Rule Base Inheritance

François Pachet
LAFORIA

Adresse : LAFORIA, Institut Blaise Pascal
Boite 169, Université Paris VI, Tour 46-00,
4, Place Jussieu
75252, Paris Cedex 05, France
Téléphone: (33).1. 44.27.70.10
Fax : (33).1. 44.27.70.00
ε-mail: fdp@laforia.ibp.fr

This paper was published in the proceedings of the conference "Représentations par objets", La grande Motte, France, June 92

Catégorie : techniques

Abstract
We study the transposition of the inheritance mechanism of class-based languages in a forward-chaining rule-based environment. This mechanism allows to specify various levels of organization for a rule base, which represent a notion of generalization/specialization. We show that this mechanism has some interesting practical effects : a conceptual one (the vision of a rule base as a specialization of other rule bases), and in relation with control (the inheritance tree is associated to a particular control structure). We describe our implementation of the mechanism in the NéOpus system, which integrates first order forward-chaining rules within Smalltalk-80. We finally give some examples of applications.

Keywords : inheritance, rule-based programming, rule bases, control strategies

Résumé
Nous étudions la transposition du mécanisme d'héritage des langages de classes dans un environnement de règles d'inférences en chaînage avant. Ce mécanisme permet d'ajouter un niveau supplémentaire d'organisation pour les bases de règles de notre système qui représente la notion de généralisation/spécialisation. Nous montrons que ce mécanisme induit un double effet pratique : conceptuel (fabrication d'un arbre de bases de règles), et de contrôle (l'arbre d'héritage est une représentation implicite d'un certain type de contrôle). Nous décrivons notre implémentation de ce mécanisme dans le système NéOpus, combinant un mécanisme d'inférence d'ordre un en chaînage avant avec Smalltalk, et en décrivons quelques applications.

Mots-clés: héritage, programmation par règles, stratégies de contrôle

1. Introduction

Objects and rules

This paper is part of a general study of the integration of a rule-based mechanism in an object-oriented environment, and of the advantages and consequences of applying object-oriented notions to rule-based programming.
Rule-based programming and object-oriented programming have common characteristics. In both cases, programming consists in defining separately objects and operations having side-effects on these objects. This constitutes a natural parallel between a rule and a method. A next parallel is between a class, seen as a set of methods, and a rule base, which is essentially a set of rules. In our system (NéOpus, see below §3), rule bases are indeed implemented as abstract classes, and rules are implemented as methods for these classes.
Now, class inheritance is widely acknowledged to be a useful and powerful mechanism for organizing procedural code, by providing a generalization/ specialization mechanism, and increasing reusability and factorization of code.
The aim of this paper is to study the application of class inheritance to rule bases, considered as particular classes, in order to propose a new organization tool for rule bases. In the first part (§2) we propose a rule base inheritance mechanism and study its practical effects; then (§3) we describe an implementation of this mechanism in the NéOpus system, and finally (§4) we give some examples.

The context of our work

The origin of our work is the Opus system [Atkinson&Laursen]. This system introduces a rule-based mechanism in Smalltalk. Our system, called NéOpus, is based on the original description, and includes several extensions [Pachet 3]. The original paper already mentioned the idea of using class inheritance for rule bases (p. 37). However, no indication whatsoever was given about its mechanism nor its implementation. We present here a solution for this problem based on our experimentation with NéOpus.

2. On the notion of inheritance in a rule-based scheme

2.1. What is a rule

Our notion of a rule is the standard production rule (or, better, "conditional action"), which consists in a *name*, a *condition part* and an *action part*. The condition part is matched against *facts* of a separate so-called "*fact-base*". When there exists a set of facts that matches the condition part of a rule, the rule is eligible for firing. Such an instance of a rule is called a *fireable rule*.
Rules are fired according to a *forward chaining cycle*, which is defined by the rule interpreter. A general cycle of an abstract rule interpreter may be defined as a three-step procedure : 1/ Selection of all fireable rules 2/Choice of one rule 3/ Firing of the selected rule. This cycle is repeated until no rules are fireable (or a given goal is achieved).

