TR-82-007

A Prototype System for Office Procedure Automation*

by

Yang-Chang Hong, Yui-Wei Ho,
Chen-Hshin Ho, and Te-Son Kuo

ABSTRACT

In this paper, a system is outlined which can provide office procedure automation. The system is mainly based on an extention of Petri nets for modeling office work as a set of human-interaction procedures. It includes five major modules: supervisor, execution monitor, mail manager, form manager, and data manager. These are processes under VAX-11/780 and being implemented at the Academia Sinica. Their design integrates major office tasks and, thereby, has the important feature of providing office workers with a single interface.

Keywords and phrases: Office Procedure Automation, Inter-Process Communication, Token Machine, Automated Office System

* This work is supported by Telecommunication Labs, R. O. C.

1. Introduction

Office automation has attracted and held great interest of the computer science researchers and office data processing community in recent years. There have been software- and hardware-intensive tools emerging as an aid in handling office tasks such as editing, filing, mailing, analyzing and transforming data. Although these tasks can be automated individually, they are initiated and directed by the people. Computer systems in this case do not play an active role, where the user is charged with the information flow control. In automated office systems, the design challenge is how the office tasks are coordinated and integrated in such a way that their initiation and cntrol can be accomplished by the system [2,6]. People can then do more creative tasks.

Several software-intensive tools attempting to automate offices in the sense mentioned have been proposed and reported in the literature [1,3,6,7,8,9]. The system described in this paper tries to achieve the same purpose. It is mainly based on Petri nets [5] incorporated with data graph which model office work as a set of predefined sequences of activities interacting with database and stations -- i.e., processing units (see appendix A and [4] for formal model definition and its detail). It consists of five modules, which are processes of VAX-11/780: namely supervisor, (Petri-net) execution monitor, mail manager, form manager, and data manager (see Figure 1). Associated each has one or more VAX mail-boxes for inter-process communications. Mail manager provides facilities for communications between offices. In the current stage, a mail terminal is used to simulate communications with outside offices. Supervisor accepts

requests for service from the administrator console, mail manager, or monitor (that is, from another office procedure). It creates a monitor process once a request for service is entered into the system and passes to the monitor the name of the procedure to be invoked and required parameters for initiation. A monitor process, once created, will be driven by the internal form of the invoked procedure. The execution of each procedure primarily follows the token machine concept (see appendix B and [5] for detail). Since the monitor is written in reentrant code, there is in fact only one single copy residing in the memory. Form manager deals with interactions between station and system. It receives commands from the monitor, displays forms or memos at stations, and passes worker-supplied messages back to the monitor. Data manager deals with queries which retrieve and manipulate data in the database. It provides as a high-level interface between monitor and VAX file system. A high-level nonprocedural specifications language is also provided for describing office procedures, which could then be translated into an extended Petri net (in its internal form) and run.

The body of the paper is divided into three parts. In the first part the overall system architecture is described. The second part is concerned with the internal forms (i.e., data structures) of office procedures. The third part is concerned with detailed implementation of the system modules. This is followed by a summary and the status of the project.

2. Overall System Architecture

The architecture of the office procedure automation system (OPAS) is sown in Figure 1. It consists of five modules: supervisor, (Petri-net) moniotr, form manager, mail manager, and data manager. They are processes under VAX-11 VMS operating system. Each process uses VMS "event flag" and "mailbox" facilities as a means of interprocess communication. Interactions among processes are also shown in Figure 1 (to be detailed later). These processes are functionally categorized into three classes:

1. System supervising -- Supervisor
2. Petri-net driving (i.e, procedure execution) -- Monitor
3. Activity serving -- Form manager, Mail manager, and Data manager.

2.1 Supervisor

The supervisor controls the whole system, including:

- Create all other processes,
- Accept procedure invocation demands from the administrator console, mail manager, or monitor,
- Interactive with the administrator console for system control, and
- Maintain an office log file.

