Titre : **Du bon usage des méta-règles en NéOpus**

Auteur : **François Pachet**

E-mail : **fdp@litp.ibp.fr**
Adresse : **Laforia, Université Paris VI, Tour 46-00, 2° étage**
4 Place Jussieu
75252 Paris Cedex 05
Téléphone : **(33.1) 44.27.70.10**
Fax : **(33.1) 44.27.70.10**

Résumé
Nous proposons ici une architecture déclarative de contrôle pour le système NéOpus. Nous allons définir les mécanismes d'évaluation de bases de règles en donnant un statut de première classe aux objets utilisés par le contrôle.
Nous décrivons d'abord rapidement le système NéOpus, moteur d'inférence en marche avant intégré dans Smalltalk-80. Ce système permet d'écrire des règles de production d'ordre un portant sur tout objet Smalltalk. Les règles sont organisées en bases de règles, qui sont elle-mêmes des objets Smalltalk. Le problème du contrôle se pose alors de façon très naturelle comme l'écriture de connaissances sur l'évaluation des objets particuliers que sont les bases de règles.
Pour ce faire, la notion d'évaluateur est introduite, comme objet associé à une base de règles, et représentant un certain état de son évaluation. Les connaissances de contrôle s'expriment alors comme des règles portant sur les évaluateurs. Représenter le contrôle se fait en créant d'une part un type d'évaluateur particulier, et d'autre part en écrivant une base de méta-règles opérant sur ce type d'évaluateur. Toute une hiérarchie d'évaluateurs de plus en plus complexes est créée ainsi qu'une hiérarchie parallèle de bases de méta-règles est construite.
Nous montrons divers exemples de bases de méta-règles implémentant les stratégies usuelles de contrôle, ainsi qu'une stratégie originale utilisant des appels récursifs au moteur en partie prémisse, tiré d'un système de raisonnement géométrique.
Enfin nous montrons comment la notion d'héritage de bases de règles, et un environnement puissant permettent de réaliser effectivement notre technique.

abstract
We descibe here a declarative architecture for control for the NéOpus system. This mechanism gives a first class status to control objects. We first describe briefly the NéOpus system, a forward-chaining first order inference engine inserted into the Smalltalk world. Rules are organized in rule bases which are themselves Smalltalk objects. The control problem is then simply stated in terms of writing rules about those particular rule base objects.
This requires the introducation of the notion of Evaluator, which represent the state of evaluation of a given rule base. Control knowledge is expressed as rules that operate on those evaluator objects. Representing a particular type of control in this architecture consists in creating a particular class of evaluator together with a particular rule base that

operate on this type of evaluator.
A hierarchy of evaluators is built in order to represent more complex control strategies, together with a hierarchy of meta-bases. We show various examples of meta-bases and evaluators that represent standard, as well as less standard, control strategies. An original example of recursive call to the inference engine is also presented, taken from a geometrical reasoning system. Last, we give some methodological tools that help this multi-level rule based programming.

Du bon usage des méta-règles en NéOpus

François Pachet

1. Introduction

Nous présentons ici l'architecture de contrôle du système NéOpus, moteur d'inférence en marche avant, opérant dans l'univers Smalltalk. Comme dans tout système de règles en marche avant, le problème du contrôle des inférences se pose rapidement, dès que l'on écrit des bases de règles non triviales.
Le système NéOpus permet de spécifier le contrôle de manière procédurale (par méthodes), localement pour chaque base de règles.
Cependant, bien que cette architecture procédurale bénéficie de tous les avantages de la programmation par objets (en particulier l'héritage), elle ne permet pas de représenter de manière satisfaisante les contrôles non triviaux.
Notamment, la notion de but, ou d'état souhaité du système, se laissent mal modéliser de cette manière. D'une manière générale les *objets de contrôle* (buts, stratégies, plans ..) n'ont pas un statut clair.
Enfin, l'architecture même de NéOpus tend naturellement à proposer une architecture de contrôle déclarative. En effet, NéOpus permet d'écrire des règles sur tout objet Smalltalk. Or les bases de règles, les règles, les conflict sets, les réseaux Rete sont tous des objets Smalltalk à part entière. Ecrire des règles parlant de ces objets ne pose donc aucun problème technique à priori. On bénéficie alors de tout l'arsenal de NéOpus pour écrire ces méta-règles, comme l'héritage des bases de règles, le typage des variables et accessoirement l'environnement de programmation.

L'architecture proposée ici va donc permettre de spécifier le contrôle des inférences à l'aide de règles, de la même manière que les règles permettent de spécifier des connaissances portant sur les *objets du domaine*.

Nous commençons par un bref rappel de la syntaxe de NéOpus. Pour plus de précisions sur NéOpus, se reporter à [Atkinson&Laursen, et Pachet (1),(2),(3),(4),(5),(6),(7)].

Note : le texte général est en palatino 12. `Le code Smalltalk est en Courier 10`, *les règles NéOpus sont en Zapf Chancery 12 (encadrées)*.

2. NéOpus

Le système Opus [Laursen87] est une transposition du moteur d'OPS5 [Brownston85] dans Smalltalk. Système hybride, Opus associe donc deux types de représentations : les objets Smalltalk, et les règles d'ordre un en marche avant. Le système actuel, appelé NéOpus, est une complète réécriture du système original, et comprends, outre quelques modifications syntaxiques et une interface évoluée, une architecture de contrôle déclarative, la notion

d'héritage entre bases de règles, divers extensions de la notion de variable de règle, et un environnement de programmation.
Utilisé comme terrain d'expérimentation dans plusieurs contextes[1], NéOpus est maintenant dans un stade opérationnel. Nous décrivons rapidement ses caractéristiques majeures, et le mode de raisonnement qui en découle.

2.1. Les règles

La transposition des règles OPS5 dans l'univers Smalltalk consiste à substituer aux habituelles relations n-aires (les literalize OPS) les objets Smalltalk eux-mêmes. Les objets Smalltalk n'étant accessibles que par envoi de messages (encapsulation), la structure des règles se trouve alors profondément étendue : Au lieu de spécifier des contraintes sur la valeur des attributs des objets (comme en OPS5), les prémisses Opus permettent de spécifier toute contrainte entre objets, exprimable sous forme d'une expression Smalltalk booléenne quelconque. De plus, les prémisses Opus permettent de filtrer plusieurs objets en même temps[2] de façon à autoriser toute expression Smalltalk. La partie action elle aussi est plus générale puisqu' elle autorise toute expression Smalltalk.
Ainsi, pour écrire par exemple que le singe doit se déplacer pour aller chercher une échelle, on écrira la règle suivante, en supposant que les classes `Singe` et `Echelle` ont été définie comme ayant une variable d'instance `position`, et les méthode d'accès (`position`, `vaA:`, `neTientRien`) canoniques :

```
allerAEchelle
        | Singe s . Echelle e |
        (s position = e position) not.
        s neTientRien.
actions
        s vaA: e position.
        s modified.
```

Cette règle s'interprétera comme : "Pour tout singe `s`, et toute échelle `e`, si la position de `s` est différente de celle de `e`, et que `s` ne tient rien, alors le singe se déplace à la position de `e`".
La partie action va modifier l'état du singe, en l'occurrence sa position. Il faut le signaler au système par la transmission `s modified`, pour qu'il (le système) puisse éventuellement remettre à jour l'état d'instanciation des règles filtrant cet objet. Pour plus de détails sur la notion de modification (qui est une instanciation particulière du *frame-problem*) se reporter à [Pachet (2),(7)].

