Pachet, F. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming*, Vol. 8, n. 4, July/August 1995, pp. 19-24.

On the embeddability of production rules in object-oriented languages

François Pachet,
LAFORIA - Institut Blaise Pascal, Boite 169,
4, Place Jussieu, 75252 Paris Cedex 05, France
email : pachet@laforia.ibp.fr

Abstract

This article addresses the problem of *embeddability of object-oriented production systems*. We propose five important practical issues concerning embeddability, among which the *impedance mismatch* problem. We described an extension of Smalltalk to support production rules (the NéOpus system) and discuss its embeddability according to these criteria.

I. Introduction

The combination of rules and objects to produce integrated *hybrid systems* was pioneered by the seminal LOOPS system [Bobrow&Stefik 83]. A number of integrated systems were consequently developped which propose a combination of various inference mechanisms and knowledge representation paradigms, such as KEE, Knowledge Craft and ART. These systems have traditionally been designed as self-contained inference factories, with few concerns about inter-software communication and reusability. A recent expert panel [AIII-90] showed that the lack of impact of Artificial Intelligence in the field of software engineering was partly due to these isolationist attitudes, and proposed that efforts should be made not only to produce autonomous inference environments, but also *embeddable* systems [Fox 90]. Indeed, embedding deductive facilities in object-oriented settings has been in the air for quite some time now. Both paradigms are suitable for different kinds of programming tasks and their integration is needed and desired by both communities: object-oriented programmers occasionally need non-deterministic, data-driven procedures because these include an implicit pattern-matching algorithm as well as a more open mode of control. Conversely, rule base programmers need more sophisticated representations of entities to represent the "facts" of the world about which rules can talk. Object-orientation allows them to replace some awkward rules by more appropriate methods or class structures.

There has been propositions of rule-based extensions for virtually all object-oriented languages on the market : in C++ [Miranker 91], [Eick&Czejdo 93], Eiffel [Fernandez & Jones 94], CLOS [KnowledgeWorks 91], to mention but a few. This profusion of systems is a sign of the vitality of research in this area but hides a lack of consensus about what this integration should be, what are the advantages one can get out of it, what are the prices to pay in terms of implementation and performance, and, lastly, what kind of methodology this hybrid programming induces. At LAFORIA lab, we have been working for a few years on this integration. We present here some of the important issues and result from our point of view, and propose five major points to classify and compare so-called *embedded object-oriented production systems* , hereafter referred to as EOOPS.

Our point of view on this integration is materialized by the NÉOPUS system, written in Smalltalk-80, which we will describe shortly here. This system is based on an original presentation of the OPUS system by Atkinson&Laursen, in OOPSLA '87. This article described a translation of the OPS-5 system [Forgy 81] in Smalltalk-80, and contained the description of the language as well as the major implementation choices. Since the publication of this article, we have been experimenting with several versions of OPUS, eventually coming up with a new system called NÉOPUS. The growing success of the system in our community of Smalltalk programmers convinced us that the hypothesis made by the original authors were not only relevant, but opened a whole new world of novel and efficient programming practice to object-oriented programmers.

II. Vocabulary

To avoid confusion, we use the term *Object* in the sense of object-oriented programming languages, i.e. entities gathering data structure (attributes, slots or instance variables) and behavior (methods). The A.I. community often uses the term *object* in a looser sense, closer to the notion of frame, i.e. without methods but with active values, and facets, which objects do not support. We will use the term *rule* to refer to forward-chaining production rules, as exemplified by the OPS-5 system [Brownston & al 85]. These may be seen as non-deterministic conditional actions. Their canonical form is the standard if-then form: *IF some-conditions THEN some-actions*. As we will see, one of the questions a hybrid system should answer is how are the conditions and actions expressed. Rules are grouped in rule bases which are activated by an inference engine. The engine works in a continuous loop in which, at each cycle, a *fireable rule* is chosen and fired. The execution stops when no more rules are fireable.

III. Mixing objects and rules : Five questions

We identified five major questions regarding the integration of objects and rules, both from an engineering and a conceptual point of view. These are general questions whose answers may be used to classify hybrid systems. We will describe the NÉOPUS answers to these questions in the next section.

A - Impedance mismatch

The mere presence of classes and methods implicitly defines a language. This language is the (infinite) set of all expressions built by sending messages to objects, which may themselves recursively be expressed by a message sending or by a variable. In Smalltalk this language is roughly equivalent to the programming language, because message passing is the only programming construct (the only exception is assignment). Other languages introduce special constructs to express message sending. But in any case, be it explicit or implicit, there is an "object language", which plays a fundamental role in object-oriented knowledge representation.

