TM-0055

Unique Features of ESP

by

Takashi Chikayama

April, 1984

Unique Features of ESP

Takashi Chikayama

ICOT
(Institute for New Generation Computer Technology)

Mita Kokusai Building, 21F.
4-28, Mita 1-Chome, Minato-ku, Tokyo 108 JAPAN

ABSTRACT

As the first major product of Japanese FGCS (Fifth Generation Computer Systems) project, the Personal Sequential Inference Machine called PSI or ψ is under development. The programming and the operating system of ψ is called SIMPOS and is intended to provide a comfortable logic program development environment. Here, we describe the unique features of the language called ESP, which is used in the development of SIMPOS and also is expected to be used in various application programs which will be developed on ψ in the following stages of the project. ESP is based on the Prolog-like machine language of ψ called KL0 (Kernel Language version 0). Thus, ESP naturally has many of the features available in Prolog. The most important ones among them are the unification mechanism for parameter passing and the depth first tree search mechanism by backtracking. The main features of the ESP language, except for those common in Prolog-like languages are: objects with time-dependent states, object classes and inheritance among them, and the flexible macro expansion mechanism.

1. Introduction

As the first major product of Japanese FGCS project, Personal Sequential Inference Machine (PSI or ψ) is under development [Uch 83]. SIMPOS is the programming and operating system of ψ [Tak 84]. The objective of SIMPOS is to provide a comfortable programming environment for logic programming which will be used in almost all the research areas of the FGCS project. Thus, SIMPOS is expected to be a truly usable system for users with various objectives. Thus, SIMPOS cannot but be a considerably large-scaled system. Besides, the first version of SIMPOS is required to be released at the end of the first stage of the FGCS project (March 1985) to be available in the following stages of the project.

If a certain standard abstraction method should not have been used throughout the system, the system would be over-complicated, and it would impossible to build up the system within the given rather short time period. To enforce a certain standard abstraction method, it is required, not if enough, to use a single language with appropriate abstraction capability throughout the whole system description.

ESP is primarily designed for this purpose, to enforce the object-oriented abstraction method. However, the design of ESP resulted in the language features appropriate for not only describing the operating system but also writing various application programs, especially for those requiring *knowledge* description. Objects of ESP is an axiom set. Sending a message to an object is trying to refute something using the axiom set. This mechanism matches very well with the model using semantic networks.

The sequential inference machine ψ, on which SIMPOS is built, has a Prolog-like high-level machine language called KL0. Thus, naturally, many features of ESP is that of a logic programming language. The *class* and *inheritance* mechanisms of ESP is built upon this inference mechanism of KL0. This is similar to the case of the Flavor system of MIT LISP Machine: ESP is to KL0 as Flavor is to LISP [Wei 81].

The notion of *time-dependent state* has also been introduced to ESP, based on the LISP-like features of KL0. Though this falls out of pure logic, it is required for utilizing the ideas widely used on various operating systems on conventional machines.

2. Time-Dependent States

Real programs must communicate with objects *outside* of the program, such as I/O devices, other computers connected *via* a computer network, the user at the terminal, *etc.*; otherwise, the user can never tell the machine to compute what she wants, and, even if she could, can never know the result. These outer objects may have time-dependent states which are interesting to the program. For example, it might be desirable for the program to know what kind of expression is *currently* appearing on the face of the user at the terminal to determine which of the available error message display styles irritates her the least.

The system must build up models of such outside objects inside the the computer. In pure logic programming style, such time dependency might be represented by logical relations between time periods and corresponding states. This relation itself is permanent and has no time-dependency. This may sound elegant, but is quite inefficient. The reason of this inefficiency is the fact that it is usually a little difficult to dispose of the part of the relation information which is no longer required by the program. The program will never want to know what kind of expression was appearing on the user's face at 3 o'clock the day before yesterday. Using a simple relation database management scheme, like those used in currently available Prolog implementations, this total recall ability not only requires almost infinitely large memory space, but also slows down the system considerably.

In conventional operating systems on conventional computers, the time of the outer world to be modeled is directly modeled by the time of the computation itself. Time-dependent states of the outer world are represented directly by the state of the computation. What is called *real-time* programming essentially means this modeling style. Thus, it is usually impossible to recall the state of the day before yesterday because the computation is being done *now*. This is profitable for saving memory space when the program never uses such information. Many of the ideas developed for operating systems of conventional computers are based on this programming style, including the efficiency consideration.

