TR-046

# DIALOGUE MANAGEMENT IN THE PERSONAL SEQUENTIAL INFERENCE MACHINE(PSI)

by

Junichiro Tsuji, Toshiaki Kurokawa
(ICOT)
Satoshi Tojo
(Mitsubishi Research Institute)
Yutaka Iima, Osamu Nakazawa
(Oki Electric Industry Company, Ltd.)
Shoji Enomoto
(Matsushita Electric Industrial Co., Ltd.)

February, 1984

# DIALOGUE MANAGEMENT IN THE PERSONAL SEQUENTIAL INFERENCE MACHINE(PSI)

*Junichiro Tsuji*, Toshiaki Kurokawa*, Satoshi Tojo***
*Yutaka Iima***, Osamu Nakazawa***, and Shoji Enomoto*****

* Institute for New Generation Computer Technology (ICOT)
** Mitsubishi Research Institute
*** Oki Electric Industry Company, Ltd
**** Matsushita Electric Industrial co., Ltd

## ABSTRACT

PSI (Personal Sequential Inference Machine) is being developed as a research tool for the FGCS (Fifth Generation Computer Systems) project, and SIMPOS (Sequential Inference Machine's Programming and Operating System) is also under development.

In this paper, we present the dialogue management component of SIMPOS. We describe and discuss the background, design, implementation, and future at our work. The window system and the Coordinator, along with the bitmapped display and mouse cursor controls, are the primary means of managing the dialogue between the user and SIMPOS.

As PSI is a workstation for the next phase of FGCS, users are assumed to be computer professionals, and dialogue management is being designed accordingly. High performance, low overhead, and allowances for customization and expansion are among our design and implementation goals. The entire system is implemented in ESP, an enhanced logic programming language incorporating an object/class methodology similar to that of Smalltalk-80.

## I. INTRODUCTION

There is increasing interest in man-machine interfaces for computer systems. Evidence for this is the increase in the number of published papers, related conferences, and new products [HFCS 82] [Sime 83].

This paper is submitted to the ACM 1984 ANNUAL CONFERENCE.

The major reasons for this increased interest are reductions in hardware cost and the availability of advanced equipment for man-machine communication. For example, 32-bit microprocessors equipped with megabyte-memories, a high-resolution bitmapped displays and mouse cursor controls are available in a reasonable cost

However, the problem facing the development of a natural, efficient interface lies in the system/software design. Management of the dialogue between user and system is a key problem [Gaines 83].

This is a challenge for the future. In fact, Japan's Fifth Generation Computer Systems (FGCS) Project aims at producing more intelligent and user-friendly computer systems. We are in the first stage of building such a system, and are now developing a Personal Sequential Inference Machine (PSI) which will be the pilot model for fifth generation machines [Uchida 83]. PSI is a high-performance personal machine and will be used as a research tool for the next stage of the FGCS project.

The Programming and Operating System for PSI, called SIMPOS (Sequential Inference Machine Programming and Operating System)[Takagi 84], is also under development. It is written entirely in ESP (Extended Self-contained Prolog) a PROLOG-like logic programming language [Chikayama 84].

Here, we present the dialogue management component of SIMPOS. Although it has not been completed as of this date (February, 1984), we describe and discuss the background, design, implementation, and future of our work. SIMPOS will be functional by the end of this year, and will serve as the environment for future FGCS research.

## II. BACKGROUND OF OUR WORK

Before joining ICOT in June, 1982, we had limited experience in man-machine interface research. We therefore tried to learn as much as possible about the principles of implementation and design from existing literature and from hands-on experience. In this chapter, we briefly review our findings and critique a few of the systems currently available.

### 1. Criteria for evaluating dialogue management

We searched for criteria by which to evaluate dialogue management by surveying a number of papers. These criteria are expected to serve both as design guidelines and as product evaluation standards. We realize that they may be rather experiential at this point and that we must wait for user evaluations before drawing any conclusions as to the ultimate usefulness of our product.

The criteria can be grouped into three categories: those concerning design principles, principal elements, and advanced functions. The design principle expresses the attitude underlying the dialogue management design stage. It is an abstract guideline. The principal elements are the logical blocks of the dialogue management system. There is room for dispute as to how these elements should be related to each other. The purpose of the advanced functions is to assist users with dialogue management. The latter two criteria are concrete guidelines.

We found a number of papers dealing with criteria for dialogue management. Their conclusions are not always the same. We have summarized the criteria in Table 1, according to the above grouping. We will discuss these criteria in the following chapters.

*****************************************************

Table 1. should be inserted here.

*****************************************************

## 2. Hands-on experience

We gained some experience with the Tools system developed at Yale University [Ellis 83], and the Symbolics' Lisp Machine programming systems [Symbolics 81]. We used JSTAR (Japanese version of STAR [Smith 82]) from Fuji Xerox, and discussed its features with users. We also worked with various Japanese word processors, and micro-computers.

