ICOT Technical Memorandum: TM-0100

TM-0100

Constructing the SIMPOS Supervisor in an Object-oriented Approach

Takashi Hattori (Oki Electric Industry),
Norihiko Yoshida (Mitsubishi Research Institute)
and Takumi Fujisaki (B-con Systems)

February, 1985

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

Constructing the SIMPOS Supervisor in an Object-Oriented Approach

Takashi HATTORI*, Norihiko YOSHIDA**, and Takumi FUJISAKI***

* Oki Electric Industry Co., Ltd.
4-11-22, Shibaura, Minato-ku, Tokyo 108, JAPAN

** Mitsubishi Research Institute
2-3-6, Ootemachi, Chiyoda-ku, Tokyo 100, JAPAN

*** B-con Systems
7-8-10, Nishi-Shinjuku, Shinjuku-ku, Tokyo 160, JAPAN

Abstract

SIMPOS (a programming and operating system for SIM) is constructed in an object oriented approach. This approach has reduced the specification and development efforts of the system. It also makes it easier to change and extend the system. This paper shows the effectiveness of this approach by taking the SIMPOS supervisor facilities -- process and stream -- as examples.

Table of Contents

Note: This work has been done as part of the SIMPOS (formally Sequential Inference Machine Software) project at ICOT (Institute for New Generation Computer Technology).

1. Introduction

SIMPOS [4] is a programming and operating system for SIM (Sequential Inference Machine) [1], [2] (or PSI specifically) which is a Prolog-based personal computer developed at ICOT. SIMPOS has been developed as a systems software prototype which is aimed for providing a good programming environment on PSI.

Before we started the design of SIMPOS, we had come to a conclusion that Prolog, as it is now, does not provide sufficient programming facilities to write an operating system in it. As a system programming language, Prolog lacks in modular programming framework and an efficient means for expressing state changes.

The underlying concept for constructing SIMPOS is an object-oriented approach in logic programming. The object-oriented approach, which has been proved as one of the useful schemes to describe system software [7], [8], [9], [10], provides both modular programming framework and the concept of states with objects. Furthermore, its encapsulation and inheritance mechanisms facilitate changes and extensions of the system. This section explains the object-oriented approach we have taken in SIMPOS.

(1) Objects in SIMPOS

An object, which is a building component in SIMPOS, is defined externally by a set of operations which the object accepts to perform given goals. Internally it is implemented by a set of Prolog clauses and slots. The clauses define what the object should perform when it accepts an operation, and each of the slots holds a value or an object which represents the state or structure of the object.

A set of objects which behaves in the same manner is defined as a class. A template of objects of the same class is given by a class definition, and an instance (object) is instantiated from this template.

Inheritance mechanism is provide to define a new class with other classes using 'is_a' relations. If a class inherits other classes, the class has externally all the operations of those classes and the operations defined in itself. Internally, the clauses of the same operation are OR'ed, and the slots are put together.

A demon predicate is a facility which makes object-oriented programming with multiple inheritance easier and more effective. Assume that a demon predicate is defined in a component class which a certain class inherits. When an operation on the object of this class is called, the demon predicate will be also called implicitly. This class does not need to call the operations explicitly. Two types of demon calls are supported; 'before' demons and 'after' demons. A 'before' demon is called before the primary predicate of the main class, and an 'after' demon is called after that. The demons are AND'ed with the primary predicate.

(2) ESP

ESP [3] is a Prolog-based object-oriented programming language, in which SIMPOS

is described. ESP provides the programming facilities described above. A class definition in ESP consists of the four parts in the following syntax.

```
  class  <class name>  has
      [ <nature definition> ; ]              ..... inheritance
      { <class slot definition> ; }          ...
      { <class clause definition> ; }        ..:.. class
[ instance
      { <instance slot definition> ; }       ...
      { <instance clause definition> ; } ]   ..:.. instance
[ local
      { <local clause definition> ; } ]      ..... local
  end.
```

The inheritance part defines the super classes which this class inherits. A list of the super classes can be specified. Their order is relevant, because the operation definitions given in these classes are OR'ed in this order. This class itself can be specified in this list.

The class part defines the class object. The class object is referred to when an object is created or when all the objects of this class have certain features in common. This part has two sub-parts. The slot definition defines the slots which this class object has, and the clause definition defines the operations which this class object performs.