On the other hand, writing rules requires a language to express their condition and action parts. This language usually allows to specify constraints on attributes, to modify attribute values, or create so-called "facts". Usually some trap-doors are provided to execute any action of the underlying implementation language in the action part of a rule.

Embedding a production system in an object-oriented system (in short OOPS) amounts to assert the equation: {fact-base = object-oriented model of the world}. Facts are represented by objects, and rules are used to express knowledge involving several objects.

The *impedance mismatch* problem is the degree of mutual compatibility between these two languages. This compatibility is not trivial because, since their apparition in the first expert systems (Mycin, Dendral), rule languages have always relied on the possibility of accessing the *structure* of the facts in working memory elements. This is inherently orthogonal to the object-oriented philosophy which tends to hide data structure from the outside of objects, as we will see later. Consequently, most hybrid systems actually consist of two different languages, one for objects, and one for rules. Although it is difficult to define precisely how this degree of mutual compatibility could be measured, it is sure that the introduction of a new language is always a source of problems. As a side-effect, a question which is less fundamental but has important practical consequences is the availability of all the syntactic constructs of the object language in rules, such as local variables, and differed expressions (lambda-expressions in Lisp, blocks in Smalltalk).

B - What about encapsulation ?

There are two orthogonal questions regarding encapsulation. These questions are related to the impedance mismatch problem but we make it a separate issue because the problems they raise are quite different.

> B1: How much does the rule base programmer have to know about the implementation details of the application to write his rules ?

There is an inherent contradiction in integrating rules and objects, concerning encapsulation: objects are designed primarily to *hide* data structure and implementation details, whereas classical production rule systems are based on the explicit manipulation of structure (attributes). Indeed, writing expressions mentioning attributes is precisely what encapsulation allows to avoid, and cannot be considered good software engineering practice!

> B2: What are the modifications to add to the application to support the addition of a rule base layer?

The main interest behind the "embedding" idea is to be able to add a rule-based layer to an existing application without having to modify and recompile the application. However, a lot of EOOPS impose constraints on the application objects, such as the *inheritance constraint* , i.e. the classes of objects matched by the rules must inherit from a determined superclass.

C - What about class inheritance ?

Class inheritance is an important aspect of object-orientation. It is important that rules take class inheritance into account in a natural way. The way rules can take class inheritance into account is by specifying how rule variables are typed, and what is the meaning of the implicit universal quantifier. There are three basic possibilities for rule variable typing. Let us, for instance, consider a rule such as:

"For any `X`, instance of `Person`, If ... then ..."

In this rule, `X` may either 1) denote *direct instances* of class `Person` 2) denote instances of *all its (potential) subclasses*, or 3) denote instances of a *part of the inheritance hierarchy* of `Person`. An EOOPS mechanism should provide a reasonable answer to this question by specifying how the typing of rule variables is realized.

D - What is the status of rule bases ?

The status of rules and rule bases is important from the point of view of integration for implementation and practical reasons. Some acknowledged problems of rule-based programming may be answered thanks to object-orientation. We identified two of them :

- Structuring rule bases

The key mechanism of rule-based programming is non-determinism (here the fact that a rule is not necessarily fired when it is fireable). Writing large bulks of non-deterministic rules is a difficult task, because large problems are not usually entirely non-deterministic. Rule bases strongly need structuring mechanisms.

- Specification of control

An important question regarding forward-chaining rule bases is the control problem. This problem, central to any A.I. system, is often a bottleneck in the development of large knowledge bases. Here again, object-orientation may be used in a variety of ways to support various types of control specifications.

E - What about logical orders and modes of reasoning ?

Logical orders

Objects are naturally seen as anonymous instances of their class. The classical, first-order way of accessing, with an implicit universal quantifier ("For any x, instance of C"), is therefore most adapted to objects. However, first order forward-chaining reasoning is not the only mode of reasoning. Reasoning without variables (so-called 0-order or propositional reasoning), although theoretically less expressive than first-order reasoning, may sometimes be extremely well adapted to some situations, where individual objects play a singular role.

Chaining

There are traditionally two modes of chaining for an inference engine, namely forward and backward chaining. As we will see, we consider forward-chaining the most interesting mechanism to be considered for integration with object-oriented languages. Other modes may be offered, such as more or less directed reasoning, where rules indicate which following rules are to be considered (as in the Humble system [Piersol 86]).

