Amalgamating Multiple Programming Paradigms in Prolog

Yoshiyuki Koseki

C&C Systems Research Laboratories
NEC CORPORATION
4-1-1 Miyazaki, Miyamae-ku
Kawasaki, 213 JAPAN

ABSTRACT

This paper discusses the issues in amalgamating multiple programming paradigms in the logic programming language, Prolog. It is shown that multiple paradigms can be incorporated without disturbing logic programming language features and efficiency. It also introduces a new programming paradigm called the *relation-oriented* paradigm. The research results are reflected in the implementation of the Prolog-based knowledge programming system PEACE, which is used to realize an expert system in a diagnostic domain. PEACE provides a *relation-oriented* programming paradigm, as well as previously discussed paradigms, such as *object-oriented, data-oriented,* and *rule-oriented* paradigms. These paradigms are nicely amalgamated in Prolog language and can be used intermixedly.

1. Introduction

Recently, several knowledge programming systems, such as [Bobrow83] have been proposed and implemented with multiple programming paradigms with the functional language Lisp. The paradigms supported in these systems so far are: *object-oriented, rule-oriented, data-oriented,* and *procedure-oriented* ones. In these systems, most inference knowledge is expressed in a rule-oriented style with an "if-then" structure, and the target structure is expressed in hierarchically organized objects (or frames). They provide an embedded uniform inference mechanism, such as a forward and backward reasoning engine to reason on the target objects.

These systems have two basic problems in representing real-world knowledge. The first drawback is from having a uniform inference mechanism. This is because a uniform mechanism prevents the utilization of multiple inference strategies in an application. The second problem is that it is hard to read a knowledge base written in this style, because the inference rules are described separately from the target object network.

To solve these problems, this paper proposes a semantic-network based knowledge programming system construction. In addition, it introduces another programming paradigm called the *relation-oriented* paradigm, based on a first order predicate logic language Prolog [Clocksin84].

Several kinds of knowledge, such as inference, inheritance and constraint propagation knowledge are considered to be pertaining to the *relations* among objects, not necessarily to the objects themselves. Therefore, it is natural for knowledge programming systems to have a capability of describing these kinds of knowledge in *relations* between target objects. With this capability, an entire knowledge base can be described in a semantic network, in which target knowledge is represented as object nodes and inference knowledge is represented as relations between nodes. By doing so, the problems with the conventional systems may be solved. First, it is possible to utilize multiple inference strategies in an application. Second, since the inference rules can be expressed in relations, inference knowledge and target object knowledge can be uniformly expressed in a network.

To implement a knowledge programming system with these features, the logic programming language Prolog is an appropriate base language. This is because Prolog is based on first-order predicate logic, and it is highly effective in representing knowledge in logical relations among objects.

However, it lacks the capability of expressing knowledge in object-oriented style. If it were possible to incorporate other programming paradigms, such as object-oriented and data-oriented ones in Prolog, a very efficient system can be obtained.

Recently, several schemes to achieve this incorporation in Prolog language have been proposed. One way is to build an entirely new system with object-oriented features. ESP is an example of this approach, designed for the PSI prolog machine at the Institute for New Computer Technology [Chikayama84]. It

incorporates Flavor-like [Weinreb81] object-oriented programming with a multiple inheritance mechanism. However, its object-oriented programming is "added on", not "amalgamated", since its object implementation is just mimicking the one on Lisp based tools. For example, object types in the system are restricted to *class* and *instance*, and it is not allowed to treat relations among objects as Prolog relations. Therefore, the relation-oriented programming style is not positively supported.

Another way is to express a semantic network in a set of Horn clauses and to use the Prolog interpreter as an inference mechanism [Koyama85]. Again, this performance is limited, because an entire network is represented with only one predicate, so that every clause matching might cause the entire network search in the worst case.

Alternatively, objects may be expressed in a nested list structure [Lee86]. But this method can not utilize Prolog expressiveness and is not efficient.

