NOT :

```
!MonkeyRulesEnglish methodsFor: 'termination'!

congratulations
  | MonkeyGoal b |

  b isSatisfied.
  NOT | MonkeyGoal b2 | b2 isActive.
actions
  Transcript show: 'bravo tous les buts ont ete atteints';cr
```

La règle `impossible` s'écrirait naturellement comme ceci :

```
impossible
  | MonkeyGoal b |

  b isActive.

actions
  Transcript show: 'impossible, le but', b printString, 'ne peut
etre atteint';cr
```

Mais nous l'omettons volontairement car nous voulons garder dans un premier temps une stratégie de déclenchement simple. Elle sera traitée dans un deuxième temps par les méta-règles.

Exemples de lancement

Voici une méthode de lancement, définie dans `MonkeyRules class`, transposée elles aussi de [Brownston]. On crée un certain nombre d'instances des classes concernées, puis on initialise le contexte de `MonkeyRules`, en enfin on lance l'exécution :

```
!MonkeyRulesEnglish class methodsFor: 'exemple'!

generalOnLadder
      | s b ladder banana blanket couch |

self setNaturalTyping. "pour prendre l'heritage en compte"

couch _ Couch new at: 7 @ 7; weight: #heavy; on: #floor.
ladder _ Ladder new at: 5 @ 5; weight: #light; on: #floor.
banana _ ObjectOPS new at: 7 @ 7; weight: #light; on: #ceiling.
blanket _ Blanket new at: 5 @ 5; weight: #light; on: #floor.
s _ Monkey new at: 5 @ 5; on: ladder.

b _ MonkeyGoal new status: #active; type: #hold; objet: banana.

self addInContext:couch and: ladder and: banana and: s and: b and: blanket.
self execute.
```

Comme nous l'avons signalé au début, la base de règles nécessite pour être correctement évaluée un parcours en profondeur d'abord du graphe d'état, de manière à satisfaire en priorité les buts fils générés par les règles, ce qui nous mène directement à l'assertion suivante :

VII.1.5. Redéfinition des buts (1)

Spécifier un parcours en profondeur d'abord du graphe d'état va se traduire en NéOpus par l'association de la méta-base `OPSMEA` à la base `MonkeyRules`. La base de règles `OPSMEA` (Cf. VI.4.5) choisit de déclencher la règle filtrée par les objets *les plus récents* (grâce à la notion de `timeTag`). Dans notre cas, les seuls objets auxquels nous voulons appliquer ce critère de fraîcheur sont les buts du singe. Nous allons donc doter ceux-ci d'une capacité à représenter leur fraîcheur, en les définissant comme sous-classe de la classe `OPS5Compatibility`[49].
Les autres objets participant aux raisonnement (les singes, echelles et autres) possèdent une fraîcheur gérée par défaut dans `Object`, qui est constante et nulle. `OPSMEA` va donc sélectionner les règles uniquement en fonction de la fraîcheur des objets `MonkeyGoal` la filtrant.

VII.1.6. Gestion de la terminaison par une méta-règle

Nous allons maintenant représenter la terminaison de la base de règles, de manière "propre" en la dissociant de la base de règles elle-même. L'idée est que le protocole de terminaison, (contenant les deux règles `congratulations` et `impossible`) est un protocole particulier, à n'utiliser qu'en fin de raisonnement au lieu de reposer sur une stratégie de contrôle implicite (par exemple reposant sur la notion de "règle plus ou moins contrainte" comme en OPS5). La simple méta-règle suivante va permettre de clore le débat, en ne déclenchant les règles de protocole terminaison *que si aucune autre règle n'est applicable* :

[49] Justement fabriquée dans le but de permettre l'utilisation simple de OPSMEA.

```
pasTermine
  | Evaluateur e. Local cs |

    e status = #loop.
    cs_ e conflictSet.
    cs aDesReglesDeProtocole: #termination.
    cs aDesReglesDeProtocoleDifferentDe: #termination.
actions
    c enleveRegles: (c reglesDeProtocole: #termination). e modified.
```

Nous utilisons maintenant l'héritage de bases de règles pour "rajouter" cette méta-règle à celles de OPSMEA, en produisant une nouvelle base de méta-règles, baptisée `SingeMeta`, sous-base de OPSMEA, et implémentant uniquement notre méta-règle `pasTermine`.

Nous obtenons alors le schéma de contrôle suivant (Cf. Figure 25) :

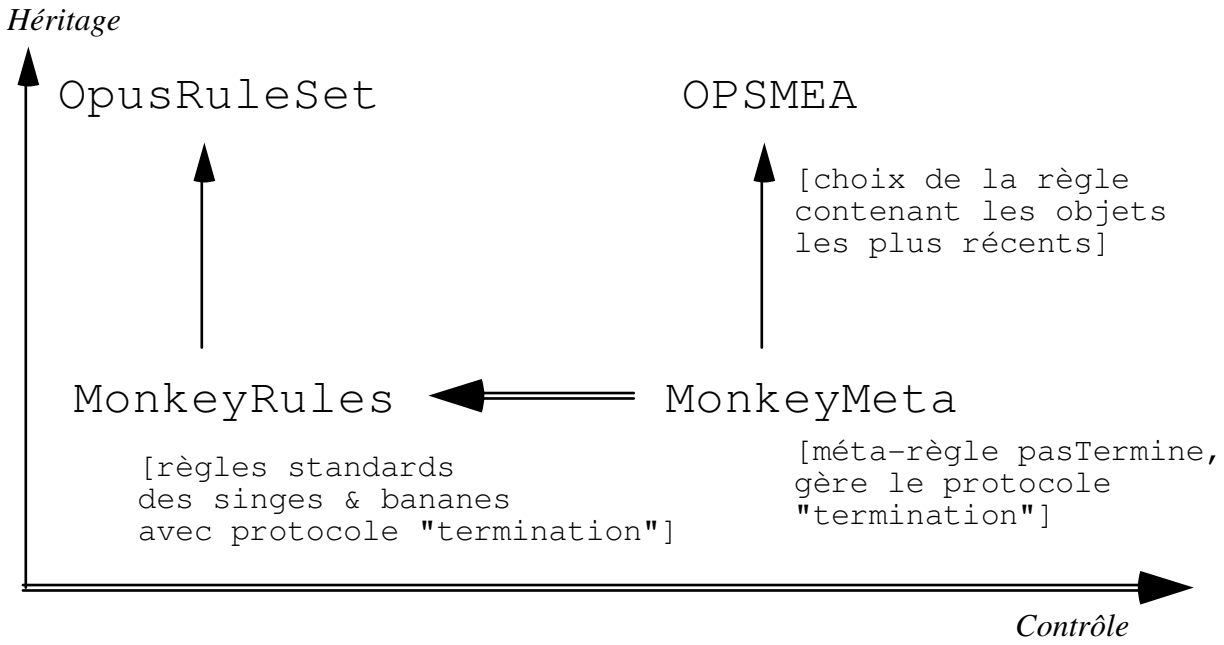


Figure 25. contrôle standard pour `MonkeyRules`

VII.1.7. Traits saillants

Cette base de règles, bien que critiquable sur le fond (notamment sur la validité et la consistance de l'expertise représentée) est néanmoins intéressante à plus d'un titre :

- Elle souligne l'importance de la représentation des objets du domaine sous forme de véritables objets plutôt que sous forme de n-uplets : association de comportement aux classes, valeurs des attributs non atomique, factorisation des propriétés grâce à l'héritage et au typage naturel.

- Elle souligne aussi le caractère extrêmement implicite du contrôle dans la base de règles originale. Ici le contrôle est clairement séparé de la base de règles du domaine.

VII.2. Le singe et les bananes : deuxième version

VII.2.1. Idée

Nous présentons ici une version intégralement différente de la précédente, mais s'appuyant sur les mêmes objets, à savoir les instances des classes `Monkey`, `ObjectOPS`, `Ladder`, et `MonkeyGoal`. L'idée est maintenant de dissocier les règles *du domaine*, c'est à dire portant sur les objets `Monkey` et `ObjectOPS`, des méta-règles portant sur *l'évaluation* de ces dernières, en gérant en l'occurrence la hiérarchie des buts.
Nous allons pour ce faire utiliser pleinement l'architecture déclarative de contrôle, en définissant deux bases de règles. Les règles du domaine seront implémentées dans une base de règles appelée `MonkeyRulesWithMeta`, et exprime des *potentialités* sur les objets du domaine, indépendamment de tout but.
Les règles exprimant des connaissances sur les objets d'ordre supérieur que sont les `MonkeyGoal` seront implémentées dans la méta-base `MonkeyRulesMeta`. Cette dernière sera sous-base de la méta-base standard `MetaButs`, qui sait gérer convenablement les buts et les assertions[50].
Nous donnons ici un bref rappel de l'usage des assertions en NéOpus. Se reporter à [Pachet 91d] pour plus de détails.

VII.2.2. Les assertions, le finalState, les stopCondition

VII.2.2.1. La fonction de contrôle

Notre idée est que les objets `MonkeyGoal` constituent des évaluateurs, au sens de [Pachet 91c], particuliers pour gérer la base de règles des singes. Nous enrichissons donc la hiérarchie d'évaluateurs standard, en formant une sous-classe, appelée `MonkeyGoal` d'une des classes d'`Evaluateur` existante.
Dans notre cas, comme nous l'avons vu précédemment, nous voulons que ces `MonkeyGoal` soient étiquetés, afin d'implémenter la notion de timeTag. Cela nous conduit à définir notre classe comme sous classe de `EvaluateurEtiqueté`:

[50] Notons que Michèle Vialatte dans [Vialatte 85] propose une critique de même nature que la notre et une réalisation dans le système Snark. Mais la représentation du contrôle n'y est pas aussi clairement séparée.

```
EvaluateurEtiquete subclass: #MonkeyGoal
  instanceVariableNames: ''
```

les notions de `status` et de `timeTag` étant implémentés au niveau des superclasses. Les variables `type`, `on`, et `at` disparaissent et sont remplacées par une représentation plus fine de la notion d'assertion.

VII.2.2.2. La fonction assertionnelle

Nous allons représenter la fonction assertionnelle des buts en utilisant l'implémentation NéOpus de la notion d'assertion. Rappelons brièvement cette notion.

VII.2.2.2.1. Définition

Une assertion dans notre contexte, est toute expression syntaxiquement correcte (au sens de Smalltalk). Elle est se différencie extérieurement des expressions Smalltalk par deux accolades.
Par exemple, l'assertion représentant le fait que le singe `aMonkey` tient une banane `aBanana` sera :

```
  {aMonkey isHolding: aBanana}
```

La principale différence entre assertions et expressions Smalltalk est que les assertions sont manipulables comme entités à part entières, et n'ont pas besoin d'être compilées pour être évaluées.
Leur définition informatique est un arbre syntaxique complètement instancié (les nœuds terminaux sont des objets Smalltalk et non pas des variables).

VII.2.2.2.2. Usage des assertions

Les assertions ne sont pas utilisables directement dans l'environnement Smalltalk. Elles sont accessibles et utilisables qu'à travers les évaluateurs, via leur `stopCondition`, la partie `finalState` des règles et certaines prémisses de méta-règles.
Le rôle principal de ces assertions est d'établir un lien entre les parties action et les parties prémisses des règles, afin de pouvoir "parler" d'une règle déclenchable, en fonction de son effet sur l'environnement. Ainsi, une assertion représente un état du système, par le biais d'une expression Smalltalk quelconque.

VII.2.2.2.3. Usage dans un évaluateur

Les évaluateurs ont comme structure essentielle la `stopCondition`, qui représente l'état souhaité par l'évaluateur. Cet état est défini pour les évaluations standards comme étant un bloc Smalltalk. Nous allons étendre cette notion, en proposant que les `stopConditions` soient des assertions.

Ainsi, l'initialisation de la base de règles, pour la même configuration d'objets initiaux (Cf. la méthode de lancement `generalOnLadder`) consistera en la création d'un `MonkeyGoal` dont la `stopCondition` est une assertion représentant l'état souhaité soit dans notre exemple :

```
{unSinge isHolding: banana}.
```

Cela se traduira par l'expression de lancement de la base de règles :

```
self executeWithAssertion: {unSinge isHolding: banana}.
```

au lieu de la création de l'objet `MonkeyGoal`, comme dans l'exemple précédent :

```
GoalMonkey new status: #active; type: #hold; objet: banana.
```

VII.2.2.2.4. Utilisation dans une règle

Les assertions sont par ailleurs utilisées dans les règles, de manière à faire le lien entre évaluateurs et règles déclenchables.

A/ En partie `finalState` d'une règle du domaine

La partie `finalState` d'une règle (située après la partie `actions`), consiste en une assertion, utilisant éventuellement les variables utilisées dans la règle. Cette assertion définit "l'état obtenu après déclenchement de la règle", et permet donc de savoir ce que ferait la règle si elle était déclenchée.

Par exemple, si l'on reprend notre règle `holdObjectNotCeiling`, cette dernière s'écrira maintenant, sans utiliser d'objet `MonkeyGoal`, et en rajoutant une partie `finalState` (à droite):

```
holdObjectNotCeiling
"premiere version"
| MonkeyGoal b. Monkey s. ObjectOPS o |
b isActive.
b type = #hold.
o _ b objet.
o weight = #light.
o isNotOn: #ceiling.
s isOn: #floor.
s holdsNothing.
s isAt: o at.
actions
s take: o.
b becomeSatisfied.
b modified. o modified. s modified.
```

```
holdObjectNotCeiling
"deuxieme version"
  | Monkey s. ObjectOPS o |
o weight = #light.
o isNotOn: #ceiling.
s isOn: #floor.
s holdsNothing.
s isAt: o at.
actions
s take: o.
o modified. s modified.
finalState
{s isHolding: o}
```

On peut ainsi dans cet exemple exprimer la différence fondamentale entre prendre et tenir :

si	`s take: o`
alors on aura :	`{s isHolding: o}`

B/ En prémisse `ditQue:` d'une méta-règle.

Accès au `finalState` d'une règle déclenchable

Afin de savoir si une règle déclenchable vérifie une certaine assertion, if faut pouvoir, en partie prémisse des méta-règles, accéder au `finalState` d'une règle déclenchable. Cela se traduit par un message d'accès envoyé à la règle déclenchable :

```
uneRegleDeclenchable finalState.
```

Il est par ailleurs intéressant d'accéder via le conflict set à l'ensemble des règles satisfaisant une assertion donnée. Le message `reglesSatisfaisant:` réalise cette requête (Cf règle `loop1SatisfaireGoal` dans `DefaultMetaButs`) :

```
regles <- c reglesSatisfaisant: e stopCondition.
```

Accès à la `stopCondition` d'un évaluateur

Accéder à la stopCondition d'un évaluateur nécessite plus de travail. Mieux, il s'agit véritablement d'une liaison d'arbre, puisque les objets apparaissant dans une assertion doivent pouvoir être accédés. Le mot-clé `ditQue:` permet de réaliser cette unification/liaison.

Par exemple, dans la règle suivante, la `stopCondition` de l'évaluateur `b`, va être unifiée avec l'assertion `{s isHolding: o}`. Si l'unification échoue, la prémisse échouera. Si l'unification réussi, alors les variables `s` et `o` (déclarées dans la règle comme `Local`) seront unifiées avec les objets correspondants de la `stopCondition` de `b`. Ce qui permettra d'écrire des prémisses sur ces objets.

```
holdObjectCeilingAtObj
  | MonkeyGoal b. Local s o. Ladder l|

    b status = #loop.
    b ditQue: {s isHolding: o}.
    o weight = #light.
    o isOn: #ceiling.
    l isOn: #floor....
```

VII.2.3. Nouvelle interprétation des règles

A partir du moment où les objets `MonkeyGoal` disparaissent des règles de base, celles-ci prennent alors une signification différente. En effet, on exprimera dans les

règles *toutes les actions potentielles d'un singe en fonction de son environnement*, et ce, indépendamment de tout but (ou de toute évaluation). Toute la gestion des buts est décrite dans la base de méta-règles.
Les méta-règles sont alors chargées deux fonctions :

Déterminer parmi les actions possibles du singe celles qui satisfont effectivement un but.
Représenter la génération des sous-buts, en fonction du contexte.

La première fonction est générale. Une méta-base appelée `DefaultMetaButs` assure ce rôle de manière canonique en une méta-règle. La deuxième fonction sera assurée par notre méta-base `MonkeyMeta`, par un jeu de méta-règles de génération de buts.