2.2. Les bases de règles

[1] Notamment au CEMAGREF, pour la représentation de machines agricoles [Charbonnel], et comme support à un système explicatif [Alvarez], à l'université de Nantes pour un système expert gérant des images médicales, pour le projet Sagesse (Ministère de l'intérieur), comme générateur de scénario, à l'Ecole Nationale des Ponts&Chaussées comme environnement de développement pour le DEA IARFA, pour un système de contrôle de ventilateurs pour patients en réanimation [Dojat&Pachet].
[2] Ce qui nécessite une modification de l'algorithme de compilation Rete [Forgy] pour prendre en compte des noeuds multi-entrée

Les règles sont organisées en bases de règles qui sont des classes abstraites Smalltalk. Elle apparaissent comme des méthodes (d'instance) pour ces classes, mais sont compilées par un compilateur particulier, qui met à jour un réseau Rete [Forgy] adapté au système NéOpus. Les bases de règles sont toutes sous-classe, directement ou indirectement, de la classe racine `OpusRuleSet`. L'activation d'une base de règles est décrite de manière procédurale dans la métaclasse `OpusRuleSet class`. En particulier, activer une base de règles s'effectue par l'envoi du message `execute` à la base de règles. Ex : `ReglesDuSinge execute`.

3. Contrôler le raisonnement

3.1. Le problème du contrôle 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. Ce choix peut se faire traditionnellement de plusieurs manières : de façon figée (avec les deux stratégies Lex et Mea en Ops5), par coefficients de priorité sur les règles (Mycin, NExpert, Humble), par méta-actions dans les règles (Snark).
L'idée que l'évaluation d'une base de règles est un processus qui, lui aussi, utilise des connaissances (ou des métaconnaissances [Pitrat]), nous conduit à chercher une représentation déclarative du contrôle.
De plus, le caractère 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.

Par ailleurs, le problème du contrôle est lié directement à la manière dont l'algorithme d'évaluation (le moteur d'inférence) est représenté. Représenter le contrôle consiste à agir de manière cohérente sur cet algorithme. Classiquement, les actions sur le contrôle d'une base de règles se font de deux manières : par **greffe**, ou par **mots-clés** :

Greffe - en fournissant des points d'entrée dans l'algorithme d'évaluation. C'est ce qui se passe avec l'architecture procédurale de contrôle de NéOpus. Grâce à l'héritage, les bases de règles peuvent redéfinir localement tout ou partie de l'algorithme d'évaluation, en redéfinissant la ou les méthodes appropriées.

Mots-clés - en proposant une liste (fixe) de méta-actions, utilisables dans le règles, qui modifieront de manière dynamique le comportement du moteur (par exemple `Préférer` dans Snark).

Ces deux possibilités sont très restrictives. La première Cf [Pachet 2] ne permet pas de donner un statut de première classe aux objets de contrôle (tout est codé sous forme de méthodes Smalltalk). La deuxième suppose que l'on connait à l'avance toutes les opérations "à caractère méta" nécessaires pour représenter toutes les connaissances de contrôle; en outre elle mélange les actions de contrôle aux règles elles-mêmes, ce qui aboutit à des bases de règles difficilement lisibles.

Nous avons donc choisi une autre voie, dite par **substitution**. Celle-ci consiste à remplacer l'algorithme de contrôle d'une base de règles `RB`, par l'évaluation d'une autre base de règles `MRB`.
L'évaluation de `MRB` aura pour effet d'évaluer `RB`. La base de règles `MRB` sera appelée méta-base (de `RB`), et les règles de `MRB` seront appelées méta-règles. Cependant rien ne différencie syntaxiquement les bases de règles des bases de méta-règles, ni les règles des méta-règles. Seule l'utilisation qui est faite des bases de règles justifiera *l'appellation* méta (c'est donc une appellation non contrôlée).

Nous allons maintenant étudier comment l'évaluation d'une base de (méta-) règles permet d'aboutir à ce résultat (i.e. l'évaluation d'une autre base de règles).

3.2. les bases de règles sont des objets

Les règles en Opus sont groupées en bases de règles, qui sont elles-mêmes des objets Smalltalk[3]. Les bases de règles ont comme structure principale un conflict set[4] (représentant l'ensemble des règles déclenchables), lui-même instance de la classe `OpusConflictSet`.
Les bases de règles contiennent ainsi toutes les informations relatives à l'état d'instanciation des règles, en particulier l'ensemble des règles candidates (obtenu via le conflict set), ainsi que le réseau Rete et toute sa structure (les jeux d'instanciation pour chacune des prémisses de chaque règle).

Evaluer une bases de règles en chaînage avant, en NéOpus, peut se décomposer canoniquement en trois *étapes* :

[3]En fait ce sont des classes, mais en Smalltalk les classes sont des objets. Cette organisation permet par ailleurs d'intégrer très naturellement NéOpus à l'environnement Smalltalk, en bénéficiant de toutes les fonctionnalités déjà présentes pour les classes (en particulier le browser, la gestion des protocoles, des catégories, l'héritage, le fileIn/fileOut ...).
[4]les bases de règles étant de classes, leurs structures sont donc représentées par des variables d'instance de métaclasses

- initialisation, et identification des objets à filtrer
- boucle :
  tant que le conflict set n'est pas vide (ou toute autre *condition d'arrêt*),
  déclencher une règle
- fin, éventuellement rendre un résultat

Ce schéma minimal d'évaluation en marche avant est facilement représentable sous forme de méthodes Smalltalk. Mais il peut bien sûr être complexifié.

Une base de méta-règles standard doit être capable de représenter cette architecture, et donc de gérer un statut de l'évaluation (état initial, boucle ou final), et de décider du choix d'une règle, suivant tel ou tel critère.
Il est clair alors que les bases de règles en tant qu'objets Smalltalk ne contiennent pas assez d'informations à elles-seules pour gérer leur évaluation : comment représenter la notion *d'étape*, de *condition d'arrêt* ?
Plutôt que de complexifier la structure des bases de règles pour y incorporer ces informations, nous décidons de donner un statut de première classe aux objets de contrôle.

3.3. Réifier le contrôle : l'Evaluateur

Nous introduisons donc un objet nouveau, appelé évaluateur, qui va représenter l'état d'évaluation d'une base de règles. Son rôle va être de permettre l'écriture de règles NéOpus représentant le contrôle d'une base de règles.

