NéOpus User's Guide

François Pachet

Laforia-IBP
Université Pierre & Marie Curie

This document is designed to be used in a hyper text manner, e.g. each section should be more or less self-contained, with explicit references to other sections.

Table of contents

The name

Opus was a system designed by Atkinson & Laursen. Néo is a Latin prefix that means "new". Hence NéOpus is a short-cut for néo-Opus, a new and extended version of Opus. Not a very nice name, but I could not come up with anything more euphonic, that would not sound too pretentious.

NéOpus overview

NéOpus is basically an extension of Smalltalk-80 with a first-order forward-chaining rule-based mechanism. There are a number of arguments to support this kind of endeavor. The simplest one is the following:
With OOP you can specify structure and behavior of single objects: this is what classes are all about, isn't it ? When it comes to specifying behavior and structures of *groups of objects,* things are not so easy using OOP. On the other hand, production rules is a good formalism to do this, but this formalism lacks the structuring facilities provided by object-oriented programming languages. So we want to combine both worlds.

The NéOpus system originated from an OOPSLA '87 paper [Atkinson and Laursen 1987] (ParcPlace), that described the "Opus" system, a transposition of OPS5 rules in Smalltalk-80. The NéOpus system is a rewriting of the Opus system that started in 89, and includes a number of original features described below. It has been the core of a number of research projects in Knowledge Representation at LAFORIA (AI lab of Paris 6 University) and other research labs in France and in Canada.
The system's implementation, as well as a discussion of the conceptual hypothesis and methodological issues may be found in my Ph.D. thesis [Pachet 1992a] (in French). See the Documentation section for more details.

Quick description of the system

NéOpus in general

NéOpus adds a facility for writing first order forward-chaining rules to Smalltalk-80. These rules may be seen as "conditional actions" and are expressed as if-then statements.

The main characteristics of the system is its ability to:
- match *any* Smalltalk object in the rules
- use *any* Smalltalk expression in the rules condition as well as action parts.

Practically this means that NéOpus respects encapsulation : writing rules does not require to know the structure of the objects, as in other rule-based systems, where the structure of objects is explicitly manipulated in rules via attributes. Also, no extra "rule language" is introduced: rules are exclusively expressed using Smalltalk messages defined by the classes of the objects that match it. This allows to reduce the so-called "impedance mismatch" that occurs with other rule-based extensions of programming languages: see, e.g. the tumultuous relationship between COOL and C++ in the Clips environment [Donnell 1994].

Rules are compiled in an extended Rete network for efficiency [Forgy 1982]. This means that "partial matches" are computed only once. When an object is created or modified, the Rete network minimizes the number of update operations.

Here is an example of a NéOpus rule. If we suppose the classes `Doctor` and `Patient`, we can write a rule that says that "when the blood pressure of the patient is greater than the maximum blood pressure, then the doctor considers a treatment", as follows:

```
considerTreatmentForHyperTension
      | Doctor d. Patient p|
      p bloodPressure > d maxBloodPressureForPatient: p.
actions
      d considerTreatmentFor: p
```

This rule is to be interpreted as "For any d instance of Doctor, and any p instance of Patient, if "p bloodPressure > d maxBloodPressureForPatient: p" is true then execute "d considerTreatmentFor: p". Of course, the interesting thing to note here is that `bloodPressure`, `maxBloodPressureForPatient:` and `considerTreatmentFor:`, are standard Smalltalk methods implemented in the classes of the corresponding variables, here `Doctor` and `Patient`.

Original features of NéOpus

Besides the standard first-order forward-chaining scheme, a couple of original and powerful features have been added to the basic Rete-based mechanism.

(1) Taking of class inheritance into account in rule variables

In the preceding rules, the expression "For any `x` instanceOf Patient" may mean several things depending on whether class inheritance is taken into account or not. NéOpus gives the possibility of specifying 2 types of typing for rule variables:

- "simple" typing: only the instances of the class are considered or
- "natural" typing: instances of subclasses are also considered.

In our example, this can be useful to have natural typing, and to have subclasses of `Doctor` that redefine method `maxBloodForPatient`: for instance. The same rule may therefore be used with different classes of objects, and yield different results. See section on Typing of Rule Variable for more details.

(2) Rule base inheritance

Rules are written as methods, and rule bases are represented as abstract classes. A novel mechanism has been implemented to transpose the notion of class inheritance to rule bases. This allows structuring rule bases in hierarchies so as to factor common rules, specialize existing rule bases, and redefine rules locally. The rule base inheritance may also be used as a default control strategy where rules defined in the lowest base are preferred in case of conflicts.

(3) Declarative control architecture

Control in NéOpus may be specified using NéOpus itself. In this scheme, rule bases are not activated by the procedural inference engine, but by the activation of other rule bases, called meta-bases. These meta-bases are standard NéOpus rule bases. This allows to use all the functionalities of NéOpus to specify control.

The benefits are :

- complete separation and independence of control knowledge with domain knowledge,

- ability to factor control specification, using rule base inheritance.

(4) Other more technical facilities
These include local variables in rules, possibility of mixing 0-order with first order reasoning, possibility of specifying initial sets of objects to be matched against the rules (context) to avoid combinatorial explosion, and the possibility to retain the current state of instanciation to perform non-continuous inferences.

(5) Facilities for multi-agent programming
Rule bases may be "instanciated" so as to have completely independent yet "identical" rule bases. Each instance has its own set of objects to be matched, initial context, conflict set, control strategies. This is particularly useful in multi-agent environments where several agents may have the same rule base, but with its own set of execution objects and parameters. See section on this subject.

(6) The programming environment for rules and rule bases follows the same spirit than the standard Smalltalk environment for writing methods and classes: rule browsers, hierarchy browsers, cross-references (implementors, senders, messages) for rules, file in/out and so forth. Some tools are specially dedicated to rule-based programming: steppers, conflict set views, graphical conflict set views, instance browsers, and dedicated rule base browsers.

