Pour en finir avec le singe et les bananes

François Pachet
Laforia

e-mail: fdp@laforia.ibp.fr

Abstract

We describe here two versions of the legendary rule base "the Monkey and the bananas", originally introduced by [Brownston], written in the NéOpus system. The first version is a direct translation of the original version in NéOpus and uses all the Object-Oriented features of the system. We criticize this version by stressing on its lack of extensibility, and by showing that control is coded in a very implicit way. A second version, using the declarative control architecture of NéOpus is then introduced. By using the notion of assertion, we separate clearly the two levels of representation : domain objects (the monkey, the bananas ...) and control objects (the goals of the monkey).

Résumé

Nous présentons ici deux versions de la base de règles du *singe et des bananes,* présentée originellement par [Brownston], dans le système NéOpus. Une première version est la traduction directe en NéOpus de la version originale, utilisant pleinement l'intégration de NéOpus dans Smalltalk. Nous critiquons cette première base de règles en montrant qu'elle n'est pas extensible, et que le contrôle y est codé de manière très implicite. Une deuxième version, utilisant l'architecture déclarative de NéOpus et la notion d'assertion est présentée, dissociant clairement deux niveaux de représentation : le niveau du domaine (les singes, les bananes ...) et le niveau de contrôle (les buts du singe).

1. Introduction

La base de règles du Singe et des bananes, présentée en entier dans [Bronwston], a comme but de représenter les actions effectuées par un objet animé (en l'occurrence un singe) afin d'attraper des objets non animés (des bananes), qui sont de poids variables, et situés dans une pièce à trois dimensions.
L'idée est de représenter par règles la génération de sous-buts nécessaires à accomplir un but initial. Par exemple, si l'objet est en hauteur, il faut chercher une échelle, si l'on doit attraper un objet, il faut préalablement lâcher celui que l'on tient ...
D'autre part, cette base de règles, et c'est peut-être là son intérêt majeur, a un but pédagogique évident, car elle met en œuvre les grands principes de la programmation en marche avant. Notamment ici, la simulation de la marche arrière par les objets buts induit un style de programmation très particulier, que nous critiquerons, mais qui fournit une bonne base de travail.

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 cependant 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".

3. Première version en NéOpus

Nous présentons ici une version NéOpus de la base de règles, en montrant comment le passage des structures de faits OPS5 aux classes Smalltalk change déjà considérablement l'allure de cette base.

3.1. 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 aussi nous affranchir de cette programmation en définissant un certain nombre de méthodes d'accès plus lisibles, qui rendront *l'implémentation* de ces objets transparente. De plus ces méthodes pourront factoriser certaines actions non liées au raisonnement à proprement parler (comme les affichages), et permettront d'alléger (et donc d'abstraire) la partie action des règles.

. La classe `TestCase` ne nous intéresse pas ici, nous effectuerons les tests en définissant des méthodes de classe Smalltalk, bien mieux adaptées.

Nous définissons donc quatres classes, en écrivant un certain nombre de méthodes d'accès. Celles-ci sont réparties en deux protocoles (au sens de Smalltalk) : 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:`)[1].

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` :

---

[1] 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.
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 d'objets modifiés (les `modified`) en partie conclusion des règles.

```
!Monkey methodsFor: 'rule testing'!

holdsSomething
      ^objectHeld exists

holdsNothing
      ^self holdsSomething not

isHolding: anObject
      ^objectHeld == anObject

isNotHolding: anObject
      ^(objectHeld == anObject) not

!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

goTo: aPosition
      Transcript show: 'le singe va a ' , aPosition printString; cr.
      at _ aPosition.

jumpOn: f
      Transcript show: 'le singe  saute a ' , f printString; cr.
      on _ f

take: anObjet
      Transcript show: 'le singe attrape ' , anObjet printString; cr.
      objectHeld _ anObjet.
      anObjet on: 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, encore une fois, tout est 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 :

```
Object subclass: #MonkeyGoal
      instanceVariableNames: 'status objet to type '
```

Trois méthodes 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.

3.2. La base de règles

Reprenons les exemples de règles décrit précédemment. Se porter à l'annexe 1 pour la base de règles complète.

Notre base de règles se définit comme sous-base de la base racine `OpusRuleSet` :

```
OpusRuleSet subclass: #MonkeyRules
      classVariableNames: ''
      category: 'OPUS-rules'
```

Les règles sont organisées en cinq protocoles : `at`, `hold`, `on`, `climb`, `termination`.

La règle de génération de sous-buts.

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* [Pachet 4], 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

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`.

```
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

```
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

La première règle utilise naturellement une prémisse négative, avec l'opérateur NéOpus NOT. La deuxième sera traitée dans un deuxième temps, par les méta-règles.

```
!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 dans un premier temps, car nous voulons garder pour l'instant une stratégie de déclenchement simple.