IV. The NéOpus philosophy

IV.A. An example

We will now describe how the NÉOPUS system answers these five questions, and illustrate them with the following (artificial) example. We want to represent rules that represent the expertise about treatments on hyper-tension. We suppose that the following classes are defined :

A class `Patient` represents the patients to diagnose and treat. This class defines the methods `bloodPressure`, yielding the blood pressure of the patient. We intentionally do not mention anything about its implementation, as we consider this to be irrelevant to rule programmers (this is what encapsulation is all about !).
A class `Doctor` represents doctors, and implements methods `maxBloodPressureFor:` which returns the maximum blood pressure for a `Patient`, according to the Doctor's expertise. This method is a function of the patient's age, weight, and is highly experimental. Its result can be computed using tables, histograms, or rules of thumb. To represent this diversity of possible computations, we are naturally inclined to use class inheritance, and define subclasses of `Doctor`, such as `PessimisticDoctor`, or `HomeopathicDoctor` which redefine locally this method.

Class `Doctor` also defines a method for considering treatment for a patient, once hyper tension is diagnosed. This method, `considerTreatmentFor:` is also highly dependent on the person's state, and the doctor expertise. Several subclasses may also be defined to represent various types of treatment, such as `HomeopathicDoctor` or `AllopathicDoctor`, to take into account the various idiosyncrasies of blood pressure treatment.

IV.B. NéOpus

Let us now describe the NÉOPUS solution to our five questions and illustrate them with our example.

A - Impedance mismatch

The impedance mismatch problem is solved in NÉOPUS by the radical decision : no rule language is introduced. Condition and action parts of rules are expressed exactly in terms of Smalltalk expressions, i.e. the *object language*. The rule syntax is therefore reduced to a name, a variable declaration part, and the IF-Then construct (here the key-word `actions`).

For example, let us state a rule that decides on a treatment when blood pressure is too high as follows:

```
decideTreatmentHypertension
   | Doctor d. Patient p|
   p bloodPressure > d maxBloodPressureFor: p.
   p hasNoTreatment.
actions
   d considerTreatmentForHypertension: p.
```

This rule should be read as : "For any `d`, instance of class `Doctor`, and any `p`, instance of class `Patient`, if the evaluation of the Smalltalk expressions: "`p bloodPressure > d maxBloodPersonFor: p`", and "`p hasNoTreatment`" both yield true, then the expression "`d considerTreatmentForHypertension: p`" should be evaluated.

B - What about encapsulation ?

NÉOPUS adopts a radical attitude towards encapsulation. We consider encapsulation as a key mechanism of object-oriented programming, and wish to enforce it in the context of rule-based programming. Our idea is therefore to respect encapsulation as much as possible, and this translates into two axioms :

B1: *Any object may be matched in a rule*. There is no constraint on the class of objects mentioned in the rules. This is very important in our context for two reasons:

- Classes written elsewhere may be used directly without any recompilation. In our example, this means that the file containing the rules may be "filed in" any time in the application containing the `Doctor` and `Patient` classes without having to recompile them.
- Classes of the environment may be used in rules, such as collections, metaclasses, windows and so on. There are approximately 60,000 objects in a standard Smalltalk image, which makes it a rich environment to reuse. Of course, NÉOPUS allows the restriction of the initial set of objects to be matched for a given class, to avoid unnecessary combinatorial explosion.

The second axiom is a consequence of the "one language" choice :

B2: *Any Smalltalk expression may be used in premise and condition*. In comparison to other rule-based systems, this means that rules are not limited to manipulations of attribute/values, and may use freely all the corpus of methods of the language. More precisely, encapsulation is respected in the sense that the writer of the rule does not need to know anything about the actual *implementation* of the classes he uses, as exactly the *interface* of classes (the set of methods they implement) - and all of it - may be used.

As for particular syntactic constructs, NÉOPUS allows the presence of local variable in rules (Cf. rule `decideNoTreatment` below). Similarly, NÉOPUS allows the presence of blocks (expressions whose evaluation is differed, written between brackets) containing rule variables. This is useful when these blocks are evaluated in a reflex-mode, in a continuous loop for instance, to implement triggers.

C - What about class inheritance ?