Nous allons décrire les évaluateurs progressivement, en complexifiant leur structure par étapes.
Nous présentons d'abord un évaluateur minimal, permettant d'écrire une première base de méta-règles, qui reproduit le même processus de contrôle que celui décrit de manière procédurale. Cette classe est ensuite complexifiée progressivement pour prendre en compte les processus de contrôle plus complexes.

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,

- 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 (Cf § suivant pour des exemples) trouveront ici une véritable définition en termes d'objet.

3.3.1. Evaluateur-zéro

Cet objet a comme structure minimale :

la *base de règles* dont il représente l'évaluation,
un *statut*, symbole représentant l'étape courante,
le *contexte* d'initialisation de la base de règles,
une *condition d'arrêt*, bloc Smalltalk rendant un booléen.

```
Object subclass: #Evaluateur
      instanceVariableNames: 'ruleBase status context stopCondition'
```

3.3.1.1. Création d'un évaluateur.

La création d'un évaluateur se fait par l'instanciation de la classe d'évaluateur (par défaut ce sera `Evaluateur`) correspondant à la base de règles évaluée, et l'initialisation des variables d'instance en fonction de l'évaluation choisie.

3.3.1.2. 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 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: aSC
      ^self defaultEvaluatorClass
            fromRuleBase: self
            status: #init
            stopCondition: aSC
            context: aContext!
```

3.3.1.3. Création simplifiée

Des messages de création simplifiés permettent la création d'évaluateurs canoniques. Le message `makeEvaluator` rend un évaluateur dont la condition d'arrêt est la vacuité du conflict set et le contexte, le contexte de la base de règles :

```
makeEvaluateur
      ^self makeEvaluateurWithMessage: conflictSet isEmpty]

makeEvaluateurWithMessage: aMessage
      ^self makeEvaluateurWithMessage: aMessage context: context
```

La classe d'évaluateur est fonction de la méta-base (via la méthode `requiredEvaluatorClass`, qui est redéfinie pour chaque méta-base).

```
!OpusRuleSet class methodsFor: 'evaluateurs'!

makeEvaluateur
    ^self makeEvaluateurWithMessage: nil!

makeEvaluateurWithContext: aContext stop: aBlockOrAnAssertion
    ^self defaultEvaluatorClass
          fromRuleBase: self
          status: #init
          stopCondition: aBlockOrAnAssertion
          context: aContext

makeEvaluateurWithMessage: aMessage
    ^self defaultEvaluatorClass
          fromRuleBase: self
          status: #init
          stopCondition: aMessage
          context: context

defaultEvaluatorClass
    "on va chercher cette information dans la metabase si elle existe"

    metaBase isNil ifTrue: [^Evaluateur].
    ^metaBase requiredEvaluatorClass

requiredEvaluatorClass
    "le classe d'evaluateurs utilisee dans la base de regles. redefinit dans les sous classes de DefaultMeta"

    ^Evaluateur
```

3.3.2. Fonctions d'un évaluateur

L'évaluateur doit être capable de renseigner sur l'état d'évaluation de sa base de règles. Pour ce faire, un certain nombre de méthodes d'accès sont définies. Elles concernent le statut, le contexte d'initialisation et la condition d'arrêt.

3.3.2.1. A propos du statut

On peut ici s'offrir le luxe de méthodes d'accès génériques, (copiées sur le comportement des processus) telles que `resume` ou `terminate`, pour l'accès au statut d'un évaluateur.

```
!Evaluateur methodsFor: 'status'!

status
	^status

status: unSymbole
	statut <- unSymbole

resume
	self status: #actif

terminate
	self status: #inactif

isActive
	^(status = #inactive) not
```

3.3.2.2. A propos de la condition d'arrêt

La seule information dont on a besoin pour l'instant à propos de la condition d'arrêt est son succès ou échec. Le message `reussi` évalue donc la condition d'arrêt et rend le booléen résultat.
Si la condition d'arrêt n'est pas un bloc Smalltalk, mais une assertion (Cf plus loin) un type particulier d'évaluation est alors effectué, de manière à rendre `false` en cas d'échec de l'évaluation.

```
!Evaluateur methodsFor: 'evaluation'!

nonReussi
	^self reussi not!

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 isNil ifTrue:
		[stopCondition _ [ruleBase conflictSet isEmpty]].
	(stopCondition isKindOf: Assertion)
		ifTrue: [^self errorSignal handle:
		[:ex | ex returnWith: false]
				do: [stopCondition evaluate]].
	^stopCondition value
```

3.3.2.3. A propos du contexte

La variable d'instance `context` de l'évaluateur est redondante avec celle des bases de règles. Les mêmes méthodes d'accès sont donc définies que pour `OpusRuleSet`. Cette redondance se justifie par le fait que les deux modes de contrôle (procédural et déclaratif) cohabitent. La gestion du contexte est la même dans les deux cas.

3.3.3. Utilisation d'un évaluateur

L'évaluateur est un objet dont l'existence n'est justifiée que par les méta-règles. Il sert d'intermédiaire entre les méta-règles et les bases de règles. Ainsi, peu de méthodes (excepté les méthodes d'accès) sont écrites dans cette classe, qui ne sert, pour l'instant, que comme structure de donnée.

3.3.3.1. 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.

3.3.3.1.1. 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 contexte '
```

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 :

3.3.3.1.2. 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.

3.3.3.1.3. Nouveau protocole d'évaluation

L'évaluation d'une base de règle 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.

3.3.3.1.3.1. Redéfinition de la méthode execute

La 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)
```

3.3.3.1.3.2. 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 :

- En cas d'absence de méta-base :
redirection vers un contrôle procédural 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 return
```

- 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` à la méta-base.

Ainsi une base de règles peut être considérée indépendamment de son exécution, et peut être exécutée de manière différente sans avoir à subir aucune modification.

3.4. DefaultMeta

Nous donnons ici la base de méta-règles standard, reproduisant le contrôle par défaut tel qu'il est défini plus haut. Cette base de méta-règles (appelée *DefaultMeta*) , sera la racine des autres bases de méta-règles, au sens de l'héritage de bases de règles [Pachet 6].

Une base de règles est filtrée ainsi que son ou ses évaluateurs, et l'exécution de la base de règles consiste en effets de bords sur la base de règles (en particulier son conflict set) et son (ou ses) évaluateur(s).

Le méta base *DefaultMeta* comporte des règles groupées en trois protocoles, décrivant les trois états standards d'évaluation décris en 3.2.2 :

a. Protocole d'initialisation

Deux méta-règles décrivent l'initialisation d'une base de règles. Nous décomposons en effet en deux le processus d'initialisation, pour mettre en évidence les effets de bords induits, en fonction de *l'existence* du contexte. Si le contexte existe, on envoie ce contexte (par le message *e sendContext*), et on déclare comme modifiés l'évaluateur `e`, ainsi que son conflict set. Si le contexte n'existe pas, on change simplement le status de `e` en `#loop`.

initObjects

```
      | Evaluateur e |
      e status = #init.
      e context exists.
actions
      e status: #loop.
      e sendContext.
      e modified.
"le conflict set a pu changer"
      e conflictSet modified.
```

initNoObjects

```
      | Evaluateur e |
      e status = #init.
      e context exists not.
actions
      e status: #loop.
      e modified.
```

b. Boucle de saturation

La saturation se décrit en deux méta-règles. La première teste si le conflict set n'est pas vide (*c nonVide*), auquel cas déclenche la première règle (*c declenchePremiereRegle*). La deuxième est déclenchée quand aucune règle n'est plus déclenchable. Le status de l'évaluateur est alors modifié.