By applying a certain unknown optimization technique, keeping relations between time periods and states in the database might be made as efficient as this *real-time* programming style some day, but we didn't have time to wait for such an innovation. Thus, the notion of *time-dependent states* is introduced into ESP to facilitate directly utilizing such already available ideas.

3. Objects and Classes

An *object* in ESP represents an axiom set, which is basically the same concept as *worlds* in some Prolog systems [Can 82][Kah 83][Nak 83]. The same predicate call may have different semantics when applied in different axiom sets. The axiom set to be used in a call is specified by passing an object as the first argument of a call and prefixing the call with a colon, as in ": *open*(*Door*)". A predicate invoked this way is sometimes called a *method* as in other object oriented languages.

An object may have time-dependent state variables called *object slots* (slots are not *logical variables*; they have *constant* values from the logic programming view point). *Values* of slots can be examined using their names by certain predicates defined in the axiom set corresponding to the object. In other words, the slot values define a part of the axiom set. The slot values can also be *changed* by certain predicate calls. This corresponds to altering the axiom set represented by the object. This is similar to **assert** and **retract** of DEC-10 Prolog, though the way of alteration is quite limited. This limitation allows such *change* without too much runtime overhead.

An ESP program consists of one or more class definitions. An *object class*, or simply a *class*, defines the characteristics common in a group of *similar* objects, *i.e.*, objects which differ only in their slot values (only values; slot names are common to the objects belonging to the same class). An object belonging to a class is said to be an *instance* of that class. A class itself is also an object which represents a certain axiom set.

4. Inheritance Mechanism

Class Hierarchy

A multiple inheritance mechanism similar to that of the Flavor system [Wei 81] is provided in ESP. A class definition can have a nature definition, which defines one or more *super classes*. When one class is a *super class* of another class, all the axioms in the axiom set of the former class are also introduced into the axiom set of the latter class, as well as the original axioms given in the definition of the latter class. Note that this inheritance is determined statically at compilation time in ESP, while similar inheritance between *worlds* is determined dynamically at runtime in various systems with the *world* feature. This allows rather complicated inheritance rules stated below without introducing too much inefficiency.

Some of the super classes and the subclass which inherits them may have axioms for the same predicate. Since basically the axiom sets of the super classes are simply merged, such axioms are OR'ed together. Using this inheritance mechanism, a semantic network consisting of IS-A hierarchy can be very easily constructed. Though the order in the OR'ed axioms has no significance as long as pure logic is concerned, the order might be essential when things outside the computer should be treated. Thus, ESP allows the specification of the order of inheritance.

The PART-OF hierarchy can be implemented using IS-A hierarchy with the object slot feature. Assume that we want to make instances of the class *lock* to be a *part of* an instance of the class *door*. First, the definition of *door* should be given. Then, a class *with_a_lock* should be defined so that instances of the class *with_a_lock* has a slot which holds an instance of class *lock*. Finally, the class *door_with_a_lock* is defined to inheriting both the class *door* and the class *with_a_lock*: A *door_with_a_lock is* a *door* and also *is an* object *with_a_lock*.

Here, we have defined the class *with_a_lock* as a separate class rather than directly making the class *door_with_a_lock* inheriting the class *door*. This is the recommended way to fully utilize the multiple inheritance feature of ESP; the class *with_a_lock* may be used afterwards for defining classes such as *window_with_a_lock*.

Non-monotonicity

By only the inheritance scheme stated above, the axiom set of a subclass is bound to be a superset of those of its super classes. This monotonicity is often inconvenient in designing the class hierarchy. Program development will be far more easier if non-monotonic knowledge can be introduced. ESP provides two ways to introduce non-monotonicity.

One is a well-known way by using *cut* operation. The *cut* built-in predicate of KL0 has the ability to prune alternatives up to the specified predicate call nesting level. Using this *cut* along with *fail*, a control structure similar to that provided by *catch and throw* in certain LISP systems can be implemented. This control structure is indispensable to implement error handling mechanism required almost everywhere in the operating system.

The other way is by using *demons*. To explain how the *demon* feature of ESP works, we will give below a little more detailed description of how clauses given in the class definition of a class and in the definitions of its super classes are organized into one *method*.

The clauses are classified into three categories: *principal clauses*, *before demon clauses*, and *after demon clauses*. Demon clauses are distinguished by the qualifier **before** or **after** put before them. Principal clauses given in a class definition for the same predicate name and the same arity form a *principal predicate*, just as a set clauses form a predicate in ordinary Prolog systems. Similarly, before demon clauses form a *before demon predicate* and after demon clauses form an *after demon predicate*.

