Pachet, F. Perrot, J.-F. Rule Firing with Metarules. *Software Engineering and Knowledge Engineering - SEKE '94*, Jurmala, Lettonie. Knowledge System Institute Ed. pp. 322-329, 21-23 juin 1994.

Rule Firing with Metarules

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

We describe a technique for the control of production rules firing in an object-oriented setting. This technique is based on the separation of control rules from ordinary domain rules. Control rules operate on "control objects" which are created during the reasoning process of the rule base under control. They constitute a separate and independent rule base which contains a declarative specification of the control strategy. Control objects build up an inheritance hierarchy and the associated metabase is constructed via rule base inheritance in a hierarchical manner which parallels the taxonomy of control object classes.

AI Topic:	Knowledge Engineering
Domain area :	Explicit control of reasoning
Language:	Smalltalk-80
Status:	Research Application
Effort:	1 person-years
Impact:	This architecture is used for real time monitoring of patients in intensive care units.

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.

This paper is a sequel of their communication at OOPSLA '87. The first author reimplemented OPUS with several improvements as part of his doctoral research [10], [11], [13]. He carried out the rule base inheritance proposal [12]. A number of experiments were conducted with his system, called NÉOPUS. In the course of these experiments, a novel and powerful technique for rule firing control emerged, based on separate "control rule bases" operating on specific "control objets". Control objects, however, are perfectly ordinary Smalltalk objects, and control rule bases are structurally identical with ordinary rule bases. This technique is the subject-matter of the present paper.

Our architecture allows the construction of a hierarchy of reusable standard control bases, associated with a parallel hierarchy of control object classes, some of which are delivered with the system. These standard bases can be used either

directly or with refinement through rule base inheritance (and class inheritance for the control objects on which they operate), after the traditional fashion of object-oriented programming. Admittedly, writing such control rule bases does require some practice. But factoring out the control from the domain knowledge clearly brings a major improvement in rule base engineering.
The paper is organized as follows: first we give a short discussion of the OPUS and NÉOPUS systems; then we describe the control problem in our precise object-oriented setting, where control objects appear in a very natural way ; finally we explain the working of control rule bases and the use of rule base inheritance.

2. Rule-based programming in Smalltalk-80 : OPUS and NÉOPUS

2.1. *From rules to objects*

The need for combining object structures and rule-based programming has been widely recognized. The fact base of a rule-based program is a model of the concrete situation that is currently being processed. To bring some semantic structure to facts, one naturally tends to see them as properties of objects that build up a universe simulating the concrete world. Individual facts are no longer represented as such, their logical value is ascertained by querying objects in the model. The fact base is thus dissolved into an object-oriented model of the world. This operation is so natural that objects have crept into rule-based formalisms (in a rudimentary form) as early as OPS-5.

As a consequence, the main rule-based knowledge representation systems (KEE, ART and followers) all have a strong object-oriented component. These object formalisms, however, have usually been defined to suit the reasoning process and rely on frames rather than on objects in the sense of, say, Smalltalk. Accordingly, they are more complex than the standard structures of object-oriented programming languages, i.e. class/instance and inheritance mechanisms. Even when they are defined using such a language, they usually constitute an additional layer on top of the "autochtonous" objects of the language (with the interesting exception of Essaim [1]).

We proceed in the reverse direction, starting with a standard object-oriented language and extending it with a rule-based layer as "thin" and "seamless" as possible. Our aim is to enrich the class/instance paradigm with rule-based deductive mechanisms. In order to benefit from the work of others, we have chosen Smalltalk-80 as our language. Our system makes full use of Smalltalk metaclasses. Its translation to a language without this facility (e.g. Eiffel) would require some rewriting but no fundamental revision.

Note that the idea of enhancing the knowledge representative capacities of standard object-oriented languages, especially of Smalltalk, is quite common. See [16] for an extension proposal to classification mechanisms.