On déclare `c` comme modifié, ainsi que `e`, puisque la condition d'arrêt de `e` a pu changer (en l'occurrence devenir vraie (Cf la notion de r-modification dans [Pachet 7]).

loop1

```
| Evaluateur e .ConflictSet c |

        e status = #loop.
        c <- e conflictSet.
        e reussi not.
        c notEmpty.
actions
        c declencheDefault.
        c modified.
"declencher une regle a pu modifier e"
        e modified.
```

loopEnd

```
| Evaluateur e |

        e status = #loop.
        e reussi.
actions
        e status: #end.
        e modified.
```

c. Terminaison

Une règle terminant la boucle si l'évaluateur a terminé. Dans ce cas, l'évaluateur est mis hors service (message `suspend`).

end

```
| Evaluateur e |

        e status = #end.
actions
        e suspend
```

Enfin, de manière a prendre en compte les éventuelles sous-classes d'évaluateur que nous allons fabriquer par la suite, il faut définir un typage naturel (Cf [Pachet 2]) pour la base `DefaultMeta`. Ceci se fait en redéfinissant la méthode `initTypage` pour `DefaultMeta` :

```
!'DefaultMeta class methodsFor: 'typing'!

initTypage
            self setNaturalTyping
```

Cette base de méta-règles `DefaultMeta` est très générale puisqu'elle définit le mécanisme de saturation en marche avant, sans but ni condition d'arrêt. Elle implémente le mécanisme par défaut de NéOpus, sans aucune stratégie de déclenchement particulière : La règle choisie est la première règle du conflict set. Mais avec cette architecture il est maintenant (plus) facile de spécifier des contrôles plus spécifiques. Nous donnons quelques exemples de bases de méta-règles moins standard dans le paragraphe suivant.

4. déclinaisons du contrôle

Nous présentons ici quelques bases de règles construite à partir de notre mécanisme. La construction d' un type e contrôle conduit à une double spécialisation : une spécialisation d'une classe d'évaluateur, et une spécialisation d'une base de méta règles préexistante.

4.1. trace du raisonnement

Produire une trace d'un raisonnement peut se faire très simplement sans modifier ni la base de connaissances, ni le moteur, mais en utilisant une méta-base spécialisée dans la trace, et qui sera associée à la base de règles courante.
Cette méta base (appelée *MetaTrace*) comporte les mêmes règles que la méta base par défaut (*DefaultMeta*), à l'exception de la règle `loop1` (effectuant un déclenchement de règle), qui comporte, en partie action, une procédure de trace (par exemple affichage dans une fenêtre ou écriture dans un fichier du nom de la règle déclenchée).
Voici par exemple la règle *loop1* redéfinie dans la base de règles *MetaTrace* :

```
loop1
  | Evaluateur e .ConflictSet c |

            e status = #boucle.
            e conflictSet = c.
            c non Vide.
actions
            | a |
            a := c premiereRegle.
            self affiche: 'je declenche la regle ',a nom.
            c declenche: a.
            c modified.
```

Ainsi, toute base de règles évaluée par cette méta-base sera-t-elle tracée, et ce à moindre coût et sans aucune recompilation. Une série de méta-bases plus complexes, et spécialisées dans l'explication ont été réalisées par [Alvarez 91].

Cette méta-base ne requiert pas d'évaluateur particulier, tout évaluateur peut être filtré par la règle. Seule la règle de déclenchement est modifiée. Un certain nombre de telles méta-bases peuvent être construites dans cet esprit :

4.2. Etc caetera

Voici une liste (non exhaustive) des bases de méta-règles qui redéfinissent simplement la règle de bouclage. Nous donnons pour chacune d'entre elle la définition de `loop1`. Les autres règles sont identiques à celles de `DefaultMeta`. De manière pratique, on utilisera l'héritage de bases de règles [Pachet 6] pour profiter pleinement de ces points communs.

MetaSequence.
déclenche la première règle suivant un tri par le nom.

```
!MetaSequence methodsFor: 'sequence loop'!

loop1
"si l'evaluateur n'est pas reussi, on declenche la premiere regle du conflict set
trie a partir du sortBlock defini plus bas : tri lexicographique"
  | Evaluateur e . OpusConflictSet c |

                e status = #loop.
                c _ e ruleBase conflictSet.
                e reussi not.
                c isEmpty not.
actions
                | uneRegle |
                uneRegle _ c firstWithSortBlock: [:a :b | a nom > b nom].
                c  declenche: uneRegle.
                c modified.
```

MEA.
Dans cette base, la règle choisie est celle filtrant le plus récent objet. Cette information est obtenue via le message `timeTag`, qui rend la date de création de l'objet (plus exactement un entier, incrémenté à chaque nouvelle création/modification de l'objet). Cette méthode est définie par défaut dans `Object` comme rendant toujours 0 (on ne peut pas recompiler `Object`). Certaines classes redéfinissent la méthode `timeTag` en incorporant une gestion effective de la date de création/modification.

```
loop1
| Evaluateur  e . OpusConflictSet c |
                e status = #loop.
                c _ e conflictSet.
                c notEmpty.
                e reussi not.
actions
                | uneRegle |
                uneRegle _ c regleAyantLePlusRecentObjet.
                c declenche: uneRegle.
                c modified.
```

MetaUser.
La règle déclenchée est demandée à l'utilisateur via un popUp menu.

```
!MetaUser methodsFor: 'loop'!
loop1
| Evaluateur e . OpusConflictSet c |

        e status = #loop.
        c _ e ruleBase conflictSet.
        e reussi not.
actions
        | uneRegle |
        uneRegle _ c askForRegle.
        c declenche: uneRegle.
        c modified.
```

MetaHistoire.
Gère un historique des règles déclenchées.

```
loop1
"si l'evaluateur n'est pas reussi, on declenche la premiere regle du conflict set"
  | EvaluateurHistorique e . OpusConflictSet c |

        e status = #loop.
        c _ e ruleBase conflictSet.
        e reussi not.
        c notEmpty.
actions
        e addRule: c first.
        c declencheFirst.
        c modified. e modified.
```

4.3. organisation des règles en paquets, gestion d'un agenda

L'organisation des règles en paquets de règles à l'intérieur d'une base de règles peut se gérer aussi au niveau 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.
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[5].

```
Evaluateur subclass: #EvaluateurAgenda
      instanceVariableNames: 'agenda'
```

[5] La notion d'agenda dans les systèmes de règles n'a pas de définition unique. Nous choisissons ici une définition simple, mais d'autres définitions peuvent aussi être représentées dans la même architecture.