2.2. Need for organizing rules

The most common means of organizing rules is to group them in *rule bases*, which represent the knowledge associated to a particular problem to solve. Rule bases may be considered as a first level of organization for rules. A lot of well known inference systems (Art [ART], OPS5 [Brownston]) do not propose any other organization mechanism for rules.
However this organization is insufficient when rule bases contain large numbers of rules [Chandrasekaran 1]. The usual ways of providing the user with better organization for rules consist in either adding sub-levels (like the notion of *task* in *SMECI* [Corby]; or *packs of rules* in *Essaim* [Alizon&Huet]), or higher levels (with the notion of *generic task* [Chandrasekaran 2], or *problem* in Essaim) to the rule base level. These mechanisms cope with the decomposition of knowledge into several components which represent distinct *conceptual* knowledge units. We thus refer to these types of organization as *conceptual organizations.*

These conceptual organizations for rules are very often related to the expression of sequentiality, via some sequencing mechanism. Indeed, expressing the sequentiality of operations is difficult in standard rule-based programming, because rules are not functionally related to each other : the sequence of firings is determined by the rule interpreter.
In these cases, the position of a rule within this organization gives an indication about when this rule should be fired. We will see (§3.2) how the class organization provided by the Smalltalk-80 environment (namely *categories* and *protocols*) may be turned into a sequential organization for rules.

However, no mechanism allows to represent various *degrees of generality* or to represent a notion of *specialization* between several groups of rules. In particular, it is not possible to easily factor rules that are common to several groups, or to define an abstract default behavior, and to refine/redefine it locally. Representing such notions as generality/default, is usually performed by *ad hoc* tools (pseudo meta-rules, numerical coefficients) or heavy logical mechanisms, such as default logic.

2.3. Using inheritance as a means of organizing rules

Inheritance in class-based languages, such as Simula or Smalltalk (referred to as *class inheritance*) constitutes indeed a classification scheme. This scheme is acknowledged to be useful, efficient, and has been applied successfully to a variety of programming languages. It does not, however, handle every kind of generalization/specialization classification (see the point of view notion in Rome [Carré&Geib]), and it is better defined as an *operational classification scheme*. Such as it is, it has become a major tool of object-oriented programming.

Class inheritance consists both in inheriting structure (*instance variables* in Smalltalk) and behavior (*methods*). In our case, the abstract classes that represent rule bases do not have instance variables [Pachet 1,4] so structure inheritance does not seem to be relevant for our purposes. We shall try only to transpose method inheritance in the rule world.
Let us first briefly recall the definition of method inheritance :

Method inheritance :

If `A` is defined as a subclass of `B`, the method inheritance will consist in *aggregating* methods of `B` and methods of its superclasses to methods of `A`.

An important characteristic of method inheritance is the overriding mechanism: if a method in `A` has the same name as a method in `B`, or in a superclass of `B`, then the method in `B` is overridden.

Using a vocabulary naturally transposed from the class world : subclass/subbase; superclass/superbase, we can now define the notion of *rule inheritance*, and state the principle of *rule base inheritance*.

Rule inheritance :
The definition of rule inheritance is transposed directly from the definition of method inheritance : If rule base `A` is defined as a subbase of rule base `B`, the rule inheritance will consist in aggregating all rules of `B`, and all rules of the superbases of `B`, to the rules of `A`.
The principle of rule base inheritance may now be stated as follows :

Principle of rule base inheritance (RBI) :

A rule base `A` defined as a subbase of rule base `B` will inherit rules of `B,` in the sense of rule inheritance.

But this inheritance mechanism once transposed in the rule world has two unexpected and important consequences that we will describe now : the lookup mechanism becomes a control structure and rule names become significant.

2.4. Consequences of rule base inheritance

2.4.1. Comparison between methods and rules

Rules share important features with methods. Both are textual entities that define operations on objects. Both are abstract entities that are instantiated at execution time. But they have also a number of intrinsic differences that will affect the transposition of the inheritance mechanism. In particular, the analogy between a rule and a method leads to a comparison between the interpretation of a message transmission and a cycle in a forward-chaining mechanism.