Exemples de lancement

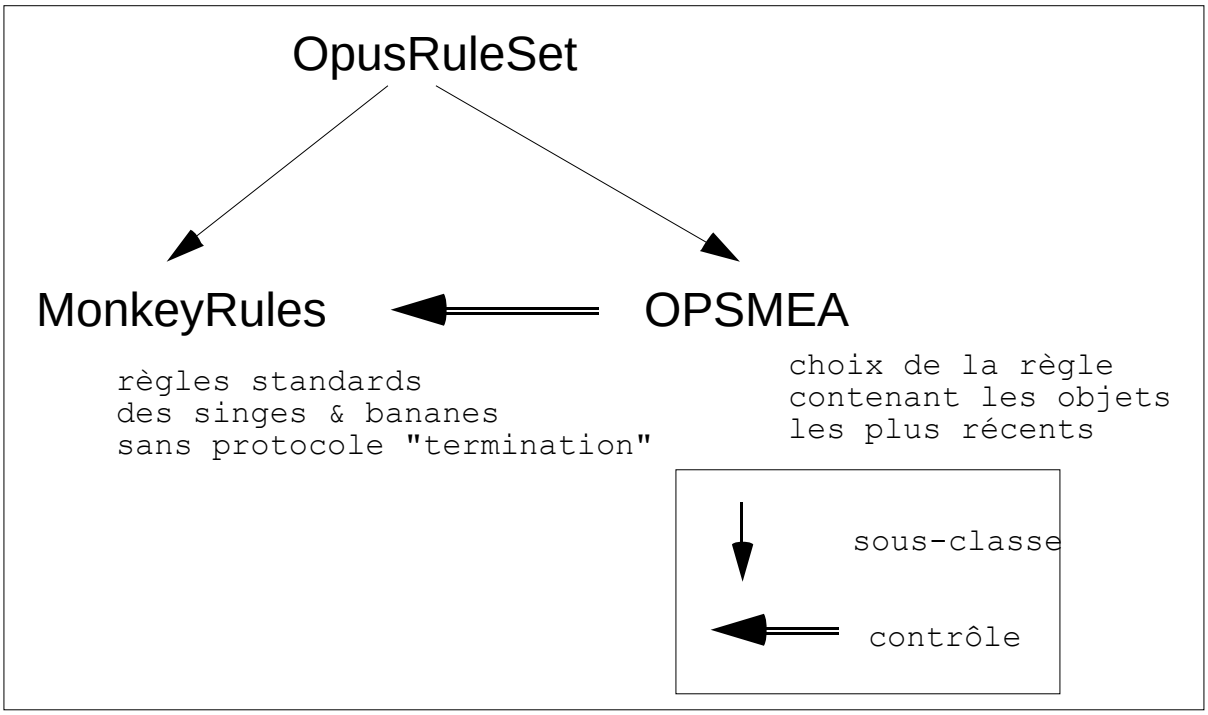
Voici deux méthodes de lancement, définies dans `MonkeyRules class`, transposées elles aussi de [Brownston]. On crée un certain nombre d'instances des classes concernées, puis on initialise le contexte de `MonkeyRules`, en enfin on lance l'exécution :

```
!MonkeyRulesEnglish class methodsFor: 'exemple'!

general
      |  s b  ladder banana blanket couch |

self setNaturalTyping."on veut utiliser l'héritage de classe Smalltalk"

couch _ Couch new  at: 7@7; weight: #heavy; on: #floor.
ladder _ Ladder new  at: 4@3; weight: #light; on: #floor.
banana _ Banana new  at: 9@9; weight: #light; on: #ceiling.
blanket _ Blanket new  at: 7@7; weight: #light.
s _ Monkey new at: 7@7; on: couch; objectHeld: blanket.

b _ MonkeyGoal new status: #active; type: #hold; objet: banana.

self addInContext:couch and: ladder and: banana and: blanket and: s and: b.
self execute

generalOnLadder
      | s b ladder banana blanket couch |

self setNaturalTyping.

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.
```

Pour être correctement lancée, la base de règles nécessite un parcours en profondeur d'abord du graphe d'état, de manière à satisfaire en priorité les buts fils générés par les règles.

Les buts sont des `EvaluateurEtiqueté (1)`**.**

Cela se traduira en NéOpus par l'association de la méta-base `OPSMEA` à la base `MonkeyRules`, via le tableau de bord NéOpus. La base de règles OPSMEA choisit de déclencher la règle filtrée par les objets les plus récents. Le fraicheur d'un objet est définie par la méthode timeTag. Cette méthode est définie par défaut dans Object commme rendant toujours 0. Elle est redéfinie dans les sous classes pour lesquelles on veut effectivement implémenter la notion de timeTag. Une telle classes est la classe `EvaluateurEtiqueté`. Nous définissons `MonkeyGoal` comme sous classe d'`EvaluateurEtiquete`. Nous verrons dans la deuxième version des raisons plus précises de choisir cette superclasse..



Contrôle standard pour `MonkeyRules`

Résultat

Voici le résultat du déclenchement pour l'exemple `generalOnLadder` (avec affichage des règles déclenchées décalé d'une tabulation), tel qu'il s'affiche dans la fenêtre Transcript :

```
        declenchement de * holdObjectCeilingAtObj *
        declenchement de * atObjectOnHolds *
        declenchement de * holdObjectNotCeilingOn *
        declenchement de * onFloor *
le singe  saute a floor
        declenchement de * holdObjectNotCeiling *
le singe attrape ladder
        declenchement de * atObject *
le singe amene ladder a 7@7
        declenchement de * holdObjectHolds *
        declenchement de * holdNil *
le singe lache ladder
        declenchement de * holdObjectCeilingOn *
        declenchement de * onPhysicalObject *
le singe grimpe sur ladder
        declenchement de * holdObjectCeiling *
le singe attrape bananas
        declenchement de * congratulations *
bravo tous les buts ont ete atteints
```

4. Critiques

Même si la réécriture de la base de règles en NéOpus redonne à celle-ci une nouvelle fraîcheur, cette première version amène un certain nombre de critiques.

4.1. Une représentation globale des objets.

La base de règles utilise implicitement des objets d'ordre zéro. Le singe, comme l'échelle, doivent être uniques, sinon la base de règles rend des résultats incohérents. En effet, les objets `but` n'ont aucun lien structurel avec les objets `Monkey`, et `Ladder`. Par exemple, lancer la base avec deux instances de singe au lieu d'une conduit à des incohérences remarquables.
Il existerait cependant deux solutions : utiliser effectivement les variables d'ordre zéro [Pachet 1] ou bien ajouter un lien fonctionnel entre `MonkeyGoal` et `Monkey`. Nous verrons plus loin une solution plus générale à ce problème.

