TR-285

Programming in ESP
– Experiences with SIMPOS –

by
T. Chikayama

June, 1987

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

Programming in ESP
— Experiences with SIMPOS —

Takashi Chikayama
ICOT

Abstract

SIMPOS is the programming and operating system for the sequential inference machine PSI developed by ICOT. The description language for SIMPOS is ESP, a logic programming language augmented with object-oriented features. This paper describes how features of ESP were used and which were effective in what aspects in the development of SIMPOS. Brief descriptions of the development history of PSI and essential features of ESP are also given.

1 Introduction

The development of sequential inference machines was initiated almost immediately after the organization of ICOT Research Center in 1982 as one of the very first tasks of the FGCS project. There was an urgent requirement of a comfortable programming environment for research and development in various FGCS software projects. Although the long-term research plan of the project aims at a highly parallel inference mechanism, *practical* computers based on such a mechanism may only be available after the ten years of the FGCS project are completed. On the other hand, the logic programming environments already available were far from being adequate for programs with serious complexities. The sequential inference machine projects were meant to fill this gap.

PSI, the personal sequential inference machine was designed to be the first of such sequential inference machines[9]. It was designed as a personal AI workstation, with reasonable execution efficiency for logic programs and a flexible man-machine interface with features such as a multiple windows system. The execution efficiency was achieved mainly by the dedicated hardware and the microcoded high-level machine language KL0. The operating system of PSI, called SIMPOS (sequential inference machine programming and operating system), was responsible for the human interface part[4].

It was evident that such a sophisticated operating system cannot but be very large-scale software. With accelerated execution speed and improved memory availability provided by the dedicated hardware, application programs on the machine were also expected to become much larger and more complicated compared with the programs on the conventional Prolog implementations with their limited performance. The simple flat structure provided by the naive Prolog language was not adequate for construction of such large-scale programs. There already had been several proposals to introduce certain program structures to Prolog, but none of them had both sufficient functionality and appropriate efficiency considerations to describe a practical operating system.

A new language called ESP was designed as the system description language for SIMPOS[1]. ESP is a language in a level still higher than Prolog, in a sense, with much improved program modularity provided by its object-oriented features. All the widely recognized advantages of the object-oriented programming methodology also apply to ESP. In addition to those features common to *procedural* object-oriented programming languages, the logic programming nature of ESP allows the inheritance hierarchy to represent the IS-A relationship of semantic networks naturally. From the efficiency standpoint, firmware-supported table lookup mechanism made unavoidable search overhead sufficiently low.

The entire SIMPOS system, from the lowest layers of the process and memory and device management modules, to the highest layers of programming tools providing sophisticated man-machine interface, was written in a single very high-level language, ESP, *in toto*. This was a great help in managing the development.

This paper is organized in the following way: A brief history of the development of PSI, ESP and SIMPOS is given in Section 2; a summary of characteristic features of the ESP language is given in Section 3; Section 4 describes how the features of ESP were used in SIMPOS and which were most effective in what aspects; and merits and

demerits of using a single programming language throughout the system construction are discussed in Section 5.

2 Brief Development History

The whole project was initiated in the summer of 1982. The first step of the project was to design the functions of the high-level machine language of PSI called kernel language version-0 (KL0), which included all the essential features of Prolog (such as unification and backtracking) with extensions to describe everything required for a stand-alone computer system. The functional design of KL0 also meant the design in outline of the architecture of PSI. Development efforts of the hardware, the firmware and the software were initiated in parallel based on the agreed design.

Through preliminary investigations of the operating system design, the object-oriented style was adopted as the principle of system construction, and the ESP language was designed in the summer of 1983. The prototype version of the PSI hardware was ready by the end of the same year; the first version of the firmware in spring 1984; and the first version of its operating system SIMPOS became available for a demonstration at the FGCS'84 conference in November. This demonstration version was not functional enough to be used practically for program developments as it did not even include a self compiler. The first stand-alone version called 0.7 was released at the end of the year; 1.0 in spring 1985; 2.0 in spring 1986; and the newest version 2.5 was released in July 1986.