Its major function is the invocation of office procedures. A procedure will be invoked if a procedure invocation command is received from the administrator console or if an MSG-PI message is read from the mail manager or monitor processes (i.e., other office procedures). The invocation include creating a monitor process and

putting the procedure name (through which the monitor can obtain its internal form) and the initiation parameters to spervisor's mailbox which are then accessible by this created monitor. Once the internal form is initiated, it will drive the monitor. In this case, we say that a "procedure instance" has been created and in progress.

The supervisor contains an office control table (OCT) for keeping track of information such as, procedure instance ID, starting time, priority, status, etc. for instance management. The administrator console has the right to make inquiries about any instance status, adjust its priority, pause/stop its execution, etc. One entry is added to the OCT table whenever a procedure instance is created. Any termination messages (MSG_MT) or error message (MSG_MER) of procedure instances will be sent directly to the supervisor which then reports to the administrators and records them in the OCT table and the office log file.

2.2 Monitor

A monitor process is created if a request for service is submitted to the supervisor. It then be driven by the internal form of the invoked procedure, which leads to the execution of the corresponding procedure instance. The execution follows the token machine concept (its realization is detailed in the Section 4).

Office activities performed in the monitor are divided into five types:

- Working data manipulation : Such as simple computation, assignment, etc.
- Form manipulation : Forms are interaction media between

system and office station. Form manipulation is accomplished by the form manager.

- Database data manipulation : The monitor uses one simplified relational algebra to retrieve or update office data in the database.
- Security identification : The identification is accomplished by examing if the password associated with the sensitive field of the form is matched against the worker-supplied password. This could be extended to a sophisticated security process if necessary.
- Time scheduling: Time predicates such as WAITFOR, UNTIL, etc. are allowed to be specified in any above activities. The monitor has to be able to handle this type of scheduling.

In the current stage, we do not emphasize on supervisor intervention to control the monitor process. Since the monitor is a subprocess of the supervisor, however, the supervisor can thus get full control over the monitor using the process control facilities provided by VMS.

2.3 Form Manager

Form manager deals with interactions between procedure instance (i.e., monitor process) and office station (or worker). The interaction media are forms. If a monitor process needs a "form i/o", an MSG-FP request has to be sent to the mailbox of form manager. The request includes the internal representation of the form to be displayed and the destination terminal(s). Corresponding to each agent terminal is a request queue which holds all the MSG-FP requests using that terminal as destination. The requests in the queue are

served on an FIFO basis. The form manager will then send worker key-in information back to the requesting monitor if input data to the monitor is needed. This completes the whole process of one form i/o activity. Note: Any sensitive field in the form could be associated with a password field to identify the worker/ station (monitor's job).

2.4 Mail Manager

Mail manager deals with interactions between offices. Any executing procedure may need messages from outside world. In this stage, this is accomplished by sending mails (MSG-FP messages) to mail manager. Before a hardcopy of the mail is made, it must be numbered and stored. This number would then be used as an ID of the mail for later references. When the mailed hardcopy is filled in and returned, the operator uses that number to notify the mail manager. The manager is prompt to perform one form i/o for displaying the stored mail on the CRT terminal so that the values returned can be properly keyed in. The input data are finally sent back to requesting procedure.

Invoking procedures from outside offices is another major function. This is done by sending the MSG-PI message (which includes the required information for procedure invocation) to the supervisor.

This mail process uses a CRT terminal plus a printer for communicating with outside offices. Note: In the second stage (on going), it is designed to be supported by an electronic mail system with a set of communication protocols for office information exchange.

2.5 Data Manager

Data manager provides procedure instances with a high-level

interface to the office database. It can be seen as a rather simplified relational DBMS in the current stage. A high level interface to a relational DBMS is under way.

All the executing procedure instances are users of this simplified DBMS. If data retrieving or updating is needed, they send to the mailbox of data manager the relational algebra command (message MSG-DBO). The results are sent back to the requesting procedure via message MSG-DBR. Currently, the available algebra operations are SELECT, UNION, and MINUS.