Creating a rule base

A rule base is a Smalltalk class with only one constraint: it must be a subclass of `NeOpusRuleSet`, or of one of its subclasses. So to create a rule base you just have to create a class, as a subclass of `NeOpusRuleSet`, or of an existing rule base. This can be done using a system or class browser, however, it is more convenient to use a NeOpusBrowser: you will get a more specific template. But using a NéOpus browser is not compulsory, the compiler will recognize if the class you are defining is a rule base or a standard class.

The special template for rule bases is just a short-cut for the usual template that 1) suppresses the useless `instanceVariableNames:` and `poolDictionary:` arguments, and 2) renames the `classVariableNames:` argument by `globalObjects:` (see section on Global objects).

You can, of course, also create a rule base dynamically, by evaluating an expression such as:

```
AClass subbase: aSymbol globalObjects: aString
```

The only constraint, once again, is that `AClass` be a descendant of `NeOpusRuleSet`.

Browsing a rule base

A rule base is just a class, so browsing a rule base is done by opening a browser on this class. Standard Smalltalk browsers will do. There are, however, special browsers that add menu options specific to rule bases. You can open these browsers by clicking on the "Browse Rules" option in the Dashboard, after having selected a rule base (see the Interface section). You can also open such a browser by evaluating :

```
NeOpusBrowser open
```

or other creation messages (see class `NeOpusBrowser` which inherits from `Browser`).

Note that when you select a class which is a rule base, the various browsers will automatically propose the "rule" template, instead of the usual "method" template, whenever you are trying to define a new rule.

Writing rules

To write rules, you have to perform the following tasks first:

- select a rule base, or create a new one,
- browse it, using a class browser (any kind of browser will do),
- write the rule, using the NéOpus rule syntax,
- compile the rule, as you would compile a method, i.e. using the *accept* option in the text menu.

Of course, since rules are compiled as methods, you can compile a rule dynamically, as you would for methods, using compiler commands. For instance, evaluating:

```
aRuleBase compile: aRuleString classified: aProtocol
```

will compile the rule `aString` in rule base `aRuleBase`, in protocol `aProtocol`, provided that aString satisfies the rule syntax (see section Rule Syntax for more information).

Rule syntax

Rule syntax is pretty straightforward. Let us take an example of a rule:

```
monkeyEatsBanana
"If a monkey holds a banana, and has goal to get another banana, then
it drops the first banana and take the second one"
      | Monkey m. Banana b1 b2. Goal g |
      m holds = b1.
      g agent = m.
      g action = #eatbanana.
      g objectToGet = b2.
actions
      m drop: b1.
      m take: b2.
      m modified.
```

As you can see, a rule is made of:

- a name, just a Smalltalk identifier, as for unary Smalltalk methods
- a declaration part. This part is for specifying the classes of the objects matched against each rule variable. Each class name (starting with uppercase) is followed by variable names (starting with lowercase). You can have as many class names and variable names as you want.

If you want a BNF syntax for the declaration part, it is:

| ClassName var $\{var\}^*$ {. Class var $\{var\}^*$ $\}^*$ |

- a condition part

The condition part is a collection of Smalltalk Boolean expressions, separated by periods.

- an action part

The action part is any Smalltalk expression. Usually action parts contains particular statements, such as *modified statements* (see section on Modified statement).

Filing in/out rule bases

Well, a rule base is a class, so you can file it in and out exactly like you file in/out a Smalltalk class, i.e. using the various menus available with browsers. The only difference with standard fileIn/out is that NéOpus writes a message in the Transcript each time it compiles a rule, and each time it creates a new rule base.

Rule Base Inheritance

More details can be found in [Pachet 1992b].

Principle

Rule Base Inheritance (RBI in short) allows you to:
- factor common rules to several rule bases, using an inheritance mechanism,
- simplify the control strategy.

The idea is simple: you can create a rule base as a direct subclass of `NeOpusRuleSet`. No inheritance mechanism will take place. You can also create a rule base as a subclass of an existing rule base. In this case, all the rules in the rule base will be inherited, i.e. the newly created rule base will behave exactly as if it contained the inherited rules.

Rule redefinition

Beware, however, of the effect of rule redefinition. Exactly as methods can be overridden, you can override a rule, by writing a rule in a rule base that has the same name as an inherited rule. What happens in this case is that the inherited rule will be removed from the rule base, at compilation time.

Executing a rule base

A rule base is a class (*bis repetitas placent*), hence an object. Executing a rule base will be performed by sending the rule base execution messages, with appropriate arguments.
There are basically three arguments to provide the rule base before executing, with default mechanisms to most usual cases:

- The so-called "context". The context is the set of objects you want the rule base to consider when it is running. You can specify the context by using the context methods such as `addInContext:` (see Context).

- The stop condition. By default, the rule base stops when no rule is fireable. You can provide another stop condition more adapted to your needs. You can specify the stopCondition by using the message `stopCondition: aBlock`.

- The metabase. By default, the control strategy used for choosing rules is fixed (Cf. various subclasses of `NeOpusConflitSet`). For more sophisticated strategies, you can decide that the execution of a rule base will be specified by another rule base. This control rule base is called the meta base.
You can set the metabase of a rule base by sending it the message `metaBase:` with a meta base as argument, or by selecting the *associate metabase* option in the menu in the Dashboard (see meta rules and metabases).

Most standard execution messages

Here are the most useful execution messages.

• Executing a rule base on all existing instances of the environment
It can be useful to consider all existing instances of the environment for the execution of a rule base. To do so, simply send the message `executeWithAllObjects` to the rule base.

