1

Reasoning with objects : the NéOpus environment

François Pachet

LAFORIA, Institut Blaise Pascal, Paris

Address : Boite 169, Université Paris VI, Tour 46-00, 4, Place Jussieu 75252, Paris Cedex 05
Phone: 33.1.44.27.70.10
Fax : 33.1.44.27.62.86
e-mail: fdp@laforia.ibp.fr

abstract

In this paper, we present an overview of the reasoning capacities of the NéOpus system, a forward chaining first-order inference engine, integrated in the Smalltalk environment. NéOpus is a complete rewriting of the Opus system described in [Atkinson&Laursen], which itself is a translation of the OPS5 rule-based system in Smalltalk.
We describe here some major improvements that were added to NéOpus in order to make it operational in a wide range of contexts.
We start by a brief description of the original Opus system, and then describe our extensions, such as the use of class inheritance in variable typing, local variables, the introduction of zero-order variables, rule base inheritance, and a declarative architecture for control. We conclude on the operational use of NéOpus as a knowledge representation tool.

Reasoning with objects : the NéOpus environment

François Pachet

I. Introduction

The aim of our work is to explore the reasoning capacities of Object-oriented languages to represent deductive knowledge. Smalltalk offers a ideal environment to explore those capacities, because it provides a very natural and clear representation for domain concepts. However the inferential capacities of Smalltalk are intrinsically limited (inheritance, in a way may be condidered as a basic inferential mechanism). Adding an extra deductive layer on top of Smalltalk is thus a very natural effort. Some systems were designed, that added a deductive mechanism on top of Smalltalk, such as constraints in ThingLab [Borning], a zero-order inference engine in Humble [Piersol], an OPS5-like rule-based architecture in Essaim [Alizon&Huet], or a prolog-like mechanism in [Lalonde].
The Opus system [Atkinson&Laursen] is the only one to propose a full-fledged first-order (handling variables), forward chaining inference engine that is totally integrated in the Smalltalk environment.
We took the description of this system by the authors as a starting point of our investigations in Knowledge Representation. Our system, called NéOpus is a complete rewriting of the Opus system, based on the original description by the authors. We made a number of experiments in various contexts which led us to a series of improvements that greatly extend its representation capacities.
We will start by a brief description of the original Opus system, and then describe four major developments that where not completely treated in the original system, namely use of inheritance in variable typing, zero-order reasoning, rule base inheritance, and meta rules. We finally describe the programming environment that supports those extensions, and conclude on some issues on rule-based representation raised by the use of NéOpus.

II. The Opus environment

motivations

The Opus system was originally described in [Atkinson&Laursen]. It consists in a first-order inference engine written in Smalltalk-80. The engine is an adaptation of the famous OPS5 [Brownston] inference engine to the Smalltalk universe.

The original objectives were to provide a production system with *maximum integration* of Smalltalk in the rule language. This objective led to a rule syntax that allows writing of arbitrary Smalltalk expressions. There is no particular rule language : the knowledge is expressed directly in terms of messages understood by the various domain objects. We rewrote the system by adding an other reusability objective. We wanted the system to be as opened as possible, to provide a testbed for experimenting with various inference techniques. The system is thus designed to be easily *extended*.

the rule syntax

Instead of having an ad'hoc representation for facts, Opus facts are any Smalltalk objects.
Each rule has a *variable declaration part*, in which the variables used in the rule are declared, by indicating which class they denote. Rules consist in a name, the variable declaration part, a set of premises, and an action part. Rule premises express constraints on Smalltalk objects, that appear as rule variables. Those constraints may be any Smalltalk expression yielding a boolean result. Action parts can be any Smalltalk expression, possibly modifying the objects that appeared in the premises.

Example

Let us simulate an auction : an auctioneer, with several persons, wanting to buy objects. We first define three Smalltalk classes `Auctioneer`, `Person`, `ObjectInAuction`, with the necessary instance variables (`persons` have `money`, `objectsInAuction` have a `price`, a `name`, `Auctioneers` have a set of `objectsToSell`, a `currentObject`, a `state` that defines the various steps of an auction ..).
For instance, here is a rule that makes the auctioneer select the person with the best proposed price for the current object.

```
bestPrice

|Person p. ObjectInAuction o. Auctioneer a|

    a hasStartedAuction.
    o == a currentObject.
    (p   proposedPriceForObject:   anObject)   >   a
bestPrice.

action
    a bestPrice: p proposedPrice.
    a modified.
```

modification of objects

Since action part of rules may be arbitrary Smalltalk expressions, it is impossible to know automatically which objects are actually modified after a rule has been triggered. A partially satisfactory solution consists in explicitely stating which objects have been modified in the action part, by sending them the message `modified`. Of course, only those objects, whose modification may change the instantiation state of a rule have to be sent this message.

