ICOT Technical Report: TM-0065

TM-0065

An Operating System for Sequential Inference Machine PSI

by
Takashi HATTORI, Toshiaki KUROKAWA,
Ko SAKAI, Junichiro TSUJI,
Takashi CHIKAYAMA,
Shigeyuki TAKAGI, Toshio YOKOI

June, 1984

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

An Operating System for Sequential Inference Machine PSI

Takashi HATTORI, Toshiaki KUROKAWA, Ko SAKAI, Junichiro TSUJI,

Takashi CHIKAYAMA, Shigeyuki TAKAGI, and Toshio YOKOI

Institute for New Generation Computer Technology

Mita Kokusai Building 21F

1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN

Abstract

SIMPOS (Sequential Inference Machine Programming and Operating System) is a programming and operating system for sequential inference machine PSI under development at ICOT. This paper disousses the reasons for adopting an object-oriented approach in SIMPOS, and explains the construction of the SIMPOS operating system by using several examples. The current state of development is also given in brief.

Table of Contents

1. Introduction

SIMPOS (Sequential Inference Machine Programming and Operating System) [4] is a programming and operating system for sequential inference machine PSI [1] under development at ICOT. PSI is considered to be both a workstation for research and development in logic programming, and a pilot model for future knowledge information processing systems (KIPS). SIMPOS was designed and is being developed as system software suitable for these dual purposes.

The processor and main memory are designed in such a way that PSI can execute fast and efficiently its Prolog-based machine language (Kernel Language version 0, or KL0) [2]. In addition to Winchester and flexible disk drives, it is equipped with a bit-mapped display and an optical mouse in order to enhance the man-machine interface. It also supports a local area network (LAN) so that resources and information can be easily shared among PSIs.

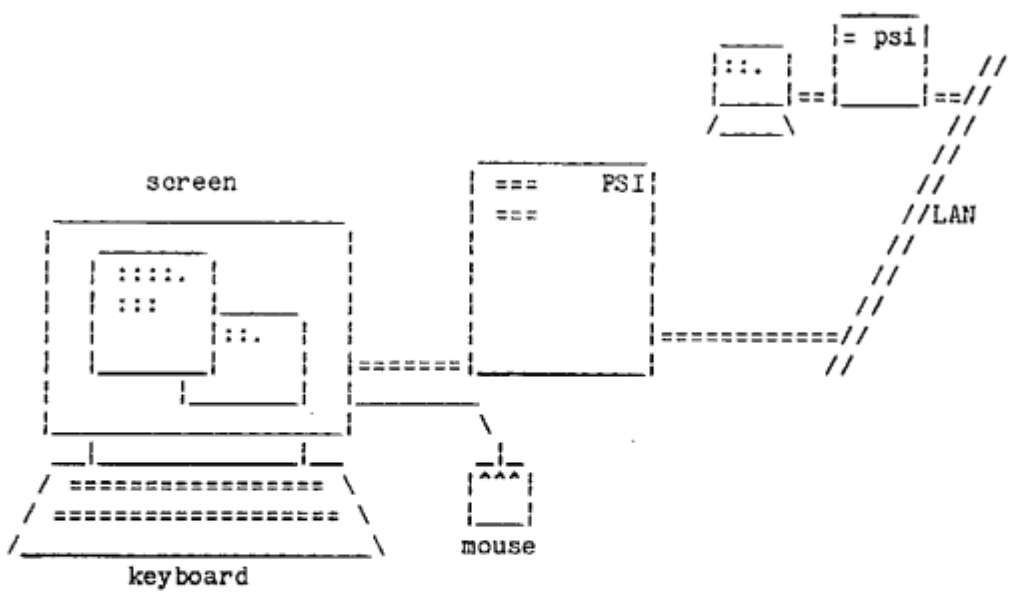

Figure 1.1 General View of PSI

SIMPOS aims mainly at providing an excellent programming environment by effective use of these hardware devices. The SIMPOS programming system includes:

- o program/text editing facilities
- o program debugging facilities
- o program library facilities

The operating system [5], underlying the programming system, supports:

- o multi-processing
- o multiple windows
- o filing and directory management
- o networking

In the following, we will discuss the conceptual framework of the SIMPOS operating system and its structure, and give several implementation examples.

