TM-0075

ESP – An Object Oriented Logic Programming Language

by
Takashi Chikayama,
Shigeyuki Takagi, Kinji Takei

September, 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

ESP — An Object Oriented Logic Programming Language

Takashi Chikayama, Shigeyuki Takagi, Kinji Takei

Research Center

Institute for New Generation Computer Technology

ABSTRACT

The Japanese Fifth Generation Computer Project, begun in 1981, will span ten years. Its first three-year effort has been allocated to the development of basic concepts and technologies, as well as to the implementation of Project development tools. The major such tool is a superpersonal computer system, which constitutes the first-step toward an overall logic programming environment. This computer system is called SIM (Sequential Inference Machine), and consists of the minicomputer-scale PSI (Personal Sequential Inference Machine) and advanced software, called SIMPOS (SIM Programming and Operating System). SIMPOS is described in a logic programming language, ESP (Extended Self-contained Prolog). ESP is designed for ease of writing systems software and is based on the thesis that a hybrid of object-oriented programming and logic programming is a strong alternative to a language based upon either concept alone [Kahn 82]. This paper introduces the background and motivation for the design of ESP, and describes its main language features.

1. BACKGROUND

The aim of the Japanese Fifth Generation Computer Project is to create prototypes for the computers of the 1990s. To be sure, today's computers will perform better and have more functions in the future. However, the social needs for computing power in the next decade cannot be satisfied by such computers, hampered as they are by the processing bottlenecks and complexity-producing programming languages resulting from dependence on conventional von Neumann architecture. The Japanese plan is to remedy this shortcoming by the introduction of a highly-parallel architecture and a language oriented toward knowledge processing. The key concept is logic programming. We believe that this programming style holds the most promise for meeting the requirements of both parallelism and knowledge processing. One of the foundamental decisions of the project was to first design a basic programming language, then to invent an architecture that fits the language. We believe this to be the best approach to liberating computer languages and architectures from the constraints of von Neumann concept. 'It is very risky to change drastically the architecture and language at the same time. This means that machines yet to be built will run programs written in languages not yet tested' [Mosaic 84]. Therefore, it is desirable to proceed using step-wise refinement, that is, to repeat the trial-and-error cycles as many times as possible during the course of development.

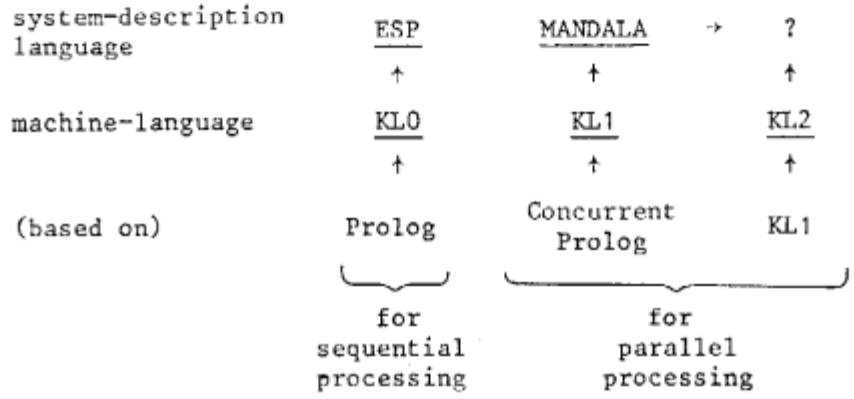
The Japanese project, begun in 1981, will span ten years, divided into three stages: the initial stage (three years), the intermediate stage (four years), and the final stage (three years). Each stage will constitute a trial-and-error cycle in which a basic language and its accompanying architecture will be developed. The initial-stage has concentrated on the development of basic concepts and technologies as well as the implementation of development tools that will be used in the subsequent stages. The major tool developed so far is a super-personal computer system, which constitutes the first-step toward the creation of an overall logic programming environment for the project. The computer system is called SIM (Sequential Inference Machine), and consists of a minicomputer-scale, von Neumann architecture-based machine called PSI (Personal Sequential Inference Machine), and advanced software called SIMPOS (SIM Programming and Operating System). SIMPOS is described in a logic programming language called ESP (Extended Self-contained Prolog), which is the subject of this paper.

The intermediate stage will be mainly devoted to improving and extending the results of the initial stage, and integrating them into inference and knowledge base subsystems. The research and development of this stage will focus on establishing computation models well suited to highly parallel processing and that will promote knowledge-based programming. The hardware subsystems to be built will consist of about a hundred processing elements.

The final stage will emphasize the optimization of both software and hardware system functions and the final determination of the architecture for a full-scale fifth generation system. This full-scale system will consist of about one thousand processing elements.

