ICOT Technical Report: TR-057

TR-057

OVERALL DESIGN OF SIMPOS
(Sequential Inference Machine
Programming and Operating System)

by

Shigeyuki Takagi, Toshio Yokoi,
Shunichi Uchida, Toshiaki Kurokawa,
Takashi Hattori, Takashi Chikayama,
Kō Sakai and Junichiro Tsuji

April , 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

OVERALL DESIGN OF SIMPOS

(Sequential Inference Machine Programming and Operating System)

Shigeyuki Takagi, Toshio Yokoi, Shunichi Uchida, Toshiaki Kurokawa
Takashi Hattori, Takashi Chikayama, Kō Sakai, Junichiro Tsuji

ICOT

(Institute for New Generation Computer Technology)

Mita Kokusai Building, 21F.

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

ABSTRACT

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

The major research theme of ψ 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.

At the end of March 1985, we will be able to show that the logic programming based operating/programming system is working well and has a good human interface.

1 Preface

As the first major product of Japanese FGCS project, ψ is under development. Here we describe the overall design of ψ's Programming System and Operating System called SIMPOS, its major language ESP, and some development tools.

The major ψ research themes are to develop:

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

ψ 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 FGCS project.

The hardware and firmware design of ψ was completed at ICOT, and the first pilot model has already been manufactured. Its firmware debugging has been finished in March 1984. Installation of SIMPOS was started in February.

SIMPOS has 5 basic design principles. They are:

- o 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.

- o Personal interactive system

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

- o 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.

- o Window features

  In order to facilitate high level interaction, ψ 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 ESP

2.1 Overview

SIMPOS is described in a user programming language called ESP. Programs written in ESP are compiled into KLO. KLO is the machine language of ψ and is a PROLOG-like logic based language with several extensions.

As based on a PROLOG-like execution mechanism, ESP naturally has many of the features available in PROLOG. The important ones among them are the use of unification in parameter passing and a tree-search mechanism based on backtracking.

The main features of the ESP language, except for those in common with PROLOG-like languages, are:

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

The assertion and atom name database features (assert, name, etc.) are not directly available, though lower level features (array access, string manipulation, etc.) for implementing them are provided.

2.2 Objects and Classes

The control structure of ESP is basically that of PROLOG: AND-OR tree search by backtracking. However, from another point of view, an ESP program is constructed in an object oriented manner.

An object in ESP represents an axiom set, which is basically the same concept as *worlds* in some PROLOG systems (M. Van Caneghem 1982). The same predicate call may have different semantics when applied in different axiom sets. The axiom set to be used in a call is specified by passing an object as the first argument of a call and prefixing the call with a colon (:).

An object may have time dependent state variables called **object slots**. Values of slots can be examined by certain predicates using their names. In other words, the slot values define a part of the axiom set. The slot values can also be changed by certain predicate calls. This corresponds to altering the axiom set represented by the object. This is similar to **assert** and **retract** of DEC-10 PROLOG, but the way of alteration is limited.

It seemed to be difficult to us, if not impossible, to describe an entire operating system in pure logic without any built-in notion of time dependency. As many of the currently available ideas for the building blocks of an operating system are based on the notion of state, much more investigation is required before starting to write an entire operating system in pure logic (this approach is being tried by Shapiro (E. Shapiro 1984)). This is why object oriented features with side effects are introduced into ESP.

An ESP program consists of one or more class definitions. An object class, or simply a class, defines the characteristics common in a group of *similar* objects, i.e., objects

which differ only in their slot values (only values; slot names are common to the objects belonging to the same class). An object belonging to a class is said to be an instance of that class. A class itself is an object which represents a certain axiom set.

2.3 Inheritance Mechanism

A multiple inheritance mechanism similar to that of the Flavor system (D. Weinreb and D. Moon 1981), rather than the single inheritance seen in Smalltalk-80 (A. Goldberg and D. Robson 1983), is provided in ESP. A class definition can have a nature definition, which defines one or more **super classes**. When a class is a **super class** of another class, all the axioms in the axiom set of the former class are also introduced into the axiom set of the latter class, as well as the original axioms given in the definition of the latter class. By this inheritance mechanism, classes form a network of *is-a* hierarchy.

Some of the super classes and the subclass which inherits them may have axioms for the same predicate name. Since basically the axiom sets of the super classes are simply merged, such axioms are ORed together. Though the order in the ORed axioms has no significance as long as pure logic is concerned, it can be specified in ESP for hand optimization and to control **cuts** and side effects.