2. Conceptual Framework

In designing SIMPOS, priority was given to simplicity and extensibility, rather than to variety of facilities. It is partially because we could not design and develop all the possible facilities at once given our limited man-power and schedule, and also because we thought it more important to clarify the conceptual framework underlying the system structure.

If an operating system is designed only from the functional aspect of the system without any conceptual framework on which to base these facilities, the system will have sufficient functions for the time being, but it may be too rigid to be modified and extended in the future. As SIMPOS will be

continuously modified to provide various functions, we first selected the conceptual framework of SIMPOS, then specified the required facilities based upon it. Our conceptual framework is an object-oriented approach, and our programming language is ESP [3].

2.1 Object-Oriented Approach

Prolog, from which the kernel language KL0 was derived, has two drawbacks for describing an operating system.

One of these is that Prolog cannot express side-effects. (Although Prolog has the built-in predicates, 'assert' and 'retract', to express side-effects, they cannot be implemented in (pure) Prolog. These predicates are not built-in to KL0, because they are too complex to be implemented as machine instructions.) The concept of state is necessary in the operating system, which manages and controls the various system resources. Side-effects give us an easy and efficient means for representing states. It is possible, but not feasible, to express states as Prolog terms and to simulate state transitions by creating a new term for each time.

The other drawback is that Prolog lacks a modularization mechanism. Software developed and used by many people, such as an operating system, requires some method of modular programming.

We have chosen an object-oriented approach as the means by which to solve both of these problems at the same time. This approach has been effectively adopted in other systems [10][11].

(1) Object

SIMPOS defines an object externally by a set of operations which are allowed on the object (or which the object accepts), and internally by a set of clauses and slots. The clauses describe the operation procedures, and the slots hold values or other objects to express the state. When an operation is performed on it, the object executes the clauses of the operation, refering to and changing the contents of the slots. It should be noted that as long as the external definition remains unchanged, the internal definition of an object can be modified without affecting the callers of the object.

An object is represented as a (heap) vector in KL0, and is identified by its location, used as an object pointer. The first element of the vector holds the KL0 code that is executed when an operation is called. The other elements are used for slots. Figure 2.1 illustrates the representation of an object.

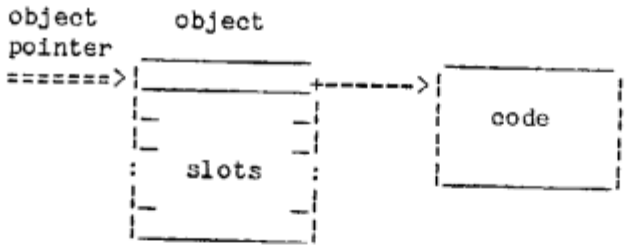

Figure 2.1 Representation of Object

(2) Class

A set of objects that behave in the same manner is defined as a class. A class definition gives a template for all objects of the same class; each object is instantiated from this template.

(3) Inheritance

Inheritance allows a new class to be defined using other classes. If class 'a' inherits class 'b' and class 'c', class 'a' has, externally, all the operations of 'b' and 'c', in addition to the operations defined in itself. Internally, the clauses of the same operation are ORed, and the slots are combined.

Inheritance is an 'is_a' relation between classes. A 'has_a' relation (or the reverse of 'is_part_of') is another kind of relation. If class 'a' has class 'b' (or more precisely, has an instance of class 'b'), class 'a' is said to 'have_a' class 'b'. We will not introduce the 'has_a' relation in SIMPOS; rather, the 'is_a' relation is used for comparable constructions. Class 'a_having_b' is defined by inheriting both class 'a' and class 'with_b' which has an instance of class 'b'.

(4) Demons

When an object consists of several component objects, an operation on the object may act on these components as well. If many classes are to have these components, it is unwise that each of the classes should define explicitly the operations performing on its components. Instead these classes should be constructed simply by inheriting a class that has these components and defines operations on them. Demons (demon calls) provide an easy way that allows such class constructions.

Assume that demon predicates are defined in the inherited component classes. The demons are ANDed with the primary predicate of the inheritor class. (Note that the inherited primary predicates are ORed as mentioned above.) When the operation on the object is called, the demon predicates are

also called implicitly. Two types of demon calls are supported: 'before' demons and 'after' demons. A 'before' demon is called before the primary predicate, and an 'after' demon is called after the primary predicate.