VII.2.4. Exemples

Nous reprenons ici nos trois règles exemples, dans la nouvelle architecture :

Règles du domaine

```
atObject
  | Monkey s. ObjectOPS o o2|
    s isNotAt: o2 at.
    s isHolding: o.
    s isOn: #floor.
actions
    s bring: o to: o2 at.
    o modified. s modified.
finalState
    {(s isAt: o2 at) &(s isHolding: o)}
```

Une méta-règle de satisfaction de but sans action

Voici la base de méta-règles `DefaultMetaButs`, qui gère les assertions, en ne déclenchant que les règles qui satisfont un but (prémisse : `c regleSatisfaisant: e stopCondition).`

```
DefaultMeta subbase: #DefaultMetaButs
   globalObjects: ''
   category: 'OPUS-rules'
```

```
!DefaultMetaButs methodsFor: 'loop'!
loop1SatisfaireGoal
"on declenche une regle satisfiant le but "
| Evaluateur e. OpusConflictSet c. Local regle|
    e status = #loop.
    c == e conflictSet.
    e reussi not.
    regle _ c regleSatisfaisant: e stopCondition.
actions
    c declenche: regle.
    e status: #end.
    c modified.
    e modified.
    e pere notNil ifTrue: [e pere modified].
```

Une méta-règle de génération de buts

```
holdObjectCeilingAtObj
  | MonkeyGoal b. Local s o. Ladder l|
    b status = #loop.
    b ditQue: {s isHolding: o}.
    o weight = #light.
    o isOn: #ceiling.
    l isOn: #floor.
    l isNotAt: o at.
actions
    |b2|
    Transcript show: 'genere un but d" etre a echelle';cr.
    b2 _ b newSon.
    b2 stopCondition: {(s isAt: o at) & (s isHolding: l)}.
    b2 go
```

Lancement

Voici alors l'exemple de lancement dans la nouvelle version. La création des objets initiaux est inchangée, mis à part l'objet `MonkeyGoal`, qui est maintenant créé implicitement par le biais du message d'activation `executeWithAssertion:` qui lance l'exécution avec un évaluateur (ici une instance de `MonkeyGoal`) ayant comme `stopCondition` l'assertion passée en argument :

```
generalOnLadder
      | s  ladder banana blanket couch |

self setNaturalTypage.
couch _ ObjectOPS new at: 7@7; weight: #heavy; on: #floor.
ladder _ Ladder new at: 5@5; weight: #light; on: #floor.
banana _ ObjectOPS new at: 7@7; weight: #light; on: #ceiling.
blanket _ ObjectOPS new at: 5@5; weight: #light; on: #floor.
s _ Monkey new at: 5@5; on: ladder.

self addInContext: couch and: ladder and: banana and: blanket and: s.
metaBase addInContext: ladder.

self executeWithMessage: {s isHolding: banana}
```

On obtient alors le nouveau schéma de contrôle (Cf. Figure 26) :

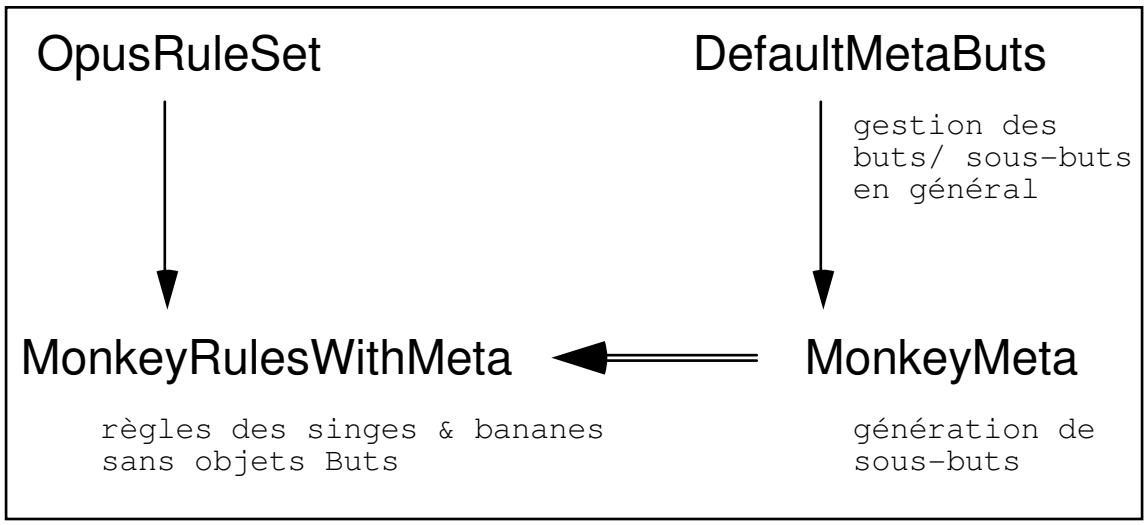


Figure 26. Nouveau schéma de contrôle.

On définit alors les mêmes méthodes de lancement que précédemment aux créations de buts près.

VII.3. Un système expert de contrôle de respirateur

VII.3.1. Problème, contexte

Le système NéoGanesh [Dojat&Pachet 92b] est un système de contrôle de respirateurs[51] pour la ventilation artificielle de patients atteints d'insuffisance respiratoire, hospitalisés dans des unités de soins intensifs. Ce système représente l'expertise nécessaire au contrôle de l'assistance mécanique prodiguée au patient, telle qu'elle est utilisée à l'hôpital Henri Mondor (Créteil) et a été validé par des tests

[51] Appelés aussi ventilateurs. Ce sont des appareils complexes comportant de nombreux réglages et dont l'emploi nécessite une longue période d'apprentissage.

en clinique [Dojat&al. 92]. Le système travaille en boucle fermée, récupère des données physiologiques provenant du respirateur et d'appareils de contrôle (monitoring), et en retour modifie les paramètres du respirateur. Le but de la manœuvre est double : d'une part le système doit fournir au patient une assistance respiratoire adaptée, et d'autre part il doit être capable de se rendre compte si le patient est capable de respirer seul, et tenter alors de le sevrer progressivement en diminuant son assistance, en fournissant au clinicien un diagnostic de sa sevrabilité.
Une première version du système (*Ganesh*, [Dojat&al 91a, 91b]) a été développée dans un environnement sommaire d'ordre 0+ [GSI Tecsi 90]. Les limites et contraintes de cet environnement ont conduit les auteurs à une réécriture complète du système de manière à le rendre plus extensible [Dojat&Pachet 92a], conduisant ainsi au système NéoGanesh.
Ce nouveau système utilise toutes les fonctionnalités de représentation de Smalltalk et de NéOpus. Les objets du domaine (appareils de contrôle, respirateur, patients et experts) sont représentés par des classes Smalltalk et l'expertise est représentée par une base de règles NéOpus, qui est contrôlée par une base de méta-règles spécialisée, elle même structurée en cinq niveaux hiérarchiques.

VII.3.2. Traits saillants

L'expertise de contrôle d'un respirateur comporte deux parties clairement séparées : l'expertise du domaine et l'expertise du contrôle. Nous avons profité de la possibilité que nous donne NéOpus pour effectivement séparer ces deux composantes, en écrivant deux bases de règles. Une première base (`ClinicRules`) contient uniquement l'expertise nécessaire à diagnostiquer le patient et à effectuer les réglages sur le respirateur. Le deuxième (`ClinicMeta`) comporte toute l'expertise de contrôle. Détaillons un peu ici les caractéristiques de ces deux bases de règles.

VII.3.2.1. Une base de règles extensible

La base de règles du domaine contient des règles qui filtrent toutes un seul objet, instance de la classe `ExpertVentilateur`. Les règles sont réparties en protocoles (9), chaque protocole étant spécialisé dans une tâche particulière. La gestion de la séquentialité de ces protocoles est assurée par la base de méta-règles. Un point important de cette base est sa généricité au sens (2) (Cf. VIII.1.2.3). En effet, cette base de règles est conçue pour être réutilisée en produisant des sous-classes de `ExpertVentilateur`, implémentant des comportements différents de l'expert "standard". Ceci est utilisé pour deux raisons :

- Utiliser la base de règles dans des modes non standards, par exemple avec des valeurs temporelles réduites, en mode de test. Ici une sous-classe de `ExpertVentilateur` (`ExpertVentilateurSimulation`) déclenche des durées d'observation beaucoup plus courtes.

- Changer certaines valeurs de seuils critiques, sans pour autant changer la base de règles. Par exemple, la règle suivante diagnostique une polypnée quand la fréquence du patient (`expert patientFrequency`) est comprise entre les deux bornes définies par les deux méthodes associées à la classe `ExpertVentilation` (`frequencyThreshold maxFrequency`). Ces méthodes renvoient une valeur qui est hautement heuristique : l'expertise médicale est codée aussi dans cette méthode. Pour changer sa valeur, nous suivons le principe de la programmation par objet, qui consiste à créer une sous-classe de `ExpertVentilation`, redéfinissant simplement les méthodes en question. Ainsi, la base de règles est inchangée et, grâce au typage naturel, seule la classe des instances filtrées par les règles est modifiée.

```
polypnea (frequencyTest protocol)
   |VentilationExpert expert|
   expert patientFrequency > expert frequencyThreshold.
   expert patientFrequency < expert maxFrequency.
actions
   expert ventilation: #polypneic.
   expert report: 'The patient is polypneic. Resp. rate too high'.
   expert modified.
```

VII.3.2.2. Une base de méta-règles structurée par l'héritage

Le contrôle de la base de règles `ClinicMeta` doit prendre en compte les points suivants :

- Séquencement des opérations par la notion de protocole.
- Changement dynamique de la séquence de protocoles suivant le contexte (l'état du patient).
- Déclenchement des règles suivant un mode "parallèle" c'est à dire sans remettre à jour l'état du conflict set à chaque tirage.
- Prise en compte des alarmes : méta-actions de priorité maximale.

Pour représenter ces spécifications de contrôle nous avons structuré la base de méta-règles en plusieurs niveaux en utilisant l'héritage de bases de règles. Chaque niveau représente une spécialisation du niveau précédent, et on a donc 4 bases de méta-règles de plus en plus spécialisées (Cf. figure 27).
Il faut noter ici que les bases de méta-règles `MetaProtocole` et `MetaProtocoleOneShot` sont générales, indépendantes de notre application. Elle peuvent donc être réutilisées pour d'autre applications (en particulier d'autres expertises médicales).
L'héritage de base de règles permet d'accorder naturellement au niveau le plus bas (les alarmes) une priorité maximale, nous épargnant ainsi un troisième niveau méta.

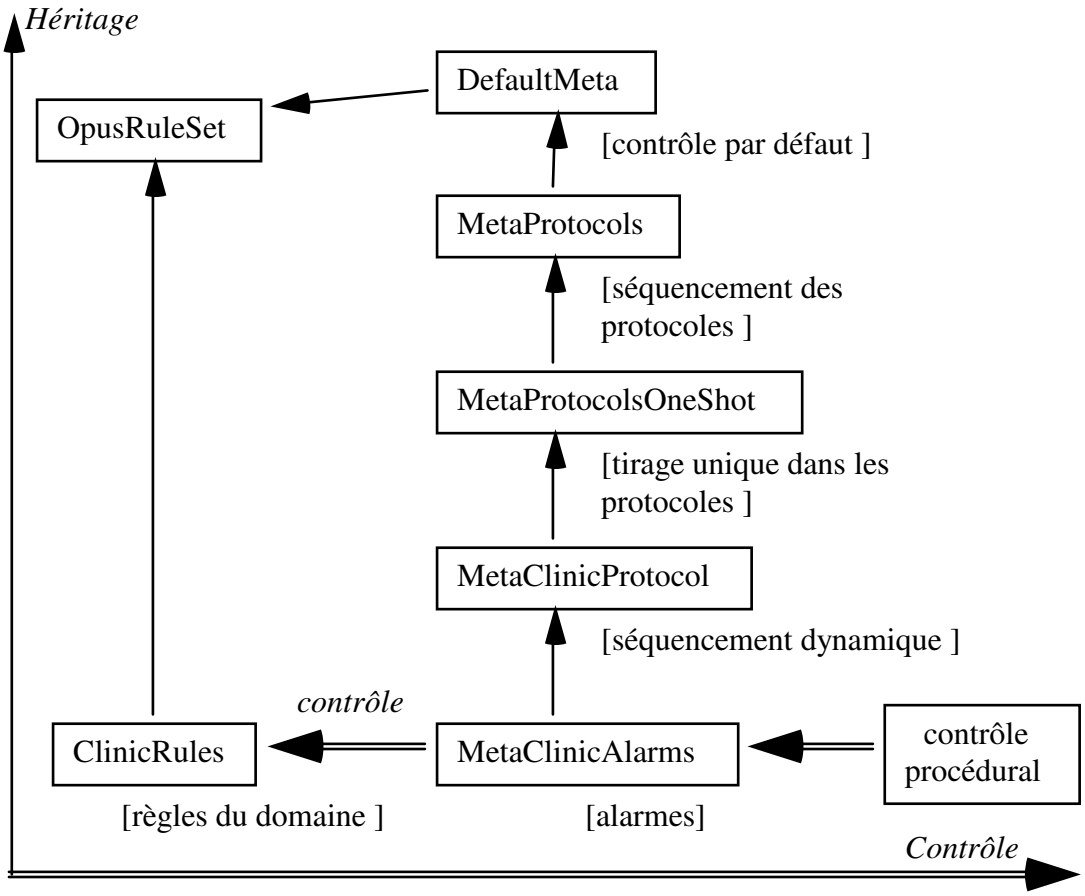


Figure 27. La structure de contrôle de `ClinicRules`.

VII.3.2.3. Extensions prévues

Le système doit à présent incorporer de nouvelles connaissances, en particulier pour la gestion des alarmes qui sont pour l'instant réduites à quelques cas de figure. De plus, une extension vers les acteurs est en cours, afin de permettre une meilleure gestion des ressources : d'une part en permettant plusieurs bases de règles de tourner en même temps (contrôlant plusieurs respirateurs pour plusieurs patients) et d'autre part en donnant un statut autonome au module d'acquisition de données (surveillance des alarmes), qui en l'état actuel du système "bloque" complètement le système, l'empêchant de "réfléchir" pendant ce temps. Enfin, une composante de gestion de l'incertitude est à l'étude, motivée par un besoin de représentation plus fine des notions de seuils.

VII.4. Transformation de graphes sémantiques

VII.4.1. Problème, contexte

Le but de ce projet est de transformer un graphe représentant un schéma conceptuel de base de données sous forme de réseau sémantique simple (et intelligible) en un ensemble de graphes dont les composantes connexes peuvent s'interpréter comme des tables du modèle relationnel classique. Ce travail s'inspire très largement d'une partie du "système expert en conception de systèmes d'informations", SECSI [Bouzeghoub 86].
SECSI comprend notamment un modèle de représentation sémantique, MORSE, qui définit un langage de description de schémas conceptuels de base de données. Ce langage permet de construire des réseaux sémantiques (graphes) en faisant intervenir un petit nombre de concepts (nœuds de graphe) et de relations (arcs) entre ces concepts. Il intègre les notions d'*objet atomique*, d'*objet moléculaire*, de *valeur*, de *type* et de *rôle*. Les objets atomiques peuvent être mis en relation avec un type (relation "d"), ou avec plusieurs valeurs (relation "i/c"). Ils peuvent s'agréger en objets moléculaires par un relation particulière ("p/a"). Les objets moléculaires eux mêmes sont susceptibles de s'agréger en d'autres objets moléculaires, par un autre type de relation ("o/r"), et peuvent également être liés dans une hiérarchie d'héritage (d'objets atomiques) par une nouvelle relation ("g/s"). Il existe enfin une relation entre objets atomiques ("df") qui exprime que deux objets de la même (?) sorte peuvent être en dépendance fonctionnelle...
En plus du langage MORSE, SECSI contient un ensemble de règles de transformation d'un graphe complexe généré dans le formalisme MORSE, en graphes simples interprétables par des tables du modèle relationnel de base de données.
Ces règles sont de deux types. Une première partie de règles gère la modification du graphe notamment pour se débarrasser des liens d'héritage entre objets moléculaires en faisant intervenir la notion de rôle. Un autre ensemble de règles est dédié à décomposer le graphe en éléments simples. C'est l'ensemble de ces règles qui a été implémenté à l'aide de NéOpus.

Ces règles s'appuient sur un ensemble de classes Smalltalk qui n'ont pas tout à fait la structure habituelle : en particulier elles ne possèdent aucune variable d'instance. La raison en est simple : ces classes sont le résultat d'une compilation à partir de spécifications énoncées dans un environnement de conception/prototypage expérimental[52]. Dans cet environnement, on spécifie un *méta-modèle* qui caractérise les types d'éléments que l'on pourra manipuler ainsi que les relations autorisées entre ces éléments. Le système se charge alors de synthétiser les classes du modèle.
Les règles de transformation de réseaux MORSE qui ont été écrites font donc intervenir principalement les relations qui existent entre objets. Toutefois certaines de ces relations sont porteuses de cardinalités qui font éventuellement aussi partie des prémisses de règles (Cf. exemple `trans1c1`).

[52] L'environnement ACK-PRO développé par ACKIA en collaboration avec le LAFORIA.

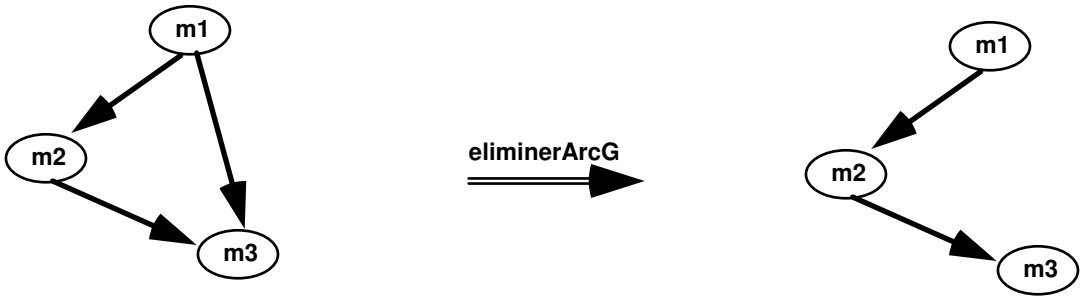
D'autre part, par l'intermédiaire de l'environnement de prototypage, une représentation graphique est associée à la structure Smalltalk d'un modèle. Aussi, pour mettre à jour une telle représentation graphique, les règles contiennent toutes une partie de leur action destinée au placement (ou déplacement) des représentations graphiques associées à chaque objet créé (ou modifié) du graphe transformé. Ceci est réalisé par l'envoi de messages au contrôleur Smalltalk associé à la vue graphique du modèle.

VII.4.2. Exemples

Voici deux exemples tirés de la base de règles de transformation. Nous donnons leur représentation dans le langage MORSE, puis leur représentation en règle NéOpus.