Methods are explicitly called via the message passing mechanism, which uses their name to access them. Therefore methods *names* (in Smalltalk : *selectors*) play a fundamental role in their interpretation by the system. Changing the name of a method has dramatic effects on the rest of the program. The name of a method is actually significant for two reasons : it is used by other methods of the program that call it, and it is the key of the overriding mechanism (a method in a subclass overrides a method with the same name in the superclass).

On the contrary, rules are not explicitly called, but are accessed via a pattern-matching procedure, followed by a conflict resolution phase.

The name of a rule plays only an secondary role (for convenient accessing via browsers for instance, or for explanation/debugging purposes) and is not used by the interpreter. Changing a rule's name does not affect the working system.
Whereas the decision to invoke a method is under full control of the programmer, the decision to fire a rule belongs to the rule interpreter (Figure 1).

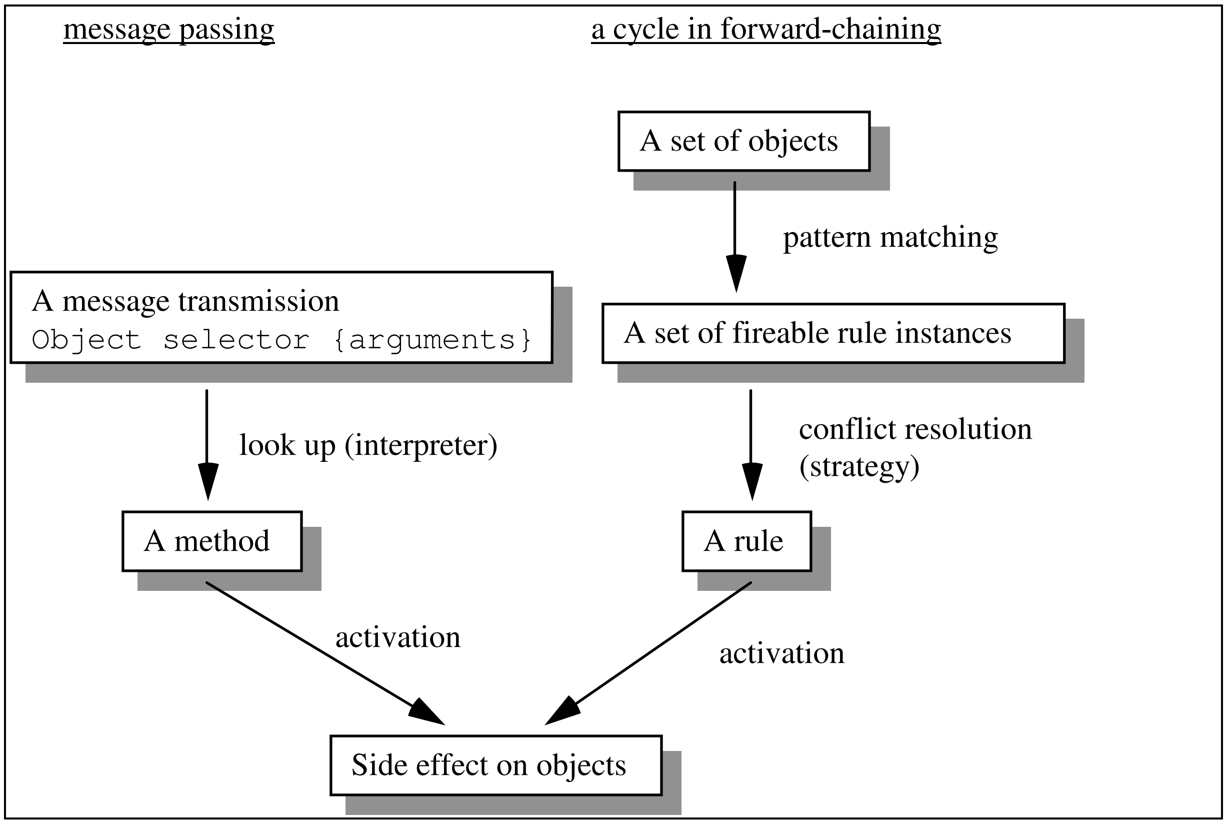

Figure 1. Comparison between a message transmission and a forward-chaining cycle

2.4.2. A simple control strategy

What is a control strategy ?
The rule-based mechanism in forward-chaining is intrinsically not deterministic: at each cycle, there may be several rules fireable, and the choice/order of firings is crucial. Since rule firings lead to side-effects on objects, firings are (very often) not reversible, so the choice of the rule to be fired is indeed the most delicate part of a rule-based system. A control strategy is a means of selecting one fireable rule. A default control strategy consists in selecting a rule at random, but in a number of cases more elaborate control strategies are needed to run the rule base correctly. A number of control strategies have been designed [Brownston, Chandrasekaran 1] to account for certain types of rule bases, but no absolute control strategy may exist, and sophisticated systems usually offer means of defining user-specific control strategies.

Now there is a *natural intention* behind the class inheritance mechanism, that we, as object-oriented programmers, would like to transpose to the rule world : creating a subclass carries the idea that the subclass is more specific. And it is indeed more specific, thanks to the lookup. In the rule world, since there is no lookup, the name-based lookup mechanism of class inheritance will find its equivalent in terms of a nameless *control strategy* that we define as the **RBI strategy** :

Definition :
The RBI strategy consists in selecting preferably rules defined in the *lowest subbase*.

An other statement of the RBI strategy consists in saying that a rule is fired only if there is no fireable rule implemented in a lower sub-base.

This is compatible with the notion of generalization/specialization, in the sense that rules defined lower in the hierarchy may be considered more specific, thus *preferable*, to rules defined higher.

