SEQUENTIAL INFERENCE MACHINE: SIM ITS PROGRAMMING AND OPERATING SYSTEM

Toshio Yokoi, Shunichi Uchida, and ICOT Third Laboratory
ICOT Research Center
Institute for New Generation Computer Technology
Tokyo, Japan

ABSTRACT

As the first major product of Japanese FGCS (Fifth Generation Computer Systems) project, Personal Sequential Inference Machine (PSI) is under development. The whole system including the software system is called SIM. Here we describe the design of the SIM's programming system and operating system SIMPOS, its major language ESP (Extended Self-contained Prolog), the development tools, and the history of research and development.

The major research theme of SIMPOS is to develop a logic programming based programming environment including system programs.

The basic design philosophy of SIMPOS is to build a super personal computer with database features and Japanese natural language processing under a uniform framework (logic programming) based system design (Yokoi et al. 1983a,b).

1 PREFACE

As the first major product of Japanese FGCS project, SIM is under development. Here we describe the overall design of SIMPOS, its major language ESP, development tools, and the history of research and development.

The major SIMPOS research themes are to develop:

- System programs in logic programming,
- A programming environment for logic programming.

SIM is the pilot model of the FGCS software development. It is a high-performance personal machine and will be used as the research tool for the middle stage of the FGCS project.

SIMPOS has 5 basic design principles. They are:

- Uniform framework-based system design

  A single uniform PROLOG-like logic programming based framework covers all of the machine architecture, language system, operating system, and programming system.

- Personal interactive system

  We hope SIM will be one kind of personal and very highly interactive computers similar to many kinds of super personal computers.

- Database features

  PROLOG has database facilities that can easily conform to relational database systems. We hope to construct a new programming system and a new operating system that fully uses the database features.

- Window features

  In order to facilitate high level interaction, SIM uses a bitmapped display and

a pointing device.

- Japanese language processing

  All computers until now have been based on Western cultures. This is a major disadvantage for peoples of other cultures when they want to use computers. Everyone should be able to use computers in his own tongue. So, the Japanese should be able to use computers in Japanese.

SIMPOS consists of a programming system (PS) and an operating system (OS). OS consists of a **kernel**, a **supervisor**, and **I/O media subsystems**. PS consists of subsystems called **experts**. PS subsystems are controlled by users, but there is a need to coordinate the subsystems or processes. This task is accomplished by the **coordinator** subsystem.

All the other subsystems are:

Window (OS),
File (OS),
Network (OS),
Debugger/Interpreter (PS),
Editor/Transducer (PS),
Library (PS).

2 DESCRIPTION LANGUAGE: ESP

2.1 Language Overview

SIMPOS is written in a user programming language called ESP (Chikayama et al. 1983) (Chikayama 1984a). ESP is specially designed and implemented for writing SIMPOS, but is found to be useful also for writing various application programs, especially those requiring hierarchical knowledge representation.

Almost all of the features of KL0, the machine language of PSI, are directly available from ESP (Takagi et al. 1983). As based on a PROLOG-like execution mechanism of KL0, ESP naturally has many of the features of logic programming languages. The important ones among them are the use of unification in parameter passing and the AND-OR tree-search mechanism based on backtracking.

The main features of the ESP language, except for those of logic programming languages, are:

- Objects with states,
- Object classes and inheritance mechanisms, and
- Macro expansion.

See (Chikayama 1984b) for further details of the features of ESP.

2.2 Implementation

Currently, all the object-oriented features of ESP are implemented using features of KL0. Programs written in ESP are compiled into KL0. Object-oriented calls are translated into calls to a runtime subroutine, which implements the mechanism. In this way, such object-oriented calls are three to four times slower than usual predicate calls of KL0.

Implementing several built-in predicates especially designed for speeding up the execution of ESP are being planned. With such firmware supports, the execution of object-oriented calls is expected to be only slightly slower than usual KL0 predicate calls.

Another approach is also taken for speeding up the execution, i.e., introducing general source program level optimizations (Sawamura et al. 1984).

3 OPERATING SYSTEM

The SIMPOS operating system consists of three layers: kernel, supervisor, and I/O media subsystems (Hattori and Yokoi

1983) (Hattori et al. 1984a,d,e) (Takagi et al. 1984).