In this paper, multiple knowledge programming paradigms are amalgamated in Prolog by the technique called *meta-programming* [Bowen82]. Additionally, it describes a new implementation of the Prolog based knowledge programming system called PEACE (Prolog based Engineering AppliCations Environment). It is shown that PEACE efficiently supports a semantic network knowledge representation realizing relation-oriented programming as well as object-oriented programming, data-oriented programming and rule-oriented programming amalgamated in Prolog.

2. Semantic Network Representation in PEACE

The ultimate goal of the authors' project was to realize a system which behaves like human experts in a diagnostic domain [Koseki87]. To achieve this goal, it is necessary to represent various kinds of knowledge, such as design knowledge about the diagnosed equipment, and maintenance technician's empirical diagnostic knowledge. To investigate what types of knowledge representations are suitable in this domain, a prototype system was developed, using a rule-based technique similar to the one described in [Shortliffe76]. There, it was found that representing diagnostic knowledge entirely in a rule-form was not natural and feasible. It was considered appropriate to incorporate the structural information of the target equipment. To represent such a kind of knowledge, network oriented representation is suitable.

To describe network-shape knowledge representations, recent knowledge programming systems provides the object-oriented (or frame-based) paradigm. In those systems, a knowledge network is described in hierarchical trees of objects. The objects are usually classified into *class-objects*, and *instance-objects*, and they are connected by the system-defined *relations*, such as *class to instance* , and *super-class to class* relations. Through these relations, an inheritance mechanism is provided. Still, it is not felt natural to classify real-world knowledge into these fixed object categories and the fixed relation categories. To represent various kinds of knowledge, such as signal paths along functional blocks of the diagnosed equipment, a more flexible system is needed which can describe a more *flat* network.

To meet this goal, PEACE was developed to describe a semantic network on the first-order predicate logic language Prolog. It is basically different from conventional object-oriented (or frame-based) systems in three aspects.

First, there is no distinction among object types. That is, there is no distinction between *class-object* and *instance-object*, no distinction between *class-method* and *instance-method*, and so on. Instead, the system only provides a uniform object representation without any system-defined types. The role of an object is determined by how it is related to the other objects. For example, if an object is related to another object with *instance-of* relation, it is treated as an *instance* object, and the other object is treated as a *class* object.

Second, the user is allowed to define his own relation types in his problem domain and to add his own semantics to them. That is, relations can be defined freely with their own inheritance specifications and *inverse-relation* definitions. Moreover, since the relations between objects are internally represented in Prolog clauses, the user is allowed to express more complex relations by providing Prolog rules.

Third, since it is constructed on Prolog which has a powerful backtracking and unification mechanism, the multiple programming paradigms can also utilize these features. For instance, slots and relations defined in an object can have multiple values which can be enumerated by backtracking.

To show how multiple programming paradigms can be amalgamated in Prolog language, let us see example descriptions in PEACE. Figure 1 shows an example family network structure in a *semantic network* form. Each node corresponds to a physical object or concept. They are simply called *object*, because the system gives no distinction in regard to object types. Each arc between objects corresponds to a semantic relation

between them. For instance, this network shows the facts that "Charlotte *isa* female", "Charlotte and Shirley are *siblings*", and "Andy's *parent* is Shirley, and so is Wayne's".

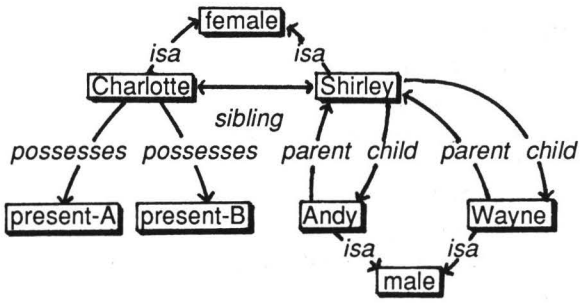

Fig. 1 An Example Family Structure

2.1. Object Description

In PEACE, a semantic network is represented as a network of *objects* connected by directed binary *relations*. An *object* is represented with its name and a set of Prolog clauses describing the contents in the format:

```
object-name ::
    relation-name # destination-object;
    slot-name : slot-value;
    slot-name :? slot-demon;
    predicate-name(arguments);
    predicate-name(arguments) ::- body.
```

Relation and slot can have basically similar effects, as long as both are used to point to another object. The difference is that relation can only point to objects and provide inheritance, automatic inverse-relation keeping, and user defined semantics as described in the following section (Relation oriented programming); whereas slot can contain any data, including complex Prolog structures and provide demon invocation as described in the other section (Data oriented programming). Note that a slot demon is defined separately and can be inherited through relations separately from slot value.

