ICOT Technical Report: TR-056

TR-056

The Concepts and Facilities of SIMPOS Supervisor

by

Takashi Hattori and Toshio Yokoi

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

The Concepts and Facilities of SIMPOS Supervisor

Takashi HATTORI and Toshio YOKOI

Institute for New Generation Computer Technology

Mita Kokusai Building 21F

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

Abstract

This report describes the design approach and some implementation aspects of the supervisor part of SIMPOS (Sequential Inference Machine Programming and Operating System) for PSI, a personal Prolog machine which is now under development at ICOT. The supervisor is a layer in SIMPOS which takes care of software resource management, execution control, and other tasks.

Table of Contents

1. Introduction

SIMPOS is an operating and programming system on PSI, a personal Prolog machine [1], which is now under development at ICOT. When we began the design of SIMPOS, we had in mind that both the machine and the software would be prototypical and that the development schedule would be tight. Hence we have chosen simplicity, extendability, and ease of change, as the design objectives of SIMPOS, and we pursued them from the standpoints of programming methodology and system structure.

(1) Object-oriented approach

We have adopted an object-oriented programming approach to build SIMPOS. We believe that this approach provides both a modular programming framework and side effects on data, which we need to write an operating system in a single language. The system description language for SIMPOS, called ESP [3], is a Prolog with class mechanism. All the facilities of SIMPOS are provided as objects in ESP.

A class definition in ESP gives a specification on both a class object and its instances with class/instance predicate definitions, class/instance slot declarations, and local predicate definitions. An instance predicate defines an operation on an object as

```
:operation(Object,...) :- ...
```

where 'Object' is of this class. A class predicate defines an operation on a class object as

```
:operation(Class,...) :- ...
```

where 'Class' is either a class object or a class name of #<class_name>. One of the class predicates often used is the one to create (instantiate) an object:

```
:new(#<class_name>,Object)
```

A local predicate is an ordinary Prolog predicate defined as

```
predicate(...) :- ...
```

and it can be called only within the class definition. A slot declaration gives a class/instance a slot for its component object. A slot is refered to by

```
Object!<attribute_name>
```

Note that all the interfaces which are visible from the outside of an object are the class/instance predicates, so as to hide the implementation details.

(2) Overall structure

We have constructed SIMPOS in several layers, between the hardware layer and the user (application) program layer. Each layer provides some facilities to the higher layers by using some of the facilities supported in the lower layers. This layered approach reduces the complexity of the entire system, though sometimes it may increase interface overhead.

SIMPOS is largely divided into two parts -- an operating system part which provides an execution environment of a program, and a programming system part which gives an programming environment for program development. The operating system part has three layers -- a kernel for hardware resource management, a

supervisor for software resource management, and i/o medium systems for input/output management. The overview of these layers is explained in [4]. In this report, we will describe the supervisor in detail.

2. Supervisor

The supervisor is a layer of the operating system, which manages software resources, controls program execution, and provides utility facilities. Execution model are important for deciding what kinds of features should be provided in a supervisor layer. Our model is that a system consists of objects and processes, so that each process performs operations on objects, and if necessary, communicates with any other cooperating processes, according to a given program under a specified environment which keeps global objects.

The supervisor, just as the other layers of SIMPOS do, implements its facilities as operations on objects. What the supervisor really provides is a collection of class definitions. We can view these supervisor-defined objects (or classes) as the basic constructs of the operating system. Some of them are used internally in the supervisor, some are used both by the supervisor itself and user programs, and others are intended for use by user programs as utilities. Users can easily customize the supervisor by adding new classes and modifying existing classes.

Now we divide the supervisor into four subparts -- object storage management which supports collections of objects, object flow management which supports inter-process communication, execution management which controls processes executing programs, and environment management which manages process execution environments. We will describe these facilities in the following sections.

Note that since PSI does not support virtual memory, some software supports are necessary to deal with data in file storage.

