TR-82-02

一個整合性辦公室程序自動化系統

OPAS : An Integrated System for Office Procedure Automation

洪永常　劉寶鈞
龐台銘　柯志昇
郭德盛　張進福
何鈺威　何正信
周敏祥　張克正
呂永和

中華民國七十一年八月

摘　要

本報告描述一個實現辦公室作業程序自動化執行的系統。這個系統藉着一個派翠網路（Petri Net）模式將辦公室內部的處理作業模塑爲各種明確定義的處理程序，這些程序在以「辦公室程序描述語言」描述並經轉譯之後存入系統之內，然後即可隨時被啟動執行。整個系統是由數個VAX—11／780計算機上的軟體處理構成，它們彼此間以訊息相互溝通而形成一辦公室作業處理的整體環境。本系統的設計集合了各種辦公室作業處理的重要功能，因此能夠對辦公人員提供一個單一的使用者介面。

ABSTRACT

In this report, a system is described which provides office procedure automation. The system is mainly based on an extention of Petri nets for modeling office work as a set of well-defined interactive procedures. The office procedures can be specified in an office procedure specification language, which are then translated and stored somewhere in the system. A translated procedure could be activated at any time and run. The system consists of several modules which are software processes on the DEC VAX-11/780 VMS operating system. They communicate with one another via messages to constitute a whole environment for office information processing. The design of this system integrates major office processing facilities and, thereby, has the important feature of providing office workers with a single interface.

C O N T E N T S

CHAPTER 1 INTRODUCTION

1.1 Motivation

Office automation is emerging as a major application area of computer technology [5]. Recently, there is an urgent need for the improvement of the office worker productivity in order to keep pace with the fast growing factory productivity [2] [16]. According to the January 1977 Economic Report of the President of the U.S.A., the average annual productivity increase of U.S.A. private bussiness firms is 3.3% during the past two decades, while the average productivity increase in industry is much greater - approximately 90% [16]. The rapidly decreasing cost of computer hardware and communication equipments had made it feasible to equip office workers with automated tools for productivity improvement. There are now many computer science researchers and data processing communities working actively on automating the information processing in office [5].

Many Office Information Systems (OIS) have been announced by computer and office machine manufacturers such as Wang, Xerox, IBM, Datapoint, DEC and Prime etc. [15]. These systems provide facilities for office activities such as word processing (including text editing and form editing), information filing and retrieving, data processing, electronic mailing, etc.. They aid the office workers with automatic tools for performing office activities. Although these activities can be individually automated, however, they are initiated and directed by the workers. Office workers may need to change working environment whenever a different task from the current one is concerned (e.g. from a word processor to a data processing minicomputor) [12]. In desiging automated office systems, the challenge is how to develope an integrated system which facilitates automatic initiation and execution control of well-defined

routine operations (or called office procedures), as well as providing a single, uniform user interface to clerical workers. The system can thus play an active role. The worker can then do more creative tasks.

1.2 Integrated Office Information Systems

Several office systems which provide integration or automation in the sense mentioned above have been reported in the literature. We will in this section give a brief survey on those systems which relate to the design of the OPAS system.

Xerox Palo Alto Research Center started a project to develope its early "1st generation" office information system in late 1976 [5]. The system, called Officetalk-zero, provides through a single interface facilities for editing, filing and intercommunicating. The designers of the Officetalk-zero believe that a new OIS should be based on data objects of single page forms and files of forms. Under this system, the office worker can manipulate electronic images of documents with a sophisticated graphic CRT terminal to fulfil his clerical work. He can select to examine incoming forms, to enter form fields (including to draw a singature), to file or route a form, to trace mailed forms, etc.. The entire Officetalk-zero environment is a distributed system consisting of multiple minicomputers interconnected by a communication network. One of the minicomputers serves as a file server while the others serve as work stations. The implementation had collected a number of existing systems, e.g., text editor, graphic packages, communication facility, file server and others to form a single integrated system. However, the effort had not aimed at the automation of office procedure. Any activity must be initiated by the user. It should be noted that the origional intention of the Officetalk-zero project was to study the procedural specification of office clerical work. But the designers had recognized that this problem is too hard to solve and turned to concentrate on user interfacing.

A famous representative of procedure automation systems is the SCOOP (System for Computerization Of Office Processing) system by Michael Zisman[3][6][10]. SCOOP aimed at the specification, representation, and automation of office procedures. It, based on Petri net[11] augumented by production rules, models the office as a system of asynchronous concurrent processes. In SCOOP's view, an office procedure corresponds to an augumented Petri net. It can be described by a nonprocedural specification language as a set of activities associated with documents and represented by some internal schema. When an office procedure is invoked, an instance of the corresponding Petri net schema is created, together with its context data structure, to keep track of the procedure in progress. An execution monitor takes change of procedure execution, which is driven by the internal representation of all active office procedures. The SCOOP procedure is quite interactive. The execution monitor reads in all user inputs and updates the appropriate procedure instances. This can cause new mail to be delivered or other actions to be started. On a lower level, a few special-purpose programs such as document generator, mailing agent and file server, etc. receive messages from the execution monitor and actually perform required office tasks. By considering the specification language, the internal representation, and the design of a prototype system using one unified model, Zisman's work had successfully shown the feasibility of office procedure automation. His notion will probably have great influence upon future office information systems.

Another system emphasizing integration as well as automation is the OFS (Office Form System) by D. Tsichritzis[12]. Tsichritzis believes that office automation should take place "evolutionarily" rather than "revolutionarily" and had chosen to mechanize the manual form system as a basis of his research. OFS stresses on form management aspects such as accountability, auditing and tracing of forms instead of fancy user form interface. Commands and activities has been provided to handle electronic

images of form with limited form features (e.g., no repeated fields, single page, etc.). Perhaps the most important contribution of the OFS system is the notion of integrating form processing and data management. That is, form instances can be managed as relation tuples with form fields being manipulated as data elements in a relation. (Another published work by Tsichritzis attempted to integrate database records and communication messages [13].) OFS also provides form oriented, condition/action based activity automation. The condition part specifies activity triggering conditions in terms of contents of form fields, arivals of form instances at specific form trays, complementarity existence of a set of forms or form instances, or activity termination, time specification. The action part may include activities like receiving forms, sending forms and running a particular program. Tsichritzis also claimed that much of office activity or procedure is very ad hoc that a general, encompassing office procedure specification tool is not so easy to achieve and may be too complicated in order to be powerful. That is why he had chosen to deal with only well-defined routine work associated with form at current stage.

A third system providing integration and automation of office tasks is the OBE (Office Procedure by Examples) extention from the well-known QBE (Query by Example) relational query language by M. Zloof [17]. OBE is an attempt to combine the facilities such as word processing, data processing, electronic mail, graphic, report writing and application development within a single interface. With the QBE database management system as the base component, users can easily extract data from the database and map it into the body of objects such as letters, forms, reports, charts, etc.. Objects can also be edited and sent to other work stations via a communication subsystem. OBE allows the user to specify trigger expressions on a set of objects manipulation activities to form automated office procedures. The types of trigger are the modification trigger

concerning with database updating and the time trigger. With the example elements concept and the two-dimentional tabulating programming facility of the QBE language, OBE has provided a solution to the problems of DBMS interfacing and user interfacing of OIS design.

The four systems mentioned above as existing examples reveal most of the major concerns of the design of tomorrow's integrated office information systems, i.e., the specification and representation of office procedures, the user interfacing, the DBMS interfacing, and the form (document) processing facilities.

1.3 The OPAS Approach

OPAS (Office Procedure Automation System) is the first step toward reality of an office automation project held at Academia Sinica [9]. The goals of this experimental system have been:

- to accomplish automation of office procedure (including specification and representation)
- to provide integration of form (document) processing, data processing, data management, and communication
- to be general and practical

An office procedure in OPAS is a well-programmed, structured set of coorperative activities and is represented by a Petri net model augumented with data flow (to be detailed in chapter two). Such a well-defined procedure can be activated by the will of a human administrator, by messages coming into the office, by arriving of specific time, or by other procedures in progress, and then executes independently with interactions to agents inside or outside the office for necessary human intervention. Each run of an office procedure is called an "instance" of that procedure type. The entire office can be viewed as containing a number of dependent or independent procedure instances executed in parellel.