2.2. *Forward-chaining rules and object-oriented programming*

Indeed, forward-chaining rules may be considered a natural extension to ordinary object-oriented programming. The execution of an object-oriented program operates a series of updates of a certain set of instances which constitutes a model of the world. Much in the same way, rule-based programming (in its forward-chaining version) uniquely relies on repeatedly updating the fact base (this is no longer true for backward chaining). The previous equation "fact base = model of the world" links the two techniques. Their basic difference is only in the control structure, which is rigidly procedural (stack discipline) for object-oriented programming and non-deterministic for rules. Yelland [16] takes this as ground for abandoning rules in his Smalltalk-based knowledge representation system. Relying on the OPUS and NÉOPUS experiences, we feel on the contrary that this difference does not prevent rules and methods to smoothly cooperate, and that it is precisely what makes it worthwhile to introduce rules in an object-oriented environment.

Turning to classical object-oriented style (the so-called message passing) causes some trouble to the knowledge representation specialist (see, e.g. [9], [14]). Objects appear as closed entities which can be addressed only via the interface defined by their class. In particular, obtaining the logical value of their properties will necessarily involve the use of procedures that must be explicitly defined in their class, whereas in a frame-based model one has direct access to the slots, which are felt to carry semantic information. As Rechenmann observes [15], the encapsulation principle gets in the way by hiding the object structure (looked upon as mere implementation detail by software engineers) and hampers the declarative and explicative capacities of the system.

This problem is visible in at least two points in the NÉOPUS system and its appearance requires additional information from the programmer: (1) the need to declare explicitly which objects have significantly changed their state as a consequence of rule firing (the `modified` action, already present in OPUS) and (2) the impossibility to elicit the goal of a rule from its text, hence the need of an additional field in the rule format to provide this type of information (which is called here an *assertion*). These are clearly at variance from the principles of declarative knowledge representation. However, we feel that the benefit gained from potentially applying rules to the whole universe of object-oriented models created by object-oriented programmers does warrant these restrictions from the generally accepted principles.

2.3. *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. [10], [17].

From the point of view of knowledge representation, this attitude amounts to considering that any programmed application can be seen as the representation of a certain knowledge, albeit in a clumsy and opaque way. When the programming language is object-oriented, however, the categories used by the language (class, instance, inheritance) give a non trivial cognitive structure to the represented knowledge, so that rule-based deduction can be applied to it in a meaningful way (which would be hardly possible with procedural languages).

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).

2.4. *Rule bases as abstract classes*

The first question to be solved in our setting is the ontological status of rules and of rule bases. 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. The basic idea is to treat OPUS rules as Smalltalk methods.
As is well known, all entities manipulated by Smalltalk are considered as first-class objects. At the creation of the system, however, some initial objects must be read in. Those objects must necessarily possess a faithful textual representation. This is preeminently the case for classes : classes have a dual representation as text (source form) and as objects (instances of metaclasses). Methods also appear both as text (human-readable) and under compiled form. In both cases, the Smalltalk compiler creates the object representation from the text.

We follow a similar pattern to combine the textual nature of rules and their reification as objects. The process will be more complex and involve the notion of fireable rule and the operation not only of the compiler but also of the Rete network.

Rule bases may be viewed as analogous with classes, since they are primarily text, needing a compilation process to render them operative. They differ from classes in a fundamental way, for they are utterly incapable of having instances. However, Smalltalk practice makes great use of so called "abstract classes", which are not intended to create instances, but represent knowledge to be inherited by subclasses. Following Atkinson & Laursen, we

shall treat rule bases as abstract classes, subclasses of class `RuleBase`. As with all Smalltalk classes, we shall 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 the 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 (see section 3.2)

2.5. *Rule base inheritance*

Since rule bases are classes that necessitate a particular compilation process, the status of inheritance for those classes has to be defined. A particular inheritance scheme for rule bases has been developed (rule base inheritance or *RBI*) that transpose the intuition of inheritance as found in class-based languages, i.e. a restricted specialization mechanism, in the world of rules. This mechanism interprets the inheritance relation between rule bases as a particular *control strategy*, that consists in firing rules defined in the lowest base of the inheritance tree (see [12] for details). Similarly to class-based languages, rule bases have now two functions : a function as a set of rules, representing a certain knowledge, and a function as a sub-base generator specialized by subclassing.