## 3. Critiques

Although our experience was somewhat limited in scope, owing to considerations of time, we list here the following critiques based on our experience:

*1) Multi-window system with pointing device is indispensable.*

Tools does supports neither the multi-window environment nor the pointing device. The interface is very limited. In the Tools environment, a "Session Manager" is provided instead of a multi-window function, so the user must update the entire screen to obtain the necessary information.

In systems equipped with a multi-window environment, the user can easily retrieve system information, such as error messages and source programs, without suspending his job. It is often said that by using the multi-window environment, the user can use the display screen as if it were his desk on which papers and tools are spread.

*2) Program Communication (data transfer between processes) is important.*

Here, we include users' programs in processes. In very old systems, such as the batch processing system, the user can only transfer data by way of a special program or manually (e.g. input the data as an argument). Modern systems generally support multi-programming/multi-processing, in which the user can run more than one processes concurrently with considerable savings in time. However, there are still problems with data transfer between processes. We call this the problem of program communication.

Tools has limited but uniform control over program communication. That is the user can transfer data in the form of a character string between any processes on the system. The Lisp Machine, however, does not have such uniform program communication control. Although communication between the editor and the interpreter is very sophisticated, it must be programmed in object code. No general method of process communication is provided.

*3) Icon is not suited to computer specialists.*

The JSTAR Icon system is attractive. However, its targeted user is the novice and its usefulness is limited to the office work. The icon system uses a kind of menu in which each item is represented by a graphic image. This method is slower, for example in documents editing, than simple menu selection or key-code command systems. We also feel that our application programs need more work before such a facility is provided to end-users. Supporting the icon system even at a very basic system level does not appear to be cost-effective.

*4) Simplicity is very important.*

Although the Lisp Machine provides very sophisticated interfaces for several system programs, it cannot be said to be simple. Discrepancies among several programs appear to be caused by this complicated interface design. The complexity of Tools results from basing the interface design the DEC-20 command processor.

It seems that simplicity is very important for next stage of the development, although we admit it may be difficult to make these sophisticated systems simple enough.

## III. Design of SIM dialogue Management

### 1. Materials and environment

To design an efficient system, materials and environment must be considered, and it has been well said that we must not be bound by present-day (possibly ill-designed) environments.

#### *1) Physical environment*

Our principal tool is the PSI, a logic programming machine for personal use. It has 16 megawords (80 megabytes) main memory at maximum, a 1200 × 912 (or 1280 × 864) black and white bitmapped display with mouse cursor control, and support Ethernet connection [Taguchi 84].

#### *2) Human environment*

Users of our system are surely not end-users, but programmers who will use it heavily at the system level, rather than at the application level. The main interests of such users lie in the efficiency of their programs, control over system facilities, and the transparency of the overall system.

Such users are, at least potentially, also the builders of our system. We do not envision system security in the sense that the user would not be allowed access to the system region. Instead, the user could access any resource including the system kernel source codes.

#### *3) Programming environment*

Our main hardware tool is a DEC2060 with more than 40 terminals. We hope to replace it with the PSI network after we finish this SIMPOS implementation.

The machine language of PSI is called KL0, which stands for Kernel Language Version 0. The system description language, on the other hand, is called ESP, standing for Extended Self-contained Prolog. This is a logic programming language, using the class mechanism, i.e., the so-called object-oriented mechanism [Chikayama 84].

The class mechanism incorporates facilities existing in Flavor [Cannon 82] and Smalltalk [Goldberg 83]. For example, it permits multiple inheritance and before/after demons, as in Flavor. It also includes class slots which represent class variables in Smalltalk. Its component slot is the original mechanism for the introduction of the so-called part-of hierarchy.

As for software tools, we use an ESP cross simulator, an ESP cross compiler, a KL0 assembler, and a KL0 compiler, all written in DEC-10 Prolog. We also use an EMACS editor [Stallman 84], and a Z editor under the Tools environment.

*4) Project environment*

One of our problems is a tight schedule for development. Our team started in June 1983, about one year ago. We plan to complete our first stage of development around the end of this year.

One of the reasons why we adopted the enhanced logic programming language ESP is that it will provide us with naturally modular programming for this project. However, it has caused the obvious difficulties attendant on the introduction of a new language. We have managed to overcome these through intensive programming sessions and consultation between team members.

Another problem is the small number of people engaged in the project. Only about thirty people are hard at work on the development of SIMPOS. As SIMPOS covers a wide range of software, only a very few people can take charge of dialogue management. However, we are happy to report that the present report results from many in-depth discussions with our colleagues.

A third problem is project management and communication among participants. In this regard, we appreciate the software tools for the DEC2060, especially MM and OZ electronic mailers, and the STINGY documentation program.