4.2. Une très forte interconnexion des règles.

Il est très difficile de rajouter des règles à cette base. Par exemple, comment rajouter simplement la notion d'empilement d'objets (pour déplacer un objet qui est sous un autre, il faut d'abord déplacer le premier etc...).

4.3. Redondance de règles.

Les règles de satisfaction de but sans effets de bords ont un caractère extrêmement redondants. Elles représentent toutes la même idée : si un but est actif, et que l'état des objets vérifie le but, alors le but est satisfait. Mais la structuration des buts ne permet pas d'exprimer une connaissance aussi générale. Les objets buts sont, d'une certaine façon, trop instanciés.

4.4. Un contrôle implicite.

La base ne marche que si l'on privilégie les règles parlant des buts les plus récents. Sinon on obtient des incohérences dans le déclenchement.
De plus, les règles de fin de session doivent être interprétées en dernier. Cela est réalisé en OPS par le simple fait que la stratégie MEA privilégie les règles les plus spécifiques. La règle `impossible` n'ayant qu'une prémisse, elle sera déclenchée en dernier. En fait, cette règle, comme la règle `congratulations`, n'a rien à faire dans cette base. Le caractère méta de ces deux règles peut être rendu par l'intermédiaire de l'adjonction d'une méta-règle de contrôle telle que :

> "si il y a des règles déclenchables du paquet "termination", mais qu'il y a d'autres règles déclenchables, alors enlever ces règles du conflict set."

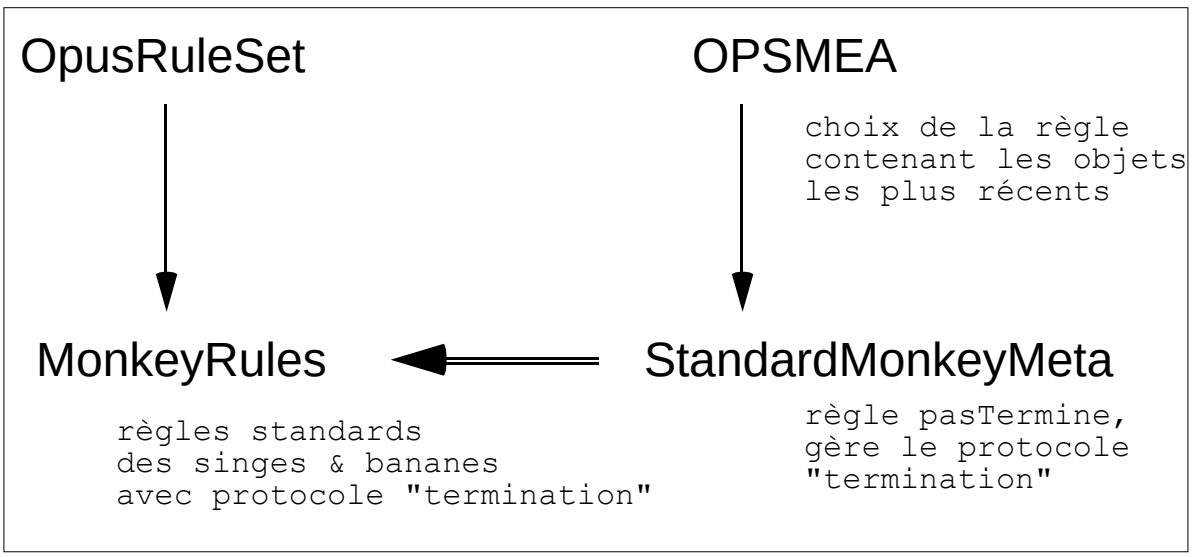
Cela se fait simplement en NéOpus, en définissant une méta-base de contrôle pour notre base de règles :

```
OPSMEA subbase: #SingeMetaStandard
      globalObjects: ''
      category: 'Monkey-rules'
```

`MonkeyMetaStandard`, ajoute simplement la méta-règle suivante :

```
!SingeMetaStandard methodsFor: 'termination'!

pasTermine

   | Evaluateur e. OpusConflictSet c  |

      e status = #loop.
      c _ e conflictSet.
      c aDesReglesDeProtocole: #termination.
      c aDesReglesDeProtocoleDifferentDe: #termination.

actions
      c enleveRegles: (c reglesDeProtocole: #termination).
      c modified
```

`MonkeyMetaStandard` étant sous-classe de OPSMEA, la règle `pasTermine` sera prioritaire par rapport aux règles héritées de OPSMEA, ce qui assure qu'aucune règle du protocole `termination` ne sera déclenchée si des règles d'autres protocoles sont déclenchables.



Contrôle standard avec termination pour la base `MonkeyRules`

4.5. Eléments de réponse

4.5.1. Les buts ont deux fonctions

Le principal défaut de cette base de règles vient en fait de la structuration des objets buts, qui remplissent deux fonctions simultanément : une fonction *assertionnelle* et une fonction de *contrôle*.

1. Une fonction assertionnelle.

La classe `GoalMonkey` définit trois variables d'instances : `type`, `at`, `to`, qui peuvent être interprétées comme une *représentation embryonnaire d'assertion sur le monde des singes*. En effet, ces trois variables représentent un certain état souhaité du système (plus exactement du singe), comme :

le singe tient la banane, ne tient rien.
le singe est à telle position
le singe est sur tel objet
le singe tient tel objet à tel endroit...

Ces assertions sur le singe sont représentées par une combinaison de valeurs pour les trois variables d'instance.