The instance part defines the template of objects. This part consists of two sub-parts, too. The slot definition defines the slots which an instance of this class will have, and the clause definition defines the operations which this instance will execute.

The local part defines the predicates which can be called only within this class. It has been introduced so that Prolog predicates can be defined and executed in ESP.

2. Process and Inter-Process Communication

SIMPOS is largely divided into the programming system and the operating system. The programming system provides users with programming tools, including editors, an interpreter/debugger, and a librarian. The operating system [5] provides basic execution environments and input/output facilities. It is further divided into three layers: the i/o medium systems, the supervisor, and the kernel. The structure of SIMPOS is shown in Figure 2.1.

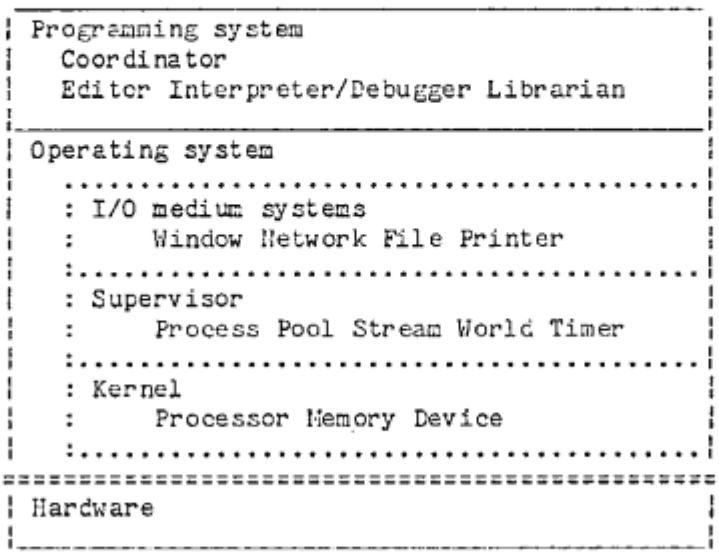

Figure 2.1 Layered Structure of SIMPOS

In order to explain the object-oriented construction of SIMPOS, we take as examples the process and stream managements which are included in the supervisor [6]. In this section, we will briefly describe the concepts and facilities of processes and streams.

2.1 Process Management

(1) Process

In SIMPOS, a process is an active entity which executes a program. A process is in one of the four states -- running, ready (to run), suspended, and dormant. A running process is the one which is currently executing a program on a processor. A ready process is in a ready queue waiting to be run. A suspended process is not in the ready queue. Its execution has been suspended for some reason, ordinarily when it tries to get an object from an empty stream, and when those reasons are removed, it becomes ready and is put into the ready queue. A dormant process is not managed by the supervisor.

A process is defined as an object which accepts operations to control its state:

- To activate a process
  :activate(Process,Program)
- To suspend a process
  :suspend(Process,Reason)
- To resume a process
  :resume(Process,Reason)
- To terminate a process
  :terminate(Process)

Some processes in the system are allocated to interrupt processes, such as trap handlers, device handlers, and the garbage collector. These interrupt processes are not dispatched by the supervisor, but by the hardware on an interrupt or a trap.

Another kind of special process is a boot process, which is created and activated by the hardware, not by another process, when the system is bootstrapped. The multiple process environment in SIMPOS is initiated on the boot process. The supervisor must make it under its control, after the system has started.

(2) Program

A program tells what a process should perform. It is an instance of a program class. When many processes execute the same program independently, each process is given a program instance of the same program class. These programs share the code but have a different set of instance slots. The entry of the main program is defined as an instance predicate of the program.

A program, after having been instantiated, is given to a process to be executed. This is called a program invocation. To keep track of program invocations, a process has an invocation stack. Each time a program is invoked, it is pushed into the stack, and becomes the current program of the process. A predicate for a program invocation is defined as:

```
:invoke(Program)
```

where 'Program' is a program class. The main program is specified by the instance predicate :goal(Program). When this predicate is called, a program instance is created and pushed into the invocation stack of the calling process. However, when a new process is activated, the activate predicate is called with a program as an argument, because the program instance should be pushed into the invocation stack of the new process, not of the calling process.

2.2 Stream Management

(1) Stream

A stream is a pipe through which objects flow. It is used for synchronization, communication, and mutual exclusion among processes. Two major operations on a stream are:

- To insert an object into the stream
  `:put(Stream,Object)`
- To remove an object from the stream
  `:get(Stream,Object)`

If the stream is empty, a get operation causes a calling process to be suspended until another process puts an object. This feature of streams is essential for process interactions. A stream can be shared by multiple processes. Each process can put or get objects from it at any time, and is

served on a first-in-first-out basis.