The design of OPAS attempts to make an uniform view on data objects which include working data, relational tuples and attributes, forms, and form fields. All these data units can be manipulated interchangeably in an office procedure. A few categories of activities on these data objects have been identified which, we hope, can cover most needs of practical office operation.

Special attention had been given to database facility, form facility, and communication within office or between offices in the development of OPAS. This first stage of OPAS implementation is aimed at the control structure (including internal representation) to achieve automation and the basic internal schemes to achieve integration. The control structure has been completed. Schemes for working data processing, database manipulation, form processing, communication, and office security have been established. Although a portion of interfacing feature is rather primitive in the current stage, e.g., simple form editing, manual mailing station for inter-office communication, simplified relational query language, etc., the design has, however, an important feature of being able to be extended to a powerful system.

The goal to be general and practical is in general very difficult to achieve. The introduction of working data into office procedures and the identification of categories of office activities may reveal some efforts toward this goal. However, this is the most difficult problem of the research that requires time and energy to solve.

The OPAS is built up with experiences from those systems previously mentioned and some ideas of our own. It is by no means a complete system yet. (Some limitations will be addressed in the later.) But the efforts so far have indeed established a feasible skeleton for automating office procedures.

1.4 Organization of This Report

This report consists of six chapters. Besides this introductory chapter, chapter two presents an introduction to the PNB office procedure model on which the OPAS system is based [8] [9] An example of procedure modeling is given for illustration. Chapter three describes an office procedure specification language for OPAS. Various office activities and important language features are discussed. In chapter four we detail the design of the OPAS system, including its architecture and operation, implementation of component modules and various features, etc.. Some suggestions for further extension of the system are given in chapter five. Finally, a conclusion in made in chapter six.

CHAPTER 2 THE MODEL FOR OFFICE PROCEDURE

To show the aspect of OPAS procedure, in this chapter we describe the PNB (Petri Net Based) office procedure model for the OPAS system. An example of procedure modeling will be given for illustration.

2.1 The PNB Model

The PNB model uses the Petri net incorporated with data graph to model office work as a set of predefined sequences of coorperative activities interacting with processing units (e.g. DBMS, work station, etc.) in the office [8]. Each well-defined office procedure corresponds to a Petri net in such a manner that the actions in the procedure are associated with transitions of the net. The firing of transitions, therefore, means the execution of associated actions.

Besides the enabling condition of the origional Petri net definition, "firing predicate" can be appended to transitions in the PNB model. A transition can fire only when it is enabled and its firing predicate has a "true" value. In evaluating the firing predicate (a Bodean expression), data may be need from the office databases, the work stations, etc.. When some activity associated with a transition gets executed, data from these sourses may be accessed, too. Figure 2-1 dipicts the description power for data flow of the PNB model, where the dashed lines denote data references. See [8] for a complete description about how the PNB model is applied on modeling various aspects of office operations. Appendix A gives a formal definition of the PNB model and its execution.

2.2 A Modeling Example

As an example of office procedure modeling, we will use the

PNB model to express the journal editing procedure in an editorial office. This procedure has been chosen for displaying a few important features of office work. For simplicity, the editing procedure described below had been tailored to have only one reviewer for each paper to be reviewed.

The PNB model of the journal editing procedure is shown in Figure 2-2.(The associated data graph is omitted here for clearity.) This procedure will be activated each time the editorial office receives a paper. The Petri net shown starts operation with a token being put in place p1. Since the t1 transition enabled now has no firing predicate, it fires immediately. The resulted action is to log the information about the paper (e.g., paper number, title, auther, submission date, etc.) into the file PAPERS. The completion of t1's firing causes two token to appear in place p2 and p3, respectively. This will enable, at the same time, two concurrent activities: one is to send an acknowledge letter (form F2) to the author (transition t2) and the other to prompt a form F3 to the editor for selecting a reviewer (transition t3). After both the two activities have been processed, the parrallelism will get synchronized in order to enable the transition t4. t4 is then enabled and fired to cause the insertion of a reviewing record into the file REVIEW (containing paper number, reviewer name, status of the reviewing, etc.). Next transition t5 fires to send a letter (form F4) and a copy of the submitted paper to the selected reviewer. (The reviewer's address may be found in another file REVIEWER containing data of every qualified reviewers.)

At this point, the editing procedure will pause to wait for the response from the reviewer, with a token appearing in place p7. However, a time-out control should be established at this time in order to make sure the reviewing be finished within limited time period.

Assume that the editorial office will inform the reviewer after waiting for two months and reselect another reviewer if the reviewer still does not send his review result back to the editorial office after another 5 days. Transitions t7, t8 and t9 are added in the net for such control. The token in place p7 enables transitions t6 and t7 in a "conflict" manner---that is, the firing of one will disable the other. In other words, transitions t6 and t7 are now competing for the token in place p7. Which one of these two transitions will fire and grab the token depends on their associated firing predicates. In the net, the predicate of transition t6 is the arrival of the message F4* (reviewer's response) while that of transition t7 is the elapsing of two months since it is enabled. (Note that both transitions have no actions at all.) If the arrival of F4* occurs within two months, the firing of t6 simply enables editing activities below p9. If, however, transition t7 fires first, token will be generated in places p8 to enable t8 and t9 which are still in conflict. Since the firing predicate of t9 is the elapsing of 5 days, transition t8 which is a duplicate of t6 is again timed for another 5 days. If this time limit is once more exceeded, the firing of t9 will disable editing activities below t6 and cause another reviewer to be selected by transmitting a token into place p3. Then the same paper reviewing process will restart.

If the form F4* comes within the specified time limit, one of the two transitions t6 and t8 will fire and cause another pair of conflict transitions t10 and t11 in order to check the response from the reviewer. If the reviewer says he does not want to review the paper, transition t10 will fire to reselect the reviewer. Otherwise, the editing process continues on transition t11. The remaining activities include a final decision for accepting or rejecting the paper and other housekeeping work as dipicted in the figure. The entire procedure terminates when a token is placed

in the place p12.

In this example, we see that the PNB model has successfully expressed a few important natures of office operation, such as concurrency, synchronization, decision making and timing control process, etc.. Detail implementation of the journal editing procedure in terms of OPAS's procedure specification language will be discussed in the next chapter.

CHAPTER 3 AN OFFICE PROCEDURE SPECIFICATION LANGUAGE

Office procedure specification language serves as the single interface between the OIS system facilities and the users. At this stage, the OPAS system provides a rather primitive nonprocedural language for office procedure specification. Once a user has designed an office procedure (i.e., the Petri net has been sketched), he can use this language to write down a "specification program" which is the representation of that procedure for OPAS. This program (procedure) is then translated to OPAS's internal representation (to be detailed in the next chapter), which is ready to be invoked and run.

3.1 Some Design Considerations

There are many aspects and issues about the design of office programming languages, such as the abstraction level, the computational completeness, the compiler consistency checking, the application dependency, etc. [5]. The design of our specification language, due to time and energy, is by no means a touch or look into these problems. It is mainly designed to provide a basic tool for office procedure specification as well as a description of the OPAS features.

Basic requirements for the design of an office procedure specification language include:

- to closely match the office model
- to reveal the user environment of the system
- to provide convenient interfaces to system facilities such as form processing, office database management, etc.
- to be high-level and abstract for office workers

The design of this language attempted to satisfy all of the above requirements. However, the last two goals seem not yet achieved very well. In the following sections, a few important language features are first introduced. Layout of the language structure and syntax then follows. An example is finally given to illustrate the use of this office procedure specification language.

3.2 A Glance at Language Features and Concepts

Several features and concepts of the specification language are presented and discussed in this section.