2. Une fonction de contrôle : les buts sont des EvaluateurEtiquetes (2).

Les buts sont hiérarchisés entre eux de manière arborescente. Un but non satisfait donne naissance à des sous-buts, dont la satisfaction contribue à celle de leur père. Cette hiérarchisation est représentée à deux endroits :

- dans la variable d'instance `status` du but (le but est actif/inactif),
-dans la stratégie de contrôle MEA, qui va "faire en sorte" que les buts soient considérés en profondeur d'abord (si plusieurs buts sont satisfiables, on choisira de satisfaire le plus récemment crée).

4.5.2. Les règles ont trois rôles

De plus les objets buts sont gérés au même niveau que les objets du domaine. Les règles ont ainsi un lourde tâche tripartite :

gérer les actions du singe.
Ce sont les règles de satisfaction de but avec effets de bord.

gérer la hiérarchie des buts/sous-buts.
Ce sont les règles de génération de sous buts.

gérer la représentation assertionnelle des buts.
Ce sont les règles de satisfaction sans effet de bord.

Nous allons donc nous attacher à dénouer ces imbrications, en donnant aux buts un statut plus légitime.

5. Deuxième version

5.1. 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 `GoalMonkey` 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[2].
Nous donnons ici un bref rappel de l'usage des assertions en NéOpus. Se reporter à [Pachet 2] pour plus de détails.

5.2. Les assertions, le finalState, les stopCondition

5.2.1. La fonction de contrôle

Notre idée est que les objets `MonkeyGoal` constituent des évaluateurs, au sens de [Pachet 2], 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`:

```
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.

5.2.2. La fonction assertionnelle

[2]Notons que Michèle Vialatte dans [Vialatte] 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.

Nous allons représenter la fonction assertionnelle des buts en utilisant l'implémentation NéOpus de la notion d'assertion [Pachet 2]. Rappelons brièvement cette notion.

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).

Usage des assertions
Les assertions ne sont pas implémentées directement dans le langage 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.

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 que (la méthode exemple `generalOnLadder`) consistera en la création d'un `MonkeyGoal`, dont la `stopCondition` est une assertion représentant l'état souhaité, soit dans notre exemple, `{unSinge isHolding: bananas}`.
Cela se traduira en :

```
self executeWithAssertion: {unSinge isHolding: bananas}.
```

au lieu de la création de l'objet `MonkeyGoal`, comme dans l'exemple précédent :

```
GoalMonkey new status: #active; type: #hold; objet: bananas.
```

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 reprends 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....
```

5.3. Nouvelle interprétation des règles

A partir du moment ou les objets buts 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 se faisant au niveau méta.
Les méta-règles sont chargées deux fonctions :

Déterminer parmi les action 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 (Cf l'héritage de bases de règles [Pachet 4]), 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.

5.4. Exemples

Nous reprenons ici nos trois règles exemples, dans la nouvelle architecture :

5.4.1. 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)}
```

5.4.2. 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. Dummy 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].
```

5.4.3. une méta-règle de génération de buts

```
holdObjectCeilingAtObj
   | MonkeyGoal b. Dummy s o . Ladder l|

      b status = #loop.
      b ditQue: {s isHolding: o}.
      o weight = #light.
      o isOn: #ceiling.
```

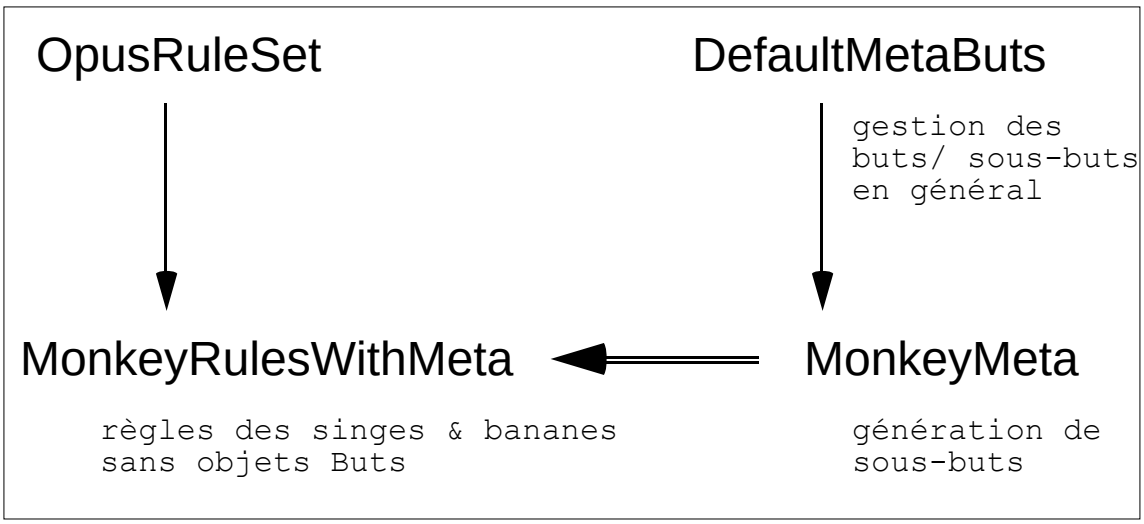
```
      l isOn: #floor.
      l isNotAt: o at.

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 crée implicitement par le biais du message d'évaluation `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}
```