For instance, here is a rule base that will flash all existing rectangles of the environment that are bigger than 1@1, and whose top left is not 0.

```
flashAllRectangles
| Rectangle r |
r origin x ~= 0.
r extent > (1@1).
actions
Transcript show: r printString;cr.
Screen default displayShape: r rounded lineWidth: 3 at: (r origin)
for Milliseconds: 20.
```

This rule is written in rule base `FlashRectangleRules`. The rule base is executed by evaluating:

```
FlashRectangleRules  executeWithAllObjects
```

This message, however, can be dangerous since the instances actually living in the environment depend heavily on the garbage collector's behavior. But it can be also a fun way to add knowledge to the Smalltalk environment this way, especially using background processes (see section on Facilities for multi-agent programming).

• Executing a rule base on one single object

aRuleBase executeWithSingleObject: anObject
executes the rule base with anObject as single context. Useful when you write rules talking of only one object.

• General case
In the general case, you have specific objects on which you want the rule base to operate. To do so, you add them in the context of the rule base and then execute it. See for instance the example method for launching the `PersonRules` rule base.

Example methods for rule bases

There is a nice trick in NéOpus with regards to example methods. Since rule bases are classes, they have a metaclass. This metaclass inherits from the metaclass of NeOpusRuleSet, where, e.g. execution message are implemented. But you can use this metaclass to write your own example methods. The Dashboard supports this idea, by allowing you to browse unary class methods for rule bases (using the Message switch), and to send one of these messages to the selected rule base (see Interface section).

Meta rules and metabases

General idea

The control IS the activation of another rule base.
Control objects are introduced to help writing metabases.
More details in [Pachet and Perrot 1994b]

Most common cases

- Tracing rule base activation

A simple metabase, `MetaTrace`, allows you to display the rule firing in the Transcript, without writing any code. To do so, you simply have to select your rule base in the Dashboard, swith to the *Meta* mode, select `MetaTrace` in the right-hand side list, and use the menu option "associate" to link the rule base to MetaTrace. Then execute the rule base as usual, using any kind of execution message, including the example methods using the *Messages* switch.

- Firing rules in sequence.

A special metabase, `MetaAgenda`, allows you to fire rules according to the alphabetical order of the protocols in which they are defined. This is very convenient as you can use browser facilities to re-arrange rule order without any programming.
To use this, simply evaluate:

```
MetaAgenda executeWithProtocols: aRuleBase
```

were `aRuleBase` is the rule base you want to execute.

Interface tools

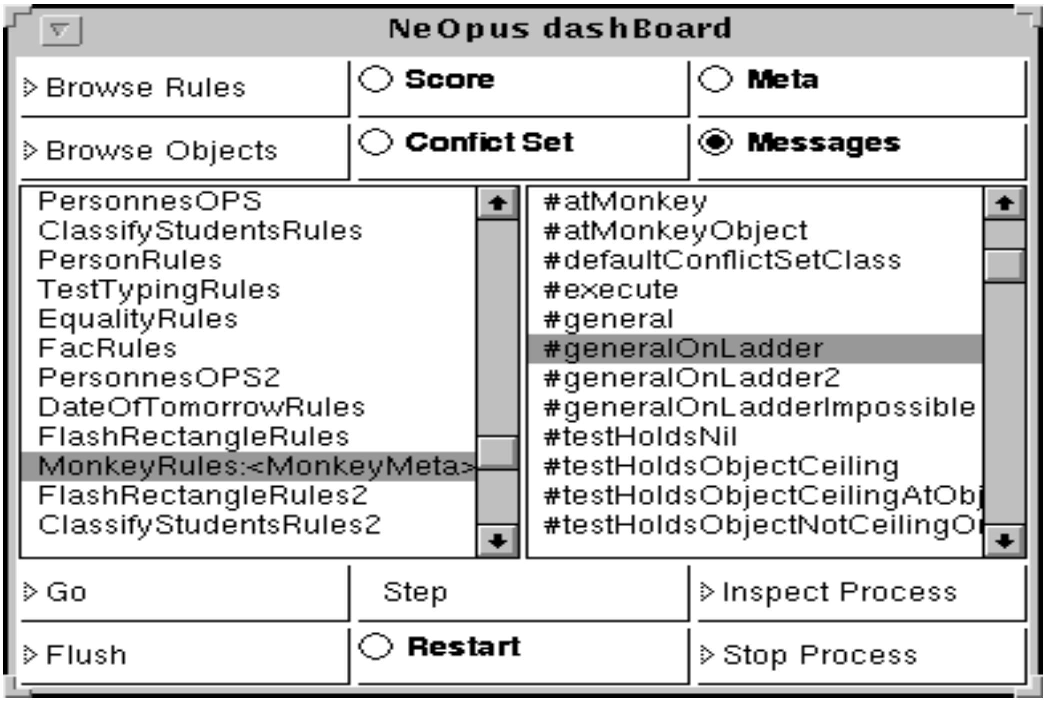
There are a number of interface tools that facilitates the use of NéOpus.

The Dashboard

This is the main NéOpus interface tool. You can open a dashboard by evaluating the expression:

```
NeOpusListenerView open
```

It basically displays the currently loaded rule bases, and allow you to browse and execute them. Various tools are accessible through this interface, as described below.