Predicates defined in an object can be treated declaratively or procedurely, as ordinal Prolog predicates can be. In other words, if they are interpreted declaratively, they can be considered to be facts and rules or axioms in an object's world. If they are interpreted procedurely, they can be considered equivalent to *methods* in the object-oriented languages, such as SMALLTALK-80 [Goldberg83].

2.2. Relation Description

Roles of a relation are described in an object *(relation-object)* with the relation name as the object name. For the example network in Fig. 1, the relation roles are described in Fig. 2.

```
:- parent ::
    $inverse(child).
:- child ::
    $inverse(parent).
:- sibling ::
    $inverse(sibling).
:- isa ::
    $inherit_pred(_);
    $inherit_slot_value(_);
    $inherit_slot_type(_).
```

Fig. 2 Relation Descriptions

The *parent* and *child* relation objects describe the relation that "parent" has *inverse-relation* "child" and the relation "child" has *inverse-relation* "parent". That is, when *'Andy'* has a parent *'Shirley'*, *'Shirley'* has a child *'Andy'* at the same time. This description is interpreted when a relation is added to a network, and the bi-directional relation is kept at all times.

Currently, the following four kinds of inheritance specifications can be set up in relation-objects, so that the inheritance can be restricted to certain predicate names, slot names, and relation names by specifying the names in the argument.

```
$inherit_pred(predicate-name),
$inherit_slot_value(slot-name),
$inherit_slot_type(slot-name),
$inherit_relation(relation-name).
```

For example, in Fig. 2, the relation *isa* has inheritance specifications which enable any predicates, any slot values, and any slot types to be inherited. The reason is that, in these inheritance specifications, "_" in an argument denotes an *anonymous* value, which matches anything.

3. Multiple Programming Paradigms

Based on the semantic network representation described in the previous section, the system can provide the multiple programming paradigms. These paradigms are: *object-oriented, relation-oriented, data-oriented, logic-oriented,* and *rule-oriented* ones.

3.1. Object Oriented Programming

Figure 3 shows the object descriptions for the family network in Fig. 1.

```
:- 'Charlotte' ::
      isa # female,
      possesses # present_A;
      possesses # present_B.

:- 'Shirley' ::
      isa # female;
      sibling # 'Charlotte'.

:- 'Andy' ::
      isa # male;
      parent # 'Shirley'.

:- 'Wayne' ::
      isa # male;
      parent # 'Shirley'.
```

Fig. 3 Object Descriptions

The symbol "#" denotes a relation. With these descriptions, a simple query like

?- 'Andy' <- parent # X. (Who is Andy's parent?)

gives an answer,

X = 'Shirley'.

You may look at the content of 'Shirley' object by typing:

?- listobj('Shirley').

And you can see that the system has created two child relations 'Andy' and 'Wayne' in 'Shirley' object. This is because the relation *child* is the inverse relation of *parent* as described in Fig. 2.

```
'Shirley' ::
      isa # female;
      sibling # 'Charlotte';
      child # 'Andy';
      child # 'Wayne'.
```

Therefore, a query like

?- 'Shirley' <- child # X. (Who is Shirley's child?)

gives an answer,

X = 'Andy';
X = 'Wayne'.

Note that the multiple answers could be obtained simply by causing backtracking (hitting semi-colon).

