Programming Environment for Expert System Shell

Wided LEJOUAD

SECOIA Project

INRIA/CERMICS Sophia-Antipolis

2004, Route des Lucioles

F-06561 Valbonne Cedex (France)

lejouad@mirsa.inria.fr

March 8, 1991

Abstract

The frequent use of object-oriented systems drove us to think about an interactive programming environment generator, able to deal with these systems thanks to specification languages. The first step of this project was to take an example of programming environment generator, such as **Centaur**, and to see its behaviour with the **Smeci** multiformalism expert system shell. Many problems have been studied for modeling class hierarchies and inheritance, type-checking rules with respect to class definitions. From this work, two editors are issued: one syntactic editor for the rule language and another for the class language, and a program that type-checks the rules in relation to class definitions.

Keywords: object-oriented systems, expert system shell, inheritance and programming environment.

1 Introduction

Current application diversity and complexity have driven computer research to turn towards a new programming style : object-oriented programming [MNC+89]. It makes flexible and easy the resolution of many problems in different computer fields.

Moreover, the frequent use of object-oriented systems has led researchers to think about an interactive programming environment generator, able to treat these systems thanks to specification languages, as it's the case with classic programming languages.

The considered interactive programming environment generator and object-oriented system are respectively Centaur and Smeci. So, we want to study the feasibility of the second one environment under the first one [Lej90].

2 Tool description

In this section, we describe the tools used to achieve this work. Our objective is to familiarize readers with some notion of expert system shell and interactive programming environment generator.

2.1 Smeci

Smeci [II89] is a shell which allows expert system development and adjustment. It is suited to the realization of expert systems in design, planning and complex diagnosis. It is based on three principles:

- The representation of factual knowledge in object form.
- The representation of deductive knowledge in rule form.
- The parallel management of many states.

An interactive graphic interface is available, it allows users to access, define, test and modify the expert system objects and rules. The knowledge representation formalisms are : classes, objects, methods, inference rules and tasks.

Smeci can be foreseen as a multiformalism programming environment, it enables user to define rules, classes, methods, objects, etc. We can also consider it as an object-oriented system integrating the notions of objects or instances, class hierarchies, inheritance rules, ...

2.2 Centaur

Centaur [Baa87] [INR89] allows to define formalisms in the form of concrete and abstract syntax specifications in Metal [KMM83], pretty-printer in PPML and natural semantics in Typol [Des88]. It is a generic interactive environment. When given the formal specification of a particular programming language -including syntax and semantics- it produces a language specific environment.

The specifications of concrete and abstract syntax are written in Metal. A Metal specification is a collection of grammar rules, with functions that specify what abstract syntax tree should be built. Pretty-printing of abstract trees is defined in the PPML formalism. A PPML specification is a collection of unparsing rules. The semantics is specified with a formalism called Natural Semantics in Typol. A Typol program is a collection of axioms and inference rules.

3 Why implementing Smeci in Centaur?

Dealing with Smeci languages: rule, class and method languages, consists in parsing, type-checking and compiling them. In Smeci, the parsing step has been performed in Yacc

[Joh79], and the type-checking one has been realized by message passing. All these works have been done "*by hand*". It's why this project has been proposed. After having dealt with Smeci languages, we were wondering if there shouldn't be an easier way to realize this work automatically by a well known tool.

Nowadays, there exists systems which generate interactive programming environments to define formalisms or languages. These systems provide environments in which a programmer can design, implement, edit, correct, test, parse, control, validate and maintain his programs. Examples of these systems: Centaur, Mentor and Cornell. Why do we not exploit them ?

4 Application overview

From Smeci system integrating task, rule base, class, object, method, slot, state tree and reasoning notions, we have only considered the rule and class languages simplified in their structure. This reduced system is called μ-Smeci.

A μ-Smeci rule is defined by its name, its declarations, its premisses and its conclusions. A μ-Smeci class or category is defined by its name, its super-class and its slot list which can be empty. A μ-Smeci slot is described by its name, its type and its value. A type system has been built for the category slots. The syntax and semantics of these formalisms have been defined. Thus, the μ-Smeci languages have been treated in Centaur.

5 Application examples

5.1 μ-Smeci rule

```
defrule   weather

let       x a thermometer
```

```
          y a season
if        degree.x = 35
then      nature.x = summer
endrule
```

Thermometer and season are two categories, the notation degree.x denotes an access to the degree slot of the x object, and nature.y denotes an access to the nature slot of the y object. defrule, let, if, then and endrule represent the μ-Smeci rule keywords.