3.1 Kernel

The kernel manages the hardware resources to fill a gap between the PSI hardware and the supervisor (Kawakami et al. 1984). The processor management realizes multiple process environments, the memory management manages memory space and performs garbage collection, and the I/O device management controls the input/output devices.

3.2 Supervisor

The supervisor provides the basic execution facilities of object storages, process interactions, and execution environments (Hattori and Yokoi 1984c). Note that these facilities can be extended and modified as a user chooses.

A pool is a container, which is also an object, of objects of any class. A list and an array are examples of pools. An object can be put into or taken from a pool (Saito et al. 1984).

A directory is a pool of objects which are associated with a name. An object can be bound and retrieved with a name in a directory. Since a directory can contain another directory as well, a tree of directories is formed, where an object is identified with a pathname.

A stream is a pipe through which objects flow (Shimazu et al. 1984). An object which is put into one end of a stream, will be retrieved at the other end. When no object is in the stream, a retrieve operation is suspended until some object is put into the stream. A stream is used for synchronization and communication among processes.

A channel is defined on top of a stream to allow message communication among processes. A message is sent to and received from the channel. A port is a message box for two-way communication, being connected to another port with channels. A message sent through the port will arrive at the connected port to be received there.

A process executes a given program, which is an instance of a program class. The main goal of the program is defined as an instance predicate, and the slots of a program instance hold objects local to the program.

An execution environment consists of a program, a library, a world, and a universe (Watanabe et al. 1984). They can be referred to at any point of the program. A world is a sequence of directories, and each process keeps one as its working world. A universe is a system-wide directory tree.

3.3 I/O Media Subsystems

I/O media subsystems manage the interfaces with the outer worlds. This subsystem consists of three subsystems: window, file, and network.

3.3.1 Window Subsystem

The window subsystem supports high-level man-machine interface of SIM (Tsuji et al. 1984) (Iima et al. 1984). It supplies multiple logical displays (windows) on a single physical display and primitive functions on them. Other functions like echoing or cursor control are supported by other subsystems, **transducer** and **coordinator**.

In the window subsystem, windows construct a hierarchy. The most superior window is the logical screen, and usual windows are inferior windows of the logical screen. Each window may have inferior windows (sub-windows) within it. For example, an editor window has a command sub-window, a text sub-window, etc. A window is shown as a rectangular area on the physical screen, and sub-windows must be inside of its supe-

rior window. Windows can be overlapped on the screen.

Each window is dedicated to one process as its own display, and it serves as a communication channel between the process and the user at the machine. An output on a window is displayed at an appropriate position on the screen by the window subsystem. On receiving an input from the keyboard, the window subsystem decides which window, called the selected window, the input should be sent to. The mouse, a pointing device, can move anywhere on the display screen, and the window manager sends the mouse click either to the selected window or the background window, according to the mouse position. The process reads keyboard and mouse inputs through its window.

3.3.2 File Subsystem

The file subsystem provides permanent storage both for data and objects (Hattori and Yokoi 1984b) (Komatsu et al. 1984).

A permanent storage of data (records) is a file, which resides in a disk volume and consists of dispersed disk pages. Three types of files are available: binary files, table (fixed length record) files, and heap (variable length record) files. A record is identified with its stored position and/or its associated key through an index file. A binder mechanism will be supported so that a virtual file with many data and index files can be constructed. A relational database management may be built on these facilities.

A permanent storage of objects is an instance file, where each object is stored as an instance record. It is one of the major features of the file subsystem, which is not provided by ordinary file systems on other machines.

A directory file is a file which associates an instance record with a name. A permanent directory is a directory which has a directory file as its permanent storage. When included in a permanent directory, a permanent object is stored as an instance record in an instance file and included in the directory file with a pathname. Therefore, it can be restored even when the system is rebooted.

3.3.3 Network Subsystem

The network subsystem provides three types of interfaces to communicate with other machines (Takayama and Hattori 1984).

Inter-machine communication facility supports data transfer between one SIM with another SIM or other different machines. The network subsystem defines the classes **node**, **socket**, **cable**, and **plug** to implement the communication.