It should be pointed out that 'before' and 'after' demons are not always sufficient to construct a class having many component classes using multiple inheritance.

2.2 ESP

ESP is a Prolog-based object-oriented programming language used to describe SIMPOS. A class definition in ESP is given in the following syntax:

```
class <class_name>
    [ <macro bank declaration> ]
has
    [ <nature definition> ; ]
    { <class slot definition> ; }
    { <class clause definition> ; }
[ instance
    { <instance slot definition> ; }
    { <instance clause definition> ; } ]
[ local
    { <local clause definition> ; } ]
end.
```

As seen above, the class definition consists of five parts: macro, inheritance, class, instance, and local. The macro part does not concern us here.

The inheritance part lists the component classes which this class inherits. The clause definitions given in these component classes, including those defined in this class, are ORed in the order in which they are listed.

The class part defines the class object that is used to create objects of this class or to refer to the features common to all objects of this class.

This part has two sub-parts. The slot definition defines the slots of the class object, and the clause definition defines the operations that the class object accepts.

The instance part defines the object template. This part also consists of two sub-parts. The slot definition defines the slots that an instance of this class will have, and the clause definition defines the operations that this instance will accept.

The local part defines Prolog predicates that can be called only within this class definition.

3. System Construction

The facilities that the operating system supports are defined as classes and provided as the instances of these classes. This section discusses how the operating system is constructed with these classes.

3.1 System Constructs

Objects of the classes defined by the operating system are called system constructs. The basic features of the operating system will be provided only with a limited number of the elementary system constructs, each having minimal facilities. Additional facilities can be provided on top of these elementary classes. The selection of elementary constructs is crucial to achieve a simple and consistent operating system. A processing model is introduced to represent the system, and the operating system defines the classes to implement this model. Two aspects of the model are program execution and input/output.

(1) Execution model

In SIMPOS, an execution entity is called a process. Many processes may exist in the system. Within its own execution environment, each process executes the a given program which performs operations on objects and calls Prolog predicates. When necessary, a process can communicate with other processes to work in cooperation. A process may also have a collection of objects in a pool.

To support this model, the following elementary classes are provided:

- o process -- execution entity
- o stream -- inter-process communication
- o pool -- object storage
- o world -- execution environment

Several classes, such as 'processor' and 'area', are defined to represent the hardware resources, but they are not be used directly by application programs.

(2) Input/output model

A process performs input/output to communicate with the outer world. A window is used to communicate interactively with the user, a file is used to store and retrieve data in disk storage, and a network is used to communicate with other machines. I/O operations are implemented as operations on the objects representing windows, files, and network.

The physical i/o devices of PSI are also represented as objects. Classes, such as 'disk' and 'keyboard', are defined.

3.2 System Structure

The operating system divides system constructs into three layers: i/o medium systems, the supervisor, and the kernel (see Figure 3.1). The i/o medium systems support logical input/output [7] [8] [9], the supervisor supports the execution model [6], and the kernel manages hardware resources, including physical input/output facilities. System constructs are hierarchical, i.e., those defined in the lower levels are used by the upper levels, but not vice versa. This layer approach clarifies the structure of the system.