(2) Stream variations

Various types of streams with additional features are defined. For example, priority control and bounded buffer control are added. Also message-based communication facilities, channels and ports, are provided.

A channel is a message-based communication primitive. It is useful to describe multiple-client/multiple-server interactions. Two major operations on a channel are:

- To send a message to the channel with specifying a sender (channel) for a reply
  :send(Channel, Message, Sender)
- To receive a message from the channel with a sender returned
  :receive(Channel, Message, Sender)

A port is a message-box for two-way communication based on channels. Four major operations on a port are:

- To send a message to connected ports
  :send(Port, Message, Sender)
- To receive a message from the port
  :receive(Port, Message, Sender)
- To connect the port to another
  :connect(Port, Another_port)
- To disconnect each other
  :disconnect(Port, Another_port)

3. Implementation

This section shows the implementation of process and stream. Although we give several ESP codes as examples, they are not complete.

3.1 Process Implementation

(1) Process

Class 'process' defines instance predicates for controling process status, and class predicates for process creation and identification. Part of class 'process' in ESP is as follows.

```
-------------------------------------------------------------------------
class  process  has

nature      with_link,
            waiting_frame ;

attribute   ready_queue,
            instance_array ;
```

```
                    % Relates a process to its context_number.

:create(Class, Process) :- !,
            % Make an instance and initiate it.  If hardware processes
            % are exhausted, then fail.
        :create(#context, Context_number, Stack_number, Context),
        :create_core(PROCESS, Process, Context), ! ;

:create_core(Class, Process, Context) :-
        :new(Class, Process),
        :get_number(Context, Context_number),
        set_vector_element(#process!instance_array,
                                Context_number, Process),
        :self(#process, Self_process),
        Process!context := Context,
        Process!priority := 0,
        Process!sub_priority := 0,
        Process!parent_process := Self_process,
        :add_child(Self_process, Process),
        Process!world := Self_process!world,
        Process!user := Self_process!user,
        Process!status := dormant, ! ;

instance

attribute   context,    % Hardware PCB
            priority,   % Corresponds to interrupt level
            sub_priority,
            mask_count      := 0,
                    % The number of interrupt mask nestings
            suspend_reason  := nil,
          ( suspend_reasons := List :- :create(#list, List) ),
            suspend_flag    := nil,
            status          := nil,
                    % nil(=dead), dormant, suspended, ready, (active)
                    % and some others
            parent_process,
          ( children_processes  := List :- :create(#list, List) ),
            killer_process,     % which terminates this process
          ( invocation_stack    := List :- :create(#list, List) ),
          ( resources           := List :- :create(#list, List) ),
            world,
            user,
            interpreter_instance,
                    % Interpreter instance which runs on this process
          ( clock       := Clock :- :create(#process_clock, Clock) ) ;
                    % The elapse time of this process

:activate(Process, Program) :-
            % Give Program to Process, and make it ready.
        :status(Process, dormant, on_activate), !,
        :entry_of_invoke(Program, Entry),
        :set_entry_point(Process!context, Entry),
```

```
        :activate(Process!context),
        :set_priority(Process!context, 0),
        :clear_masks(Process),
        :inhibit_interrupt(#process),
        Process!status := ready,
        :add(#process!ready_queue, Process),
        :allow_interrupt(#process), ! ;

:resume(Process, Reason) :-
            % Remove Reason from 'suspend_reasons'.  If there's no
            % such Reason, then fail.  If 'suspend_reasons' becomes
            % empty, then make Process ready.  If already resumed,
            % then do nothing.
        :inhibit_interrupt(#process),
        %CHECK REASON
        (   Process!status == suspended, !,
            (   Reason = stream(Stream), !,
                Process!suspend_reason := nil,
                (   Process!suspend_flag == nil, !,
                    Process!status := ready,
                    :add(#process!ready_queue, Process)
                ;   true ),
                :allow_interrupt(#process)
            ;   :remove(Process!suspend_reasons, Reason), !,
                (   :empty(Process!suspend_reasons), !,
                    Process!suspend_flag := nil
                ;   true ),
                (   Process!suspend_reason == nil,
                    Process!suspend_flag == nil, !,
                    Process!status := ready,
                    :add(#process!ready_queue, Process)
                ;   true ),
                :allow_interrupt(#process)
            ;   :allow_interrupt(#process), !,
                fail )
        ;   Process!status == ready, !,
            :allow_interrupt(#process)
        ;   :allow_interrupt(#process), !,
            fail ),
        ! ;

:suspend(Process, Reason) :-
            % Add Reason to 'suspend_reasons'.  If Process is ready,
            % then make it suspended.
        :inhibit_interrupt(#process),
        %CHECK REASON
        (   Process!status == ready, !,
            (   Reason = stream(Stream), !,
                Process!suspend_reason := Stream
            ;   :add_last(Process!suspend_reasons, Reason),
                Process!suspend_flag := on ),
            Process!status := suspended,
            :remove(#process!ready_queue, Process),
```

```
            :allow_interrupt(#process)
        ;   Process!status == suspended, !,
            (   Reason == stream(Stream), !,
                Process!suspend_reason := Stream
            ;   :add_last(Process!suspend_reasons, Reason),
                Process!suspend_flag := on ),
            :allow_interrupt(#process)
        ;   :allow_interrupt(#process), !,
            fail ),
        ! ;

end.
```

