PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

UNIQUE FEATURES OF ESP

Takashi Chikayama

ICOT Research Center
Institute for New Generation Computer Technology
Tokyo, Japan

ABSTRACT

This paper briefly describes some of the unique features of the language ESP. ESP is the system description language of SIMPOS, the programming and operating system of the Personal Sequential Inference Machine ψ. ESP is based on the Prolog-like machine language of ψ called KL0 (Kernel Language version 0). Thus, ESP naturally inherits essential features of logic programming languages from KL0, the most important ones among them are the unification mechanism for parameter passing and the depth first tree search mechanism by backtracking. ESP is, at the same time, an object-oriented language with the notions of objects with time-dependent states, object classes and their hierarchical structure. ESP also has a very flexible macro expansion mechanism.

1 INTRODUCTION

1.1 Background

As one of the first major products of the FGCS (Fifth Generation Computer Systems) project, Personal Sequential Inference Machine (PSI or ψ) is now under development (Uchida et al. 1983). SIMPOS is the programming and operating system of ψ (Takagi et al. 1984). The objective of ψ and 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.

As SIMPOS is expected to be a truly usable system for users with various objectives, it 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 design, the system would be over-complicated, and it would be impossible to build it up 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 not only for describing the operating system but also for writing various application programs, especially for those requiring description of hierarchical knowledge structure.

1.2 Language Summary

The sequential inference machine ψ, on which SIMPOS is built, has a Prolog-like high-level machine language called KL0. ESP is translated into KL0 and then executed by the machine. All the features of ESP is thus built upon the features of KL0[1]. This is similar to the case of the Flavors system of MIT Lisp Machine (Weinreb and Moon 1981): ESP is to KL0 as Flavors is to Lisp.

ESP has logic programming features, object-oriented features and also macro expansion features.

1.2.1 Logic Programming Features

ESP inherits logic programming features of KL0. ESP uses parameter passing by unification and has the built-in depth-first AND-OR tree search mechanism by backtracking. Thus, any Prolog programs can be translated almost directly into ESP without any essential change.

1.2.2 Object-Oriented Features

An object of ESP represents an axiom set in logic programming view point. Sending a message to an object is trying to refute some proposition using the axiom set.

Objects belong to a *class*. An ESP program consists of one or more class definitions. A class may have one or more *superclasses*. The axiom set associated with an object is, basically, the *union* of the axioms defined in the class and the axioms defined in the the superclasses of the class. This inheritance mechanism matches very well with the semantic network model with *IS-A* hierarchy.

Besides its logic language features, KL0 also has Lisp-like features corresponding to *cons*, *rplaca*, etc.. The notion of *time-dependent state* has also been introduced to ESP, based on such LISP-like features of KL0. Though

[1] Though certain KL0 built-in predicates are planned to be implemented specially for ESP, they are utterly for improving the efficiency.

this falls out of pure logic, it is required for utilizing the ideas widely used on various operating systems on conventional machines, including efficiency considerations.

1.2.3 Macro Expansion Features

ESP also has a very flexible macro expansion mechanism. When macros are expanded, the invocation is replaced by the expanded form as with other languages with macro expansion capability. Besides, insertion of certain goals into the clause in which the invocation appears can be specified. The exact position of such insertions depends on where the macro invocation appears, especially, whether in the clause head or in the body goals. By virtue of this mechanism, functional notations such as arithmetical expressions can appear as arguments of either a clause head or a body goal.

2 TIME-DEPENDENT STATES

2.1 Modeling the Real World

Real programs must communicate with objects *outside* of the computer executing 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 he or she wants, and, even if she could, can never know the result.

Outer-world objects to be modeled 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 time-dependency of the outer-world objects inside the the computer.

2.2 Naive Implementation

In pure logic programming style, such time dependency might be represented by logical relations between time periods and corresponding states. Such relations themselves are permanent and have no time-dependency.

This non-time-dependent representation is logically elegant, but, unfortunately, its *naive* implementation will be 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 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 unbearably.

2.3 Real-Time Programming Style

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 time-dependent 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 *amnesia* is profitable for improving the efficiency of the program if such information will never be used. 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

3.1 Methods

An *object* in ESP represents an axiom set, which is basically the same concept as *worlds* in some Prolog systems (Van Caneghem 1982, Kahn and Carlsson 1983, Nakashima 1983). The same predicate call may have different semantics when applied in different axiom sets.