A *method* is implemented by a *method predicate*. The body of a method predicate consists of an AND combination of the following three:

- AND combination of calls of all the *before demon* predicates defined in templates of the inherited classes, in the order of inheritance.
- OR combination of calls of all the *principal* predicates defined in templates of the inherited classes, in the order of inheritance.

- AND combination of calls of all the *after demon* predicates defined in templates of the inherited classes, in the *reverse* order of inheritance. The order is reversed so that before and after demon predicates defined in various classes nest properly.

All of these calls shares the same arguments. One typical usage of before demons is to check out whether the object is in an appropriate state and whether the arguments given to a method call are also appropriate. A typical usage of after demons on this line is checking the *return* values. It is possible because, when a principal predicate *returns* some value, it is through unification of the variables in given arguments as is common in logic programming languages. As after demons receive the same arguments, they can be examined there.

For example, assume that the class *door_with_a_lock* should have the method *open* which only succeeds when the door is unlocked. We already have a class *door* which has the method *open*, but this always succeeds. We should define the class *with_a_lock* so that it has a before demon clause for *open* which checks out the status of the lock and succeeds only when it is unlocked. Now, inheriting two classes, we can define the desired class *door_with_a_lock*. In this case, the *open* method of the class *door_with_a_lock* will be something like:

$$open^{door\_with\_a\_lock}_{method} \text{ :- } open^{with\_a\_lock}_{before},\ open^{door}_{principal}.$$

This demon mechanism is used in various parts of SIMPOS. Especially, the window subsystem, one of the modules requiring the most complicated control, fully utilizes this mechanism. Without this kind of non-monotonic mechanism, the design of SIMPOS would have been much more complicated job.

5. Macros

Motivation

One of the most-heard-of complaints of the programmers using logic programming languages is that the languages basically do not allow functional notations except in certain special cases (like in arithmetical expressions in DEC-10 Prolog). For example, to pass the sum of X and Y as an argument of a predicate p, it is usually required to write a program as "$add(X, Y, Z),\ p(Z)$" or in a similar way. The motivation of introducing macro expansion feature of ESP is to allow functional notation such as "$p(X + Y)$", which is apparently more readable especially when the expression becomes a little longer. To merely solve this problem, schemes like proposed by [Egg 82] would have been enough. However, we sought for more general and flexible way.

Macros are for writing *meta* programs which specify that programs with so and so structures should be translated into such and such programs. One of the most crucial points in designing the macro expansion feature is choosing the *meta language* for this meta program. There are two commonly used language families in which macros are extensively used: LISP-like languages and assembly languages. Macros are by far usable in LISP than in assembly languages because the meta language is LISP itself in case of LISP, while, in assembly languages, the meta language is a utterly different language with specialized functions though it usually *looks* quite similar. This is because programs can be easily treated as data in LISP, while it is not possible in assembly language. From this view point, Prolog-like languages are similar to LISP: Programs can be treated as data. Moreover, with the built-in pattern matching and logical inference capabilities of the logic programming languages, definition of macros can be made more flexible than in LISP.

In various languages with the macro expansion capability, a macro invocation is simply replaced by its expanded form. Though this simple expand-and-replace type macro expansion mechanism may be powerful enough for LISP-like functional languages, it is never enough for a Prolog-like logic based language. For example, a macro which expands the goal "$p(a,\ f(X+Y))$" to a goal sequence "$add(X,\ Y,\ Z),\ p(a,\ f(Z))$" rather than to "$p(a,\ f(add(X,\ Y))$" cannot be defined with a simple expand-and-replace mechanism.

Expansion Mechanism

The full macro definition form of ESP is:

$$Pattern \Longrightarrow Expansion\ \textbf{when}\ Generator\ \textbf{where}\ Checker\ \text{:-}\ Condition.$$

The pattern which is unifiable with the *Pattern* is expanded to the *Expansion* if the *Condition* succeeds. At this time, the *Generator* and the *Checker* are also spliced into the expanded program at appropriate places: When a macro invocation appears in a body goal, the *Generator* is inserted *before*, and the *Checker* are appended *after* the goal including the macro invocation; when the invocation appears in the head, the *Generator* is appended at the end of the body and the *Checker* is inserted at the beginning of the body.