3.2.1 Procedure Indentification

Each office procedure in OPAS is associated with an eight-character procedure name assigned by the office programmer. This name serves as the procedure ID. However, an office procedure may be invoked in such a way that more than one instance of it is spawned and runs in the system at the same time. An ID, called the procedure effective ID, will be assigned by OPAS to each procedure instance which is formed by concatenating its procedure ID and activation time. This procedure effective ID is used to identify the corresponding procedure instance throughout the system.

3.2.2 Specification of Agents

The human agents, as other processing units of a procedure, should not be fixed for every procedures in the office. For example, worker A may serve as the order receiptionist while worker B is working as the order administrator in an order processing business procedure. However, both workers A and B may also be working in another office procedure for another position . One philosiphy of OPAS is that the total number of agents (work sta-

tions) within an office is fixed, but office procedures may partition them into different task groups in an overlapping manner. Thus, any procedure may flexibly "allocates" any workers belonging to the office as its agents. In the specification language, office agents are named AGENT1, AGENT2, ... , AGENTn. (The number of agents is determined by the OPAS system generation.) A procedure specification will select its processing agents from this set and may assign mnemonics to them.

3.2.3 Invocation Parameters

Many procedures are invoked by parameters. In an order processing procedure, for example, the parameters may be the customer ID, goods and quantity requested, etc.. These parameters are called initial or invocation parameters, whcih are set up during the invocation process. Each office procedure is allowed to have data objects as invocation parameters. They are declared in the specification program. An "invocation parameter entry form" must also be defined for office procedures having invocation parameters. When a procedure is invoked manually, this form will be prompted on the termianl asking for entering invocation parameters. If the invoker is another active procedure instance, the passing of parameters will be accomplished by OPAS.

3.2.4 Data Objects

Data object in an OPAS procedure is defined as the data unit for office activities (to be detailed later). There are three kinds of objects in the specification language: namely, working objects, database objects and form objects.

Working objects (single or grouped) are temporary, scratch data. They are needed in lengthy computations, or can serve as

saving buffers for database and form objects. Database objects are data (fields, tuples and relations) in the office database. Form objects include forms and their fields. Working data items, fields in tuples or forms have the same semantics in the implementation. So they can be manipulated interchangeably in the specification program.

All kinds of objects fall into two data types, alphabetic and numeric. Objects are internally stored as character strings. However, numeric objects are computational, but alphabetic objects are not.

3.2.5 Declaration of Objects

Data objects can be defined or declared in a specification program as in traditional programming languages. A working object must be defined in terms of its data type, length and format. Although it is possible to process database objects in the specification program without any declarations, for simplicity the current design of OPAS restricts that record buffer (group working item) revealing relational tuple format be specified for manipulating fields in the tuple. The declaration of form objects will be discussed below.

3.2.6 Form Definition and Form Features

A form in OPAS mirrors the paper form in the manual office in that they both consist of two parts: the predefined information and fields to be filled. A form definition therefore must contain the layout of the "form blank" [12] (i.e., a form with empty fields) plus specification of field characteristics such as data type, format, etc.. In a specification program, the programmer defines a form by giving its name and dimension in character position, followed by drawing the form blank between two start/end markers with field

positions being indicated by corresponding field name (prefixed with a special character % to be distinguishable from predefined form text). The field name must not be longer than the field length. The form blank drawn in this way is then argumented with field descriptions, including field type, data type and field format, for every field names appearing in it. Forms are restricted to be single page and cannot contain repeated fields or group fields [7] [12] currently.

The data type and format of a form field have the same semantics as working objects. As for the field type, three possible types are provided, namely protected, unprotected, and identification. Protected fields are output fields in a form. They are filled when a form is generated and can not be altered when the form is sent to some work station for interaction. Unprotected field are those fields to be entered by human agents when the form is displayed on the terminal. Identification fields are special unprotected fields used to identify the authority of the human operator filling the form. Such fields can be included in any form to provide a basic password security control for office procedure execution. Identification fields will be typed with no echo.

A form can be sent to either the work stations or the mail station for interaction. In a specfication program, the user may specify more than one "send operataon" on a form with different destinations. That is, forms can be passed among work stations. A "field masking" facility is also provided which allows fields unauthorized for access to be marked out (see later discussions). Note that in the current design, the multi-instance concept of form processing is not yet covered.

3.2.7 Basic Activities

Referring to facilities of existing systems [1] [16] [17]

as well as practical considerations, we have identified the following operations required for office processing:

.computation
.form processing
.file management
.authentication
.procedure instantiation

In the specification language, four categories of basic activities are designed to accomplish these office operations. They are the ASSIGNTO activity, the INVOKE activity, the SENDTO activity and the database operations, as described below.

3.2.7.1 ASSIGNTO Activity

The ASSIGNTO activity performs computation on data objects. Its syntax is

ASSIGNTO <object> <expression>

which means to assign the value of the numeric or literal expression <expression> to an object <object> .

The ASSIGNTO activity mirrors the assignment statement in traditional programming languages. One important use of it is to transfer object values, for example, from fields in a relational record to form fields. The creation of a form could be viewed as filling the protected fields via ASSIGNTO operations.

3.2.7.2 INVOKE Activity

The basic activity INVOKE is used to invoke office procedures. The syntax is

INVOKE <proc-name> WITH <obj-list>

Where <proc-name> is the name of the procedure to be invoked and <obj-list> is a list of objects to be used as invocation parameters. Once the INVOKE activity is executed, the specified office procedure is activated and initialized with the invocation parameters.

3.2.7.3 SENDTO Activity

The SENDTO activity is designed to provide form interaction between the office procedure and its agents. The syntax is

SENDTO <dest> <form> WITH <field-list>

Which means to send the form <form> to the destination <dest> for interaction. <field-list> specifies fields in the form involved in the interaction.

The destination in a SENDTO activity can be either a work station or the mail station. When the destination is some work station, the form image is displayed on that terminal and the issuing procedure instance will wait for the worker to fill the unprotected and identification fields if any are specified in the field list. After these fields are filled, the SENDTO activity is said to be completed (and the issuing procedure then obtains the values of these fields). If the mail station is the destination, a hardcopy of the form image will be produced and sent out of the office via manual mailing system. When the hardcopy is filled and returned, a data entry process (actually, an interaction on a special work station) is activated to put the returned message back to the system. The issuing procedure instance does not pause at the current transition to wait for the message return. Such interoffice interaction is synchronized by specifying an EXISTDOC predicate at another transition in

the Petri net. See section 3.2.8 for details.

One important feature of form interaction provided in the OPAS system is the field masking facility. The field list in a SENDTO operation contains only those fields authorized for interaction. That is, different SENDTO actions on the same form may have distinct fields in the field list. This feature was desinged for exercising security control over the office tasks.

The identification field in the form is another consideration toward security control. The user can specify an identification field for each interaction to make sure that the key-in will be done by the authorized people. In the current design, the form interface program will handle the checking of the authorized workers by not accepting the key-in until the identification field is entered correctly. Of course, the security control can be extended to a rather sophisticated checking process in the future.

Before the SENDTO interaction, protected fields of form involved should be filled using the ASSIGNTO activity. After the SENDTO operation is finished, unprotected fields specified contain input values which can be, for example, copied to other data objects by ASSIGNTO activities. Such assignment operations can be implicitly done by specifying data sources or sinks for fields in the field list. For example, the activity

SENDTO AGENT1 F1 WITH FD1=D1, FD2, D3=FD3

transmits the form F1 to the AGENT1 work station with three fields FD1, FD2, FD3 involved where FD1 and FD3 have source object D1 and sink object D3 respectively. If the field FD2 is unprotected, after the interaction it is viewed as an object containing some

value. If otherwise it is protected, then it should have been set by an ASSIGNTO (or another SENDTO) activity previously.

3.2.7.4 Database Operations

The database operations are commands for manipulating data in the office databases. They provide the important user-DBMS interface of the specification language.