## 2. Design goal

Our goal is to construct a dialogue system which satisfies the design criteria listed in Table 1. Hardware and users are defined in our environments: a cursor controlled bit-map display for the hardware, and computer specialists as the users. We are designing for the specialist, not the novice.

In other area, we have tried to focus the following points:

*1) Simplicity*

*2) Ease of customization*

*3) Least overhead (no loss in most cases)*

*4) Supporting program communications*

*5) Utilizing multiple windows and the mouse cursor control*

We want to produce a simple system, but we also want to use it to construct more sophisticated systems. A simple system, if not the final product, is a good starting point.

There are dangers regarding customization. In our experience, too much customization sometimes causes problems. User may find another facility totally different from their own, and this can block natural cooperation among participants. We need a kind of sharable customization, which we hope can be realized using class hierarchy, that is which automatically assures a modular structure.

An exact definition of overhead is difficult, but we hope to produce very simple system that would require less of it. We omitted several facilities in the current design, such as the undo/redo capability with conversation recording, to comply with this criterion.

## 3. Initial step - division of the Window System and the Coordinator

Our first step was to divide the functions according to the design goals. In other words, we had to consider the architectured components required for the above design goals. As we have explained, we are using a bitmapped display and cursor control as hardware components. We naturally decided to build the window system as a software component.

However, there is a question whether dialogue management should be entirely embedded into the window system, or whether dialogue management functions should be divided into smaller pieces. As noted earlier, the Lisp Machine selected the former method and the result is a very complicated window system. It is unmanageable both for the user and the program designer.

Our decision was the latter method. We restricted the window system to management of the very basic dialogue functions, and thus invented the "*Coordinator*" which manages the principal part of the dialogue between user and system. We left application-oriented dialogue management to the application program designer, although we provide the building blocks to make it work. For example, dialogue management in the text editor is left to the editor designer, although he can use existing blocks, such as the translation table or the whiteboard, which are explained in the next chapter.

## 4. Design of the Window System

### *1) The definition of a window*

For a user, each window is a viewport on a process (editor, debugger, etc.). The window of each process is a logical display, and each window is a rectangular area on the bitmapped display showing some process information.

### *2) Multi-window environment*

When using the computer, various information is shown by many processes. It is very helpful for users to be able to refer to such information at will. However, in ordinary systems, it is hard to see editor, compiler, and debugger information simultaneously. A multi-window environment solves this problem. Several windows are displayed on the screen at the same time, each window showing information from a different process. Using this environment, users can refer to all available information.

*3) Simplicity*

We designed the window system to manage only basic dialogue functions, i.e. control of keyboard and mouse inputs, control of the output to each window, and control of the window allocation on the screen. Input echoing is handled by the process itself. Input control means management of the connection between the keyboard and processes. The window system determines which window is connected to the keyboard, and passes the keyboard inputs to the process which owns that window. The window system then translates the keyboard's input codes into character codes or commands according to the *translation table* prepared for each process. The window system offers users the ability to manipulate the screen image. Such manipulations include *move_window*, *select_window*, and *reshape_window*.

*4) Window hierarchy*

Some process require several windows simultaneously. For example, an editor may use a text window for showing file contents, a command window for command input, and so on. In such a case, the windows are closely related and must be treated as a set. We offer the window hierarchy for this purpose. By this mechanism, some windows are defined as *inferior* windows to others, which are called *superior* windows. In the example above , the editor window is defined as the *superior* of the text window, the command window, and so on. Inferior windows are, obviously, located within the superior window. By manipulating the superior window, the user can treat inferior windows as a set. The most superior window is the full screen, and the normal windows (the editor window, etc) are inferior windows of the screen. Each window may have inferior windows (called sub-windows) within it, and each inferior window can have its own inferior windows.

*5) Window status*

Each window has one of the following five statuses:

| | |
|---|---|
| `selected` | : *connected to the keyboard.* Only one window can have this status. |
| `shown` | : *completely displayed on the screen.* The meanings of the mouse button clicks for this window, are defined in the window itself. The superior window must have shown status. |
| `exposed` | : *completely displayed on the superior window.* When the superior window does not have the shown status, even if the window is completely displayed on the screen it does not have shown status, but |

this status, .

overlapped : *partially or completely hidden in the superior window.*

This window is hidden by another inferior window of its superior window.

deactivated : *not managed by the window system.*

This window is not displayed on the screen until activated. However, its dot image is not destroyed; it is stored in the bitmapped memory area.

Please note that the difference between exposed and overlapped statuses is determined within the same level of the window hierarchy. Figure 1 shows the example of the statuses. In this figure, B1 and B2 are inferior windows of B, and C1 and C2 are inferior windows of C.

****************************************************

**Figure 1. should be inserted here.**

****************************************************

*6)Input/Output Control*

The input/output control is performed using the window status, as described above. The relationship between window statuses and input/output functions is given in Table 2.