Clauses called **demon clauses** define **demon predicates**, which are ANDed, rather than ORed, either before or after, as specified, the disjunction of usual axioms. They are used to add non-monotonic axioms. For example, a door *with a lock* has a **demon** for the predicate **open** for making sure it is already unlocked. In this way, a class **with_a_lock** can be defined separately from the class **door** as a class that contains non-monotonic knowledge.

Part-of hierarchy can also be implemented using the *is-a* hierarchy and object slots. Assume that we want to make instances of class **A** to be a *part of* an instance of class **B**. First, the definition of **A** should be given. Then, a class **with_A** should be defined so that instances of the class **with_A** has a slot which holds an instance of class **with_A**. Finally, class **B** is defined to be a subclass of **with_A**; in other words, the class **B** *is-a* class **with_A**.

2.4 Macros

Macros are for writing meta programs which specify that programs with so and so structures should be translated into such and such programs. Macros can be defined in the form of an ESP program, fully utilising the pattern matching and logical inference capability of the logic programming language.

In various languages with macro expansion capability, a macro invocation is simply replaced by its expanded form. Though this simple macro expansion mechanism may be powerful enough for LISP-like functional languages, it is never enough for a PROLOG-like logic based language. For example, a macro which expands

$$p(a, f(X+Y))$$

to a sequence

$$add(X, Y, Z), \quad p(a, f(Z))$$

cannot be defined with a simple expansion mechanism.

Macros of ESP are not only expanded at the place of the macro invocation. Certain additional goals can be spliced in before or after the goal in which the macro invocation is given. If the macro is invoked in the head, these goals will be added at the top or the end of the body.

The same macro definition:

$$X + Y => Z \quad \textbf{when} \quad add(X, Y, Z)$$

can be used in two ways. The clause "$add1(M, M+1)$." is expanded into the clause "$add1(M, N)$:-$add(M, 1, N)$.", while the body goal "$p(M+1)$" is expanded into a goal sequence "$add(M, 1, N), p(M)$".

2.5 Implementation

Currently (in March 1984), a cross compiler of ESP into KLO is available.

The implementation of the object oriented calling mechanism is rather straightforward:

each object has a slot containing a database of codes corresponding to the axiom set associated with the object.

The current implementation uses slot name atoms for accessing object slots. Such access has been found to be very fast thanks to the built-in hashing mechanism of KLO. Certain other firmware supports for accelerating the execution are also planned.

3 Operating System

The operating system part of SIMPOS consists of 3 layers; kernel, supervisor, and I/O media subsystems.

3.1 Kernel

The kernel manages the hardware resources and fills a gap between the ψ hardware and the supervisor. It includes the processor manager which realizes multiple process environments, the memory manager which allocates and deallocates memory space and performs garbage collection, and the I/O device manager which controls the input/output devices.

3.2 Supervisor

The supervisor provides the basic facilities useful for program execution, such as object storages, inter process interactions, and execution environments. For details, refer to (Yokoi and Hattori 1983). Note that a user may extend and modify these facilities as he 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.

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. 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 between processes.

A channel is defined on the top of a stream to allow message communication between two processes. A port is a message box for two-way communication, connected to other ports. A message sent through the port will arrive at these connected ports, and a message sent from one of these ports will arrive at this port.

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.

A process has several environments: a program, a library, a world, and a universe. They can be referred to at any point of the program. A world is a sequence of directories held by a process as its working world. A universe is a system directory tree held in a class slot of class **directory**.

3.3 I/O Media Subsystems

I/O media subsystems manage the interfaces with the outer worlds. This subsystem consists of 3 subsystems: window, file, and network.

3.3.1 Window Subsystem

The window subsystem is the main part of high level man-machine interface of ψ (Kurokawa et al. 1984). It supplies multiple logical displays for processes in ψ on a single physical display. The Lisp Machine developed at MIT also supplies such an environment. The Lisp Machine window subsystem manages the major part of the man-machine interface. But our window subsystem manages only the primitive functions. Other functions like echoing are done by other subsystems, transducer, coordinator, etc. This concept increases the modularity of the whole system, and make each subsystem simpler.

For each process, one window is dedi-

cated for its own display and it need not mind other windows. In the window subsystem, each window is defined as an instance of the window class and each predicate for the window is written as methods of the class. So the window manager need not consider the interaction among the windows and each process can consider its window as its own display. Each window is a rectangular area which is a part of the physical screen, and is the communication channel to the process.

In the window subsystem, windows construct a hierarchy. The most superior window is the logical screen, and normal windows (editor window, etc.) are inferior windows of the logical screen. Each window may have inferior windows (called subwindows) within it, and each inferior window can have its own inferior windows. For example, an editor window has command sub-window, text sub-window, etc. Subwindows can neither have a size that exceeds their superior window's size, nor go out from the superior window. They must be inside of the superior window.