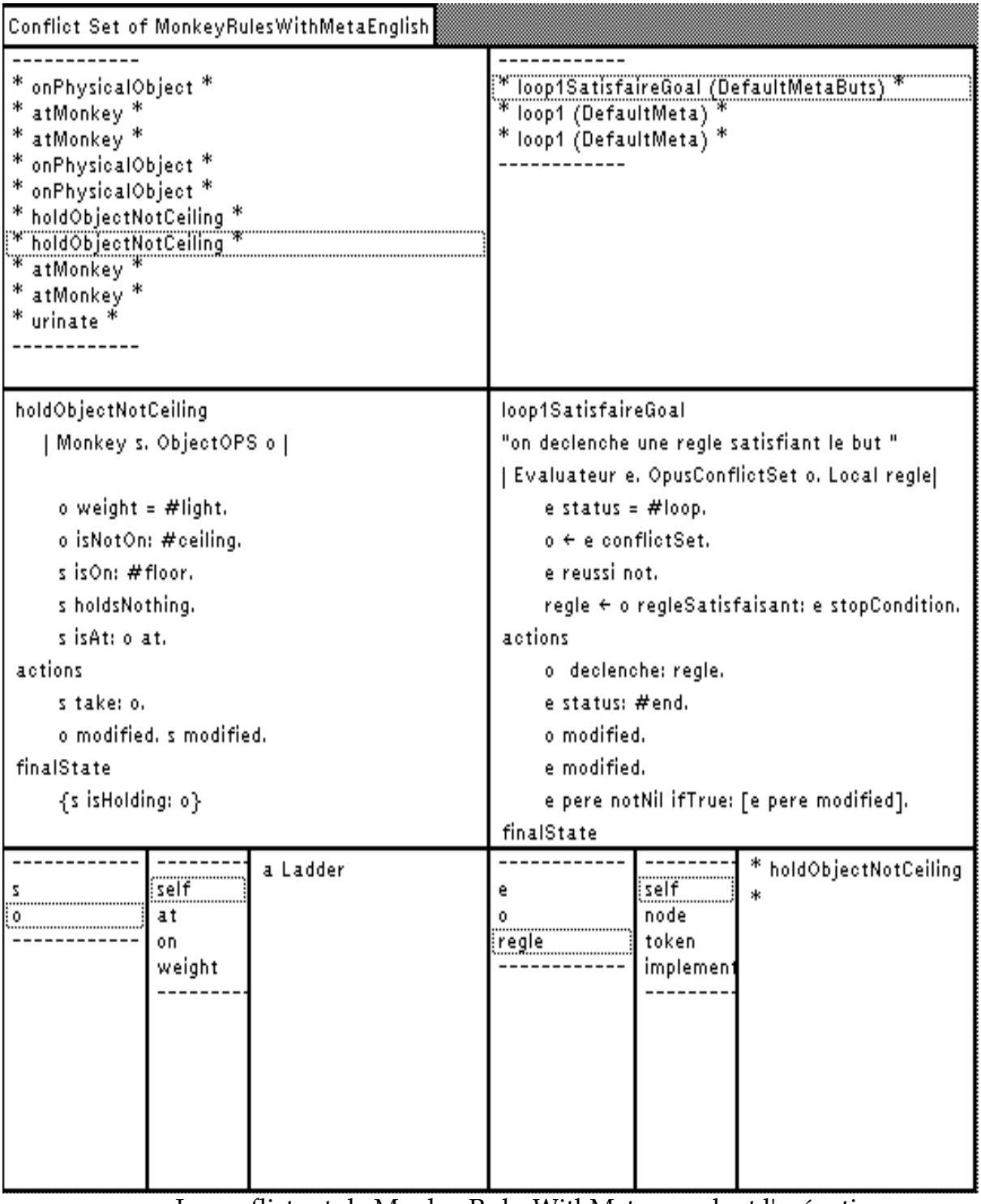


Nouveau schéma de contrôle.

On définit alors les mêmes méthodes de lancement que précédement aux créations de buts près.

Etat du conflict set pendant l'exécution

Lors de l'exécution, le nombre de règles déclenchables est à priori beaucoup plus grand que dans la première version, puisque toutes les actions potentielles du singe y sont représentées.
Ici, on voit que 10 règles sont déclenchables, correspondant à 10 actions possibles, indépendamment de tout but le singe peut prendre l'échelle, prendre la Couch, grimper sur les objets avoisinants ...). La méta-règle `loop1SatisfaireGoal` va cependant choisir celle qui satisfait une `stopCondition` d'un évaluateur, ici l'action de prendre l'échelle.



Le conflict set de MonkeyRulesWithMeta, pendant l'exécution

6. Conclusion

Ces deux exemples complets de base de règles en NéOpus ont un caractère essentiellement démonstratif. Dans la première version on met en oeuvre toute la technologie objet de Smalltalk appliquée à l'écritre de règles. Dans la deuxième

on donne au contrôle un statut de première classe en utilisant l'architecture de contrôle déclarative, ainsi que l'héritage de bases de règles.