Currently, PSI, KL0, ESP and SIMPOS are widely used in software research and development efforts in the FGCS project. The newest version of SIMPOS has about 230,000 lines of source code with 1,400 classes and 18,000 predicates.

3 The ESP Language and Its Implementation

ESP is a logic programming language augmented with object-oriented features. This section briefly describes the characteristic features of the language.

3.1 Logic Programming Features

All the features provided by KL0 are immediately available in ESP programs. As Prolog like features of KL0 such as the unification-based parameter passing mechanism and the backtracking-based execution control mechanism have thus been inherited to ESP, it is basically an extended version of Prolog. Prolog programs can be translated into ESP without any crucial revision.

In addition to standard Prolog features, KL0 has various extended logic programming features. Among them, the following extended execution control mechanisms are most important:

Exception: Implicit invocation of a certain predicate (called the *exception handler*) on finding an error in the execution of a built-in predicate. A typical case is when

an arithmetical overflow is found. The execution of that built-in predicate is virtually substituted by the execution of the exception handler predicate. Thus, it is possible to continue the execution by simply returning from the exception handler, possibly after unifying output arguments with appropriate values.

Bind-hook: Instantiation driven invocation mechanism which is essentially the same with the *freeze* feature of Prolog-II[2].

Multiple-level cut: An extended *cut* feature where the program can specify up to which invocation level alternatives should be neglected. A combination of this feature and enforced failure enables a Catch-Throw like non-local exit program construct.

Persistent alternatives: Alternative branches which can never be removed by *cut* instructions.

Other extensions to Prolog include:

- Low-level resource manipulation features.
- Lisp-like *impure* features for updating persistent databases.
- Powerful string manipulation features.

3.2 Object-Oriented Features

3.2.1 Basic Notions

From the logic programming viewpoint, objects in ESP correspond to axiom sets. In object-oriented languages, the same message may invoke different procedures depending on the objects that receive the message. In logic programming languages, the same goal may invoke different predicates depending on which axiom set is used to refuted the goal. This is the basic concept of the object-oriented features of ESP. This basic notion is common to the *world* mechanism provided by Prolog-II[6]. The key difference is that an axiom set, i.e., a collection of procedures, is treated as a *data* object, which is the very idea of object-oriented programming.

ESP has two kinds of predicates. One is called a *local predicate*, with almost the same semantics as predicates in Prolog: The predicate to be invoked is determined statically at compilation time by the predicate name and the number of arguments given to the invocation. The only difference is that the scope of predicate definitions is limited (to one *class* definition, see below) to improve modularity. The other is called a *method* which realizes object-oriented invocations: The predicate to be invoked is determined dynamically both by static information such as the predicate name and the number of arguments, and by dynamic information, i.e., the axiom set associated with the *object* given as the first argument of the goal. This allows a highly flexible dynamically parameterized programming style.

Objects can have a fixed number of value holders called *slots*. Slots are identified by their names. The association of slots and their values is considered to be a part of the axioms associated with the objects. Slot values can be updated as *side-effects*. Such modification is a quite restricted version of *assert* and *retract* operations. By such a restriction, semantical ambiguities and implementational difficulties in modifying the program currently in execution can be avoided. Another great advantage is, of course, execution efficiency.

An object belongs to a *class*. Objects belonging to the same class have the same axiom set except that values associated with slots may be different (the set of slot names they have is the same). Objects belonging to a class are said to be *instances* or *instance objects* of the class. For each class, there is one object called a *class object*. The class objects can be identified by the name of the class, but instance objects are anonymous. The class object has an axiom set different from the instance objects of the class.

An ESP program consists of one or more *class definitions*. A class definition describes various attributes of the class, including the axiom set associated with objects of the class.

3.2.2 Inheritance

A class can have one or more *superclasses*. Such superclasses can have their own superclasses again. All of them are also superclasses of the original class. The superclasses of a class thus forms a tree of classes, called an inheritance tree. The superclasses of a class specified explicitly in its definition are called *direct* superclasses.