Inter-process communication facility allows two processes on different SIM nodes to communicate with each other, just as if they exist on the same node. A remote channel is defined to represent an original channel on the other node. A process can send a message to the remote channel and another process on the remote node can receive it from the corresponding original channel.

Remote object operation facility provides a means of dealing with objects on a remote node. A remote object on a local node represents an object on the remote node, and can be manipulated just as an ordinary object to operate on the original object. The network subsystem will support this facility to make SIMPOS to be a network operating system.

4 PROGRAMMING SYSTEM

The programming system of SIMPOS is a collection of expert processes. An expert process is a process which has an independent communication window (called **e_window**) with the user. It performs the

special action upon the user's request.

This view is different from the views such that the programming system is a collection of dumb software tools, nor is it a collection of programs to support the program production. Our view frees us from the overhead of the controlling process to manage the available tools or the information between the programs.

From the user's viewpoint, he can invoke, control, and terminate any expert through the expert's **e_window**. He need not navigate the complicated process invocation tree to accomplish his task. He need not bother about the unexpected destruction of his work through wrong navigation.

4.1 Coordinator

In SIMPOS, there is no explicit supervising process such as Shell in UNIX. However, there is a work-behind process named **Coordinator**. **Coordinator** itself is not an expert process but a process that manages the set of experts (Kurokawa and Tojo 1984).

As noted above, the user may think that he controls the expert directly through the window, but actually, **coordinator** helps the user's control via the window interface that is the associated key command table of the window.

The details of **Coordinator** are found in (Kurokawa 1984). The principal functions of **coordinator** are simply described below:

- Send a user's key command through the window to an expert,
- Create, delete, and activate an expert via system menu,
- Get and manipulate special commands from an expert, and
- Help communications between experts via the whiteboard.

The whiteboard is just like a blackboard where an expert puts a message to another expert, who in turn picks up the message by the user's instruction.

The other way to solve this communication problem is to set a communication channel with another expert. But, in this case, the channel should be set between the experts before the user decides the partner of the expert. It is not easy to tell who talks to who before communication becomes necessary.

The ultimate solution in this line would be to set a communication channel between any two experts, even though the cost becomes very high as the number of experts grows. And still, a few problems remain. The user may change the partner after he ordered the expert to put the message. It may difficult to denote both the partner and the message using only the mouse click.

Using the whiteboard, we can set virtually complete communication channels between experts. The user can select any expert after he has ordered one to put the message. This operation will be realized with one mouse click.

Each user has a directory to create experts. It contains the experts' names and the program names to create experts. The user can change the directory and the command table as he likes.

A user has his own directory which is inherited from the system's common directory, i.e., the standard set of experts.

An expert has its own set of key command table associated with its window. However, **Coordinator** permits the user to change the key command table of the window only when that window accepts the *change key command table* command from the user.

This freedom is achieved at the least

cost of execution. This minimum overhead and the maximum provision of user control is the main achievement of **Coordinator**.

4.2 Debugger/Interpreter

This subsystem interprets programs and provides information concerning the control flow of the programs. The basic facilities of the Debugger/Interpreter subsystem is similar to the debugging facility of DEC-10 PROLOG (Bowen et al. 1981). New features are:

- Procedure and clause box control flow model,
- Calls between interpretive and compiled codes, and
- Multi-window user interface.

DEC-10 PROLOG uses *Box Control Flow Model* for its debugger. It considers that each predicate is the debugging unit. In this view, each clause looks like a black-box and cannot be traced whether the unification of its head or body fails. The predicate call simply fails in both cases. However, it is often the case that the clause head is correctly selected, but the definition of the body is erroneous. When the *Procedure and Clause Box Control Flow Model* is used, it is possible to check whether unification of the head or that of the body fails (Figure 1).

In SIM, it is possible for interpretive and compiled codes to mutually call each other. However, Debugger cannot trace in the compiled code. Debugger treats the invocation of compiled codes just like a simple built-in predicate invocation. If interpretive codes are invoked from compiled codes, there is no way to pass the trace information to the interpretive codes. In such a case, Debugger restarts tracing with no trace information.