VII.4.2.1. Premier exemple simple

Règle MORSE : "on utilise la transitivité de la relation de spécialisation, pour supprimer l'arc entre une classe et la superclasse de sa superclasse (ici entre `m1` et `m3`) :

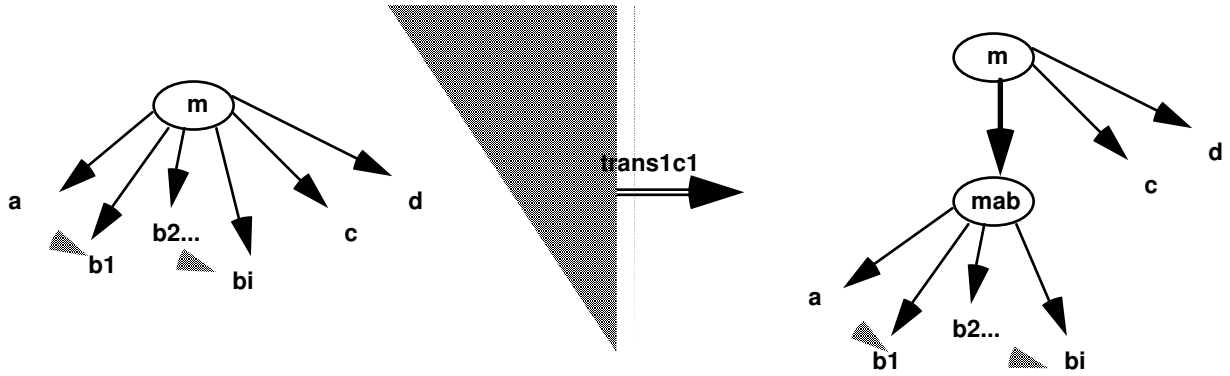


Règle NéOpus : La règle s'écrit naturellement en utilisant les messages `estSousClasseDe:` et `decrocheSousClasse:`, définis dans la structure de représentation MORSE (la classe Smalltalk `MOL`).

```
eliminerArcG
        | MOL m1 m2 m3. Global LeController|
    m3 estSousClasseDe: m2.
    m2 estSousClasseDe: m1.
    m3 estSousClasseDe: m1.
actions
    m1 decrocheSousClasse: m3.
    LeController enleveLien: #MOF entre: m1 et: m3.
    m1 modified.
```

VII.4.2.2. Deuxième exemple

Règle MORSE : "On décroche un ensemble d'atomes (a, b1, .. ,bi) liés en dépendances fonctionnelles (les flèches grises) de l'objet moléculaire qui les agrège (`m`), pour générer un nouvel objet moléculaire (`mab`) constitué de cet ensemble d'atomes, et lié à `m` par une relation d'*agrégation moléculaire*"



Règle NéOpus : Ici encore la règle NéOpus comporte en partie action des messages envoyés au contrôleur pour gérer l'affichage des objets graphiques.

```
trans1c1
|MOL m. ATO a. Global LeController. Local n|
    a estAtomeDe: m.
    a estLieEnDf.
    n := m cardPMaxAvecAtome: a.
    selfBase est: n superieurA: 1.
actions
    |i lesDepsEtA |
    i := m cardPMinAvecAtome: a.
    "extraire les dependances"
    lesDepsEtA := a atomesEnDf.
    "mise a jour de la structure"
    lesDepsEtA do: [:eachDep | m delieEn: #ATO avec: eachDep.
    "suppresssion des ojbets graphiques de type lien"
                    LeController enleveLien: #ATO entre: m et: eachDep].
    "création du nouvel objet moléculaire mab"
    mab := MOL newAtomesEnPUnUn: lesDepasEtA nom: (m nom, '/', a nom,
'...').
    "liaison avec l'ancien"
    m accrocheConstituant: mab cardsO: (Array with: i with: n).
    "placement de l'objet graphique de mab"
    LeController placerRepresentationDe: mab entre: m etListe: lesDepsEtA.
    "placement des objets graphiques de type lien entre mab et ses constituants"
    lesDepsEtA do: [:ato |
            LeController placerRepresentationDArc: #ATO entre: mab et: ato].
    m modified. mab go.
```

VII.4.3. Le contrôle de la base de règles

La transformation de graphes est décrite de manière séquentielle (une succession d'étapes) dans [Bouzeghoub 86]. Nous avons donc structuré les règles en autant de protocoles. La séquentialité de ces protocoles est assurée par la base de méta-règles (générale, non spécifique à cette application) `MetaAgenda`.

VII.4.4. Traits saillants

Bien que cette base de règles ait été pensée au départ comme un simple "exercice de style" elle est intéressante à plus d'un titre.
D'une part elle est une preuve imparable du caractère hautement réutilisable des applications (bien) écrites en Smalltalk, puisque elle combine *trois* environnements développés indépendamment (et *à priori* non conçus pour être combinés ensemble) : l'environnement de définition de méta-modèles AckPro, l'environnement de création d'objets graphiques et NéOpus.

D'autre part elle met en évidence un manque de la programmation par objets pure et simple. En effet, la coordination entre objets et objets graphiques doit être explicite dans notre contexte. Les règles NéOpus fournissent ici un outil méthodologique relativement adapté pour coordonner modèles et représentations graphiques. De plus, l'écriture de règles à été dans cet exemple un "moteur" réificatoire : les besoins en expression des règles ont permis de définir et de structurer les comportements "utiles" des objets (ici les instances de `MOL`), qui étaient définis au départ comme n'ayant aucun comportement particulier (à part les simples méthodes d'accès). Les règles peuvent être vues ici comme un outil méthodologique de définition de classe.

Enfin, notons ici l'emploi de la pseudo-variable `selfBase` (Cf IV.1.7.4.4), qui nous permet de cacher un calcul lourd tout en le rendant réutilisable. En effet, la méthode invoquée ici est implémentée dans la métaclasse de la base de règles (`MorseRules class`) et effectue à la fois des tests de type et de valeurs :

```
!MorseRules class methodsFor: 'premisses'!
est: n superieurA: x
  ^n = 'n' or: [n = 'N' or: [n asNumber > x]]
```

VII.5. MusES : un système d'analyse de séquences d'accords

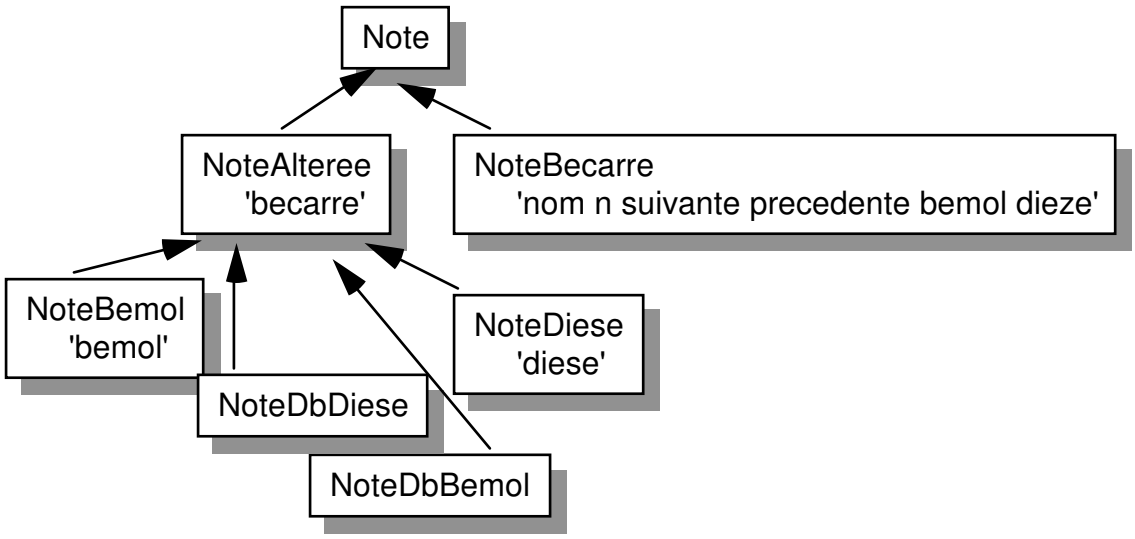
La musique est un terrain particulièrement propice pour la représentation de connaissances par la complexité et la diversité des connaissances mises en jeu. Si beaucoup de travaux en Intelligence Artificielle sont liés à des problèmes musicaux

(représentation de processus parallèles [Gautron 85], problèmes de pédagogie [Rousseau 90], environnements d'aide à la composition [Cointe 84], [Scaletti&Johnson 88]), peu sont concernés par la représentation des connaissances musicales de base, en particulier harmoniques.
Le système MusES[53] est une suite de nos travaux à l'IRCAM [Pachet 88] sur plusieurs systèmes de représentation de connaissances musicales [Pachet 91e, 91f]. Il a été réécrit en Smalltalk/NéOpus, et comporte un certain nombre de connaissances harmoniques, décrites à la fois sous forme d'objets et de règles NéOpus. Nous décrivons ici deux bases de règles de MusES : une base simple permettant de déduire la composition d'un accord en fonction de son appellation, et un système plus complexe d'analyse harmonique de suites d'accords de Jazz.

VII.5.1. Théorie du domaine

La théorie des notes, des intervalles, des gammes et des accords est non triviale. Le système musical occidental est basé sur une série de notes séparées par un ou deux demis-tons (do ré mi fa sol la si). Mais les choses se compliquent avec la notion d'altération (une note peut être bémol, dièse, bécarre, voire double-bémol ou double dièse). En effet il peut y avoir alors plusieurs manières différentes de référencer une *même note* (sur le clavier), mais chaque dénomination porte en elle toute une série d'informations subtiles. Par exemple, il n'y a aucune différence sur un piano entre *Do dièse* et *Ré bémol*. Mais appeler cette note Do dièse dénote une intention tout à fait différente de l'appellation Ré bémol : Do dièse indique quelque chose sur les autres notes de la gamme sous-jacente. Quand on "dit" Do dièse, on dénote une note, et en même temps on énonce une propriété *sur tous les Do de la gamme.* Dire Do dièse c'est aussi dire "tous les Do sont dièses". C'est donc aussi ne rien dire sur les Ré (qui peuvent être bécarres, dièse ou bémols). Dire Ré bémol signifie inversement "tous les Ré sont bémols" mais ne présuppose rien sur les D o.
Sans rentrer dans les détails de la théorie, disons que la représentation n'est pas triviale, si on veut respecter ces différences subtiles.
Nous avons représenté la théorie des notes en utilisant l'héritage, qui pour une fois se prête remarquablement au jeu. Les notes sont représentées par la taxonomie suivante :

[53] MusES pour **Mus**ical **E**xpert **S**ystem.



Ces objets sont capables de représenter la série des notes naturelles, avec les altérations et de calculer des intervalles. Une classe `Intervalle` est introduite, pour représenter cette notion fine (pour le mêmes raisons qu'évoquées plus haut, les intervalles portent dans leur dénominations des informations de nature fort diverses. Ainsi un intervalle ne se réduit pas à un nombre de demis-tons, c'est aussi une indication harmonique importante. Les notions de `Gamme` et d'`Accord` sont ensuite introduites, puis les notions d'`Analyse` et de `Clé`, pour former une théorie relativement complète des objets harmoniques de base.

Exemples de sessions avec le système MusES :

Par exemple la quinte juste de Fa# est calculée en prenant la quinte de Fa (le bécarre de Fa#) qui est elle-même calculée en égrenant[54] les notes de base : Fa Sol La Si Do : Do, à laquelle on rajoute un dièse pour que la somme des demis-tons fasse 7 (une quinte juste fait 7 demis-tons) :

```
(Note noteNamed: 'Fa#') quinteJuste --> Do#
```

De même, la tierce majeure de Fa double dièse est-elle La double diese (et non pas Si !!) :

```
(Note noteNamed: 'Fa##') tierceMajeure --> La##.
```

Et ainsi de suite :

```
(Note noteNamed: 'Dob') quinteDiminuee  --> Solbb
```

Avec une perle pour la septième diminuée de do bémol :

```
(Note noteNamed: 'Dob') septiemeDiminuee -->
        erreur: intervalle interdit[55]
```

On peut ainsi calculer les notes des gammes majeures, mineures harmonique et mélodique dans tous les tons :

```
(Note noteNamed: 'Do#') gammeMineure --> (Do# Re# Mi Fa#
Sol# La Si# )
```

On rajoute aussi la notion d'accords générés par une gamme, puis de transposition par intervalle :

```
Note do transposeDe: (N mi intervalleAvec: N fa ) -->  Reb
```

Générons par exemple tous les accords de 5 notes sur Ré majeur :

```
Note re gammeMajeure genereAccordsPolyphonie: 5.
 --> OrderedCollection ((Re maj7 9) (Mi min 7 9) (Fa# min 7
dim9) (Sol maj7 9) (La 7 9) (Si min 7 9) (Do# min dim5 7
dim9))
```

[54] L'algorithme est décrit ainsi, de manière très compacte dans [Dubois 21]. L'être humain aime égrener (Cf. le calcul de la date du lendemain, Ch. VI.2.7). Nous ne représentons pas ici l'égrenage aussi finement et nous contenterons d'un égrenage procédural.

[55] En effet, do bémol ne peut avoir de septième diminuée : elle s'appelerait alors Si triple bémol ?. Or les triples bémols ne peuvent exister sous peine de déclencher une cascade infinie de quadruple, puis quintuple bémols ... Aucune théorie du solfège ne mentionne ce cas, à notre connaissance.

Il est alors aisé de calculer toutes les analyses harmoniques *possibles* pour un accord donné (dans les trois gammes : majeure, mineure harmonique et mineure mélodique) :

```
(Accord newFromString: 'Re min 7') tonalitesPossibles
   --> ListeDAnalyses ({II de do Majeur} {III de Sib Majeur}
{VI de Fa Majeur} {IV de La MineureHarmonique} {II de Do
MineureMelodique})
```

Notons enfin que notre théorie des notes et des intervalles n'est pas du tout exhaustive. En particulier on ne tient pas compte ici des octaves : il n'y a qu'une seule instance de Do, Ré ... Mais les discours que nous voulons tenir sur ces notes ne prennent pas ces notions en compte.

VII.5.2. Deux bases de règles

Muni de notre théorie des objets du domaine, nous allons maintenant écrire des connaissances portant sur ces objets.

VII.5.2.1. Formation des accords de jazz

La nomenclature des accords de Jazz suit des lois assez particulières et nécessite un certain apprentissage. De plus ces lois varient d'une école à l'autre [Baudoin 90]. Le problème est de trouver les notes d'un accord en fonction de son nom. Nous avons écrit ces lois sous forme de règles NéOpus portant sur les objets accords. Un exemple de telle règle est la formation des accords parfait : un accord parfait majeur est composé de la tonique, de la tierce majeure et de la tierce mineure. L'accord parfait mineur, lui se distingue de l'accord parfait majeur uniquement par sa tierce qui est mineure. D'autres règles permettent de déduire des notes non déclarées. Par exemple, un accord dit "de onzième", sans déclaration de septième (resp. neuvième), sous-entends une septième mineure (resp. neuvième majeure).
Cette base comporte 22 règles et permet de reconnaître tous les accords de Jazz standards.

Exemples de règles :

Composition des accords parfaits majeurs :

```
majeur
"un accord majeur comprend la tonique, la tierce majeure et la quinte juste"
  | Accord unAccord. Local saTonique|
    unAccord estMajeur.
    saTonique _ unAccord tonique.
    (unAccord contientNote: saTonique tierceMajeure) not.
actions
    unAccord ajouteNote: saTonique.
    unAccord ajouteNote: saTonique tierceMajeure.
    unAccord ajouteNote: saTonique quinteJuste.
    unAccord modified
```

Sous entendu des neuvièmes dans les accords de onzième :

```
onziemeImpliqueNeuvieme
"Pour tous les accords de onzieme, on suppose la neuvieme sauf si mention
explicite du contraire"
  | Accord unAccord. Local saTonique|
    unAccord aUneOnzieme.
    unAccord aUneNeuvieme not.
    saTonique _ unAccord tonique.
    (unAccord contientNote: saTonique neuviemeJuste) not.
actions
    unAccord ajouteNote: saTonique neuviemeJuste.
    unAccord modified
```

Théorie de la quinte augmentée : déclarer une quinte augmentée a deux effets : ajouter une quinte augmentée, et supprimer la quinte juste (ce qui la différencie de la treizième diminuée qui, elle, ne touche pas à la quinte quelle qu'elle soit) :

```
quinteAugmentee
"pour un accord de quinte augmentee, c'est LA quinte qui est augmentee. Il ne
doit pas y avoir de quinte juste (par opposition aux accords 11augmentee, qui
AJOUTENT une quinte augmentee)"
  | Accord unAccord. Local saTonique|
    unAccord aUneQuinteAugmentee.
    saTonique _ unAccord tonique.
    (unAccord contientNote: saTonique quinteAugmentee) not.
actions
    unAccord enleveNote: saTonique quinte.
    unAccord ajouteNote: saTonique quinteAugmentee.
    unAccord modified
```

Les règles sont regroupées en protocoles, en fonction des enrichissements progressifs. Ces protocoles sont utilisés ensuite en séquence, par la méta-base `MetaAgenda`. La liste des protocoles reflète l'ordre naturel des calculs : 1. tonique, 2. accords parfaits, 3. quintes, 4. septiemes, 5. neuviemes, 6. onziemes et 7. treiziemes.

Exemple
La base de règles ainsi décrite est utilisée en "mode démon", via la méthode `touteTesNotes` de la classe `Accord` :

```
!Accord methodsFor 'acces demoniaque'!

toutesTesNotes
  self ruleBase executeWithSingleObject: self.
  ^toutesLesNotes

ruleBase
  ^Opus ruleBaseNamed: #FormationDesAccords
```

Par exemple, la transmission suivante :

```
(Accord fromString: 'Re min 7 dim5 dim9 ) toutesTesNotes
```

renvoie, après lancement de la base de règles :

```
#(Re Fa Lab Do MiB)
```

VII.5.2.2. Analyse de grille

VII.5.2.2.1. Cadre et énoncé du problème

Le problème de l'analyse d'une grille d'accords est le principal problème que le musicien de Jazz doit résoudre avant de commencer à jouer. Il s'agit pour une grille d'accords donnée de déterminer quelles sont les tonalités sous-jacentes pour chaque accord de la grille. Ces tonalité vont permettre de déterminer la gamme sur laquelle il est licite[56] d'improviser.

Ce problème est intéressant à représenter dans notre cadre car :

- Il est non trivial. Il n'existe pas d'algorithme qui résolve le problème de manière satisfaisante.
- Il existe un très large corpus d'exemples commentés (les standards de Jazz les plus connus se trouvent dans [RealBook 81], [FakeBook86] ou [New RealBook] pour l'analyse desquels il existe un consensus.
- Le problème fait intervenir des connaissances à la fois structurelles, provenant du solfège et de l'harmonie standard mais aussi très heuristiques. Aucun livre d'harmonie ne décrit d'algorithme ni de mode d'emploi pour analyser. Seules des

[56]Il est bien entendu que si la théorie interdit certaines notes en fonction du contexte harmonique, en réalité (en Jazz aussi bien qu'en musique classique) toutes les notes sont permises. Mais, comme le souligne Dubois dans [Dubois 21] (Cf introduction) les licences doivent être faites en connaissance de cause.

règles "inverses" sont proposées, à partir desquelles l'analyse doit se faire, et s'apprendre tout seul.

Ainsi, l'analyse d'une grille d'accord tout comme celle d'une phrase d'un langage quelconque, peut se décrire à l'aide d'un certain nombre de règles de transformation, comme le propose [Steedman 84] avec une grammaire générative pour les grilles de blues. Utilisées en chaînage arrière, ces règles permettant d'analyser toutes les grilles d'accords licites. Mais cette approche ne convient pas tout à fait à notre problème, pour plusieurs raisons :

- Elle suppose que l'on connait à l'avance toutes les structures minimales de grilles. C'est vrai pour le blues en 12 mesures qui a été longuement étudié, mais c'est faux en général. La plupart des grilles de Jazz n'ont pas de structure minimale connue (par ex: *Good Bye Pork Pie Hat* de Charlie Mingus). Elle sont pourtant tout à fait analysables.
- Cette théorie ne supporte que les grilles parfaitement correctes. En particulier il est difficile dans ce cadre d'énoncer certaines irrégularités que se permettent les compositeurs modernes. Par exemple, le remplacement plus ou moins aléatoire des accords de tonique par des accords de septième de dominante.