L'agenda est lui même un objet, instance de la classe `Agenda`, et 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
```

Enfin, la méthode `fini` permet de tester si l'agenda est terminé ou non :

```
!Agenda methodsFor: 'test'!

fini
      ^indexPaquetCourant = paquets size
```

Une méta-base (`MetaAgenda`) représente ce type d'évaluation. Elle ne déclenchera les règles que si elles appartiennent au paquet courant de l'agenda. Cela se traduira par une méta-règle du type suivant, ou l'on utilisera une variable locale (déclarée comme `Local`), affectée à la première règle du paquet courant :

```
boucleDans UnPaquet
  | EvaluateurAgenda e . ConflictSet c . Local regle  |

      e status = #boucle.
      e conflictSet = c.
      c nonVide.
      regle := c regleDuPaquet: e agenda paquetCourant.
actions
      c declenche: regle.
      c modified.
```

La gestion des paquets se fait alors par une méta-règle qui passera au paquet suivant, par exemple, lorsque plus aucune règle n'est déclenchable (on aura défini préalablement la méthode *aucuneRegleDePaquet*: dans la classe `OpusConflictSet`) :

```
passageAuPaquetSuivant
  | EvaluateurAvecAgenda e . ConflictSet c |
      e status = #boucle.
      e conflictSet = c.
      c nonVide.
      c aucuneRegleDePaquet: e agenda paquetCourant.
actions
      e passeAuPaquetSuivant. e modified.
```

La terminaison consiste à passer l'évaluateur au status #end, lorsqu'aucune règle n'est plus déclenchable, et que l'agenda est terminé.

```
!MetaAgenda methodsFor: 'agenda'!

finAgenda
  | EvaluateurAgenda 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!
```

4.4. Gestion de priorités sur les règles

On peut gérer une liste de règles à déclencher en priorité. Cela se fait en créant une classe d'évaluateur spécialisée `EvaluateurPriorites`. Cette classe définit une variable d'instance pointant sur une liste de règles (plus exactement leur noms). Deux méthodes d'accès sont simplement définies (`priorites` et `priorites:`) :

```
Evaluateur subclass: #EvaluateurPriorites
      instanceVariableNames: 'priorités '
```

On définit parallèlement dans le conflict set des méthodes permettant de tester qu'une règle est bien dans une liste donnée.
La méthode `premiereRegleDeNomDans:` permet de récupérer la règle déclenchable la plus prioritaire. La méthode `aUneRegleDeNomDans:` permet de tester si une règle déclenchable est dans la liste passée en argument.
Les deux règles de bouclage sont alors redéfinies de manière à déclencher la règle la plus prioritaire, par rapport à la liste de règles de l'évaluateur.

```
!MetaPriorites methodsFor: 'loop'!
```

loop1

```
"on declenche la premiere regle si aucune regle n'est dans la liste de priorites"
  | EvaluateurPriorites e . OpusConflictSet c |

e status = #loop.
c _ e  conflictSet.
e reussi not.
(c aUneRegleDeNomDans: e priorites) not.
actions
| a |
a _ c first.
c  declenche: a.
c modified. e modified
```

loop2

```
"on declenche la premiere regle prioritaire"
  | EvaluateurPriorites e . OpusConflictSet c .
  Local regle |

                e status = #loop.
                c _ e  conflictSet.
                e reussi not.
                regle _ c premiereRegleDeNomDans: e priorites.
actions
                c  declenche: regle.
      c modified. e modified
```

5. Extensions : la notion d'assertion

5.1. Appels récursifs en partie prémisse

Grâce à la représentation des évaluateurs comme objets de première classe, des raisonnements plus complexes peuvent être représentés en faisant intervenir plusieurs évaluateurs concurrents, créés de façon dynamique au cours du raisonnement.
C'est le cas par exemple lors de certains raisonnements géométriques [Pachet 4], faisant intervenir des objets intermédiaires. Ces objets sont créés pour les besoins de la cause, et peuvent provoquer un autre raisonnement; comme par exemple, dans le théorème suivant:

> "soit le triangle ABC. Soit F le point A déplacé de BC, Si le parallélogramme ABCF est un losange, alors ABC est isocèle, de base BC".

Ce type de connaissance peut très bien se représenter en NéOpus, en introduisant la notion d'évaluateur récursif, dérivé (sous-classe) de celle d'évaluateur avec condition d'arrêt, introduite en 4.1., et ayant comme structure supplémentaire l'objet d'attention pour lequel le but est à atteindre, ainsi que la prémisse dans la quelle l'évaluateur à été créé.

```
EvaluateurAvecConditionArret subclass: #EvaluateurRecursif
      instanceVariableNames: 'nouvelObjet premisse'
```

Ces évaluateurs sont créés par un appel particulier en partie prémisse d'une règle. Cet appel aura pour effet la création d'un nouvel évaluateur pour la base de règle courante, et sa prise en compte par la méta base associée à la base de règles.
La syntaxe de cet appel consiste envoyer le message suivant à la base de règles.

```
considere: unObjet jusquA: uneConditionDArret
      nouvelEvaluateur := Evaluateur new.
      nouvelEvaluateur  conditionDArret: uneConditionDArret.
      nouvelEvaluateur nouvelObjet: unObjet.
      nouvelEvaluateur premisse: premisseCourante.
      nouvelEvaluateur goInto: metaBase
```

Cet évaluateur sera géré, comme tous les évaluateurs de la base de règles au niveau supérieur, à l'aide de méta-règles particulières, qui sauront en particulier, une fois la condition d'arrêt satisfaite pour cet évaluateur, continuer la propagation au niveau de la prémisses où l'évaluateur à été crée.

Ainsi le théorème précédent peut s'énoncer de la façon suivante (en supposant la classe `Triangle`, et la méthode *estUn: unType* qui permet de représenter qu'un objet géométrique est de type *unType*)

```
triangleIsoceleParLeLosanze
        |Triangle t. Local BC A B C P F|

        BC <- t HorizontalSide.
        A <- t pointOpposedTo: BC.
        B <- BC leftMost.
        C <- BC rightMost.
        P <- A + C - B.
        F <- Parallelogram new with: A with: B with: C with: P.
GeometricalRules
  considere:
(GeometricalObject new addType: newObject as: Parallelogram)
        jusquA: [newObject estUn: Losange].