2.4.3. Rule names become significant

An important consequence of method inheritance is to provide with the possibility of redefining (overriding) a method. Redefining a method consists simply in defining a method with the same name in a subclass. The mechanism of redefinition is actually performed by the lookup.
In the rule world, since there is no lookup, the question is : what does it mean to redefine a rule ? This question is not trivial because the very question of expressing similarities between two rules is complex.

Rule base inheritance gives us the opportunity of defining simply and operationally rule redefinition, by using rule's names :

Definition : rule redefinition
redefining a rule simply consists in defining a rule with the same name than a rule defined higher in the inheritance tree.

We want this redefinition to have the same effect than with methods : if a rule base `A` redefines a rule defined higher in the inheritance tree, the inherited rule is removed from the set of rules of `A`.

This definition has an important consequence : rule's name become significant with regards to redefinition : naming a rule is not just a handy way of accessing it, but its is also a way of overriding an unwanted inherited rule.

However, rule overriding must not be confused with *rule subsumption*. Overriding a rule in a subbase consists in replacing the inherited rule by an other one. The inherited rule does not need to actually subsume the redefined rule, in the sense of rule-based programming (i.e. "*rule `a` subsumes rule `b` iff any set of objects that matches `b` matches `a`*"). The rule redefinition mechanism which is induced by rule base inheritance is independent of the actual content of the rules.

Since we do not want the redefined rules to be *ever* fireable, this rule redefinition mechanism is not performed via a specific control strategy, but is implemented statically at the rule compilation time (see §3.4).

2.5. Practical features of rule base inheritance

There are a lot of practical effects of this mechanism. Organizing rules in a hierarchy of rule bases brings new perspectives to rule-based programming. We list some of them here.

2.5.1. A conceptual hierarchy

Classes in object-oriented settings have two functions : a function as an instance generator, and a function as a potential superclass. In the same fashion, these two functions of classes are now transposed for rule bases. Rule bases have now two roles in the environment : a role as a group of rules, representing an abstraction of some knowledge to be applied; and a role as a rule base to be specialized. This ability to represent various degrees of generality has an effect on the methodology of rule-writing : a rule base is considered not only as a group of cooperating rules, but as a specialization of some other rule base. This new role for rule bases is interesting for several reasons :

- In the process of writing a rule base, some significant intermediary levels may emerge. Inheritance may be seen as a methodological tools for building rule bases.