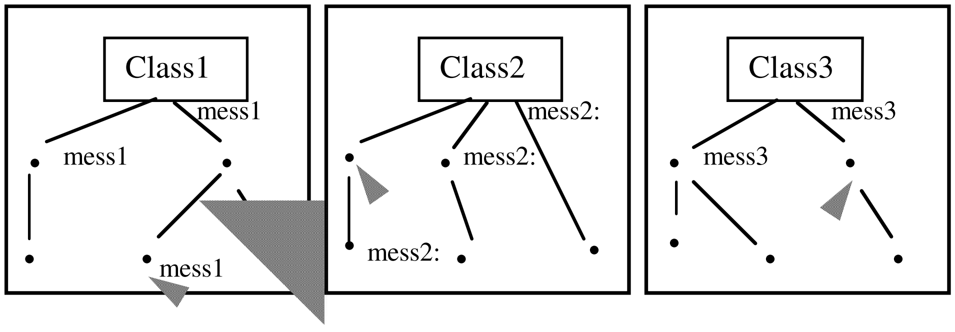
In NÉOPUS, we distinguish two ways of typing rule variables, to take class inheritance into account. In the first - called *simple typing* - variables denotes direct instances of the classes with which they are defined in the variable declaration part. On the other hand, in *natural typing*, variables denote any instance of any subclass of the class used to declare them. In this latter case, increased abstraction is given to rules, as their interpretation may vary depending on the actual instances matching it.

For instance, if let us consider different subclasses of `Doctor`, having different implementations for methods `maxBloodPressureFor:` and `considerTreatmentForHypertension:`, such as `OptimisticDoctor` (who considers that young people should rarely be given treatment), and `HomeopaticDoctor` (who has a standard way of diagnosing hypertension, but a non-standard different way of considering treatments). Similarly, let us consider a subclass of `Patient`, such as `HospitalizedPatient`, for which method `hasNoTreatment` is redefined (checks in the hospital file for instance). Now the preceding rule `decideTreatmentHypertension` may be used, without any modification or recompilation, but with a different interpretation.

This is important with regard to integration because it means that class inheritance can be used naturally to redefine behavior locally.

Moreover, rules are given more abstraction by delegating the message semantics and implementation details to the classes. The interpretation of a rule is therefore *dynamic*, since the condition and action parts of the rules are entirely expressed in terms of messages sent to the matched objects. More precisely, let r be a rule that declares n variables v_i (i=1, n). Let C_i be the class declared for v_i. Now, if each C_i has k_i concrete subclasses, the total number of possibly different interpretations of r is $\Pi\ k_i$ (i = 1, n). As a consequence, the number of rules is drastically reduced, since the same text may be used for any combination of classes. In a classical rule-based approach, the programmer would have to write $\Pi\ k_i$ (i = 1, n) different rules instead of one.

In our (artificial) example, there are 3 possible classes for `Doctor`, and 2 for `Patient`, making a total of 3*2 = 6 different interpretations of the rule. Moreover, each addition of a new subclass would require the addition of a set of new rules. Thanks to encapsulation, only a minimal set of rules needs to be written.



```
r1
| **Class1** v1. **Class2** v2 **Class3** v3 |
 If  v1 mess1.
     v2 mess2: v3.
 Then
    v3 mess3.
```

Figure 1. A rule with the corresponding class hierarchies. The rule uses three messages (`mess1` through `m3`) which are defined and redefined in the classes of the corresponding hierarchies.

It is important to note here that in this scheme, the actual semantics of a rule is determined not by the rule alone, but by the rule together with its instanciation set, at execution time (usually called the *fireable rule* and represented by a first-class object in NéOpus).

D - What is the status of rule bases ?

Structuring rule bases

In NÉOPUS, rules are grouped in rule bases. Rule bases are represented by abstract, Smalltalk classes. This is motivated by several reasons :

- The Smalltalk environment for classes may be used freely for rule bases (browsers, file in, file out, cross-references). This point has been commented by [Atkinson&Laursen 87].
- An *inheritance mechanism*, similar to class inheritance, is introduced for rule bases, to structure them. This mechanism (Cf. [Pachet 92a] for more details) allows to factor rules which are common to several rule bases, thereby providing an other way of reducing the total number of rules written.