2. Architecture

Rules are organized in rule bases. Rule bases are implemented as Smalltalk classes, which are subclasses of a root class `OpusRuleSet`. Having rule bases being Smalltalk classes is interesting because it provides "for free" all the Smalltalk functionalities for classes, as `fileIn`, `fileOut`, and cross-references.
Rules are represented as instance methods for those classes. A special parser parses the code of the rules in order to compile them in the Rete network.

For instance, a rule base named `AuctionRules` may be defined, as a subclass of class `OpusRuleSet`:

```
OpusRuleSet subbase: #AuctionRules
    classVariables: ''
    category: 'OPUS-rules'
```

The preceding rule `bestPrice` would then be defined as an instance method of `AuctionRules`.

If instance methods are interpreted as Opus rules, class methods are interpreted as standard Smalltalk methods. There are used to implement example methods, that may create initial instances to be matched by the rules. A set of methods are implemented to activate the rule base, such as `execute` (the normal default activation method).

Here, an example method could be :

```
!AuctionRules class methodsFor: 'example'!

example
"creation of instances"
p1 := Buyer new; name: #Joe; money: 10000.
p2 := ...
o1 := ObjectInAuction new ...
o2 := ...
a := Auctioneer new; ..
self execute
```

3. Rete compilation

Rules are compiled in a Rete network [Forgy]. The main idea of the Rete compilation is to associate a Smalltalk method to every premise and to the conclusion part of an Opus rule. Those methods are compiled in a separate class, called dynamic class, which is uniquely associated to each rule base.
Then Rete nodes are created for every premise of a rule, and a particular Rete node for its conclusion part. The network is used at activation time by propagating tokens, which represent sets of objects matching the corresponding premise of the rule. The tokens are sent initially to input nodes. When a token reaches a terminal node, the corresponding rule is added to a conflict set, and is ready to fire.

Those networks are implemented by a instanciating class `OpusNetwork`. Rete nodes are represented by instances of root class OpusNode, or of its subclasses. A number of subclasses of OpusNode were initially designed : nodes implementing positive premises (the standard Opus premise), negative premises (testing the absence of objects satisfying a given expression), and terminal nodes (implementing the action part of the rule). As we will see below, this taxonomy was largely extended in order to represent the various extensions to the rule language.
For debugging purposes, Rete networks may be visualized and animated during execution time (see figure 5).

4. Applications

We entirely rewrote the system according to our evolving needs and specifications, and added several interesting features, such as the use of inheritance in variable typing, the use of local variables in rules, the ability of mixing 0-order variable in rules, an inheritance mechanism for rule bases, a declarative architecture for control, and a supporting programming environment.
Our system, called NéOpus, is used in several real-world applications, such as :

- a system thats performs geometrical reasoning [Pachet 1],
- a system that plays openings in Bridge [Alvarez],
- a system that control ventilator for patients after surgery,
- a systems that control medical image analysis,
- a system that automatically generate a graphical interface using a Smalltalk tool box,
- a system that transforms semantic networks into relational networks according to certain transformation rules,
- a scenario generator.

Besides those applications, the system is used as a basis for teaching Knowledge Representation with Objects in a PhD cursus at the University of Paris VI.

III. Extensions

We describe here our developments to the basic Opus system, that constitute the NéOpus system. More informations on the implementation and motivations of those may be found in [Pachet 1,4].

III.1. Use of inheritance in variable typing

Rule variable typing

The standard Opus system intrinsically offers first-order inference capacities. The variables appearing in rules are interpreted "universally" (i.e. a variable x declared as being of class C, is interpreted as "any instance of class C").
In the original system, rule variables represented any instance of a given category. Category where arbitrary sets of objects, in which any Smalltalk objects could be enrolled, modified or removed. We did not implement the notion of category because we prefered the direct relationship between rule variables and Smalltalk classes. Also, the notion of category is not bidirectional : (an object may belong to several categories), which strongly limits the use of object dependency in rules.
In NéOpus, rule variables are declared by specifying their Smalltalk class. This leads to a decision regarding Smalltalk inheritance. If a rule variable x is declared as being of class C, do we restrict the objects denoted by x to all instances of C or to all sub instances (instances of subclasses of C) ?
We introduced the notion of variable typing, in order to give the opportunity of choosing for each rule base, between those two alternatives.
Simple typing is a typing where rule variables denote only direct instances of their class, whereas *natural typing* considers also all sub instances.

Use of typing

Natural typing is a very powerful extension to Opus, since it allows more objects to be concerned by rules. However, no computation is made on the basis of this typing. A very natural feeling is that rules filtering objects higher in a hierachy of classes are more general than rules filtering "lower" objects. The notion of preferability here is to be handled at the activation time, when the system has to chose a rule for firing.
For instance, the rule with a variable declaration such as : `| Object o|`, may be considered more general than the rule with variable declaration `| Person p|`, since `Person` is a sub class of `Object`.
But if several variables are declared, no decision *a priori* can be made, solely on the basis of the rule variable declaration. Unlike method combination (as in CLOS), no coherent rule combination can be imagined.