3. Object Storage -- Pools

An object storage, which we call a pool, can keep objects in it. A pool is useful for dealing with a collection of objects as a whole. Since any object can be defined as having other objects as its components with accessing operations for them, a pool is simply a general object which is provided only for storing and retrieving objects.

(1) As pools

Class 'as_pool', which is a mixin class to be inherited by any pools, defines the general operations on a pool, such as

- To add an object into a pool

  :add(Pool,Object)

- To remove an object from a pool

  :remove(Pool,Object)

- To retrieve an object from a pool

  :get(Pool,Object)

A pool is illustrated in the figure below, where an 'O' represents an object.

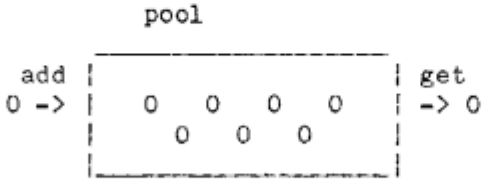

Figure 3.1 Pool

(2) Sequence

A pool of class 'sequence' allows accessing an element with its position in it. This means that each element is identified by its position in a pool. A sequence can be illustrated in the following figure.

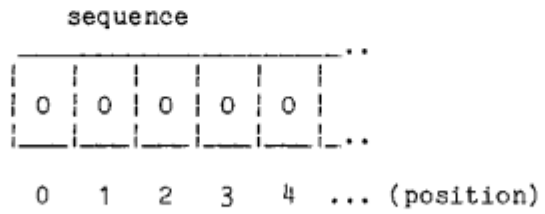

Figure 3.2 Sequence

There are two major subclasses of class 'sequence' -- class 'array' and class 'list'. An array can keep a fixed number of elements in it. Some operations to an array are:

- o To put an object in a position

  :put_at(Pool,Object,Position)

- o To get an object from a position

  :get_at(Pool,Object,Position)

Any element of an array can be accessed with the same overhead, independently of position.

A list can keep any number of elements in it. The position of each element is changed by adding and removing other elements in the list. Several of the operations on a list are

- o To add an object at a position

  :add_after(Pool,Object,Position)

:add_before(Pool,Object,Position)

- To get an object at a position

  :get_at(Pool,Object,Position)

- To remove an object at a position

  :remove_at(Pool,Object,Position)

Some demerits of lists are that the accessing overhead depends on the position of an element in a list, and that some memory words are left as garbage, after an element is removed from the list.

(3) Index

An index is a pool whose element is identified by its associated key value. A key value is either an atomic term or a string. Some of the operations on an index are:

- To add an object with a key

  :add_at(Pool,Object,Key)

- To get an object with a key

  :get_at(Pool,Object,Key)

- To remove an object with a key

  :remove(Pool,Object,Key)

An index is illustrated below.

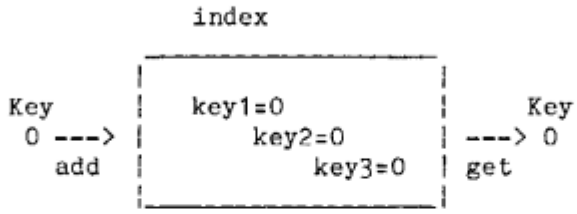

Figure 3.3 Index

An index is implemented as a hash array which is an array of lists. Each entry of these lists has a key and an object. A key value is hashed by a standard hash function to calculate a position in the array. Then the list is searched for a matching entry with the given key.

Note that since a position of an element in a sequence is considered as a special type of key value, a sequence is thought as a type of index. However in implementation, class 'index' inherits class 'array', which is a subclass of class 'sequence', through class 'hash_array'.

A sparse array is an array where the number of stored elements is expected to be much less than the largest position of the elements. It is implemented as an index in which a position of an element is used as a key.

(4) Bucket

It is often useful if we have a multi-layer (multi-dimensional) sequence. A bucket is such a sequence, whose elements are also sequences.