In case of the *world* mechanism, the axiom set to be used is determined by the execution context. In ESP, it is determined by the *object* passed as the first argument of the call. Such predicates with argument-dependent semantics are called *methods* as is in other object oriented languages[1].

Method calls are distinguished from calls to a built-in or a local predicate by prefixing the call with a colon, as in "*: open(Door)*". Here, the variable *Door* is supposed to have the value which is an object associated with the axiom set including one for the predicate predicate *open*.

3.2 Classes

An object belongs to an *object class*. An object class, or simply a class, defines the characteristics common in a group of *similar* objects. An object belonging to a class is said to be an *instance* of that class.

An ESP program consists of one or more class definitions. A class definition defines various attributes of the class, including axiom set associated with the instances of the class. A class itself is also considered to be an object which represents a certain axiom set, mainly concerning instance creation.

3.3 Object Slots

An object may have time-dependent state variables

[1] ESP also has predicates whose semantics doesn't depend on its first argument. Such predicates has the static scope of one class definition, and thus are called *local* predicates.

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 the method *get_slot* defined in the axiom set corresponding to the object. In other words, the slot values define a part of the axiom set associated with the object.

Instances of the same class has slots with the same names. However, their slot values can be different, while the rest of their associated axiom sets are identical.

The slot values can also be *altered* by the method *set_slot*. This corresponds to altering the axiom set represented by the object. This feature is similar to **assert** and **retract** of DEC-10 Prolog, though the way of alteration is quite limited. This limitation allows an efficient implementation of this feature. In DEC-10 Prolog, **assert** and **retract** are only possible for interpretive programs, which are much less efficient compared with compiled programs.

ESP provides short-hand notations[1] for manipulating object slots. For example, "*X*!*a*" is for getting the value of a slot *a* of *X* and "*X*!*a* := *V*" is for updating it.

4 INHERITANCE MECHANISM

4.1 Class Hierarchy

A multiple inheritance mechanism similar to that of the Flavors system (Weinreb and Moon 1981) is provided in ESP. A class definition can have a *nature* definition, which defines one or more *superclasses*. If a superclass has in turn a superclass, it is also a superclass of the original class. Thus, superclasses of a class forms a tree structure.

This class hierarchy and all the inheritance relations between classes are determined statically at compilation time in ESP, while similar inheritance between *worlds* is determined dynamically at runtime in various Prolog systems with the *world* feature. This allows rather complicated inheritance rules of ESP stated below without introducing too much inefficiency.

4.2 Inheritance of Methods

The axiom set associated with instances of a class is the union of the axioms defined in the class definition of the original class and those defined in its superclasses.

Some of the superclasses and the subclass which inherits them may have axioms for the same predicate. Since basically the axiom sets of the superclasses 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, which is the same as the order of trying axioms when several classes have axioms for the same predicate.

[1] Macros actually. See below for details.

4.3 Inheritance of Slots

Instances of a class have the slots defined in the class definition of the original class and the slots defined in its superclasses. However, if some of these slots have the same name, instances have only one slot with that name.

A PART-OF hierarchy can be implemented using IS-A hierarchy with the object slot feature. Assume that we want to make a lock to be a *part of* a door. First, the definition of the class of simple *doors* without any lock should be given. Then, a class *with_a_lock* should be defined so that its instances have a slot which holds an instance of class *lock*. Finally, the class *door_with_a_lock* is defined to be 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*.

5 NON-MONOTONICITY

5.1 Motivation

By only the inheritance mechanism stated above, the axiom set of a subclass is bound to be a superset of those of its superclasses. Thus, a predicate call successfully refuted in a superclass is bound to be successfully refuted also in the subclass. Inheritance mechanism having this nature is said to be *monotonic*.

Because of this monotonicity, *door_with_a_lock* cannot be a subclass of *door*. This is because an instance of the class *door_with_a_lock* cannot be opened when locked, while a *door* can always be opened. For utilizing the simple monotonic inheritance mechanism in this case, the class of simple doors *without* a lock should be a subclass of doors *with* a lock. The designer of the class hierarchy must have in mind when designing the class *door* that some day some one might want to use a door with a lock.