5.2 μ-Smeci class

```
category  Port  of mother  Object :
          Setting : object Setting ;
          Dike : list object Dike ;
          Agitation : object Agitation ;
          length : interval 0 .. 100 ;
```

In Smeci, every class has only one super-class called mother category. A list of slots is given to show their syntax. The slot types are defined in the Metal specification, a slot can have no value.

5.3 μ-Smeci rule language in Metal

As it is explained above, a Metal specification for the μ-Smeci rule language RL, includes:

- A *concrete syntax* for RL:

  This consists in BNF-like rules which describe the textual representations of legal programs. These rules will be analysed by a parser generator (Yacc) to construct a parser for RL.

```
rules
   <defrule> ::= "defrule" <ident>
                 "let"     <dec_s>
                 "if"      <exp>
                 "then"    <conc_s>
                 "endrule" ;
```

- An *abstract syntax* for RL:

  The abstract syntax defines the structure of internal arborescent representations of programs. It is made of *operators* -corresponding to the nodes of the trees representing programs- and *phyla* or types of these operators.

```
abstract syntax
   defrule   ->   IDENT DEC_S EXP CONC_S ;


   IDENT     ::=  ident ;
   DEC_S     ::=  dec_s ;
   EXP       ::=  and  or  equal  access ;
   CONC_S    ::=  conc_s ;
```

- A collection of tree building functions:

  These functions are responsible of building trees when the BNF-like rule is encountered during parsing. They provide the connection between the concrete and the abstract syntax of RL.

```
defrule(<ident>,<dec_s>,<exp>,<conc_s>)
```

<ident>, <dec_s>, <exp> and <conc_s> are sub-trees of the tree labelled by the defrule operator.

5.4 Unparsing of μ-Smeci rule language in PPML

The developed Metal specification is used to define the μ-Smeci rule language syntax and to produce abstract trees. The PPML formalism will be used to associate concrete layout representation to abstract structures. So, a pretty-printing specification is a set of rules, where each rule expresses a translation of an abstract syntax tree into a pretty-printing language.

```
defrule(*ident, *decs, *exp, *concs) ->


          [<v> [<hv>  "defrule" *ident]
               [<hv>  "let"     *decs ]
               [<hv>  "if"      *exp  ]
               [<hv>  "then"    *concs]
               [<hv>  "endrule"       ]
          ];
```

Where the left-hand side matches any tree labelled by the defrule operator and having the same arity, while the right-hand side corresponds to a pretty-printing format specified in a *box language*. The same work has been done for the μ-Smeci class language: the syntax in Metal and the pretty-printer in PPML.

5.5 μ-Smeci class language in Metal

The same approach has been adopted to define the μ-Smeci class language in Centaur. So, we start by defining the concrete syntax specifications, then the abstract syntax ones. We give an example of the category definition:

```
rules

    (1)  <defcateg>  ::=  <categ_with_slots> ;
           <categ_with_slots>

    (2)  <defcateg>  ::=  <categ_simple> ;
           <categ_simple>

    (3)  <categ_with_slots> ::= "category" <ident>
                                 "of" "mother" <ident> ":"
                                 <slots_s> ";" ;
           <defcateg>(<ident>.1,<ident>.2,<slots_s>)

    (4)  <categ_simple>     ::= "category" <ident>
                                 "of" "mother" <ident> ";" ;
           <defcateg(<ident>.1,<ident>.2,slots_s-list(()))
```

In this example, we show the BNF-like rules required for defining the category syntax followed by the tree building functions. In the first and second rules, we express that a category may have an empty or full list of slots. In the third and fourth rules, we reveal the syntax of the syntax of the two cases, and we build the corresponding abstract trees.

The operator definitions and their declarations are written in the abstract syntax part of the Metal specification as follows:

```
abstract syntax

    defcateg   ->  IDENT IDENT SLOTS_S ;
    slots_s    ->  DEFSLOT * ... ;




    IDENT     ::=  ident ;
    SLOTS_S   ::=  slots_s ;
    DEFSLOT   ::=  defslot ;
```

The lowercase identifiers represent the operators or the abstract tree nodes, while the uppercase ones are the types or phyla of these operators. `defcateg` is a ternary node, accepting an `IDENT`ifier as a first and second son, and a list of `SLOTS_S` as a third son. `DEFSLOT * ...` denotes a star list which can have any number of elements including none. All these phyla must be respected in the semantics definition in Typol by the *judgement* notion, which gives the structure and the type of a sequent in an inference rule.