C'est pourquoi nous choisissons la démarche inverse, consistant, une fois de plus, à greffer au dessus de notre représentation des objets, décrite plus haut, une couche de représentation déductives en marche avant. Le problème est alors envisagé de la manière suivante :

- Repérer des formes connues dans la grilles. Les formes connues sont des petites séquences d'accords, pour lesquelles l'analyse est immédiate. Par exemple les fameux 2-5-1, ou les anatoles (turnover en Anglais). Ces (reconnaissances se laissent alors bien représenter par règles de production, puisque leur effet et de créer une forme reconnue suivant un contexte déterminé.
- Tenter de "coller" les bouts restant à ces îlots analysés. Cette démarche a déjà été décrite pour résoudre un problème similaire : celui de l'analyse hiérarchique d'une phrase musicale (une suite de notes) dans [Lerdhal&Jackendoff 83]. Un certain nombre de règles (qualifiées de règles de "grouping") permettent ainsi de proposer des groupements d'accords raisonnables.
- Nous ajoutons enfin un élément harmonique qui consiste à minimiser le nombre de modulations, i.e, de changement de tonalité.

VII.5.2.2.2. Objets introduits pour l'analyse

Les objets reconnus par l'analyse sont représentés par des classes Smalltalk organisées en un arbre d'héritage. On distingue alors les structures amorphes (une suite d'accords contigüs), et les structures connues comme `DeuxCinq`, `Anatole`, `MarcheHarmonique` (Cf. figure 28).

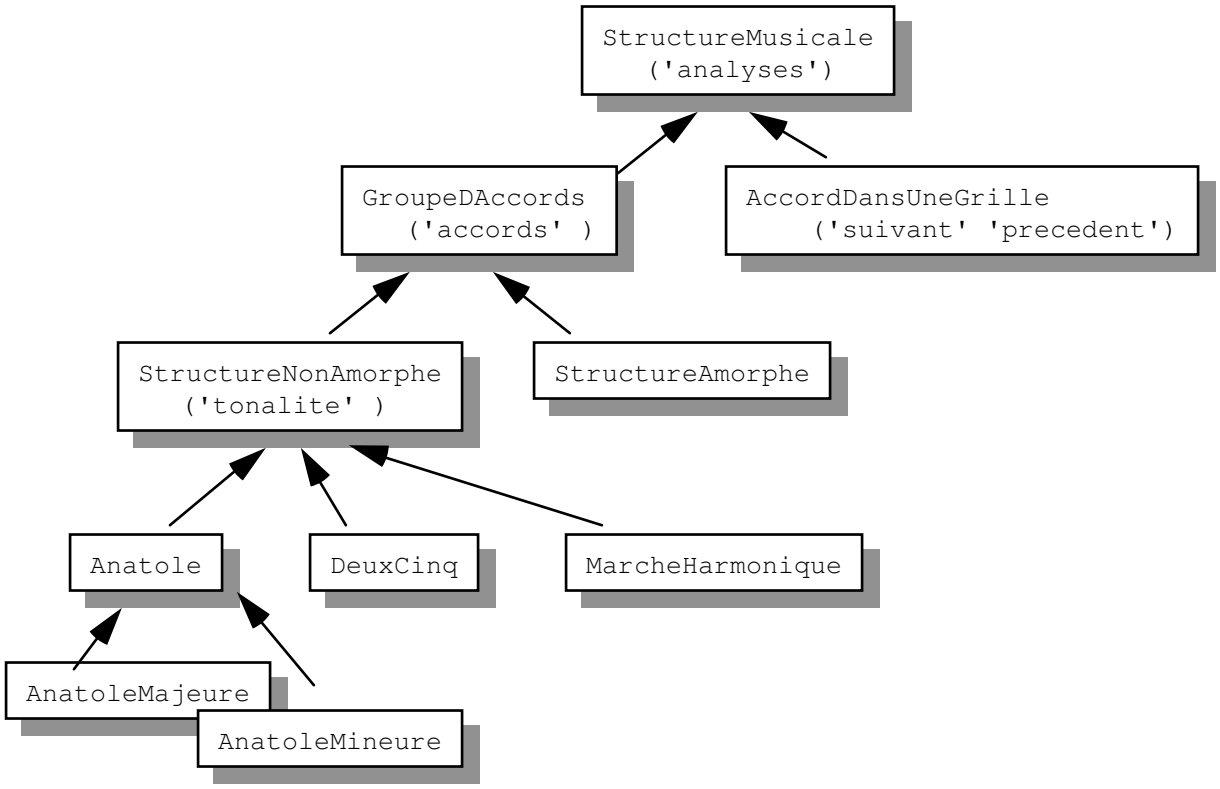


Figure 28. Structures musicales à reconnaître

L'analyse d'un accord se représente par une variable d'instance, que l'on va rajouter à la classe Accord, et représentant toutes les analyses possibles ou déduites pour cet accord. Ainsi, la classe `Accord` est redéfinie comme sous-classe de `AccordDansUneGrille` (Cf. Figure 28) :

```
AccordDansUneGrille subclass: #Accord
   instanceVariableNames: 'tonique structure notes tonalitesPossibles'
```

La méthode `tonalitesPossible` dans la classe `Accord` permet de calculer toutes les analyses possibles d'un accord, conformément à la théorie standard (Cf. la micro-session au §VII.5.1).
Le résultat d'une analyse est donc d'ajouter ou d'enlever des analyses possibles de cette liste, via les messages `ajouteAnalyse:` et `enleveAnalyse:`. Des messages d'accès permettent de connaître les analyses courantes comme : `estAnalyse`.

VII.5.2.2.3. Exemples de règles

Les règles de reconnaissances de structures musicales s'énoncent sans problème. La partie condition cherche tous les n-uplets d'accords successifs, vérifiant certaines

contraintes. La recherche de n-plets successifs se fait en utilisant les variables locales déclenchantes, conformément à notre théorie (Cf. Chapitre V.3 et le principe du `modified`).

VII.5.2.2.3.1. Règles de reconnaissances

Par exemple voici la reconnaissance d'une anatole : 4 accords consécutifs reliés entre eux par une contrainte de type : 1 Majeur/6 Mineur/2 Mineur/ 5 Septième de dominante :

```
!AnalyseGrilleRules2 methodsFor: '1.structures classiques'!
anatole
"anatoles en : 1 - 6 - 2 - 5"
  | Accord c1 c2 c3 c4|
    c1 estMajeur.
    c2 <- c1 suivant.
    c3 <- c2 suivant.
    c4 <- c3 suivant.
    c3 aUneSeptieme.
    c2 tonique = (c1 tonique + #sixteMajeure).
    c3 tonique = (c1 tonique + #secondeMajeure).
    c4 tonique = (c1 tonique + #quinteJuste).
actions
    |x|
    x <- AnatoleMajeure fromChords: (Array with: c1 with: c2 with: c3 with: c4).
    x tonalite: (c1 tonique gammeMajeure). x go.
```

La partie action de la règle consiste à créer un nouvel objet représentant la forme détectée, ici `AnatoleMajeure`.

VII.5.2.2.3.2. Règles de grouping

Les règles de grouping vont être appliquées après la phase de reconnaissance. Elles permettent de grouper des accords ne faisant pas partie d'une structure connue, mais analysables dans la même tonalité.
Par exemple, voici une règle qui dit que si un accord est après une structure connue, et qu'il peut s'analyser dans la tonalité de cette structure alors on l'analyse ainsi :

```
analyseApres
"on analyse un accord suivant un accord analyse dans la meme tonalite si possible"
  | Accord c1 c2 |
    c1 estAnalyse.
    c2 <- c1 accordSuivant.
    c2 nonAnalyse.
    c2 nonAmbigu.
    c2 tonalitesPossibles includes: c1 analyse.
actions
    c2 ajouteAnalyse: c1 analyse. c1 modified.
```

Une règle qui dit que les accords d'une structure s'analysent dans la tonalité de la structure. Noter ici l'usage du typage naturel, pour filtrer toutes les instances de toutes les structures non amorphes, et l'usage subtil du pseudo-message `areModified`, envoyé à tous les accords de la structure, et permettant ainsi de n'écrire qu'une seule règle pour toutes les structures, quelques soient leurs nombres d'accords[57] :

```
analyseStructureNonAmorphe
"les deux-cinq et autres anatoles.
Cette regle est rusee : elle marche pour toutes les structures non amorphes (typage naturel) et quelque soit le nombre d'accords (grace a l'emploi ruse de areModified)"

  | StructureNonAmorphe s. Local sesAccords |
    s tonalite exists.
    sesAccords _ s accords.
"un des accords au moins est non analyse"
    (sesAccords select: [:x | x estAnalyse not]) size > 0.
actions
    sesAccords do: [:ac | ac analyse: s tonalite].
    sesAccords areModified
```

VII.5.2.2.4. Discussion

Remarque sur le typage naturel et la conception objet

Cet base de règles est très riche en enseignements. En particulier, l'utilisation du typage naturel ici à clairement influencé la conception des objets du domaine. Dans notre première représentation en effet, la classe `StructureNonAmorphe` n'existait pas, car elle était inutile. Une classe abstraite (au sens de la programmation objet)

[57] Nous évitons le problème rencontré dans l'exemple des n reines, ou l'on devait écrire une règle par n-uplet de reines possibles.

`StructureMusicale` définissait les structures en général, puis une sous-classe `StructureAmorphe` définissait le comportement particulier des structures amorphes. Enfin autant de sous classes de `StructureMusicale` étaient définies pour représenter toutes les structures connues.
Mais le besoin apparût de parler des structures *non amorphes* (règle `analyseStructureNonAmorphe`). Il fallut alors regrouper toutes les classes des structures autre que `StructureNonAmorphe` et ainsi donner naissance à une classe particulièrement abstraite : `StructureNonAmorphe`, ne comportant aucune variable d'instance ni méthode, mais simplement servant à *parler* des structures non amorphes en général, via le typage naturel.
Notons qu'un tel exemple existe en Smalltalk avec les contrôleurs. La classe abstraite `Controller` définit les comportements de base des contrôleurs. La classe `NoController`, sous-classe de `Controller` définit le comportement particulier des contrôleurs "amorphes". Mais il n'existe pas de classe `YesController`, inutile puisque toutes les sous-classes de `Controller` n'ont rien en commun qui ne soit déjà dans `Controller`.

VII.6. Autres bases de règles

Nous ne pouvons pas détailler ici toutes les bases de règles écrites en NéOpus. Signalons simplement l'existence de quelques unes d'entre elles :

VII.6.1. Un système d'analyse intelligente d'images tomoscintigraphiques

Le projet IAMI (Intelligent Analysis of Medical Images) [Forte&Bizais 91], [Forte 91] est basé sur une approche objet pour représenter les images médicales tomoscintigraphiques pulmonaires[58] (notion d'objet-image [Forte 91, Lavaire 91b]), et les traitements associés permettant leur analyse. Après une approche initiale basée sur Prolog, Smalltalk/NéOpus à été choisi [Forte&al. 92], [Lavaire &al. 91] pour bénéficier à la fois des capacités de représentation de Smalltalk et de capacités inférentielles. Plusieurs bases de règles, organisées hiérarchiquement grâce à l'héritage de bases de règles servent à piloter l'ensemble du système, en l'occurrence : piloter l'interface utilisateur, traiter les choix de l'utilisateur, interroger une base d'images, choisir et activer les procédures de traitement d'images (écrites en C, et appelées depuis Smalltalk), et récupérer les résultats.

[58] La tomoscintigraphie est une technique permettant de réaliser des séries d'images en coupe (tomo) à l'aide d'une gamma-caméra (scintigraphie) après injection de micro-sphères radioactives. Appliquée au poumon, les images ainsi obtenues permettent aux médecins de diagnostiquer la gravité d'embolies pulmonaires.

VII.6.2. Un générateur de scénarii

VII.6.2.1. Problème, contexte

Ce système [Benhouhou92a, 92b] s'inscrit dans une étude sur la simulation de processus évolutifs dans le temps, en particulier les catastrophes naturelles[59]. Plus précisément il s'agit ici de générer automatiquement des *scénarii* de catastrophes décrivant des sinistres naturels. Ces scénarii doivent servir à des fins de formations d'officiers de la sécurité civile.
Le système a pour but de compléter des scénarii initiaux en fonction d'expertises (en particulier météorologiques) propres aux services de la Sécurité civile.
Les règles NéOpus permettent, à partir d'un scénario simple initial, de déduire l'apparition d'*événements* nouveaux en fonction de certaines caractéristiques d'événements initiaux.

Exemple
Toutes les règles de la base ont la même structure : elles filtrent une instance (supposée unique) de `Scenario`, et des instances des classes d'événements correspondant à l'énoncé de la règle.
Voici une règle typique de l'application : elle dit que la présence simultanée de `Vent` et de `ChuteDeNeige` "légère" et de hauteur > 10, provoque l'apparition de `Congere`.

```
!MBRules methodsFor: 'meteo'!
congeres
|Scenario s.  Vent v. ChuteDeNeige c |
s aEvenement: v.
s aEvenement: c.
s nAPasEvenementDeType: ApparitionDeCongeres.
c simultaneA: v.
c procheDe: v.
c hauteur > 10.
c qualite = #legere.

actions
|congeres time |
congeres _ ApparitionDeCongeres new.
time _ (v date min: c date).
"la congere apparait 10 minutes plus tard"
time_ time addTime: (Time readFromString:'0:10').
congeres date: time ; lieu: c lieu.
s ajouteEvenement: congeres.
congeres go.        s modified.
```

[59] Dans le cadre du projet Sagesse du Ministère de l'Intérieur pour la réalisation de systèmes d'aide à la décision en situations de crise.

Il faut noter dans cette règle plusieurs points caractéristiques :

- les prémisses `s aEvenement: v` et `s aEvenement: c` ont un caractère purement opérationnel : elles servent simplement à assurer la liaison entre les variables de la règle,
- la prémisse `s nAPasEvtDeType: ApparitionDeCongeres`, sert uniquement à empêcher la règle de boucler (Cf V.1.2.5),
- les prémisses véritablement *significatives* sont donc les 4 suivantes,
- la partie action de la règle consiste à créer une nouvelle instance de `Congere`, correctement initialisée, et à la faire prendre en compte par le système (`congeres go`).

Les prémisses à caractère purement opérationnel (ici les les 3 premières) sont appelées "prémisses de service" et seront générées automatiquement par le système. La partie action suit elle aussi toujours la même structure : création d'un nouvel événement, mise à jour du scénario avec cet événement, et prise en compte de l'événement par le système.

VII.6.2.2. Traits saillants

Génération dynamique des règles

Un point intéressant de ce système est la possibilité à l'utilisateur de créer ses propres classes d'événements. Il a alors la possibilité aussi de définir les règles d'apparition pour cette classe d'événement. Le système demande en ce cas à l'utilisateur de préciser uniquement les prémisses *significatives* (à l'aide d'un Browser particulier), et se charge tout seul de compléter la règle NéOpus. Il génère et compile la règle, qui est alors ajoutée à la base de règles initiale, et prête à fonctionner pour la prochaine génération de scénario.

Les trois systèmes suivants ont été réalisé dans le cadre du cours de [IARFA90, IARFA 91] :

VII.6.3. Un système qui joue au "Compte est Bon"[60]

En représentant explicitement les stratégie possibles et simples (comme "diviser le nombre cible par une des plaques", retrancher un des nombres et multiplier par un autre, etc). Le système obtient des performances tout à fait raisonnables par rapport aux humains.

[60] De la célèbre émission d'Armand Jammot sur la deuxième chaîne française "les chiffres et les lettres".

VII.6.4. Un système de gestion retour arrière "objet"

L'idée est de permettre de faire des retours arrières au cours du raisonnement, pour pouvoir émettre des hypothèses et au besoin les infirmer, et alors défaire les inférences. Le mécanisme proposé ici n'est pas du type "maintenance de la vérité" comme dans les systèmes à TMS [Doyle 79] ou ATMS [deKleer 86a] pour des raisons fondamentales, précisées au Chapitre VIII.1.2.4.2. Nous ne suivons pas non plus l'approche de [LaLonde&Pugh 88] consistant à greffer sur Smalltalk un mécanisme de backtrack de type Prolog, pas généralisable dans notre contexte de marche avant.
En revanche, l'idée est de fabriquer une classe (abstraite) d'objets capable de gérer eux-même les retours arrières, et donc de sauvegarder/restaurer les valeurs de leurs variables d'instance à la demande, de manière synchronisée. Cette idée des objets backtrackables avait déjà été utilisée dans le contexte de la fabrication d'un simulateur de situation de crise [Pachet 89], et a été reprise ici et étendue. Le système tourne sur les exemples classiques de crypto-arithmétique (send + more = money), et le problème des cinq maisons (étant donné certaines contraintes sur les habitants et leur coutumes, déduire qui habite dans quelle maison).

VII.6.5. Calcul de surface corrigée

Ce système permet de calculer la surface corrigée d'un appartement, en fonction de certaines caractéristiques de l'appartement (surface corrigée des pièces, coefficient d'habitabilité des pièces, ensoleillement, éclairement).
Smalltalk est utilisé ici à la fois pour représenter les objets du domaine (l'appartement, les pièces, les fenêtres), et pour représenter/saisir ces objets graphiquement, via un interface graphique. La base de règles (30 règles) est utilisée dans un mode "démon" : à chaque modification d'une caractéristique sensible de l'appartement, celle-ci est automatiquement redéclenchée pour mettre à jour la surface corrigée.

VII.6.6. Un système d'explication au bridge

[Alvarez 91] utilise NéOpus pour tester des techniques originales de génération d'explication. Deux bases de règles représentent les expertises nécessaires à jouer la première et la deuxième carte aux enchères de bridge. Une base de méta-règles permet de contrôler le lancement répété de ces bases de règles et de synthétiser un historique des résultats, qui est ensuite analysé. Les deux bases de règles sont contrôlées par la même méta-base. Celle-ci est donc dédoublée (Cf. le problème de la métaclasse cachée) pour permettre le contrôle simultané des deux bases.

VII.6.7. Un environnement d'assistance à la programmation

L'idée de cette application [Bernier 91] est d'utiliser NéOpus comme système de gestion sophistiqué de l'environnement Smalltalk, dans l'optique "Génie logiciel" d'un environnement d'assistance à la programmation. L'objectif est d'étudier la capacité de NéOpus à représenter de manière satisfaisante les connaissances nécessaires à la réalisation d'outil d'aide à la programmation.
Une première réalisation a été faite consistant à réécrire le formateur et l'explicateur Smalltalk (écrits en Smalltalk sous forme procédurale et très sommaires) sous forme de règles NéOpus, pour en augmenter la lisibilité, les performances, et le rendre plus maintenable/modifiable. L'étude[61] du formateur Smalltalk à mis en évidence les points suivants :

- Les objets représentant l'*arbre syntaxique* sont très bien organisés, sous forme d'un arbre de classes dont la racine est `ProgrammeNode`, et comportant autant de sous-classes qu'il existe de types d'expressions Smalltalk.
- En revanche, le formatage d'un arbre syntaxique est représenté de manière complexe, difficile à modifier, sous forme de méthodes disséminées dans toutes les classes considérées.