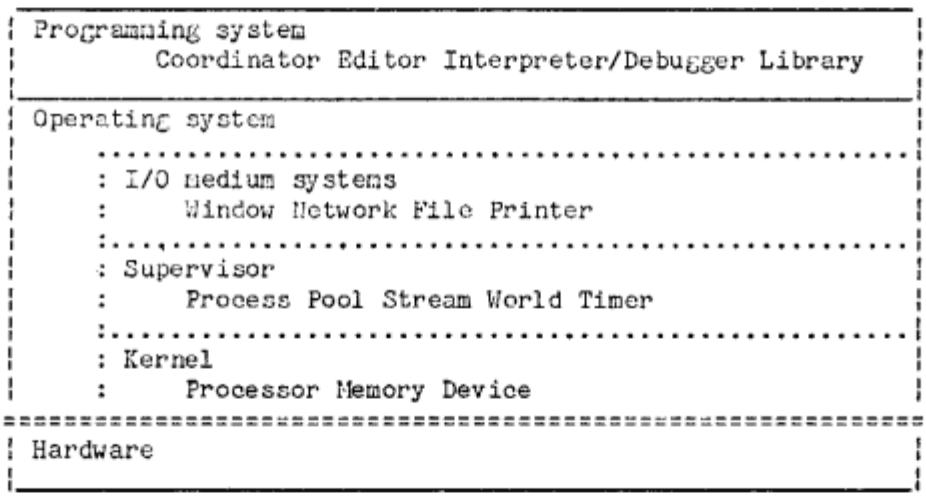

Figure 3.1 Layered Structure of SIMPOS

4. Implementation Examples

In this section, some implementation examples are given to show how the SIMPOS operating system is constructed using an object-oriented approach.

4.1 Simple Class

We take class 'file' as a simple example of a system construct. This class provides the facilities for accessing a file in disk storage. Part of the class definition is:

```
class   file    has

attribute
        pool_of_files;

:create(Class, File):- !,
        :new(Class, File);
                % Instantiate an object.

:make(Class, File, Pathname):- !,  % Create a new file.
        :new(Class, File),
        :create_file(File, Pathname);

:open(Class, File, Pathname):- !,  % Open an existing file.
        :retrieve(Class, File, Pathname),
                % Retrieve the physical file by its pathname.
        :open(File);

instance

attribute
        region,                 % keeps the physical file region
        opened := no,           % open/close flag
        disposed,               % dispose flag
        io_status;

:create_file(File, Pathname):- ...;
        % Create the physical file by the given pathname.

:open(File):- ...;              % Open the file.

:close(File):- ...;             % Close the file.

:dispose(File):- ...;           % Dispose the file.

end.
```

A file accepts operations such as 'open' and 'close' as defined above, and has several slots ('region', 'opened', 'disposed' and 'io_status') to represent various attributes. The class object of class 'file' accepts operations such as 'make' and 'open'. It has the class slot 'pool_of_files', which manages

opened files.

Note that class 'file' actually inherits several classes, including the classes 'as_permanent_object', 'as_resource', and 'with_lock'.

4.2 Inheritance

Inheritance is essential for defining various classes that have additional features to those of the elementary classes.

(1) Single inheritance

Although ESP supports multiple inheritance, it is still possible and often easy to produce a system construct (or to define a class) using single inheritance. With single inheritance, a new class is defined by inheriting a superclass and modifying it. Modifications are made by adding new operations (clauses and slots) and by overriding the defined operations.

It should be noted that inheritance plays different roles in specification and implementation. In specification, where external definition is relevant, if an operation is considered to provide the same functionality in all the subclasses (the classes which inherit the superclass), it should be specified in the superclass. However, its implementation may differ from one subclass to another; in such cases, a different definition must be provided in each subclass.

For example, class 'binary_file' is defined by inheriting class 'file':

```
class   binary_file has
nature
```

```
        file;

    instance

    :read(Binary_file, Buffer, File_marker):- ...;
            % Read the record at the file marker into the buffer.

    :write(Binary_file, Buffer, File_marker):- ...;
            % Write the record in the buffer at the file marker.

    :tap(Binary_file, File_tap):- !,    % Create a tap of the file.
            :create(#binary_file_tap, File_tap, Binary_file);

    end.
```

This class does not override the operations defined in class 'file', but it adds new operations, such as 'read' and 'write'. Note that externally these operations should rather be defined in class 'file'.

(2) Multiple inheritance

With multiple inheritance, we should take a different approach. First we decompose the necessary facilities into abstract classes, then build concrete classes by selecting from these. Concrete classes are those which can instantiate objects, whereas abstract classes provide a set of facilities to the concrete classes. However, it is not always easy to define an appropriate set of abstract classes. In practice, we will make use of both single and multiple inheritance.

The window system is a good example of how multiple inheritance is used. Various types of windows can be defined using multiple inheritance from the window-building classes. For example, if some windows have borders, some have labels, and others have both, then an elementary window class, a class having a label, and a class having a border are defined as window-building classes. Windows having the required features are defined by inheriting these classes.