For instance, let us introduce a new rule for the case no treatment should be considered:

```
decideNoTreatment
"If pressure is normal then consider abandoning treatment"
|Doctor d. Patient p|
   p bloodPressure < d maxBloodPressureFor: p.
   p hasTreatmentForTension.
actions
   d considerAbandoningTreatmentFor: p.
```

We can group the rules `decideTreatmentHypertension` and `decideNoTreatment` in a rule base called `StandardDiagnosis`. Several sub-bases of `StandardDiagnosis` may now be defined, to either add new rules, or redefine inherited rules. For instance, we can write a specialized subbase of `StandardDiagnosis`, called `CompleteDiagnosis`, that adds rules for treatment of *hypotension*, such as :

```
decideTreatmentHypotension
   | Doctor d. Patient p|
   p bloodPressure < d minBloodPressureFor: p.
   p hasNoTreatment.
actions
   d considerTreatmentForHypotension: p.
```

`CompleteDiagnosis` has to redefine the rule `decideNoTreatment`, to take hypotension into account, as follows :

```
decideNoTreatment[1]
"If pressure is normal then consider abandoning treatment.
Redefined in subbase to take hypotension into account"
|Doctor d. Patient p. Local bloodP|
   bloodP := p bloodPressure.
   bloodP > d minBloodPressureFor: p.
   bloodP < d maxBloodPressureFor: p.
   p hasTreatmentForTension.
```

[1] Notice here the use of a local variable (`bloodP`), which is used as in standard Smalltalk methods, to reduce the number of computations and increase readability.

```
actions
   d considerAbandoningTreatmentFor: p.
```

By virtue of the rule base inheritance mechanism, the rule base `CompleteDiagnosis` contains the three following rules : `decideTreatmentHypotension` (inherited), `decideTreatmentHypotension` (added) and `decideNoTreatment` (redefined). Of course, class inheritance is still taken into account and rule base `CompleteDiagnosis` may be used using any of the typing mode (natural or simple), independently of the typing used for the super base `StandardDiagnosis`.

Specification of control

NÉOPUS proposes two solutions to the control problem, that make full use of object-orientation. In the first one, control is seen as a procedure for rule-bases, and implemented by class methods. Executing a rule base consists in sending the rule base (considered as an object) one of the standard execution messages, such as `execute`, or `executeWithObjects:` (to specify which objects should be filtered by the rules).
In complex cases, however, the procedural solution has well-known limits. In particular it is difficult to write methods to specify dynamically-changing control strategies. The other solution proposed by NéOpus is to consider the control problem as a standard knowledge representation problem, and use - in a reflexive way - NéOpus to solve it. In this scheme, the activation of a rule base is entirely defined by another rule base (called meta-base). This meta-base contains rules that specify control, including the actions to perform to choose a rule in case of conflict, and the management of the fireable rule and the conflict set. In this latter scheme, meta-bases may be built independently of domain rule bases, and reused in a variety of applications. For instance, a meta-base was written to specify a control in which rules are grouped in packs (called protocols in the Smalltalk terminology), and each pack is considered according to a given sequence. This meta-base is independent of any application and is used in a variety of domains. Several subbases are written to extend this specification for more sophisticated modes of controls (Cf.[Pachet & Perrot 1994], and [Pachet&Dojat 92] for a real-world application).

E - What about logical orders and modes of reasoning ?

Logical orders

To overcome the rigidity of first-order reasoning, NÉOPUS introduces the notion of individual objects that may be seen as a translation of 0-order inference in the world of EOOPS. In this scheme, certain objects may be considered as *named individuals*, and given a special treatment by the inference engine (in our example, we could consider the *Hospital* as an individual object). Rules can talk about these objects (by declaring them "global") and the system may perform 0-order inferences with them. These objects are represented by class variables for the rule base which uses them. Unlike standard global variables, these objects are really part of the inference mechanism in that their *modifications* are taken into account by the system and rules instanciations updated when necessary.

Chaining

Since many expert system shells support multi-directional chaining (i.e. forward, backward and so-called bi-directional), this would appear to be an obvious evaluation criterion. However, we believe there are several good arguments for not supporting multi-directional chaining in rule-based systems embedded in an Object-Oriented language.

First, procedural languages (such as Object-Oriented Programming languages) are intrinsically backward chaining : procedures are called explicitly and in turn generate trees of procedure calls. What is not present is unification (the equivalent of pattern matching) and backtracking. However, backtracking can easily be simulated in an Object-Oriented framework (Cf. [Lalonde & Van Gulik 88] for instance).

A second, more technical argument, is that backward chaining is intractable in a general object-oriented setting. The problem is that backward chaining implies the possibility of linking the action parts of rules to the condition parts (or goals to sub-goals). This is possible only if conditions and actions are structurally simple, which is the case for 0, 0+ or attribute-value based formalisms, but *not* for general formalisms in a first order object-oriented language. There is indeed a contradiction between the desire to use an OOP language to express rules, and the desire to support backward chaining. Backward chaining can only be supported if we are prepared to impose hard constraints on the programmers which would seem (to them) to be artificial.