Three commands are currently available for retrieving and modifying data in the office databases. They are:

SELECT <rel> WHERE <qualifier> { GIVING <rel> / INTO <group-obj> }

<rel> UNION <group-obj> GIVING <rel>

<rel> MINUS <group-obj> GIVING <rel>

The SELECT operation performs retrieving of data in relation <rel> which satisfy the qualification <qualifier> . The result itself is also a relation (GIVING <rel>) or a group object in the procedure (INTO <group-obj>). The UNION operation inserts a relational tuple or record in <group-obj> into a relation, while the MINUS operation deletes the tuple from the relation. The introduction of record buffers in the above database operations inherits traditional data processing concepts. Such user model may be criticized to be not abstract or high-level enough for non-specialists such as clerical workers. A possible database interfacing design emphasizing on data abstraction will be discussed in chapter six.

3.2.8 Transition Firing Predicates

The specification language must provide means for describing the firing predicate with each transition in an office procedure.

Three categories of firing conditions were identified form the operation nature of the OPAS office model. They are the object predicates the time predicates, and the document existence predicate.

The object predicates check the value of objects. The syntax is

<obj1> <optr> <obj2>

Where the two operands <obj1> and <obj2> are data objects in the procedure and <optr> can be one of the six relational operators >, =, <, >= (greater than or equal to), <= (less than or equal to), and ≠ (not equal to). These comparison operations facilitate the data dependent flow control within office procedures.

The time prodicates specify time constraints for transition firing. Two kinds of time specification, namely,

UNTIL <date> <time>

and

WAITFOR <days> : <hours>

are provided in the language. The UNTIL predicate specifies the absolute date and/or time when the associated transition can fire. The WAITFOR predicate, on the other hand, provides relative time scheduling counted from the time the associated transition is enabled.

The document existence predicate was designed for interoffice form interaction. When a form (with unprotected fields) is sent

to the mail station, the receiving of the expected return information must be synchronized in another transition by specifying the predicate

EXISTDOC <form>

where <form> is the name of the form in question. Before the inquired form returns, the predicate will always evaluate a false value, The associated transition will, therefore, never fires until the form is sent back.

Predicates in the above three categories are called atomic predicates. Atomic predicates can be combined by Boolean operators AND, OR, and NOT to form "predicate expressions". A predicate expression serves as the general firing predicate for transitions. At any time, the evaluation of a predicate expression with a true value will trigger the firing of the associated transition.

3.3 The Structure of The Language

Our nonprocedural office specification language contains three sections: namely, the procedure identification section, the objects definition section, and the procedure detail section. A brief description about the individual sections is presented below. See appendix B for the entire syntax of this language.

3.3.1 Procedure Identification Section

The office programmer gives the following information in the procedure identification section: the name of the procedure, the invocation parameter list, and the agents specification. There is no significant restriction on the naming of an office

procedure, but only the first eight characters of the name will be taken as the procedure identification. The invocation parameter list specifies working objects or form fields (protected fields only) as invocation parameters of the procedure. The agents specification declares office agents involved in the procedure. Mnemonics can be assigned to each agent.

3.3.2 Objects Definition Section

All the working objects and forms appearing in the procedure detail section must be defined in the objects definition section. The associated information had already been given in section 3.2.5 and 3.2.6.

3.3.3 Procedure Detail Section

The procedure detail section is the major part of the procedure specification. It contains descriptions for the overall Petri net structure of the procedure as well as the activity details in each transition.

This section starts with the initial and termianting marking specification, followed by a number of transition specification for all transitions in the procedure. A transition is specified by stating its transition ID, input and output places, predicate expression, and the action associated. The predicate or the action part of a transition can be null. The action part, if specified, consists of a list of basic activities which will be executed one after another if the corresponding transition fires.

3.4 An Example Office Procedure Specification Program

Appendix C gives an example specification program written

in our office procedure specification language for the journal editing procedure discussed in section 2.2. Note how the various tasks in figure 2-2 are implemented in this program by record buffers (group objects), forms, and programmed actions in transitions.

CHAPTER 4 DESIGN OF THE OPAS SYSTEM

One major goal of this report is to develope a framework which realizes automation of office procedure. The resulted OPAS prototype consists of several software processes communicating with one another as well as work stations in office. The following sections describe the architecture and implementation details of this prototype system.

4.1 Overall System Architecture

The overall architecture of the OPAS system is shown in Figure 4-1. The environment includes an administrator console, a number of agent stations, a mail station, the office database, and the internal form file. The administrator console is the control center of the system. It functions to monitor the operation of the entire system. System administrator at the console interacts with OPAS for demanding and controlling procedure execution, as well as inquiring system status. Agent stations are work stations in the system. Procedure instances communicate with agent stations for human intervention. The mail station provides a channel for interoffice communication. Incoming messages enter the system through the CRT terminal of it, while the printer prints outgoing mails.

The OPAS system consists of five modules: supervisor, (Petri net) monitor, 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 [4] . Interactions among processes via messages are also shown in Figure 4-1. (Appendix D gives format of these messages.) The five modules 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.

The supervisor accepts requests for services from the administrator console, the mail manager, and the monitor (that is, from another office procedure). It creates a monitor process once a procedure invocation request is entered into the system, and passes to the monitor the name of the procedure and required parameters for initiation. A monitor process, once created, will be driven by the internal form of the invoked procedure. Since the monitor is written in reentrant code, there is in fact only one single copy residing in the memory. The form manager deals with interaction between agent station and system. It receives commands from the monitor, displays forms or memos at station, and passes worker-supplied inputs back to the monitor . The data manager deals with queries which manipulate data in the office database. It acts as a high-level interface between monitor and VAX-11 file system. The mail manager governs communication between monitor process and outside office. It interacts with the mail station to process incoming and outgoing mails.

4.2 The OPAS Modules

In this section we present a detail functional description for the five OPAS modules. Appendix H gives detail implementation of them.

4.2.1 Supervisor

The supervisor controls the whole system, including:

.Create all other processes,
.Accept procedure invacation demands from the administrator console, mail manager, or monitor,
.Interact 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 sent from the mail manager or monitor processes (i.e., other office procedure). The invocation includes creating a monitor process and putting the procedure name (through which the monitor can obtain its internal form) and the initiation parameters to supervisor's mailbox which can then be accessed by this created monitor process. 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's 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 messages (MSG_MER) of procedure instances will be sent directly to the supervisor, which then reports to the administrator and records them in the OCT table and office log file.

4.2.2 Monitor

A monitor process is created if a request for service is submitted to the supervisor. It is then 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.

The monitor's work to realize procedure execution can be devided

into three parts, namely, driving the Petri net, evaluating predicate expression, and performing basic activities. Most of these are done internally in the monitor process itself (by accessing the procedure internal form) except for the three basic activities SENDTO, INVOKE, and database operation which require service from the other OPAS modules. Intercommunication messages MSG_FP, MSG_PI and MSG DBO are sent to form manager (or mail manager), supervisor and data manager respectively from monitor processes for accomplishing these operations. Results are sent back to the requesting monitor via other messages (e.g., MSG_FFR and MSG_DBR).

In the current stage, we do not emphasize on supervisor intervention to the monitor process. Since the monitor is implemented as a subprocess of the supervisor, however, the supervisor can easily get full control over the monitor using the process control facilities provided by VMS [4].

4.2.3 Form Manager

Form manager deals with interactions between procedure instance (i.e., monitor process) and agent station. The interaction medium is form . If a monitor process needs a "form I/O", a 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 station. Corresponding to each agent station is a request queue which holds all the MSG_FP requests using that station 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.

4.2.4 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 and returned, the operator at the mail station 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 the requesting procedure.

Invoking procedures from outside offices is another major function. In this case, this is done by sending the MSG_PI messages (which includes the required information for procedure invocation) to the supervisor.

This mail process uses a CRT termianl plus a printer for communicating with outside offices. 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.

4.2.5 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. 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 relational algebra commands (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.