3. Internal Forms

Internal forms basically are a set of data structures through which an office procedure described in a specification language is translated and run. This design must be able to provide information for the following tasks:

- Procedure execution control,
- Procedure execution (i.e., Petri-net driving), and
- Activity realization.

Corresponding to these tasks are the three types of data structures:

(1) Procedure control data structure -- It contains one block, Procedure Control Block (PCB).

(2) Petri-net driving data structure -- The driving primarily follows the token machine concept.This type of data structure consists of two tables, namely , Marking Table (MT) and Active Transition List (ATL).

(3) Activity realization data structure -- It contains seven tables, namely, Transition Detail Table (TDT), Predicate Expression List (PEL), Predicate Component list (PCL), Activity List (AL), Actural Argument List (AAL), Objects Description Table (ODT), and objects Data Area (ODA).

The following subsections are the layout and illustration of these data structures.

3.1 Procedure Control Block (PCB)

The layout and explanation of PCB, one entry per procedure instance, are depicted below:

[Procedure Control Block] layout

type	field	comment
char*8	PRC-ID	procedure-ID, i.e, the file name of inter-nal forms of the procedure invoked
char*24	PRC-AT	activation time, VMS ASCII time
int	PRC-PRI	execution priority
char*1	PRC-STA	execution status
int	PRC-CMK	current Petri-net marking, pointer (ptr) pointing to one TMT entry
int array	PRC-CTR	current executng transition, ptr to one TDT entry
int array	PRC-IPL(n)	invocation parameter list, a list of ptrs to ODT entries which are invocation parameters.

The PCB function is to provide information for controlling office procedure instances. The priority PRC-PRI of any executing procedure can be properly adjusted by the supervisor. Its status PRC-STA must reflect to one of the states (e.g., ready-to-go, running, blocked, etc) during execution. A procedure "instance" ID is formed by concatenating its activation time PRC-AT to the PRC-ID. We call it as the procedure effective ID. It serves as the identification name of the corresponding instance in the system. The PRC-CMK and PRC-CTR record the current execution point of the corresponding instance. PRC-IPL is a variable list of pointers pointng to ODT entries. These entries wll be established during the course of procedure instance creation.

3.2 Petri-net Driving Data Structures

Two tables, namely, marking table (MT) and active transition list (ATL) are designed to realize the Petri-net driving.

[Marking Table] layout ₹₹ one entry per marking

type	field	comment
int	MK₹NAT	number of active transitions in the corresponding marking
int	MK₹FAT	ptr to the 1st active transition in ATL

One entry in MT stores one marking in a reachability tree derived from the token machine. All active transitions in a marking are stored in the consecutive entries of the ATL table. The MK₹FAT, used to point to the first entry, and the MK₹NAT are sufficient to fetch all te entries in the marking. The first entry in MT is the initial marking and the terminating marking is indicated by an entry with MK₹NAT=MK₹FAT=Ø.

[Active Transition List] layout ₹₹ one entry per transition

type	field	comment
int	AT₹TP	ptr to tne internal form of the corresponding transition (i.e., to one TDT entry)
int	AT₹OMK	ptr to next marking in MT (i.e., the output marking obtained from the firing of the correspond₹ ing transition)

Since each transition in a net could be active in more than one marking, the internal form associated with that transition is not stored in ATL. Instead, it is stored in TDT and a pointer, AT₹TP, in ATL is used to point to it.

Figure 2 is an example for illustrating the use of the tables MT and ATL.

3.3 Activity Realization Data Structures

This type of data structure includes the internal forms (TDT, PEL, and PCL) of transitions and their predicates, internal forms (AL and AAL) of activites in the transition, and ODT and ODA tables for storing the program objects used in the office procedure. They are explained below.