7. Annexes

7.1. Annexe 1 : La base de règles complète (première version) :

```
OpusRuleSet subbase: #MonkeyRulesEnglish
      globalObjects: ''
      category: 'Monkey-rules'!
MonkeyRulesEnglish comment:
'Le singe et les bananes, transposition directe de la version en OPS5'


!MonkeyRulesEnglish methodsFor: 'at'!

atMonkey
"si le but est que le singe soit qq part et que le singe est ailleurs ne
tenant rien alors le singe y va"
   | MonkeyGoal b.  Monkey s|

      b isActive.
      b type = #at.
      b objet isNil.
      s isNotAt: b to.
      s holdsNothing.

actions
      s goTo: b to.
      b becomeSatisfied.
      s modified.
      b modified.!

atMonkeyObject
"si le but est que le singe soit qq part et que le singe est ailleurs tenant
qq chose alors le singe y va amenant ce qu'il tient"
   | MonkeyGoal b.  Monkey s. ObjectOPS o|

      b isActive.
      b type = #at.
      b objet isNil.
      s isNotAt: b to.
      s isOn: #floor.
      s isHolding: o.

actions
      s bring: o to: b to.
      b becomeSatisfied.
      b modified. s modified. o modified.!

atMonkeyOn
"si le but est que le singe soit qq part et que le singe est ailleurs pas a
terre alors genere un sous but d'etre a terre"
   | MonkeyGoal b.  Monkey s|

      b isActive.
      b type = #at.
      b objet isNil.
```

```
      s isNotAt: b to.
      s isNotOn: #floor.

actions
      |(MonkeyGoal new status: #active; type: #on; objet: #floor) go

atMonkeySatisfied
"si le but est que le singe soit qq part et que le singe y est alors
satisfaction"
   | MonkeyGoal b.  Monkey s|

      b isActive.
      b type = #at.
      b objet isNil.
      s isAt: b to.

actions
      Transcript show: 'le singe est deja a ', b to printString;cr.
      b becomeSatisfied.
      b modified!

atObject
"si le but est d'amener un objet qq part et que le singe tient l'objet
ailleurs a terre, alors le singe amene l'objet a cet endroit"
   | 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.!

atObjectOnFloor
"si le but est que le singe amene un objet qq part et que le singe n'est pas a
terre mais tient l'objet alors generer un but d'etre a terre"
   | MonkeyGoal b. Monkey s. ObjectOPS o|

      b isActive.
      b type = #at.
      o _ b objet.
      s isAt: b to.
      s isHolding: o.
      s isNotOn: #floor.

actions
      (MonkeyGoal new status: #active; type: #on; objet: #floor) go

atObjectOnHolds
"si le but est que le singe amene un objet qq part et que l'objet est ailleurs
et le singe ne le tint pas alors genere un but  de tenir l'objet"
   | MonkeyGoal b. Monkey s. ObjectOPS o|

      b isActive.
      b type = #at.
```

```
      o _ b objet.
      o weight = #light.
      o isNotAt: b to.
      s isNotHolding: o.

actions
      (MonkeyGoal new status: #active; type: #hold; objet: o) go

atObjectsatisfied
"si le but est que le singe amene un objet qq part et que l'objet y est deja
alors satisfaction"
   | MonkeyGoal b.  ObjectOPS o|

      b isActive.
      b type = #at.
      o _ b objet.
      o weight = #light.
      o isAt: b to.

actions
      Transcript show: 'l''objet', o printString,' est deja a ', b to
printString;cr.
      b becomeSatisfied.
      b modified! !

!MonkeyRulesEnglish methodsFor: 'hold'!

holdNil
"si le but est que le singe ne tienne rien et que le singe tient qqchose alors
le singe lache"
   | MonkeyGoal b. Monkey s |

      b isActive.
      b type = #hold.
      b objet isNil.
      s holdsSomething.

actions
      | heldObject|
      heldObject _ s holds.
      s drop.
      b becomeSatisfied.
      b modified.
      s modified.
      heldObject modified.!

holdNilSatisfied
"si le but est que le singe ne tienne rien et que le singe ne tient rien alors
satisfaction"
   | 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.!
```

```
holdObjectCeiling
"si le but est que le singe tienne un objet qui est au plafond et que
l'echelle est sous l'objet a terre et que le singe est sur l'echelle ne tenant
rien alors le singe attrape l'objet"
   | MonkeyGoal b. Monkey s. ObjectOPS o . Ladder l|

      b isActive.
      b type = #hold.
      o _ b objet.
      o weight = #light.
      o isOn: #ceiling.
      l isOn: #floor.
      l isAt: o at.
      s isOn:  l.
      s holdsNothing.

actions
      s take: o.
      b becomeSatisfied.
      b modified. o modified. s modified.!

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. Monkey s. 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
      |b2|
      b2 _ MonkeyGoal new status: #active; type: #at; objet: l; to: o at.
      b2 go!

holdObjectCeilingOn
"si le but est que le singe tienne un objet qui est au plafond et que
l'echelle est sous l'objet a terre et que le singe n'est pas sur l'echelle
alors generer un sous but d'etre sur l'echelle"
   | MonkeyGoal b. Monkey s. ObjectOPS o . Ladder l|

      b isActive.
      b type = #hold.
      o _ b objet.
      o weight = #light.
      o isOn:  #ceiling.
      l isOn: #floor.
      l isAt: o at.
      s isNotOn:  l.

actions
      (MonkeyGoal new status: #active; type: #on; objet: l) go

holdObjectHolds
```

```
"si le but est que le singe tienne un objet et que le singe est au meme
endroit tenant autre chose alors generer un but de ne tenir rien"
   | MonkeyGoal b. Monkey s. ObjectOPS o |

      b isActive.
      b type = #hold.
      o _ b objet.
      o weight = #light.
      s isAt: o at.
      s holdsSomething.
      s isNotHolding: o.

actions
      (MonkeyGoal new status: #active; type: #hold; objet: nil) go

holdObjectHoldsSatisfied
"si le but est que le singe tienne un objet et que le singe le tient deja
alors satisfaction"
   | MonkeyGoal b. Monkey s. ObjectOPS o |

      b isActive.
      b type = #hold.
      o _ b objet.
      o weight = #light.
      s isAt: o at.
      s isHolding: o.

actions
      Transcript show: 'le singe tiens deja cet objet';cr.
      b becomeSatisfied.
      b modified!

holdObjectNotCeiling
"si le but est que le singe tienne un objet qui n'est pas au plafond et que le
singe est au meme endroit a terre ne tenant rien , et qu'il n'y a pas d'objet
a tenir (?) alors lacher"
   | 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.!

holdObjectNotCeilingAtMonkey
"si le but est que le singe tienne un objet qui n'est pas au plafond et que le
singe est n'est pas au meme endroit alors generer un but d'etre au meme
endroit"
   | MonkeyGoal b. Monkey s. ObjectOPS o |

      b isActive.
      b type = #hold.
```

```
      o _ b objet.
      o weight = #light.
      o isNotOn: #ceiling.
      s isNotAt: o at.

actions
      (MonkeyGoal new status: #active; type: #at; objet: nil; to: o at) go

holdObjectNotCeilingOn
"si le but est que le singe tienne un objet qui n'est pas au plafond et que le
singe est au meme endroit mais pas a terre alors generer un but d'etre a
terre"
   | MonkeyGoal b. Monkey s. ObjectOPS o |

      b isActive.
      b type = #hold.
      o _ b objet.
      o weight = #light.
      o isNotOn: #ceiling.
      s isNotOn:  #floor.
      s isAt: o at.

actions
      (MonkeyGoal new status: #active; type: #on; objet: #floor) go

!MonkeyRulesEnglish methodsFor: 'climb'!

onPhysicalObject
"si le but est que le singe soit sur un objet, et que le singe est au meme
endroit, tenant rien, mais pas sur l'objet alors le singe grimpe sur l'objet"
   | MonkeyGoal b. Monkey s |

      b isActive.
      b type = #on.
      b objet ~= #floor.
      b objet isOn: #floor.
      s isAt: b objet at.
      s holdsNothing.
      s isNotOn: b objet.

actions
      s climbOn: b objet.
      b becomeSatisfied.
      b modified.
      s modified!

onPhysicalObjectAtMonkey
"si le but est que le singe soit sur un objet, et que le singe est ailleurs a
terre, alors generer un sous but d'etre au meme endroit"
   | MonkeyGoal b. Monkey s |

      b isActive.
      b type = #on.
      b objet ~= #floor.
      b objet isOn: #floor.
      s isNotAt: b objet at.

actions
```

```
      (MonkeyGoal new status: #active; type: #at; objet: nil; to: b objet at)
go.

onPhysicalObjectHold
"si le but est que le singe soit sur un objet, et que le singe est au meme
endroit, tenant quelquechose, generer un sous but de ne tenir rien"
   | MonkeyGoal b. Monkey s |

      b isActive.
      b type = #on.
      b objet ~= #floor.
      b objet isOn: #floor.
      s isAt: b objet at.
      s holdsSomething.

actions
      (MonkeyGoal new status: #active; type: #hold; objet: nil) go.

onPhysicalObjectSatisfied
"si le but est que le singe soit sur un objet, et que le singe y est alors
satisfaction"
   | MonkeyGoal b. Monkey s |

      b isActive.
      b type = #on.
      b objet ~= #floor.
      b objet isOn: #floor.
      s isAt: b objet at.
      s isOn: b objet.

actions
      Transcript show: 'le singe est deja la';cr.
      b becomeSatisfied.
      b modified.

!MonkeyRulesEnglish methodsFor: 'on'!

onFloor
"si le but est que le singe soit a terre, et que le singe n'y est pas, alors
le singe saute a terre"
   | MonkeyGoal b. Monkey s |

      b isActive.
      b type = #on.
      b objet = #floor.
      s isNotOn: #floor.

actions
      s jumpOn: #floor.
      b becomeSatisfied.
      b modified.
      s modified!

onFloorSatisfied
"si le but est que le singe soit a terre, et que le singe y est pas, alors
satisfaction"
   | MonkeyGoal b. Monkey s |

      b isActive.
      b type = #on.
```

```
      b objet = #floor.
      s isOn: #floor.

actions
      Transcript show: 'le singe est deja a terre';cr.
      b becomeSatisfied.
      b modified.

!MonkeyRulesEnglish methodsFor: 'termination'!

congratulations
"tous les buts ont ete atteints. Felicitations"
   | MonkeyGoal b |
      b isSatisfied.
      NOT | MonkeyGoal b2 | b2 isActive.
actions
      Transcript show: 'bravo tous les buts ont ete atteints';cr!

impossible
"un but n'a pas pu etre atteint"
  | MonkeyGoal b |
      b isActive.
actions
      Transcript show: 'impossible, le but',
      b printString, 'ne peut etre atteint';cr!
```