4.3 Internal Form of Office Procedure

By internal form we mean the internal representation of office procedure specification which is used to drive the monitor. This driving leads to the execution of office procedures. The internal form actually is a set of data structures providing necessary 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 Component List (PCL), Activity List (AL), Actural Argument List (AAL), Objects Description Table (ODT) ,and Objects Data Area (ODA).

The following subsections describe these data structures. Layouts of individual tables are given in appendix E.

4.3.1 Procedure Control Block

The procedure control block (PCB) provides information for controlling office procedure instances. It contains an instance's procedure identification (PRC_ID), activation time (PRC_AT), execution status (PRC_STA), Petri net pointers (PRC_CMK, PRC_CTR), and invocation parameter descriptor (PRC_IPL). The fields PRC_ID and PRC_AT together form the effective ID of the procedure instance which serves the identification of the corresponding instance throughout the system. The status PRC_STA must reflect one of the states (e.g., ready-to-go, running, iddle, etc.) during execution. Petri net pointers PRC_CMK and PRC_CTR record the current execution point of the corresponding instance. PRC_IPL is a list of pointers to objects in ODT which have been specified as invocation parameters.

There is no usual priority information in PCB. Since the priority control is actually done by the supervisor by means of adjusting VMS process priority of procedure instances, such information is kept in the OCT table inside the supervisor.

4.3.2 Petri Net Driving Date Structures

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

The specification language translator uses the token machine concept to derive the reachability tree[11]consisting of every possible Petri net markings for an office procedure. These markings constitute the MT and the ATL. All active transitions in a marking are stored as consecutive entries of the ATL. Each MT entry contains only a pointer (MK_FAT) and a length (MK_NAT) field for accessing these transitions. The first entry in MT is always the initial marking, while the terminating marking is indicated by null MK_NAT and MK_FAT fields.

An ATL entry consists of an output marking field (AT_OMK) and

a pointer (AT_TP) to the detail information of transition. The output marking is defined as the resulted marking due to firing of a transition. Since each transition in a net could be active in more than one marking, the context of transitions are stored in another table with the AT_TP pointer in ATL for access.

Figure 4-2 gives an illustration of the Petri net driving data structure where (a) and (b) are a simple Petri net and the reachability tree derived respectively, while (c) shows the contents of the corresponding MT and ATL.

4.3.3 Activity Realization Data Structure

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

Each TDT entry (plus related entries in PEL, AL etc.) describes the context of a Petri net trnasition. It also contains some control items (e.g.,the in-firing flag TR_FIRE) for transition selection and firing. This internal form must include the firing predicate and activities associated with a transition. Since both two parts are variable-length items, they are seperately stored in the PEL and AL data structures, respectively. Pointers and length counts (fields TR_PEP, TR_NAC, TR_FAC) are kept in TDT for access purpose. Another counter (TR_CAC) serves to count activities performed during transition firing.

PEL and PCL are data structures for storing firing predicates of transitions in the net. All atomic predicates (see section 3.2.8) are listed in PCL, while PEL contains predicate expressions with entries in PCL as operands. The expression in PEL are stored in reverse Polish notation.

AT and AAL describe activities in each transition. Each type of basic activities is represented by an operation code followed by a list of arguments. (See appendix F for formats of activity internal form.) As in the above data structures, the variable-length argument list is stored in a separate table AAL with pointer and length count in AT (AC_FAG, AC_NAG) for access purpose.

ODT and ODA together provide the facility to manipulate objects in an office procedure. The descriptor table ODT describes the object type, data type, length, storage address in ODA, and other characteristics of each defined object. The ODA, on the other hand, provides character storage for all objects. Objects of either alphabetic or numeric data type are stored as character string in ODA. (Appendix G gives a tabular description of characteristics and coding of objects in ODT.)

Figure 4-3 depicts the organization of these activity realization data structures.

4.4 Some Notes on Implementation

4.4.1 Petri Net Driving

The Petri net structure of an office procedure consists of the marking table and the active transition list in the internal form. The monitor drives the net by tracing entries in these two tables.

Each MT entry has its active transitions listed in ATL. When the net is in some marking, its active transitions will be checked sequentially for their associated predicates (in PEL) to determine which transition can fire. The first active transition with a true predicate value is selected to fire. The

firing of a transition will cause its associated actions to be performed. After the firing is completed, the Petri net will be in the state of the output marking of that fired transition. The selection of a transition from this new marking will repeat until the terminating marking is reached.

The evaluation of predicate expression of an enabled transition may invole objects comparison, form checking, and time checking. Comparison of objects is a simple operation on objects stored in the ODT and ODA. Form checking operation is done by examining a flag field in ODT entry of the form. This flag is set when the form is sent to mail station and cleared when the form is returned. The processing of time predicate is somewhat complicated. When a time predicate is encountered for the first time, the predicate is viewed as false but a VMS set-timer call [4] is issued with the time specified. This system call will result in a software interrupt to be generated to the monitor process when the specified time arrives. Then, the corresponding interrupt handler inside the monitor will record this fact by setting a bit in the internal form of that original time predicate. After this, the time predicate is viewed to contain a true value. Of course, any set-timer call which is not yet on time must be cancelled [4] if the corresponding transition is disabled due to the firing of another transition.

4.4.2 Working Storage

Each OPAS procedure has working storage built up by ODT and ODA. This working storage is used for holding temporary results, form images, and relational tuples during procedure execution.

Data objects in the working storage are manipulated by the ASSIGNTO activity as well as other activities. Since all objects are stored as character string, they are converted to binary values before computation and the result will be converted back

afterwards for storing. In the operation of a procedure instance, it is likely that data exchange will happen frequently among objects in the working storage as well as among the working storage, the office databases, and the agent stations, just like what occurs in a traditional computer program.

4.4.3 Form Implementation

A two-dimensional form is cut, line by line, into a list of fields which are stored in ODT and ODA. These fields can then be manipulated seperately as objects in basic activities such like ASSIGNTO. In a SENDTO activity, included fields of form are grouped to make a "form packet" which is the transmission unit of form among OPAS modules. For example, the monitor may send a form packet (message MSG_FP) to the form manager for interaction. A form driver routine will take charge of constructing, displaying the form as well as accepting worker's key-in. Figure 4-4 shows the format of form packet. Each fields in it consists of two parts, namely a descriptor part and a text part. The descriptor describes the field in terms of its type, length, location, etc., while the text part gives the field value.

The form packet scheme may hopefully serves as a basis for further extentions of OPAS features, for example, the automatic communication between offices, multiple form instance concept, and management of form, etc..

4.4.4 Database Interaction

In the design of database facility, we chose to employ a stand-alone DBMS process (the data manager) for database management. Database operations are carried out by message interaction between the monitor and the data manager. On performing a database manipulating activity, the monitor sends to the data manager

a MSG_DBO request containing a query command string (generated from the internal form of the activity), and then waits for the results. The data manager, after having processed the query command, sends the message MSG_DBR containing resulted data or a status code to the issuing monitor process. The monitor will copy the resulted data (if any) into its working storage. The interaction is then completed, with data retrieved into memory or modification done in the database.

4.4.5 Implementation of Message-driven Modules

Three VMS system services make the implementation of the OPAS system successful. They are mailbox, 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. Since each module communicates in a one-to-many manner, any module cannot "wait" for messages from any other. The VMS AST facility enables each process to deliver an asynchronous mailbox reading and continue. Once the message is ready to read, the process is interrupted (asynchronously trapped) to execute a special program section for accepting (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 the reading of mails from the mail station and forms in the system.

4.4.6 System Generation and Bootstrapping

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 relations must be generated and loaded in the office database.
4. All the OPAS terminals must be assigned.

Facilities are provided to accomplish these tasks.

OPAS is bootstrapped by running the supervisor from the administrator console, which in turn automatically creates other OPAS modules (form manager, mail manager, and data manager) in such a way that the system is ready to accept requests for services from the administrator console and the mail station.

CHAPTER 5 SUGGESTIONS FOR FURTHER EXTENTION

During the development of this first version of the OPAS system, a few important features were identified but not implemented due to the time limit. Some of them are proposed in the following sections as a guideline for future extention.