- A rule base may be constructed from an initial pre-existing rule base, thus improving reusability. Building pre-defined and reusable sets of rules is a well known problem of expert-system designers. Rule base inheritance gives a partial but operational solution. As in class inheritance, *abstract rule bases* (analogous to abstract classes) will define standard, general knowledge. Subbases will refine/override it.

2.5.2. A hidden, implicit control structure

The problem of separating the control structure from the rule definitions is recurrent in rule-based programming. But systems that are able to provide user-specific control strategies are rare, and often difficult to use.
The inheritance tree is indeed a (simple) way to specify rule sequencing, using the RBI strategy (§2.4.2). It represents a kind of "static preference" relation between rules. The position of a rule in the rule base tree indicates, in a non-numerical way, its degree of preference compared to rules in other levels. As such, this specification has some particular characteristics :

- It is a *static* notion.

The position of a rule in the hierarchy does not depend on the actual instantiation of the rule. There are cases when a rule may be considered more general than an other one *for a given set of instantiation*, but not with other ones. The inheritance mechanism cannot take this into account.

- It *simplifies* rule conflicts.

The fact of organizing rules in a hierarchy of rule bases solves a great deal of conflict cases, which otherwise should have required some elaborate conflict resolution strategy. In this respect, rule base inheritance is a means of simplifying the conflict resolution phase.
However, the strategy defined by the rule base hierarchy does not handle all conflict cases as seen above (§2.4.2). When several rules of the lowest levels are fireable, the choice has to be made according to others criteria. A solution consists in trying to create intermediary levels in the inheritance tree that will avoid such cases. This

solution is not always possible, essentially because the inheritance is static. But a better design of the hierarchy can substantially limit the number of such cases.

- It supports *mixing* with other strategies.

Nothing prevents the interpreter to use standard or non standard control strategies in cases of unresolved conflict. This has to be specified in the rule interpreter. Compared to the object world, there is increased flexibility : there are only few object-oriented languages in which the lookup mechanism of the interpreter supports mixing with other lookup mechanisms.

- It is *implicit*.

It is interesting to define a control strategy implicitly, if its definition is clear, which is the case here. The effort of explicitly describing the control of the reasoning is aimed at avoiding complex, intertwined and obscure control hacks that break the declarative aspect of the rules. Here, the inheritance tree, though imperfect, is a clear representation of control.

3. An implementation in NéOpus

3.1. Opus and NéOpus

NéOpus finds its origins in the Opus system [Atkinson&Laursen]. The Opus system integrates first-order forward chaining production rules in the Smalltalk-80 environment. Opus rules consists in a name (a symbol), a variable declaration part, in which the variables appearing in the rule are declared by their Smalltalk class, a condition part and an action part. The rule language is Smalltalk itself without limitation : any Smalltalk expression may be used in rule's condition part as well as action parts.

NéOpus extensions to Opus mainly consist in object-oriented notions transposed in the rule world, such as the rule base inheritance mechanism as seen here. Other extensions include variable typing (which integrate class inheritance in the typing of the first-order variables), so-called *triggering variables* (which take into account functional dependencies between objects in the rules), and a declarative architecture for control (§4.2 and [Pachet 2]), in which the control mechanism is expressed in terms of meta-rules.

3.2. Rule bases are classes

Rules are organized in rule bases, which are Smalltalk (abstract) classes. Rules appear as methods for these classes, in a specific browser, but are compiled by a particular Rete compiler for efficiency. These classes serve only as a support for organization, and for rule base inheritance.
Since rule bases are classes, they benefit from all the Smalltalk features for organizing classes. In Smalltalk, methods within a class are grouped in *protocols*, and classes themselves are grouped in *categories*. Categories and protocols are simply used in Smalltalk for reference purposes. They may be described as a primitive conceptual organization for Smalltalk code.