SIM has a bitmapped display screen. Debugger uses the window subsystem that

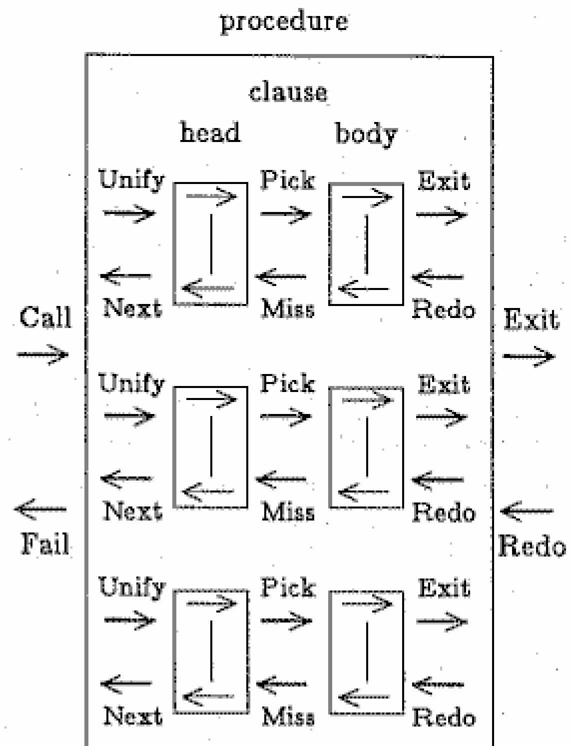

Figure 1. Procedure and Clause Box Control Flow Model for interpretive code

offers a multi-window user interface with the mouse. A user can select one of the control options at break points, look at ancestors or spy points, check the values of slots, or see the class definitions using the library subsystem. This information is shown in subwindows of Debugger and all the selections can be done using the mouse click.

4.3 Editor and Transducer

An editor is a typical component of a programming system and an indispensable software tool in using a computer system. Though there can be editors to manipulate abstract structures completely different from texts, here we limit our discussion to the editors which edit texts or data expressed in texts.

Even text expressions usually have nested structures and the editor for SIM

(called Edips) is designed to manipulate structured texts generally. But we do not believe that there can be a general purpose editor which is convenient for every structure. A good general editor is one that is convenient for a specific purpose and can be used for general purposes even if less powerful. Under this criterion, Edips is designed to be especially convenient for editing ESP programs and can manipulate other structures. In addition, Edips has the following features:

- Customization with macro definition
- A small number of commands easy to memorize
- Failsoft with many recovery environments

To make Edips general, we allow users to define the syntax. Though other general structure-editors usually use BNF, we do not adopt it because usual editing operations are neither to trim a branch of the syntax tree nor to traverse the tree. Editing operations are more closely related to the text expression of edited data. So we adopted an operator precedence grammar with user definable parentheses. An operator precedence grammar is more simple and has better correspondence to the text expression.

Every token in the text expression of edited data is classified into six categories:

- Atom
- Prefix operator
- Infix operator
- Postfix operator
- Left parenthesis
- Right parenthesis

Each operator has a precedence. For editing purpose, however, too many precedence levels should not be adopted, because precedence introduces structures without direct correspondence to the text structure. As for an ESP editor, two or three levels are necessary and sufficient. They are for:

- logical symbols such as ":-", ",", ";"
- function symbols such as "+", "—", "*", "/".

If necessary,

- predicate symbols such as "<", ">", "="

will be added.

In addition to the operator precedence grammar, we adopt the usual regular expressions for defining the tokens. The text, which is a sequence of characters, is first transformed to a sequence of tokens by automata and then parsed to a structure. Thus the grammar is two-leveled. However, since the both levels are very simple, it is easy to treat the grammar, but it has enough expressive power to define the syntax of almost all the structured programming languages.

It is desirable that the parser and the pretty printer for the grammar can be used by other programming tools such as compiler, interpreter and debugger. Therefore, those tools are made as utilities (named the transducer) separate from the editor. Thus Edips consists of the editor kernel and the transducer.

4.4 Library

The library subsystem manages all the classes and predicates on SIM. It controls the registration of classes, loading program files, compiling, and building class objects by the analysis of inheritance.