7.2. Annexe 2 : La base de règles complète (deuxième version)

7.2.1. La base de règles

```
OpusRuleSet subbase: #MonkeyRulesWithMetaEnglish
      globalObjects: ''
      category: 'Monkey-rules'!


!MonkeyRulesWithMetaEnglish methodsFor: 'at'!

atMonkey
   | Monkey s. ObjectOPS o|

      s isNotAt: o at.
      s holdsNothing.

actions
      s goTo: o at.
      s modified.
finalState
      {s isAt: o at}!

atMonkeyObject
   | Monkey s. ObjectOPS o o2|

      s isNotAt: o2 at.
      s isOn: #floor.
      s isHolding: o.

actions
      s bring: o to:  o2 at.
```

```
      s modified.  o modified.
finalState
      {s isAt: o2 at}!

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)}! !

!MonkeyRulesWithMetaEnglish methodsFor: 'hold'!

holdNil
   |  Monkey s |

      s holdsSomething.

actions
      | heldObject|
      heldObject _ s holds.
      s drop.
      s modified.
      heldObject modified.
finalState
      {s holdsNothing}!

holdObjectCeiling
   | Monkey s. ObjectOPS o . Ladder l|

      o weight = #light.
      o isOn: #ceiling.
      l isOn: #floor.
      l isAt: o at.
      s isOn:  l.
      s holdsNothing.

actions
      s take: o.
      o modified. s modified.
finalState
      {s isHolding: o}!

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}! !

!MonkeyRulesWithMetaEnglish methodsFor: 'climb'!

onPhysicalObject
   | Monkey s. ObjectOPS o |

      s holdsNothing.
      (o = #floor) not.
      o on = #floor.
      s isAt: o at.
      s isNotOn: o.
actions
      s climbOn: o.
      s modified
finalState
      {s isOn: o}! !

!MonkeyRulesWithMetaEnglish methodsFor: 'urinate'!

urinate
   |  Monkey s |

      s exists.

actions
      Transcript show: 'le singe urine ';cr.

finalState
      {s hasUrinated }! !

!MonkeyRulesWithMetaEnglish methodsFor: 'on'!

onFloor
   |  Monkey s |

      s isNotOn: #floor.

actions
      s jumpOn: #floor.
      s modified
finalState
      {s isOn: #floor }! !
```