5.6 Unparsing of μ-Smeci class language in PPML

The abstract syntax trees built in the Metal specification must be unparsed in PPML. We give an example of pretty-printing a μ-Smeci class as it has been mentioned in the μ-Smeci rule language:

```
rules

  (1)  defcateg(*name,*mother,slots_s()) ->

          [<h> "category" *name "of mother" *mother ";"] ;


  (2)  defcateg(*name,*mother,*defslot) ->

          [<v> [<h> "category" *name "of mother" *mother ":"]
               [<hv> *defslot] ";" ]
```

The first rule gives the format of a category definition without slots and the second one gives the format of a full list of slots. The formats are given in *box language* where <h> denotes an horizontal separator, <v> a vertical separator and <hv> an horizontal separator with an automatic vertical folding applied when the horizontal composition does not fit into the width of the associated output device.

5.7 Type-check of the rule language in Typol

We present an example of inference rule developed to implement the type-check of μ-Smeci rules with respect to class language in Centaur. We have checked variable declarations, types, slot existence and assignments in a μ-Smeci rule in relation to category definitions. We have implemented a simple semantics of slot inheritance in the classes.

```
dec  (env_s[], envclasses |- decs  : envvar) &
prem (envvar , envclasse  |- pred  : type  ) &
conc (envvar , envclasse  |- concs : concs )
-----------------------
envclasse  |- defrule(id, decs, pred, concs) ;


                    .

                    .

                    .


set dec is
    <body>
end dec ;
```

In this example, we can notice the structure of a Typol *inference rule*. An inference rule has two parts: *premisses* and *conclusion*, separated by an horizontal line. A conclusion is a sequent and premisses are a list of sequents separated by ampersands. The rule explains when a μ-Smeci rule is well typed: if given an empty environment of declarations and an environment of classes (defined in Metal), the declarative part of the μ-Smeci rule produces a new environment called envvar. From this environment, we type-check the premisses and the conclusion. A **set** is a named collection of inference rules. This collection of rules is a complete formal system, it gives to Typol program the modularity characteristic.

Finally, it is important to notice that all the specification languages manipulated by Centaur have as a common feature the use of abstract synatx operators. It is the system responsibility to adapt them to different tools.

6 Studied problems

The definition of Smeci languages in different Centaur specification formats is feasible. But some difficulties have been faced in modeling class hierarchies, type-checking inference rules with respect to these classes and dealing with inheritance.

Simple inheritance in Smeci is developed in Typol in a static way: value, method and instance inheritance are not dealt with. In order to see Centaur's behaviour towards *multiple inheritance*, we have studied many systems which integrate it [FP90] [Dug86] [HO86] [BI82] [HI86]. After study, we were able to deduce a solution to deal with it by Typol inference rules: it's the **class precedence list** mechanism of Clos [Kee89] which allows to degrade multiple inheritance to simple one [DH88].

6.1 Class precedence list principle

Clos calculates the *class precedence list* of each class, it is a total ordering on the set of the given class and its superclasses. The total ordering is expressed as a list ordered from most specific to least specific. The element order in a class precedence list must respect the two following rules:

- *Each class has priority over their superclasses.*

- *The local precedence order, which is a list consisting of the class followed by its direct superclasses in the order mentioned in the defining form, must be respected.*

Clos starts by the class definition, applies the two rules and gets a set of constraints in a local level. From this step, it applies rules on the direct superclass definitions, then on their superclasses. This process is repeated until the root class is encountered. The result is an *ordering constraint set over the classes.* The next step consists in finding a total ordering on the class precedence list satisfying all the ordering constraints. Clos proceeds by *Topological Sorting* [BDG+88] of this set of constraints.

7 Evaluation

The possibility of programming a class precedence list algorithm in Typol, presents a solution for modeling multiple inheritance in Centaur but makes Typol losing its natural semantics implementation characteristic. So, we hardly express procedural programming concepts by a great number of declarative rules.

For simple inheritance in Smeci, we dealt with the slot inheritance of one class by its subclass, the disadvantage is that we have to compute inherited slot list at each time it is used.