Finally the main advantage of backward-chaining systems such as Prolog, when used as extensions to already existing procedural formalisms, is the possibility of specifying trees of goals/subgoals. We claim that this notion is better seen as a "control problem" and should be solved by proposing particular control strategies.

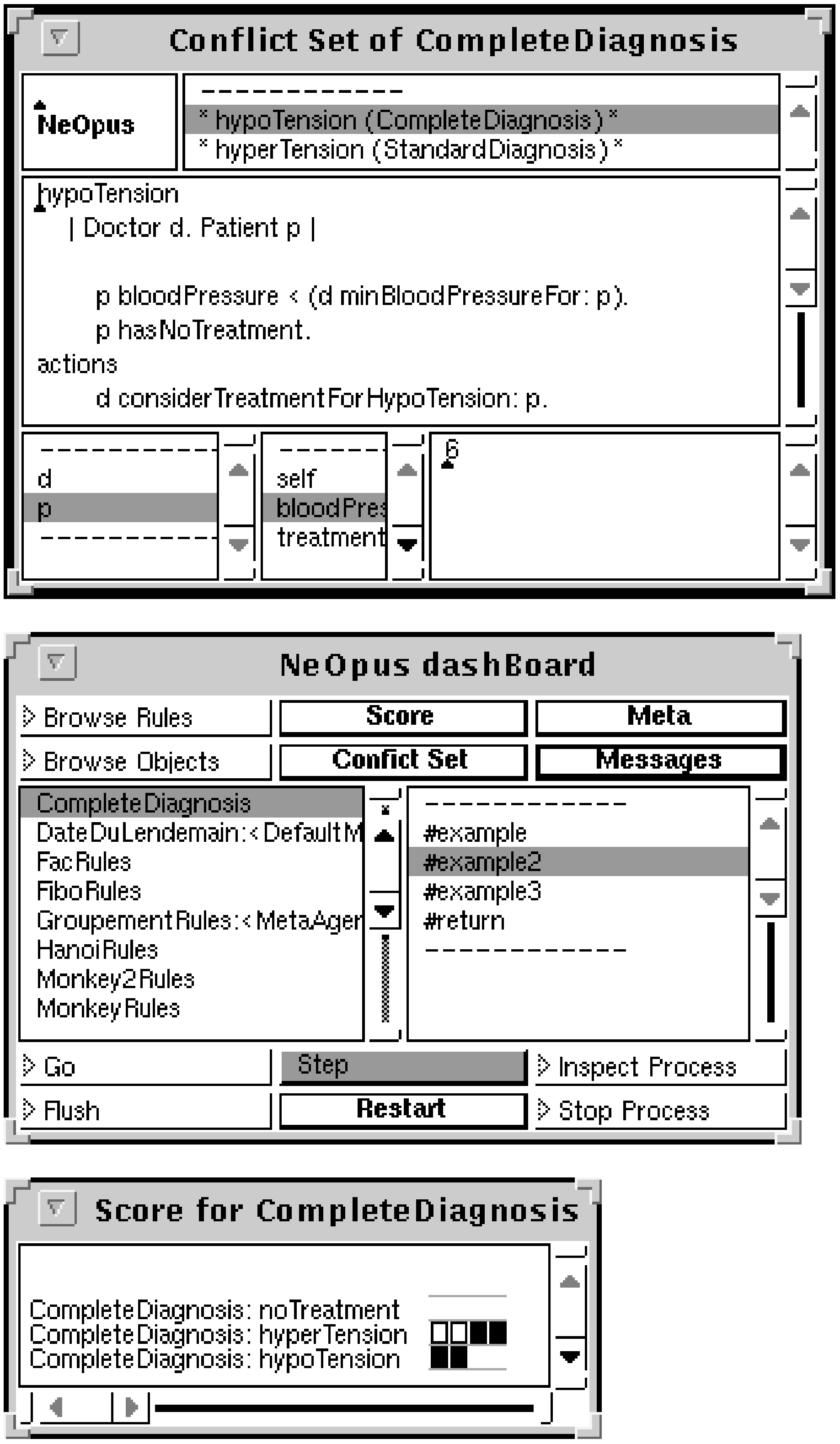
V. Implementation

Rules are compiled in NÉOPUS using an extended Rete network architecture. This compilation technique was pioneered by Forgy [Forgy 81], and is considered one of the best for pattern-matching with many objects (it is for instance, the core of the CLIPS system [CLIPS 92] made by the NASA for real-time expert systems). It was extended to take full-fledged objects into account by Atkinson & Laursen, in the OPUS system. The transposition of Rete in the object world makes it necessary for the programmer to *declare explicitly* which objects are *modified* by the execution of the action part of a rule (this is referred to as the *modified problem*). We will not give details on this architecture here, but the reader can refer to [Atkinson&Laursen 87] and [Pachet 92b] for a complete description. Concerning the *modified problem*, we refer to the proceedings of the OOPSLA'94 workshop on EOOPS [EOOPS Workshop 94] which provides several viewpoints and solutions on this issue.

VI. Interface

A number of related problems find rather elegant solutions in the NÉOPUS system. The programming interface is translated from the programming interface of Smalltalk-80, which is one the most sophisticated at the moment. A set of specialized rule browsers, instance browsers, conflict-set view, steppers were designed to increase rule-based programming, while retaining the Smalltalk flavor.

A particularly well adapted mode of debugging and visualization for rule execution was implemented, using a musical score metaphor, introduced by [Domingue & Eisenstadt 91]. With this tool, rule base execution is visualized as a musical score, where time is explicitly represented as an x axis. To each rule corresponds a position on the y axis, like a pitch on a score. Each time a rule becomes *fireable*, a white square is displayed on the score at the corresponding location. Each time a rule is fired, a black square is displayed (Cf. Figure 2). As Domingue & Eisenstadt argue, this allows to emphasize - and sometimes reveal ! - temporal relationships between rules, which are usually not shown in standard textual traces.



```
------------
* hypoTension (CompleteDiagnosis) *