It is very difficult to predict all such extensions to the class hierarchy when designing only the top-most part of it, especially when the system to be designed is considerably large and complicated. Program development will be far more easier if non-monotonic knowledge can be inherited.

5.2 Cut

ESP provides two mechanisms for introducing non-monotonicity. One is a somewhat extended version of the well-known *cut* operation. The *cut* built-in predicate of KL0 has the ability to prune alternatives up to the specified predicate call nesting level.

One of the most important usage of *cut* is for *method overriding*, i.e. replacing the axiom defined in a superclass by one defined in its subclass. This can be effected by placing a *cut* in the axiom definition of the subclass

```
class bird has   nature animal;
instance
    :fly(Bird) :- ... ;
    ...
end.

class penguin has   nature bird;
instance
    :fly(Penguin) :- !, fail;
    ...
end.
```

Figure 1. Penguins Do Not Fly

which cuts up to an appropriate level so that the axiom of the superclass should be cut out. As this overriding is often required, cut symbols (!) appearing in the toplevel of method clauses[2] are automatically translated into such multiple-level cut instructions.

Negative knowledge can be implemented using overriding and explicit failure. For example, in Figure 1, birds fly, penguins are a bird, and yet, penguins do not fly.

Using multiple-level *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 for implementing error handling mechanism required almost everywhere in the operating system.

5.3 Demons

The other mechanism for introducing non-monotonicity 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 superclasses are organized into one *method*.

5.3.1 Details of Method Inheritance

Method clauses given in class definitions are classified into three categories: *principal clauses*, *before demon clauses*, and *after demon clauses*. Demon clauses are distinguished syntactically 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 of 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 the inherited classes, in the order of inheritance.
- OR combination of calls of all the *principal* predicates defined in the inherited classes, in the order of inheritance.
- AND combination of calls of all the *after demon* predicates defined in 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 share the same arguments.

[2] Principal clauses, actually. See below for details.

$$
\begin{aligned}
&method\_predicate(A_1, \ldots, A_m) :- \\
&\quad b_1(A_1, \ldots, A_m), \quad \ldots \quad, b_n(A_1, \ldots, A_m), \\
&\quad (p_1(A_1, \ldots, A_m); \quad \ldots \quad; p_n(A_1, \ldots, A_m)), \\
&\quad a_n(A_1, \ldots, A_m), \quad \ldots \quad, a_1(A_1, \ldots, A_m).
\end{aligned}
$$

Figure 2. Method Combination

```
class with_a_lock has
instance
    component lock is lock;
    before:open(Obj) :- .unlocked(Obj!lock);
    ...
end.

class door has
instance
    component state := closed;
    :open(Door) :- Door!state := open;
    ...
end.

class door_with_a_lock has
    nature door, with_a_lock;
end.
```

Figure 3. Door with a Lock

When a class has n superclasses (including the original class itself), and each $class_k$ has before demon, principal and after demon predicates b_k, p_k and a_k correspondingly for the same method, then the method predicate will be as shown in Figure 2.

5.3.2 Using Demons

Assume, for example, 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* (Figure 3). In this case, the method *open* of the class *door_with_a_lock* will be something like:

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

In general, one of the principal predicates does the main job for the method. In the above example, $open^{door}_{principal}$ plays this role. Before demons, if exist,

check out whether the object is in an appropriate state for receiving the message and whether the arguments given to a method call are also appropriate. In the example, $open^{with\_a\_lock}_{before}$ checks out the state of the lock of the door. After demons check out the *return values*. 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.

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.

6 MACROS

6.1 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 places (e.g. 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 such as "$add(X, Y, Z)$, $p(Z)$".

The motivation of introducing macro expansion feature to 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, simpler schemes such as one proposed by (Eggert and Schorre 1982) would have been enough. However, we sought for more general and flexible way.

6.2 Meta Language

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 widely used language families in which macros are extensively used: Lisp-like languages and assembly languages. Macros are by far easier to use and also more powerful in Lisp than in assembly languages. This is because the meta language is Lisp itself in the case of Lisp, while, in assembly languages, the meta language is essentially an utterly different language with specialized functions though it usually *looks* quite similar.