5.1 Complete High-level DBMS Interface

One reason to select the relational approach as the office database model in the design of OPAS is that a high-level, set-handling data sublanguage can be defined easily for clerical workers. The database operations in the procedure specification language (see section 3.2) basically resemble the relational algebra approach. However, as mentioned before, the design is not yet complete in that the user must use formatted record buffers for accepting and manipulation relational tuples. In this section a design to accomplish set-handling for database data is sketched. In designing the DBMS interface for a procedure automation system, the major problem is that data buffers must exist in the procedure specification program but the user's interpretation of them should be high-level or abstract. The workspace concept in data sublanguage DSL ALPHA may satify our needs. Here workspace is interpreted as the communication area between office procedure and the DBMS which is capable of holding an entire relation. A procedure can have one or more workspaces. Take data retrieval as an example. A SELECT operation can be specified to pull the resulted relation into a workspace w:

SELECT customer WHERE credit < 500 INTO w;

The workspace w now is viewed as a rectangular array containing rows of tuples. Data in it can be referenced by adding prefix "w" to attribute names, e.g., w.credit. This notation refers to an aggregate object which is the set of values in the credit column

of the workspace w. For example, aggregate objects with prefix w in the activity

SENDTO MAILSTATION f4 WITH fd1=w.name, fd2=w.addr, fd3=w.credit

will cause multiple instances of form F4 to be mailed, each with different customer name, address and credit value.

A set of functions can be provided to manipulate aggregates in workspaces. Typical examples include MAX, MIN, SUM, AVG, CNT, etc. For example, SUM(w.order-amount) gives the sumation of dollar amount in ORDER relational tuples retrieved into workspace w, while CNT(w.*) counts the number of tuples. Other relational algebra operations such as JOIN, PROJECT, UNION, MINUS, etc. can also be applied to the workspace, treating the latter as containing tuples of a relation.

Implementation of the above requires that the monitor program loops internally to process the set of values in the workspace. Considering the design of the OPAS system, the workspace can be a dynamic area in the monitor data structure ODA or they can be a scratch file in disk. The data manager, after having processed a request (e.g., a SELECT operation), should transmit both the resulted block of tuples and a tuple description to the monitor mailbox. The monitor will store both these two messages and associate them with the internal representation of the corresponding workspace. Then, whenever a reference of data in the workspace is encountered, the monitor process starts a loop to process the data in aid of the associated tuple descriptor. Processing in the loop may be the distribution of multiple form instances, the sumation or averaging of attribute values,etc., as illustrated above. In fact, internal implementation of basic activities discussed in chapter 4 must be changed into loops to realize the workspace processing.

5.2 Additional Form Features and Form Processing

Current implementation of the OPAS system had narrowed the usage of forms to be the interaction medium between system and station. Significant restrictions include no repeating form fields, single page (or display screen) form size, and no multiple form instances. With these limitations, form processing capabilities in manual paper offices are by no means covered yet in OPAS.

Repeating groups in form are necessary for users. There are obviously repeating entries in bussiness forms, for example, items in invoices. To adopt repeating fields, the structure of the form packet in our implementation must be refined. Additional design must also be consistenly applied to form description, display and editing functions, etc.. The form size problem also follows natually. It seems not proper to allow repeating entries but reject the existence of multiple form pages. Moreover, there is not any storage problem with form since it exists as electronic image. Thus the handling of multiple form pages should also be included in the above refinement.

In ordinary offices, official documents are usually filed for future reference after they are produced and processed. For example, if forms are invoices we may want to ask queries about monthly sales. This need suggests that form field values be managed as data. A simple, yet sufficient realization is to treat form instances as relational tuples, with form field values considered as attribute values [8]. The form packets in OPAS are well ready for such handling. Form relations can be defined with literal fields and descriptor data in a form packet treated as special FILLER attributes which cannot be accessed individually. Then form instances can be stored in the office database and retrieved later by database activities. For example, the following activities

```
SELECT   order   WHERE   cust-name = 'ABC CO.'   INTO   w ;
SENDTO   AGENT3   w.* ;
```

result a tracing of ABC CO.'s orders at the AGENT3 station.

Another problem which had been neglected in current implementation is the handling of multiple instnaces of the same form type in an office procedure. Basically, form instances of the same type can be chained and stored in a dynamic area in ODA. However, aspects of processing such as identification of individual instances, handling of form aggregates, etc., do require further study and design.

5.3 Abstractions in The Specification Language

Abstraction on data structures as well as on control structures should be emphasized on the design consideration of office procedure specification languages [5]. One language feature which may assist in providing both data and control abstractions is the aggregate-oriented processing of data objects.

Many office operations are frequently performed on groups of documents [7]. For example, in journal editing procedure more than one reviewers would be assigned to review a paper and thus multiple copies of reviewing documents must be processed within the procedure. Other examples are daily memo destribution, trace of filed forms, etc.. If objects can be manipulated in aggregates, the user specification can be condenced and low-level control structures removed. Consider the following example. Assume in the journal editing procrdure the number of reviewers for a paper is variably determined by the chief editor. The activities (suppose "rev(*)" means all occurrences of the repeated reviewer field in the reviewer-selection form)

```
SELECT   reviewer   WHERE   name = rev(*)   INTO   w ;
SENDTO   MAILSTATION   letter-to-reviewer   WITH
  faddr = w.addr, fname = w.name, ftitle = paper-title,
  fauthor = paper-author ;
```

can be specified to conveniently prepare and mail reviewing forms to each reviewer selected. A contrast is that the user must employ explicit loop control (using transition predicates and counter ob-

jects) to implement each activity above, which is certainly much more low-level and boring. In specifying such aggregate-oriented processing, the user needs not concern with the number of duplicates in the aggregates. In fact, it is dynamic as in the example above. Repeated operations will be initiated according to the run-time context of the aggregate objects.

Abstract object types may also be provided instead of atomic alphabetic/numeric classification. Such high-level data types may include NAME, ADDRESS, DATE, DOLLARS, etc.. They will help in offering better error detection and automatic conversion in object manipulation.

5.4 Replacement of Passive Work Station

Another limitation of the OPAS system may arise from the automatic control of office processing. The system design took into consideration only various aspects for realizing precedure automation and nothing else. Work stations in the system turn out to be in a passive position in that interaction between work station and procedure instance are totally initiated by the system. Workers at the work stations can do nothing but sit and wait for the calling from some procedure instance. Such system operation is certainly not complete enough. If, besides interacting with procedure instances, the workers can access through the work station system facilities such as information retrieval, electronic mail, etc., they can work in an active manner and take charge of those office tasks which are difficult to be organized into well-defined office procedures. The OPAS system will then be more practical and complete.

A feasible improvement is to replace simple terminals in OPAS with stand-alone, microprocessor based word processing system. These word processors should be link to the VAX minicomputer in such a way that the office database, OPAS's utility programs(e.g.,

the specification language translator) can be accessed from the word processor stations and OPAS itself can interact with these stations. Then we can construct an environment in which clerical workers work in aid of automated tools to carry out creative or advanced office tasks while routine office procedures automatically run in parallel.

The extention to such a powerful system may encounter many interfacing and integration problems. One approach to simplify the complexity is to view the office procedure instances automatically running as message (or form) generating agents. An interface program may reside at each work station, taking charge of communicating with these message generation agents and other stations. The worker at a work station changes operation modes to do word processing, to issue data retrieving queries, or to process incoming messages or forms, etc.. In the case of messages processing, forms sent from automatic procedure instances can be queued on a mail tray and selectively examined or filled, exactly like messages from other work stations.

CHAPTER 6 CONCLUSION

As addressed in [2], automatic work flow control is a key to the improvement of office productivity. In this report, we have presented a system which supports automatic execution control of office procedures.