From this work, a syntactic editor for rule language, a syntactic editor for class language and a type-checker ensuring the validity of rules in relation to class definitions, are issued. This result could make up a subset of Smeci system, in which we could edit and control programs. It can be illustrated in the following figure:

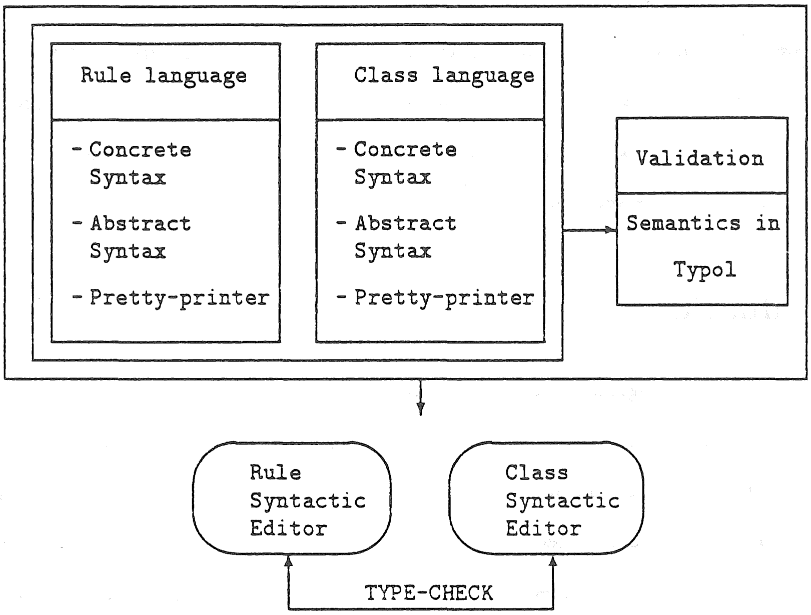

Figure 1: Application representation and issues

8 Conclusion

This work concerns the designers of multiformalism object-oriented systems in Artificial Intelligence or others. Its purposes are as follows:

- It's interesting to know the existence of tools able to realize many tasks in an easier and more flexible way.
- It's interesting to see the behaviour of Centaur, which is developed for classic language design, with some particular systems manipulating several languages at the same time and consisting of object-oriented programming notions: classes, instances and inheritance.
- Object-oriented programming tends to be more and more used, so we will need to design a lot of languages of this generation. Therefore, testing Centaur behaviour with these formalisms is interesting for object-oriented approach as well as artificial intelligence field which uses these new concepts.

Acknowledgements

This paper would not have been possible without the help of the Secoia team members led by B. Neveu and in particular the assistance of **Olivier Corby** enriched by numerous discussions and advice. I also want to acknowledge C. Fornarino for his useful remarks related to this work.

References

[Baa87] P. Borras and al. Centaur : the system. Rapport de recherche RR-777, INRIA, décembre 1987.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. *Common Lisp Object System Specification.* ISO/IEC, February 1988.

[BI82] A. H. Borning and D. H. H. Ingalls. Multiple inheritance in smalltalk-80. In *The National Conference on AI*, pages 234–237, Pittsburg, August 1982.

[Des88] T. Despeyroux. Typol: A formalism to implement natural semantics. Rapport de recherche 94, INRIA, Mars 1988.

[DH88] R. Ducournau and M. Habib. La multiplicité de l'héritage dans les langages à objets. Rapport de recherche 87/1, Université de Bretagne occidentale, Juin 1988.

[Dug86] P. Dugerdil. A propos des mécanismes d'héritage dans un langage orienté objet. *CIIAM*, pages 67–77, 1986.

[FP90] M. Fornarino and A.M. Pinna. *Un modèle objet logique et relationnel: le langage OTHELO.* PhD thesis, Université de Nice, Avril 1990.

[Hl86] J. Hendler. Enhancement for multiple-inheritance. *Sigplan Notices 21(10) ACM*, pages 98–107, October 1986.

[HO86] D. C. Halbert and P. D. O'Brien. Using types and inheritance in object-oriented languages. Rapport de recherche, Digital Equipment Corporation Object-Based Systems Group., Avril 1986.

[II89] INRIA and Ilog. *SMECI : Le manuel de référence.* Gentilly, 1989.

[INR89] INRIA. *The CENTAUR User's Manual.* Sophia-Antipolis, 1989.

[Joh79] S. Johnson. *YACC : Yet Another Compiler Compiler.* Bell Laboratories, 1979.

[Kee89] S. E. Keene. *Object-Oriented Programming in COMMON LISP.* Addison-Wesley Publishing Company, 1989.

[KMM83] G. Kahn, B. Mélèse, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming 3 North-Holland*, pages 151–188, 1983.

[Lej90] W. Lejouad. Environnement de programmation pour générateur de systèmes experts: Smeci sous centaur. Rapport de dea, INRIA - Sophia Antipolis, Septembre 1990.

[MNC+89] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Les langages à objets.* InterEditions, 1989.