III.2. O-order reasoning

motivation

First-order reasoning is far more powerful than zero-order reasoning, because if provides a factorisation of code (the same rule may be triggered with several objects). More over, first order reasoning is very natural in Smalltalk, since Smalltalk objects themselves are not named, and are easily accessed by pattern matching.
However, in a number of cases, the notion of *singular object* (a particular, known, existing object) is very useful. For instance, one could not imagine Smalltalk classes with no name !.
This notion requires a naming system that may handle such singular objects. Smalltalk offers two ways of accessing objects by their name : global variables (reserved for system administration purposes) and class variables (which are common to a set of subclasses and their instances).
Using global variables in rules is of course possible. For instance, the conclusion of a rule may be to display a message in the Transcript window. Simply writing "`Transcript show: 'a message' `" in a rule's action part will work. The problem is that those variables are not considered by the system as rule variables, and thus their modification is not taken into account (see above).
In order to allows the effective use of named variables in rules, a new type of rule variable was designed. The idea is to use the class variables of the rule base as named objects. Those objects may be used as standard class variables in Smalltalk methods, and may be used in rules, by declaring them with the key word "`Global`". Their modifications in rules is taken into account by the system.

Example

In our preceding auction example, it is clear that `persons` and `objectsInAuction` are well represented by standard first-order variables. It is awkward to represent the Auctionneer by such a variable, since he is unique in this context (of course several instances of class `Auctioneer` may physically exist, but only one of them interests us). We will represent it by a named (or class) variable.
This leads to a new definition of our `AuctionRules` rule base :

9

```
OpusRuleSet subbase: #AuctionRules
    classVariables: 'TheAuctioneer'
    category: 'OPUS-rules'
```

Here is how our rule is now written :

bestPrice

```
|   Person aPerson. ObjectInAuction anObject.
    Global TheAuctioneer|

 TheAuctioneer hasStartedAuction.
 anObject == TheAuctioneer currentObject.
 (p  proposedPriceForObject:  anObject)  >  TheAuctioneer
bestPrice.

action
 TheAuctioneer bestPrice: p proposedPrice.
 TheAuctioneer modified.
```

Implementation

The implementation of named variables consists in redefining the parser so that it creates a special kind of premise for those premises which contains named variables.
For those premises, a subclass of Rete node is created, called `ReteNodeWithNamedVariable`, that will handle the appropriate propagation of tokens.

Some basic access methods are also defined at the rule base level, to handle the modification and initialization of named variables.

III.3. Dummy variables

Motivations

The fact that Opus premises may be arbitrary Smalltalk expressions has some drawbacks that do not exist when only attribute/value premises are allowed. Let us consider for instance a premise testing that the country of origin of a person' father's car is Europeean, and that its color is red :

cars
```
    | Person aPerson|

    aPerson father car countryOfOrigin = #Europe.
    aPerson father car color = #red.
    ...
```

This syntax leads to heavy premises. It would be very natural to factorize the person's father's car in a "local" variable (in the Smalltalk sense).
We introduced such local variables in rules. They are declared by the keyword `Dummy`, and are interpreted accordingly : a premise containing a dummy variable is not expected to return a *boolean* result, but a *non nil* result. If the result of the assignment is nil then the premise is considered false, else is it considered true.
Our preceding rule **cars** now becomes :

cars
```
    | Person aPerson . Dummy aCar|

    aCar := aPerson father car.
    aCar countryOfOrigin = #Europe.
    aCar color = #red.
    ...
```

Local variables are only syntactic. In particular, they may not be declared as modified, since they are not filtered in the Opus sense.
The implementation of these variables consists in creating a special subclass of

`ReteNode`, called `ReteNodeDummy`, for which the premise test is changed, and for which and no propagation is done for the dummy variable.

Dummy triggering variables

A more subtile variation on dummy variables is the dummy triggering variable. The idea is that opus rules do not take into account the natural dependency of objects. If we take the preceding auction rule `bestPrice` for instance, it is clear that the second premise is somewhat awkward. The object `anObject` is functionnaly accessible from the object `TheAuctioneer`. It would be more natural to write an assignement instead of a test :

```
    anObject := TheAuctioneer currentObject.
```

However, `anObject` may not be here declared as a dummy variable, since its modification has to be taken into account by the system.
Dummy triggering variable are created to cope with those situations, when objects are functionnally accessible, but must be handled as full rule variable as far as propagation is concerned.
Those variable are declared the same way as standard rule variables, but may be assigned in a premise, to allow the above expressions.
The rule `bestPrice` may now be written (with a dummy variable `price`, a dummy triggering variable `anObject`, and a namedvariable `TheAuctioneer`):

bestPrice

```
|Person aPerson. ObjectInAuction anObject. Global
TheAuctioneer. Dummy price|

 Auctioneer hasStartedAuction.
 anObject := Auctioneer currentObject.
 price := p proposedPriceForObject: anObject.
 price > TheAuctioneer bestPrice.

action
 TheAuctioneer bestPrice: price.
 TheAuctioneer modified.
```

As for dummy variables, the implementation of dummy triggering variable consists in adding the ad'hoc parsing functionality, that creates a particular subclass of ReteNode, here `ReteNodeWithTriggerDummy`. This class redefines the propagation methods, in order to take into account the dependency of the assigned variable.