****************************************************

**Table 2 should be inserted here.**

****************************************************

Our machine has a keyboard and a mouse as input devices. The keyboard is a standard text-input device. Input from the keyboard is normally echoed to the window, depending on the process. For example, passwords input during the log-in procedure must not be output for security reasons.

There is only one keyboard. The window system must recognize the selected window to which keyboard input is to be sent.

We also have the mouse cursor control, as an input device. The mouse can point to any location on the display screen. The position of the mouse determines the window to which the mouse button click is sent. (Here, the position of the mouse does not mean the physical position of the mouse device, but the position of the mouse cursor on the screen.) The mouse device computes the relative movement. The mouse button click applies only to the ***shown*** window, because the user cannot see the all of the "*not shown*" (exposed or overlapped) window, and the window cannot easily recognize which part of itself is hidden by another window. Therefore, even if the mouse button click causes a change in the window's output image, it may not be apparent to the user. If the user wants to communicate with the "*not shown*" window, he can select it with the mouse button or by using the system menu. When the mouse cursor is positioned outside the shown window, the default meaning of the mouse button clicks is taken into service.

The most typical use of the mouse is the selection of a menu item. The menu displays the input method, such as "*find and select*". For this purpose, we will provide a special *menu window*. The menu window displays several subjects, and by using the mouse, users can select the ones they want.

## 5. Design of the Coordinator

### *1) The form of the dialogue*

The form of the dialogue can be defined as the sequence of user inputs and system outputs. Here, user inputs are limited to keystrokes and mouse functions. This limitation results from the restrictions inherent in current input devices. In the future, we may be able to use other methods such as voice input, pressure sensitivity, and so on. Still, in general, the forms of dialogue are almost unlimited. At one extreme, from a human orientation, is the use of natural language for input/output. At the other extreme, is the use of the mouse as the sole means of input, which has been used to some degree in Xerox Star [Smith 82].

Our emphasis on simplicity in both design and implementation resulted in the multi-window system with a mouse cursor control. In the same way, we want to restrict the dialogue input to single key codes, mouse button clicks, and selection through the menu window. We do not wish to place any restrictions on the output.

This approach satisfies our design criteria for ease of implementation and minimum overhead, because it is easy for users to assign a special meaning to each key and mouse button click. It requires little processing overhead to decode the user input into machine executable form, and no overhead to output the information. Our command language is so simple that we need not implement a syntactic parser or a complicated error-correcting routine to process, parse, and decode the command language.

It might be argued that single keystroke commands may cause critical mistakes, but such mistakes can occur even using highly sophisticated commands, such as those in natural language. The advantages of a minimum number of keystrokes, we believe, will result in easy, efficient conversation.

The idea of the single keystroke command is not new. The EMACS editor, originally developed at MIT, is a very famous example. The Z editor, used in the TOOLS system, and the ZMACS editor, used in the Lisp Machine, use this kind of command system.

Arguments for iconic input have been deliberately rejected because of heavy use of the window system. One author claims it would be useful neither for our users nor our application builders. Its slow response and complicated screen configurations do not allow for efficient usage.

*2) Problems with our form of dialogue*

However, there remain problems with single keystroke commands.

Although there are many key codes available, (128, 256, or 512), many more will be necessary for efficient use of this type of command system. It may be necessary to introduce arguments and/or command modifiers. Otherwise, a context mechanism would have to be introduced to assign multiple commands to the same code according to the dynamic environment.

We have decided to adopt the latter solution. We consider the window as the context of the dialogue. The key-command assignment will be different for each window.

The another problem is that users may easily forget commands or keycodes. HELP facilities are necessary to remind users of keystrokes and mouse button click meanings.

*3) Separation of dialogue levels*

Although the user communicates with the PSI computer system as a whole, he can think himself as talking to a special component of the whole system. We call that component an *expert* in this paper.

We can define two levels of dialogue: expert and system. At the expert level, the user talks with an "expert". The commands use a specialized expert's vocabulary.

At the system level, the user talks to the whole system, i.e. he can select an expert, stop its execution, create a new expert, and KILL an expert. The set of commands in this level must be common throughout dialogue processing. To execute this level of commands, a special manager for the control of experts is needed. The *Coordinator* is such a manager. That is, system-level commands are sent to the Coordinator, while expert-level commands are sent to an expert.

This separation brings up another problem, how to distinguish between command levels, and who designs the commands. We have introduced a table of commands, which describes the keystrokes and mouse button clicks, the contents of commands, and their destinations. Each window designer can select his own command set. It includes the system-wide keystroke and mouse button click assignments for system-level commands. Further details are given in the implementation section.

*4) Management of dialogue*

Expert-level dialogue must be managed by the expert itself. At the system level, the Coordinator manages the dialog, i.e., it helps the user to change the target of the dialogue. It also supports the creation and deletion of the expert.

