Report on the NéOpus system experience

François Pachet, Jean-François Perrot

LAFORIA
Institut Blaise-Pascal, Boite 169
4, Place Jussieu
75252 Paris Cedex 05, France
E-mails: pachet/jfp@laforia.ibp.fr

Abstract

The NéOpus system integrates production rules with Smalltalk objects in a seamless way, avoiding the so-called "impedance mismatch problem" by proposing a rule formalism which does not constraint the objects and expressions used in condition or action parts. We outline here the basic asumptions underlying the design of the Opus and NéOpus systems, and stress on two original features of NéOpus that proved particularily interesting: rule base inheritance, and the declarative architecture for control.

1. Introduction

In 1987 Atkinson and Laursen showed how first-order, forward-chaining rules could be accommodated in Smalltalk-80 in an intimate and seamless way [2]. Their system, called Opus, can be viewed as OPS-5 revisited from a Smalltalk perspective. Opus compiles rules using an object-oriented realization of Forgy's Rete network. Among its most salient features are:

(1) that rules apply to all Smalltalk objects, thus opening the way to innumerable applications ;

(2) that rules are treated as Smalltalk methods and rule bases as abstract classes, which permits reusing most of the Smalltalk environment;

(3) that fireable rules, hence conflict sets, appear as first-class objects, thereby giving a firm grip on firing control problems.

They suggested several developments, notably a scheme for the inheritance of rule bases deduced from the standard class inheritance mechanism of Smalltalk. The first author reimplemented Opus with several improvements as part of his doctoral research [11], [12], [14], [15]. He carried out the rule base inheritance proposal [13]. A number of experiments were conducted with his system, called NéOpus. We outline here the basic characteristics of the Opus and NéOpus systems, and stress on two original features of NéOpus : rule base inheritance, and the declarative architecture for control.

2. Compilation of rules in Opus and NéOpus

2.1. Rules that apply to any object

The main issue from the point of view of applicability is generality, i.e. the possibility to define rules applicable to objects that have been defined independently, typically in an already existing application. In our view, this is the main achievement of Opus. The NéOpus experience bears out the validity of this approach. Rule-based components have effectively been added to independently designed systems, see e.g. [11], [17].

From the technical point of view, the fact that every object has a well-defined communication interface (defined by its class) can be turned into an advantage by deciding that rules will be expressed entirely with expressions of the underlying language (Smalltalk-80), without specific linguistic constructs. Conditions are expressions with boolean value, action parts are procedure calls (via messages). This freedom of expression, however, has to be paid back with by the "modified" problem (Cf. section 3.2).

2.2. Rule bases as abstract classes

One is naturally tempted to declare that rules will be first-class objects, and rule bases as well. Following Atkinson & Laursen, we take a less naïve approach and stress the predominantly textual nature of rules and rule bases: we treat rule bases as abstract classes, subclasses of class RuleBase. As with all Smalltalk classes, we endow them with object properties and methods defined in their metaclasses, which are subclasses of the metaclass RuleBase class.
Of course, the Smalltalk compilation process is redefined for these classes. To each rule base is associated a Rete network. Each rule compilation in a given rule base will result in an updating of the rule base's network, according to standard Rete policy, i.e. one Rete node per condition. One extra node is also created for the action part of the rule (called a terminal node). The main idea of the Opus compilation is to associate a Smalltalk method to every condition 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, and are associated to the corresponding Rete nodes.

The methods associated with the premisses of a rule will implement the test required for the tokens propagated in the network. The method representing an action part will be associated with so-called terminal nodes, which will be used for representing fireable rules.

<u>3. Rete compilation in NéOpus : main drawbacks and limitations</u>

The freedom of expression of NéOpus has several positive and negative consequences that we outline here.

3.1 The tradeoff between expressivity and efficiency

NéOpus allows maximum expressivity, in order to fully respect encapsulation. Indeed, using objects in rules is interesting only if arbitrary methods may be used to specify the matches, and not only so-called "accessing" methods. In a way, the whole "integration" issue starts when conditions and actions are not limited to instance variable accessing methods.
But the fact that any expression may be used in condition as well as in action parts has to be paid by the programmer, in the form of the modified statement.
For example, here is a simple rule taken from the "Monkey and banana" example [4]. Comments are between double quotes.

```
holdObjectNotCeiling                  "the rule name"
  | Monkey m. PhysicalObject o |      "the variable declaration"
      o weight = #light.              "a condition"
      o isNotOn: #ceiling.            "an other one"
      m isOn: #floor.
      m holdsNothing.
      m isAt: o at.
actions
      m take: o.                      "an action"
      o modified. s modified.         "two modified statements"
```

In the current version of NéOpus, it is the programmer's responsibility to declare which objects have been modified after the action part of the rule is executed (here the monkey AND the objects are modified after the expression "m take: o" is evaluated). In the case of complex methods used in condition or action parts, this modified statement may be complex to write, and sometime even impossible (Cf. the "Petri net example" in [14]).