III.4. Rule base inheritance

Since rule bases are classes, the idea of using the inheritance mechanism of Smalltalk is natural. Inheritance of rule bases can be used as a useful means of organizing hierarchically set of rules.

Implementation

The implementation of this mechanism is not straightforward. Object-Oriented languages usually separate static inheritance (decided at compilation time) used for instance variables, from dynamic inheritance (decided at execution time) used for methods.
Rules being closer to methods than to instance variables, dynamic inheritance comes first to mind. But the compilation of a rule leads to two different kinds of compilations : the compilation of the rule in the Rete network, and the compilation of the methods implementing the various premises and the action part in the dynamic class.
This leads to a combination of static (for Rete networks) and dynamic (for dynamic classes) inheritance. Since dynamic class inheritance is parallel to rule base inheritance (see diagram below), the inheritance of the methods implementing the rule in the dynamic class is the standard Smalltalk (dynamic inheritance). However, because of the nature of rule triggering (unlike methods, rules are not looked up), the updating of a Rete network is propagated down to the inheritance tree.
If we suppose a rule base RB, and a sub base of RB called RB2, compiling a rule in RB will result in the compilation of Smalltalk methods in the dynamic class of RB (but not in the dynamic class of RB2), and in the updating of both RB's Rete network, and RB2's Rete network (cf figure 1).

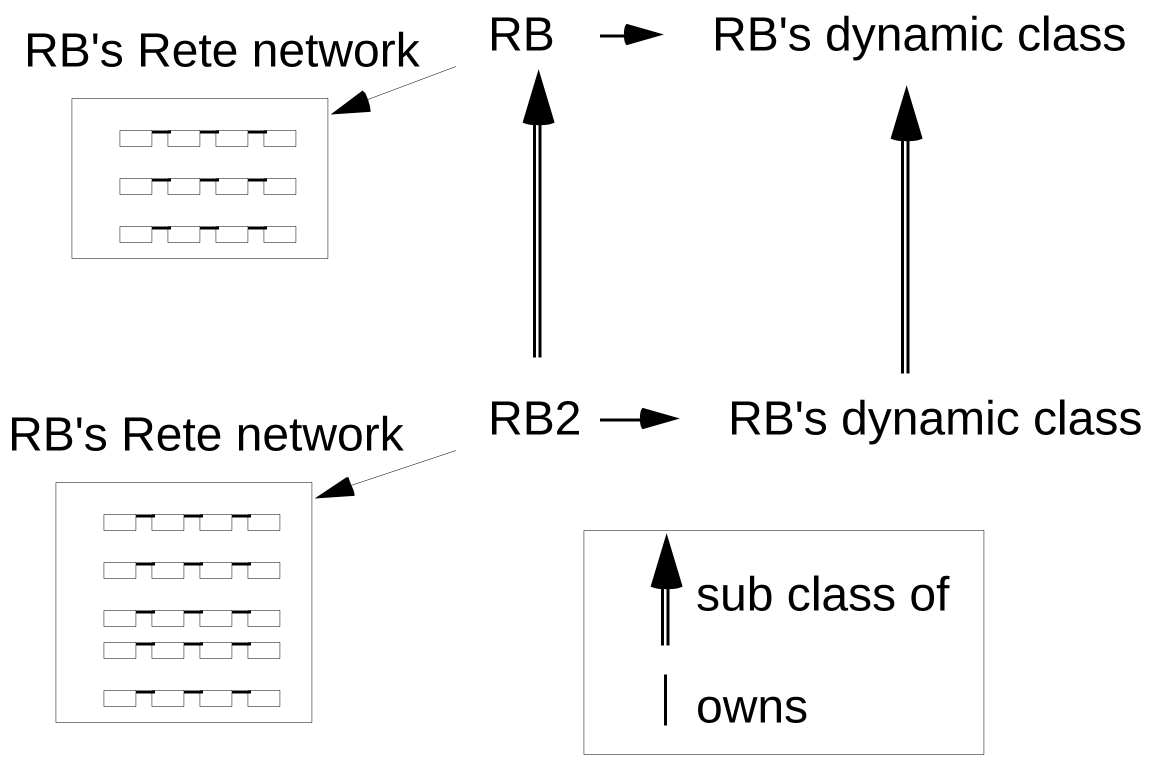


figure 1. The rule base inheritance scheme

Use of rule base inheritance

Like variable typing, rule base inheritance introduces a notion of generality in rules. Rules defined at a higher level in the rule base hierarchy are more general than rules defined lower in the hierarchy. Deciding that this criteria was clear enough, we chose it as a default sorting criteria for rules : the default conflict resolution strategy sorts rules according to the rule base in which the rule is implemented.