Figure 3.1 Part of class 'process'

As shown above, class 'process' inherits two classes for chaining process instances; class 'with_link' is inherited to put the process into the ready queue, while class 'waiting_frame' is to put the process into the waiting queue of a stream.

(2) Program

A program executed by a process must be an instance of some program class which inherits class 'as_program' which gives a basic framework for programs. On process activation, the goal predicate defined in each program class is automatically invoked. For example, a program that reads characters and writes them on a window is coded as bellow.

```
class  parrot  has

nature      as_program ;

instance

:goal(Program) :-
        :create(#window, Window),
        :activate(Window),
        loop(Window) ;

local

loop(Window) :-
        :read(Window, Char),
        (   Char == etx, !
        ;   :write(Window, Char),
            fail ) ;
loop(Window) :- !,
        loop(Window) ;

end.
```

Figure 3.2 An example program: class 'parrot'

To execute this program by a new process, you must do as:

```
:create(#parrot, Parrot_Program),
:create(#process, Process),
:activate(Process, Parrot_Program)
```

On invocation of a program, the program is pushed onto the invocation stack of the process and the instance predicate :goal(Program) is called. When the goal finishes, the program is popped from the invocation stack. If the stack becomes empty, the process terminates.

Class 'as_program' is partially defined as:

```
--------------------------------------------------------------------------
class  as_program  has

:create(Class, Program) :- !,
        :new(Class, Program), ! ;

instance

:goal(Program) ;
            % A top-level predicate with a default name.
            % This must be overridden by a user program.

:assign(Program) :- !,
            % Push Program into 'invocation_stack'.
        :self(#process, Process),
        :add_program(Process, Program), ! ;

:resign(Program) :- !,
            % Pop Program from 'invocation_stack'.  If it is the root
            % program, then the process dies.
        :self(#process, Process),
        :remove_program(Process, Any_program),
        (   :no_program(Process), !,
            :die(Process)
        ;   true ), ! ;

:resign_core(Program) :- !,
            % Pop Program from 'invocation_stack'.
        :self(#process, Process),
        :remove_program(Process, Any_program), ! ;

:invoke(Program) :- !,
            % Invoke the default goal.
        :assign(Program),
        (   :goal(Program), !,
            :resign(Program)
        ;   :resign(Program),
            fail ), ! ;

end.
```

Figure 3.3 Part of class 'as_program'