3. Control and Control Objects

3.1. *Control of rule firing, conflict sets*

The control problem in forward-chaining systems is a direct consequence of the non-deterministic control structure: at each cycle, the system has to chose which of its potential actions it should perform. The main task of the inference engine is therefore to manage the set of these potential actions, referred to as the *conflict set*. Many systems have emphasized the need for explicit and separate representation of control (see e.g. [3], [5], [6]) or the reflexive aspect of meta level architecture [3]. Cohen & al [7] propose an explicit and powerful representation of control in terms of interacting "strategy frames", but the proposed formalism is not applicable in the context of object-oriented programming. Following this tradition of explicit and separate representation of control, we revisit the control problem in our object-oriented context.

3.2. *The control problem in the NÉOPUS framework*

Rules are subject to much more manipulation by a rule-based system than are methods by the runtime of an object-oriented language. Therefore the analogy method/rule cannot be followed too far. The main point is that the entities that are really manipulated are not the rules themselves, but their instances known as "fireable rules". These appear quite naturally as objects generated through the operation of the Rete network.

The Rete network as interpreted by Atkinson & Laursen bears some analogy with a blast furnace continuously transforming iron ore into metal. It has entry points labeled with the classes that appear in the various rules of rule base, and terminal nodes that are labeled with the rules. Objects from the context are fed into it, as soon as they are created or modified, through the entry points that correspond to their classes. They are packaged in specific objects called *tokens* that are then propagated through the network until they reach the final nodes. A token which reaches the terminal node labeled with rule R contains a complete instanciation for R's variables, yielding a fireable rule. Therefore the fireable rule is faithfully represented by the pair *(terminal node, token)*, which is a first-class object in an obvious manner. See [13] for details.
Accordingly, class `FireableRule` defines access methods that permit querying about either the rule itself (e.g. number of premisses) or the filtered

objects that make up the token, and the all-important method `fire`.

Since fireable rules are first class objects, the conflict set as a collection of fireable rules appears also as an object, instance of class `ConflictSet`. By adequately defining the behavior of this class, we can program several control strategies.
The final control act is to choose a fireable rule in the conflict set and to fire it. 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 (method called `proceduralEvaluate`) is easy to define in a procedural manner, since all the pertaining information is accessible from the rule base.

3.3 *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. As we shall see, 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.

3.4. *Dynamically constructed control objects*

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.

3.5. *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 (as seen in 3.3).

4. Metarules

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.

4.1. *Substitution - regression*

We propose an architecture for managing control objects that is completely substituted to the standard procedural activation (the method `proceduralEvaluate` seen in 3.2). 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.

For example, here is a metarule taken from the M&B problem, that generate sub goals. This metarule is defined in the metabase of the M&B rule base, and states that *if there is a goal for the monkey to hold a physical object, and that certain conditions are satisfied, then a new goal to be on the floor should be created.*

Goals are represented as assertions contained in the `stopCondition` attribute of evaluators.

```
holdObjectNotCeilingOn
 |EvaluatorSubGoaling e. Local s o |

 e status = #loop.
 e goalIsThat: {s isHolding: o}.
 o weight = #light.
 o isNotOn: #ceiling.
 s isNotOn: #floor.
 s isAt: o at.
actions
 |e2|
 e2 <- e newSon. "evaluator created"
 e2 stopCondition: {s isOn: #floor}.
 e2 go
```

Here is an other example of metarule that actually fires a rule from the rule base being activated. In the M&B problem, this is typically the case when there is a rule in the domain rule base whose `finalState` matches the `stopCondition` of an evaluator.

```
 satisfyGoal
|EvaluatorSubGoaling e. local cs rule|
  e status = #loop.
  cs = e conflictSet.
  e satisfied not.
  rule := cs ruleSatisfying:
               e stopCondition.
actions
  cs fire: rule.
  e status: #end.
  cs modified.
  e modified. e father modified
```

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.