III.5. meta rules

The idea of controlling rule bases by a particular rule base, instead of having a procedural control is also a very natural extension of Opus because of the basic syllogism : A) Opus rule bases, conflict sets, rules ... <u>are</u> Smalltalk objects, and B) Opus rules may filter <u>any</u> Smalltalk object.
A declarative architecture for control is set up in NéOpus, that allows rule bases to be controlled by other (meta) rule bases. The idea is to substitute entirely the activation of a rule base, by the activation of a meta base that will

handle all the steps of the activation process, including the conflict resolution. In order to do so, a reification of control is performed, by introducing a particular class, called Evaluator, that represent the evaluation state of a rule base. Details on this architecture may be found in [Pachet 3].

A series of standard meta bases implement standard conflict resolution strategies, (such as LEX and MEA from OPS5), and less standard strategies (strategies based on agenda managements, strategies handling priorities). Specialized meta bases are defined by subclassing one of the standard meta base, using rule base inheritance.

The association of a meta base for a rule base can be done by Smalltalk method, or by a simple selection in the NéOpus dashboard. This allows for instance to trace a rule base activation, without recompiling any code, simply by selecting a tracing meta base in the browser.

IV. The programming environment

The NéOpus system reaches a level of complexity such that powerful interface tools are essential. The programming environment of NéOpus allows the efficient use of the system's functionalities by providing a series of interface tools such as a dashboard, rule browsers, conflict sets views, Rete network views, instance browsers.

A dashboard allows, for each rule base in the environment, to browse the rules, to browse the instances concerned by the rules, to visualize the Rete network, the conflict set, to associate a meta base, to activate metaclass methods (usually implenting examples), to select a step mode, and various rule base parameters such as variable typing, or trace mode.