First, to identify an element in a bucket, we introduce a path-position which is represented as

[Position1, Position2, ...]

where 'PositionN' specifies the position in the N-th layer of the bucket.

Now in order to define a bucket, we have only to add a few new predicates to each sequence class. We define a mixin class 'as_bucket'. For example, it defines a get operation as

```
:get(Bucket,Object,[Position|Path_position]) :-
        :get(Bucket,Sub-bucket,Position),
        :get(Sub_bucket,Object,Path_position).
```

When a sequence class has a get operation defined as

```
:get(Sequence,Object,Position) :- ...
```

a bucket can be defined by inheriting class 'as_bucket' first, and a sequence class next. Then a get operation of this bucket works as it is supposed to do.

Furthermore, we can have a multi-layer index by defining a path-key as

```
[Key1, Key2, ...]
```

and letting each layer of a bucket decide what it should do with its key or position in the path-key. Note that a path-key may contain a position as its component.

(5) Other pools

Other pools will be also defined. Some of them are bags and sets, and sorted sequences. A bag is an unordered pool, where an element is not identified by its position, and a set is a bag which does not allow any duplicated objects. A sorted sequence is a sequence where each element is sorted according to its predefined order.

(6) Tap

Access to elements in a pool by specifying their positions is called direct access, which is so far described. Another accessing mode is sequential, where elements in a pool are accessed one after another without specifying their position each time. A tap is an object which supports sequential accessing.

A tap, which is attached to a pool, pours out objects from the pool. It is shown in the figure below.

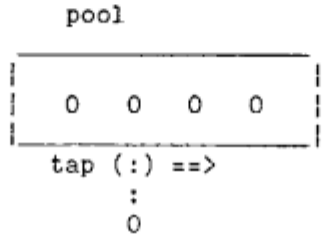

Figure 3.4 Tap

A tap is created on a pool by

```
:tap(Pool,Tap)
```

and we can call

```
:get(Tap,Object)
```

or

```
:put(Tap,Object)
```

on the tap to get or put an object at the next position. Also we can move a tap to a specific position by

```
:move(Tap,Position)
```

A tap can be implemented in general for any class of pool, if a pool knows the next position as:

```
:next(Pool,Current_Position,Next_Position)
```

Then a get operation is defined as

```
:get(Tap,Object) :-
        :position(Tap,Current_Position),
        :pool(Tap,Pool),
        :get_at(Pool,Current_Position),
        :next(Pool,Current_Position,Next_Position),
        :set_position(Tap,Next_Position).
```

However, if we want a more efficient implementation of a tap, we can define a tap class specific to each pool class.

4. Object Flow -- Streams

An object not only remains in a storage, but also flows from one storage to another through a stream. A stream is used for synchronization, communication, and mutual exclusion among processes.

(1) Stream

A stream is a pipe through which objects flow. It looks like a tap on a pool, though objects in a stream are not pooled. Two major operations on a stream are

- To insert an object into the stream

  ```
  :put(Stream,Object)
  ```

- To remove an object from the stream

  ```
  :get(Stream,Object)
  ```

If the stream is empty, a get operation causes a calling process to be suspended until another process puts an object. This feature of streams is essential for process interactions. A stream is shown in the following figure.

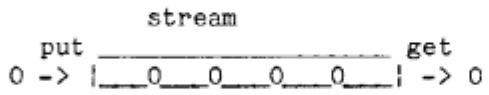

Figure 4.1 Stream

A stream can be shared by multiple processes. Each process can put or get objects from it at any time, and is served on a first-in-first-out basis.

A stream can be used for several process control mechanisms.