actions
t estUn: TriangleIsocele.
t modified.
```

Cette création dynamique d'évaluateurs n'est pas équivalente à un mécanisme de chaînage arrière (comme l'est par exemple la négation par l'échec dans certains systèmes de règles). En effet, créer un nouvel évaluateur ne signifie pas forcément que le système va s'en occuper tout de suite, ni même que le système va s'en occuper du tout !.
Une base de méta-règles (*MetaRecursifs*) permettra de décider des stratégies à suivre pour les évaluateurs crées dynamiquement.
La base de méta-règles implémentant la prise en compte systématique des évaluateurs récursifs ajoutera simplement deux règles à la base de méta-règles `DefaultMeta`; pour initialiser ces évaluateurs, et redéclencher la règle ou l' évaluateur a été créé en cas de succès de cet évaluateur.

La prise en compte d'un objet intermédiaire par un but récursif se traduira par une méta-règle du type suivant :

```
initEvaluateurRecursif
  | EvaluateurRecursif e  |

        e status = #init.
        e objets isEmpty not.
actions
        e objets areGoing.
        e status: #loop.
        e modified.
        e conflictSet modified
```

Le redéclenchement de la prémisse au cas ou un évaluateur intermédiaire a réussi peut s'écrire alors de la façon suivante (on a définit la méthode `retrigger` dans `EvaluateurRecursif` qui va redéclencher la prémisse suivante de la règle qui a provoqué la création de cet évaluateur) :

```
boucleRedeclencheEvaluateur
"si l'evaluateur est  reussi, on redeclenche la premisse suivante"
  | EvaluateurRecursif e . OpusConflictSet c |
            e status = #loop.
            c == e conflictSet.
            e reussi.
actions
            e retrigger; suspend; modified.
            c modified.
```

Ici encore des base de méta-règles plus spécialisées peuvent être écrites, qui spécialiseront ces bases générales.

5.2. La notion d'assertion

5.2.1. Idée, Motivations.

La notion d'assertion va permettre de parler d'une règle déclenchable de manière plus précise. En effet, pour l'instant les seules informations que l'on a sur une règle déclenchable sont : son nom, le paquet dans lequel elle est définie (le protocole), les objets la filtrant. On ne sait rien sur ce que fait véritablement la règle. Le choix de la règle à déclencher est donc indépendant de l'action de la règle sur l'environnement.
Pour permettre de parler des règles en fonction de leur effet sur l'environnement Smalltalk, nous introduisons la notion d'assertion. Une assertion est un objet Smalltalk représentant un état du système. Cet objet s'apparente à un arbre syntaxique instancié. Toute expression Smalltalk booléenne peut être une assertion. Les assertions se distinguent syntaxiquement des expressions Smalltalk par le fait qu'elles sont entre accolades. Nous verrons que les assertions peuvent être aussi des expressions non Smalltalk.
Par exemple l'assertion suivante :

```
{unSinge tient = aBanane}
```

décrit le fait que l'objet dénoté par `unSinge` (dans le contexte d'évaluation de l'assertion) tient l'objet dénoté par `uneBanane` (idem).

La principale différence entre une assertion et une expression Smalltalk est que l'assertion est manipulable (comme arbre syntaxique), alors que les expressions Smalltalk ne sont accessibles que comme chaînes de caractères, ou bien comme méthodes compilées (instances de `CompiledMethod`).
C'est un objet à part entière instance de la classe `Assertion` ou d'une de ses sous-classes. Les sous classes de `Assertion` représentent les différents types d'expressions Smalltalk. Par exemple, la classe `Message0` représente les variables (ou constantes), `Message1` les messages unaires (comme `unePersonne pere`), `Message2` les messages à un argument (comme `unSinge tient = aBanane`).

5.2.2. Utilisation dans une règle

Nous allons utiliser les assertions pour décrire l'effet d'une règle sur l'environnement de manière utilisable dans les méta-règles. Pour ce faire, nous introduisons une troisième partie aux règles NéOpus : la partie `finalState`. En plus de la partie condition et de la partie action, la partie `finalState` va justement être une assertion, décrivant l'effet de la règle lorsqu'elle est déclenchée.

Exemple
Reprenons notre exemple du singe et des bananes. La règle suivante va, quand elle sera déclenchée, être décrite par l'assertion représentant le fait que le singe est maintenant à la position de e, soit :

allerAEchelle
	| *Singe s . Echelle e* |
	(s position = e position) not.
	s neTientRien.
actions
	s vaA: e position.
	s modified.
finalState
	{s position = e position}

Il est intéressant de regarder ce que génère le compilateur à partir de cette définition. Dans la classe dynamique de la base de règles la méthode `allAEchelleFinalState` est générée, en plus des méthodes nécessaires à la création du réseau Rete :

```
!MonkeyRulesDynamic methodsFor: 'essai'!

allAEchelleFinalState
  ^Message2
    objet: (Message1 objet: (Message0Variable objet: i1 name: #i1)
         selector: #position)
    selector: #=
    argument: (Message1 objet: (Message0Variable objet: i2 name: #i2)
         selector: #position)
```

Cette méthode rendra une instance de `Message2`, qui représente les expressions Smalltalk à un argument, dont l'objet est lui même une instance de `Message1` (représentant les expressions Smalltalk sans argument; ici i1 position), le selector est #=, et l'argument une autre instance de `Message1` .. 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 dans [Pachet 7]).
On voit ici que l'état obtenu après déclenchement de la règle `allerAEchelle` peut être consulté avant que celle-ci ne soit déclenchée. Cet état est obtenu en envoyant le message `allerAEchelleFinalState` au token arrivant au bout du réseau Rete (le noeud et le token sont tous les deux contenus dans l'instance de règle déclenchable qui est alors créée).

5.2.3. 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.

Une base de règles pour gérer les assertions par défaut

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]. 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}
```

5.2.4. 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 dans [Pachet 3].

6. Une programmation multi-niveau

Ce type d'architecture induit une programmation à plusieurs niveaux. Les méta-bases étant elle-mêmes des bases de règles, le mécanisme précédent peut aussi bien s'appliquer à elles-mêmes, si le besoin s'en fait sentir. Comme nous l'avons vu plus haut, si aucune méta-base n'est sélectionnée pour une base de règles, un contrôle procédural par défaut est attribué.
Dans l'exemple précédent, ou plusieurs évaluateurs sont concurrents, plusieurs méta-règles pourront être déclenchables à un moment donné. Il faut encore choisir parmi les différents évaluations. Ce choix nécessite alors un niveau supplémentaire (niveau 3), qui saura décider quelles évaluations privilégier.
Pour spécifier un parcours en profondeur d'abord, par exemple, il faudra choisir l'évaluation la plus récemment créée (les évaluateurs sont étiquetés à leurs création par un index incrémenté). On pourra utiliser la méta-base `MetaProfondeur`, pour contrôler la méta-base `MétaRecursif`. La base MetaProfondeur peut à son tour être elle-même contrôlée par une méta-base, par exemple MetaTrace, et ainsi de suite.
Notons toutefois que le mécanisme n'est pas à proprement parler réflexif, puisqu'une méta-base ne peut se contrôler elle-même.
Dans notre exemple, pour implémenter un parcours en profondeur d'abord, trois niveaux sont nécessaires. Trois niveaux semblent suffisant en pratique pour représenter les stratégies communément employées.