(3) Interrupt process

An interrupt process is instantiated from class 'interrupt_process', which inherits class 'process' since an interrupt process is a special kind of process. Unlike an ordinary process, an interrupt process is resumed with an interrupt or a software trap, and when suspended, it releases the processor. Thus, class 'interrupt_process' overrides the instance predicates :suspend and :resume of class 'process'. Here is part of class 'interrupt_process'.

```
class  interrupt_process  has

nature      process ;

attribute   context_stack,
            stack_pointer ;

instance

attribute   interrupt_index ;

:resume(Process, Reason) :-
        :inhibit_interrupt(#process),
        %CHECK REASON
        (   Process!status == suspended, !,
            (   Reason = stream(Stream), !,
                Process!suspend_reason := nil,
                (   Process!suspend_flag == nil, !,
                    Process!status := ready,
                    %TRAP TO PROCESS
                        ...
                ;   true ),
                :allow_interrupt(#process)
            ;   :remove(Process!suspend_reasons, Reason), !,
                (   :empty(Process!suspend_reasons), !,
                    Process!suspend_flag := nil
                ;   true ),
                (   Process!suspend_reason == nil,
                    Process!suspend_flag == nil, !,
                    Process!status := ready,
                    %TRAP TO PROCESS
                        ...
                ;   true ),
                :allow_interrupt(#process)
            ;   :allow_interrupt(#process), !,
                fail )
        ;   Process!status == ready, !,
            :allow_interrupt(#process)
        ;   :allow_interrupt(#process), !,
            fail ), ! ;
```

```
:suspend(Process, Reason) :-
            % Add Reason to 'suspend_reasons'.  If Process is ready,
            % then make it suspended.
        :inhibit_interrupt(#process),
        %CHECK REASON
        (   Process!status == ready, !,
            (   Reason = stream(Stream), !,
                Process!suspend_reason := Stream
            ;   :add_last(Process!suspend_reasons, Reason),
                Process!suspend_flag := on ),
            Process!status := suspended,
            %RELEASE PROCESSOR
                ...,
            %ALLOCATE PROCESSOR
                ...,
            %SET REASON PROPERLY
            Process!suspend_reason := nil,
            :clear(Process!suspend_reasons),
            Process!suspend_flag := nil,
            Process!status := ready,
            :allow_interrupt(#process)
        ;   Process!status == suspended, !,
            (   Reason == stream(Stream), !,
                Process!suspend_reason := Stream
            ;   :add_last(Process!suspend_reasons, Reason),
                Process!suspend_flag := on ),
            :allow_interrupt(#process)
        ;   :allow_interrupt(#process), !,
            fail ), ! ;

end.
```

Figure 3.4 Part of class 'interrupt_process'

As shown above, the interrupt handling mechanism is encapsulated in class 'interrupt_process'. Anyone who does :resume(Process, ...) need not be concerned at all with whether scheduling or a trap takes place, that is, whether Process is an ordinary process or an interrupt process.

(4) Boot process

A boot process is a sole instance of class 'boot_process', which executes the boot program inheriting class 'as_boot_program'. After initiation, this boot process goes under control of the supervisor, and acts exactly as an ordinary process. For this to be possible, class 'boot_process' inherits class 'process' and overrides the predicates, :create and :activate.

3.2 Stream Implementation

(1) Stream

A stream consists of two queues; a queue of processes waiting for events (in

the case of stream, for an object being put) and a queue of objects which are waiting to be taken out by processes.

Class 'waiting_queue' defines queues of the first. This queue is constructed on a doubly-linked list, and frames (entries) in the queue must be instances of class 'waiting_frame'. Class 'process' inherits class 'waiting_frame'. Class 'post_queue' defines queues of the second. Any class of objects, including integers, atoms, and strings, can be put into the queue.

A stream has a 'post_queue' slot which has an instance of class 'post_queue', rather than it inherits class 'post_queue'. It is because a channel, which is a variation of stream, will have a post queue for messages different from that of a stream. Class 'channel' will inherit class 'stream' with overriding its 'post_queue' slot (see 3.2 (2)). If class 'stream' has inherited class 'post_queue', it would be difficult to implement a channel in such a way.

A stream also has a 'waiting_queue' slot which has an instance of class 'waiting_queue'. It might be better that class 'stream' inherits class 'waiting_queue' rather than that it has the slot. But we chose to have the slot because of some subtle problem when initiating a stream.

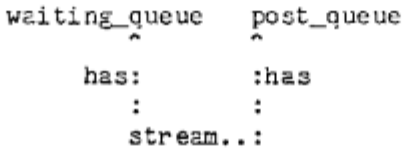

Figure 3.5 Class relations of class 'stream'