Next, let us describe objects for *female*, *male* and *human* object, in Fig. 4.

```
:- female ::
      isa # human;
      sex(female);
      height :? when_empty(160).
:- male ::
      isa # human;
      sex(male);
      height :? when_empty(175).
:- human ::
      disp_sex ::- origin <- sex(X), display(X), nl;
      weight :? (when_empty(X) ::- origin <- height : Y,
                  X is Y - 115).
```

Fig. 4 Objects with methods and demons

The predicate *sex* is an example of the declarative usage of predicates. The query,

?- 'Shirley' <- sex(X).

generates the answer

X = female

by the inheritance mechanism. When this query is invoked, the object interpreter (activated by "<-" operator) tries to satisfy the goal *sex(X)* predicate in the object *'Shirley'*. Since the interpreter can not find it, it looks for relation descriptions of *'Shirley'* and finds out the relation *isa* inherits the predicate *sex*, because *isa* object has $inherit_pred(_) which matches the goal $inherit_pred(sex). Then, it tries the goal *sex(X)* again in the *female* and succeeds in matching X with *female*.

The predicate *disp_sex* in *human* is an example of the procedural usage of predicates. It denotes a procedure (method) to print out the object's sex. The dummy object *origin* is used in it to point to the originating object of the inheritance chain. For example, the query,

?- 'Shirley' <- disp_sex.

prints out the answer:

female.

After climbing up the inheritance chain of *isa* relations, it tries the goal *disp_sex* in *human*. In *human*, because *origin* is *'Shirley'* in this case,

origin <- sex(X).

succeeds in matching X to *female*, and the word *female* is printed out.

3.2. Relation Oriented Programming

Since the directed binary relations between objects are internally represented as Prolog predicates, they can be used for *relation-oriented* programming.

A binary relation between objects:

```
'Andy' ::
    parent # 'Shirley'.
```

is internally represented as two Prolog facts, such as:

```
'Andy'(parent,'Shirley').
parent('Andy','Shirley').
```

When Andy's characteristics are requested, that is, when the relation name "R" is unknown in the query:

```
?- 'Andy' <- R # X.
```

the interpreter searches the first kind of internal facts with a goal 'Andy'(R,X), and gives the result:

```
R = parent
X = 'Shirley'.
```

In this way, the number of matchings is restricted to the number of relations in the object and the search through the entire objects can be avoided.

On the other hand, when the relation name is known, it searches the second kind of facts with a goal parent(X,Y). Again, in this way, the search through the entire Prolog data base is avoided and the number of matchings is restricted to the number of related pairs for the relation in the worst case.

To search a set of the objects, by which relations can be expressed logically, only the Prolog rules to express its logic are to be added. For example, to express relations *aunt* and *sister* in the example family network, you should add two rules:

```
aunt(X,Z) :- parent(X,Y),sister(Y,Z).
    (Aunts are parent's sisters)
sister(X,Y) :- sibling(X,Y), Y<-sex(female).
    (Sisters are any siblings who are female)
```

The query like,

```
?- 'Andy' <- aunt # X.
```

gives an answer,

```
X = 'Charlotte'.
```

Using this feature, we can easily define inference rules pertaining to a certain relation. For example, a rule "*All of aunts give presents to their nephews*" can be described as:

```
present_rule :-
    (aunt(X,Y),                    %for all of nephew-aunt pairs(X,Y)
    call((Y <- possesses # P,      %check if Y has a present P
        P <- instance_of # present,
        Y <- possesses #-= P,      %get the present P from Y
        X <- possesses #+= P,      %give the present P to X
        !)),
    fail;true).
```

The *call* operator is used to restrict the scope of the cut (!) operator. By executing this rule, Charlotte's presents are given to all of her nephews, Andy and Wayne. In this way, the relation-oriented programming can be accomplished nicely, in combination with the object-oriented programming.

3.3. Data Oriented Programming

Slots can have demons. The system provides several kinds of demons: *when_empty, referred, constrain, after_put, removed,* and *after_add*. In the example, a query,

```
?- 'Andy' <- height : X.
```

gives an answer,

```
X = 175
```

This is because *when_empty* demon in *male* worked since there was no *height* slot found in *'Andy'* and *isa* inherits any slot types. A query,

```
?- 'Andy' <- weight : X.
```

gives an answer,

```
X = 60
```

since *when_empty* demon in *human* calculates weight by subtracting 115 from its height as a default.