- o Semaphore

  A semaphore and p/v operations on it can synchronize processes.

```
:p(Semaphore) :-
        :get(Semaphore,Token).
:v(Semaphore) :-
        :put(Semaphore,token).
:init(Semaphore)
        :put(Semaphore,token).
```

  An integer semaphore can be implemented by initializing a semaphore so that it includes the specified number of tokens:

```
:init(Semaphore,Count)
```

- o Event

  An event synchronizes processes by passing an event code.

```
:post(Event,Code) :-
        :put(Event,Code).
:wait(Event,Code) :-
        :get(Event,Code).
```

o Message communication (as discussed later)

A stream with additional features will be defined by inheriting class 'stream' and some mixin classes. For example, priority control and bounded buffer control may be added.

(2) Channel

A channel provides a basic means for inter-process communication, as a stream of messages. An object which is to be sent through the channel, is packed into a message, and it is unpacked to an object on being received. A message is intended to be self-descriptive and has a sender slot (a channel for reply, if available), a receiver slot (a channel to receive it), and a body (an object to be sent). Class 'message' inherits class 'as_message' to include the first two slots. This mixin class can be used for any other message classes.

Class 'channel' inherits class 'stream' and has two basic operations for message transfers:

o To send an object to a channel

:send(Channel,Object)

o To receive an object from a channel

:receive(Channel,Object)

The other set of operations on a channel is also defined:

o To send a message

:send_message(Channel,Message)

o To receive a message

:receive_message(Channel,Message)

so that a user program can deal with messages directly.

A channel is illustrated in the figure below, where 'M' represents a message.

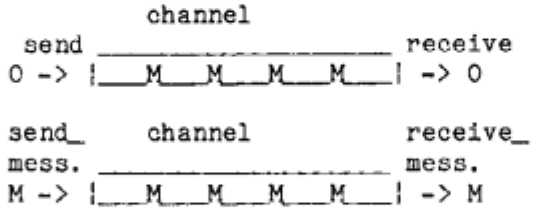

Figure 4.2 Channel

Note that in our model, a message is not sent directly to a process, but to a channel from which a process can receive it. This separation of a channel from a process has been taken, because we want to make a process to be able to receive messages from multiple message sources.

(3) Port

A port is a keeper of channels, so that a process can do two-way communication with a sort of virtual circuit. Two ports are connected before the communication begins between them, and they are disconnected when the communication ends. We allow a port to be connected to many ports. A send operation broadcasts a message to all connected ports, and a receive operation receives a message from all the ports in the order of arrival. Some operations on a port are

- To connect to another port

  :connect(Port,Another_port)

- To disconnect all the connected ports

  :disconnect(Port)

o To disconnect a specified port

:disconnect(Port,The_other_port)

Class 'port' inherits class 'channel' for its input channel, and holds a list of output channels. A connection is done so that each of the two ports is included into the list of output channels of the other port and so is the other way round. A port and its connection are shown in the following figure.

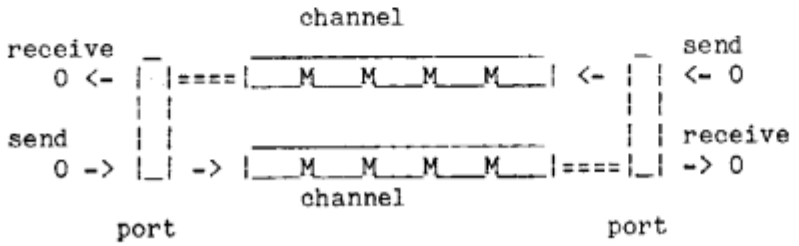

Figure 4.3 Port

(4) Stream pool

Although a single stream can serve for a process to get objects put by other processes in a first-in-first-out order, it is often desirable that a process can get objects from one of multiple streams by merging them. But it is not possible to implement such a multiple receive operation with the facilities so far introduced, because a get operation on each stream causes a calling process to wait until an object is put, if the stream is empty. Therefore, we define a stream pool which takes care of merging streams.

A stream pool (see the figure below) is a pool of streams with a multiple get operation defined as

:get(Stream_pool,Object)

or

:get(Stream_pool,Object,Stream)

We do not define a send operation on a channel pool, though.