The design of the OPAS system attempts to achieve procedure-level automation of office work as well as integration of office information processing facilities. To accomplish automation, control strcture has been established by incorperating the supervisor module which governs system operation, the moniter module which realizes office processing, and the various server modules form manager, data manager and mail manager. Execution of an office procedure is then implemented by spawning a computer process (the monitor) which runs by itself to fulfil the required task. The integration is achieved by providing form processing, database management, and data processing facilities through the single office procedure specification language interface. Schemes to support these processings, e.g., the form packet, the working storage and DBMS interfacing, had also been included in the design of the moniter and other related modules. The entire system is being implimented on a VAX-11/780 computer under the VMS operating system. The prototype system described in this report has been completed by the July of this year. Continuing efforts are under way on advanced topics such as interoffice communication, security control, form management, and database facility, etc..

Appendix A. Formal Definition of The PNB Model

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 Model Execution Definition

The execution rule of a PNB diagram can be defined by a doublet $\Gamma = (M, \tau)$ 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 markings. 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_o. Of course, m_f, the terminating marking belongs to the set, i.e., $m_f \in M$;

(iv) τ: $M \times T \rightarrow M$ is a "firable function" of transition 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) $m \geq f_t$ and

b) c_t = TRUE in (c_t, a_t).

A transition is enabled if only a) holds.

Appendix B. Syntax of Office Procedure Specification Language

Notations:

<ps>	start symbol of production rules
<--->	nonterminals
capital words	terminals (key words)
a|b	either a or b
[ ]	optional
{ }	multiple occurrences

```
<ps> ::= <id-section>  <data-section>  <procedure-section>
         END
<id-section  ::= PROCEDURE IDENTIFICATION PROCEDURE-NAME[IS]
                 <symbol> ; [PARAMETERS [ARE] <symbol-list>;]
                 {<system-device> [IS] <symbol> ;}
<system-device> ::= AGENT01 | AGENT02 | ... | AGENT99 | MAILSTATION
<data-section> ::= OBJECTS DEFINITION { <data-decl> ; }
<data-decl> ::= <item-decl> | <group-decl> | <doc-decl>
<item-decl> ::= <symbol> [IS] <data-type> [<format>]
<group-decl> ::= GROUP <symbol> <item-decl> {[ ,] <item-decl>}
                 ENDGROUP
<doc-decl> ::= FORM <symbol> [ LINE <integer> [,] COLUMN
               <integer> ] <marker> <doc-text> <marker>
               { <field-decl> } , ENDFORM
<data-type> ::= ALPHABETIC|NUMERIC
<format> ::= <integer> | <integer> . <integer>
<marker> ::= >
<doc-text> ::= <string-or-field> { <string-or-field> }
<string-or-field> ::= <string> | <field-symbol>
<field-symbol> ::= % <symbol>
<field-decl> ::= <field-symbol> <data-type> [<format>]
                 <field-type> <value>
<field-type> ::= PROTECTED|UNPROTECTED|IDCODE
```

```
<procedure-section>  ::= PROCEDRUE DETAILS  <init-term-marking>
                              <transition>  {;  <transition> }
<init-term-marking>  ::= INITIAL-MARKING [IS]   <integer-list>  ;
                         TERMINATING-MARKING [IS] <integer-list> ;
<transition>  ::= TRANSITION-NUMBER  <integer>  TRANSITION-NAME
                  [IS]  <symbol>  ; INPUT-PLACES [ARE]  <integer-list>
                  OUTPUT-PLACES [ARE]  <integer-list>  ;
                  PREDICATE [IS]  <pred-expression> ; ACTIONS [ARE]
                   <action-list> ENDTRANSITION
<pred-expression>  ::= - | <pred-expl>
<pred-expl>  ::=  <pred-opn> |(<pred-expl>) |<pred-expl>
                  <pred-opr>    <pred-expl>
<pred-opn>  ::=  <data>    <rel-opr>    <data> |
                  EXISTDOC  <symbol> |
                  UNTIL  <date>     <time>  |
                  WAITFOR  <days-hours>
<pred-opr>  ::= AND|OR
        <rel-opr>  ::= >|=|<|≠ | >= | <=
        <date>  ::=  <two-digits>  -  <month-name>  -  <two-digits>
                     <week-day>
        <time>  ::=  <two-digits>  :  <two-digits>
        <days-hours>  ::=  <three-digits>  :  <two-digits>
        <month-name>  ::=   JAN|FEB|MAR|...|DEC
        <week-day>    ::=   SUN|MON|...|SAT
<action-list>  ::= -   <action>  {;  <action> }
<action>  ::= INVOKE  <symbol>  WITH  <data-list>  |
              ASSIGNTO  <symbol> <expression>  |
              SENDTO  <dest> <symbol>  WITH <doc-param-list>  |
                                          <db-operation>
<db-operation>  ::= SELECT  <symbol>  WHERE  <qualifier>
                    GIVING  <symbol>  | <symbol>  UNION  <symbol>
                    GIVING  <symbol>  | <symbol>  MINUS  <symbol>
                    GIVING  <symbol>
```

```
<expression>  ::= <string-exp>  | <arith-exp>
<string-exp>  ::= <string-data>  | <string-data> // <string-exp>
<arith-exp>   ::= <num-data> | (<arith-exp>) | <arith-exp>
                  <arith-opr>    <arith-exp>
<arith-opr>   ::=  + | - | * | /
```