Le formateur Smalltalk a donc été complètement disséqué pour séparer d'une part les connaissances relatives au formatage de telle ou telle partie de code, et d'autre part l'aspect "parcours d'arbre" du formateur. Cette séparation entraîne l'écriture de règles NéOpus très simples, dans lesquelles on filtre deux objets : d'une part le nœud de l'arbre syntaxique formaté, et d'autre part un objet `Formateur` qui représente le texte une fois formaté (le résultat). Les prémisses consistent alors simplement à indiquer quelles sont les modifications locales à apporter au texte résultat en fonction d'un type donné de nœud.
Une autre application envisagée et plus intéressante de cette approche est l'écriture d'un explicateur Smalltalk, plus élaboré que celui de l'environnement (le *explain*).

VII.6.8. Vers un système d'acteurs

Smalltalk se prête remarquablement bien à la simulation de processus concurrents [Bézivin 84, 86]. Ceci est du en partie au fait que les objets de base utilisés par le système pour le contrôle de l'exécution (les processus, les sémaphores, le contrôleur de processus (`ProcessorScheduler`)) sont de véritables objets Smalltalk, dont le comportement peut être redéfini [Bézivin 88]. Un certain nombre de travaux étudient justement l'extension de Smalltalk vers les langages d'acteurs, comme le système Actalk [Briot 89], ou ACTRA [Lalonde&al. 86]. Il est naturel d'envisager l'utilisation de NéOpus dans ce cadre. Un tel développement est en cours, suivant le modèle multi-agents MACE de L. Gasser [Gasser 87], sur un exemple de pollution chimique, et utilisant le simulateur de temps partagé du système *Concurrent Smalltalk-90* [Hokamura&Tokoro90].

[61] On devrait dire plus honnêtement "l'étude-orientée-NéOpus".

Dans le cadre de NéOpus il se pose deux problèmes principaux à l'exécution concurrente de bases de règles :

(1) le problème de la métaclasse cachée

Il est impossible comme nous l'avons vu de lancer l'exécution simultanée de la même base de règles, avec deux contextes d'initialisation différents. Par exemple, sous l'environnement de Concurrent Smalltalk-90, la transmission suivante aboutit à une situation inconsistante :

```
[FiboRules executeWithSingleObject: (Fibo new arg: 10)] fork.
[FiboRules executeWithSingleObject: (Fibo new arg: 10)] fork.
```

Parce que la même base de règles est activée sur des contextes initiaux différents.

(2) le problème de la propagation des changements

La modification d'un objet par une base de règles peut remettre en cause l'état d'instanciation de règles d'autres bases de règles.

Le problème de la métaclasse cachée peut se résoudre par l'héritage de bases de règles (Cf. Chapitre IV.1.7.8), en créant autant de sous-bases que d'utilisations concurrentes désirées. Le second problème est beaucoup plus complexe, puisqu'il faut définir un protocole d'accès pour les objets susceptibles d'être utilisés par plusieurs bases de règles.

VIII. Vers un schéma triadique

Avant-propos

Cette partie synthétise notre expérience du NéOpus comme système de représentation de connaissances. Dans une première partie nous résumons les principaux apports du système en tant que *mécanique* d'inférence. En particulier nous synthétisons les conséquences de nos choix fondamentaux (Principes Tout objet, Toute Expression et de Fermeture).
Dans une deuxième partie, nous constatons un écueil récurent des systèmes écrits en NéOpus et tentons une explication épistémologique. Nous proposons alors une *philosophie générale* de l'utilisation de NéOpus et un élément de réponse au problème fondamental de la représentation des objets/discours sur les objets, en introduisant la notion abstraite d'*aspect*.

VIII.1. Synthèse de la mécanique NéOpus

En tant que système de production intégré à l'environnement Smalltalk, NéOpus peut être considéré comme une double extension : extension par rapport au langage Smalltalk lui-même, ou extension par rapport à un système de production traditionnel.

VIII.1.1. Apport de NéOpus par rapport à Smalltalk

VIII.1.1.1. Des règles comme expression du désordre

D'une part l'apport de NéOpus par rapport à Smalltalk est trivial, puisqu'il ajoute à Smalltalk un nouveau mécanisme de programmation. En particulier Smalltalk comme tous les langages de programmations procéduraux est basé sur un mode d'expression *séquentiel* en *chaînage arrière* [Perrot 89b]. NéOpus permet de programmer dans un mode non séquentiel, et en chaînage avant. [Laurière 87] résume cette idée classique de règle de production en soulignant la polysémie du mot ordre. Ordre désigne en effet à la fois le caractère impératif des instructions et leur caractère séquentiel (i.e. séquence fixe et déterminée par le programmeur). En ce sens donc, NéOpus est un moyen d'introduire du désordre.

VIII.1.1.2. Des règles comme description de systèmes

D'autre part, si la programmation par objets permet de représenter de façon assez satisfaisante certains comportements associés aux *objets* du discours, il n'en est pas de même pour spécifier des comportements associés à des *groupes d'objets*. Une règle (ou un ensemble de règles) NéOpus peut donc être vue comme un moyen de spécifier un *système* de manière comportementale. Dans ce cadre, NéOpus doit être comparé à de véritables systèmes de représentation systémiques par objets, comme Systalk [Wolinski 90]. La différence principale avec un tel système est que les systèmes que décrivent les règles sont des systèmes fantômes et éphémères n'ayant pas d'existence à proprement parler. D'autre part les systèmes ainsi décrits n'ont pas de représentation structurelle.

VIII.1.2. Apport de NéOpus par rapport aux langages à règles de productions

L'apport de NéOpus par rapport aux autres systèmes hybrides a deux sources :

VIII.1.2.1. Des *pré-objets* aux objets

La richesse relative des objets Smalltalk induit une plus grande puissance d'expression des règles. Comme nous l'avons vu en introduction, les systèmes de production traditionnels (Mycin, Snark, OPS5, dans une certaine mesure Kee, Art,

Essaim) reposent sur une représentation des objets du discours très frustre : relations binaires pour Snark, relations n-aires pour OPS5, frames pour Art. Toutes ces représentations héritent d'une même racine : la relation binaire, associant attributs et valeurs.
Les prémisses de règles s'expriment alors comme des contraintes sur les valeurs d'attributs. En Opus, les prémisses peuvent être tout message Smalltalk booléen. Par la nature même des objets, accessibles uniquement par envoi de message, la notion d'attribut disparait totalement pour faire place à l'expression de relations entre objets, ces relations étant l'expression d'un calcul. De même pour les parties action des règles, qui, traditionnellement consistent en modifications d'attributs d'objets. Ici, tout message Smalltalk étant licite, la notion d'action est étendue à tous les messages compris par les objets.

VIII.1.2.2. Un phénomène de *contagion* réussie

Les mécanismes de NéOpus originaux sont le résultat d'une contagion culturelle réussie, en l'occurrence celle de OPS5 au contact de la culture objet. En effet, les mécanismes "objet" ont infiltré OPS5 au point de le métamorphoser. Rappelons en ici les symptômes :

Extension de la notion de *variable* de règle, pour prendre en compte les notions Smalltakiennes de variables : variable locale, variable globale et variable de classe.

Prise en compte des *dépendances fonctionnelles* entre objets (notion de variable locale déclenchante).

Notion de *typage* des variables de règle (naturel/simple) pour prendre en compte l'héritage de classe.

Notion d'*héritage* de base de règles, qui n'est autre que la traduction directe d'une certaine philosophie de programmation. Nous retrouvons en particulier ici une idée très importante en programmation : celle de programmation par l'exemple (au sens particulier du Chapitre I.2.4.2.4).

Notion de *décomposition héritable* pour le contrôle procédural.

VIII.1.2.3. Quatre capacités d'abstraction

A la lumière des exemples présentés ici, nous pouvons dégager quelques principes originaux de NéOpus, qui donnent à ce système des capacités d'abstraction à quatre titres.

1/ Abstraction liée à l'ordre un : c'est la notion classique d'ordre un,

2/ Abstraction liée au typage naturel

Le fait de pouvoir typer une variable d'ordre un par tout un arbre de classes et pas seulement par une seule classe est assez unique à notre connaissance dans les systèmes à base de règles en marche avant. Une règle peut donc être interprétée différemment suivant les objets qui la filtrent (Cf. l'exemple de la dérécursivation de fonctions au chapitre V.1.2.6). Cette caractéristique de généricité a de grandes conséquences méthodologiques. En effet, on peut alors concevoir une règle comme une *entité réutilisable* et utiliser l'héritage de classe pour faire varier l'interprétation d'une règle.

3/ Abstraction liée à l'héritage de bases de règles

Cet aspect de la généricité est lié à la généricité que l'on retrouve par l'héritage de classe standard, transposé aux règles.

4/ Abstraction du contrôle

L'architecture de contrôle déclarative de NéOpus introduit la notion originale de réutilisabilité des spécifications de contrôle.

VIII.1.2.4. Conséquences négatives de la disparition des faits

Mais la disparition de la notion d'attribut-valeur a aussi des conséquences négatives.

VIII.1.2.4.1. Pas d'optimisations sur Rete

Les optimisations sur Rete ne sont pas applicables à NéOpus, pour les raisons évoquées au Chapitre III.3.5.7. En particulier, il est impossible de factoriser des prémisses communes à plusieurs règles.

VIII.1.2.4.2. Jonction difficile avec un système de maintien de la vérité

Les systèmes de maintien de la vérité (TMS [Doyle 79] ou ATMS [deKleer 86a]) permettent de maintenir un réseau des inférences produites par un système de raisonnement. Les "faits" du systèmes sont ainsi reliés entre eux par un réseau de justification permettant de répondre aux questions du type "quels faits ont permis de déduire tel ou tel autre", et donc d'en déduire les conséquences du retrait d'un fait ayant servi à en déduire d'autres.
Le couplage d'un tel système avec des systèmes de type Rete est courant (Art, Knowledge Craft, X-TRA [Charpillet&al 89a, 89b]), bien qu'il pose des problèmes compliqués (par exemple de coopération avec le système de raisonnement [deKleer 89c], ou de gestion de la négation [deKleer 89b], [deKleer 88], [Dressler 88]). Mais un

tel couplage ne peut se faire que dans une optique "attribut-valeur", puisque les nœuds des graphes (TMS ou ATMS) représentent des entités *logiques* (i.e. à caractère booléen), identifiées le plus souvent à des triplets (objet-attribut-valeur).
Dans notre cadre, la partie action des règles n'est pas représentable sous forme d'un *objet à caractère logique*. Le système n'a aucun moyen de connaître les effets de la partie action d'une règle, et ne peut donc pas maintenir un tel réseau de justifications pour ces actions.
Plusieurs travaux ont tenté de résoudre ce problème de la représentation des dépendances "inférentielles" entre objets. En particulier [Euzenat&Rechenmann 87], [Euzenat 90] ramènent ce problème à un problème de "cache" de méthode. Ainsi, le résultat d'une méthode n'est pas recalculé à chaque envoi du message correspondant, mais seulement lorsque les objets susceptibles de changer ce résultat ont été modifiés. Mais cette approche est très dépendante de leur mode de représentation (le système Shirka [Rechenmann 85]) et ne peut être reprise dans notre contexte.

La seule solution envisageable dans notre cadre serait d'utiliser la notion d'assertion (Cf. Chapitre VI.6.2). En effet, les assertions ayant un caractère intrinsèquement booléen, on peut tout à fait imaginer de les relier entre elles en fonction des inférences ayant servi à les produire. Nous n'avons pas développé cet aspect.

VIII.2. Synthèse méthodologique

Si les apports et enjeux techniques de NéOpus sont clairs, les conséquences méthodologiques le sont moins. Le système NéOpus soulève des questions fondamentales sur la représentation du discours sur le monde. Nous nous sommes contentés dans notre approche de suivre la voie traditionnelle, inaugurée par les premiers systèmes experts, et qui consiste à *ne pas distinguer* la représentation du monde et la représentation d'un *savoir sur* ce monde.

VIII.2.1. Une confusion originelle

Les premiers systèmes de représentation à base de "faits" ont introduit une confusion qui subsiste dans notre système mais qui est rendue plus apparente par la nature même des objets. Il s'agit de la confusion entre la représentation de *ce qui est* d'une part, et la représentation de ce que le système *sait* sur ce qui est.
Dans un système classique (sous-entendu "à base de faits") cette distinction n'a aucun sens : les faits sont des entités mortes par nature et servent à représenter tout ce qui est utile à la bonne marche du système.
Mais dans notre contexte les choses se précisent : les objets du discours sont représentés par des entités informatiques (les objets) qui ont une certaine prétention à représenter le réel. Bien sûr, comme dans tout modélisation, ces objets reflètent une

certaine perception de ce réel par le modélisateur qui leur a donné naissance. Cette perception est donc, par définition, fortement subjective, et la représentation qui en découle contient déjà beaucoup de connaissances sur le monde. D'autre part cette représentation est réductrice, et très contrainte. Néanmoins nous partons du principe qu'en programmation par objets les objets représentent une certaine perception du réel.
Lorsqu'un système de raisonnement se greffe sur cette représentation, son seul moyen d'expression est alors d'agir sur la représentation de cette réalité. Il n'y a alors pas de différence entre *raisonner* et *agir*. Ainsi, tout est fait au moment de la modélisation.
On a alors le schéma suivant (Cf. Figure 29), dans lequel le réel est modélisé sous forme d'objets, et où le système de raisonnement n'est autre qu'une mécanique de réagencement arbitraire et anarchique de ces objets.

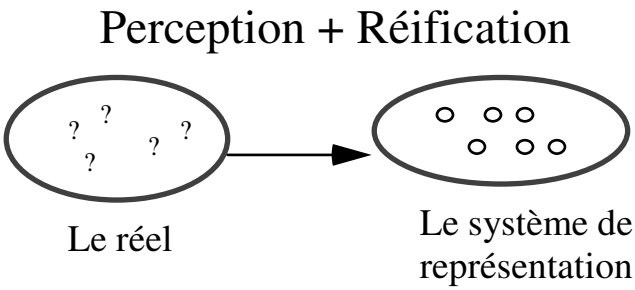


Figure 29. Système de représentation de la réalité

Raisonner dans ce cadre consiste alors simplement à modifier les objets représentant le réel. Plutôt que de parler de systèmes de représentation de connaissances, nous devrions plutôt parler de *systèmes de représentation de la réalité*. Les systèmes dit inférentiels ne sont en effet alors que des mécaniques plus ou moins adaptées pour remplir les attributs des objets. Comme le résume H. Wertz [Wertz 85] :

> Si on considère les atomes cognitifs comme des frames, le processus de compréhension des fonctionnalités d'un système se traduit par l'activité de remplissage des slots de ces frames.

VIII.2.2. Exemple typique de la transitivité de l'inégalité

Illustrons nos propos par un exemple : la transitivité de l'inégalité. Nous avons déjà rencontré cet exemple (Chapitre V.1.4.2) pour un problème purement mécanique. Nous le reconsidérons à la lumière de nos propos.
On considère la transitivité de l'inégalité (on la préférera à celle de l'égalité, qui relèverait plutôt de systèmes de satisfactions de contraintes). Elle s'énonce comme:

Soient a, b, c des nombres,		
si		a < b
et	b < c	
alors		a < c

Il est clair que dans une telle règle, on *parle* d'objets, référencés par des variables `a b` et `c` et qui peuvent être de type entier par exemple (ou tout type acceptant une relation d'ordre totale). Le problème qu'elle va soulever est l'énorme différence de nature qui existe entre les deux types de discours sur ces objets et les interprétations qu'ils drainent :

	a < b	en prémisse, avec un sous-entendu *est-il vrai que ... ?*,
et	a < c	en conclusion, avec un sous-entendu *je dis que.*

Cette différence est d'autant plus importante qu'elle est *cachée* par le jeu des symboles mathématiques : on exprime deux sous-entendus différents par le même symbole (ici <). Malgré son apparente simplicité, aucun système à base de règles, que ce soit en marche avant ou en marche arrière, n'est capable de représenter de manière satisfaisante de genre d'énoncés.
Si nous supposons en effet que nous représentons le monde de manière simulatoire, nous allons définir la classe des entiers, munis des opérateurs arithmétiques les plus courants. En l'occurrence, cette classe existe déjà en Smalltalk (la classe `Integer`), et la méthode < permet de représenter la relation d'ordre entre entiers.
En ce cas, les prémisses `a < b`, et `b < c` (`a`, `b` et `c` étant instanciés) s'interprètent comme des expressions booléennes (ayant un résultat vrai ou faux). En revanche la partie droite `a < c` n'a aucun sens : son évaluation rendra aussi un booléen, mais ce *quelque soit les prémisses de la règle*. De plus, pourquoi vouloir *affirmer* que `a < c`, alors qu'il suffit de le vérifier![62].

Cette question prend maintenant un éclairage différent si l'on accepte que la règle n'énonce pas un discours sur la réalité, mais sur la *connaissance de* cette réalité par le système. Ainsi, notre règle s'interpréterait plutôt comme :

Soient trois entiers `a`, `b`, `c`
Si je sais que "`a < b`"
et si je sais que" `b < c`",
alors (maintenant) je sais (ou plutôt j'apprends) que "`a < c`".

En ce cas, les objets `Inferieur` (introduits par P. Laublet dans son système ForreEnMat [Laublet 90]) représentent un savoir du système à propos des nombres. Ce sont donc des objets qui sortent du monde des nombres, et qui ne se justifient que lorsqu'on accepte que le discours qui les utilise est un discours sur la connaissance, et non directement un discours sur le monde.

[62] On peut dire ici que "`a < b`" est plus facile à dire qu'à faire.

VIII.2.3. Représenter un énoncé

Représenter des connaissances sur des objets consiste souvent à représenter des énoncés, au sens logique, c'est à dire des propositions à caractère booléen portant sur des objets. Par exemple :

'p1 est un ancêtre de p2',
'a est plus grand que b',
'les segments AB et AC ont même longueur',
'le patient est hypoventilé',
'le troisième accord s'analyse comme IIème degré de Sol Majeur',
'il va y avoir un arrêt brutal des communications à 15:30'

NéOpus n'étant pas un système logique, nous ne pouvons pas représenter ces propositions par des clauses du premier ordre. Le problème est alors de trouver une représentation de ces énoncés qui permette l'inférence, c'est à dire qui soit utilisable dans les règles.
Plus précisément, de par la nature des règles de production, il y a deux représentations distinctes à trouver :

une représentation en tant que prémisse, notée représentation P,
une représentation en tant qu'action, notée représentation A.