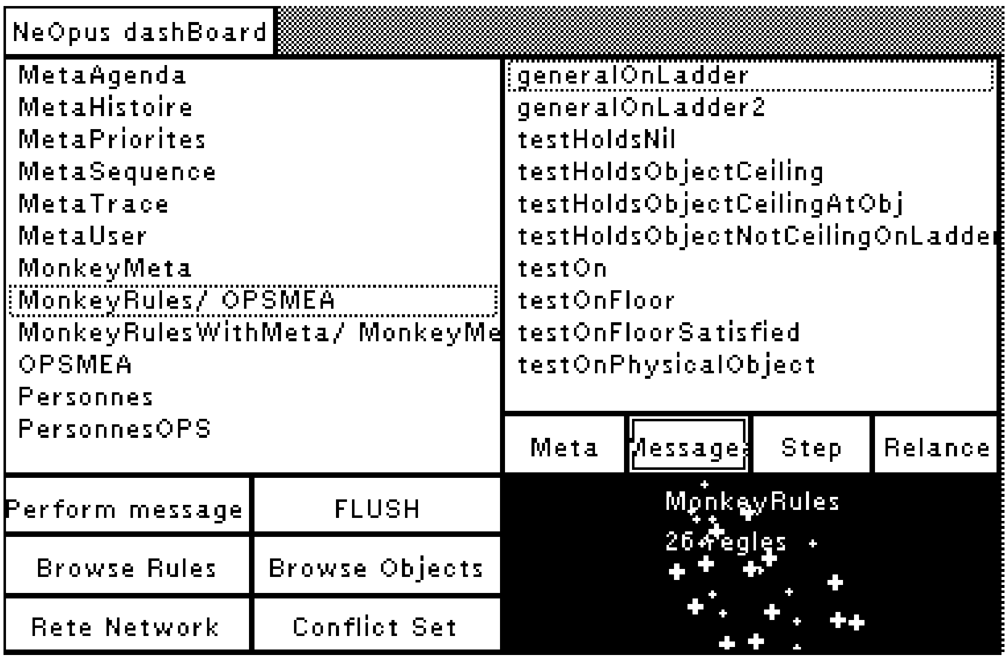


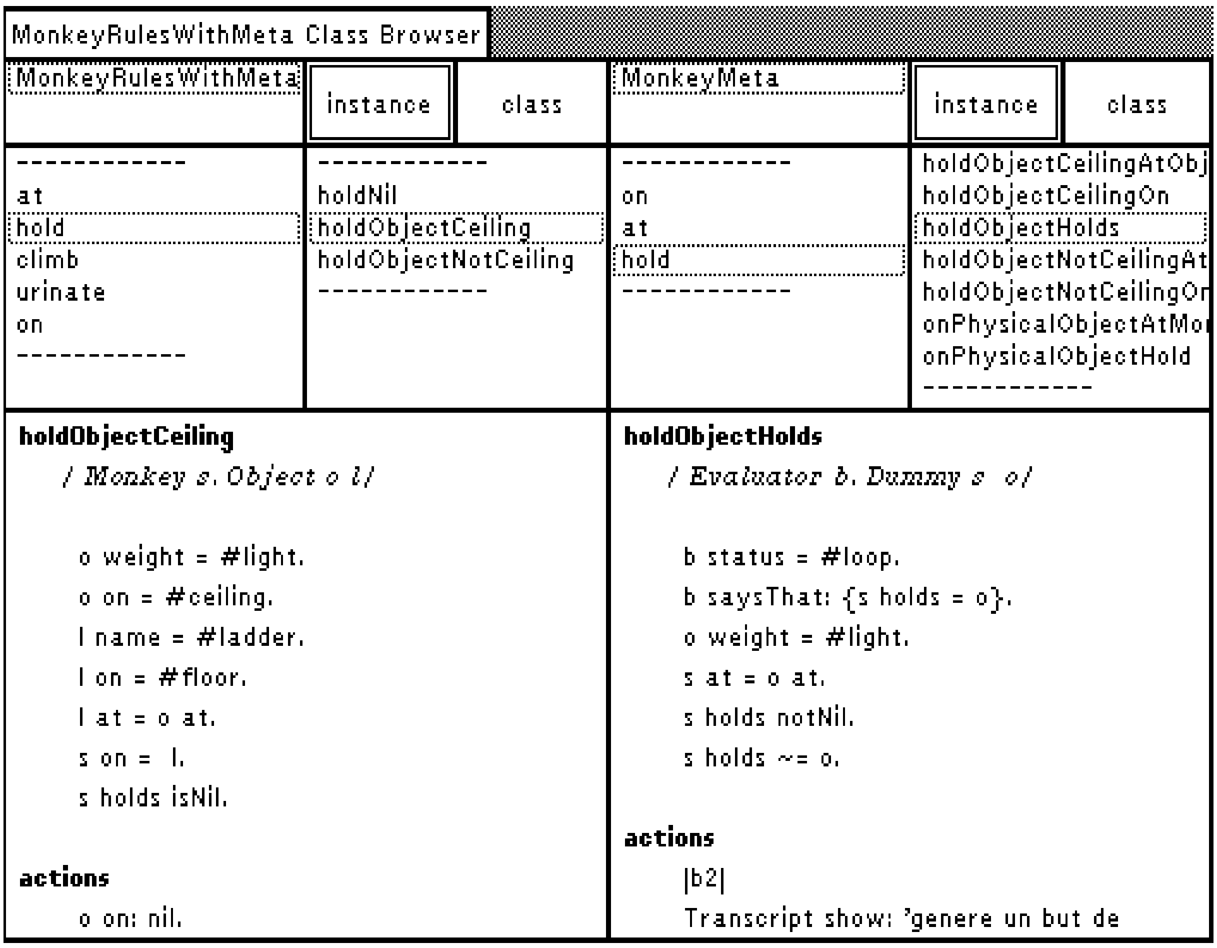
figure2. the NéOpus dashboard

multi level programming

Rule browsers and conflict set views accomodate for meta control, by offering multiple views, one for each rule base in the meta hierachy.

Here we illustrate the interface with a rule base that implement the Monkey and banana example using meta rules. A rule base called `MonkeyRulesWithMeta` is controlled by a meta-rule base called `MonkeyMeta` (itself subbase of a meta rule base implementing a depth-first goal satisfaction strategy). The browser for `MonkeyRulesWithMeta` is thus a "double" browser, in which both rule bases may be browsed in parallel. Similarily, the conflict set view of `MonkeyRulesWithMeta` is double, to allow, at execution time the visualization of control.

Rete network views (figure 5) provide a graphical representation of the Rete network. Each node is accessible via a local menu, allowing the inspecting of the node, and of the associated Opus rule. At execution time, if the view is opened, an animation visualizes the flow of tokens in the network. A scroll/zoom facility is also included that helps managing big networks.