7. Méthodologie

Cette architecture est rendue possible par l'emploi de deux outils méthodologiques : l'héritage des bases de règles, qui reproduit le mécanisme de l' héritage de classes et permet la factorisation de code, et un interface utilisateur puissant.

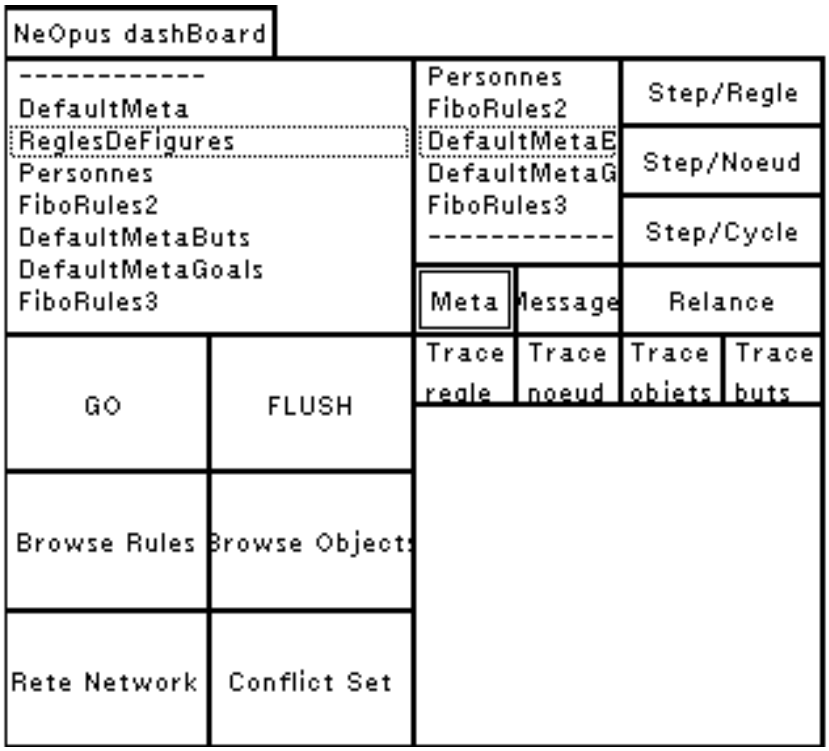
7.1. l' Héritage de bases de règles

Un mécanisme d'héritage, analogue à celui proposé par les langages objets à été transposé au domaine des bases de règles [Pachet 6]. Une base de règles étant une classe, elle peut hériter d'une autre base de règles, auquel cas les règles de la super-base seront héritées. De la même façon que pour les méthodes, en cas de conflit (plusieurs règles de même nom définie dans la hiérarchie), la règle définie au plus bas niveau est préférée.
Ce mécanisme très pratique permet de constituer très facilement des bibliothèques de bases de méta-règles, en ne redéfinissant effectivement que les règles qui différent des règles des super bases, ou en ajoutant des règles supplémentaires.
Si les divers niveaux méta permettent de spécifier le contrôle, l'héritage des bases de règles entre elles permet de spécialiser chacun des niveaux.
Une hiérarchie de bases de règles est constituée, parallèlement à une hiérarchie

d'évaluateurs.
Un bon exemple d'utilisation d'héritage des bases de règles de contrôle peut être trouvé dans [Pachet 5] ou on représente les connaissances de contrôle d'une expertise médicale.

7.2. L'environnement de NéOpus

Cette programmation multi-niveaux est rendue possible de manière pratique grâce à un environnement interactif très complet. En particulier, des browsers multiples permettent d'éditer les règles d'une base, ainsi que des méta-bases; un éditeur de conflict set, multiple lui aussi, permet d'inspecter en cours de raisonnement les objets et l'état des bases de règles. Un éditeur graphique permet de visualiser aussi les réseaux de compilation Rete associés aux bases de règles.
Nous montrons les browsers obtenus pour une évaluation de l'exemple géométrique, pour lequel trois niveaux sont définis : la base de règles de départ (*ReglesDeFigures*), sa méta-base (*DefaultMetaButs*) et la méta-méta-base (*DefaultMetaProfondeur*)



Le tableau de bord Opus permettant de sélectionner pour chaque Base de règles, une base de méta règles.