Each class has a class source file, a class template file, and a class object file on some secondary storage. Class templates and class objects exist only in the main storage, but are saved to and restored from the secondary storage.

Class source files are text files coded

by the users. A class source file can have just one class definition. Like source files, template files and object files also have just one class information in each.

A class template is built from a single source file. It holds all the information of that class except those from inheritance analysis. The predicates of that class are kept as interpretive codes when the template is built. They are compiled when the user requests. After the compilation, both interpretive and compiled codes are kept. Templates can be saved or restored before compiling the predicates.

Class objects are built from some class templates. In a class object, all the inheritances are analyzed and solved. It is an executable image of an object oriented program.

Another feature of the library subsystem is to manage predicates. It contains the features of referring to one predicate of a class, i.e., object oriented invocation, and the invocation from compiled codes to interpretive codes or the converse. This mechanism is implemented by indirect references. All the invocation of predicates are done via indirect references. When some interpretive codes are invoked, that indirect word points the entry of the interpreter. This mechanism causes a uniform invocation scheme even if both the interpretive and compiled codes are mixed.

For object oriented invocation, it is necessary to find which method should be invoked during the execution time. Here, the library has to distinguish those predicates that have the same predicate name but are defined in different classes. In the compiled codes, all the references are processed and changed to the direct invocation of the specific predicate, but in the interpretive codes, the library has to search the predicates during the execution time.

The compiler is simply a subroutine of the library subsystem. It compiles a single predicate from interpretive codes. This process is done only in main storage. After the compilation, library holds both interpretive and compiled codes. The user can specify which code should be used for building up a new class object. The template file is automatically rebuilt after the compilation.

4.5 Exception Handling and Help System

Generally speaking, the exception is one of the important concept in the software system. For example, the followings are included in the exception: the various errors such as hardware-detected zero division, device-detected I/O errors, software-detected errors, and the global exits in the complicated nested procedure invocations, and the various help facilities provided for the user at his work.

In the traditional system, those exceptions are not well treated. They are handled at each subsystem and/or system level separately and independently.

In SIMPOS, the exception handling system is provided throughout the all components and all the system levels. Our principal target and the basic frame are as follows:

- Uniform framework

  An exception is generated and signalled by a part of the software/hardware. This action-taking part is called *detector*. The signalled exception must be handled by some process with a exception handling program. This processing part is called *handler*.

- No limitation for exception registrations

  In the evolving system such as SIMPOS, it cannot be forecasted how many kinds of exceptions are used. Instead, it

should be provided the way to register any number of exceptions with as fine classification as possible. The multiple inheritance mechanism should support the above target.

- Flexible exception handling

  The same exception has the various meanings depending on the environment when and how it occurs. And each occurrence must be handled differently according to the meaning.

  The *detector-handler* scheme is powerful in this sense of the context mechanism. *Detector* need not concern about the environment. And *handler* also need not concern about the context, only if the choice of the handler is performed depending on the context.

  The implementation is done as below.

There are two basic classes: *event* and *situation*. *Event* is the name of the exception in general. There are two basic subclasses in event: *error* and *help*. *Error* has several subclasses such as warning, fatal error, and normal error. *Help* has also several subclasses such as help in general, and keyboard help and so on. *Situation* is the name of the context for the exception handling. The handler for the possible (or dangerous) exceptions are set in *situation*. *Situation* has a stack like structure where each program can set the necessary handlers for its process and delete the useless handlers when the process normally ends.

One of the problem in this scheme is that there is a case when the several handlers can be invoked for a single event. The situation has a mechanism to sort the handlers and apply them just in the mode of non-determinism.

The help facilities are investigated in the broadest sense. Not only the conventional on-line manual lookup facility but also the completion mechanism for keyboard or the limited spell correction are included.

The global exit such as catch-throw or errset in Lisp language is also realized with the use of this *event-situation* mechanism. Also a command-loop which is common in the application programs is provided.

5 SOFTWARE DEVELOPMENT TOOLS

5.1 ESP Cross System

All of SIMPOS is written in ESP. Since they were designed and coded before the hardware had become available, we needed a cross system of ESP for software development.