It is important to note here that the activation of a rule base is totally separated from the activation of its metabase(s) from a Rete point of view. Each rule base has an associated Rete network which does not interfere with the metabase's network. Rete networks propagations can be seen as series of un-interrupted tides. A the end of each tide, the conflict set of the rule base is updated and the procedure that caused the tide is given back control. This procedure is either the procedural activation method (`proceduralEvaluate`) or the action part of a metarule (see e.g. metarule `satisfyGoal`). In this latter case, the rule firing will usually be followed by an other tide in the metabase's network, caused by a modified declaration (such as `conflictSet modified` in metarule `satisfyGoal`).

4.2. *Hierarchies of control objects and metabases*

The NÉOPUS experience in writing metabases convinced us that metabases share a lot of common behavior. Because writing metarules is not a trivial task, this common behavior is worth factoring. This the main motivation in using rule base inheritance for writing metabases. Each metabase is defined as a sub-base of an existing metabase. A root metabase, called `DefaultMeta` defines a standard control behavior whose effect is similar to that of the `proceduralEvaluate` method.

Control is therefore expressed as a double specification : structure is defined in the evaluator objects, and their management in the metabase. This double programming is effectively achieved by the support of class inheritance for evaluators, and rule base inheritance for metabases. Those two inheritance schemes are not necessarily parallel: evaluator classes may support various types of metabases.

4.3. *Taxonomies of metabases*

A taxonomy of metabases is built, starting from root base `DefaultMeta`. Metabases become more and more complex as the tree grows, and more and more application-dependent. A good example of the genericity of metabase can be found in [8], where the control of a real-time rule-based system is defined as a metabase with a five-level inheritance tree. The first three levels (starting from `DefaultMeta`) are general and application independent. They define a control strategy that fires rules in parallel, according to a dynamically constructed agenda. The last ones are application-dependent, and contain high priority metarules. The system works in a real time closed loop and monitors patient in intensive care units.
A yet more sophisticated example is an extension of NÉOPUS based on an novel architecture for explanation-based learning [18]. In this application,

evaluator objects are dynamically created to learn about the sequence of firings, and generate more efficient control metarules.

5. Conclusion

We have introduced the notion of control objects together with metarules. The specification of control in this architecture gives the rule base programmer all the power and capacities of abstraction of object-oriented programming, to specify independent and reusable control knowledge. This is achieved by proposing two inheritance schemes that combine elegantly with each other, and a library of general and ready-to-use control metabases. Our approach is validated by the use of the system in various real world applications.

6. 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] Nebel, B. *Reasoning and Revision in Hybrid Representation Systems*. Lecture Notes in AI 422, Springer (1990).

[10] 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).

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

[12] Pachet F. *Rule base inheritance*. Conference on Object-centered Representations, La grande Motte, France, June (1992).

[13] Pachet F. *Knowledge Representation with objects and rules: the NéOpus system.* PhD thesis, Paris VI University, Sept. (1292).

[14] Patel-Schneider, P. *Practical, Object-Based Knowledge Representation for Knowledge-Based systems*. Information Systems (Oxford), Vol 15, N. 1, pp 9-19, (1990).

[15] Rechenmann, F. *personal communication* (1992).

[16] Yelland, Ph. M. *Experimental Classification Facilities for Smalltalk*, Proc. of OOPSLA '92, p. 235-246, (1992).

[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).

[18] Zerr, F. *ARIANE, Integration of an explanation-based learning mechanism in an industrial expert system shell*. PhD thesis, Université Paris-Sud, (1992).