VIII.2.4. Exemple

Le calcul d'ancêtres tel qu'il est décrit dans le chapitre IV est un bon exemple. On doit y représenter l'assertion sur deux objets p1 et p2, instances de la classe Personne : '*p1 est un ancêtre de p2*'. En admettant la définition de la classe `Personne`, telle que les personnes connaissent leurs ancêtres (une liste de personnes), on aura les deux représentations pour notre énoncé :

Prémisse:	`p ancetres includes: p2`
Action:	`p ancetres add: p2.`

VIII.2.5. Le P et le A des énoncés

En résumé, un énoncé est représentable si les deux conditions suivantes sont réunies :

1- Il existe une expression Smalltalk évaluable **P** dont le résultat de l'évaluation est la valeur de cet énoncé à tout moment.

2- Il existe une expression Smalltalk évaluable **A**, dont l'effet est tel que l'évaluation de **P** rende un résultat vrai.

Le problème de la représentation de connaissance en NéOpus (et plus généralement dans tout système de règles de production, *mais lorsque les objets du discours sont de faux objets, ce problème est cachée par la nature même du système*) peut donc se résumer à trouver un cadre conceptuel satisfaisant le principe ci-dessus.
Mais tandis que les systèmes traditionnels proposent une conception des objets uniquement orientée par l'écriture de règles, nous avons ici affaire à des objets qui obéissent à des lois de conception différentes. Les objets ne sont pas, à priori, conçus pour représenter les assertions portant sur eux. En particulier, ils ne sont pas toujours conçus pour *rendre compte* ni pour *prendre en compte* les connaissances que l'auteur de règles veut leur faire porter.
Une autre formulation du problème est donc celle-ci :

Comment faire pour que les objets soient capables de rendre compte et de prendre en compte les informations nécessaires à la représentation des énoncés logiques portant sur eux?

VIII.3. Vers un schéma triadique : La notion d'aspect

Notre attitude peut se généraliser en introduisant la notion d'aspect, comme mécanisme généralisé d'enrichissement structurel au service du problème fondamental de représentation énoncé ci-dessus.

VIII.3.1. Représenter la connaissance par des objets

Le problème qui se pose alors est tout simplement de savoir comment représenter la connaissance qu'a le système *sur* des objets. Nous avons jusqu'à présent caché ce problème en considérant que cette connaissance se *manifestait* par des requêtes sur des objets (ou des groupes d'objets), et *s'affirmait* par des modifications d'objets. Mais cette approche conduit à une perversion des modélisations, qui ont alors la lourde charge de représenter à la fois le réel et la connaissance qu'en a le système.
Notre idée est que les objets doivent alors jouer un rôle supplémentaire qu'ils ne peuvent pas tenir. Nous allons les enrichir par un mécanisme général d'aspects, consistant à représenter *ailleurs* la connaissance que le système a sur eux.
Nous montrerons ensuite comment réinterpréter notre expérience NéOpus sous cette nouvelle lumière.

VIII.3.2. Définition : un quatrième aspect des objets

[Ferber 89] dégage trois *aspects* ("aspect" étant à prendre cette fois dans le sens courant) des objets en représentation de connaissances : structurel, conceptuel et actanciel.
Dans notre cadre, il manque un aspect, qui est celui de la représentation du *savoir sur*. En effet si les règles sont censées produire des inférences, où et comment sont représentées ces inférences ? Nous voulons donc compléter ce cahier des charges des objets en y ajoutant un chapitre fondamental : la représentation du savoir sur les (autres) objets.

Définition de l'aspect
On parlera d'*aspects* d'un objet, appelé *modèle*, pour désigner des objets servant à représenter un certain savoir du système sur le modèle

Ainsi, le schéma méthodologique est-il raffiné : deux types très distincts d'objets doivent nécessairement coexister dans un système de représentation inférentielle.
La notion de réification est introduite par [Ferber 89] pour nommer l'acte de transfert entre un objet réel et une entité informatique. Un postulat fondamental et radical de la représentation par objet est même proposé, énonçant que tout peut se représenter sous forme d'objet. Mais tous les objets ne sont pas pour autant égaux.
Nous voulons ici souligner l'échec d'une approche considérant qu'une fois le monde représenté, un système de raisonnement est simplement un système qui manipule ce monde. Nous introduisons donc deux types d'objets : ceux servant à représenter le monde, et ceux servant à représenter la connaissance qu'en a le système.
On peut alors parler plus justement d'un système de raisonnement, dans la mesure où le raisonnement n'est plus simplement limité à des actions de remplissage d'attributs, mais s'apparente plus à une activité de réflexion. Nous retrouvons ici la dualité du mot réflexion, qui signifie à la fois mouvement inversé (celui de la lumière sur un miroir par exemple), et activité mentale :

Réfléchir (Petit Larousse) :
1. Renvoyer dans une autre direction : les miroirs réfléchissent les rayons lumineux.
2. Penser, méditer longuement.

Cette dualité incite à penser que pour qu'il y ait réflexion il faut qu'il y ait miroir, et donc dualité de représentation. Nous proposons que les aspects jouent ce rôle de miroir. On obtient alors un deuxième schéma (Cf. Figure 30), dans lequel le raisonnement apparait non pas comme la modification directe des objets représentant le monde, mais comme la manipulation d'aspects représentant un savoir sur ces derniers.

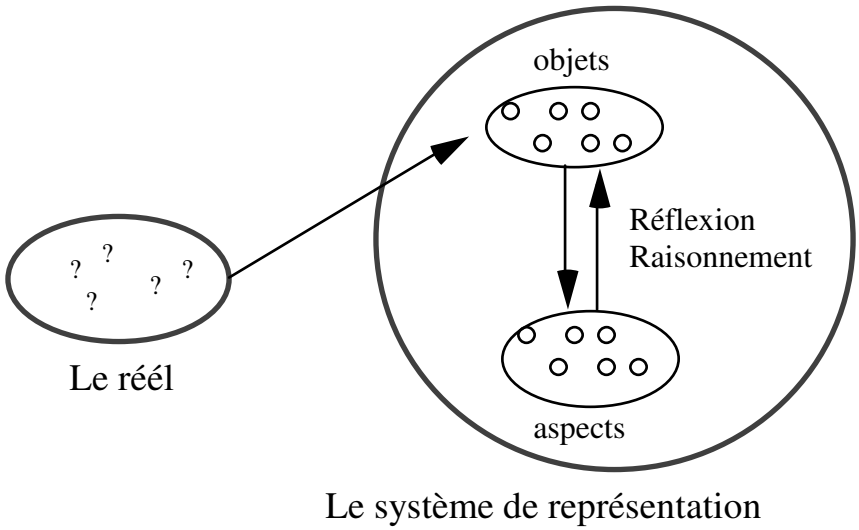


Figure 30. Système de représentation et de raisonnement

VIII.3.3. Le mécanisme de base : le P et le A des énoncés

La classe `Aspect` n'existe pas à proprement parler. C'est une notion générale et abstraite, qui se reflète dans certaines utilisations de classe Smalltalk. Cependant nous définissons dans cette notion le minimum indispensable pour fonctionner à savoir un pointeur vers le modèle :

```
Object subclass: #Aspect
  instanceVariableNames: 'modele'
```

L'aspect, par défaut ne définit pas de lien inverse du modèle vers ses aspects. Ce lien peut être rajouté, par exemple dans le mécanisme de délégation explicite, mais ne fait pas partie de la notion d'aspect la plus générale.

De manière générale, la manipulation d'aspects se fera via des méthodes d'accès plus ou moins évoluées, du type de celles que nous avons rencontrées dans les exemples utilisant la délégation (la géométrie, ou l'analyse harmonique).
Nous pouvons les classer en deux types :

Méthodes d'accès en lecture, représentant les `P` des énoncés.
Ces méthodes testent la *présence* d'un aspect vérifiant certaines propriétés (par exemple de type). Elles servent à représenter les `P` des énoncés. Par exemple, la méthode `estUn: unType` de la classe `Figure`, ou la méthode `estAnalyseEn:` de la classe `Accord`.

Méthodes d'accès en écriture, représentant les `A` des énoncés.
Ces méthodes testent la *présence* d'un aspect vérifiant certaines propriétés (par exemple de type). Elles servent à représenter les `A` des énoncés. Ainsi, apprendre quelque chose sur un objet revient à augmenter (ou modifier) un aspect de cet objet.

Par exemple, la méthode `addType: unType` de la classe `Figure`, ou la méthode `analyseEn:` de la classe `Accord`.

VIII.3.4. Une triade objets/aspects/règles

Les aspects représentent les connaissances qu'à le système sur leur modèle. Ces connaissances sont alors *exprimées* par des règles, qui vont manipuler ces aspects: tester la présence d'aspects de tel ou tel type, en créer de nouveaux ou en supprimer. Dans ce cadre, raisonner consiste alors simplement à manipuler des aspects d'objets, dans le but d'accroître notre connaissance sur eux. On obtient alors un schéma triadique du raisonnement par objet :

> Les **objets** représentent le monde tel qu'il est perçut par le modélisateur.
> Les **aspects** représente la connaissance du système sur ce monde. Ils sont liés aux objets (leurs modèles) par un mécanisme de délégation.
> Les **règles** permettent de manipuler, créer, détruire ou modifier les aspects sur le monde.

VIII.3.5. Réinterprétation de notre expérience

VIII.3.5.1. La transitivité de l'inégalité (2)

Dans notre nouveau cadre, nous pouvons considérer que :

Le réel (i.e. les nombres) est représenté des objets Smalltalk (instances de la classe `Nombre` par exemple),
Notre *connaissance sur les nombres* se représente par d'autre objets, qui seront des aspects de ces nombres. Par exemple, si on s'intéresse à la transitivité de l'inégalité, on introduira les objets `Inferieur` comme des aspects d'entiers.

Ainsi, on peut définir des méthodes d'accès aux aspects dans la classe `Nombre` comme pour représenter les P et A des nos énoncés :

```
'Nombre methodsFor: 'acces aux aspects'!

estConnuCommeEtantInferieurA: unNombre
qui rend vrai s'il existe un aspect de self de la classe
Inferieur, avec comme argument unNombre.

estReconnuCommeEtantInferieurA: unNombre
qui augmente la liste des aspects de self pour prendre en compte
un nouvel objet Inferieur, avec unNombre comme argument.
```

Dans ce cas on arrive à une formulation de la règle `abc` du type :

```
transitiviteDeLInegalite
   | Nombre a b c |
   a estConnuCommeEtantInferieurA: b.
   a estConnuCommeEtantInferieurA: b.
   actions
   a estReconnuCommeEtantInferieurA: c.
   a modified
```

La règle que nous avons écrite au chapitre V.1.4.2 peut maintenant être envisagée sous un autre angle. Elle est une autre transcription de notre discours, mais sa différence principale avec la règle `transitiviteDeLInegalite` que nous venons décrire est qu'elle filtre *directement* les objets `Inferieur`. Les nombres n'existent alors plus vraiment. C'est pourquoi notre représentation nous semble plus raisonnable.

VIII.3.5.2. Le système MusES

Le système MusES est un bon exemple de la différence entre représentation du monde et représentation de la connaissance sur ce monde. La notion d'accord est représentée par la classe `Accord` (ainsi que toutes les autres classes servant à la genèse des accords : les notes, les intervalles ...). Mais l'*analyse* d'un accord dans une situation donnée est un savoir parfaitement extérieur à l'accord lui-même. C'est d'autant plus clair qu'un accord ne s'analyse qu'en fonction du contexte. Ainsi, plusieurs occurrences d'un même accord pourront s'analyser de différemment, comme nous l'avons vu.
Les objets représentant l'analyse d'un accord (Cf. la classe `Analyse`) peuvent se considérer comme des *aspects* d'accords. Nous les avons reliés aux accords par un mécanisme de lien explicite (en ajoutant dans la classe `Accord` une variable d'instance pointant sur la liste des analyses possibles). Si cette représentation satisfait nos besoins, elle prend un éclairage particulier une fois admis que les analyses sont des aspects.

VIII.3.5.3. La géométrie

L'exemple de la géométrie suit le même principe : les figures géométriques représentent la réalité des objets géométriques. Les types déduits par le système représentent la connaissance qu'a le système à propos de ces figures.

VIII.3.5.4. Le contrôle

Le problème du contrôle peut aussi se prêter au jeu des aspects. Les évaluateurs peuvent être vus comme des aspects particuliers d'une base de règles, servant à

enrichir la connaissance que le système a de leur activation. Ainsi, la proposition contrôler = raisonner s'instancie ici en contrôler = manipuler des aspects = manipuler des évaluateurs.

VIII.4. Le point de vue en représentation de connaissances

La notion de point de vue est récurrente en Intelligence Artificielle, mais il en existe autant de "points de vue" que de systèmes. De manière générale la notion de point de vue sert à représenter la complexité de la multitude des représentations possible d'un même objet. Le système PIE [Goldstein&Bobrow 80] introduit la notion de point du vue pour représenter plusieurs angles de vision d'un même objet. Cette notion est implémentée à l'aide d'un mécanisme de délégation explicite [Lieberman 86a, 86b]. Cette idée de point du vue comme mécanisme d'agrégation structurelle est reprise dans le système Systalk [Wolinski 90], [Perrot&Wolinski 91, 92], pour représenter différentes *facettes* de robots. B. Carré dans le langage ROME [Carré 89] propose de représenter la notion de point de vue multiple d'un objet à l'aide d'une interprétation particulière de l'héritage multiple : un point de vue se spécifie en distinguant une super-classe particulière du graphe d'héritage.
La notion de point du vue du système Art [ART 97] introduit la notion de *monde possible* : un objet est présent dans un certain monde, mais pas dans d'autres, en fonction des diverses hypothèses élaborées en cours de raisonnement.
Dans un ordre d'idée tout à fait différent, P. Krief dans le système MPVC [Krief 90] introduit la notion de point de vue comme une certaine interprétation d'un parcours d'un modèle de représentation.
Enfin, P. Volle dans le système Madeleine [Volle 88, 89] propose de baser un mécanisme de raisonnement sur les objets en introduisant la notion de *co-référence* [Ferber&Volle 88a, 88b]. Dans cette optique, raisonner consiste à associer des objets coréférents. Cette idée est proche de la notre, mais ne considère plus l'objet comme entité de base du discours : la coréférence entre deux objets est représentée par un partage de pointeurs, mais l'objet central n'est pas représenté.

Parmi ces tentatives diverses de donner un sens à la notion de point du vue, notre notion d'aspect se distingue des approches existantes non pas par un mécanisme nouveau mais par la problématique qui la fait naître : nous parlerons d'aspect d'un objet, qu'il soit représenté de telle ou telle manière, lorsque l'on parlera d'une *connaissance sur* l'objet qui se distingue de l'objet lui-même.

IX. Conclusion

Avant propos

Cette conclusion est volontairement courte : la véritable conclusion de notre travail est décrite dans le chapitre précédent.

Conclusion

Notre étude est double. D'une part nous avons étudié les conséquences du plongement d'un mécanisme d'inférence classique dans le monde des objets au sens strict. Cette intégration donne lieu à un certain nombre de mécanismes inférentiels dont nous avons décrit l'implémentation et que nous avons appliqués à des exemples variés. D'autre part nous avons tiré de notre expérience du système un certain nombre d'enseignements aboutissant à la notion d'aspect, esquissée en fin de parcours.

Notre travail soulève plus de questions qu'il n'en résout. La direction de recherche principale qu'il nous semble intéressant d'explorer est celle consistant à étudier les rapports entre la représentation des connaissances et la représentation de la réalité sur lesquelles les connaissances s'appliquent. La notion d'assertion que nous avons introduite au Chapitre VI.6.2 est un début dans cette voie.

X. Références

[Alizon&Huet 88]
Alizon F., Huet G. Essaim. Un environnement de programmation Smalltalk destiné à la construction de systèmes experts. Note technique CNET NT/LAA/SLC/299, 1988.

[Alvarez 91]
Alvarez I. Explication comparative dans les systèmes experts. Conférence internationale sur les systèmes experts et leurs applications : systèmes experts de seconde génération, Avignon, pp. 173-184, (1991).

[Art 87]
ART Reference Manual, v 3.0, January 1987, Inference Corporation.

[Atkinson&Laursen 87]
Atkinson R., Laursen J. Opus : A Smalltalk Production System. OOPSLA'87 pp. 377-387.

[Bachimont 90]
Bachimont B. Cohérence et Convergence dans un Tableau noir : Organisation, Formalisation et Sémantique de l'architecture de contrôle ABACAB. Thèse d'université, Paris VI. Décembre 1990.

[Batali 88]
Batali J. Reasoning about self-control. In Meta-level Architectures and Reflection, P. Maes et D. Nardi eds. North Holland, 1988.

[Baudoin 90]
Baudoin P. Le Jazz mode d'emploi. Volume 1. Editions Outre Mesure, Collection Théories, Paris, 1990.

[Benhouhou 92a]
Benhouhou A. Générateur de scénarios pour la formation professionnelle basé sur un simulateur de processus. Congrés sur les Systèmes Experts, Avignon 1992.

[Benhouhou 92b]
Benhouhou A. Un générateur de scénarios pour la gestion de crise naturelles : une approche objet. Conférence Représentation par objets, La grande Motte, Juin 1992.

[Benoit & al. 86]

Benoit C. & al. Knowledge Representation and communication mechanisms in LORE. ECAI, Vol. 1, pp. 246-255, Brighton, 1986.

[Bernier 91]
Bernier J.-F. Techniques d'Intelligence Artificielle appliquées au génie Logiciel. Rapport de stage du DEA LAP, LITP, Université Paris VI, Septembre 1991.

[Bézivin 84]
Bézivin J. Simulation et langages orientés objet. Bigre+Globule n° 41, pp. 194-211, Novembre 1984.

[Bézivin 86]
Bézivin J. Langages objets et prototypage. Bigre+Globule n° 51, pp. 23-32, Octobre 1986.

[Bézivin 88]
Bézivin J. Langages objets et programmation concurrente : quelques expérimentations avec Smalltalk-80. Bigre+Globule n° 59. Avril 1988.

[Blake&Cook 87]
Blake E, Cook S. On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk. ECOOP '87, pp. 45-54.

[Bobrow&Stefik 83]
Bobrow D. Stefik M. The LOOPS Manual, Xerox Corporation (1983).

[Bodin&Pichat 90]
R. Bodin, E. Pichat. Ingénierie des données. Masson, 1990.