To avoid overhead, no general mechanism for recording the input/output sequence is provided. This is left to the expert designer. However, there is a mechanism for recording the experts with which the user has made contact.

Communication between experts can be controlled by themselves or by the Coordinator. One advantage of communication through the Coordinator is that the establishment of fixed communication paths is not necessary. Another method might be the introduction of virtual communication paths between experts.

## IV. IMPLEMENATATION

### 1. Overall features

Our language, ESP, has determined the main features of our implementation. That is, the class hierarchy with multiple inheritance and demons is used not only in coding, but also throughout the design implementation. Many basic constructs of the operating system kernel, such as programs, processes, ports, and pools, have also determined our style.[Hattori 83]

### 2. Implementation of the Window System

#### *1) Window Hierarchy*

The window hierarchy is constructed recursively. In other words, the screen has inferior windows, windows have sub-windows as their inferior windows, and sub-windows themselves have inferior windows.

This window hierarchy is not constructed at the basis of the class inheritance mechanism. It may change dynamically, whereas the class hierarchy is fixed at definition. We define the classes *as_superior* or *as_inferior*. The class, *as_superior*, has the slot, *inferiors*, which lists inferior windows pointers. The class, *as_inferior*, has the slot, *superior*, which holds the superior window pointer. By changing the contents of these slots, the user can dynamically add and remove inferior windows. Windows may overlap each other on the screen, and this is controlled within the superior window (which one is top, second, ....). Each window knows its own inferior windows, and manages their display priorities. The full screen is defined as special window and ordinary windows are its inferiors, so control can be used even for the full screen.

### *2) Keyboard input and translation table*

Each expert has its own command set and its key assignments may change with the context. Therefore, it is better to assign each key to a command related to a particular window. The window system provides a translation table to accomplish this.

The translation table is a class instance, stored in a slot of *e_window*. By rewriting the slot, the user can dynamically change the translation table.

We supply the class *default_translation_table* (Figure 2.) as the standard table and the user translation table inherits this class. The key code definition in the user's translation table overrides the definition of default_translation_table.

```
*****************************************************
           Figure 2. should be inserted here.
*****************************************************
```

### *3) Mouse Input*

The PSI machine has a mouse cursor control. The mouse cursor can move anywhere on the screen. A mouse movement may produce a change in a window connected to the mouse. The mouse button click is often used to control window allocation, window creation, and window deletion. This means that the mouse needs special controls other than that of the keyboard. Including the echoing to screen, most of the mouse control is incorporated in the window system. In mouse movement, only the difference from the last position is considered as the raw input from the mouse device. With each mouse movement, the window system calculates the new position and checks mouse_owning_window. Each process defines the meaning of the mouse button click by means of the translation table, and the meaning is valid only in that window. Each process may define the mouse cursor pattern shown in its window, but the mouse cursor is actually controlled by the window manager, not by the process. The window system checks mouse_owning_window pattern, and using this pattern, positions the cursor as indicated by the mouse. Since this is a high-level user interface, mouse cursor movement must smooth, which requires rapid processing of the mouse control.

### *4) Menu*

Menus are a very useful means of command input. There are many kinds of windows and a menu is a typical one. The menu window deals with neither keyboard input nor process text output. It displays various items and lets the user select one. The menu window appears at the current mouse position. The size of the menu window is determined by the number of items and by the largest item string. The programmer can define the menu window only inherited the menu classes and supplied with the items, such as lists of item strings and types. The window system determines the size and creates the bitmapped image. Figure 3 shows sample codes that define a system menu.

******************************************************

Figure 3. should be inserted here.

******************************************************

## 3. Implementation of Coordinator

The following items relate to Coordinator implementation:

- *registration of the expert that is the process the user communicates with*
- *representation of each expert and the set of experts*
- *management of experts: creation, selection, deletion, etc.*
- *keystroke input and mouse button click processing*
- *program communication support*

Expert registration, representation, and management depend on the data representation of the expert. The details are beyond the scope of this paper, and will appear in a forthcoming paper concerning the construction of the programming system kernel [Kurokawa 84]. Only an outline is presented in this section.

Registration is performed by coupling the expert name and body. The body is a program that works for the user and communicates with him through the window. The body will be executed in the expert process after creation of the expert. There can be any number of copies of the expert to do many different tasks. For example, there will be several text editors, one for each of the different types of text.

The expert is implemented through the special classes, namely *e_program* and *e_process*. They have special properties for communication with users and the Coordinator.

There is a special object, namely *e_process_pool*, which manages the current set of the experts. Using the object-oriented programming technique, e_process_pool is not simply a medium to store the e_processes, instead it handles methods, such as broadcast, for all the experts.

The expert window is special in the sense that it contains the translation table which decodes keystrokes and mouse button clicks into the character codes and the commands, then sends it to the destination. The translation table is actually an ESP program represented as a class. The code looks like the following:

********************************************************

Figure 4. should be inserted here.