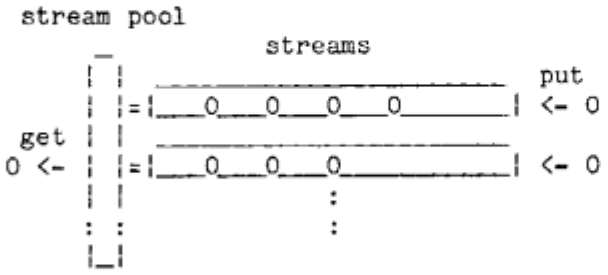

Figure 4.4 Stream Pool

Note that a channel pool is defined just as a stream pool, so that a process can receive a message from multiple channels.

(5) Source/sink channel

The input/output operations on i/o devices are supported by the i/o medium systems of SIMPOS. They are implemented as operations on a certain object, such as a file and a window, and are not considered as channels. However, at a certain higher level, these i/o operations must be also regarded as channels. Source/sink channels are the special channel for doing so.

A source channel is a channel to read data from an input device, and a sink channel is a channel to write data to an output device. An example of source channels is an input character stream from a keyboard, and an example of sink channels is an output character stream to a screen.

5. Execution -- Processes

So far we have not mentioned what executes operations on objects. Here we define a process and a program. At the supervisor level, we separate a process from a program, such that a process does what a program says. The reason is that the separation makes the structure and management of the supervisor clearer and easier.

(1) Process

In SIMPOS, a process is an active entity which executes a program. A process has four states -- running, ready (to run), suspended, and dormant. The dispatcher and the scheduler controls the states of processes. A running process is the one which is currently executing a program on a processor. A ready process is in a ready queue waiting to be run by the dispatcher. A suspended process is not in the ready queue. Its execution has been suspended for some reason, ordinarily when it tries to get an object from an empty stream, and when those reasons are removed, it becomes ready and is put into the ready queue by the scheduler. A dormant process is not managed by the scheduler. A process is in this state when it has just been created or terminated. An activation of the process is necessary to make it known to the scheduler.

A process is also defined in SIMPOS as an object which accepts operations to control the state of processes:

- To activate a process

  :activate(Process,Program,Goal)

- To suspend a process

  :suspend(Process,Reason)

o To resume a process

:resume(Process,Reason)

o To terminate a process

:terminate(Process)

When implemented on PSI, each process is given a hardware process environment of PSI, such as a register set, a stack group, and heap memory. The architecture of PSI restricts the number of processes in a system to 63. Out of these several processes are allocated to interrupt processes, so that the available number of ordinary processes is around 50.

(2) Program

A program says what a process should do. A program definition is nothing other than a class definition, except for a few conventions made on it.

First, a program which a process executes is an instance of a program class, not a class object itself. This is because the same program should be executed independently by many processes which share the code but have different data for each instance. The instance slots of a program instance are local to the program and are accessible at any point of the program, whereas the class slots are shared by all the instances of the same program.

Secondly, a so-called main program is defined as an instance predicate of a program. The body of the program can be a general Prolog program, which calls local predicates defined in this class, or class/instance predicates defined in this class and others. For example, a goal predicate, which is a default predicate for the main program, is usually defined in a program definition as

instance

```
        :goal :- p(X),  (X).
local
        p(X) :- ...
        q(X) :- ...
```

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

```
:invoke(#program,Goal)
```

where 'Goal' specifies the main program. When this predicate is called, a program instance is created and pushed into the invocation stack of the calling process. However, when a process is activated with a given program, a different predicate, which is defined as an instance predicate on a process,

```
:activate(Process,#program,Goal)
```

is to be used instead. this is because a program instance should be pushed into the invocation stack of the process to be activated.