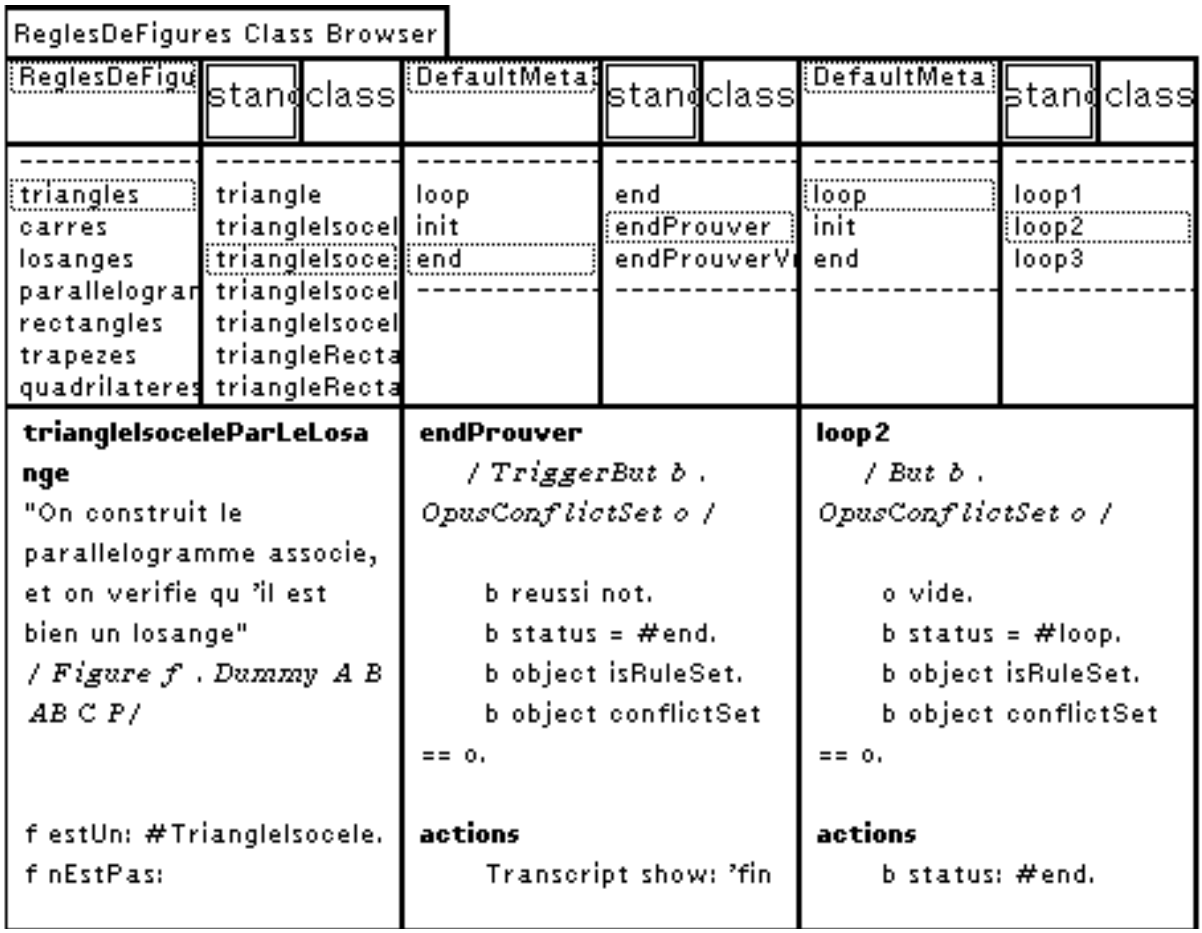


figure 3: le triple browser

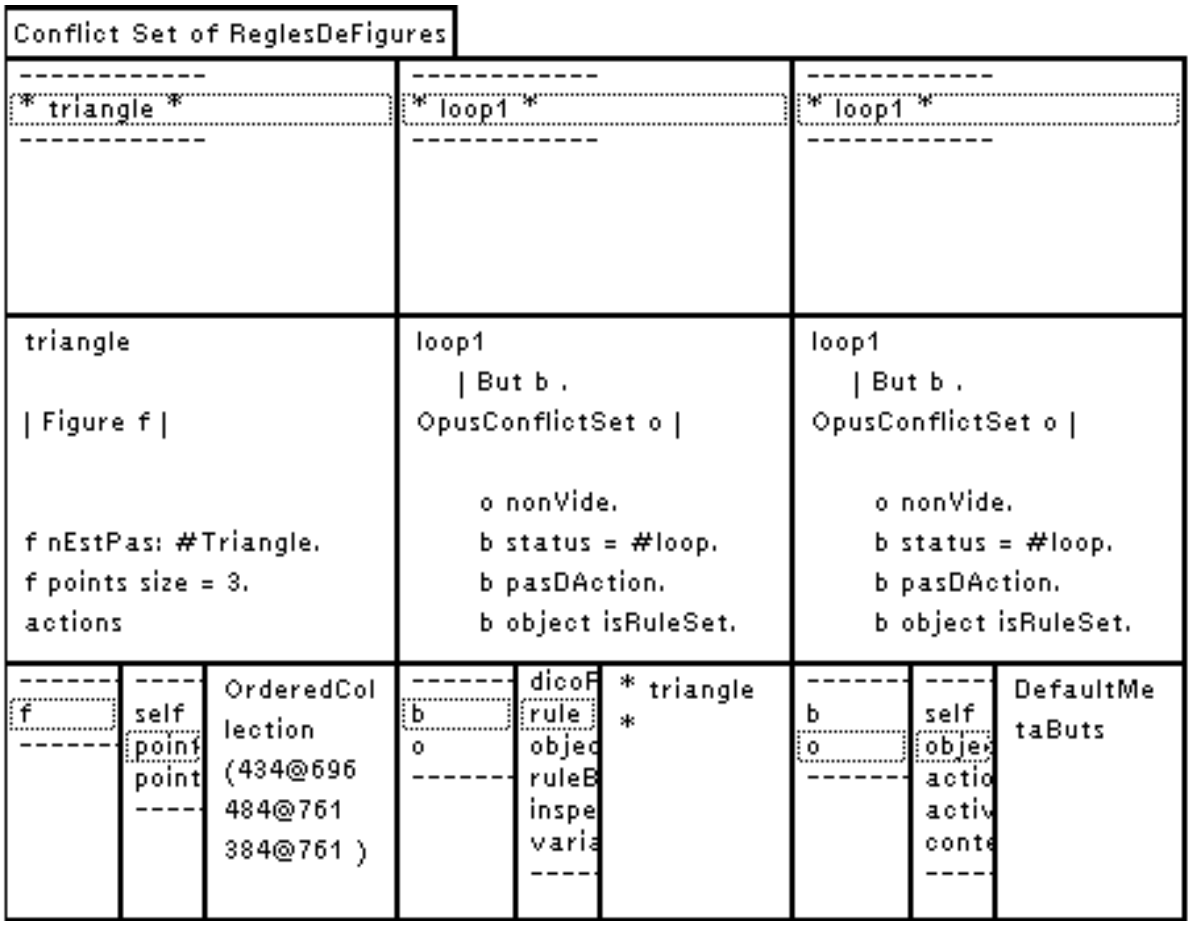


figure 4 : le conflict set multiple

8. Conclusion

Nous avons décrit l'architecture déclarative de contrôle de NéOpus. Le problème du contrôle y est envisagé de manière naturelle en donnant un statut de première classe aux objets de contrôle. La notion d'évaluateur est introduite, comme réification du mécanisme d'évaluation, et permet de représenter l'évaluation d'une base de règles de manière déclarative. Les mécanismes standards d'évaluation sont décrits avec cette architecture, en construisant une double hiérarchie : une hiérarchie d'évaluateurs et une hiérarchie de méta-bases.

9. Références

Alvarez I.
Explication comparative dans les systèmes experts. 3ièmes Journées nationales sur l'explication. Avignon 91.

Brownston L. & al.
Programming Expert systems in OPS5. An Introduction to rule-Based Programming. Addison-Wesley Publishing Company 1985.

Charbonnel S.
Etude et réalisations de l'environnement du générateur de systèmes experts NéOpus. Rapport de stage de DESS, CEMAGREF, Nantes 1990.

Forgy C. L.
Rete : A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence Vol. 19 (1982) pp. 17-37.

Ganascia J.G
L'hypothèse du Knowledge Level : théorie et pratique. Rapport Laforia n° 20/91.

Goldberg A., Robson D.
Smaltalk-80 : The Language and its Implementation.Addison-Wesley, 1983.

Laursen J. , Atkinson R.
Opus : A Smalltalk Production System. OOPSLA '87 pp. 377-387.

Pachet F. (1)
Reasoning with objects : the NéOpus environment, East EurOOpe, Bratislava, Septembre 91. Rapport LAFORIA n°13/91, Juillet 91.

Pachet F. (2)
NéOpus mode d'emploi. Rapport LAFORIA n° 14/91, Paris 1991.

Pachet F. (3)
Pour en finir avec le singe et les bananes. Rapport LAFORIA n°15/91, Paris 1991.

Pachet F. (4)
Mixing Rules and Objects : An experiment in the world of Euclidean Geometry. ISCIS V, Turkey 1990.

Pachet F. (5)
Representation of a Medical Expertise using the Smalltalk Environment : Putting a Prototype to Work. Rapport LAFORIA n° 17/91.

Pachet F. (6)
Rule Base Inheritance. Rapport LAFORIA à paraitre.

Pachet F. (7)
Thèse. A paraitre. LAFORIA, Paris 1991.