********************************************************

In the above program, the translation table contains the fourth argument (the meaning of the command) as a string that will be used for the HELP program. The translation table codes are executed by the window manager.

The context of the conversation is represented by the expert window. Each expert has its own *e_window* in which the private command set is defined, and the object is implicitly defined in the communication, which can be stored in the *whiteboard*.

The whiteboard also accomplishes the communication between experts. The whiteboard is actually implemented as a circular stack containing expert messages. Using *e_window*, the user directs the expert to store a message in the whiteboard. (This is done by a keystroke or mouse button click.) He can also request the expert to get a message from the whiteboard.

The user directs an expert to set his current object in the whiteboard, and he also orders another expert to get the object from the whiteboard, and to perform some operation on it. In this way, an expert can communicate with any other expert and communication is under user control. If a set of experts want to communicate among themselves without user interaction, they must be designed to define their own communication channels.

## 4. Experience with ESP

One of the special features of our project is the introduction of logic programming. So far, DEC-10 Prolog has been the only language meeting the requirements of experienced logic programmers. PSI is a special machine for the logic programming language, KL0, which has the same basic mechanism as Prolog. It is natural that we use a logic programming language for the implementation.

However, we did not use machine language itself. We adopted the enhanced language, ESP, which has, as its main feature, a class mechanism similar to that of Smalltalk or Flavor, as previously noted. We adopted that language for its modularity, easy maintenance, expandability, and, critical factor, its graceful introduction of *states* into logic programming.

As is known of pure Prolog, the representation of a state, in its conventional meaning, is a difficult problem. Concurrent Prolog [Shapiro 84] provides another method, but it is not usable at the present time.

There are dozens of people with differing backgrounds, participating in the SIMPOS implementation. Also, we are on a tight schedule, and do not have enough personnel to manage the interface between our members. For all these reasons, we rely heavily on the class mechanism of ESP to give modularity to our project with little interface overhead.

It might be argued that the debugging of object-oriented language is rather difficult unless it has a sophisticated browser facility, as in Smalltalk [Goldberg 84]. Our feeling here is that, until we build a powerful window system, we cannot create an effective browser system.

From our experience, we can list the following merits:

*1) Size of source*

Our source codes are rather small compared with those of the Lisp Machine. For example, our window program is about 100,000 characters, and the Coordinator is about 20,000 characters. Although the Lisp machine's window system has many facilities (including those of our window system and Coordinator), its code contains more than 1,000,000 characters.

*2) Modularity*

Although we meet only once a week, we do not need complicated procedures to maintain consistency with each other. One reason is the fact that all source code is stored in one computer. We can check it, add comments, and send them, all via electronic mail.

*3) Maintenance*

We have found that existing code (actually classes) can be used for other purposes. For example, we used existing I/O programs for special programs, such as Initial Program Loader I/O program. The class mechanism seems to support such modifications. This common use of programs will facilitate system maintenance.

We think that our adoption of ESP is the right decision for our work.

## V. SUMMARY

### 1. Overall evaluation

We have not yet fully evaluated our dialogue management. However, using the criteria from Table 1, we can summarize our partial evaluation as shown in Table 3.

****************************************************

Table 3. should be inserted here.

****************************************************

### 2. Key items in design and implementation

We can list the following items which have been effective in design and implementation:

- ESP - an enhanced logic programming language using the class mechanism
- separation between windows and Coordinator - simple structure
- translation table to handle single keys and mouse button click command
- whiteboard to support program communication
- window hierarchy
- window status for input/output control

### 3. Future work

As yet, we have implemented only the kernel of dialog management. For the future, we are considering following facilities:

#### *1) Advanced input/output media*

As the technology advances, we can use many other useful devices for man-machine communication. Voice input, pressure-sensitive screen input, color graphics display, and synthesized voice output are candidates. We want to develop these technologies for use in advanced man-machine communication.

#### *2) Advanced help facility*

A limited HELP facility has been implemented; it is invoked by the user with an assigned key-code or from a command menu. However, a more advanced, more intelligent HELP function is needed to make our system easier to use. The use of natural language would be optimal. Precise understanding of the user's intention (in context) would be an extremely valuable addition.

*3) Undo/redo capability*

This capability will be incorporated after the introducion of session management. The basic capability of undo is that it would provide the recovery function over the entire SIMPOS system. The belief that logic programming backtracking would enable this is somewhat naive. Some type of processing, such as destructive system resource operations , should be, in principle, deterministic and another mechanism for undoing is required. Perhaps the key to the design of such a function would be a method of deciding the areas to which undo could, and could not, be applied.