[Transition Detail Table] -- one entry per transition

type	field	comment
char*20	TR-SN	symbolic name of the corresponding transition
char*1	TR-FIRE	in-firing flag
int	TR-XCH	exclusive chain link (ptr to next transition in the chain)
int	TR-PEP	starting location of predicate expression of the corresponding transition in PEL
int	TR-NAC	number of activities in the corresponding transition
int	TR-FAC	ptr to the 1st activity in AL
int	TR-CAC	activity counter

Each TDT entry (plus PEL, AL, etc.) describes an internal form of some transition in the procedure. It also contains some control items for transition selection and firing. The internal form must include predicates and activities associated with the transition. Since both predicates and activities are variable-length items, they are stored in PEL and AL, respectively. The TR-PEP and TR-FAC contain pointers pointing to the starting locations of these two items. The TR-CAC counts the number of activities of some transition that have been executed. TR-FIRE and TR-XCH will be detailed in the next section.

[Predicate Expression List]

type	field	comment
int array	PE-PEL	predicate expression list

[Predicate Component List]

type	field	comment
char*1	PC-OPR	atomic predicate operator
int	PC-OPN1	atomic predicate operand 1, ptr to an ODT entry
int	PC-OPN2	atomic predicate operand 2, ptr to a ODT entry

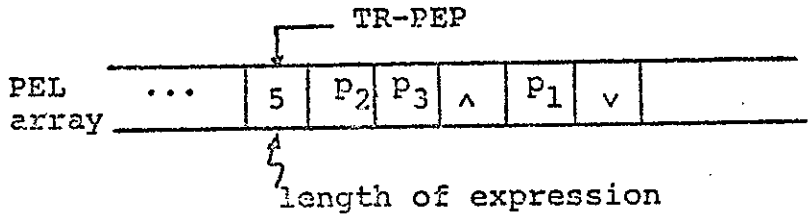
An atomic predicate (or called the predicate component) is of the form

[(operand 1)] op (operand 2)

where op may be a comparison operator, or other operators, e.g., EXIST, UNTL, etc., and [ ] denotes an option. A predicate or predicate expression, in general, is a Boolean expression of atomic predicates. All the predicates of the transitions are stored in PEL and PCL. PCL stores the atomic predicates and PEL stores the predicate expressions in reverse polish notation. For example, consider the predicate expression

"(A=∅)∨(B=1)∧(C=2)"

A=∅, B=1, and C-2 are atomic predicates and stored in PCL. The expression in PEL looks like:



where p1, p2, and p3 are the addresses of the three atomic predicates

(A=Ø, B=1, and C=2) in PCL,respectively, and"∧" and "v" are Boolean operators.

[Activity List] laout ₸₸ one entry per activity

type	field	comment
char*1	AC₸COD	activity code
char*1	AC₸NAG	number of arguments in the corresponding activity (argument list length)
int	AC₸FAG	ptr to argument list in AAL

[Actual Argument List]

type	field	comment
int array	AR₸AAL	array for arguments in some activity

AT and AAL describe activities in each transition. Each activity is assigned one code during translation. Like the case above, the variable item, the argument list is stored in a separate table AAL. The AC₸NAG and AC₸FAG are used to fetch this list.

[Objects Description Table] layout

type	field	comment
char*2	OD₸TYP	type information of objects
char*2	OD₸LEN	length information of objects
ch ar*2	OD₸XD	additional description
int	OD₸STA	ptr to char array ODA

[Objects Data Area]

type	field	comment
char array	OD₸ODA	storage for all objects in a procedure

ODT and ODA allow the objects defined in an office procedure to be processed. The object values in ODA are stored as a string of characters whose starting location is in OD▼STA.