The options are the following. First the buttons:

- *Browse rules*, open a browser on the currently selected rule base. If none is selected, opens a browser on all classes.
- *Browse objects*, open an instance browser on the classes of the selected rule base. If none is selected, opens an instance browser on all classes (useful in a lot of situations).
- *Score*, opens a score view on the currently selected rule base.
- *Conflict Set*, opens a conflict set view on the currently selected rule base.
- *Meta*, switch to the meta mode, i.e. displays the list of available metabases on the right-hand list. A menu is then available in this list to associate/dissociate the currently selected rule base with a given metabase.
- *Messages*, switches to the message mode, i.e. displays the list of metaclass messages of the currently selected rule base. One can then select a message, and execute it using the "Go" Button.
- *Go*, executes the currently selected rule base. If no message is selected (in the right hand list), then executes the rule base using the default execution method. If a message is selected, then sends the message to the rule base. Particularly useful to test example messages, without typing anything.
- *Step*, put the selected rule base in step mode. Each time a rule becomes fireable, the rule base suspends itself. Process is resumed by clicking on the Restart button.
- *Restart*, Cf. Step.
- *Inspect Process*, opens an inspector on the current process.
- *Stop Process*, kills the current process.

Now there are also various menus available. In the left-hand list, there are two menus, depending on whether a rule base is selected or not. The options are pretty straightforward:

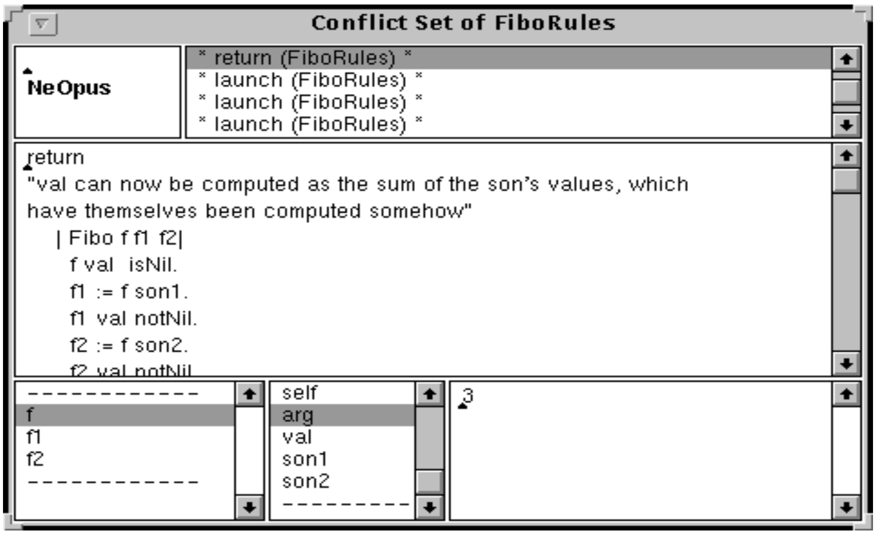
If a rule base is selected, one can browse its dynamic class, release its instances (Cf. rule base instances), set the typing mode to simple or natural (Cf. Typing of rule variables), filter the categories to limit the number of rule bases shown in the list, recompile the rule base, etc. If no rule base is selected, one can create a new rule base, or update the category list (useful when rule bases are removed for instance).
On the right hand side, menus depend on the mode (Meta or Messages). In Message mode on can simply browse the selected message (or all messages if none is selected). In the Meta mode one can associate/dissociate a rule base from a meta base.

The NéOpus Browsers

These browsers are almost exactly like standard browsers, with some extensions in the menus to browse related classes for instance, or reset the Rete network associated to the rule base.

Conflict set View

This interface allows to browse a conflict set during the execution of a rule base. Particularly useful in conjunction with the Step mode.

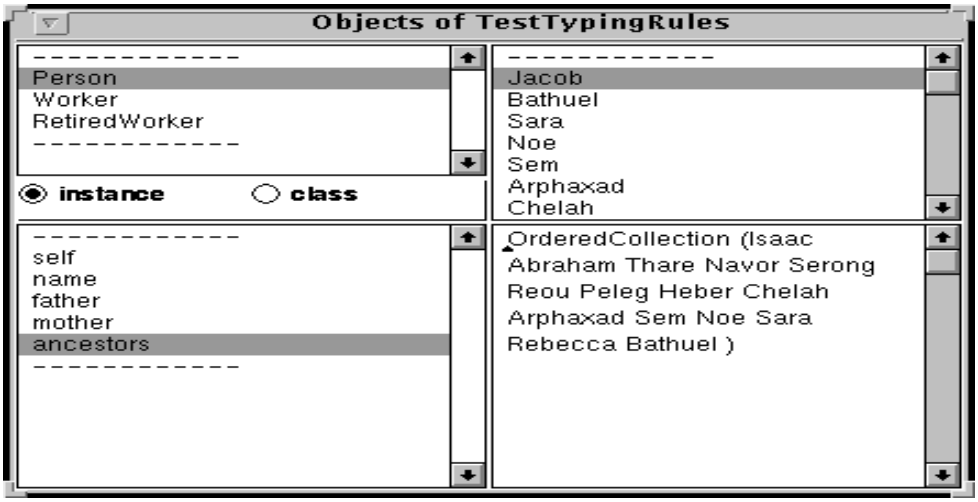


Instance browser

Instance browsers are general purpose tools, that allow to browse instances, instead of classes. It is surprising that these tools do not exist in the standard environment. These browsers look very much like standard class browsers, except for the method and protocol views, replaced by more appropriate inspector views on the instances themselves. You can open various kinds of instance browser by evaluating expressions such as:

- `NeOpusInstanceBrowser open`
opens an instance browser on all instances of the environment.
- `NeOpusInstanceBrowser openOnClasses: aListOfNames,`
on all instances of the given classes.

Options in the menu allow you to expand/contract a given class, to show or hide its subclasses. These browser are also available as goodies (in the uiuc goodies library, under the name InstanceBrowser). Beware of the fact that just clicking on an instance in these browsers prevents the instance from being garbaged.



Score view

This is an experimental idea, from the paper by [Domingue and Eisenstadt 1991]. These views may be opened form the dashboard interface. You can even "play" the trace of execution, by associating a sound to each rule.