The set of axioms associated with an object is the *union* of all the axiom sets defined in all the superclasses. Thus, a method has all the alternatives provided by all the superclasses. By this inheritance mechanism, the inheritance tree corresponds to an IS-A inheritance semantic network. Though the order of the inherited axioms is not essential as long as pure logic is concerned, it is actually strictly defined in ESP to allow the programmers' efficiency considerations and to give *cut* a reasonable semantics (see below).

The inheritance mechanism explained above is *monotonic* in the sense that all the goals that can be executed successfully for objects of one class can also be executed successfully for objects belonging to any of its subclasses. With monotonic inheritance mechanisms only, an almost complete design of the system must be at hand before starting programming the root class of the inheritance hierarchy, because inheriting the class and also modifying some of its functions non-monotonically will be impossible.

The only non-monotonic feature in Prolog is the *cut* operation, which also is made available in ESP to cut inherited alternatives. This realizes the so-called *method overriding* feature in a quite generalized manner.

Another form of non-monotonicity is introduced by the *demon combination* feature, similar to that provided by Flavors[S]. Method predicates that are defined as demons are added to normal method predicates as *conjunction* rather than *disjunction*. Programming sophisticated control structures will be much easier using this feature.

3.3 Implementation

All the features of ESP can be implemented by compiling it into KL0. In the early stages of the development of SIMPOS, ESP was actually implemented without any special firmware support. In this sense, ESP is a language one level higher than KL0.

3.3.1 Hardware

The hardware architecture of PSI does not differ much from conventional computers except for its data tag handling mechanism and unique memory management scheme[5].

Based on the experiences with the current version of PSI, a new hardware called PSI-II is being designed in ICOT. The processor is primarily intended for element processors of the parallel inference machine pilot model, Multi-PSI, but the same processor will also be available for personal machines.

3.3.2 Logic Programming Features

The KL0 interpreter is written in microcode based on the *classical* structure-sharing method. Its execution speed is around 40 KLIPS for deterministic list concatenation and around 30 KLIPS on average. The small difference of these two figures suggests possible efficiency improvements in simpler inferences. The implementation on PSI-II based on Warren's abstract instruction set[7] is predicted to have average performance of around 100 KLIPS, and more than 200 KLIPS for list concatenation.

3.3.3 Object-Oriented Features

Methods are accessed by their name and arity at runtime, requiring a certain runtime table lookup procedure. Slots could have been accessed using their displacement rather than their names if the object code were duplicated for each class. However, as the memory overhead by such duplication would have been a serious problem, slots also are accessed by their names in the current implementation of ESP.

Due to the rather complicated multiple inheritance feature, a naive table lookup implementation analyzing the inheritance hierarchy at runtime might involve a very large overhead. The current implementation avoids this overhead by analyzing the inheritance hierarchy at linkage time, making the table lookup a simple hash table search.

Several built-in predicates are provided by the firmware for accelerating the execution of ESP programs. They are for method invocations and slot accesses. Accesses to an object slot by its name (including argument preparation and output unification) have about 30% overhead compared with accesses to an element of an array by its index; typical invocations of a method (including argument preparation and head unification) also have about 30% of execution overhead compared with invocations of a local predicate.

These figures on the ratio of overhead are quite small compared with implementations of non-logic-based object-oriented languages. This is mainly because the granularity of

operations is much larger for Prolog; and partly because the current implementation of Prolog-like features of KL0 is not the best possible: both make the denominator larger.

The PSI-II implementation of the object-oriented features of ESP currently being designed, uses basically the same principles, but details are much better optimized. Preliminary invenstigations indicate the same 30% overhead for 100 KLIPS execution.

4 ESP Features in SIMPOS

Various features of ESP were extensively used in the description of SIMPOS. How such features were used and which were most effective in what parts of SIMPOS are reported in this section.

4.1 Unified Design Principle

The primary reason for introduction of the object-oriented features to ESP was that the preliminary overall design of SIMPOS was based on object-oriented notions. The flexibility of the method call and inheritance mechanism allowed consistent decomposition of required functions into submodules.