Now these organization levels find natural equivalents in terms of rule base *conceptual organization*. As seen in §2.1, it is interesting to use this organization to express sequentiality. Categories (of rule bases) and protocols (of rules) together with adequate control strategies, can indeed be used for expressing the sequentiality of rules (Cf. examples below §4.2 and [Pachet 2]).

The implementation of rule base inheritance is largely directed by the Rete compilation of rules that we will describe now.

3.3. Rule compilation

The main idea of the Rete compilation in Opus [Atkinson&Laursen] is to associate a Smalltalk method to every premise and to the conclusion part of an Opus rule. These methods are compiled in a separate class, called dynamic class, which is uniquely associated to each rule base.
Then Rete nodes are created for every premise of a rule, and a particular Rete node for its conclusion part. The network is used at activation time by propagating tokens, which represent sets of objects matching the corresponding premise of the rule. Empty tokens are sent initially to input nodes. When a token reaches a terminal node, the corresponding rule is added to a *conflict set*, and is ready to fire. Strategies for selecting the appropriate rule to fire are defined in the conflict set.
The initial Forgy's Rete network is also extended in order to take into account the substitution of OPS5 facts by real Smalltalk objects.

3.4. Implementation of the inheritance mechanism

The rule base inheritance mechanism is implemented in two steps : Rete network updating (which has to take into account rules coming from superbases) and proper conflict resolution strategy, defined in the conflict set.

3.4.1. A mixing of static/dynamic inheritance

Object-oriented languages usually separate *static inheritance* (decided at compilation time) used for instance variables, from *dynamic inheritance* (decided at execution time) used for methods.
Rules being paralleled to methods, dynamic inheritance comes first to mind. But the compilation of a rule leads to two different kinds of compilations : the compilation of the rule in the Rete network, and the compilation of the methods implementing the various premises and the action part in the dynamic class.
This leads to a combination of static (for Rete networks) and dynamic (for dynamic classes) inheritance. Since dynamic class inheritance is parallel to rule base inheritance (see Fig. 2), the inheritance of the methods implementing the rule in the dynamic class is the standard Smalltalk (dynamic) inheritance.
However, because of the nature of rule triggering (unlike methods, rules are not looked up), the updating of a Rete network is propagated down to the inheritance tree, at compilation time.
If we suppose a rule base `RB`, and a sub-base of `RB` called `RB2`, compiling a rule in `RB` will result in the compilation of Smalltalk methods in the dynamic class of `RB` (but

not in the dynamic class of RB2), and in the updating of both RB's Rete network, and RB2's Rete network (see Figure 2).