Most of the programs are written in PROLOG and some are in PASCAL on a main-frame machine. They are:

- ESP interpreter,
- ESP cross compiler (to KL0),
- KL0 cross compiler,
- KL0 cross linkage editor, and
- Miscellaneous utility programs for inspecting database of the linkage editor.

5.2 Runtime Support System

The object-oriented calling mechanism of ESP is realized by translating them into calls to runtime subroutines by the ESP compiler. The runtime support system of ESP is a set of such runtime subroutines. The runtime support system is written directly in KL0 and provides the following features:

- Basic object-oriented calling mechanism of ESP.
- Mnemonimnemonic tracing of object-oriented calls. It is based on the procedure box control flow model and various interactive control (skip, redo, etc.) are possible.

- Inspection of object slot values in mnemonic form.

All the input and output required for the above features are implemented using the built-in "*read_console*" and "*display_console*" predicates which utilize the I/O devices of the console processor. The console processor is provided primarily for hardware and firmware debugging and was ready with the PSI hardware. Thus, mnemonic debugging of ESP programs was made possible almost directly after the firmware of PSI had been completed.

When the firmware execution support for ESP gets ready, it will be responsible for the basic calling mechanism. The runtime support will be a package for treating trace exceptions generated by the firmware.

5.3 IPL

When debugging of SIMPOS first began, programs to be debugged must be compiled and linked completely on the main-frame machine, transferred to a mini-computer through a network, and then loaded down to PSI. The network transfer and down-loading took almost unbearably long time.

When several parts of the kernel and the supervisor of SIMPOS got ready, a rather flexible initial program loader was implemented in ESP. Using this IPL, modules of programs to be debugged are separately compiled, partially linked and stored in flexible disks (Ueda et al. 1984). Loading from the flexible disk and the final linkage are done by the IPL on PSI itself. This sped up debugging considerably.

5.4 System Tracer

The system tracer is a program also written in ESP. It runs as a separate process and traces the execution of other processes in the mnemonic form. The system tracer was implemented for tracing KL0 level execution while the runtime support system can only trace ESP-level execution (Sato et al. 1984).

Currently, an effort to unify these two debugging tools, the runtime support system and the system tracer, is in progress.

6 BRIEF HISTORY

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 of about 20 members, excluding the ICOT members, 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 and cross-debugged on the ESP simulator since October 1983.

The first PSI was produced in December 1983, and firmware debugging was up at the end of February 1984. PSI 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. The major parts of the I/O media systems were

operational in September. The programming system is now under debugging on PSI.

The preliminary version of SIMPOS will be ready for internal uses in October, and the first version of SIMPOS will be completed at the end of the current fiscal year.

7 CONCLUSION

About 40 guys from ICOT, Mitsubishi Electric Co., Ltd., NEC Corp., Oki Electric Industry Co., Ltd., Matsushita Electric Industrial Co., Ltd., and Sharp Co., Ltd. are engaged in the development of SIMPOS. Their effort has made clear the powerfulness and generality of logic programming. The current status of SIMPOS will be shown in the demonstration at the conference hall of FGCS'84 and ICOT.

Improvements and enhancements of SIMPOS will be continued in parallel with other research activities and SIM will grow into a main component in the infra-structure of this project.

REFERENCES

Bowen, D. L., Byrd, L., Pereira, F. C. N., Pereira, L. M., Warren, D. H. D. DECsystem-10 PROLOG User's Manual. Dept. AI., Univ. of Edinburgh, p. 101, 1983.

Chikayama, T., Takagi, S., Sakai, K. Personal Sequential Inference Machine PSI – Its Language System –. Proceedings of The 27th annual conference of Information Processing Society of Japan, 1983 Also in ICOT Technical Memorandum TM-0022, 1983.

Chikayama, T. ESP Reference Manual. ICOT Technical Report TR-044, 1984a.

Chikayama, T. Unique Features of ESP. *International Conference on Fifth Generation Computer Systems* 1984 Also in ICOT Technical Memorandum TM-0055, 1984b

Hattori, T., Yokoi, T. Basic Constructs of the SIM Operating System. New Generation Computing, vol. 1 no. 1, pp.81-85, 1983 Also in ICOT Technical Memorandum TM-018, 1983.