(3) Interrupt process

Some processes in the system are allocated to interrupt processes, such as trap handlers, device handlers, and the garbage collecter. These interrupt processes are not dispatched by the supervisor, but by the hardware at an interrupt or trap. Therefore, another process class, 'interrupt_process', is defined to manage them differently. Some of the differences between an

ordinary process and an interrupt process are that

o When an interrupt process is activated, a trap is raised to dispatch it by hardware. When an ordinary process is activated, it is placed in the ready queue by the supervisor.

o When an object is put into a stream on which a process waits, it is resumed. For an interrupt process, a resume operation causes a trap to dispatch it. For an ordinary process, the same resume operation calls the software scheduler. Note that a stream does not have to distinguish the class of the process.

(4) Boot process

Another special kind of process is a boot process. It is a process which is created and activated by the hardware when the system is bootstrapped, before the supervisor knows it. We define class 'boot_process' which inherits class 'process', but overrides a create and an activate operations, so that the boot process comes under control of the supervisor after the system has started, by the boot process itself calling these operations.

6. Environments -- Directories and Worlds

Most existing programming languages support both local and global environment for a program execution. A local environment is one local to an invocation of a given program and may be either temporary or permanent during the invocation. A global environment is one global to all programs, and is usually permanent.

Prolog has a local and temporary environment of a predicate as its calling parameters and a global environment as asserted clauses. In addition, Prolog

has a few implicit global environments, for example, to keep standard input/output files which are refered to by default. But in general, it is not probable nor efficient for each program to keep an environment as its calling parameters and to pass it around to its subordinate programs. It is much easier to keep an object in a global environment which can be refered to at any point of the program, even though global environments make a program less understandable.

In SIMPOS, we support global environments which keep objects accessible to a process at any time. Each process has within its execution environment -- a program invocation, a library, and a directory. The first of these was explained in the previous section. It provides a local environment to a process, not to a predicate call. The latter two are defined below.

(1) Library

A library holds a collection of classes. A class object can be retrieved with its class name. These class objects are in a global environment, because they can be accessed by all processes.

Furthermore, a class object can keep an ordinary object in its attribute slot. These objects in class slots are also considered to be in a global environment. For example, if class 'directory' has a class slot of the name 'root', a root directory can be retrieved by the class predicate

```
:root(#directory,Root_directory).
```

Note that the class slots defined in each class cannot be increased in number at runtime.

(2) Directory

A directory is a pool (an index), in which each object is associated with its name. The number of objects in a directory is not fixed and can be increased at runtime.

On a given directory, a user can perform operations such as

- o To retrieve an object with name

  :find(Directory,Object,Name)

- o To add an object with name

  :bind(Directory,Object,Name)

- o To remove an object with name

  :remove(Directory,Object,Name)

Any class of object can be included in a directory, and even another directory can be included. By doing so, a tree structure of directories can be constructed. We call this a directory tree. A pathname is used to identify an object in this directory tree. For example, a pathname can be represented as

"Name1>Name2>..."

and is used in a retrieve operation

:retrieve(Directory,Object,"Name1>Name2>...")

A specified object with the above pathname is retrieved in the following steps.

- o First find an object with Name1 in Directory.
- o Then find an object with Name2 in the directory found above.
- o Repeat a search for an object until the path-name is exhausted.

A user process may keep a directory (tree) to insert and retrieve any object for its own uses.

(3) System directory tree

The supervisor keeps the system root directory in a class slot of class 'directory'. A directory tree, whose root is this directory, is said to be a system directory tree and is a global environment to all processes in a system. A pathname is extended to allow the form of

">Name1>Name2>..."

where the first '>' means the system root directory. This type of pathname is used to retrieve an object without specifying a directory explicitly. For example, by calling a class predicate

:retrieve(#as_global_object,Object,">Name1>Name2>...")

an object will be retrieved in the system directory system. On the other hand, if the first '>' of a pathname is omitted as in

:retrieve(#as_global_object,Object,"Name1>Name2>/...")

this predicate assumes a world of the calling process as a specified directory tree. (See World.)