3.2 Rete efficiencies and inefficiencies

In NéOpus, there is no (and there cannot be) "discrimination" network. The Rete network is limited to the joint network. Like in the standard Rete algorithm, the left memory stores all the tokens that successfully passed the previous conditions. Unlike in the standard Rete algorithm, the right memory stores all the instances of the classes of the "free" variables appearing in the rule under the form of a dictionary (free variable / set of objects).
Since premisses are expressed as arbitrary Smalltalk expressions, it is not possible to compute the exact set of nodes which are actually concerned by the modification of an object. This may of course involve a lot of unnecessary updates of the network when an object is declared as "modified". Accordingly, another classical optimization of Rete, which consists in factoring out shared premisses is impossible in NéOpus, for the same reasons.

This is still an open issue in NéOpus, as well as in most of EOOPS. However, there are two arguments to support our approach:

1) even with our "limited" Rete compilation, the comparison between NéOpus and totally interpreted systems (such as Essaim [1]) with similar knowledge bases ([9]) is in favor of NéOpus for large rule bases.

2) our experience in hybrid rule bases never led us to significant problems with inefficiency. In other words, although the efficiency of NéOpus is clearly not optimal, this does not seem to be a bottleneck for developping large EOOPS rule bases (our largest rule base - it deals with diagnosis and treatment of heart disease - contains 300 rules).

4. Rule base inheritance

Since rule bases are implemented as abstract classes, we have to define a status for the inheritance relation between rule bases. A particular inheritance scheme for rule bases has been developed, called rule base inheritance or RBI (see [13] for details). This mechanism transposes the intuition of inheritance as found in class-based languages, i.e. a restricted specialization mechanism, in the world of rules.
This transposition of class inheritance in the world of rule bases consists simply in redefining the compilation process to mimick method inheritance : when a rule base A is defined as as "subbase" of a rule base B, all the rules of B are added to A. This aggregation is performed at compile and definition time. Each time a rule is compiled in a rule base, the compiler propagates the compilation in the sub bases of the rule base, recursively.
This transposition of inheritance has two interesting effects:

- It gives rule names an important status, relating to the "overriding mechanism". A sub base A of a rule base B may override an inherited rule r, simply by having a rule called r. Once again, this overriding mechanism is realized at compilation time, but the effect is strictly the same as for methods: inherited rules which are overriden are simply ignored.

- The real "intuition" of inheritance is represented by a particular control strategy (called RBI strategy). This hard-wired strategy consists, in case of conflict, in "preferring" a rule in the lowest subbase.

This mechanism proved very useful in two ways:

1) it helps organizing large knowedge bases, by providing a means of reusing rules in a customary fashion.

2) It helps reduce the complexity of control specification. The hierarchy of meta bases takes into account the "static" relations of precedence between rules. To represent

more sophisticated control specification, the declarative architecture (cf next section) is used.

5. A declarative architecture for control

Many systems have emphasized the need for explicit and separate representation of control (see e.g. [5], [6], [7]) or the reflexive aspect of meta level architecture [3]. Following this tradition, we designed a declarative architecture for specifying control that take full account of OO programming and of the various features of NéOpus, including Rule Base Inheritance.
In NéOpus, fireable rules and conflict sets are represented by first-class objects. By adequately defining the behavior of these classes, we can program several control strategies. In our framework, this is operated in a natural way by sending adequate messages to the conflict set as object.
In simple cases, the choice of the rule to be fired is made via a fixed criterion applied to the conflict set itself, e.g. choosing the most constrained rule, or the newest one, etc. In such a case, a method of class ConflictSet does the job. It is then activated in a loop by a method addressing the rule base as object, i.e. defined in the metaclass of class RuleBase or of one of its subclasses. This basic loop (a method called proceduralEvaluate) is easy to define in a procedural manner, since all the pertaining information is accessible from the rule base.