*4) Session management*

To support the above undo/redo capability, we are thinking of introducing session management. For session management, all user input and/or all the command or character codes sent to the process are kept on record, along with all process output. This record is used for undo/redo operations, that is, the input for the reverse operation would be given, or the same input would be resent. The overhead of maintaining these inputs and outputs is the major problem. Another problem is that sometimes the user does not know whether his input will be accepted by session management or by the process itself.

It would be easy to implement such a session management facility at the user's front end, but it would be less efficient. At the back end, it might be implemented to work only for special commands, such as undo/redo. This would reduce processing time.

Session management also requires a text-editing capability. Using the translation table and existing (and forthcoming) facilities, it would not be difficult to implement the session management facility.

Thanks to the multi-window function, we need not implement a complex session manager.

## ACKNOWLEDGEMENT

Our thanks are due to Mr. Kazuhiro Fuchi, Director of ICOT Research Center, and Mr. Toshio Yokoi, Chief of our laboratory. We would also like to express our appreciation to members of our laboratory for many helpful discussions.

## REFERENCES

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

[Dzida 78] Dzida, W., "User - Perceived Quality of Interactive Systems", IEEE Vol. SE-4, No.4, pp.270-276, (Jul. 1978) also in Proc. 3rd ICSE, pp.188-195, (1978)

[Ellis 83] Ellis, J.R. Mishkin, N. van Leunen, M. and Wood, S.R. "Tools: an Environment for Time-shared Computing and Programming", SOFTWARE - Practice & Experience, Vol.13, pp.873-892, (Oct. 1983)

[Gaines 83] Gaines, B.R. and Shaw, M.L.G. "Dialog engineering" in Sime, M.E. and Coombs, M.J. *Designing for human-computer communication*, Academic Press, pp.23-53, (1983)

[Goldberg 84] Goldberg, A. "Smalltalk-80, The Interactive Programming Environment", Addison-Wesley, (1984)

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

[Hayes 82] Hayes, P. J., "Cooperative Command Interaction through the Cousin System", Proceedings of International Conference on Man/Machine Systems, pp.59-63, (July 1982)

[HFCS 82] Proceedings of a Symposium on Human Factors in Computer Systems, (Mar. 1982)

[Kurokawa 84] Kurokawa, T. and Tojo, S. "Coordinator - a kernel of the programming system of Personal Sequential Inference Machine (PSI)", forthcoming, (1984)

[Maguire 82] Maguire, M. "An evaluation of published recommendations on the design of man-computer dialogues", Int. J. Man-Machine studies, Vol. 16, pp.237-261, (1982)

[Norman 75] Norman, D. A., "Design Rules Based on Analyses of Human Errors", Comm. ACM, Vol. 26, No. 4, pp.254-258, (Apr. 1983)

[Norman 83] Norman, D. A., "DESIGN PRINCIPLES FOR HUMAN-COMPUTER INTERFACES", ACM CHI'83 Proceedings, pp.1-10, (Dec. 1983)

[Shapiro 84] Shapiro, E. "Systems Programming in Concurrent Prolog", ACM POPL, pp.93-105, (1984)

[Sime 83] Sime, M.E. and Coombs, M.J. "Designing for human-computer communication", Academic Press, (1983)

[Smith 82] Smith, D. C., Irby, C., Kimball, R. and Harslem, E. "The star user interface: an overview", Proc. NCC, pp.515-528, (1982)

[Stallman 84] Stallman, R. M., "EMACS: The Extensible, Customizable, Self-Documenting Display Editor", in Barstow, D. R., Shrobe, H. E., and Sandewall, E. (eds.) *Interactive Programming Environments*, McGraw-Hill (1984)

[Symbolics 81] The Lisp Machine Manual, Symbolics Inc. (1981)

[Taguchi 84] Taguchi, A., Miyazaki, N., Yamamoto, A., Kitakami, H., Kaneko, K., and Murakami, K. "INI: Internal Network in the ICOT Programming Laboratory and its Future --- Impact of the FGCS on Future Communication Networks ---", ICOT TM-0044,

(Feb. 1984)

[Takagi 84] Takagi, S., Yokoi, T., Uchida, S., Kurokawa, T., Hattori, T., Chikayama, T., Sakai, K., Tsuji, J. "Overall Design of SIMPOS - (Sequential Inference Machine Programming and Operating System)", Submitted to the Second International Logic Programming Conference (Feb. 1984)

[Thimbleby 82] Thimbleby, H., "Interactive Systems Design: A Personal View", Proceedings of International Conference on Man/Machine Systems, pp.118-122, (Jul. 1982)