2. MOTIVATION

According to the project plan, three logic programming languages will be developed at the machine-language level, and two or three at the system-description-language level through the initial and intermediate stages, as shown in the following:



KL0 (Kernel Language version 0) is the machine-language of the sequential inference machine, PSI. It is based on Prolog with various extensions and some deletions. The main extensions are:

- Extended control structure
- Process switching
- Operations with side-effects, and
- Hardware-oriented operations.

The following features are included in the deletions:

- Database management, and
- Name-table management.

The features omitted in KL0 are supported in the system-description language, ESP. All the software for PSI is to be described in ESP.

While KL0 and ESP, as direct descendants of Prolog, are sequential languages, Concurrent Prolog's descendants, KL1, Mandala and KL2, are highly parallel languages. KL1 and Mandala are currently under development, the results of which will be applied to the KL2 design in the intermediate stage. Mandala is to KL1 what ESP is to KL0, and furthermore Mandala and ESP have many points in common. We expect the Fifth

Generation Project to have a substantial impact on the reduction of software development problems. The primary aim here is to facilitate the construction of large, complex, reliable programs that are executable in parallel, systematically managible, and practically verifiable. A tentative solution involves the amalgamation of hierarchical modular programming and logic programming.

ESP is designed for ease of writing systems software and is based on the thesis that 'a hybrid of object-oriented programming and logic programming is a strong alternative to a language based upon either concept alone' [Kahn 82]. The ESP Program is compiled into KL0 and executed on a PSI machine.

ESP plays a dual role, in that it is of both practical use for implementing the SIMPOS programming and operating system, and of experimental use for assessing the potential of logic programming in systems description. The direct motivation for introducing object-oriented programming features into ESP is that pure logic programming provides little support for program modularization, which is essential to systems programming. Though the desirability of supporting objects (or actors) in Prolog is well-recognized, much more experience with non-trivial programs is needed before questions regarding the most efficient implementation of such a hybrid can finally be settled. One such question, for example, is the debate over 'top-level implementation' vs. 'low-level implementation'. The SIMPOS implementation in ESP is a full-scale experiment in this context.

3. LANGUAGE FEATURES

ESP is a hybrid of a logic programming language and an object-oriented language. The logic programming features of ESP are almost directly derived from its base language, KL0, and are common to existing Prolog-like languages, with the exception of several extended features, particularly in built-in KL0 predicates. The main features of ESP, aside from those provided by KL0, include:

- Objects with states,
- Object classes and inheritance mechanisms,
- Macro expansion.

The assertion and atom name database features (i.e., assert, name, etc.) are not provided directly; instead, some lower-level provisions (e.g., array access and string manipulation functions) are taken into consideration for flexibility.

3.1 Objects and Classes

An object in ESP represents an axiom set, which is basically the same concept as that of "worlds (or mondes)" in some Prolog systems

[Caneghem 82]. The same predicate-call may have different interpretations when applied in different axiom sets. The axiom set to be used in a certain call can be specified by giving its (object) name as the first element in the argument list of the call. For example,

```
:creat (#process, Process)
```

The colon preceding the call specifies that the interpretation of the call will depend on the axiom set specified in the first argument.

An object may have time-dependent state variables called 'object slots'. Slot values can be examined by certain predicates using their names. In other words, the slot values define part of the axiom set. The slot values can also be changed by certain predicates. This corresponds to altering the axiom set represented by the object. This is similar to 'assert' and 'retract' in DEC-10 Prolog. However, only slot values can be changed in ESP, while any axiom can be altered in DEC-10 Prolog. As many currently available concepts and techniques for building operating systems are based on the notion of 'state' and 'state transition', much more investigation time would have been required if we had been forced to write it entirely in logic programming without the use of state-transition semantics (this approach has been investigated by E. Shapiro [Shapiro 84]. For this reason, the side-effect feature was introduced into ESP.

A class is a common description of the characteristics of similar objects, i.e., objects that differ only in their slot values. Every object within ESP is defined by a particular class and is called an instance of that class. A class itself is treated as an object representing a specific axiom set.