[Borne&Pachet 91]
Borne I., Pachet F. From Object-Oriented Design to Visual Programming. Proceedings of Computer-Aided Engineering and Education, Prague, 4 Septembre 1991.

[Borning 79]
Borning Alan. ThingLab - A Constraint-Oriented Simulation Laboratory. Xerox Palo Alto Research Center, rapport SSL-79-3. July 1979.

[Bouaud 89a]
Bouaud J. K, une méthode de pattern-matching en programmation réflexe. RFIA' 89. pp. 1603-1612.

[Bouaud 89b]
Bouaud J. K, un Langage pour l'Implémentation d'Outils de Représentation des Connaissances. Thèse de l'Université Paris VII, Juin 1989.

[Boulez 63]
Boulez P. Penser la Musique aujourd'hui. Denoël/Gonthier, Paris, 1963.

[Bourgeois 90]
Bourgeois R. ICEO. Intension, coréférences et objets dans la fédération de formalismes de spécification. Thèse de doctorat. Université Paris VI. Paris, Décembre 1990.

[Bouzhegoub 86]
Bouzhegoub M. SECSI : Système Expert en Conception de Systèmes d'Informations. Thèse de l'université Paris VI, Mars 1986.

[Brachman 77]
Brachman R. What's in a Concept : Structural Foundations for Semantic Networks. International Journal of Man-Machine Studies. Vol. 9 (2), pp. 127-152 (1977)

[Brachman&Shmolze 85]
Brachman R., Shmolze J. An overview of the KL-One Knowledge Representation System. Cognitive Science n° 9, 1985.

[Braun 88]
Braun G. Sur la programmation de constructions géométriques. Thèse d'université, Strasbourg, Juin 1988.

[Briot 85]
Briot J.-P. Instanciation et héritage dans les langages orientés objet. Thèse de 3ième cycle, Paris, 1985.

[Briot 89]
Briot J.-P. Actalk : A testBed for Classifying and Designing Actor Languages in the Smalltalk-80 environment. ECOOP '89, pp. 109-130.

[Brownston& al. 85]
Brownston L., Farrell R., Kant E., Martin N. Programming Expert Systems in OPS5. An Introduction to Rule-Based Programming. Addison-Wesley Publishing Company, 1985.

[Buchanan&Shortliffe 84]
Buchanan B.G., Shortliffe E.H. Rule-Based Expert System. The Mycin experiments of the Standford Heuristic Programming Project. Addison-Wesley, 1984.

[Carré 89]

Carré B. Méthodologie orientée objet pour la représentation de connaissances, concepts de point de vue, de représentation multiple et évolutive d'objets. Thèse de l'Université de Lille, 1989.

[Carnegie 85]
Knowledge Craft. Reference Manual. Carnegie Group Inc., Pittsburgh, 1985.

[Caseau 87]
Caseau Y. Etude et réalisation d'un langage objet : LORE. Thèse d'état, Université de Paris-sud, 1987.

[Chandrasekaran 83]
Chandrasekaran B. Towards a Taxonomy of Problem-Solving Types. The AI Magazine, Winter/Spring 1983, pp. 9-17.

[Chandrasekaran 87]
Chandrasekaran B. Towards a functional architecture for intelligence based on generic information processing tasks. Proc. of the Tenth IJCAI, Milan (Italy), Vol. 2, 1987, pp. 1183-1192.

[Charbonnel 90]
Charbonnel S. Etude et réalisations de l'environnement du générateur de systèmes experts NéOpus. Rapport de stage de DESS au CEMAGREF, Nantes 1990.

[Charpillet & al. 89a]
Charpillet F., Théret P., Boivin T., Haton J.-P. Présentation générale et spécification du noyau ATMS de X-tra. RFIA '89.

[Charpillet & al. 89b]
Charpillet F., Théret P., Boivin T., Haton J.-P. X-TRA, un moteur d'inférence comportant deux modes de compilation de règles TREAT et RETE et un système de maintien de la vérité de type ATMS. Rapport du CRIN 89-R226, Juin 89.

[Clancey 83a]
Clancey W. The epistemology of a Rule-Based Expert System - A Framework for explanation. A.I. n° 20 (1983) pp. 215-251.

[Clancey 83b]
Clancey W. The advantages of Abstract Control Knowledge in Expert-System Design. Report N° STAN-CS-83-995. Stanford University, 1983.

[Codd 70]
Codd A.-F. A relational model of data for large shared data banks. Comm. ACOM, Vol 13, n° 6, June 1970.

[Cointe 84]
Cointe P. Implémentation et interprétation des langages orientés objet : application aux langages Smalltalk, Objvlisp et Formes. Thèse d'Etat, Paris VI, 1984.

[Cointe 87]
Cointe P. Metaclasses are First Class : the ObjVlisp model. Proceedings of OOPSLA '87. pp. 156-167.

[Cointe&Briot 89]
Cointe P., Briot J.-P. Programming with ObjVlisp metaclasses in Smalltalk-80, OOPSLA '89, New Orleans, USA.

[Computer Thought Corporation]
OPS5+ Reference Manual, v 2.0, 1986.

[Copeland&Maier 84]
Copeland G. Maier D. Making Smalltalk a Database System. Proceedings of ACM/SIGMOD. International Conference on the Management of Data. 1984, pp 316-325.

[Corby 87]
Corby O. BIB en SMECI, RFIA'87, Paris, pp. 581-586.

[Cordier &al. 86]
Cordier M.-O., Faller B, Rousset M.-C. L'optimisation de l'opération de "pattern-matching" dans les systèmes experts. TSI, Vol. 5, n° 3, 1986.

[Coupey 89]
Coupey P. Etude d'un réseau sémantique avec gestion des exceptions. Thése de doctorat. Université Paris-Nord, Janvier 89.

[Davis 80]
Davis R. Meta-rules : Reasoning about control. Artificial Intelligence Journal, N°15, Dec 1980, pp. 179-222.

[deKleer 86a]
deKleer J. An Assumption-based truth maintenance system. Artificial Intelligence, n° 28 (1986).

[deKleer 86b]
deKleer J. Extending the ATMS. Artificial Intelligence, n° 28 (1986).

[deKleer 86c]

deKleer J. Problem solving with the ATMS. Artificial Intelligence, n° 28 (1986).

[deKleer 88]
deKleer J. A General labelling algorithm for assumption-basd truth maintenance systems, proceedings of the 7th National Conference on Artificial Intelligence, St Paul, Minessota, USA 1988.

[Deutsch 89]
Deutsch P.L. The past, present, and future of Smalltalk. ECOOP '89, pp. 109-130, Cook, London, 1989.

[Dieng 89]
Dieng R. Coopération pour l'analyse d'un raisonnement. RFIA '89, pp.319-333, Paris, 1989.

[Dixneuf & al. 87]
Dixneuf P., Meller A., Porcheron M. Eloise - le chaînage avant - Guide de l'Utilisateur. Rapport Eloise 1.0, CGE, Septembre 1987.

[Dojat &al. 91a]
Dojat M., Brochard L., Lemaire F., Harf A. A Knowledge-Based System for the management of the weaning procedure of mechanically ventilated patients. International Journal of Clinical Monitoring & Computing, à paraître.

[Dojat &al. 91b]
Dojat M., Brochard L., Harf A. Un système à base de connaissances pour la ventilation des patients en soins intensifs. 8ème congrès RFIA, Lyon, pp. 1079-1083, Novembre, 1991.

[Dojat &al. 92]
Dojat M., Brochard L., Harf A. Evaluation d'un système d'aide à la décision pour le sevrage des patients ventilés artificiellement. XXème Congrès de la Société de Réanimation de langue Française, Paris, January, 1992.

[Dormoy 86]
Dormoy J.-L. Quelles connaissances pour utiliser efficacement les connaissances ? Colloque d'I.A. Strasbourg 15-19 Septembre 1986, pp. 193-215.

[Dormoy 87]
Dormoy J.-L. Analyse heuristique de règles par règles. Colloque d'I.A. Caen 15-19 Septembre 1987, pp. 203-216.

[Doyle 79]

Doyle J. A Truth maintenance system Artificial Intelligence, n° 12 (1979).

[Dressler 88]
Dressler O. Extending the basic ATMS. ECAI'88.

[Dubois 21]
Dubois T. Traité d'Harmonie. Heugel, Editions A. Leduc, Paris 1921.

[Ducourneau&Habib 89]
Ducourneau R., Habib M. La multiplicité de l'héritage dans les langages à objets. TSI, Vol. 8, n° 1, Janvier 1989.

[Dufourd 90]
Dufourd J.-F. Programmation et résolution de problèmes de construction géométrique. Rapport R-9003. ULP, Université de Strasbourg, Janvier 1990.

[Engelmore & al. 88]
R. Engelmore, T. Morgan. Blackboard Systems. Addison-Wesley Publishing Company, 1988.

[Euzenat&Rechenmann 87]
Euzenat J. Rechenmann F. Maintenance de la vérité dans les systèmes à base de connaissance centrée-objet. Actes du 6ième congrès AFCET-INRIA RFIA, Antibes 1987.

[Euzenat 90]
Euzenat J. Un système de maintenance de la vérité à propagation de contextes. Thèse de l'université J. Fourier, Grenoble, 1990.

[FakeBook 86]
The Fake book, Syosset, New York, 1986.

[Farreny&Ghallab 87]
Farreny H., Ghallab M. Eléments d' Intelligence Artificielle. Hermes, Paris, 1987.

[Ferber&Volle 88a]
Ferber J., Volle P. Corerefentiality : the key to an intensional theory of object oriented knowledge representation. Rapport LAFORIA n° 88/24, 1988.

[Ferber&Volle 88b]
Ferber J., Volle P. Using coreference in object-oriented representations, ECAI '88.

[Ferber 89]

Ferber J. Objets et agents: une étude des structures de communication et de représentation en Intelligence Artificielle. Thèse d'Etat, Université Paris VI, Juin 1989.

[Forgy 81]
Forgy C. L. OPS5 User manual. Department of Computer Science, Carnegie Mellon University, 1981.

[Forgy 82]
Forgy C. L. Rete : A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence Vol. 19 (1982) pp. 17-37.

[Forte 91]
Forte A.-M. Système basé sur la connaissance pour l'identification, la caractérisation et la mise en correspondance d'entités anatomiques et fonctionnelles en imagerie médicale multi-modalité. Thèse de Doctorat de l'Université François Rabelais, Tours, Novembre 1991.

[Forte&Bizais 91]
Forte A.-M., Bizais Y. Analysing and Interpreting Pulmonary Tomoscintigraphy Sequences : Realization and Perspectives. SPIE 91, Medical Imaging V, San Jose, CA, USA, 23 Février-1er Mars, (1991)

[Forte&al. 92]
Forte A.-M., Bernardet M., Lavaire F., Bizais Y. Object-Oriented versus Logical Conventional Implementation of a MMIIS. SPIE'92, Medical Imaging VI, Newport Beach, Ca, USA, 23-28 Février (1992).

[Fouet 86]
Fouet J.M. Compilation de connaissances dans la machine Gosseyn. Colloque d'I.A. Strasbourg 15-19 Septembre 1986, pp. 255-269.

[Fouet 87]
Fouet J.M. Utilisation de connaissances pour améliorer l'utilisation de connaissances : La machine Gosseyn. Thèse d'Etat. Paris VI, 1987.

[Ganascia 90]
Ganascia J.G. L'âme machine. Editions du Seuil, Collection Science Ouverte. 1990.

[Ganascia 91]
Ganascia J.G. L'hypothèse du Knowledge Level : théorie et pratique. Rapport Laforia n° 20/91.

[Gasser&al. 87]

Gasser L., Braganza C., Herman N. Implementing Distributed AI Systems Using MACE. In Distributed Artificial Intellligence, Edité par M. N. Huns, Pitman-Morgan Kaufmann, 1987.

[Gautron 85]
Gautron P. Une approche logicielle de l'improvisation dans le Jazz. Thèse de 3ème cycle, Paris VI, IRCAM 1985.

[Georgeff 82]
Georgeff M.P. Procedural Control in Production systems. Artificial Intelligence, 18 pp. 175-201, (1982)

[Ghallab 80]
Ghallab M. Near Optimal Decision Trees for Matching Patterns in Inference and Planning Systems. 2nd International Meeting on Artificial Intelligence, Leningrad, Octobre 1980.

[Ghallab 88]
Ghallab M. Compilation de bases de connaissances. Actes des journées Nationales PRC-Greco intelligence Artificielle. Toulouse pp. 231-253. Teknea, Mars 88.

[Goldberg&Robson 83]
Goldberg A., Robson D. Smalltalk-80 : The Language and its Implementation. Addison-Wesley, 1983.

[Goldberg 84]
Goldberg A. Smalltalk-80 : The Interactive Programming Environment. Addison-Wesley, 1984.

[Goldstein&Bobrow 80]
Goldstein I, Bobrow D. Extending Object Oriented Programming in Smalltalk-80. Lisp Conference, Stanford pp. 75-81 (1980).

[Graubé 89a]
Graubé N. Metaclass Compatibility. OOPSLA '89, special issue of Sigplan, Vol. 24, N° 10, New Orleans, USA October 1989.

[Graubé 89b]
Graubé N. Architectures réflexives et implémentations des langages à taxonomie de classes en Lisp : Applications à ObjVlisp, CLOS et Telos. Thèse de l'Université Paris VI, Décembre 1989.

[Grize 90]
Grize J.-B. Logique et argumentation. 4 ième colloque de l'ARC - Progrès de la recherche cognitive, pp. 13-23, Paris 28-30 Mars 1990.

[GSI Tecsi 90]
Intelligence Service II. Manuel d'utilisation, 1990.

[Hayes-Roth 85]
Hayes-Roth B. A blackboard Architecture for control. Artificial Intelligence N° 26, pp. 251-321. (1985)

[Hayes-Roth&al. 83]
Hayes-Roth F., Waterman D.A., Lenat D.B. Building Expert systems. Addison-Wesley, 1983.

[Hofstadter 85]
Hofstadter D. Metamagical Themas : Questing for the Essence of Mind and Pattern. Bantam Books 1985.

[Hokamura&Tokoro 90]
Okamura H. Tokoro M. Concurrent Smalltalk-90. TOOLS Pacific'90, Sydney, Australie, pp. 231-244, Décembre 1990.

[Hottois 89]
Hottois G. Penser la logique. Une introduction technique, théorique et philosophique à la logique formelle. Editions Universitaires De Boeck, Bruxelles, 1989.

[IARFA90]
Recueil des Projets du cours "Smalltalk et représentation de connaissances" de J.-F. Perrot pour le DEA IARFA. Développements de nouveaux mécanismes dans NéOpus. Ecole Nationale des Ponts & Chaussées, 1990.

[IARFA91]
Recueil des Projets du cours "Smalltalk et représentation de connaissances" de J.-F. Perrot pour le DEA IARFA. Développements de systèmes experts utilisant Smalltalk et NéOpus. Ecole Nationale des Ponts & Chaussées, 1991.

[Jackson 86]
Jackson J. Control, AM records 395106-2, 1986.

[Jimenez 90]
Jimenez C. Sur l'explication dans les systèmes à base de règles : Le système PROSE, Thèse d'université Paris VI, Novembre 1990.

[Kornman 90]
Korman S. Introspection and correction in production systems. Cognitiva 90, pp. 843-846.

[Krasner&Pope 88]
Krasner G. E., Pope S. T. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. ParcPlace Systems, 1988.

[Krief 90]
Krief P. MPVC : Un système interactif de construction d'environnements de prototypage de multiples outils d'interprétation de modèles de représentation. Thèse d'Université, Paris VIII, Juin 1990.

[LaLonde 87]
LaLonde W. R. A Novel Rule Based Facility For Smalltalk. Proceedings of ECOOP'87 pp.193-198.

[LaLonde&Van Gulik 88]
LaLonde, W. R., Van Gulik, M. Building a Backtracking Facility in Smalltalk Without Kernel Support. Proceedings of OOPSLA '88 pp. 105-122.

[LaLonde&Pugh 90]
LaLonde W. R., Pugh J.R. Inside Smalltalk ,Volumes 1 et 2. Prentice Hall, 1990.

[LaLonde&al. 86]
LaLonde W. R., Thomas D., Pugh J.-R. An examplar based Smalltalk, OOPSLA '86, Portland, Oregon, USA, September 86, pp. 322-330.

[Laublet 90]
Laublet P. Représentation des connaissances et démonstration dans le système ForreEnMat. Rapport LAFORIA n° 22/90, Juillet 1990.

[Laublet 91]
Laublet P. Hybrid Knowledge Representation and Theorem Proving in Mathematics. Rapport LAFORIA n° 04/91, Février 1991.

[Laurent &al. 87]
Laurent J.-P., Thome F., Ayel J., Ziebelin D. Evaluation comparative de trois outils de développement de systèmes Experts (KEE, KC, ART). Intelligence Artificielle, volume 1, n° 2, Mai 1987.

[Laurent 83]
Laurent J.-P. La structure de contrôle dans les systèmes experts. TSI, Vol. 3, pp. 147-162, Paris, 1983.

[Laurière 78]
Laurière J.-L. A Language and a Program for Stating and Solving Combinatorial Problems. Artificial Intelligence vol 10 (1978) pp. 29-127.

[Laurière 87]
Laurière J.-L. Résolution de problèmes par l'homme et la machine. Editions Eyrolles, 1987.

[Lavaire &al.91a]
Lavaire F. Etude de faisabilité et prototypage d'un Système Expert Orienté Objet. Projet industriel, DESS Génie Informatique, Université de Nantes, Avril 1991.

[Lavaire 91b]
Lavaire F. Projet IAMI : Système Intelligent d'Analyse d'Images Médicales. Utilisation du générateur de systèmes experts NéOpus. Rapport de stage de DESS Génie Informatique, Université de Nantes, IMM, CHR, Juillet 1991.

[Léa Sombé 89]
les A sont B. Raisonnements sur des informations incomplètes en Intelligence Artificielle. PRC-GRECO Intelligence Artificielle, Editions Teknea, 1989.

[Lenat&Feigenbaum 91]
Lenat D., Feigenbaum E.-A. On the thresholds of knowledge. Artificial Intelligence, 47, pp. 185-250, (1991).

[Lenat&Guha 90a]
Lenat D., Guha R.-V. Cyc, a midterm report. AI magazine, 11 (3), pp. 30-59, (1990).

[Lenat&Guha 90b]
Lenat D., Guha R.-V. Building Large Knowledge-Based Systems. Reading, Mass., USA. Addison-Wesley, 1990.

[Lerdahl&Jackendoff 83]
F. Lerdhal, R. Jackendoff. A Generative Theory Of Tonal Music. Cambridge, MA, MIT Press, (1983).