(4) World

A world keeps a dynamic and global environment of a process. It is constructed as a sequence of directories, providing the same interface with a directory. Note that an element of a world is not necessarily a directory, and that it can

be any object, for example, another world, as far as it supports the same interface as a directory.

With a world, we can have many objects with the same name in a single global environment. When retrieving an object with the name, we will get the one which is included in the first directory with such an object. Assume that a world includes two directories such as

```
World = [..., Directory1, ..., Directory2, ...]
```

where 'Directory2' includes an object with a name 'input'. As long as a process is under this world, retrieval of an object with this name produces this object. But if a process inserts another object with the same name into 'Directory1', this object will be retrieved later with the same name. This looks as if an object with a name is overridden by another object dynamically, yet preserving the old object.

As mentioned before, each process has a working world which is a local environment of the process and is used as a default user directory tree when an object is retrieved with a pathname. The working world is kept in an instance slot of the process and is retrieved at any time by

```
:world(#process,World)
```

Though this is a class predicate, this call returns the working world of the calling process.

(5) Global object

We call objects, which can be inserted in the system directory tree or a world,

global objects. They have additional operations and information for them such as pathname retrieval, protection, profile, version, and mutual exclusion, by inheriting a mixin class 'as_global_object' in general.

- o Protection

  A global object should be protected from illegal use in processes. Protection of an object is controlled by the access permission of the object to accessing processes. Each global object has an owner and protection seals for user types, which are owner, members of the same group as the owner, public, super users. Note that class 'user' and class 'group' must be defined to describe users and groups of the system.

- o Profile

  A profile of objects gives some information such as creation date, modification date, and reference date. These are to be initialized and updated from certain instance predicates.

- o Version

  Users often want to name the modified object with the same name, and yet to distinguish it from the old (original) object. Giving a version number, together with a name, to an object accomplishes this.

- o Mutual exclusion

  The mutual exclusion of operations on an object is necessary if an object is shared by many processes and keeps its consistency. It is implemented by an object lock, so that each object has a stream for a lock.

The actual implementation of these facilities is still under consideration. It

is partly because some hardware supports are preferable to achieve them in reasonable overhead, and partly because we think that they are important but not crucial to our prototype system yet.

7. Permanent Objects

The objects, described so far, are created (or instantiated) in main memory. Each time the system goes down, these objects vanish. In this sense, they are temporary. Sometimes we want to create a permanent object which will exist even after the system goes down. It is possible but inconvenient that we have a permanent object by creating it each time the system is brought up. In SIMPOS, the supervisor, with the aid of a file system, supports permanent objects by storing them in file storage.

An object is stored in file memory as a record of a file, which is called an instance file. It is not possible, in general, to store an object the same as it is in main memory. Data convertions between object and record must be defined. In the current version, we assume that each class, whose instances are to be permanent, should define its own conversion predicates, so that the supervisor calls them when accessing a record.

We define a mixin class 'as_permanent_object' to be inherited by any permanent object class. This provides an instance file as a class attribute, a record pointer as an instance attribute, and access operations to file records.

(1) Instance file

An instance file is a table (fixed-length record) file, where the record of an object is stored. Usually it is kept in a class attribute of each permanent

object class.

A record in an instance file is identified by a record pointer, which corresponds to an object pointer in main memory. A record pointer consists of a file identifier and a file marker, so that a record is uniquely identified in file memory. A file identifier is represented as a file number when in file memory and as a file object when in main memory. A file marker indicates a record position in the instance file.

The file system supports directory files. A directory file is a file which associates a record (pointer) with its key (name). The operations on a directory file is just like those on a directory in main memory. For example, a record with a given name can be found as

```
:find(Directory_file,Record_Pointer,Name)
```

(2) Permanent directory

A permanent directory is a directory which is linked to a directory file and works as its temporary copy. When a user tries to retrieve an object with a specific name in a permanent directory, it first tries to find an object with the name, and returns the one found. Otherwise, it will find an object (a record) in the associated directory file with the name, and read in the record and convert it to an object.