3.2 Inheritance Mechanism

The multiple inheritance mechanism provided in ESP is similar to that of the Flavors system [Weinreb 81], rather than to the single inheritance mechanism of Smalltalk-80 [Goldberg 83]. An ESP program consists of one or more class definitions. We can define a class as follows:

```
class <class name>
    [<macro bank definition>]

has
    [<nature definition>;]
    {<class slot definition>;}
    {<class clause definition>;}

[instances
    {<instance slot definition>;}
    {<instance clause definition>;}]

[local
    {<local clause definition>;}]
```

The 'macro bank definition' is discussed in the next section. The 'nature definition' defines one or more super classes as follows:

nature <super class name> {,<super class name>}

When one class is a super class of another, all the axioms in the former's axiom set are inherited by the latter's. By this inheritance mechanism, classes organize a hierarchical network of is-a relations. The currently defined class and some of its super classes may have axioms with the same predicate name. The axiom sets of the super classes are simply merged, and consequently ORed, in a descendent class. The sequential order of ORed axioms can be specified in ESP to optimize program execution and to control cuts and side-effects.

Demon clauses define demon predicates, which are not ORed but ANDed, either before or after the disjunction of normal axioms. Demon clauses are used to add axioms non-monotonically. For example, a door with a lock has a demon for the predicate open that checks whether it has already been unlocked before the open-action takes place.

```
% Simple Door
class door has
instance
    component
        state := closed;          % Open or closed
    :closed(Door) :-              % Is closed?
        Door!state = closed;
    :open(Door) :-                % Opening
        Door!state := open;
    :close(Door) :-               % Closing
        Door!state := closed;
    :make_way(Door) :-            % If already open,
        Door!state = open, !;     %   do nothing.
    :make_way(Door) :-            % If not,
        :open(Door);              %   then open it.
end.

% Lock
class lock has
instance
    component
        state := unlocked;        % Locked or unlocked
    :locked(Lock) :-              % Is locked?
        Lock!state = locked;
    :lock(Lock) :-                % Locking
        Lock!state := locked;
    :unlock(Lock) :-              % Unlocking
        Lock!state := unlocked;
end.
```

```
% With Lock -- MIXIN
class with_a_lock has
instance
    attribute
        lock is lock;
    before:open(Obj) :-          % Must be unlocked
        :unlocked(Obj!lock);     %   before opened.
    :lock(Obj) :-                % Locking object is
        :lock(Obj!lock);         %   locking the lock.
    :unlock(Obj) :-              % Unlocking object is
        :unlock(Obj!lock);       %   unlocking the lock.
    :locked(Obj) :-              % Object is locked when
        :locked(Obj!locked);     %   the lock is locked.
end.

% Door with Lock(s)
class door_with_a_lock has
    nature
        door,                    % A door
        with_a_lock;             %   with a lock
end.
```

In this way, the class with_a_lock can be defined separately from the class door, as such a class contains knowledge with non-monotonicity.

A part-of hierarchy can also be implemented using the is-a hierarchy and object slots. Assume that we want to make instances of class A part-of an instance of class B. First, A must be defined. Then, a class with_A must be defined so that instances of the class with_A have a number of slots, one of which holds an instance of class A. Finally, class B is defined to be a subclass of the class with_A; in other words, the class B is-a class with_A.

Returning to the definition of a class, there are two kinds of slots and clauses: those for the class itself (class slots and class clauses), and those for each instances of the class (instances slots and instance clauses).

Finally, clause definitions are used for defining Prolog-like clauses. From a declarative point of view, a clause expresses an axiom in the form of a Horn clauses. From a procedural point of view, a clause specifies procedural steps to be taken when a predicate is called. In addition to the class and instance clauses described above, there are also local clauses, which define non-object-oriented local predicates.

Predicates, with the exception of local predicates, are called using methods, rather than directly. A method is a certain AND-OR combination of predicates defined in a class or its super classes.

3.3 Macros

Macros are implemented for writing meta programs to specify how a program embedded with macro forms is to be translated into an executable program. A macro can be defined in the form of an ESP program, taking full advantage of the pattern matching and logical inference capabilities of logic programming.

In various languages having macro expansion capabilities, a macro invocation is simply replaced by its expanded form. Though this simple expansion mechanism may be powerful enough for LISP-like functional languages, it is inadequate for a Prolog-like logic language. For example, a macro which expands p(a, f(X + Y)) to a sequence add (X, Y, Z), p(a, f(Z)) cannot be defined using a simple mechanism. ESP macros are not only expanded at the point of macro invocation, but additional goal can also be spliced in before or after the goal in which the macro is invoked. If the macro is invoked in the head, these goals will be prefixed or postfixed to the body clause. The macro definition; 'X + Y ⇒ Z when add (X, Y, Z)', can be used in two ways. The clause 'add1 (M, M + 1)' is expanded to the clause 'add1 (M, N):- add (M, 1, N)', while the body goal 'p (M + 1)' is expanded to the goal sequence 'add (M, 1, N), p (N)'.