[Lieberman 86a]
Lieberman H. Delegation and Inheritance : Two mechanisms for sharing Knowledge in Object-Oriented Systems. Bigre+Globule n° 48, Janvier 1986.

[Lieberman 86b]
Lieberman H. Using Prototypical Objects to implement shared behavior in object-oriented systems, OOPSLA '86, pp. 214-223 (1986).

[Loops 87]
Xerox Loops reference Manual. Xerox Corporation, September 1987.

[Maier&Stein 90]
Maier D, Stein J. Development and Implementation of an Object-Oriented DBMS. Research directions in Object-Oriented Programming. Shriver B. Wegner Publishers. pp. 355-392.

[Masini & al. 89]
Masini G., Napoli A., Colnet D., Léonard D., Tombre K. Les langages à objets. InterEdition, Paris, 1989.

[Mazon78]
Grammaire de la langue russe. Institut d'études slaves, Paris, 1978.

[McCarthy&Hayes 69]
MacCarthy J, Hayes, P.-J. Some philosophical problems from the stand point of Artificial Intelligence, in machine Intelligence 4, B. Meltzer & D. Mitchie (Eds), Edinbourgh University Press, Edinbourgh 1969.

[Miranker 90]
Miranker D. P. Treat : a New and Efficient Match Algorithm for A.I. Production Systems, Pitman, London, 1990.

[NewRealBook 88]
The New Real Book, Syosset, New York, 1988.

[Oracle 88]
Oracle V.5 SQL User's Manual. Oracle-France S.A., Paris, 1988.

[Pachet 88]
Pachet F. Vers un système expert de suivi d'improvisation. Rapport de DEA. Ircam, 1988.

[Pachet 89]
Pachet F. A Practical Use of Metaclasses. Proceedings of TOOLS' 1, pp. 233-242, November 13-15, Paris (1989).

[Pachet 90]
Pachet F. Mixing Rules and Objects : An experiment in the world of Euclidean Geometry. ISCIS V, 30 Oct. - 2 Nov., Nevsehir, Turkey, pp. 797-805, (1990). Aussi : Rapport LAFORIA n° 19/90.

[Pachet 91a]
Pachet F. Reasoning with objects : the NéOpus environment. Conférence East EurOOpe, Bratislava, Tchécoslovaquie, Septembre, (1991). Aussi : Rapport LAFORIA n°13/91.

[Pachet 91b]
Pachet F. NéOpus mode d'emploi. Rapport LAFORIA n° 14/91, Paris 1991.

[Pachet 91c]
Pachet F. Pour en finir avec le singe et les bananes. Rapport LAFORIA n°15/91, Paris 1991.

[Pachet 91d]
Pachet F. Du bon usage des méta-règles en NéOpus. Rapport LAFORIA n°16/91, Paris 1991.

[Pachet 91e]

Pachet F. A meta-level architecture for analysing jazz chord sequences. Proceedings of International Conference on Computer Music, 1991, pp. 266-269, Montreal.

[Pachet 91f]
Pachet F. Representing Knowledge Used by Jazz Musicians. Proceedings of International Conference on Computer Music 1991, pp. 285-288, Montreal.

[Pachet 91g]
Pachet F. Rule base inheritance. Conférence "Représentations Par objets", La grande Motte, Juin 1992.

[Pachet&Dojat 91a]
Pachet F., Dojat M. Representation of a Medical Expertise Using the Smalltalk environment: putting a prototype to work. Proceedings of TOOLS 7, Dortmund, Germany, March 31-April 2, (1992). Rapport LAFORIA n° 17/91.

[Pachet&Dojat 91b]
F. Pachet, Dojat M. NéoGanesh: an Extendable Knowledge-Based System for the Control of Mechanical Ventilation. 14 th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, October 29-Novembre 1st, Paris, à paraître, (1992).

[Parchemal 88]
Parchemal Y. Sepiar : un système à base de connaissances qui apprend à utiliser efficacement une expertise. Thèse de l'université Paris VI, Laforia, 1988.

[ParcPlace 88]
Objectworks. Smalltalk-80 r. 2.5, Advanced User's guide. ParPlace systems, 1988.

[Pastre 84]
Pastre D. Muscadet : un système de démonstration automatique de théorèmes utilisant connaissances et métaconnaissances en mathématiques. Thèse d'état, Paris VI, 1984.

[Pastre 90]
Pastre D. Déduction naturelle et utilisation de connaissances. Rapport Laforia n° 25/90, Octobre 1990.

[Perrot 86]
Perrot J.-F. Introduction à la programmation par objets, le système Smalltalk-80. Laforia, CNRS, 1986.

[Perrot 89a]
Perrot J.-F. Sur une base de règles de Jean-Louis Laurière. Communication aux journées de Villetaneuse, Mai 1989.

[Perrot 89b]
Perrot J.-F. Notes sur la progammation de règles de production. Laforia, communication interne, 1989.

[Perrot 92]
Objets et Intelligence Artificielle : des objets et des règles. Cours SOL, Février 1992.

[Perrot &Wolinski 92]
Perrot J.-F., Wolinski F. Modélisation par objets en robotique. TSI Vol. 11, 1992.

[Piersol 86]
Piersol, K.W. The Humble Reference Manual. Xerox Special Information Systems, Juin 1986.

[Pinson 87]
Pinson S. Représentation de la stratégie de contrôle dans un système d'évaluation d'entreprise. Colloque d'I.A. Caen 15-19 Septembre 1987, pp. 299-322.

[Pitrat 90]
Pitrat J. Métaconnaissances. Hermes, Paris, 1990.

[RealBook81]
The real book. The Real Book Press, 1981 (Edition pirate samizdate mais de référence).

[Rechenmann 85]
Rechenmann F. Shirka : mécanismes d'inférence sur une base de connaissances centrée-objet. RFIA, Grenoble, 1985.

[Reiter 80]
Reiter R. A logic for default reasoning. Artificial Intelligence, Vol. 13, 1980, pp. 81-132.

[Revault 91]
Revault N. Représentation de connaissances ergonomiques. Rapport de stage de DEA IARFA, Université Paris VI/ENPC, Septembre 1991.

[Rich 87]
Rich E. Intelligence Artificielle. Masson, 1987.

[Roche 88]
Roche C. Object in Expert Systems. Artificial Intelligence and Cognitive Sciences, Manchester University Press, 1988.

[Roche&Laurent 89]
Roche C, Laurent J.P. Les approches objets et le langage LR02 (KEOPS). TSI, Janvier 1989.

[Rousseaux 90]
Rousseaux Francis. Une contribution de l'intelligence Artificielle et de l'apprentissage symbolique automatique à l'élaboration d'un modèle d'enseignement de l'écoute musicale. Thèse d'université, Paris VI, Février 1990.

[Rousset 88]
Rousset M.C. Tango, moteur d'inférence pour une classe de systèmes experts avec variables. Thèse de 3ème cycle, Orsay, 1988.

[Savéant 90]
Savéant P. Raisonnement hypothétique et temps multiforme discret dans les systèmes de production : étude et implémentation. Thèse de l'université Paris VI, Décembre 1990.

[Scaletti&Johnson 88]
Scaletti C. A., Johnson R. E. An interactive Environment for Object-Oriented Music Composition and Sound Synthesis. Proceedings of OOPSLA '88 pp. 222-233.

[Shor & al. 86]
Shor M., Daly T., Ho S.L., Tibbits B. Advances in Rete pattern matching. AAAI '86, pp. 226-232.

[Shortliffe 76]
Shortliffe E.H. Computer-based Medical Consultations : MYCIN. Elsevier, USA, 1976.

[Smeci 88]
Smeci, version 1.4, manuel de référence, Ilog, Paris 1988.

[Steedman 84]
Steedman M.J. A Generative Grammar for Jazz Chord Sequences. Music Perception, Fall 1984, Vol. n° 2, N° 1, pp. 52-77.

[Thirouin91]

Thirouin L. Le hasard et les règles. Le modèle du jeu dans la pensée de Pascal. Bibliothèque d'histoire de la philosophie. Librairie philosophique J. Vrin, Paris, 1991.

[Vegdhal 86]
Vegdhal S. R. Moving Structures between Smalltalk images. Proceedings of OOPSLA '86 pp. 466-471.

[Vere 80]
Vere S. A. Multilevel Counterfactuals for Generalizations of Relationnal Concepts and Productions. Artificial Intelligence Vol 14 pp. 139-164 (1980).

[Vere 87]
Vere S. A. Induction of relational productions in the presence of background information. Proceedings of IJCAI 1987, pp. 349-362.

[Vergnaud 91]
Vergnaud G. Sciences cognitives en débat. Première école d'été du CNRS sur les Sciences cognitives. Ed. G. Vergnaud, CNRS, 1991.

[Vialatte 85]
Vialatte M. Description et implémentation du moteur d'inférence SNARK. Thèse de doctorat, Paris 6, Mai 1985.

[Vogel 88]
Vogel C. Génie cognitif. Masson, Paris 1988.

[Volle 88]
Volle P. Objects and Logic for representing Knowledge. Rapport Laforia n° 88/35.

[Volle 89]
Volle P. Coréférence et mécanismes déductifs dans un langage de représentation par objets. PRC-IA journée du pôle I. Caen 15 Juin 89.

[Voyer 87]
Voyer R. Moteurs de systèmes Experts. Editions Eyroles, 1987.

[Voyer 89a]
Voyer R. Implémentation d'architectures efficaces pour la représentation des connaissances. Application aux langages Loopsiris et OKS. Thèse de troisième cycle de l'université Paris VI. Laforia, 1989.

[Voyer 89b]
Voyer R. LOOPSIRIS : un générateur de systèmes inférentiels. Rapport LAFORIA n° 89/44.

[Voyer 89c]
Voyer R. Oks : un langage pour la représentation des connaissances. Rapport LAFORIA n° 89/45.

[Voyer 89d]
Voyer R. La compilation en réflexe : un nouveau modèle d'implémentation efficace des systèmes avec variables fonctionnant en chaînage avant. Rapport LAFORIA n° 89/46.

[Voyer 89e]
Voyer R. Oks : un langage de programmation par réflexes. Rapport LAFORIA n° 89/47.

[Wertz 85]
Wertz H. Intelligence Artificielle, application à l'analyse de programmes. Masson, Paris, 1985.

[Wirfs-Brock&Wilkerson 89]
Wirfs-Brock A., Wilkerson B. Variables Limit Reusability. Journal of Object-Oriented Programming, Vol. 2, n° 1. May-June, 1989, pp. 34-40.

[Wolinski 90]
Wolinski F. Etude des capacités de modélisation systémique des langages à objets appliquées à la représentation de robots. Thèse de l'université Paris VI, Septembre 1990.

[Wolinski & Perrot 91]
Wolinski F., Perrot J.-F. Representation of Complex Objects : Multiple Facets with Part-Whole Hierarchies. Proceedings of ECOOP '91. Genève, Juillet 91. Rapport Laforia n° 7/91.

XI. Annexes

Avant-propos

Nous présentons ici la syntaxe complète des règles NéOpus, (incluant les appels récursifs et les assertions) ainsi que la typologie complète des nœuds Rete.

XI.1. Syntaxe complète des règles NéOpus

XI.1.1. Syntaxe

Voici la syntaxe complète des règles NéOpus, dans la syntaxe BNF, sauf pour les expressions Smalltalk décrites en commentaire.

Remarques : les mots-clés sont en gras (actions, [,], {, }, NOT). Les blancs et retour-chariot ne sont pas pris en compte.

```
<règle NéOpus> ::==
                <nom>
                <déclaration de variables>
                <prémisses>
          actions
                <partieAction>
          {finalState
                <assertion>}
```

<nom> ::== un symbole Smalltalk

<déclaration de variables> ::= | <déclaration> { . <déclaration> } *|

<déclaration> ::== <type><nomDeVariable>{<nomDeVariable> }*

<type> ::== <nomDeClasse> | <type-motclé>

<nomDeClasse> ::== <nom de classe Smalltalk>

<type-motclé> ::== **Local | Global**

<prémisses> ::== <prémisse>
ou <prémisse> <prémisses>

<prémisse> ::== <prémisse positive>
ou <prémisse négative>
ou <prémisse locale>
ou <prémisse locale déclenchante>
ou <prémisse création but>

<prémisse positive> ::==toute expression Smalltalk, utilisant les variables déclarées précédemment, et terminée par un point.

<prémisse négative> ::==**NOT** <déclaration de variables> <prémisse positive>

<prémisse locale> ::==
affectation d'une variable déclarée comme Local en partie déclaration.

<prémisse locale déclenchante> ::==

affectation d'une variable déclarée comme d'une classe Smalltalk en partie déclaration.

<partieAction> ::==
toute séquence d'expressions Smalltalk, avec éventuellement une déclaration standard de variables locales (entre barres verticales), et utilisant les variables déclarées en partie déclaration, et un certain de pseudo-messages qui sont expansés pour leur compilation dans les classes dynamiques (Cf V.2.2) pour la liste des pseudo-méthodes).

Voici la syntaxe des constructions utilisées pour manipuler les assertions :

<assertion> ::== **{** <prémisse positive>**}**
Rrmq : les messages apparaissant dans l'assertion n'ont pas à être des messages Smalltalk

<prémisse création but> ::==
<prémisse création but sans variable> **|**
<prémisse création but avec variable>

<prémisse création but sans variable> ::== **[**<prémisse positive>**]**

<prémisse création but avec variable> ::==
[<declaration variable but> **|** <prémisse positive>**]**

<declaration variable but> ::== déclaration de variable de bloc Smalltalk.

XI.1.2. Typologie complète des prémisses

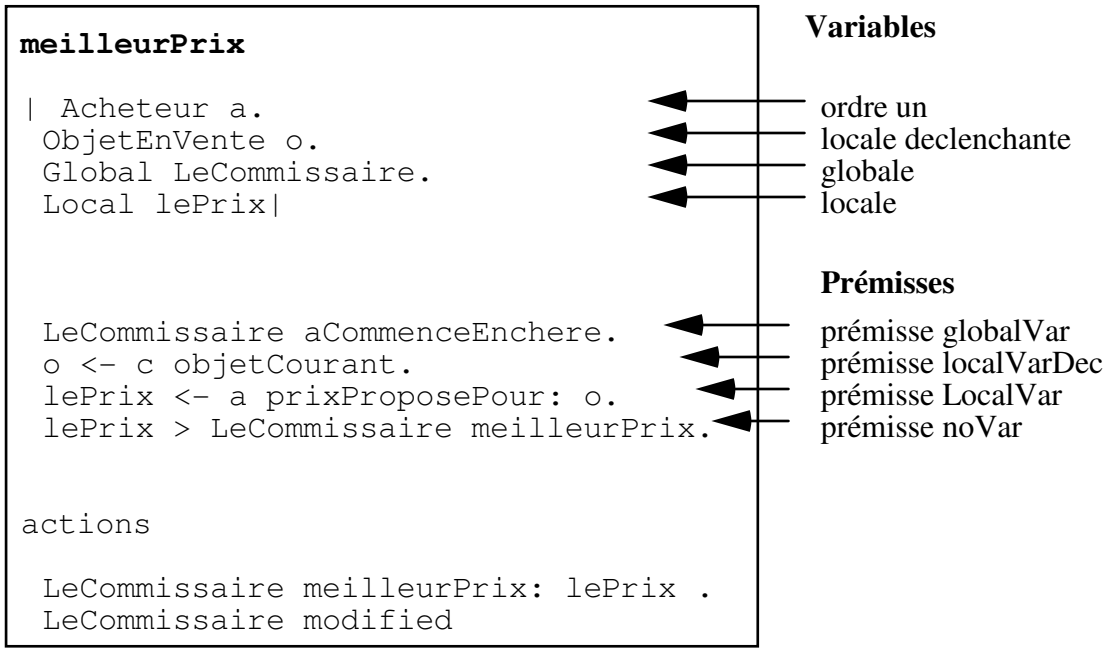
Les prémisses ont un type (un symbole) qui va déterminer la classe de nœud Rete adéquate. Ce type est déterminé en fonction de la nature des variables dans la prémisse : variables libres (première apparition dans la règle), liées (non libres), affectée (variables locales ou locales déclenchantes), présence de variable globale, nombre de variables libres, et prémisses particulières.

Voici la liste des types de prémisses pris en compte actuellement par le parser.

#general :
prémisse comportant au moins deux variables libres, dans affectation, ni variable globale.
#noVar
prémisse sans aucune variable libre, ni affectation.
#oneVar
prémisse avec une et une seule variable libre.
#localVar0
prémisse avec une et une seule variable locale (Dummy). Pas de variable libre.
#localVarOne
prémisse avec une variable locale et une seule variable libre.
#localVarGeneral
prémisse avec une variable locale, et des variables libres.

#localVarDec0
prémisse avec une variable locale déclenchante, et aucune variable libre.
#negativeOneVar
prémisse négative avec une seule variable négative. Pas de variable libre.
#negativeGeneral
prémlisse négative avec plus d'une variable libre négative. Pas de variable libre.
#createGoalNoVar
prémisse de création de but, sans variable.
#createGoalVar
prémisse de création de but avec variable.
#globalVar
prémisse avec une variable globale. Pas de variable libre, ni locale.

XI.1.3. Exemple complet de règle NéOpus



XI.2. Pseudo-méthodes

Voici la liste complète des pseudo-méthodes, qui sont expansées pour être compilées dans les classes dynamiques.

go expansé en : `goFor: self ruleBase,`
modified expansé en : `modifiedFor: self ruleBase,`
remove expansé en : `removeFor: self ruleBase,`
areModified expansé en : `areModifiedFor: self ruleBase,`
areRemoved expansé en : `areRemovedFor: self ruleBase,`
areGoing expansé en : `areGoingFor: self ruleBase,`
selfBase expansé en : `self ruleBase,`

Les messages de broadcast (`areModifiedFor:`, `areGoingFor:` et `areRemovedFor:`) sont, eux, implémentés dans la classe abstraite `Collection`, pour être compris de tous les objets à caractère collectif (Cf. IV.3.6.4).

XI.3. Résumé du vocabulaire

- variable de règle : variable déclarée en partie déclaration d'une règle.
- variable d'ordre un: variable déclarée comme étant d'une classe Smalltalk.
- variable locale : variable déclarée Dummy (ou Local).
- variable globale : variable déclarée comme Global.
- variable locale déclenchante : variable de règle Smalltalk, affectée dans une prémisse.
- typage
  - des variables de règle (simple ou naturel)
  - des prémisses
  - des variable dans une prémisse (libre liée affectée)

XI.4. Index des notions importantes

XII. Table des matières