7.2.2. la base de méta-règles

```
DefaultMetaButs subbase: #MonkeyMetaEnglish
      globalObjects: ''
      category: 'Monkey-rules'!


DefaultMetaButs subbase: #MonkeyMetaEnglish
      globalObjects: ''
      category: 'Monkey-rules'!


!MonkeyMetaEnglish methodsFor: 'on'!
```

```
atMonkeyOn
   | MonkeyGoal b. Dummy s o|

      b status = #loop.
      b ditQue: {s isAt:  o at}.
      s isNotAt: o at.
      s isNotOn: #floor.

actions
      |b2|
      Transcript show: 'genere un but d etre a terre';cr.
      b2 _ b newSon.
      b2 stopCondition: {s isOn: #floor}.
      b2 go! !

!MonkeyMetaEnglish methodsFor: 'at'!

atObjectOnFloor
   | MonkeyGoal b. Dummy s o o2|

      b status = #loop.
      b ditQue: {(s isAt: o at) & (s isHolding: o2)}.
      s isAt: o at.
      s isHolding: o2.
      s isNotOn: #floor.

actions
      |b2|
      Transcript show: 'genere un but d etre a terre';cr.
      b2 _ b newSon.
      b2 stopCondition: {s isOn: #floor}.
      b2 go!

atObjectOnHolds
   | MonkeyGoal b. Dummy s o o2|

      b status = #loop.
      b ditQue: {(s isAt: o at) & (s isHolding: o2)}.
      o weight = #light.
      s isNotAt: o at.
      s isNotHolding: o2.

actions
      |b2|
      Transcript show: 'genere un but de tenir ',o printString;cr.
      b2 _ b newSon.
      b2 stopCondition: {s isHolding: o2}.
      b2 go! !

!MonkeyMetaEnglish methodsFor: 'hold'!

holdObjectCeilingAtObj
   | MonkeyGoal b. Dummy 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!

holdObjectCeilingOn
   | MonkeyGoal b. Dummy s o . Ladder l|

      b status = #loop.
      b ditQue: {s isHolding: o}.
      o weight = #light.
      o isOn: #ceiling.
      l isOn: #floor.
      l isAt: o at.
      s isNotOn:  l.

actions
      |b2|
      Transcript show: 'genere un but d'' etre sur echelle';cr.
      b2 _ b newSon.
      b2 stopCondition:
            {s isOn: l }.
      b2 go!

holdObjectHolds
   | MonkeyGoal b. Dummy s  o|

      b status = #loop.
      b ditQue: {s isHolding: o}.
      o weight = #light.
      s isAt: o at.
      s holdsSomething.
      s isNotHolding: o.

actions
      |b2|
      Transcript show: 'genere un but de lacher ',s heldObject printString;cr.
      b2 _ b newSon.
      b2 stopCondition:
            {s holdsNothing}.
      b2 go!

holdObjectNotCeilingAtMonkey
   | MonkeyGoal b. Dummy s o |

      b status = #loop.
      b ditQue: {s isHolding: o}.
      o weight = #light.
      o isNotOn: #ceiling.
      s isNotAt: o at.

actions
      |b2|
      Transcript show: 'genere un but de se deplacer ';cr.
```

```
      b2 _ b newSon.
      b2 stopCondition:
            {s isAt: o at}.
      b2 go!

holdObjectNotCeilingOn
   | MonkeyGoal b. Dummy s o |

      b status = #loop.
      b ditQue: {s isHolding: o}.
      o weight = #light.
      o isNotOn: #ceiling.
      s isNotOn:  #floor.
      s isAt: o at.

actions
      |b2|
      Transcript show: 'genere un but d etre a terre';cr.
      b2 _ b newSon.
      b2 stopCondition: {s isOn: #floor}.
      b2 go!

onPhysicalObjectAtMonkey
   | MonkeyGoal b. Dummy s o |

      b status = #loop.
      b ditQue: {s isOn: o}.
      o ~= #floor.
      o isOn: #floor.
      s isNotAt: o at.

actions
      |b2|
      Transcript show: 'genere un but de se deplacer';cr.
      b2 _ b newSon.
      b2 stopCondition: {s isAt: o at}.
      b2 go!

onPhysicalObjectHold
   | MonkeyGoal b. Dummy s o |

      b status = #loop.
      b ditQue: {s isOn: o}.
      o ~= #floor.
      s isAt: o at.
      s holdsSomething.

actions
      |b2|
      Transcript show: 'genere un but de lacher ',s heldObject printString;cr.
      b2 _ b newSon.
      b2 stopCondition: {s holdsNothing}.
      b2 go! !

!MonkeyMetaEnglish class methodsFor: 'evaluateurs'!

requiredEvaluatorClass
      ^MonkeyGoal
```

8. Références

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

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

Pachet F. (1)
NéOpus mode d'emploi. Rapport LAFORIA n° 14/91, Paris 1991.

Pachet F. (2)
Du bon usage des méta-règles en NéOpus. Rapport LAFORIA n°16/91, Paris 1991.

Pachet F. (3)
Mixing Rules and Objects : An experiment in the world of Euclidean Geometry. ISCIS V, Turkey 1990.

Pachet F. (4)
Reasoning with objects : the NéOpus environment, East EurOOpe, Bratislava, Septembre 91. Rapport LAFORIA n°13/91, Juillet 91.

Vialatte M.
Description et applications du moteur d'inférence Snark. Thèse de troisième cycle, Paris 6, 1985.