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

[Watts 82] Watts, R. A., "A Friendly Interface for the Lay User", Proceedings of International Conference on Man/Machine Systems, pp.64-67, (Jul. 1982)

|  | Criterion | Reference |
|---|---|---|
| Design Principles | simplicity | [Shneiderman 80] |
|  | user modeling | [Gaines 83] |
|  | consistency | [Norman 75] |
|  | learning | [Dzida 78] |
|  | adaptablility | [Maguire 82] [Watts 82] |
| Principal Elements | command language | [Gaines 83] [Hayes 82] [Maguire 82] [Norman 83] |
|  | hardware | [Maguire 82] |
|  | user's control | [Dzida 78] [Gaines 83] |
| Advanced Functions | help | [Maguire 82] |
|  | feedback | [Norman 75] [Maguire 82] |
|  | undo/redo | [Norman 75] |
|  | customization | [Thimbleby 82] |

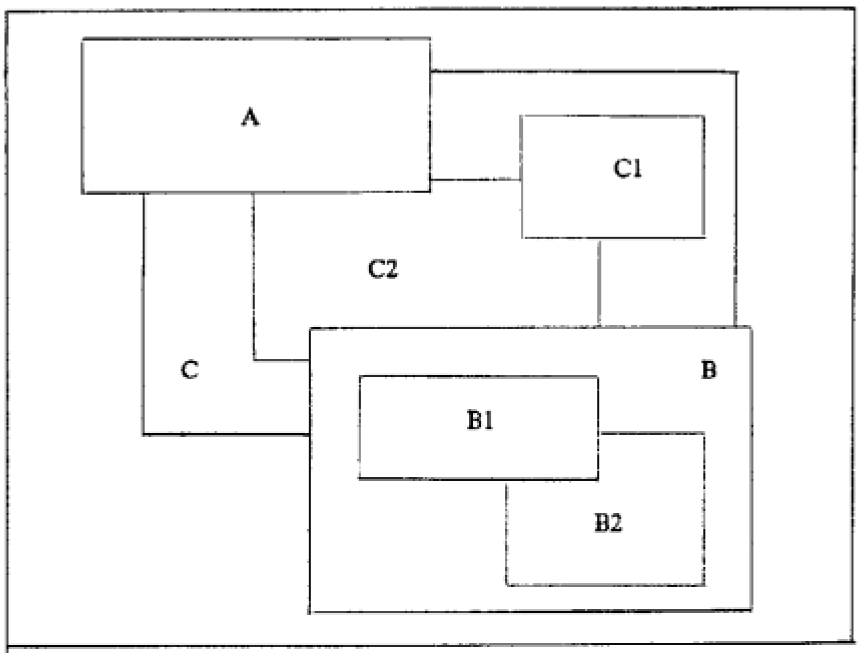
Table 1. Summary of criteria for dialogue management

|  | Input |  | Output |
|---|---|---|---|
|  | Keyboard | Mouse |  |
| SELECTED | done | done | done |
| SHOWN | waiting | done | done |
| EXPOSED | waiting |  | done |
| OVERLAPPED | waiting |  | waiting/done |
| DEACTIVATED | not available |  |  |

Table 2. Window status and input/output

| | Criterion | Partial Evaluation |
|---|---|---|
| Design Principles | simplicity | yes, this has been our goal |
| | user modeling | specialist |
| | consistency | depends on system/application programmers |
| | learning | must be easy |
| | adaptablility | there will be an instruction program for novices |
| Principal Elements | command language | single-key commands and/or mouse button clicks |
| | hardware | bit-mapped display, mouse |
| | user's control | the user holds the control |
| Advanced Functions | HELP | under construction |
| | feedback | multi-window |
| | undo/redo | future implementation |
| | customization | yes, using the class mechanism |

Table 3. Partial Evaluation of PSI Dialogue Management



|  |  |
|---|---|
| A: selected | C: overlapped |
| B: shown | C1: exposed |
| B1: shown | C2: overlapped |
| B2: overlapped |  |

Figure 1. Window Statuses

```
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, X, Port, X, "send character as it is");
   end.
```

Figure 2. Default Translation Table

```
class system_menu
   has
      component
         menu_list;
      :create(System_menu_class, Directory, System_menu) :- !,
             :new(System_menu_class, System_menu),
             :get_invariant_part(#system_menu, Menu_list),
             :new(#list, Experts_list),
             make_from_to(Directory, Experts_list),
             :add_first(Menu_list, Experts_list),
             :create(#multi_column_menu, Menu_list, Menu_window),
             System_menu!window := Menu_window;
      :get_invariant_part(System_menu_class, Menu_list) :-
             #system_menu!menu_list == 0, !,
             :new(#list, Menu_list),
             :new(#list, Media_system_list),
             :create(#menu_item, 'window', common, window, _, Window_item),
             :add_last(Media_system_list, Window_item),
             :create(#menu_item, 'file', common, file, _, File_item),
             :add_last(Media_system_list, File_item),
                          .
                          .
             :add_last(Menu_list, Media_system_list),
                          .
                          .
             #system_menu!menu_list := Menu_list;
                          .
                          .
                          .
   end.
```

Figure 3. System-Menu Definition

```
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#l, #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 4. A Translation Table Program in ESP