Meta language can be itself because programs can be easily treated as data in Lisp. From this view point, Prolog-like languages are similar to Lisp: Programs can be treated as data. Thus, we've chosen ESP as the meta language for ESP. With the built-in pattern matching and logical inference capabilities as a logic programming language, definitions of macros can be made even more flexible than in Lisp.

6.3 Expansion Mechanism

Pattern => *Expansion*
when *Generator* **where** *Checker*
:- *Condition*.

Figure 4. Macro Definition

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 might be powerful enough for Lisp-like functional languages, it is never enough for a Prolog-like 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.

The full macro definition format of ESP is as shown in Figure 4. The pattern which is unifiable with the *Pattern* is expanded to the *Expansion* if and only if the *Condition* succeeds. At this time, the *Generator* and the *Checker* are also spliced into the expanded program at *appropriate* places, that is:

- 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 of a clause, the *Generator* is appended at the *end* of the body and the *Checker* are inserted at the *beginning* of the body.

For example, in the macro definition:

$$X + Y \Rightarrow Z \text{ when } add(X, Y, Z)$$

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

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

is expanded into the clause:

$$increment(M, N) :- add(M, 1, N).$$

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 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 such as computing values of constant expressions in compilation time can also be achieved using this feature.

7 IMPLEMENTATION

7.1 Current Implementation

Currently (August 1984), a cross compiler from ESP to KL0 is available on a main-frame machine. Linking the object code with a 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 allocated in one of the heap areas[1]: Its first entry is a pointer to the object descriptor which consists of two table addresses. This descriptor is shared by the instances belonging to the same class. Thus, only one word per object is dedicated for the object-oriented calling mechanism. Other entries of the vector are for storing object slot values.

One of the tables pointed from the object descriptor is called the *method table*. The method table associates the predicate name atoms with the method predicates. Another table is called the *slot table* which associates the slot name atoms with the slot position offset inside the object. These tables are actually represented as a KL0 predicate in the current implementation.

Object-oriented method invocations are translated by the ESP–KL0 compiler into calls to a runtime subroutine with the method name atom and the original arguments as its arguments[2]. The runtime subroutine looks at the original first argument, which is the vector representing the object, and then its first item, which is the object descriptor, and then its first item again, which is the method table. This method table is looked up (*called*, actually) using the method name and the number of arguments as the key, obtaining a code object, which is the *method predicate*. Finally, this predicate is actually called with the original arguments using the higher order feature of KL0.

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, there is no need for the method predicate which implements the demon combination. Several compilation-time optimization of this type is made.

The efficiency of the current implementation of ESP, especially concerning its execution speed, is not quite satisfactory. Method calls are 3 to 4 times slower than calling KL0 predicate directly. Accessing slots has almost the same overhead. The sources of the overhead are the calling overhead of the runtime support subroutines and the overhead of method and slot table lookup.

7.2 Planned Improvements

Introducing several new built-in predicates of KL0 for reducing the overhead in the current implementation is being planned. The calls to the runtime subroutines will be replaced by built-in predicate calls and the function of the runtime subroutine will be realized by the firmware. The calling overhead of runtime subroutines will be replaced by almost negligible opcode fetch overhead. Slot accesses will also be compiled into built-in predicate calls.

Though the method table lookup mechanism in the current implementation works fairly efficiently by virtue of the built-in clause indexing mechanism of KL0, it can be further accelerated by the planned firmware support. In the firmware implementation, the tables will be implemented as hash tables rather than general KL0 predicates, which not only reduces the execution time but also the memory space required for storing the tables.

In the planned firmware implementation, method calls are expected to be roughly twice as fast as in the current implementation. Improvement expected in slot accesses is even more drastic, because it will not include any predicate call at all.

7.3 Further Optimization

It might also be effective for reducing table lookup overhead to make cache for several most recently called method names, their first arguments (i.e. receiver objects) and the corresponding predicate code addresses. However, to obtain reasonably high hit ratio, the size of the cache should not be too small, because it is not usually the case that the same method of the same object is called many times consecutively. Thus, for reducing lookup overhead for the cache, certain special hardware such as an associative memory system would be required.

8 EXPERIENCES

8.1 SIMPOS

Preliminary versions of the three lowermost layers of SIMPOS — called the *kernel*, the *supervisor* and the *I/O subsystems* — have been coded in ESP and are almost fully debugged on ψ. Improved versions of these modules and several other parts, mainly the programming system, of SIMPOS are also being debugged currently.