A standard example of first-order power: Fibonacci

This example is very representative of the first-order mechanism, typical of Rete, OPS-5 and NéOpus. You can load this example by filing in the files FiboObjects.st and then FiboRules.st.

The idea is to compute the famous Fibonacci value by representing explicitly the tree of computation. Of course, this method is terribly inefficient. The aim here is just to show the recursive effect of first-order forward-chaining programming. It also provides a good benchmark for testing the efficiency of first-order engines.

We first define a class Fibonacci, with attributes:
- `val`: the value to be computed by rules
- `n`: the argument of Fibonacci
- `son1`, `son2`: the two sons created by the recursive computation;

together with simple accessing methods to these instance variables.

We then create a rule base called `FiboRules`, which contains the three following rules:

```
ending
"val can safely be computed as 1 if arg is less than 2"
     | Fibo f  |
      f val isNil.
      f arg  <= 2 .
actions
      f val: 1.  f modified.
```

```
launch
"create the two sons for Fibo if val is greater than 2"
    | Fibo f  |

      f arg > 2.
      f val isNil.
      f son1 isNil.
actions
      f son1: (Fibo new arg: ( f arg - 1)).
      f son2: (Fibo new arg: ( f arg - 2)).
      f son1 go. f son2 go. f modified.

return
"val can now be computed as the sum of the son's values, which
have themselves been computed somehow"
    | Fibo f f1 f2|
      f val  isNil.
      f1 := f son1.
      f1 val notNil.
      f2 := f son2.
      f2 val notNil.
      actions
      f val: ( f1 val + f2 val).   f modified.
```

Note that the same treatment can be applied to other computations of recursive functions (Factorial, towers of Hanoi). You can load the corresponding rule bases, they are very instructive! Remember to first load the file containing the class definition (if it was not filed in before), and THEN file in the file containing the rule base.

Typing of rule variables

This paragraph talks about the relationship between class inheritance and rule variables. The idea is to specify what exactly the rule variable denotes, in relation with subclass hierarchies. In NéOpus, there are two ways of typing rule variables:

- *Simple typing*: a rule variable denotes only direct instances of its definition class,
- *Natural typing*: a rule variable denotes direct as well as indirect instances of its definition class.

The rule base `TestTypingRules` (included the PersonRules.st file) illustrates this mechanism. First we define a class `Person`, then class `Worker` as a subclass of `Person`, and class `RetiredWorker`, subclass of `Worker`. Then we write a simple rule base with only one rule as follows:

```
oneStupidRule
"acknowledges the presence of a group of three people"
    | Person p. Worker w. RetiredWorker r |
      p exists. w exists. r exists.
actions
      Transcript show: 'a group of three people';cr.
```

Then we instantiate these classes, and execute the rule base as follows (Cf. example method in the metaclass of `TestTypingRules`). With natural typing, the rule should be triggered 6 times, with simple typing only once:

```
exampleTyping
      | x y z  |
      x := Person new.  y := Worker new.  z := RetiredWorker new.

      self setNaturalTyping.
      Transcript show: 'With natural typing :';cr.
      self executeWithObjects: (Array with: x with: y with: z).

      self setSimpleTyping.
      Transcript show: 'With simple typing :';cr.
      self executeWithObjects: (Array with: x with: y with: z).
```

The modified statement

NéOpus does not know when an object has been modified by the execution of a rule's action part. Actually the system cannot infer this information because of the very principle of encapsulation. Unfortunately the RETE algorithm needs this information to recompute correctly the instanciation state of the rule base. The only person who knows which objects have been modified is the rule programmer himself. So we ask him to do this job, through modified statement.

The simplest example here is the rule `ending` in rule base `FiboRules`:

```
ending
"val can safely be computed as 1 if arg is less than 2"
    | Fibo f  |
      f val isNil.
      f arg   <= 2 .
actions
      f val: 1.   f modified.
```

The modified statement is necessary for the rules to be retried with the Fibo instances having this Fibo as son.

Note that you can also use *modified* statement outside rules. In this case, they have an argument which is the rule base in which the object is declared modified. For instance:

aFibo modifiedFor: FiboRules

There are actually three such messages you can use in rules:

- `go` (or `goFor`: if used outside rules), to state that an object should be taken into account by the rule base,
- `remove` (`removeFor`:), to state that `anObject` should not be taken into account any longer,
- `modified` (`modifiedFor`:), actually equivalent to `remove` followed by `go`.

The modified statement problem is actually a hard one. If you want more details, you can read the various presentations made at the OOPSLA workshop on EOOPS

[Pachet 1994b]. The solution adopted in NéOpus is definitely a practical one, and does not solve the problem in its generality, which seems to be extremely hard.

Global objects and 0-order reasoning

First order is not always appropriate to talk about objects. Sometimes it is useful to be able to denote a particular, named object within a rule. In NéOpus this is possible, through the - distorted - use of class variables. So-called 0-order objects, or global objects may be efined as class variables for rule bases, and used as 0-order objects in rules. These objects have to be declared in rules as "Global". They can then be used in rules, and declared as modified in action parts.

Negative premisses

The notion of negative condition is described already in OPS-5-like systems. The negation is a logical negation, i.e. gives the possibility of expressing things like "If there is no X such as ...". The conditions have been implemented in NéOpus, but they are very inefficient.

The syntax is the following:

<negative condition> ::== NOT <variable declaration part> <positive condition>.