* hyperTension (StandardDiagnosis) *
```

```
hypoTension
   | Doctor d. Patient p |


      p bloodPressure < (d minBloodPressureFor: p).
      p hasNoTreatment.
actions
      d considerTreatmentForHypoTension: p.
```

```
-----------
d
p
-----------
```

```
------
self
bloodPres
treatment
```

```
6
```

```
CompleteDiagnosis
DateDuLendemain:<DefaultM
FacRules
FiboRules
GroupementRules:<MetaAger
HanoiRules
Monkey2Rules
MonkeyRules
```

```
------------
#example
#example2
#example3
#return
------------
```

```
CompleteDiagnosis: noTreatment
CompleteDiagnosis: hyperTension     □□■■
CompleteDiagnosis: hypoTension      ■■
```

Figure 1. Some elements of the NÉOPUS interface.

VII. Conclusion

NÉOPUS has been used in a variety of applications, ranging from real-time control of respiratory devices in intensive care units (Cf. [Pachet&Dojat 91]), theorem-proving [Laublet 93], data model transformation [Blain &al. 94] and harmonic analysis [Pachet 91b]. Since no modification of the domain classes is needed whatsoever to write rules bases, a simple file-in of the NÉOPUS system turns any Smalltalk application into a fully-fledged knowledge base. Besides, since rule bases are standard Smalltalk classes, hence global objects, they may be executed by any object of an application.
Finally, their execution may be easily controlled by asynchronous messages without any modification of the inference mechanism. A number of actor-based applications using the Actalk system [Briot 89] have been extended with a NÉOPUS rule base component, to experiment with multi-agent programming. Lastly, its open character of makes it a very valuable tool for teaching knowledge representation techniques in the framework of object-orientation.
NÉOPUS is implemented in Smalltalk-80, version 4.1. Implementations in other dialects of Smalltalk is in progress, as a collaborative project with the Canadian company OTI.

The system is available by anonymous ftp[2]. We encourage interested readers to test and use the system, and communicate any comments.

VIII. References

[AAAI-90] Panel "AI and Software Engineering : Will the Twain Ever Meet ?". *Proc. of AAAI 90*, pp. 1123-1129. (1990).

[Atkinson&Laursen 87] Atkinson, R. and Laursen, J. Opus: a Smalltalk Production System. *Proc. of OOPSLA '87* pp. 377-387. (1987).

[Blain & al 94]. Blain, G. Sahraoui, H. Revault, N. Perrot, J.-F. A Meta-modelisation technique. *OOPSLA'94 Workshop on Artificial Intelligence and Object-Oriented Software Engineering*. Portland, Oregon, Oct. (1994).