Through the development of SIMPOS, it was made clear that the multiple inheritance mechanism made sharing of codes quite easy. For example, functions of the module called *supervisor*, which provides various table structures, directory handling, process handling, stream-oriented inter-process communication, etc., could be directly used by higher level modules by only inheriting one or several of the classes defined there.

Revisions of program modules were also found to be quite easy. As modules of SIMPOS are separately developed even when they are closely related, facility of such revisions seemed to be one of the most crucial point in the SIMPOS development effort. In most cases, no changes at all is required in modules other than the revised one if its interface specification remains the same. At worst, what is required is only recompiling the classes inheriting the revised class.

[1] The logical memory space of ψ is divided into up to 256 areas and any of them can be freely allocated for heap and stack areas.

[2] This is actually effected by the macro expansion feature.

However, sometimes such mistakes as linking programs inheriting different versions of superclasses were made. In such cases it was a little hard to find what was wrong. Some sort of automatic management of program versions seems to be indispensable.

The macro expansion mechanism was profitable for improving the readability of the programs. Especially, allowing arithmetic expressions as arguments was most profitable.

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 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.

8.2 Other Applications

A natural language parser has been implemented in ESP (Miyoshi and Furukawa 1984). The hierarchical structure of the grammatical categories is mapped to the class hierarchy of ESP in this attempt, reducing the number of parameters of the parsing routine required otherwise.

Currently, a project of re-implementing another natural language parser BUP (Matsumoto et al. 1983) in ESP is also going on.

9 CONCLUSION

Though we only have had quite brief experience with ESP, its language features already have proved their merits. If SIMPOS were directly coded in KL0, the design and development of SIMPOS would have been much more toilsome job.

As inheritance relationship among classes is analyzed statically at compilation time, certain programming support for the management of program versions is indispensable. This function will be included in SIMPOS as the *library* feature.

The current implementation of ESP does not yet have the required efficiency, especially in its execution speed. By introducing several built-in predicates, it is expected to be greatly improved. Further improvement requires certain special hardware such as method table cache using associative memory, which is left to the future investigations.

ACKNOWLEDGMENTS

In the earliest design stage of ESP, a design task group was organized and the basic functional design of ESP was discussed. The group included the author, Toshio Yokoi, Takashi Hattori, Toshiaki Kurokawa, Norihiko Yoshida, Akihiko Konagaya, Hideo Shimazu and many others in the SIMPOS research and development group. Shigeyuki Takagi implemented KL0 compiler which compiles the object of the ESP compiler into binary format executable on ψ. The design of the firmware support for ESP was only possible with the help of Minoru Yokota and other members of the ψ firmware development group.

REFERENCES

Chikayama, T. ESP Reference Manual *ICOT Technical Report*, TR-044, 1984.

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

Kahn, K. M., Carlsson, M. *LM-Prolog User Manual Release 1.0* UPMAIL, Dept. of Computer Science, Uppsala University, 1983.

Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H., Yasukawa, H. BUP: A Bottom Up Parser Embedded in Prolog *New Generation Computing 1*, 2, 145–158, 1983.

Miyoshi, H., Furukawa, K. Object Oriented Parser in the Logic Programming Language ESP *International Workshop on Natural Language Understanding and Logic Programming*, Rennes, France, 1984, *to appear.*

Nakashima, H. A Knowledge Representation System Prolog/KR *Mathematical Engineering Technical Report*, METR 83-5, Dept. of Math. Eng. and Inst. Phys., Univ. of Tokyo, 1983.

Takagi, S., Yokoi, T., Uchida, S., Kurokawa, T., Hattori, T., Chikayama, T., Sakai, K., Tsuji, J. Overall Design of SIMPOS *Proceedings of the Second International Logic Programming Conference*, Uppsala, 1984.

Uchida, S., Yokota, M., Yamamoto, A., Taki, K., Nishikawa, H. Outline of the Personal Sequential Inference Machine PSI *New Generation Computing 1*, 1, 75–79, 1983.

Van Caneghem, M. *PROLOG II Manuel D'Utilisation* Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, 1982.

Weinreb, D., Moon, D. *Lisp Machine Manual* 4th ed., Symbolics, Inc. 1981.