By binding a permanent directory into a system directory tree, we can retrieve a permanent object just as a global object.

8. Remote Objects

Remote objects are those which exist in other machines connected via a network system. An operating system, which supports a personal computer network, should deal with these remote objects, just as objects in a self node.

(1) Object identification

An object in a self node is identified by an object pointer, which is given at creation time. However, an object pointer is effective only in the same machine, since it might be changed during garbage collection. Hence, to point to a remote object, we have to use another method.

A remote object will be identified by a pair of a node and an object number. A node is an object which represents a network node. An object number is a unique number to identify an object in that node. (The node objects are stored as permanent objects in each node and can be retrieved with their given pathnames.) An object table which associates the object pointers with their object numbers must be maintained in each node by the network manager.

An operation on a remote object is encoded to a message whose body consists of an operation identifier (represented by a string), an object number, and parameters (including objects and terms). Then it is sent to the network manager which in turn sends it to the remote network manager on the node. When receiving this message, the remote network manager decodes it to the operation and performs it on the specified object.

We define a mixin class 'as_remote_object' to be inherited by any remote object class. It provides operation-to-message encoding and message sending. Also we define another mixin class 'as_pervasive_object' to be inherited by any object class whose instances are accessed from other nodes. It provides message-to-

operation decoding and operation execution.

(2) Remote directory

A remote directory is a directory which is linked to a directory on another node. An object which is retrieved from a remote directory is a remote object which represents the corresponding object in the remote node.

After connected to another node, the manager creates a remote directory for the system root directory whose object number is assumed as 1, and gives it to the node object. If necessary, this remote directory can be inserted into the system directory tree of the self node, so that any object in it can be retrieved with its pathname, just as a global object.

(3) Remote channel

As a useful remote object, a remote channel is defined by a network system. As mentioned before, a channel is a means for inter-process communication. When a system is connected to another node, inter-process communication should be allowed in the same way even between processes on different sites. A remote channel does this job.

A remote channel represents a channel on another node. Sending a message through the remote channel on the self node is effectively sending the message to the channel on the remote node. Note that receiving a message from a remote channel is not allowed, because it will complicate the implementation of remote channels.

Of course, before the remote communication begins, a sender process must know

the channel which a receiver process has in a remote site. It is possible if a receiver process, after creating a channel, inserts it with an agreed pathname into the system directory tree, or if a sender process retreives an remote channel for it with the agreed pathname through the remote directory tree. Note that this is nothing other than the way which is taken by the processes both at the same site.

9. Conclusion

We have described the supervisor of SIMPOS in the above sections, where all the constructs are defined as classes. According to these ideas and specifications, we have now finished the functional design and the coding. Although we have never implemented an operating system with an object-oriented approach before, we have found that the class mechanism together with inheritance greatly reduces the specification and implementation efforts.

Currently the number of classes in the supervisor and the estimated code size are as follows.

	No. of classes	Lines of codes
pool	61	2,000
stream	37	800
execution	11	1,100
environment*	15	1,500
timer**	11	600
total	135	6,000

*) A library is not included.
**) Timer management has not been discussed. It supports clocks and watches.

We should say that there are still some remaining problems in the current design. Some of them are about global objects mentioned before. We expect that they will be solved and implemented in the near future.

Acknowledgements

We would like to thank all the researchers of ICOT 3rd Lab. for their comments on the current design, and the implementation group for the detail design.

References

[1] H.Nishikawa, et al., "The Personal Inference Machine (PSI): Its design philosophy and machine architecture", ICOT TR-013 (June 1983).

[2] T.Chikayama, et al., "Fifth generation kernel language version 0", ICOT internal document (June 1983).

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

[4] T.Hattori, et al., "SIMPOS: An operating system for a super personal computer PSI and its design overview", to appear as ICOT TR.

[5] T.Hattori, "The concepts and facilities of SIMPOS file system", to appear as ICOT TR.