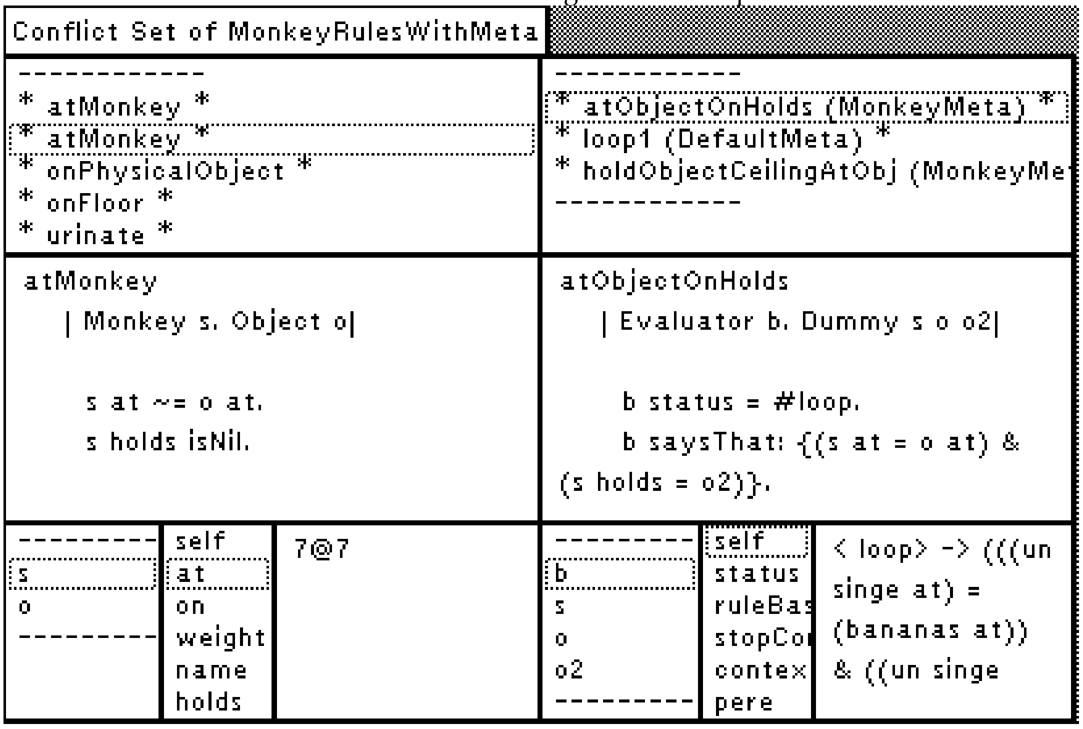


```
MonkeyRulesWithMeta Class Browser

MonkeyRulesWithMeta | instance | class || MonkeyMeta | instance | class

------------        | ------------              | ------------ | holdObjectCeilingAtObj
at                  | holdNil                   | on           | holdObjectCeilingOn
hold                | holdObjectCeiling         | at           | holdObjectHolds
climb               | holdObjectNotCeiling      | hold         | holdObjectNotCeilingAt
urinate             | ------------              | ------------ | holdObjectNotCeilingOn
on                  |                           |              | onPhysicalObjectAtMo
------------        |                           |              | onPhysicalObjectHold
                    |                           |              | ------------
```

```
holdObjectCeiling
    / Monkey s, Object o l/

      o weight = #light.
      o on = #ceiling.
      l name = #ladder.
      l on = #floor.
      l at = o at.
      s on =  l.
      s holds isNil.

actions
      o on: nil.
```

```
holdObjectHolds
    / Evaluator b, Dummy s  o/

      b status = #loop.
      b saysThat: {s holds = o}.
      o weight = #light.
      s at = o at.
      s holds notNil.
      s holds ~= o.

actions
      |b2|
      Transcript show: 'genere un but de
```

figure3. A multiple browser



```
Conflict Set of MonkeyRulesWithMeta

------------                  | ------------
* atMonkey *                  | * atObjectOnHolds (MonkeyMeta) *
* atMonkey *                  | * loop1 (DefaultMeta) *
* onPhysicalObject *          | * holdObjectCeilingAtObj (MonkeyMe
* onFloor *                   | ------------
* urinate *                   |
```

```
atMonkey
   | Monkey s, Object o|

     s at ~= o at.
     s holds isNil.
```

```
atObjectOnHolds
   | Evaluator b, Dummy s o o2|

     b status = #loop.
     b saysThat: {(s at = o at) &
(s holds = o2)}.
```

```
---------  | self   | 7@7
s          | at     |
o          | on     |
---------  | weight |
           | name   |
           | holds  |
```

```
---------  | self    | < loop> -> (((un
b          | status  | singe at) =
s          | ruleBas | (bananas at))
o          | stopCo  | & ((un singe
o2         | contex  |
---------  | pere    |
```

figure4. The conflict set in the middle of the activation

Rete network views provide a graphical representation of the Rete network. Each node is accessible via a local menu, allowing the inspecting of the node, and of the associated Opus rule. At execution time, if the view is opened, an animation visualizes the flow of tokens in the network. A scroll/zoom facility is also included that helps managing big networks.