An example can be found in rule base `classifyStudentRules`, which classify students according to their notes, in one single rule:

```
oneRule
      | Student x y|
      x ~~ y.
      x hasNoteLessThan: y.
      x place <= y place.
      NOT | Student z | (z ~~x & (z~~y)) and: [z hasNoteBetween: x
and: y].
actions
      x isAfter: y.
      x modified
```

The rule base can then be executed as follows:

```
example
"creates a list of students with notes from 0 to 5,  and place them
in reverse order. The rule base will sort them properly"
   | s e |
   s := OrderedCollection new.
   0 to: 5 do: [:i |
            e := Student note: i.
            s add: e.
            self addInContext: e].
   self execute.
```

Rule base instanciation

Sometimes, you may want to run the same rule base several times in parallel, with different contexts of activation. This resembles the notion of "instance" with standard classes. We provide a "emulation" of rule base instances that allows you to do the trick. You can create a pseudo instance of a rule base using the `new` message. This will in fact create a subclass of the rule base, with a computed name, thereby inheriting all the rules of the rule base.
You can create as many instances of rule bases as you want. The only thing to take care of is the fact these instances will not be garbaged, since they are classes (fortunately, the GC does not garbage classes). So you may want to garbage them once in while to avoid memory saturation. This is done by selecting the "Release instances" in the Dashboard menu, or by sending the message `releaseInstances` to the rule base from which you created instances.

This mechanism is particularly usseful in conjunction with the facilities for multi-agent programming (see this section).

Facilities for multi-agent programming

Basic idea

It is very easy to run several rule bases in pseudo parallel by using the Process classes of Smalltalk. The idea is simply to have each rule base running as a separate process. The only thing to take care of is then to be sure that these processes will "yield the way" to each other, to ensure a fair distribution of the processor time. This is very simply done by inserting a "Processor yield" statement in various places in NéOpus. These places are the following:

- in method `trigger:` of class `NeOpusConflictSet`
- in method `contains:dummyLimit:` of class `NeOpusToken`

By default, the "Processor yield" statement is a comment in these methods. If you want to execute rule bases in pseudo-parallel, remove the comments in these two methods. In fact, removing the comment from the first method suffices. Removing it from both will increase the number of Process yielding. Beware, however, that these statements may slow down quite drastically the overall performance of rule bases.

Using rule base instances in parallel

Once created, several rule base instances can be used in parallel, each one having its own context of activation. For instance, here is a script that executes two instances of the same rule base (`HanoiRules`), in pseudo-parallel:

```
!HanoiRule class methodsFor: 'examples'!
twoSimultaneously
r1 := HanoiRules new.
r2 := HanoiRules new.
[r1 example] fork. [r2 example fork].
```

Inside NéOpus

Dynamic classes

Dynamic classes is where rules are compiled into Smalltalk code. You should not even be aware of their existence, but if you really want to know... To each rule base is associated a dynamic class, whose name is the rule base name, appended by "Dynamic". For instance, the dynamic class of `FiboRules` is `FiboRulesDynamic`. The dynamic class is created automatically when the rule base is created.

You can browse dynamic classes using the dashboard (*browse dynamic class* option), the special NéOpus Browser, or the standard browser: they are all packed in the category 'token-classes'.
When a rule is compiled, it is first parsed, and then compiled into several Smalltalk methods: one method for each condition, and one method for the whole action part. The method names for conditions are computed using an incremental index (e.g. P7832, P7833, etc.). The method for the action part has the same name as the rule. Note that there is no risk of confusion here since the class in which they are compiled is not the rule base itself, but the dynamic class, which is not related to the rule base in an inheritance relation.

Parsing

The NéOpus parser is awful for the time being. It uses the Parser compiler from ParcPlace (apok utilities). The awful part is, as usual, the semantic actions. But it works and that's the main thing to ask from a parser.

Rete network

The Rete network in NéOpus is both a simplified and extended version of Forgy's standard Rete network. It is simplified because there is no discrimination network - objects are simply instances of classes, so there is nothing to discriminate - and because there is no factoring of common conditions in the network. It is extended because nodes can handle more than one object.

Library of examples

To start with NéOpus, I would advise browsing through the following simple examples, all in the directory "kb". Note that you can fileIn all these files at once using the filein-kb.st file:

- `FiboRules` and `FacRules`: load FiboObjects.st then FiboRules.st
- `HanoiRules`: load HanoiObjects.st then HanoiRules.st
- `EqualityRules`: load EqualityObjects.st then EqualityRules.st
- `FlashRectangleRules`: load FlashRectangleRules.st
- `PersonRules` (inspired from [Brownston et al. 1986]),
  Load PersonObjects.st then PersonRules.st (includes several rule bases)
- `DateOfTomorrowRules`: load DateObjects.st, then DateRules.st
- Monkey and bananas, standard version, also from [Brownston et al. 1986].
  Load files MonkeyObjects.st, OPSMA.st, and then MonkeyRules1.st.
  The rule base `MonkeyRules` must be executed using metabase `MonkeyMeta`, itself subbase of `OPSMEA`.

List of current applications

The main current application of NéOpus is the NéoGanesh system (noticed the prefix ?). This closed-loop system controls a ventilator in real time, to provide respiratory help for patients in intensive care units. The system is described in details in Michel Dojat' thesis [Dojat 1994]. More compact description emphasize technical problems of a real-world application [Dojat and Pachet 1992b; Dojat and Pachet 1992a]. Several papers emphasize the reusable aspects of the NéoGanesh architecture [Dojat and Pachet 1995b; Dojat and Pachet 1995a; Dojat and Pachet 1995c]. A typical use of rule base inheritance is described for representing temporal reasoning in [Dojat and Sayettat 1994].

The Metagen architecture has been developed and used in Laforia for a few years, to specify model transformations in various domains [Blain et al. 1994; Revault et al. 1994].

MusES is a system for representing common sense knowledge about tonal music [Pachet et al. 1995]. Several rule-based systems were built on top of MusES using NéOpus, such as an automatic analysis system [Mouton and Pachet 1995; Pachet et al. 1995], and a system for simulating musical improvisation [Ramalho and Pachet 1994].