As the implementation language ESP was also based on the same principle, notions in the overall design corresponded almost directly to ESP classes and still further decomposition required for actual implementation could also be based on the same principle. This unified design principle had various advantages.

Straight implementation: As there was almost no gap from the higher-level design of the system to the actual implementation, it was quite straightforward to write down the programs.

Program readability: As conceptual notions correspond to object classes almost one to one, it was much easier to understand programs even when they were not well-documented.

Finding design problems: When implementation of some module becomes awkward, it is sometimes due to certain inadequacies in the conceptual design. In the case of SIMPOS, as the implementation usually reflects the conceptual design faithfully, it was easy to find such problems in the higher level design.

4.2 Extended Execution Control

Some of the extended execution control features of ESP were found to be indispensable, at least to implement an operating system.

Handling errors in user programs is one of the most important task of an operating system. An operating system cannot expect user programs to be error free. The same problem occurs in different layers of the operating system itself: Lower layers of the system should check out errors of upper layers. The same argument applies also to different layers of application programs.

SIMPOS provides a general mechanism for handling exceptional cases (called *events*) depending on the *state* of the computation (called the *situation*). What to do for various events is determined by looking up a table associated with the computational process called the *situation stack*. It is possible to specify, conversationally if so desired, either to resume computation normally possibly after certain error recovery, to *fail* the *operation* which caused the event, to go back to one of the appropriate resumption points recorded in the situation stack such as the top-level user interaction loop, or to kill the whole process. The extended execution control features are extensively used to implement this mechanism.

Exception: The exception mechanism is used to activate the situation mechanism by *raising* an event when some error is found in built-in predicates. Because of this feature, user programs possibly with errors can be executed safely in the machine code rather than interpreted.

Multiple-level cut: The catch and throw mechanism implemented using the multiple-level cut feature is used to return program control to an appropriate resumption point.

Persistent alternatives: Persistent alternatives are used to maintain various databases in order, even when unexpected failure occurred. An important example of such a database is the situation stack.

The `bind_hook` feature was not actually used in SIMPOS. It is most effective in application programs. For example, an ESP implementation of a language based on situational semantics called CIL[3] uses this feature extensively, resulting in much better performance compared with its implementation in Prolog.

4.3 Dynamic Method Search Mechanism

The method call mechanism of ESP delays the decision of which predicate to call depending on the arguments given at runtime. This mechanism has the following advantages:

Module independence: Modules can be designed highly independently. A program fragment invoking a certain method should only be aware of *what* that method will do in an abstract sense. *How* that invocation will be executed does not matter. Thus, the design details of *invoked* modules can be changed without modifying the *invoking* modules at all. Although this kind of *encapsulation* technique is also applicable to simpler procedure-based modularization schemes, the independence of modules becomes more explicit in the object oriented scheme.

Extensibility: A new *invoked* module can be very easily added to the system. As the table of methods to be looked up is somehow attached to the *invoked* object rather than the *invoking* code, no modification in the *invoking* modules is required.

For example, the *unparser*, which translates the internal representation of data into a character sequence, could be designed without knowing how the generated characters should be used. This could be effected simply by calling the character output method (`:putc` in SIMPOS) of the object passed as the argument. The destination can be redirected easily by giving some destination object, a file or a bitmap display window object for example, to the unparser.

Simply defining a new class with the method `:putc` and passing its instance will suffice to add a new way of treating the resultant character sequence. For example, if only the length of the generated character sequence is of interest, the method defined in such an object should only increase the contents of some counter slot. This somewhat resembles the unified treatment of I/O in UNIX where all the I/O are to and from so-called *files*, which actually could be a terminal device or an interprocess communication stream called a *pipe*. However, the object-oriented approach is more general in that not only I/O but everything can be parameterized in the same manner.