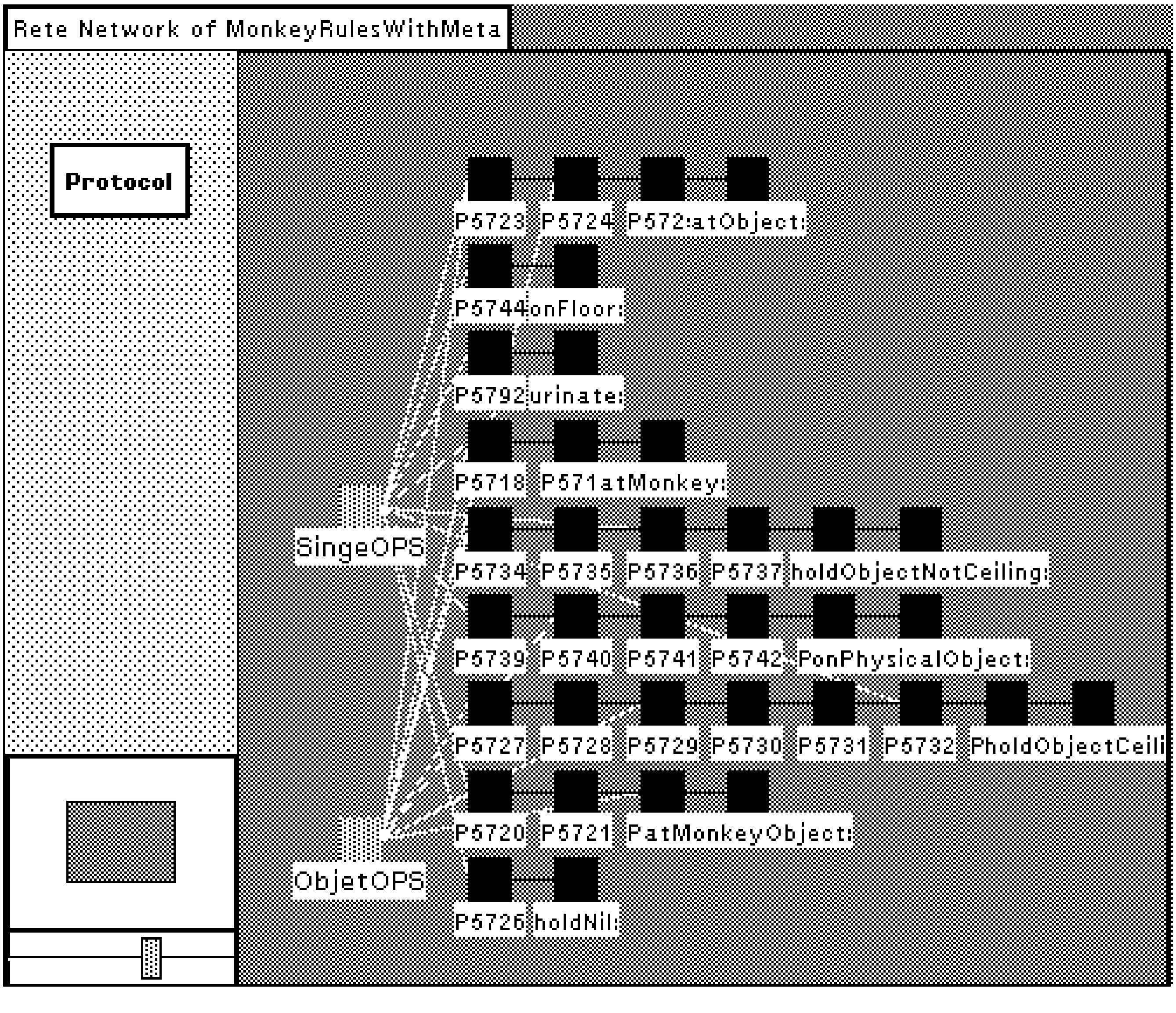


figure5. the view of a Rete network

V. Conclusion

We have shown four major extensions to the original Opus system, that makes NéOpus a powerful representation tool. The combination of several mechanisms, such as natural typing, rule base inheritance, zero-order variables, and meta control is made possible for two reasons. The first one is that all those concepts are the results of two years of experiments with the systems. They are simple and clean concepts that supports cohabitation well. The second is the role of the interface. Meta programming is supported by multiple views that reflects well the hierarchy of meta levels. However, a number of issued are yet to be solved. The fact that there is no rule language supposes that objects hold all the necessry informations to express knowledge about them. This is not always true, and objects often have to be enriched to cope with needs in rule expression [Pachet 1]. Also, the extension of Opus premises to arbitrary Smalltalk expressions yields a series a new problems. For instance, implementing a backtracking or a TMS facility is made impossible by using standard techniques. Lastly, the declarative control architecture requires a description of rules that supports meta reasoning : expressing in terms of objects what a rule is doing on its environment is not an easy task.
Further works on the NéOpus system includes a first-class representation of Smalltalk assertions that will solve a great deal of those problems.

VI. References

Alizon F. Huet G.

Essaim : un environnement de programmation Smalltalk destiné à la construction de systèmes experts. Note technique CNET NT/LAA/SLC/299, 1988.

Alvarez I.

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

Forgy C. L.

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

Borning Alan

ThingLab -- A Constraint-Oriented Simulation Laboratory.
Xerox Palo Alto Research Center. July 79.

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

Lalonde Wilf R.
A novel Rule Base facility for Smalltalk. Proceedings of Ecoop 1987, pp 193-198, Paris.

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

Pachet F. (1)
Mixing Rules and Objects : An Experiment in the World of Euclidean Geometry. ISCIS V, Cappadocia, Turkey 1990.

Pachet F. (2)
NéOpus mode d'emploi. Rapport LAFORIA, Paris 1991.

Pachet F. (3)
Une architecture déclarative de contrôle pour NéOpus. Rapport LAFORIA, Paris 1991.

Pachet F. (4)
PhD thesis. To be published. Laforia, Paris 1991.