For example, in the macro definition:

$$X+Y \Longrightarrow Z\ \textbf{when}\ add(X,\ Y,\ Z)$$

"$X+Y$" is the *Pattern*, "Z" is the *Expansion*, and "$add(X,\ Y,\ Z)$" is the *Generator*. The *Checker* and the *Condition* are omitted in this example. This same definition can be used in two ways. The clause:

$$add(M, M+1).$$

is expanded into the clause:

$$add(M, N)\ \text{:-}\ add(M, 1, N).$$

while the body goal:

$$\ldots,\ p(M+1),\ \ldots$$

is expanded into a goal sequence:

$$\ldots,\ add(M, 1, N),\ p(N),\ \ldots.$$

Note that, in more complicated macro definitions, the *Condition* can be used not only for deciding whether the invocation pattern should be expanded or not, but also for computing a part (or whole) of the *Expansion* by writing variables in the *Expansion* and instantiating them in the *Condition*. Simple optimizations like computing values constant expressions in compilation time can also be achieved using this feature.

Modifications of the language details of ESP have been very frequently required for these several months for various reasons. However, the corresponding modifications of the compiler were quite easy. This was because many of the language features of ESP, including those requiring rather complicated compilation, are actually implemented using this macro expansion feature. Almost all of the modifications required rewriting of only a few lines of such macro definition code.

6. Implementation

Currently (April 1984), a cross compiler from ESP to KL0, the machine language of ψ, is available on a main-frame machine. Linking the object code with the small runtime support system written directly in KL0, the program can be executed on the ψ machines.

The implementation of the object oriented features is rather straightforward. An object is represented by a vector: Its first entry is the pointer to the table corresponding to the axiom set associated with the object; other entries are for storing object slot values. The table actually represented as a KL0 predicate and is called the *method table*. This vector is allocated in the heap area rather than in the stack area so that setting slot values as side-effects is possible.

Object-oriented method invocations are translated into calls to a runtime subroutine with two arguments: The method name atom and a vector of the original arguments. The runtime subroutine looks at the first argument, which is the vector representing the object, and then its first item, which is the method table. This method table is called with the given arguments. The clause of the method table with its first argument being the given method name is selected and the corresponding *method predicate* is called from its body.

Though this table look up works fairly efficiently by virtue of the built-in clause indexing mechanism of KL0, certain firmware supports for accelerating the execution further are planned. In most cases, some of the predicate calls appearing in this object-oriented invocation mechanism are redundant. For example, when a method consists of only one principal predicate, the method table may directly call the principal predicate. Compilation-time optimization of this type is also planned.

In the current implementation, object slots are accessed by their name atoms using the same table. In certain cases, by a simple optimization, slots can be accessed by their displacements rather than their names. This optimization is also planned.

7. Conclusion

ESP is the language used in the Programming/Operating System (SIMPOS) of the logic programming based inference machine (ψ) now under development in the first stage of the Japanese FGCS project. ESP has an object-oriented abstraction mechanism, in which each object corresponds to an axiom set. ESP also has a powerful macro expansion feature which allows functional notation in logic programs. These unique features has proved their merits in the earlier stages of the development of SIMPOS.

A cross compiler of ESP to the machine language of ψ is currently available. SIMPOS is being debugged using the compiler and its first version is expected to be completed at the end of the first stage of the FGCS project (March 1985).

REFERENCES

[Chi 84] T. Chikayama, ESP Reference Manual: ICOT Technical Report TR-044, 1984.

[Egg 82] P. R. Eggerd, D. V. Schorre: Logic Enhancement: A Method for Extending Logic Programming Languages. Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, 74-80, 1982.

[Kah 83] K. M. Kahn, M. Carlsson: LM-Prolog User Manual, Release 1.0. UPMAIL, Dept. of Computer Science, Uppsala University, 1983.

[Nak 83] H. Nakashima: A Knowledge Representation System: Prolog/KR. METR 83-5, Dept. of Math. Eng. and Inst. Phys., Univ. of Tokyo, 137, 1983.

[Tak 84] S. Takagi, T Yokoi, S. Uchida, T. Kurokawa, T. Hattori, T. Chikayama, K. Sakai, J. Tsuji: Overall Design of SIMPOS, to appear in Second International Logic Programming Conference, Uppsala, 1984.

[Uch 83] S. Uchida, M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa: Outline of the Personal Sequential Inference Machine PSI. New Generation Computing 1 No. 1, 75–79, 1983.

[Can 82] M. Van Caneghem: PROLOG II Manuel D'Utilisation, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, 1982.

[Wei 81] D. Weinreb, D. Moon: Lisp Machine Manual, 4th ed., Symbolics, Inc. 1981.