Part of class 'stream' is shown in Figure 3.6.

```
class  stream  has

:create(Class, Stream) :-
        :new(Class, Stream);

instance

attribute ( waiting_queue    := Queue :- :create(#waiting_queue, Queue) ),
          ( post_queue       := Queue :- :create(#post_queue, Queue) );

:put(Stream, Object):-
        :inhibit_interrupt(#process),
        :add_object(Stream!post_queue, Object)
        (   :empty(Stream!waiting_queue), !,
        ;   :unhook(Stream!waiting_queue, Process),
            :resume(Process, stream(Stream)) ),
        :allow_interrupt(#process);

:get(Stream, Object):-
        :inhibit_interrupt(#process),
        (   :remove_object(Stream!post_queue, Object), !
```

```
        ;   :self(#process, Process),
            :hook(Stream!waiting_queue, Process),
            :suspend(Process, stream(Stream)),
            :remove_object(Stream!post_queue, Object), ),
        :allow_interrupt(#process);

end.
```

Figure 3.6 Part of class 'stream'

(2) Stream variations

Several variations of streams are defined on top of class 'stream', which is a primitive class for all other stream type classes. ESP is powerful enough to implement these stream variations quite easily. They are defined by inheriting class 'stream' and changing or adding functions of it.

(a) Channel

A channel is different from a stream in that it has a post queue for messages rather than for general objects and that it handles a sender of messages.

Class 'channel' is defined by inheriting class 'stream' and adding several predicates. Part of class 'channel' is shown in Figure 3.7. Although a channel uses the predicates and slots inherited from a stream, the slot 'post_queue' is overridden in the channel, so that the post queue of the channel can manage messages more efficiently.

```
class  channel  has

nature      stream;

instance

attribute ( post_queue  :=
                    Queue :- :create(#post_queue_for_message, Queue) );
                % Overrides 'post_queue' of class 'stream'

:send(Channel, Message, Sender) :-
        :set_sender(Message, Sender),
        :put(Channel, Message);

:receive(Channel, Message, Sender) :-
        :get(Channel, Message),
        :get_sender(Message, Sender);

end.
```

Figure 3.7 Part of class 'channel'

The predicates, :send and :receive, are defined by using :put and :get of class 'stream'. Note that all the predicates of class 'post_queue_for_message' which

are called by a channel must be compatible to those of class 'post_queue' called by a stream. A message is defined by class 'message'.

(b) Port

Class 'port' is easily implemented by inheriting class 'channel'. A port has an additional slot called 'out_port' which contains a list of connected ports. Class 'port' adds its own predicates such as :connect and :disconnect.

```
class  port  has

nature      channel;

instance

attribute ( out_port    := List :- :create(#list, List) );

:send(Port, Message, Sender) :- !,
        :count(Port!out_port, Count),
        (   Count == 1, !,
            :get_first(Port!out_port, Out_port),
            :channel:send(Out_port, Message, Sender)
        ;   :stick_tap(Port!out_port, Tap),
            send(Port, Tap, Message, Sender) );

local

send(Port, Tap, Message, Sender):-
        (   :get(Tap, A_port),!,
            :copy(Message, New_message),
            :channel:send(A_port, New_message, Sender),
            send(Port, Tap, Message),!
        ;   true );

end.
```

Figure 3.8 Part of class 'port'

(c) Bounded-buffer function

A bounded channel, which is a channel with a bounded buffer, is used for buffered message communication. It has two waiting queues; one is for processes waiting for messages and the other is for processes waiting for buffers. The slot 'post_queue' in class 'bounded_channel' contains a bounded post queue instead of a post queue by overriding the slot of class 'channel'.

The class relations of a bounded channel is illustrated as follows. A semaphore which is a component of a bounded post queue inherits a waiting queue.

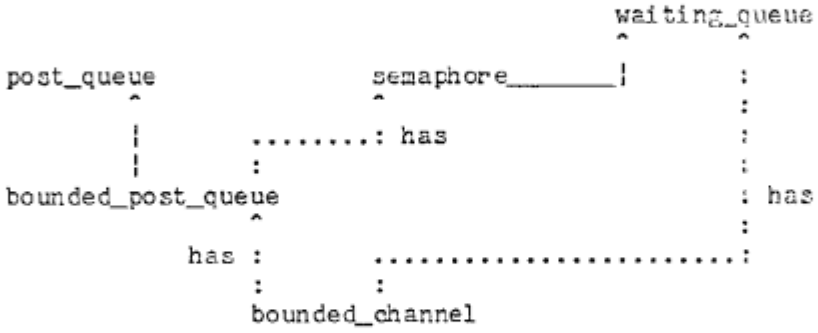

Figure 3.9 Class relations os class 'bounded_channel'