[Bobrow&Stefik 83]. Bobrow, D.J., Stefik M. The LOOPS Manual. *Xerox Palo Alto Research Center*, Dec. (1983).

[2] The site is ftp.ibp.fr, login under anonymous with electronic address as password. The sources are tarred and compressed under softs/laforia/sourcesNeOpus. A Web page is also available at http://www-laforia.ibp.fr/~fdp/NeOpus.html. It runs on Smalltalk 4.1, on any platform supported by ParcPlace. It requires the Parser generator tool from APOK (ParcPlace). A list of current users is maintained and information broadcast to users.

[Briot 89] Briot, J.-P. Actalk : A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 environment. *Proc. of ECOOP '89*, pp. 109-130. (1989).

[CLIPS 92] CLIPS Reference Manual, v 5.1, *NASA*, Jan. (1992).

[Brownston&al. 85] Brownston, L. Farrell, R. Kant, E. Martin, N. Programming Expert Systems in OPS5. An Introduction to Rule-Based Programming. *Addison-Wesley Publishing Company*, (1985).

[Domingue & Eisenstadt 91] Domingue, J. Eisenstadt, M. A new metaphor for the graphical explanation of forward-chaining rule execution. *Proc. of IJCAI '91*, Sydney, Australia, pp. 129-134. (1991).

[Eick&Czejdo 93] Eick, C.F. Czejdo, B.Reactive rules for C++. *Journal of Object-Oriented Programming*, Vol 6, n° 6, pp. 56-62, Oct. (1993)/

[EOOPS Workshop 94] Pachet, F. Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems. *LAFORIA Internal Report*, University of Paris 6, n. 94/24. (1994).

[Fernandez & al 94] Fernandez, R. Zubizareta, J.-R.Embedding of rule-based expert system capabilities in object-oriented applications by using or simulating active behaviour. *Proc. of TOOLS Europe '94*, Paris, March (1994).

[Forgy 81] Forgy, C.L. OPS-5 User Manual. *Department of Computer Science*, Carnegie-Mellon University, (1981).

[Fox 90] Fox, M. Looking for the AI in Software Engineering : An Applications Perspective. *In [AAAi-90]*, pp.1128-1129. (1990).

[Humble 86] Piersol, K.W.The Humble Reference Manual. *Xerox Special Information Systems*, June (1986).

[KnowledgeWorks 91]. Knowledge Works: Guide for new Users. *Harlequin*, 1991.

[Lalonde & Van Gulik 88]. Lalonde, W. Van Gulik, M. Building a backtracking facility for Smalltalk without kernel support. *Proc. of OOPSLA '88*, pp. 105-123 (1998).

[Laublet 93] Laublet, P. Hybrid Knowledge Representation and Theorem Proving in Mathematics. In *Artificial Intelligence in Mathematics*, J.H. Johnson S. McKee & A. Vella (Eds), Oxford University Press, (1994).

[Miranker 91]. Miranker, D. & al. The C++ embeddable rule system. *Proc. of Int. Conf. on Tools for Artificial intelligence*, San Jose, Nov. 91, pp. 386-393 (1991).

[Pachet 91] Pachet, F. Reasoning with objects : the NéOpus environment. *Proc. of Conf. East EurOOpe*, Bratislava, Tchécoslovaquia, Sept. (1991).

[Pachet 91b] F. Pachet, A meta-level architecture for analysing jazz chord sequences. *Proc. of International Conference on Computer Music*, pp. 266-269, Montréal, Canada, (1991).

[Pachet 92a] Pachet, F. Rule Base Inheritance. *Proc of Conf. "Object-Oriented Representations"*, La grande Motte, France, June (1992).

[Pachet 92b] Pachet, F., Knowledge Representation with objects and rules : the NéOpus system. *Ph.D. Thesis* (in French), Universite Paris-6, Laforia, Paris, France, Sept. (1992).

[Pachet&Dojat 91] Pachet, F. Dojat, M. Representation of a Medical Expertise Using the Smalltalk environment: putting a prototype to work. *Proc. of TOOLS 7*, Dortmund, Germany, March 31-April 2, (1992).

[Pachet & Perrot 1994]. Pachet, F. & Perrot, J.-F. Rule Firing with Metarules. *Proc. of Software Engineering and Knowledge Engineering - SEKE '94*, Jurmala, Latvia. Knowledge System Institute Ed. pp. 322-329, 21-23 June (1994).

[Piersol 86] Piersol, K.W. The Humble Reference Manual. *Xerox Special Information Systems*, June (1986).