The advantages of the dynamic parameterization feature were most effective in the following aspects:

Flexible interface: Allowing user-defined behavior in an operating system module was possible by simply allowing user-defined objects to be passed as arguments of a system-defined module which invokes some method of the argument object. This provided more flexible application program interface. The SIMPOS I/O subsystems benefited most by this function. The same advantage can be found in the interface of different layers in the system hierarchy. For example, closing open files and releasing locked semaphores on unexpected process termination are effected by the same invocation of the same method, in the process management module, of different objects.

Incremental extension: In many cases, addition of quite new features to SIMPOS only required addition of some new classes to the system, even when such features had not been taken into account when designing the part of the system which may invoke some method of the newly added classes. The earlier versions of SIMPOS, with less functionality and even with less basic *notions*, were improved by simply adding new classes realizing new notions without any modification of the existing and functioning part of the system. For example, closing a network connection on unexpected termination of a process is also effected by the same invocation of the same method as closing of an open file in the process management module, although such network connections were not taken into account when the process management scheme was designed.

4.4 Inheritance Mechanism

The inheritance mechanism allows so-called *differential programming* styles, where only the *differences* with some other module need to be newly written and all the common features of existing modules are reused. This mechanism has the following advantages:

Code size minimization: This has direct effect on the size of the program code. As sharing of common features is easy, duplication of code can be minimized both in source code and in object code. Usually, this advantage is more effective when a program becomes larger and more complicated.

Modularization: As ESP allows multiple superclasses of a class, it is quite easy to define a new class by combining simple functions provided by predefined simpler and possibly incomplete classes. This encourages fine-grained modularization providing smaller modules with simpler functions.

Program readability: As only the *differences* with some other class need to be described, class definitions tend to be more compact. It also becomes possible to incrementally understand the system beginning with classes with simpler functionality and proceeding step by step to more complicated classes, concentrating on *differences* with already understood functions.

The advantages of the inheritance mechanism was most effective in the following aspects:

Incremental enhancement: Addition of new functions to existing features was quite easy by simply adding classes realizing only the newly introduced functions. A class with desired functionality can be made by using both newly introduced classes and already existing classes. This was advantageous because the development effort and, more importantly, the debugging effort could concentrate on the newly introduced features.

Orthogonality in provided features: By the multiple inheritance feature of ESP, a class in an application program can inherit several classes of SIMPOS mixing various functions they provide for the required set of functions. Without this feature, the operating system had had to prepare all the possibilities of such combinations, resulting in a combinatorial explosion. This was most effective in the windows system.

Flexible interface: The application programs can *customize* the features of SIMPOS by using predefined classes and redefining some of the methods. This makes possible almost unlimited customization without thorough rewriting. The same advantage, again, is utilized by the interface between different layers of SIMPOS. For example, many of the bitmap display windows for various tools provided by SIMPOS required low customization for a better human interface.

Ease in management: When some class required modification, the changes were automatically propagated to all the classes inheriting them. This considerably facilitated development management.

5 Single Language Principle

Advantages and disadvantages of using a single high-level programming language ESP throughout the description of the operating system and application programs are discussed in this section.

5.1 Conformity between Different Layers

In most implementations of high-level languages (especially, non-procedural high-level languages) on conventional operating systems, some of the basic principles of the language (memory management principles, for example) are quite different from that of the operating system. Thus, making the operating system functions directly available to the user programs in the high-level language may be problematic in the language system. To maintain consistency, language systems usually provide some limited interface with the operating system. In such implementations, the application programs cannot utilize the full functionality of the operating system.

To solve this problem, some language systems allow linkage with routines written in other lower-level languages such as C or assembly languages. However, interfacing with such lower-level languages requires knowledge of implementation details.

The same situation would obtain in interfacing different layers of a single system if they were written in different languages.

There are of course no such problems in a single language system such as SIMPOS.

5.2 Open System

One of the most important advantages of using ESP for the description of SIMPOS was that the system could be quite open to the users. In conventional operating systems, features of the operating system are only available as *subroutines*. In SIMPOS, application programs can utilize the features in a very flexible manner through the *inheritance* mechanism of ESP. This mechanism was also used extensively for interfacing different layers of ESP.