The definitions of class 'bounded_channel' and class 'bounded_post_queue' are shown in Figure 3.10 and Figure 3.11. The latter is an example of using demon predicates.

```
-------------------------------------------------------------------------
class  bounded_channel  has

nature      channel;

:create(Class, Channel, Limit) :-
        :create(Class, Channel),
        :create(#bounded_post_queue, Queue, Limit),
        Channel!post_queue := Queue;

instance

attribute   post_queue;

end.
-------------------------------------------------------------------------
```

Figure 3.10 Class 'bounded_channel'

```
-------------------------------------------------------------------------
class  bounded_post_queue  has

nature      post_queue;

:create(Class, Queue, Limit) :-
        :create(Class, Queue),
        :create(#semaphore, Semaphore, Limit),
        Queue!semaphore := Semaphore;

instance

attribute   semaphore;

before
:add_object(Queue, Object) :-
        :p_operation(Queue!semaphore);
```

```
after
:remove_object(Queue, Object) :-
        :v_operation(Queue!semaphore);

end.
```

Figure 3.11 Class 'bounded_post_queue'

4. Development History

The design of SIMPOS was begun at ICOT in the fall of 1982. The functional specification was prepared at the end of fiscal 1982, and the class specification was completed at the end of fiscal 1983. In parallel with these activities, ESP is defined and implemented on a cross system.

The first PSI, which was produced in December 1983, was made available to the software group in March 1984, and other PSIs later. Single-process environment supports were made available in April and enabled simple program debugging on PSI. In May, the IPL became operational, so as to allow linking programs directly on PSI. With multiple-process environment facilities which were supported in June, each subsystem was able to be fully debugged.

When the window subsystem was runnable on PSI for the first time, we measured the processor time of character input/output and mouse tracking for performance evaluation. The result was quite unsatisfactory. A character input with echoing took a few seconds. The reasons were analyzed to be overhead of inter-process communication and process switching, the speed of the PSI micro-interpreter, and overhead of object-oriented calls and slot accessing implemented in software.

To remedy this situation, we redesigned and recoded the supervisor and the window system. This reconstruction of the system was quite easily done (actually in a few weeks) without much affecting the rest of the system, because of modularity of the object-oriented approach in ESP. Also, the firmware group of PSI improved the micro-interpreter and implemented object-oriented calls and slot accessing in firmware. With these efforts, the time of a character input has become 30 ms. The process switching time, for example, has been reduced from about 20 ms to 2 ms.

5. Conclusion

From our experience with SIMPOS, we believe that the object-oriented approach is an effective means of reducing the effort of both specification and implementation. Although one of the well-known drawbacks of this approach is the overhead originated from its dynamic nature of execution, it is excusable as far as the system is running on a powerful personal workstation, considering it provides a flexible and extensible system as needed for a good programming environment. However, further research and development on object-oriented systems should be pursued for wider applications.

Acknowledgements

We would like to thank S.Saito (E-con Systems), H.Watanabe, M.Tatsishi (both Oki Electric), and H.Shimazu (NEC) who were the members of the SIMPOS supervisor group, as well as the members of the 3rd Lab. of ICOT who were involved in the project of SIMPOS and PSI.

References

[1] S.Uchida, et al., "Outline of the Personal Sequential Inference Machine PSI", New Generation Computing, vol.1, no.1, 75-79 (1983).
[2] T.Chikayama, "KL0 Reference Manual", to appear as ICOT TR.
[3] T.Chikayama, "ESP Reference Manual", ICOT TR-044 (Feb. 1984).
[4] S.Takagi, et al., "Overall Design of SIMPOS", ICOT TR-057 (April 1984) and the Proceedings of Second International Logic Programming Conference (July 1984).
[5] T.Hattori, et al., "SIMPOS: An Operating System for a Personal Prolog Machine PSI", ICOT TR-055 (April 1984).
[6] T.Hattori and T.Yokoi, "The Concepts and Facilities of SIMPOS Supervisor", ICOT TR-056 (April 1984).
[7] A.Goldberg and D.Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley (1983).
[8] D.Weinreb and D.Moon, "Flavors: Message Passing in the Lisp Machine", MIT A.I.Lab. A.I.Memo No.602 (Nov. 1980).
[9] D.G.Bobrow and M.Stefik, "The Loops Manual", Xerox Corp. (1983).
[10] G.Curry, et.al., " Traits: An Approach to Multiple-Inheritance Subclassing", ACM (1982).