Each window may have one of the following 5 states:

selected: Connected to the keyboard. Only one window can have this status at a time.

shown: Completely displayed on the physical screen, and the mouse button click in this window is interpreted using the key-command definition of this window.

exposed: Completely displayed on its superior window. However, when the superior window does not have the shown status, even if the window is completely displayed on the screen, it does not have shown status, but exposed status.

overlapped: Partly or completely hidden by its superior window. This window is hidden by another inferior window of its superior window.

deactivated: Not managed by the window subsystem. Windows in this status will never be shown on the physical screen until activated. However, its memory image is not destroyed.

These states are managed by the window manager. The I/O function is determined by these states. The relation between the window states and the I/O functions are shown in table 3-1.

Table 3–1. Window Status

Status	key–in	mouse	output
Selected	done	done	done
Shown	wait	done	done
Exposed	wait	wait	wait
Deexposed	wait	wait	wait
Deactivated	fail	fail	fail

Whenever there is a keyboard input, the window subsystem has to decide which window the input should be sent to. The window manager has the instance slot **selected_window** which keeps the **selected** window. As another input device, ψ has a pointing device **mouse**. The mouse can move anywhere on the display screen, and the window manager can recognize the window, which the mouse click is sent to, by the position of the mouse. The mouse click is treated by the window's definition in only the **shown** window. It is because if the mouse click changes the window's output image, the user may not see it since he cannot see the whole of the not **shown** window, and the window manager cannot recognize which part of the window is hidden.

3.3.2 File Subsystem

The file subsystem provides permanent storage both for data and objects.

A permanent storage of data (records) is a file. 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 user can define a virtual file with many data and index files. A relational database management may be built on these facilities.

A permanent storage of objects is an instance file. It is the main feature of the file subsystem not provided by other machines' ordinary file systems.

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 after the system is rebooted.

3.3.3 Network Subsystem

The network subsystem provides three types of interfaces to communicate with other machines.

Inter-machine communication is supported to connect one ψ with another ψ or other different machines. The network subsystem defines the classes **node**, **socket**, **cable**, and **plug** to implement the communication.

Inter-process communication allows two processes on different ψ 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.

The introduction of remote objects is a main feature of the network subsystem. A remote object represents an object in a remote node. It can be manipulated just as an ordinary object.

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.

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 principal functions of **coordinator** are as follows:

- o Send a user's key command through the window to an expert,
- o Create, delete, and activate an expert via system_menu,
- o Get and process special commands from an expert, and
- o 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 theme 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 (D. L. Bowen et al. 1981). New features are:

- o Procedure and clause box control flow model,
- o Calls between interpretive and compiled codes, and
- o 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 (see fig. 4-1).

In ψ, 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,

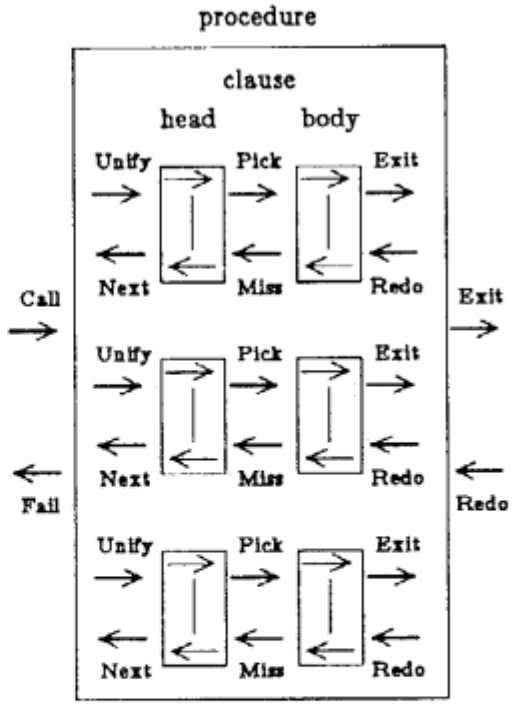

Fig. 4–1. Procedure and Clause Box Control Flow Model for interpretive code

there is no way to pass the trace information to the interpretive codes. In such a case, Debugger restarts tracing with no trace information.

ψ has a bitmapped display screen. Debugger uses the window subsystem that 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 sub-windows of Debugger and all the selections can be done using the mouse click.