4. Implementation of System Modles

4.1 Some Notes on OPAS Implementation

4.1.1 Form Handling

Forms are a major medium of communication between system and office agent. Any form instance transmitted between monitor and form manager or between monitor and mail manager consists of a list of form fields, called the "form i/o packet." Each field is defined by two parts, namely, the descriptor part and the text part. The descriptor part describes the field in terms of its type, length, location , etc., and the text part gives the field value. A form driver routine residing in form manager and mail manager serves as displaying forms on DEC VT100 terminals, as well as accepting field values input by the workers. Its design allows several simple editing functions to be performed. For example, the reverse-background display of input fields, automatic field advancing for a list of fields, and the abili-ty to permit the worker to move the cursor freely are provided in current stage. It will be extended to provide facilities for storing form i/o packets in a form database so that forms can be managed as they are done in manual office systems.

4.1.2 Initialization Parameters

Many procedures need some initialization data before they are invoked for execution. For example, in an order-processing procedure, the initialization data may include tne customer ID, the amount of goods requested, etc. In OPAS, this kind of data is called the "invo-cation parameters." They are designed and defined by the office designer during procedure preparation. Whenever an office procedure

is translated, the designer is asked to define an "invocation parameters entry form" using some facility provided by OPAS. This form will then be saved in the inernal form file of the corresponding procedure. It will be displayed on the CRT terminal for accetping invocation parameters if any request for service is made.

4.1.3 System Generation

Before the OPAS system can get operation, several tasks have to be done:

1. All the office procedures translated must be registered and stored in the system.
2. An Invocation Control Table (residing in the supervisor as well as in the mail manager for relating each registered procedure to its invocation parameter entry form) must be built.
3. All the global relations must be generated and stored in the office database.
4. All the OPAS terminals must be assigned.

These tasks can be accomplished by using the facilities provided by OPAS.

4.1.4 Bootstraping OPAS

OPAS is bootstrapped by running the supervisor process, which in turn creates other processes in OPAS in such a way that the system is ready to accept requests for service from the administrator console and the mail manager.

4.2 Use of VMS System Services

Three VMS system services make the implementation of the OPAS

system successful. They are mail box, AST i/o, and event flag. As described previously, associated with each process is one or more VMS software mailbox for inter-process communication. The AST i/o (Asynchronous System Trap) plays a rather important role in the design of OPAS processes. It enables each process to deliver an asynchronous mailbox reading and continue. Once the message is ready to read, the process s interrupted (asynchronously trapped) to execute a special program section for acceptng (and processing) the message. When finished, it goes back to its routine work. The AST i/o is also used by form manager for simultaneously handling all the agent terminal i/o's -- i.e. i/o between system and station.

The event flag is another VMS-provided facility for inter-process communication. The monitor process uses it to handle reading mails from the mail terminal and forms in the system.

4.3 Implementation

Monitor

[Initializations]

1. Read in the MSG-MIP message.
2. Read in procedure internal form using procedure ID in MSG-MIP.
3. Set invocation parameter values.
4. Set PCB entries.
5. Create the Monitor mailbox (mailbox name = procedure effective ID)
6. Start Petri-net driving (the following)

[Transition Selection]

1. Start with the 1st active transition of the current marking, check its predicate expression; if predicate expression is true, the transition is fired, else try rest active transitions.
2. If all active transitions of the current marking have a false predicate, Monitor hibernates.

[Transition firing]

1. Fire the selected transition by performing activities associated with it one by one.
2. After done, go to [marking advance].

[Marking advance]

1. Set the current marking to the output marking of the current transition,
2. Go to [transition selection].

Form Manager

[Initializations]

1. Read in "agent terminal assignment" file.
2. Assign agent terminals accordingly.
3. Create mailbox
4. Issue an AST mailbox reading to get request.
5. Hibernate and wait for requests.

[Main Loop] (once wakened from hibernating)

1. If "form completed chain" is not empty, process its entries (using input values gathered to build an MSG_FFR message and send it back to the requesting Monitor process); For any

terminal whose request queue is not empty, start next form I/O in the queue.

2. If "Start I/O chain " is not empty, process its entries one after another. (A form I/O is started by prompting the form and issuing AST reads to accept input values.)
3. If both chains are empty, then hibernate; else goto step 1.