5.1 Control Objects

It is often necessary to base the choice of the rule to be fired on quite elaborate information, such as a history, a trace of rule firings, an agenda, a tree of goals etc. The additional features needed might be added to class ConflictSet (via subclassing). But this would overload it with information that may be quite complex and foreign to its primary role. We propose to formalize this information without interfering with the definition of rule bases and conflict sets, by means of separate entities which we call control objects. This is not a really new idea. in fact, well-known control strategies such as subgoaling make use of specific objects that clearly fall into our category of control objects. We systematize this idea with the full backing of object-oriented programming.

The main idea behind our notion of a control object is to introduce an independent object that will contain all the necessary information pertaining to the control of a rule base. This primary object, called an Evaluator, represents the present state of the reasoning process. This representation can carry more or less details, according to the structural complexity of the evaluators involved. In its most elementary form, an evaluator has two attributes: status and stopCondition. Status takes discrete atomic values (e.g. #start, #loop, #end), whereas stopCondition is a boolean expression (in Smalltalk, a block). This minimum definition is sufficient to represent the standard activation of a rule base, as defined in the procedural architecture. The interesting aspect of this notion is two-fold. Firstly, evaluators being independent objects, they are reusable and application-independent. Secondly, the advantage of having a separate class appears when specifying more complex strategies: class Evaluator is designed to be specialized by subclassing, and the specialization's will not interfere with the rule base or conflict set definitions. More complex control strategies usually require the introduction of new data structures. These structures will be represented by attributes of particular subclasses of Evaluator, following the pure object-oriented style.

For example, managing the notion of rule pack whose sequence is declared in an agenda is now straightforward. The class EvaluatorWithAgenda is introduced as a subclass of Evaluator. It has an additional attribute that contains an instance of class Agenda. Class Agenda is itself defined by, say, a list of rule packs to evaluate sequentially, and an index to the current rule pack. This new class of evaluator will be reusable by all the rule bases requiring this kind of control (see e.g. [8]). The same scenario applies for any control strategy that requires additional structure, such as: managing a history or a trace of rule firing, selecting rules according to priority lists, and so forth.

In many applications, control information has to be somehow integrated in domain knowledge. A typical example is the so-called subgoaling technique presented e.g. by Brownston for the Monkey & Bananas problem [4]. Specific objects called goals are introduced and maintained together with the domain objects. They are organized in a tree-like hierarchy. Most of the rules have at least one condition which deals with a goal, and many rules have conclusions that create or modify goals. In our architecture, the objects necessary for managing subgoaling are naturally represented by adequate subclasses of evaluators. In the Monkey & Bananas problem, a class EvaluatorSubGoaling will be defined, as a subclass of Evaluator, that adds the attributes necessary for managing father/son relationships between goals and their subgoals.

5.2. Talking about a fireable rule : assertions

However, the choice of the rule to be fired may require to know more about the fireable rule than simple syntactic information. The main thing to know about a rule is the consequence of its firing on the simulated world. But this information is not easily accessible, essentially because of the encapsulation principle. Indeed, as we saw in section 2.3, rule action parts are represented either as texts or compiled methods. Neither of these representations is suitable for manipulation by an inference engine. In other words, because attributes are hidden by the communication interface, the system does not know anything about what a rule does before it actually fires it.

Our solution to this problem is to introduce a new syntactic construct that gives the programmer the ability to state the intention of a rule. This construct, called an assertion is basically a representation of a fact about the world (in the logical sense). It is expressed as a Smalltalk expression (between brackets {}) and is manipulated as an instantiated syntactic tree. Each NéOpus rule text has an additional field (the finalState field) that contains such an assertion. When a fireable rule is created, the assertion is instantiated with the objects that match the rule.

For example, here is a rule taken from the M&B rule base. This rule states that if a monkey and a physical object verify a set of conditions, then the monkey takes the object (method take:). The finalState part of the rule declares that in this case, the monkey will hold it, by the assertion {s isHolding: o} :

```
holdObjectNotCeiling
  | Monkey s. PhysicalObject 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}
```

Those assertions may now be used for specifying more elaborate control strategies, involving dynamically created control objects. Since assertions are also boolean expressions, they may be used for representing the stopCondition of evaluators, instead of the former blocks.