Hattori, T., Tsuji, J., Yokoi, T. SIMPOS: An Operating System for a Personal Prolog Machine PSI. ICOT Technical Report TR-055, 1984a.

Hattori, T., Yokoi, T. The Concepts and Facilities of SIMPOS File Subsystem. ICOT Technical Report TR-059, 1984b.

Hattori, T., Yokoi, T. The Concepts and Facilities of SIMPOS Supervisor. ICOT Technical Report TR-056, 1984c.

Hattori, T., Kurokawa, T., Sakai, K., Tsuji, J., Chikayama, T., Takagi, S., Yokoi, T. An Operating System for Sequential Inference Machine PSI. ICOT Technical Memorandum TM-0065, 1984d and ICOT Technical Memorandum TM-0061, 1984 (in Japanese).

Hattori, T., Tsuji, J., Uchida, S., Yokoi, T. Overview of SIMPOS Operating System. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-1, 1984e (In Japanese).

Iima, Y., Nakazawa, O., Enomoto, S., Tsuji, J. Window Subsystem of SIMPOS. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-6, 1984 (In Japanese).

Kawakami, T., Ueda, N., Horie, M., Hattori, T. Resource Management of SIMPOS. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-2, 1984 (In Japanese).

Komatsu, M., Mano, T., Konagaya, A., Hattori, T. File Subsystem of SIMPOS. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-8, 1984 (In Japanese).

Kurokawa, T., Tojo, S. Coordinator - a kernel of Personal Sequential Inference Machine (PSI). ICOT Technical Report TR-061, 1984.

Saito, S., Watanabe, H., Shimazu, H., Yoshida, N., Hattori, T. Execution Management of SIMPOS – Pool –. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-4, 1984 (In Japanese).

Sato, Y., Watanabe, H., Hori, A., Ueda, N., Chikayama, T. System Tracer of SIMPOS. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-10, 1984 (In Japanese).

Sawamura, H., Takeshima, S., Kato, A. PROLOG Source-Level Optimizer: A Catalogue of Optimization Methods. ICOT Technical Report TR-047, 1984 (in Japanese).

Shimazu, H., Yoshida, N., Saito, S., Watanabe, H., Hattori, T. Execution Management of SIMPOS – Process and Stream –. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-3, 1984 (In Japanese).

Takagi, S., Chikayama, T., Hattori, T., Tsuji, J., Yokoi, T., Uchida, S., Kurokawa, T., Sakai, K. Overall Design of SIMPOS. Proceedings of Second International Logic Programming Conference, 1984 Also in ICOT Technical Report TR-057, 1984.

Takagi, S., Chikayama, T., Yokota, M., Hattori, T. Introduction of Extended Control Structures for Prolog. Proceedings of The 26th annual conference of Information Processing Society of Japan, 4D-11, 1983 (In Japanese).

Takayama, Y., Hattori, T. Network Subsystem of SIMPOS. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-7, 1984 (In Japanese).

Tsuji, J., Kurokawa, T., Tojo, S., Iima, Y., Nakazawa, O., Enomoto, S. Dialogue Management in the Personal Sequential Inference Machine (PSI). ICOT Technical Report TR-046, 1984.

Ueda, N., Tojo, S., Kurokawa, T. IPL Scheme of SIMPOS. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-9, 1984 (In Japanese).

Watanabe, H., Shimazu, H., Yoshida, N., Saito, S., Hattori, T. Execution Management of SIMPOS – World –. Proceedings of The 29th annual conference of Information Processing Society of Japan, 4E-5, 1984 (In Japanese).

Yokoi, T., Taguchi, A., Kurokawa, T., Hattori, T., Tsuji, J., Sakai, K. Structures and Design Principles of the Operating System for the Personal Sequential Inference Machine (SIM). Proceedings of The 26th annual conference of Information Processing Society of Japan, 6D-8, 1983a (In Japanese).

Yokoi, T., Taguchi, A., Kurokawa, T., Hattori, T., Tsuji, J., Sakai, K. Structures of an Operating System on a Logic Programming Language. Proceedings of The 26th annual conference of Information Processing Society of Japan, 6D-7, 1983b (In Japanese).