3.4. Logic Oriented Programming

Since the system is built on the logic programming language Prolog and preserves the features of the language, such as backtracking and a unification mechanism, all of the above mentioned programming paradigms can be incorporated in the logic programming style.

For example, the *setof* predicate can be used nicely with object oriented programming. The query to get all of Charlotte's nephews,

```
?- setof(X,(aunt(X,'Charlotte'),X<-sex(male)),S).
```

gives an answer,

S = ['Andy','Wayne'].

3.5. Rule Oriented Programming

The Prolog interpreter itself works as a backward chaining rule interpreter with a backtracking mechanism. To realize a forward chaining mechanism on Prolog is a relatively easy task. The simplest way to describe a rule with a Prolog clause is like:

```
fire_rule :- premise_1, premise_2,
            !,
            conclusion_1, conclusion_2.
```

But the rule control mechanism must be written by the user. PEACE provides a special rule interpreter to give more flexible control. It interprets a set of rules described in a *rule-object* which has the format:

```
rule_set_name ::
    $control(control_specification);
    if premise
    then conclusion;
    ...
```

The *premise* and the *conclusion* are described in PEACE and Prolog predicates. A Production system can be realized using the semantic network as a working memory. The provided control specification types are do_1, do_all, while_1, and while_all.

4. Implementation

PEACE is implemented on engineering workstations running standard Prolog interpreters and compilers, including C-Prolog [Pereira84] and others.

The interpreter is realized with the metaprogramming technique [Bowen82, Miyachi84], known as the *Prolog-in-Prolog* technique, which is to write the Prolog interpreter in Prolog itself. This method is also used in realizing metaProlog [Bowen85]. It gives great flexibility to the system implementation, but degrades execution performance. However, the degradation was permissible in building a diagnostic expert system [Koseki87].

Object descriptions are parsed when fed to the system and are stored as Prolog assertions. Slots, predicates, and relations pertaining to an object are internally represented as Prolog assertions with the same functor name as the object name, with different numbers of arguments. The multiple inheritance mechanism works interpretively so that dynamic additions of objects cause no troubles. Therefore, future efforts toward automatic knowledge acquisition may be relatively easy.

Since most of the Prolog compilers support the incremental compiling which enables it to compile a selected portion of a program, static objects which are never modified during execution can be compiled for speed up, without much effort to develop a special object-compiler. In addition, with the Prolog compiler's clause-indexing function [Bowen81], the time complexity for searching a predicate (including slot and relation) in an object may become constant.

A user-friendly interface is provided with a menu-driven and mouse-driven environment on commonly-used engineering workstations. Most of the basic operations including browsing the knowledge base network can be done by mouse operations. An example user interface screen is shown in Fig. 5.

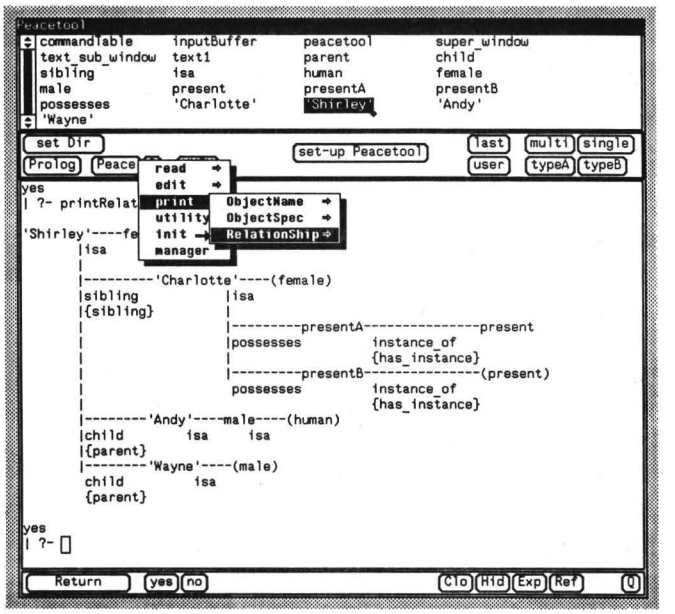

Fig. 5 User Interface Screen

5. Conclusion

A way to amalgamate multiple programming paradigms, such as relation-oriented, object-oriented, data-oriented and rule-oriented ones, was shown with examples on a Prolog-based knowledge programming system PEACE. By combining Prolog's logic programming capabilities, it was possible to achieve more flexibility in representing the real world knowledge than when using conventional knowledge representation systems.