4.3 Editor

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. So the editor for ψ (called Edips) is designed to be a general structure-editor. 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:

- o Customization with macro definition,
- o A small number of commands easy to memorize, and
- o 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 6 categories:

- o Atom,
- o Prefix operator,
- o Infix operator,
- o Postfix operator,
- o Left parenthesis, and
- o 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, 2 or 3 levels are necessary and sufficient. They are for:

- o logical symbols such as

  ":-", ";", ","

- o function symbols such as

  "+", "−", "*", "/".

If necessary,

- o predicate symbols such as

  "<", ">", "="

will be added.

As explained above, the operator precedence grammar is very simple, but has enough expressive power to define the syntax of almost all 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. So, those tools are made as separate utilities from the editor. Edips consists of the editor kernel and those utilities which are also used by other tools.

4.4 Library

The library subsystem manages all the classes and predicates on ψ. 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.

5 Development Tools

Almost all of the OS/PS programs are written in ESP. Since they were designed and coded *before* the ψ machine becomes available, we need a cross system of ESP for software development.

Most of the programs are written in PROLOG. The programs are:

- o ESP cross simulator,
- o KLO cross compiler,
- o KLO cross assembler,
- o ψ microprogram cross assembler,
- o Cross linkage editor for both KLO and microprogram,
- o Table generator for the microprogram.

Some programs, the execution speed of which is critical for debugging (microprogram simulator, etc.) are developed in PASCAL.

One of the powerful support tools is **Customisable Assembler** (S. Takagi 1983). It is the kernel of both the KLO assembler and the ψ microprogram assembler. Only the machine-dependent parts such as the length of the object word, field definitions, mnemonic definitions, and checking conditions are changed. Machine-dependent

parts are pre-processed and are compiled with the assembler kernel into a machine-dependent assembler.

The definition of KLO is about 500 lines while the definition of the ψ microprogram is approximately 1100 lines. About 80% of them are conversion tables from mnemonics to field values. The kernel part is about 900 lines of PROLOG program. Compared with many so-called *generalized assemblers* or *universal assemblers*, this assembler has only 1/5 to 1/10 as many codes. Its assembly speed is, however, approximately comparable.

Using PROLOG's unification and backtracking mechanism, it is possible to write a sophisticated error-checking routine. If one field overlaps another, the unification fails and the next alternative value setting is tried. Setup conditions are processed in the same way. If an assembler variable X is unified to the value **case_1** while one field is processed, the process for any other field cannot unify **case_2** for X. So, the unification fails and the process backtracks. Finally, when all of the unification is successfully completed, the object bit-pattern is generated and written out to the object file.

6 Conclusion

A logic programming based inference machine (ψ) and its Programming/Operating System (SIMPOS) is now under development. The first pilot hardware has already been manufactured and firmware debugging was finished. Installation of SIMPOS was started in February.

The first release of ψ and SIMPOS for FGCS research and development will be at the end of March 1985. We will continue its improvements and enhancements. At that time we will be able to show that the logic programming based Operating/Programming system is working well and has a good human interface.

Many investigations and researches are necessary for building logic programming based programming and operating systems. We hope this work will contribute to such researches.

ACKNOWLEDGMENTS

The authors thank Mr. G. Hagio and Mr. H. Ishibashi for their contribution to our project.