[Mailbox AST reading handler]

(activated once the reading is completed)

1. Insert form packet request into "request queue" of the corresponding terminal.
2. If this is the 1st request in the queue, put the corresponding terminal into "start I/O chain".
3. Wake up Form Manager from hibernating.

[Terminal AST reading handler]

(activated whenever any agent types a character on the terminal.)

1. Store the character typed if it belongs to an input field.
2. Echo the character.
3. If the character is not a carriage return, issue the next one-character AST read; else (i.e., the form on that terminal is completed) put the terminal into "form completed chain".

Mail Manager

[Initializations]

1. Read in Invocation Control Table.
2. Create mailbox
3. Issue an AST mailbox reading to accept any requests.

[Mailbox reading handler]

1. If the form is to be returned (having input fields), register and save it into "Pending Form Table."

[Main Loop]

1. Prompt the ready message to the operator and wait for commands.
2. If the command is "enter form", use registration number given to retrieve the saved form packet and start form I/O; after done, use field values entered to build an MSG-FFR message and send it back to the requesting Monitor process.
3. If the command is "invoke procedure", use the procedure ID given to retrieve the invocation parameter entry form and request for initilization parameters; after done, build and send an MSG-PI message to the supervisor.
4. go to step 1.

5. A Summary

The Office Procedure Automation system outlined in this paper is being implemented in a VAX-11/VMS environment. This prototype system using a Petri-net-based model has been completely implemented. The data manager is now being extended towards a powerful, high-level interface to a database management system in such a way that the system can manage the messages (or forms). The mail manager is also being extended to provide inter-office communication facilities. We plan to extend this system to a distributed office system with chinese data processing capabilities.

6. References

1. Chang, J. M., AND CHANG, S. K., "Database Alterting Techniques for Office Activities Management," IEEE Trans. on Communication, COM-30, 1, January 1982, pp.74-81.

2. Chang, S. K., Knowledge-Based Database Systems, Chapter 14 (a forthcoming textbook).

3. Ellis, C. A., and Nutt, G. J., "Office Information Systems and Computer Science," ACM Computing Survey, 12, 1, March 1980, pp.27-60.

4. Ho, C. H., Hong, Y. C., Ho, Y. W., and Kuo, T. S., "An Office Work-flow Model," Proceedings NCS, Taiwan, 1981, pp.354-368.

5. Peterson, J. L., "Petri nets," ACM Computing Survey, 9, 3, September 1977, pp.223-252.

6. Tsichritzis, D., "OFS: An Integrated Form Managemetn System," Proceedings VLDB, 1980, pp.161-166.

7. Tsichritzis, D., "Integrating Database and Message Systems," Proceedings VLDB, September, France, 1981, pp.356-362.

8. Zisman, M. D., "Representation, Specification, and Automation of office Procedures," Ph.D. Dissertation, Wharton School, University of Pennsylvania, 1977.

9. IEEE Computers, 14, 5, May 1981, pp.13-22.

APPENDIX

A. PNB model definition

A Petri-net-based model is a 6-tuple $\Omega = (T, P, D, \Phi, \Delta, \Sigma)$, where

(i) T is a finite set of transitions;

(ii) P is a finite set of places;

(iii) D is a finite set of depositories;

(iv) $\Phi = I \cup O: T \rightarrow P$, where

I : is a mapping of a transition to its set of input places, and

O : is a mapping of a transition to its set of output places;

(v) $\Delta = i \cup o: T \rightarrow D$, where

i: is a mapping of a transition to its set of input depositories, and

o: is a mapping of a transition to its set of output depositories;

(vi) Σ is a set of doublet (c_t, a_t) over T, where

c_t : is a boolean expression associated with transition $t \in T$, and

a_t : is a simple or compound action of $t \in T$.

B. PNB execution definition

The execution rule of a PNB diagram can be defined by a double-let $\Gamma = (M, \tau)$ over Ω, F, and B, where

(i) F: is a set of incident markings. An incident marking f_t is a marking with only one token present in each place of I(t), $t \in T$.