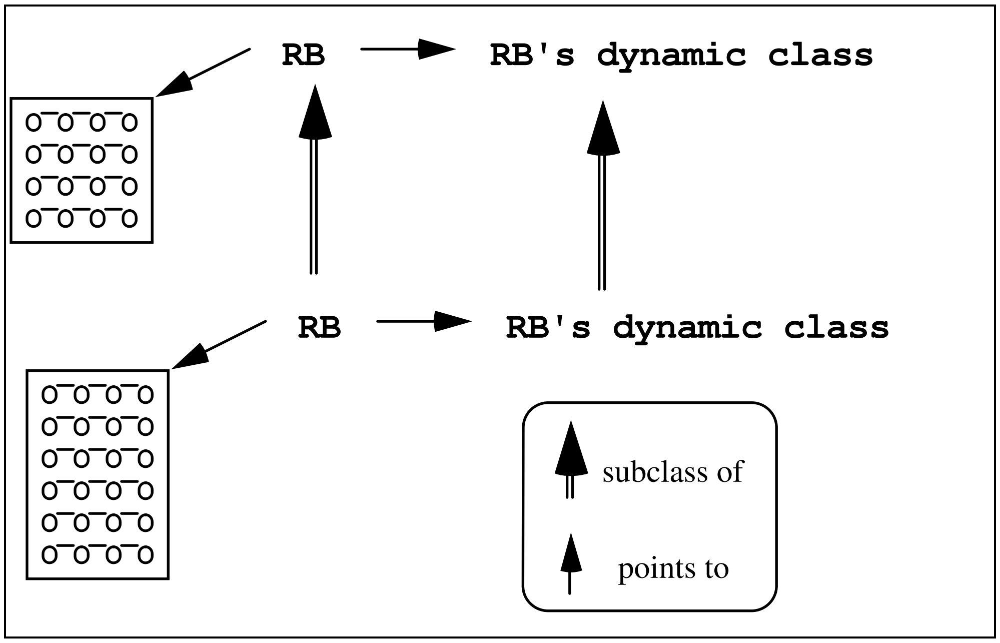

Figure 2. An implementation of the rule base inheritance scheme

3.4.2. Implementing the control strategy

Fireable rules are represented by a class implemented by a couple of instance variables : 'terminalReteNode token', where token is a token having passed successfully all the nodes of a rule, and terminalReteNode is the node associated to the action part of the rule. Once this class is defined, it is easy to represent conflict sets, simply as a class whose structure is essentially the list of fireable rules instances.

```
Object subclass: #OpusFireableRule
  instanceVariableNames: 'token terminalReteNode'

Object subclass: #OpusConflictSet
  instanceVariableNames: 'fireableRules'
```

When a rule is fireable, an instance of FireableRule is added to the list of fireable rules as defined by the method addFireableRule:. (This method is actually invoked when a token reaches a terminal nodes of the Rete network).

The default strategy is to fire the *first* rule of the conflict set, as defined in the method trigger. The first FireableRule of the list is chosen :

```
trigger
  self trigger: fireableRules first
```

Now, implementing the appropriate semantics for rule base inheritance consists simply in ensuring that the fireable rules are sorted in ascending order. The following method initializes the set of fireable rules in the conflict set with an instance of SortedCollection, as follows :

```
!OpusConflictSet methodsFor: 'initialize'!

initRules
      fireableRules := SortedCollection sortBlock:
                                  self inheritanceSortingBlock

inheritanceSortingBlock
      ^[:a :b |   a implementingRuleBase isSubBaseOf:
                  b implementingRuleBase]
```

The sort block takes two fireable rules as input, and uses the method `implementingRuleBase`, which yields the lowest rule base in which the rule is implemented.

4. Examples of rule base hierarchies

Here are two examples of applications of the rule base inheritance mechanism that show different aspects of its potential. The reader can fin more details in the references.

4.1. Representing default knowledge

Representing default knowledge is a recurrent issue of Artificial Intelligence systems. Rule base inheritance is an elegant approximation of a default knowledge mechanism. Moreover, rule base inheritance integrates well with the standard notion of class inheritance, because it allows the programmer to follow his natural intuition of inheritance, that consists in building *hierarchies* based on a reduced notion of specialization. Indeed, a standard use of RBI is to create sub-bases for knowledge concerned with sub-classes. If we assume a (general) class representing doors (say class `Door`), we can write a rule base concerned with doors in general (for instance, representing the knowledge associated to *closing* doors), `DoorsRules`. Now, if we refine the notion of doors by creating subclasses (e.g. `CarDoor`, `SlidingDoor`), the RBI mechanism leads us naturally to refine `DoorsRules` by creating sub-bases (`CarDoorRules`, `SlidingDoorRules`) associated respectively to the subclasses of `Door`.

4.2. Building predefined reusable rule bases.

A good example of a library of reusable rule bases is found in the declarative architecture for control of NéOpus. In this architecture, the control itself is defined in terms of *meta-rules*. These meta-rules are defined in rule bases called *meta-bases*. They define the control (or activation) of a rule base. The details of their implementation will not be described here. However, specifying control in terms of meta-rules is not indeed an easy task, and organizing these meta-rules is essential. But the interesting characteristics of these meta-bases is that they form a natural hierarchy, and share important parts. This can be easily taken into account by the rule base inheritance mechanism. A hierarchy of meta-bases was built according to the RBI mechanism. Each meta-base represents a particular type of evaluation. An example of the use and refinement of the hierarchy can be found in the NéoGanesh system [Dojat&Pachet]. This system was designed to control in real-time a ventilator providing respiratory help to patients in intensive care units.