APPENDIX I

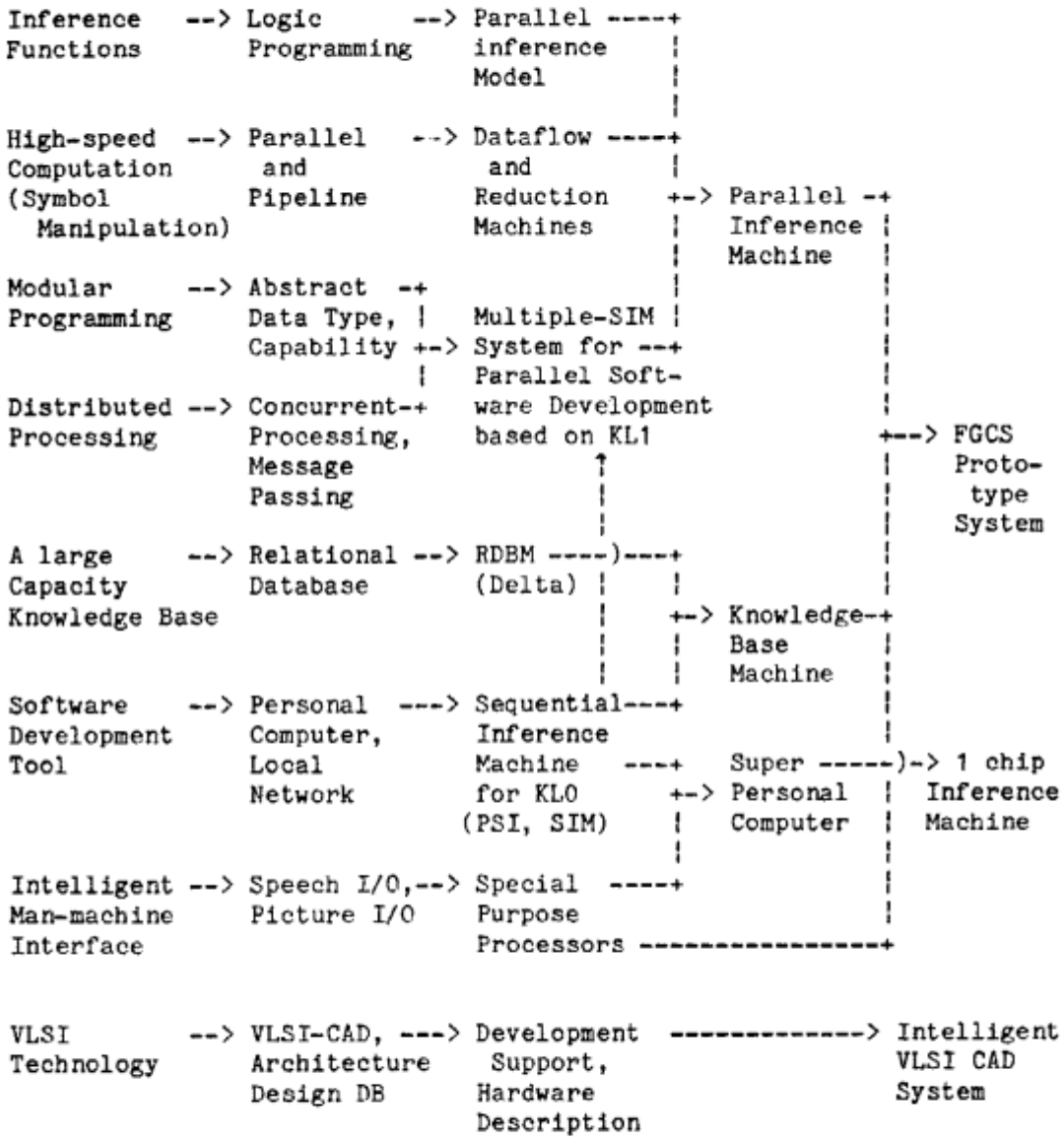

Fig. I-1 An Approach to the Fifth Generation Computer

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. ESP – Extended Self-contained Prolog – as a Preliminary Kernel Language of Fifth Generation Computers. New Generation Computing 1, No. 1, 11–24, 1983.

Chikayama, T., Yokota, M., Hattori, T. Fifth Generation Kernel Language: Version-0. Proceedings of the logic programming conference '83, 7.1 1–10, 1983.

Fuchi, K. The Direction of the FGCS Project will Take. New Generation Computing 1, No. 1, 3–9, 1983.

Goldberg, A., Robson, D. SMALLTALK-80 — The Language and its Implementation. Xerox Palo Alto Research Center, p. 714, 1983.

Hattori, T., Yokoi, T. Basic Constructs of the SIM Operating System. New Generation Computing 1, No. 1, 81–85, 1983.

ICOT Report of the FGCS Project's Research Activities 1982. ICOT Journal 1, No. 2 , 1983.

Krasner, G. SMALLTALK-80 — Bits of History, Words of Advice. Xerox Palo Alto Research Center, p. 344, 1983.

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

Nishikawa, H., Yokota, M., Yamamoto, A., Taki, K., Uchida, S. Design Philosophy and Architecture of the Sequential Inference Machine PSI (In Japanese). Proceedings of the logic programming conference '83, 7.2 1–12, 1983.

Shapiro, E. Systems Programming in Concurrent Prolog. Eleventh Annual Symposium on Principles of Programming Languages (to appear).

Takagi, S. Customizable microprogram assembler. ICOT Technical Report TR-021 (In Japanese), p. 25, 1983.

Uchida, S., Yokota, M., Yamamoto, A., Taki, K., Nishikawa, H. Outline of the Personal Sequential Inference Machine PSI. New Generation Computing 1 No. 1, 75–79, 1983.

Van Caneghem, M. PROLOG II Manuel D'Utilisation, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, 1982.

Weinreb, D., Moon, D. Lisp Machine Manual, 4th ed., Symbolics, Inc. 1981.

Yokoi, T., Hattori, T. The concepts and facilities of the SIMPOS supervisor (to appear as an ICOT Technical Report).