4. IMPLEMENTATION

The implementation of the object-oriented features is 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 object slot values. The table represented as a KL0 predicate is called a method table. This vector is allocated in the heap area, rather than in the stack area, so that slot values can be set as side-effects.

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

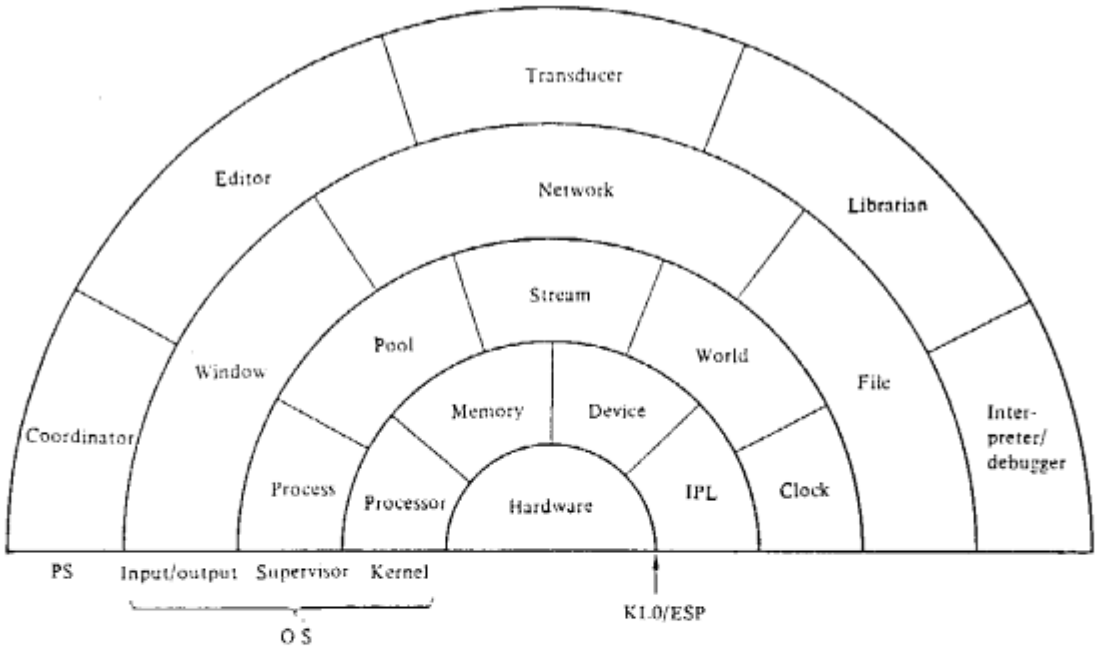
Though this table look-up works fairly efficiently by virtue of the built-in KL0 clause indexing mechanism, additional firmware supports are planned to further increase execution speed. Some of the predicate calls appearing in this object-oriented invocation mechanism are redundant.

For example, when a method consists of only one principal predicate, the method table may call the principal predicate directly. Compilation-time optimization is also planned to reduce this redundancy.

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 by their names. This optimization is also planned.

5. CONCLUDING REMARKS

As shown in the figure below, the SIMPOS system has a hierarchical structure consisting of four layers. The layers, in order of their proximity to the hardware, are the resource management program (Kernel), the execution management program (supervisor), the I/O system, and the programming system.



SIMPOS system configuration

Each layer consists of several subsystems or modules. The functional contents of the subsystems or modules are similar to those of an advanced programming and operating system such as the LISP Machine Software Environment.

The first version of the SIMPOS system is now being debugged. Almost the entire system is described in ESP. Exceptions are a few number of short runtime-subroutines. It is too early to assess performance. However, it may be said, based on our experience in writing a full-scale system, that ESP has sufficient expressive power for systems programming.

REFERENCE

[Caneghem 82] Van Caneghem, M.: PROLOG II Manual D'utilisation, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseille (1982).

[Goldberg 83] Goldberg, A., Robson, D.: SMALLTALK-80-The Language and its Implementation. Xerox Palo Alto Research Center (1983).

[Kahn 82] Kahn K.M.: Intermission-Actors in Prolog, K.L. Clark & S.-A. Tarnlund(ed.) Logic Programming, pp.213 - 228 (1982).

[Mosaic 84] Metzger, N.: To Find the Way Forward, Mosaic, Vol. 15, No. 1, pp. 2 - 7 (1984).

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

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

[Chikayama 84b] Chikayama, T.: KL0 Reference Manual, ICOT Technical Report, (to appear).

[Takagi 84] Takagi, S. et al.: Overall Design of SIMPOS (Sequential Inference Machine Programming and Operating System), TR-057 (1984).