5.3. Substitution - regression

We further propose to maintain these control objects by means of a separate set of rules, called control rules or metarules. These rules will deal only with control objects and with the conflict set. They constitute a separate rule base, called metabase, which completely specifies the control strategy.
We propose an architecture for managing control objects that is completely substituted to the standard procedural activation (the method proceduralEvaluate). In this scheme, a metabase is associated to the current rule base to be activated. The activation of the rule base consists simply in activating (recursively) its metabase. The basic inference loop, choice of rules to be fired, and more generally the management of control objects will all be defined by rules of the metabase. More precisely, the firing of a metarule operates either a modification of the control model (universe of control objects) and/or the firing of a domain rule via the conflict set of the rule base being activated.
Of course, this architecture raises a regression problem : how is the metabase itself activated ? This problem is simply solved by forbidding loops in the control tree : a rule base is either activated in a procedural manner or by the activation of its metabase. A metabase may not be activated by itself.

6. Discussion

We described the design of the NéOpus system, emphasising on its freedom of expression, and stressed on two original features of NéOpus: rule base inheritance and a declarative architecture for control. Although this freedom of expression is paid by several inherent inefficiencies, we show how this freedom may be used to write complex (control) knowledge bases.

7. References

[1] Alizon F., Huet G. Essaim: A Smalltalk Programming Environment for the Construction of Expert Systems. Technical report, CNET NT/LAA/SLC/299, Lanion, France, (1988).
[2] Atkinson R., Laursen J. Opus : A Smalltalk Production System. Proc. of OOPSLA'87 pp. 377-387, (1987).
[3] Batali J. Reasoning about self-control. In Meta-level Architectures and Reflection, P. Maes et D. Nardi eds. North Holland, (1988).
[4] Brownston L., Farrell R., Kant E., Martin N. Programming Expert Systems in OPS5. An Introduction to Rule-Based Programming. Addison-Wesley Publishing Company, (1985).
[5] Chun, R. Perry B. An Environment for the control and Software Integration of Expert Systems. Proc. of SEKE '93, pp. 499-506, (1993).
[6] Clancey, W. The advantages of Abstract Control Knowledge in Expert-System Design. Report N° STAN-CS-83-995. Stanford University, (1983).
[7] Cohen, P. Delisio, J. Hart, D. A Declarative Representation of Control Knowledge. IEEE transactions on Systems, Man, and Cybernetics, Vol 19, n 3, May/June (1989).
[8] Dojat M, F. Pachet. NéoGanesh: an Extendable Knowledge-Based System for the Control of Mechanical Ventilation. 14 th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, October 29-Novembre 1st, Paris (1992).
[9] Laublet, P. Hybrid Knowledge Representation and Theorem Proving in Mathematics. Proc. of Conf. Artificial Intelligence in Mathematics, pp. 181-196, Glasgow, April 1991. To appear in Artificial Intelligence in Mathematics, J.H. Johnson, S. McKee & A. Vella (Eds), Oxford University Press, (1994).
[10] Nebel, B. Reasoning and Revision in Hybrid Representation Systems. Lecture Notes in AI 422, Springer (1990).
[11] Pachet F. Mixing Rules and Objects: An experiment in the world of Euclidean Geometry. ISCIS V, 30 Oct. - 2 Nov., Nevsehir, Turkey, pp. 797-805, (1990).
[12] Pachet F. Reasoning with objects: the NéOpus environment. Proc. of Int. Conf. East EurOOpe, Bratislava, Tchécoslovaquia, Sept. (1991).

[13] Pachet F. Rule base inheritance. Conference on Object-centered Representations, La grande Motte, France, June (1992).
[14] Pachet F. Knowledge Representation with objects and rules: the NéOpus system. PhD thesis, Paris VI University, Sept. (1992).
[15] Perrot, J.-F., Pachet, F. Rule firing with metarules. Software Engineering and Knowledge Engineering. Jurmala, Latvia. Knowledge Systems Institute Eds. June 1994.
[16] Patel-Schneider, P. Practical, Object-Based Knowledge Representation for Knowledge-Based systems. Information Systems (Oxford), Vol 15, N. 1, pp 9-19, (1990).
[17] Wolinski, F. Perrot, J.-F. Representation of complex Objects: Multiple Facets with Part-Whole Hierarchies. Proc. of European Conference on Object-Oriented Programming, Geneva, July (1991).