The system has been successfully used in realizing

an expert system in a diagnostic domain and has proved to be effective in representing various kinds of knowledge, such as target equipment structure and the diagnosis technician's empirical knowledge. In particular, the relation-oriented programming technique was effective in representing symptom-hypothesis relations and the structure of the diagnosed equipment [Koseki87].

ACKNOWLEDGEMENTS

The author thanks Shin-ichi Wada, the co-worker on the diagnostic expert system and also thanks Nobuyasu Wakasugi, Masaki Kondo, and Mitsugu Oishi for developing PEACE. He also expresses deep appreciation to Hajimu Mori, and Satoshi Goto for continuous encouragement and support. Finally, he would like to thank Yasuo Iwashita of the Institute for New Generation Computer Technology for various kinds of support.

REFERENCES

[Bobrow83]
D. G. Bobrow and M. J. Stefik, "*The LOOPS Manual,*" Knowledge-based VLSI Design Group memo KB-VLSI-81-13, XEROX Corp, 1983.

[Bowen81]
D. L. Bowen, *DECsystem-10 PROLOG User's Manual,* Dept. of Artificial Intelligence, Univ. of Edinburgh, 1981.

[Bowen82]
K. A. Bowen and R. A. Kowalski, "Amalgamating Language and Metalanguage in Logic Programming," *Logic Programming,* Academic Press, 1982, pp. 153-172.

[Bowen85]
K. A. Bowen, "Meta-Level Programming and Knowledge Representation," *New Generation Computing, 3,* Ohmsha Ltd, 1985, pp. 359-383.

[Chikayama84]
T. Chikayama, S. Takagi, and K. Takei, "ESP - An Object Oriented Logic Programming Language," *ICOT Technical Report TM-0075,* Institute for New Generation Computer Technology, 1984.

[Clocksin84]
W. F. Clocksin and C. S. Mellish, *Programming in Prolog,* 2nd ed., Springer-Verlag, Berlin, 1984.

[Goldberg83]
A. Goldberg, and D. Robson, *SMALLTALK-80: The Language and its Implementation,* Addison-Wesley, 1983.

[Koseki87]
Y. Koseki, S. Wada, T. Nishida and H. Mori, "SHOOTX: A Multiple Knowledge Based Diagnosis Expert System for NEAX61 ESS," *Proc. of the International Switching Symposium 1987,* Phoenix, March 1987, pp. 78-82.

[Koyama85]
H. Koyama, H. Tanaka, "Definite Clause Knowledge Representation," *Proc. of the Logic Programming Conf. '85,* Tokyo, Japan, 1985, pp. 95-106.

[Lee86]
N. S. Lee, "Programming with P-shell," *IEEE Expert,* Summer, 1986, pp. 50-63.

[Miyachi84]
T. Miyachi, S. Kunifuji, H. Kitakami and K. Furukawa, "A Knowledge Assimilation Method for Logic Databases," *Proc. of the 1984 International Symp. on Logic Programming,* Atlantic City, 1984, pp. 118-125.

[Pereira84]
F. Pereira, *C-Prolog User's Manual Version 1.5,* Edinburgh Computer Aided Architectural Design, Feb. 1984.

[Shortliffe76]
E. J. Shortliffe, *Computer Based Medical Consultations: MYCIN,* Elsevier, New York, 1976.

[Weinreb81]
D. Weinreb and D. Moon, *LISP Machine Manual,* 4th ed., Symbolics, Inc., 1981.