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: