TR-061

Coordinator – the Kernel of the Programming System for the Personal Sequential Inference Machine (PSI)

by

Toshiaki Kurokawa and Satoshi Tojo

(Mitsubishi Research Institute)

April, 1984

ICOT

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

(03) 456-3191 ~ 5
Telex ICOT J32964

Coordinator - the kernel of the programming system for the Personal Sequential Inference Machine (PSI)

*Toshiaki Kurokawa and Satoshi Tojo**

Institute for New Generation Computer Technology (ICOT)
*Mitsubishi Research Institute

Abstract

In this paper, the programming system of the personal sequential inference Machine (PSI) is described. The kernel of the programming system is called the Coordinator, and its main features include: communication interface, command sending, management of programming system elements, Help facility, and miscellaneous facilities, such as that for logging on.

The design objectives are simplicity, efficiency, and expandability. These are achieved through our design and implementation, using the powerful support of our language, ESP, which has an object-oriented class inheritance mechanism similar to those of Smalltalk-80 and Flavors.

1. Introduction

To achieve a highly interactive personal computer environment, which is one goal of our Personal Sequential Inference Machine (PSI) [Uchida 83], a support system is required. One of the primary aims of the Japanese Fifth Generation Computer Systems Project is to implement more intelligent, user-friendly computer systems. PSI is designed to achieve this objective throughout the use of a powerful logic programming language.

However, the language alone is not enough. An operating system is necessary to support the user in basic resource and process management. Again, however, additional supports are needed for sophisticated communication with the system. We call the collection of such facilities the PSI Programming System.

Research and development on programming systems in general has a rather long tradition in the young field of computer science, but programming systems suited to highly interactive personal computers have not been studied so intensively.

In this paper, we report on our attempt to construct such a sophisticated programming system. We have concentrated on the Coordinator, the kernel of the programming system. We describe our goals and our implementations, and include a summary of our work.

The Coordinator, like other SIMPOS (Sequential Inference Machine Programming and Operating Systems) modules [Hattori 83], is written entirely in ESP, (Extended Self-contained Prolog) [Chikayama 84], a dialect of Prolog incorporating an object/class methodology similar to that of Smalltalk-80 [Goldberg 83].

2. Programming System Kernel

Nowadays, the need for a programming system is generally accepted, but the precise meaning of the term has not been well defined. There are various definitions and each has its own rationale. Here, we consider the programming system of SIMPOS to be a collection of expert processes.

Later, we will formally and syntactically define "*expert process*". Informally, an expert is a process with which the user performs a job, for example, an interpreter, a text editor, or a compiler. These are examples of programming tools. There may be other programs, such as a computer game program, which can be seen as an expert. A database management program can also be an expert. In general, the expert is an independent process that communicates mainly with the user.

This view differs from that of the programming system as a collection of dumb software tools, or of that in which the programming system is seen as a collection of programs to support software production. For example, in Unix*, the software tools are tend to be small. They are combined by the Shell command language, and used many places as only a part of the program.

Our view frees us from the overhead of managing available utilities or the little-understood process of program production.

We admit that this view is not sufficiently developed as to constitute a sophisticated programming system. However, our intention here is to first establish the kernel of the programming system, then to proceed gradually towards a more sophisticated system. (More research is necessary on the mechanics of programming and man-machine communication, which are intrinsically related to the Fifth Generation Computer Systems Project.)

From the user's viewpoint, an expert can be invoked, controlled, and terminated through the expert's window. In our system, there is no explicit supervisory process (such as the Shell in Unix). However, there is a background process which is called the Coordinator. A user need not be concerned with the complex history of invoked processes to accomplish a task and need not fear unexpected destruction of work through errors in tracing such histories.

Aspects of the man-machine interface are discussed in detail in a separate paper [Tsuji 84] along with the design and implementation of the window function.

3. Coordinator : Design and Implementation

3.1 Design goals for the Coordinator

The design goals are simplicity, efficiency, and expandability. Simplicity and efficiency means that the user need not be concerned with the operation of the Coordinator.

It appears to the user that he/she controls the expert directly through the window. (In fact, the

*) Unix is a trademark of Bell Laboratories.

Coordinator assists the user's control via the window interface.)

In this sense, the Coordinator is entirely different from a command processor, such as the Shell in Unix. The Shell is a command processor with a command language in which the user can program his instructions.

Actually, the command language is an extension of the existing Job Control Language, which originally was the Operating System language. As Ingalls argues [Ingalls 82], the command language can be replaced by the language interpreter itself if the whole system is constructed around a powerful programming language. The examples he mentions are Lisp and Smalltalk.

Now we can add our language, ESP, and our system, SIMPOS, as further examples. We can use ESP as a command language, and our interpreter as a command interpreter. In fact, at an early stage of development, all resource and process management functions were coded in ESP.

Thus the Coordinator is mostly concerned with the basic management of expert processes by the user through the experts' windows. An expert process is an any process the user wishes to use. It must belong to a special class, an *'e_process'*, which has a special mechanism for communication between the user and the Coordinator.

Expandability is achieved partly through the registration of expert processes and partly through command set definitions for expert processes. It is also planned that the Coordinator will become a part of a more sophisticated interface system the user and the SIMPOS.

The principal functions of the Coordinator are as follow:

- sending the user commands to experts through the window;
- creating, activating, suspending, resuming, and deleting experts;
- executing special commands for experts;
- acting as a communications interface between experts via the *'whiteboard'*;
- providing Help functions to users;
- providing miscellaneous facilities such as logging on, interfacing with resource managers, etc.

3.2 Implementation of the Coordinator

To implement the Coordinator, the representation of experts, the set of experts to be used, and the expert's window are the main problems. In addition, the methods of communication between users and experts and between experts themselves are important points.

1) The representation of experts

An expert is represented as a special class of the *'e_process'*. An instance of the *e_process* has the following attributes:

- a special class of the *'program'*, called *'e_program'*.
  (Note that the *'program'* is a name of a class of objects in SIMP OS. A program in the usual sense is the method defined around the object in ESP. The *'program'* is not a method but a class of object. The instance of the *'program'* has a special method *'goal'* to execute the main function. The instance can also be *'invoked'* in the process environment.)

- a special class of the *'window'*, called *'e_window'*.
  (The window contains a mechanism for man-machine communication.)

- a special class of the *'port'*, called *'e_port'*.
  (*'Port'* is the name of the class of objects that enable communication between processes. In fact, this port is used for communication with the Coordinator.)

- the name of the expert
  (Actually, the name of the object stored in the *e_process_pool* explained bellow, paired with the expert. The name is derived from the *e_program* name, which the user defined in the *'e_program_directory'*.)

Thus, the expert has a window to communicate with the user and a port to communicate with the Coordinator. Moreover, the window has a table for translating user commands into a form that can be used by the Coordinator to help control the expert.

2) The representation of the set of experts

In a sense, the set of experts is itself the Programming System of PSI itself. If we had a mechanism for defining any set to be an object, we could define the set of experts to be the Programming System.

However, we do not have such a mechanism and it is more realistic, in terms of efficiency of both memory and speed, to have a special process, that is, the Coordinator, and a special object for representing the set of experts.

We represent this set as a special instance of the class, *'e_process_pool'*. *'Pool'* is a term used in SIMPOS that means a collection of objects. The *e_process_pool* is derived from the class, *'list_index'*, which can store any number of objects and retrieve any object using an index. Experts are indexed by names assigned to their *e_program* by the user. Later, the index object will be replaced by *'cascade_index'*, which is more efficient. This replacement will be very easy, thanks to the class mechanism.

In fact, the *e_process_pool* is implemented to be an active object for processing special commands, such as *'broadcast'* or *'remember'*, because these commands can not be sent to a special expert.

3) The representation of the expert's window

The expert's window is an object of the class, *'e_window'*. It is a special window in the sense that it has a *translation_table* and an *e_process*, that is, the expert itself.

The *translation_table* of the *e_window* contains keystroke commands and the mouse-button-click commands along with the corresponding SIMPOS code or commands that are sent to the expert. SIMPOS uses a 16-bit code based on JIS kanji-code. A command is provided to define the keystroke command for the expert process.

This translation is performed in the window process with very little overhead. Thus, we have introduced a simple and efficient mechanism for handling keystrokes and mouse commands for experts.

Some commands are sent to the Coordinator, which then handles the expert itself. These commands are called *'meta-commands'*. Examples include kill, lull, status, visit, memorize, broadcast, remember, read_from_whiteboard, write_to_whiteboard, and system_menu_invoke.

The expert object (actually, a pointer to the object) is stored in the *e_window* to tell the Coordinator which expert is linked to this window. When a code or command is to be sent to an expert, the expert is identified through a special port maintained by the window manager.

4) Communication between the user and the expert

Communication is performed through the *e_window*. There are two kinds of inputs: those processed in the expert and those sent to the Coordinator to manage the expert.

The latter is called *'meta-command'*. For example, "kill" is a meta-command because system intervention is necessary to kill an expert. "Create" is another example in which some additional process is required to create an expert. "Visit" indicates that the communication between the user and the current expert is interrupted and another expert is selected for the next communication.

5) Communication between experts

There are two types of communication between the user and the expert. One is the communication embedded in the experts' program. The other is the communication specified at the user's discretion.

The former is performed through the experts' ports. The problem of synchronization is effectively managed by *port* objects.

The latter type of communication is much more difficult. It cannot be determined beforehand to whom the communication should be made. It is possible to establish communication paths between the all experts, but it is highly impractical because of the requirements of memory and control.

Another approach is to set up a virtual communication network connecting all the experts.

However, this mechanism is complicated and overhead is significant.

Our solution to this mechanism is to provide a special object, *'whiteboard'*, to assist in communications among experts. The Coordinator's *whiteboard* is just like a blackboard in which an expert places a message in order to communicate with another expert.The term 'blackboard' is deliberately avoided, because it has a special meaning that the blackboard not only contains the message but also controls the process invocation [Erman 80] [Nii 79]. The *whiteboard* has, on the other hand, no control over the process activation/termination. Only the user has the control over the processes.

The *'whiteboard'* is a buffer in which an object is stored by the expert as directed by the user. The user can also order the expert to pick an object from the *whiteboard*. For example, it is possible to route an object from the interpreter to the *whiteboard*, the user then visits the editor expert to edit the object, then, re-visits the interpreter to execute the edited program.

The inheritance relation of the herein mentioned classes are summarized in the Figure 1.

****

Figure 1. should be inserted here.

****

3.3 Meta-commands for expert management

There are two sets of commands as mentioned in section 3.2. The following meta-commands are provided to control experts:

1) visit

Starts (or re-starts) communication with an expert. The intended expert is pointed out by the mouse cursor. Usually, visiting an expert involves terminating the current communication with the current expert.

2) create (using the e_program)

Create an expert using the *e_program*. The *e_window* and the *e_port* are also created automatically.

3) kill

Deletes an expert, i.e. releases all the resources including its process, (even the physical process in the sense of hardware resources) and its execution is aborted.

4) lull

Suspends execution. When visiting another expert, the current expert's execution is not always suspended. It may continue execution, but it will not communicate with the user until it is visited. That is, the expert will be in a state of I/O suspension. However, the 'lull' command suspends execution.

5) arouse

Wakes the lulled expert. Before visiting the lulled expert, the user must send this command.

6) memorize

Stores the identity of the expert that the user is now visiting in the history table. The user can visit the old expert by invoking 'remember' command.

7) remember

Visits an old expert memorized in expert history.

8) broadcast

Sends a message to all experts that execute a class of the *e_program*. For example, using this command, the user can send a message to all the text-editors in the system to find a particular passage.

9) write

Causes the expert to write an object to the *whiteboard*. This command is mediated by the Coordinator.

10) read

Reads the object from the *whiteboard*. The processing of the object is up to the user.

11) invoke the system menu

The system menu is a special menu of various jobs. A typical job is the creation of expert processes. There are other jobs, such as logging onto the system, assisting the user with Help information, and managing the file system.

This list does not claim to be complete. Our intention is to show that it is possible to do the above in the current implementation of the Coordinator. It is easy to extend the list, thanks to the class mechanism provided by the ESP language.

4. Customization and Extendability

We think customization and extendability are important considerations for a constantly expanding system such as ours. In this section, we want to explain how these goals are achieved by our implementation.

4.1 The set of available experts

There is a class of objects, *'user'*, for each SIMPOS user. Each user has a directory for his/her own set of experts. It contains the experts' names and the corresponding *e_program* to be executed by each.

When a user logs onto SIMPOS, his/her expert directory is searched and the system_menu displays that user's list of experts.

The Coordinator does not prevent users from defining their own set of experts, that is, their own programming systems. It should be noted that users need not make their directories from scratch. They can modify the system's standard directory. They can also add experts originally developed by others.

Even during SIMPOS execution, users can change their expert directories, and, in so doing, can update their own programming systems.

4.2 The command set for the expert

The command set (including single-keystroke commands, and mouse button clicks), forms the basis of the communication language between the user and SIMPOS.

The expert has its own command table associated with its *e_window*. The table tells how each keystroke should be interpreted, as a command or as an internal code, and also where the contents should be sent. (Mouse button clicks are also encoded and are included as special keystrokes.)

The Coordinator provides a method whereby the user can change the window key-command table. *E_windows* accept the *'key-command_table-change'* command, but other classes of windows may not accept the command.

It is possible to modify the table during execution, so the user can use a limited number of keystrokes to indicate virtually infinite number of commands.

A Help function is included so that even if a user forgets the meaning of a keystroke, the system will supply the meaning [Tsuji 84]. Thus, users can construct the most convenient set of commands for their applications.

4.3 General mechanism -- class and inheritance

Customization and extendability are also possible throughout SIMPOS due to the class and inheritance mechanism which is supported by the system programming language, ESP.

The class and inheritance mechanism of ESP is more powerful than that of Flavors on the Lisp Machine [Cannon 82].

ESP has a multiple-inheritance mechanism and a before-demon and the after-demon. Moreover, it has the following capabilities:

1) The *'has-a'* relation (part-of inheritance) is supported.
2) The language is based on logic programming.

Throughout this mechanism, users can extend any existing function (even those supplied with the system) for their own use, and can exchange functions with others.

For example, it is easy to create new experts by modifying existing ones. Moreover, the user can modify the *e_window* so that the output is more informative for the user's work.

This freedom is achieved at the least possible cost in program overhead. This is true, in part, because the instance of the class is a kind of active object. For example, the command interpretation table for the *e_window* is constructed as a set of programs. It is not a dumb table handled only by other active process.

The next figure shows the ESP program for the system standard table, the *default_translation_table*. Users can modify the table to produce their own tables by using the inheritance mechanism.

****

Figure 2. should be inserted here.

****

Low overhead and the provision of user control are the main themes of the Coordinator.

5. Summary

In this paper, we described the kernel of our programming system, the Coordinator, and discussed its design and the implementation. We defined the programming system as a set of expert processes which can be defined by each user.

The Coordinator manages the set of the experts and helps the user to communicate with, query, and send commands to the expert. Our design goals are simplicity, efficiency, and extendability. These goals are achieved through our implementation together with the general facilities of our language, ESP.

The next targeted objective of our work is to develop a more sophisticated programming environment for advanced applications, using the Coordinator as a component of a larger system.

REFERENCES

[Chikayama 84] Chikayama, T. "ESP Reference Manual", ICOT TR-044, (Feb. 1984)

[Cannon 82] Cannon, H.I. "Flavors - A non-hierarchical approach to object-oriented programming", MIT Memo, (1982)

[Erman 80] Erman, L.D. and Lesser, V.R. "The HEARSAY-II speech understanding system: A tutorial", in Lea, W. (ed.) Trends in speech recognition, Prentice-Hall, (1980)

[Goldberg 83] Goldberg, A. and Robson, D. "Smalltalk-80, The Language and its Implementation", Addison-Wesley, (1983)

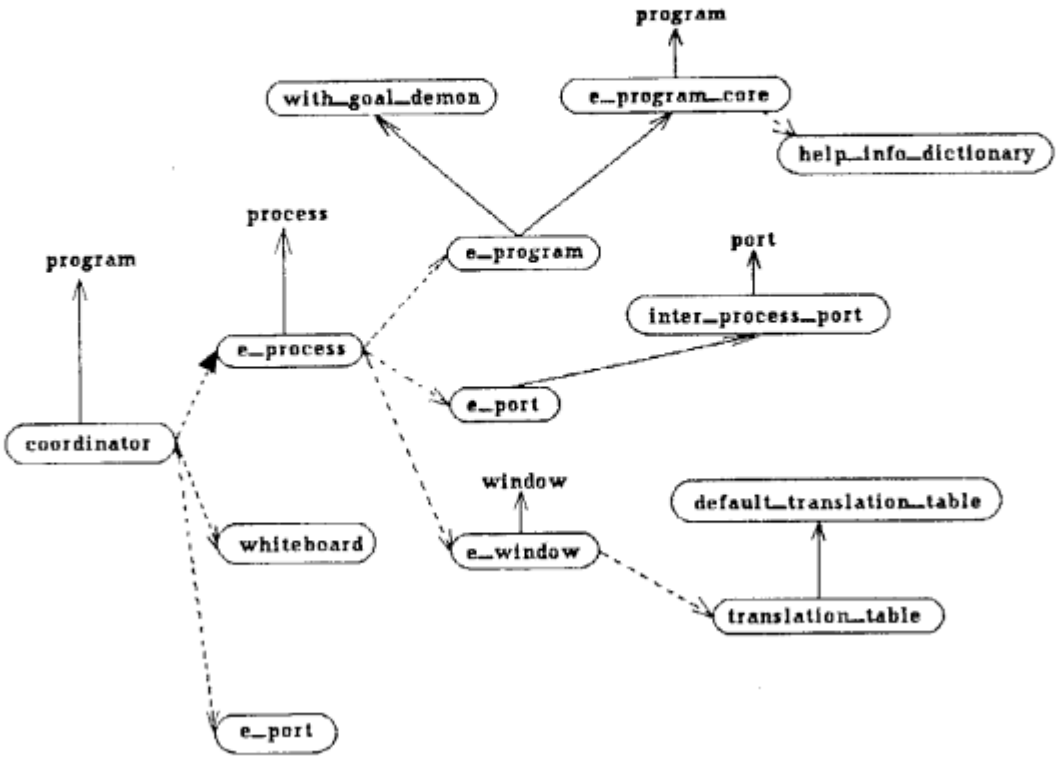
[Hattori 83] Hattori,T.,Yokoi,T.,"Basic Constructs of the SIM Operating System, New Generation Computing,vol.1 no.1 pp.81-85 (1983).

[Ingalls 82] Ingalls, D.H. "Why Operating Systems May Disappear", memo from Xerox PARC (Dec. 1982)

[Nii 79] Nii, H.P. and Aiello, N. "AGE(Attempt to Generalize): A knowledge-based program for building knowledge-based programs", IJCAI-6, pp.645-655, (1979)

[Tsuji,J 84] Tsuji,J., Kurokawa,T., Tojo,S., Iima,Y., Nakazawa,O., and Enomoto,S. "DIALOGUE MANAGEMENT IN THE PERSONAL SEQUENTIAL INFERENCE MACHINE(PSI)" ICOT TR-046 (Feb. 1984)

[Uchida 83] Uchida,S.,Yokota,M.,Yamamoto,A.,Taki,K.,Nishikawa,H. "Outline of the Personal Sequential Inference Machine: PSI",New Generation Computing, no.1 pp.75-79 (1983).



Those classes defined in the Coordinator are encircled.

The solid line indicates '*is-a*' inheritance.

The dashed line indicates '*has-a*' inheritance.

Figure 1. Relation of the classes in the Coordinator

```
class default_translation_table
   has
   instance
      :look_up(Table, Window, control#"k", #coordinator,
               {kill, E_process, _, _}, "kill") :- !,
         :get_e_process(Window, E_process)
      :look_up(Table, Window, control#"l", #coordinator,
               {lull, E_process, _, _}, "lull") :- !,
         :get_e_process(Window, E_process)
      :look_up(Table, Window, control#"a", #coordinator,
               {arouse, E_process, _, _}, "arouse") :- !,
         :get_e_process(Window, E_process)
      :look_up(Table, Window, control#"s", #coordinator,
               {status, E_process, _, _}, "status") :- !,
         :get_e_process(Window, E_process)
      :look_up(Table, Window, mouse#1, #coordinator,
               {visit, E_process, _, _}, "visit") :- !,
         :get_e_process(Window, E_process)
      :look_up(Table, Window, control#"m", #coordinator,
               {memorize, E_process, _, _}, "memorize") :- !,
         :get_e_process(Window, E_process)
      :look_up(Table, Window, control#"b", #coordinator,
               {broadcast, _, Program_name, Command}, "broadcast") :- !;
      :look_up(Table, Window, control#"v", #coordinator,
               {remember, _, _, _}, "remember") :- !;
      :look_up(Table, Window, control#"s", #coordinator,
               {invoke_system_menu, _, _, _}, "invoke system menu") :- !;
      :look_up(Table, Window, mouse#rr, #coordinator,
               {invoke_system_menu, _, _, _}, "invoke system menu") :- !;
      :look_up(Table, Window, control#"r", #coordinator,
               {read, _, Object, _}, "read whiteboard") :- !;
      :look_up(Table, Window, control#"w", #coordinator,
               {write, _, Object, _}, "write whiteboard") :- !;
      :look_up(Table, Window, X, Port, X, "send character as it is") :- !;
   end.
```

Figure 2. A Translation Table Program in ESP