(ii) B: is a set of outgoing marking. An outgoing marking b_t

is a marking with only one token present in each place of O(t), $t \in T$.

(iii) M: is a set of "reachable markings" including the initial marking m_0. Of course, m_f, the terminating marking, belongs to the set, i.e. $m_t \in m$.

(iv) $\tau : M \times T \rightarrow M$ is a "firable function" of transition t in T. If a transition t fires under marking m, we say $\tau(m,t) = m'$, with $m' = m - f_t + b_t$, where $m, m' \in M$, $f_t \in F$, and $b_t \in B$. A transition t fires under marking m if

a) $m \geq f$ and

b) c_t =True in (c_t , a_t)

A transition is enabled if only a) holds.

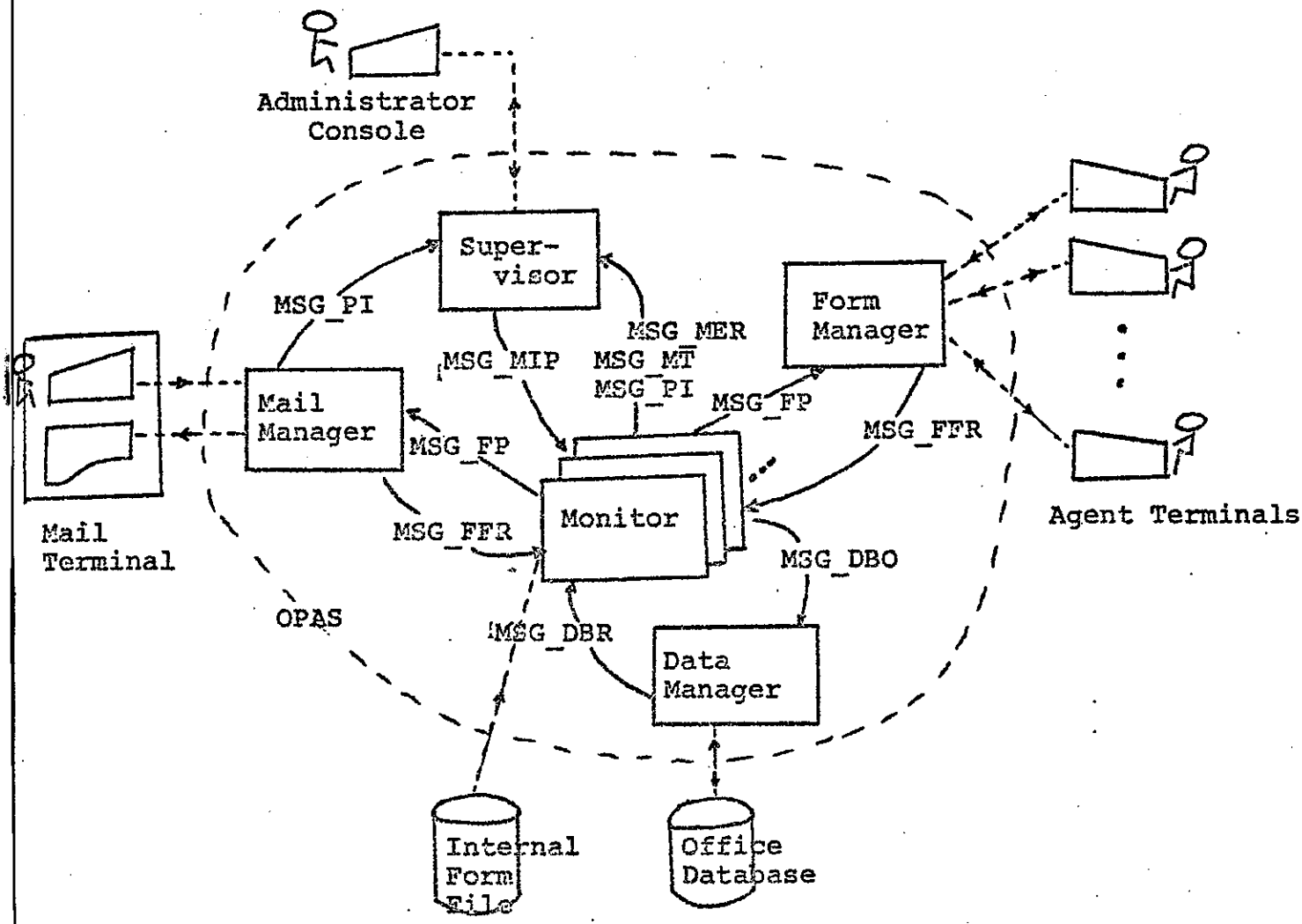
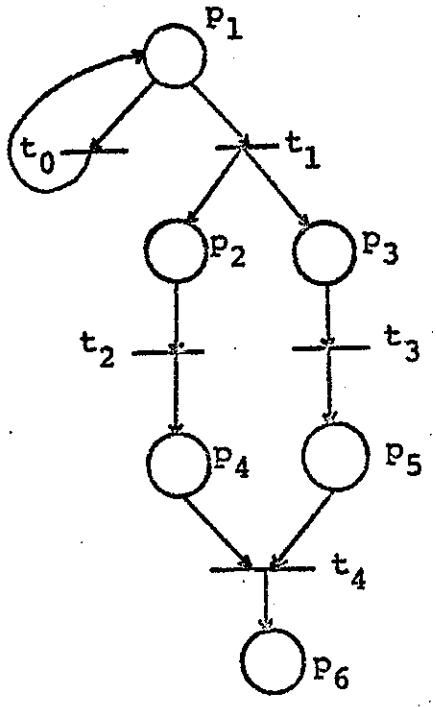