5. Extensions

Several extensions to the rule base inheritance scheme as developed here may be considered. We indicate here the most important ones.

Multiple rule base inheritance

The first one is to extend simple inheritance to multiple inheritance. Although multiple inheritance is indeed a more powerful mechanism than simple inheritance, we did not want to follow this direction. It seemed to us that the main advantage of rule base inheritance is it simplicity. Rule base inheritance simplifies the problem of conflict resolution. Multiple rule base inheritance would complicate it, by adding the problem of solving multiple inheritance lookup conflicts.

Representing knowledge is indeed a very difficult task, and only simple and clear tools can be useful. Using multiple inheritance for rule bases would perhaps be useful in some cases, but would also introduce yet another level of complexity that we did not want to handle.

Other control strategies

The proposed control strategy is not necessarily the only one nor the best. An other one could be just the opposite : preferring rules defined in the highest superbase (following the intuition of inheritance as found in the Beta language). Or also trying to combine rules (thus establishing a parallel with method combination) may constitute interesting solutions. Our implementation has been designed so as to support various control strategies for rule base inheritance. However, the proposed one, because of its simplicity, seems to fit better than more complex ones.

Applying rule base inheritance in non object-oriented contexts

The rule base inheritance mechanism described here could be implemented in other rule-based environments. Our initial motivations were largely directed by the fact that in NéOpus, rule bases were implemented by classes. But the mechanism is general. In particular it could be applied to other inference systems (O or O+ order), in non-object-oriented environments.

6. Conclusion

We have described a transposition of the inheritance mechanism of object-oriented language in the world of rule bases. This mechanism allows to represent the notion of generality for rules, and has some interesting practical effects, such as the factorization of rules, the ability to redefine rules, and a simple and efficient control structure. We described an implementation of this mechanism in the NéOpus system, and showed practical applications. This mechanism has now proved to be useful for building rule bases. It may be considered as a methodological tool as well as a programming tool, and thus constitutes a major step towards *object-oriented rule-base programming*.

7. References

Alizon F. Huet G.

Essaim : un environnement de programmation Smalltalk destiné à la construction de systèmes experts. Note technique CNET NT/LAA/SLC/299, 1988.

ART

ART Reference Manual, v 3.0, January 1987, Inference Corporation.

Atkinson R.,Laursen J.

Opus : A Smalltalk Production System. OOPSLA '87 pp. 377-387.

Brownston L. & al.

Programming Expert Systems in OPS5. An Introduction to Rule-Based Programming. Addison-Wesley Publishing Company, 1985.

Carré B., Geib J.-M.

The point of view notion for multiple inheritance. Proceedings of OOPSLA'90, Ottawa, pp. 312-321.

Chandrasekaran B. (1)

Towards a Taxonomy of Problem-Solving Types. The AI Magazine, Winter/Spring 1983, pp 9-17.

Chandrasekaran B. (2)

Towards a functional architecture for intelligence based on generic information processing tasks. Proc. of the Tenth IJCAI, Milan (Italy), Vol. 2, 1987, pp 1183-1192.

Corby O.

BIB en SMECI. Proceedings of RFIA'87, Paris, pp. 581-586.

Dojat&Pachet

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

Forgy C. L.

Rete : A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence Vol 19 (1982) pp 17-37.

Goldberg A., Robson D.

Smalltalk-80 : The Language and its Implementation. Addison-Wesley, 1983.

Pachet F. (1)

NéOpus mode d'emploi. Rapport LAFORIA n° 14/91, Paris 1991.

Pachet F. (2)

Du bon usage des méta-règles en NéOpus. Rapport LAFORIA n°16/91, Paris 1991.

Pachet F. (3)

Reasoning with objects : the NéOpus environment, East EurOOpe, Bratislava, Septembre 91.

Pachet F. (4)

Représentation de connaissances par objets et règles. Thèse de l'Université Paris VI. Paris, 1992. A paraître.