Class 'bare_window' is an elementary class that all window classes must inherit either directly or indirectly.

```
class   bare_window has

nature
        with_display, as_inside, * ;

:create(Class, Parameters, Window) :-   % Create a new window.
        :new(Class, Window),
        :initialize(Window, Parameters),
        :start(Window) ;

instance

attribute
        width, height ;

:refresh(Window) :- !,
        :get_size(Window, Width, Height),
        :clear_rectangle(Window, 0, 0, Width, Height) ;

:initialize(Window, size(Width, Height)) :- ! ,
        Window!width := Width,
        Window!height := Height ;

end.
```

Class 'with_border' supports functions that draw a window border. The border consists of the four lines which outline the window.

```
class   with_border has

nature
        as_inside, * ;

instance

component
        border_x, border_y,
        border_area_width, border_area_height ;

attribute
        border_flag     := on,
        border_width    := 2 ;

:draw_border(Window) :- ...;        % Draw the border.
```

```
after :refresh(Window) :-
        Window!border_flag == 0, ! ;
after :refresh(Window) :- !,
        :draw_border(Window) ;

end.
```

Class 'with_label' supports functions that draw a window label. The label shows the name of the window.

```
class   with_label has

nature
        as_inside, *, as_label ;

instance

:draw_label(Window) :- ... ;        % Draw the label.

:clear_label(Window) :- ... ;       % Erase the label.

:refresh_label(Window) :-
        :clear_label(Window),
        :draw_label(Window) ;

after :refresh(Window) :-
        :refresh_label(Window) ;

end.
```

Class 'as_label', which is inherited by class 'with_label', provides concrete facilities for labeling a window.

A Window that has both a border and a label is instantiated from the following class:

```
class   window_with_border_and_label    has

nature
        bare_window, with_border, with_label;

end.
```

In the above window class, the 'refresh' predicate uses demons. Class

'bare_window' has the primary 'refresh' predicate, which refreshes the window area. Class 'with_label' and class 'with_border' have 'after' demons of the 'refresh' predicate. Refreshing this window first clears the window area, and then redraws the border and the label.

5. Conclusions

The design of SIMPOS was begun at ICOT in the fall of 1982, and the functional specification was prepared at the end of fiscal 1982. In June 1983, a software group was established for the detailed functional specification and implementation. After several modifications, the class specification was finally completed at the end of fiscal 1983.

In parallel with these activities, the requirement specifications of ESP were discussed and finalized by the summer of 1983. The language design and implementation of ESP was then started. The ESP support system is now operational on a development system. It includes an ESP cross compiler, an ESP cross linker, and an ESP simulator.

SIMPOS has been coded in ESP from the class specifications. It is now under cross-debugging on the ESP simulator. Since PSI was made available to the software group in March 1984, some programs have been ported to PSI and are also being debugged on this machine. The preliminary version of SIMPOS will be ready for internal uses at the end of September, and the first version will be completed at the end of the current fiscal year.

From our experience with SIMPOS, we feel that the object-oriented approach is an effective means of reducing the effort of both specification and implementation. However, one of the well-known drawbacks of this approach is

the overhead originated from its dynamic nature of execution. Since SIMPOS is not completed as a running system yet, it is too early to judge if our approach, eventually SIMPOS, is also acceptable in terms of efficiency. Considering that most conventional operating systems are forced to check passed parameters at run-time, we are hopeful that our approach will not significantly degrade system performance, as it does not require such checking (an object-oriented call performs this function).

Acknowledgement

We would like to thank all the members of the software group for their active involvement in the SIMPOS project, and also the File and Window Subgroups for providing the class definition examples given in this paper.

Reference

[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 to appear in 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] T.Hattori and T.Yokoi, "The Concepts and Facilities of SIMPOS File System",

ICOT TR-059 (April 1984).

[8] T.Hattori and T.Yokoi, "The Concepts and Facilities of SIMPOS Network System", to appear as ICOT TR.

[9] J.Tsuji, et al., "Dialogue Management in the Personal Sequential Inference Machine (PSI)", ICOT TR-046 (March 1984), and also to appear in the Proceedings of ACM '84 Annual Conference (Oct. 1984).

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

[11] D.Weinreb and D.Moon, "Flavors: Message Passing in the Lisp Machine", A.I.Memo No.602, MIT A.I.Lab. (November 1980).