Diapason is a system for teaching diagnosis in electrical installations, with French EDF [Moinard 1994; Joab et al. 1995].

CardExp is an expert system for detecting cardiovascular diseases, developed at university of Montréal, which uses NéOpus [Dufresne et al. 1995].

On smaller scales, NéOpus was used for various local research prototype applications, developed at Laforia for Ph.D. and master thesis: mathematical reasoning [Laublet 1993] [Laublet 1994], meta-modeling [Sahraoui 1995], image analysis [Forte 1991; Forte et al. 1992], explanations [Alvarez 1992], real time operating systems [Benyahia 1994], tutorial systems [Benhouhou 1996], meta-level representations [Fillod 1995], programming environments [Charbonnel 1990], [Lavillonniere 1994].

Methodology for EOOPS

Two attempts have been made to extract methodological guidelines from our experience in using NéOpus. First, the idea that rule-based programming introduces a difference between two categories of objects: objects making up the world as it is *perceived* by the expert, and objects representing the world as it is *conceived*. Rules are then seen as a way of animating the conceived world from configurations of perceived objects, in a Gestalt perspective [Pachet 1994c].
Another work in progress is to use the pattern/framework notions and technology and apply it to EOOPS [Pachet 1995b]. This allows to identify recurrent "patterns" of rules/classes, that tackle well defined small design problems. Typical such patterns are the patterns developed for a framework in temporal representation [Pachet and Dojat 1995].

Documentation

NéOpus is described in details in [Pachet 1994a]. More general descriptions can be found in [Pachet 1990; Pachet 1991c; Pachet and Perrot 1994a; Pachet 1995a]. A tutorial on NéOpus was presented at the TOOLS conference [Perrot 1992]. A user's manual in French is available in [Pachet 1991a]. Metarules are described in [Pachet and Perrot 1994b] and a full example is in [Pachet 1991b]. Rule Base Inheritance is described in [Pachet 1992b]. The issues related to the *modified* statement have been discussed in the workshop on EOOPS [Pachet 1994b; Pachet 1994a]. Attempts at identifying methodological guidelines have been made in [Pachet 1994c].

References

[Alvarez 1992] **Alvarez I.** Explication morphologique: un mode d'explication fondé sur la géométrie. Université Pierre & Marie Curie, Ph.D. thesis, 1992.

[Atkinson and Laursen 1987] **Atkinson R and Laursen J**. *Opus: A Smalltalk Production System*. OOPSLA'87, p. 377-387,1987.

[Benhouhou 1996] **Benhouhou A.** Une base de connaissances événementielle pour le guidage interactif des situations de crise. Université Pierre & Marie Curie, Ph.D. thesis, 1996.

[Benyahia 1994] **Benyahia I.** Automatisation de la circulation des informations en temps réél. Université Pierre & Marie Curie, Ph.D. thesis, 1994.

[Blain et al. 1994] **Blain G, Revault N, Sahraoui H, Perrot J-F**. *A Metamodelization technique*. OOPSLA 94 Workshop on AI and Software Engineering, Portland, 1994.

[Brownston et al. 1986] **Brownston L, Farrel R, Kant E, Martin N**. Programming Expert Systems in OPS5: an introduction to rule-based programming, Addison-Wesley 1986.

[Charbonnel 1990] **Charbonnel S.** Etude et réalisations de l'environnement du générateur de systèmes experts NéOpus. CEMAGREF - Université de Nantes, Masters thesis, 1990.

[Dojat 1994] **Dojat M.** Contribution à la représentation d'expertises dynamiques médicales. Application en réanimation. Université de technologie de Compiègne, Ph.D. thesis, 1994.

[Dojat and Pachet 1992a] **Dojat M and Pachet F**. *NéoGanesh: an extendable Knowledge-Based System for the Control of Mechanical Ventilation*. 14th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, Paris, p. 920-921,1992a.

[Dojat and Pachet 1992b] **Dojat M and Pachet F**. Representation of a medical expertise using the Smalltalk environment: putting a prototype to work. TOOLS 7. G. Heeg, B. Magnusson and B. Meyer. Dortmund (Germany), Prentice Hall**:** 379-389, 1992b.

[Dojat and Pachet 1995a] **Dojat M and Pachet F**. Effective Domain-Dependent Reuse in Medical Knowledge Bases. Computers and Biomedical Research : to appear, 1995a.

[Dojat and Pachet 1995b] **Dojat M and Pachet F**. *Representing Medical Context Using Rule-Based Object-Oriented Programming Techniques*. Artificial Intelligence in Medicine Europe, Pavia (It), Springer-Verlag, p. 423-424,1995b.

[Dojat and Pachet 1995c] **Dojat M and Pachet F**. *Three Compatible Mechanisms for Representing Medical Context Implicitly*. IJCAI Workshop on "Context in Knowledge and Reasoning Modelling", Montréal, 1995c.

[Dojat and Sayettat 1994] **Dojat M and Sayettat C**. *Aggregation and forgetting: two key mechanisms for across-time reasoning in patient monitoring*. AAAI Spring