Fig.1 OPAS System Architecture.



(a) petri-net

MT

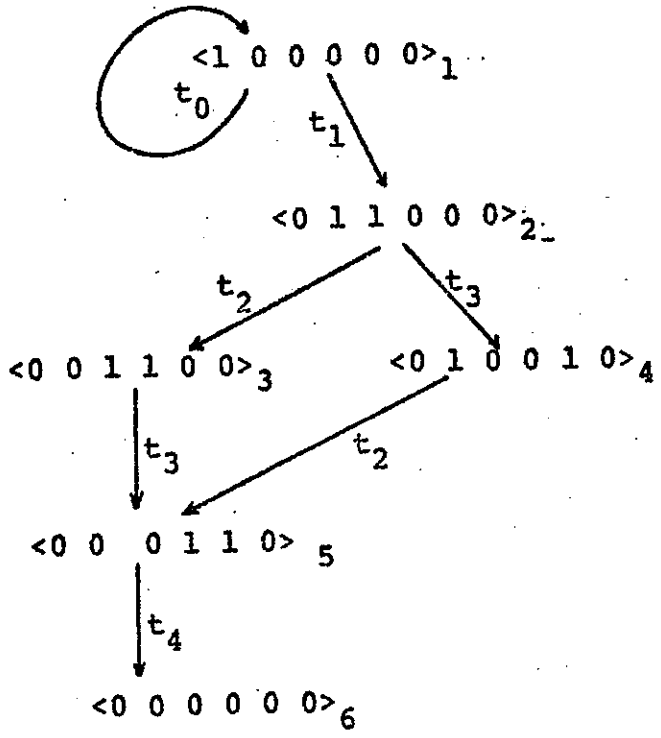
	MK_NAT	MK_FAT
1	2	1
2	2	3
3	1	5
4	1	6
5	1	7
6	0	0

terminating marking

ATL

	AT_TP	AT_OMK
1	t_0	1
2	t_1	2
3	t_2	3
4	t_3	4
5	t_3	5
6	t_2	5
7	t_4	6

(c) corresponding MT and ATL



(b) reachability tree

Fig.2 Petri-net driving data structures MT and ATL.