Adoption of this system organization extensively using the inheritance mechanism was only possible by using the same language in different layers. If, for example, lower layers were written in some lower-level languages such as C, the interface of that layer and upper-level layers would have been much more awkward.

5.3 Efficiency

The language for a single language system must provide features high-level enough to be able to describe highly complicated application programs. However, usually the *existence* of high-level features somewhat decreases the execution efficiency of all programs including those which do not actually use such features. For example, in the case of Prolog, whether variable bindings should be trailed or not must be checked out even when the program runs deterministically.

This efficiency penalty pays in programs where high-level features are actually effective. However, in a single language system, lower level layers where such high-level features are not required at all also suffer from this overhead. This is where the most important drawback in using a single language throughout the system lies.

Some people were quite doubtful about SIMPOS implementation, because the language adopted, ESP, was based on Prolog, which was reputed to be quite inefficient, augmented with also reputedly-inefficient object-oriented features. 40 KLIPS for append means that a simple loop requires 25 μs, which is slower by one order of magnitude than general purpose processors of the same scale. As if to corroborate these doubts, displaying one character to a bitmap display window took a few seconds in one of the earliest pilot versions of SIMPOS, in the summer of 1984.

However, this efficiency disadvantage was not actually so serious. Within only a few months, various improvements in the data and algorithm design levels led to increasing the program speed by more than two orders of magnitude (newly introduced firmware support increased speed about three times), and the system attained speeds suitable for practical usage. Considering the size and complexity of the windows system, the time required for this improvement was remarkably short. This was due in large part to high modularity, cleanliness and flexibility of module interface, and ease in development management provided by the ESP language.

It is well understood that what governs the efficiency of a program is the adequacy of high-level design in algorithms and data structures, rather than extent of low-level hacking. As changes in the specification of a program are unavoidable and usually the way a program is used cannot be predicted precisely before it is run, it is practically impossible to design the optimal algorithm beforehand. Whether or not the program can be easily modified to meet ever changing needs is one of the greatest factors that determine the efficiency of the final software product. Thus, a programming language providing features which allow easier modifications finally provides higher execution efficiency, even if it is slow and requires a lot of memory for simple benchmark programs.

6 Conclusion

Various features of ESP were extensively used in implementing SIMPOS and many of them were quite effective in many aspects. Considering the functionality of the system, the time period required for the development was rather short compared with conventional operating systems. The strategy of using a very high-level single programming language throughout the system description and in application programs was one of the most important reasons of this success.

The efficiency drawback of using a single high-level language seems to be somewhat overestimated. The flexibility provided by the higher-level features is quite beneficial in that large-scale revisions for efficiency improvement becomes much easier.

Acknowledgements

Katsuto Nakajima helped the author in providing performance figures of the current and PSI-II implementations of ESP. Figures about the size of the current version of SIMPOS are by Hiroyoshi Ishibashi and Kaoru Yoshida.

References

[1] T. Chikayama. Unique features of ESP. In *Proceedings of FGCS'84*, ICOT, 1984.

[2] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Last steps towards an ultimate Prolog. In R. Schank, editor, *Proceedings of IJCAI-81*, IJCAI, 1981.

[3] K. Mukai. *Horn Clause Logic with Parameterized Types for Situation Semantics Programming.* ICOT Technical Report TR-101, ICOT, 1985.

[4] S. Takagi et al. Overall design of SIMPOS. In *Proceedings of the Second International Conference on Logic Programming*, Uppsala, 1984.

[5] K. Taki et al. Hardware design and implementation of the personal sequential inference machine (PSI). In *Proceedings of FGCS'84*, ICOT, 1984.

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

[7] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, 1983.

[8] D. Weinreb and D. Moon. *Lisp Machine Manual, 4th edition.* Symbolics, Inc., 1981.

[9] M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida. *The Design and Implementation of a Personal Sequential Inference Machine: PSI.* ICOT Technical Report TR-045, ICOT, 1984. Also in New Generation Computing, Vol.1 No.2, 1984.