Symposium on Artificial Intelligence in Medicine: Interpreting clinical data, Stanford (Ca), p. 27-31,1994.
[Domingue and Eisenstadt 1991] **Domingue J and Eisenstadt M**. *A new metaphor for the graphical explanation of forward-chaining rule execution*. IJCAI 91, Sydney (Australia), Morgan-Kaufman, p. 129-134,1991.
[Donnell 1994] **Donnell B**. Object/rule integration in CLIPS. Expert Systems Journal 11 (1): 29-45., 1994.
[Dufresne et al. 1995] **Dufresne A, Gecsei J, Cromp P, Alexe C**. *Cardexp: a graphical and hypermedia interface to a knowledge base to learn decision making.* InterAct'95, 1995.
[Fillod 1995] **Fillod O.** Un système qui apprend à résoudre des problèmes combinatoires en utilisant des métaconnaissances. Université Pierre & Marie Curie, Master thesis, 1995.
[Forgy 1982] **Forgy C L**. Rete: A fast algorithm for the many pattern/ many object pattern match problem. AI 19 : 17-37, 1982.
[Forte 1991] **Forte A-M.** Système basé sur la connaissance pour l'identification, la caractérisation et la mise en correspondance d'entités anatomiques et fonctionnelles en imagerie médicale multimodalité. Université François Rabelais, Tours, Ph.D. thesis, 1991.
[Forte et al. 1992] **Forte A-M, Bernardet M, Lavaire F, Bizais Y**. *Object-Oriented versus Logical Conventional Implementation of a MMIIS*. SPIE'92, Medical Imaging VI, Newport Beach, Ca, 1992.
[Joab et al. 1995] **Joab M, Paumelle I, Delforge B.** Architecture du système Diapason, Technical Report LIF - Université Pierre & Marie Curie, 95-1, 1995.
[Laublet 1993] **Laublet P.** FORREnMat: un système à base de connaissances pour l'étude expérimentale du raisonnement mathématique. Université Pierre & Marie Curie, Ph.D. thesis, 1993.
[Laublet 1994] **Laublet P**. Hybrid Knowledge Representation and Theorem Proving in Mathematics. Artificial Intelligence in Mathematics. J. H. Johnson, S. McKee and A. Vella, Oxford University Press, 1994.
[Lavillonniere 1994] **Lavillonniere V.** Système expert d'aide à la conduite du tracteur et de son outil. CEMAGREF - Université Pierre & Marie Curie, Master thesis, 1994.
[Moinard 1994] **Moinard C**. *Diagral: a diagnosis and repair system*. Fifth International Workshop on Principles of Diagnosis - DX'94, New Paltz, 1994.
[Mouton and Pachet 1995] **Mouton R and Pachet F**. *Numeric vs symbolic controversy in automatic analysis of tonal music*. IJCAI'95 Workshop on Artificial Intelligence and Music, Montréal, p. 32-40,1995.
[Pachet 1990] **Pachet F**. *Mixing Rules and Objects: an Experiment in the World of Euclidean Geometry*. ISCIS V, Nevsehir (Turkey), p. 797 - 805,1990.
[Pachet 1991a] **Pachet F.** NéOpus mode d'emploi, Technical Report Laforia-IBP, 91/14, 1991a.
[Pachet 1991b] **Pachet F.** Pour en finir avec le singe et les bananes, Technical Report Laforia-IBP, 91/15, 1991b.
[Pachet 1991c] **Pachet F**. *Reasoning with objects : the NéOpus environment*. Conférence East EurOOpe, Bratislava, Tchécoslovaquie, p. 72-87,1991c.
[Pachet 1992a] **Pachet F.** Représentation de connaissances par objets et règles : le système NéOpus. Université Pierre et Marie Curie, Ph. D. thesis, 1992a.
[Pachet 1992b] **Pachet F**. *Rule Base Inheritance.* Colloque "Représentations Par Objets", La grande Motte, p. 187-200,1992b.
[Pachet 1994a] **Pachet F.** Proceedings of the OOPSLA'94 Workshop on EOOPS, Technical Report University Pierre & Marie Curie, 94/24, 1994a.

[Pachet 1994b] **Pachet F**. *Report on the OOPSLA'94 Workshop on EOOPS*. Addendum to the OOPSLA'94 proceedings, Portland, Oregon, 1994b.

[Pachet 1994c] **Pachet F**. *Vers un modèle du raisonnement dans les langages à objets*. Colloque "Langages et Modèles à Objets", Grenoble, p. 111-123,1994c.

[Pachet 1995a] **Pachet F**. On the embeddability of production rules in object-oriented languages. Journal of Object-Oriented Programming 8 (4): 19-24, 1995a.

[Pachet 1995b] **Pachet F**. Patterns for EOOPS. Communications of the ACM (submitted) , 1995b.

[Pachet and Dojat 1995] **Pachet F and Dojat M.** Un framework pour la représentation de connaissances temporelles en NéOpus, Technical Report Laforia-IBP, 95/19, 1995.

[Pachet and Perrot 1994a] **Pachet F and Perrot J-F.** Report on the NéOpus system. Workshop OOPSLA'94 sur les EOOPS, Technical Report Laforia-IBP, 94/24, 1994a.

[Pachet and Perrot 1994b] **Pachet F and Perrot J-F**. *Rule Firing with Metarules*. Software Engineering and Knowledge Engineering - SEKE '94, Jurmala, Latvia, p. 322-329,1994b.

[Pachet et al. 1995] **Pachet F, Ramalho G, Carrive J, Cornic G**. *Representing temporal musical objects and reasoning in the MusES system*. International Congress in Music and Artificial Intelligence, Edinburgh, p. 33-48,1995.

[Perrot 1992] **Perrot J F.** Rule-Based Object-Oriented Programming, TOOLS'7 Conference, Tutorial notes 1992.

[Ramalho and Pachet 1994] **Ramalho G and Pachet F**. *From Real Book to Real Jazz Performance*. International Conference on Music Perception and Cognition (ICMPC), Lièges (Belgium), p. 349-350,1994.

[Revault et al. 1994] **Revault N, Sahraoui H, Blain G, Perrot J-F**. *A Metamodeling technique: The Metagen system*. TOOLS Europe 16, Versailles, Prentice-Hall, p. 127-139,1994.

[Sahraoui 1995] **Sahraoui H.** Application de la méta-modélisation à la génération des outils de conception et de mise en oeuvre des bases de données. Université Pierre & Marie Curie, Ph.D. thesis, 1995.