Appendix C. An Example Specification Program --- The Journal Editing Procedure

```
PROCEDURE IDENTIFICATION
  PROCEDURE-NAME journal-editing;
  PARAMETERS ARE paper-no,title,author,address,sub-date;
  AGENT2 IS editor;  MAILSTATION IS mailprint;
OBJECTS DEFINITION
  GROUP papers-t
    paper-no ALPHABETIC 5, title ALPHABETIC 70, author ALPHABETIC 30,
    address ALPHABETIC 30, sub-date ALPHABETIC 8,
    paper-status ALPHABETIC 1
  ENDGROUP;
  GROUP review-t
    pno2 ALPHABETIC 4, reviewer2 ALPHBETIC 30, date2 ALPHABETIC 8,
    status2 ALPHABETIC 1
  ENDGROUP;
  GROUP reviewer-t
    rno1 ALPHABETIC 4, rname1 ALPHABETIC 30, raddr1 ALPHABETIC 30
  ENDGROUP;
  FORM auth-ackn COLUMN 80, LINE 7
       >
       To: %corr
           %addr
       From: CACM Editor

       This is to acknowledge the receipt of your paper entitled
       "%topic".

       Thank you for your intrest in our journal.
       >
       %corr ALPHABETIC 30 PROTECTED,
       %addr ALPHABETIC 30 PROTECTED,
       %topic ALPHABETIC 70 PROTECTED
  ENDFORM:
  FORM revr-sel COLUMN 80, LINE 15
       >
       To:    CACM Editor
       From:  Journal-editing-procedure
       Subject:  %subject1

       Please select one reviewer for the paper entitled
            %title1
       by
            %author1


       Reviewer:  %reviewer1
                                        Sign here:  %ed-sign
```

```
      %subject1 ALPHBETIC 40 PROTECTED VALUE "REVIEWER SELECTION",
      %title1 ALPHABETIC 70 PROTECTED,
      %author1 ALPHABETIC 30 PROTECTED,
      %reviewer1 ALPHABETIC 30 UNPROTECTED,
      %ed-sign ALPHABETIC 8 IDCODE VALUE "XKGZMAAA"
ENDFORM;
FORM to-revr
      >
      To:    %name
             %addr1
      From: CACM Editorial Office
      Subject:  Submission of paper to be reviewed

      We would like to invite you to review the paper entitled
         %p-title
      for us.  Please fill the form appended and mail it to us
      within two months.

      Thank you very much!
      >
      %name ALPHABETIC 30 PROTECTED,
      %addr1 ALPHABETIC 30 PROTECTED,
      %p-title ALPHABETIC 70 PROTECTED
ENDFORM;
FORM revr-response
      >
      To:    CACM Editorial Office
      From: %revr
      Subject:  Paper Review Response

      Do you accept this review request?(Y/N) %c

      Review opinion:
      %ol1
      %ol2
      >
      %revr ALPHBETIC 30 PROTECTED,
      %c ALPHABETIC 1 UNPROTECTED,
      %ol1 ALTHABETIC 80 UNPROTECTED,
      %ol2 ALPHABETIC 80 UNPROTECTED
ENDFORM;
FORM to-revr-1
      >
      To:     %revr-1
              %addr-1
      From:  CACM Editorial Office

      We have to remind you about the paper entitled
      "%tit-1"
      which was submitted to you at %date-1.  Please send the
      response form to us within 5 days.
```

```
      Thank you very much.
      >
      %revr-1 ALPHABETIC 30 PROTECTED,
      %addr-1 ALPHABETIC 30 PROTECTED,
      %tit-1 ALPHABETIC 70 PROTECTED,
      %date-1 ALPHABETIC 8 PROTECTED
ENDFORM;
FORM ed-decision
      >
      To:    CACM Editor
      From:  Journal-Editing-Procedure
      Subject:  Final decision to accept paper or not

      The reviewer's opinion of the paper
        %tt
      is
        %o1
        %o2

      Final desision:(A--accept,R--reject) %f-dec

                                          Sign here:  %sig
      >
      %tt ALPHABETIC 70 PROTECTED, %o1 ALPHABETIC 80 PROTECTED,
      %o2 ALPHABETIC 80 PROTECTED,
      %sig ALPHABETIC 8 IDCODE VALUE "XKGZMAAA"
ENDFORM;
FORM auth-ackn-1
      >
      To:    %corr-1
             %addr-2
      From:  CACM  Editorial Office
      Subject:  Response for paper submission

      This is to inform you that your paper
        %title-1
      submitted at %date-2 has been %arr .

      Thank you for your intrest in our journal.
      >
      %corr-1 ALPHABETIC 30 PROTECTED,
      %addr-2 ALPHABETIC 30 PROTECTED,
      %title-1 ALPHABETIC 70 PROTECTED,
      %arr ALPHABETIC 6 UNPROTECTED,
      %date-2 ALPHABETIC 8 PROTECTED
ENDFORM;
```

```
PROCEDURE DETAILS
  INITIAL-MARKING IS 1;
  TERMINATING-MARKING IS 4,12;
  TRANSITION-NUMBER 1
    TRANSITION-NAME paper-login;
    INPUT-PLACES 1; OUTPUT-PLACES 2,3;
    PREDICATE  - ;
    ACTIONS
            ASSIGNTO paper-status "E";
            papers UNION papers-t;
  ENDTRANSITION;
  TRANSITION-NUMBER 2
    TRANSITION-NAME author-acknowledge;
    INPUT-PLACES 2; OUTPUT-PLACES 4;
    PREDICATE   - ;
    ACTIONS
            SENDTO mailprint auth-ackn WITH corr=author,
                   addr=address,topic=title;
  ENDTRANSITION;
  TRANSITION-NUMBER 3
    TRANSITION-NAME reviewer-selection;
    INPUT-PLACES 3; OUTPUT-PLACES 5;
    PREDICATE  - ;
    ACTIONS
            SENDTO editor revr-sel WITH subject1,title1=title,
                   author1=author,reviewer1,ed-sign;
  ENDTRANSITION;
  TRANSITION-NUMBER 4
    TRANSITION-NAME log-review;
    INPUT-PLACES 4,5; OUTPUT-PLACES 4,6;
    PREDICATE  - ;
    ACTIONS
            review UNION (pno=paper-no,reviewer=reviewer1,
                          date=$DATE,status="B");
  ENDTRANSITION;
  TRANSITION-NUMBER 5
    TRANSITION-NAME inform-reviewer;
    INPUT-PLACES 6; OUTPUT-PLACES 7;
    PREDICATE  - ;
    ACTION
            SELECT reviewer WHERE rname=reviewer1 INTO reviewer-t;
            SENDTO mailprint to-revr WITH name=reviewer1,
                   addr2=raddr1,p-title=title;
            SENDTO mailprint revr-response WITH revr=reviewer1,
                                                c,o11,o12;
  ENDTRANSITION;
  TRANSITION-NUMBER 6
    TRANSITION-NAME wait-response-1;
    INPUT-PLACES 7; OUTPUT-PLACES 9;
    PREDICATE
            EXISTDOC revr-response;
```

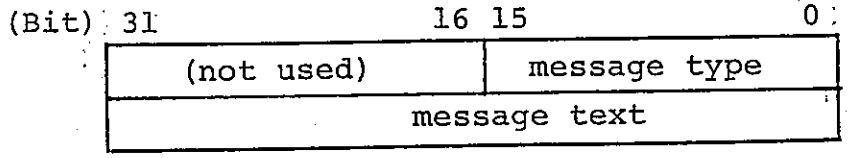
```
  ACTIONS  - ;
ENDTRANSITION;
TRANSITION-NUMBER 7
  TRANSITION-NAME two-month-timing;
  INPUT-PLACES 7; OUTPUT-PLACES 8;
  PREDICATE
          WAITFOR 60:00;
  ACTIONS
          SELECT review WHERE pno=paper-no INTO review-t;
          SENDTO mailprint to-revr-1 WITH revr-1=reviewer1,
                              addr-1=raddr1,tit-1=title,date-1=date2;
ENDTRANSITION;
TRANSITION-NUMBER 9
  TRANSITION-NAME reselection-timeout;
  INPUT-PLACES 8; OUTPUT-PLACES 3;
  PREDICATE
          WAITFOR 7:00;
  ACTIONS
          ASSIGNTO subject1 "Reviewer reselection due to timeout";
          review MINUS review-t;
ENDTRANSITION;
TRANSITION-NUMBER 8
  TRANSITION-NAME wait-response-2;
  INPUT-PLACES 8; OUTPUT-PLACES 9;
  PREDICATE
          EXISTDOC revr-response;
  ACTIÓNS  - ;
ENDTRANSITION;
TRANSITION-NUMBER 10
  TRANSITION-NAME review-rejected;
  INPUT-PLACES 9; OUTPUT-PLACES 3;
  PREDICATE
          NOT c="Y";
  ACTIONS
          ASSIGNTO subject1
                   "Reviewer reselection due to rejection";
          SELECT review WHERE    pno=paper-no INTO review-t;
          review MINUS review-t;
ENDTRANSITION;
TRANSITION-NUMBER 11
  TRANSITION-NAME final-decision;
  INPUT-PLACES 9; OUTPUT-PLACES 10;
  PREDICATE
          c="Y";
  ACTIONS

        SENDTO editor ed-decision WITH tt=title,o1=o11,
                                       o2=o12,f-dec,sig;
ENDTRANSITION;
```

```
TRANSITION-NUMBER 12
  TRANSITION-NAME paper-rejected;
  INPUT-PLACES 10; OUTPUT-PLACES 11;
  PREDICATE
          f-dec="R";
  ACTIONS
          SENDTO mailprint auth-ackn-1 WITH corr-1=author,
                 addr-2=address,title-1=title,arr="rejected",
                 date-2=date2;
          papers MINUS papers-t;
ENDTRANSITION;
TRANSITION-NUMBER 13
  TRANSITION-NAME paper-accepted;
  INPUT-PLACES 10; OUTPUT-PLACES 11;
  PREDICATE
          f-dec="A";
  ACTIONS
          SENDTO mailprint ahth-ackn-1 WITH corr-1=author,
                 addr-2=address,title-1=title,acc="accepted",
                 date-2=date2;
          papers MINUS papers-t;
          ASSIGNTO paper-status "D";
          papers UNION papers-t;
ENDTRANSITION;
TRANSITION-NUMBER 14
  TRANSITION-NAME clear-files;
  INPUT-PLACES 11; OUTPUT-PLACES 12;
  PREDICATE  - ;
  ACTIONS
          review MINUS review-t;
ENDTRANSITION;
END
```

Appendix D. Intercommunication Message Format

The common message format is



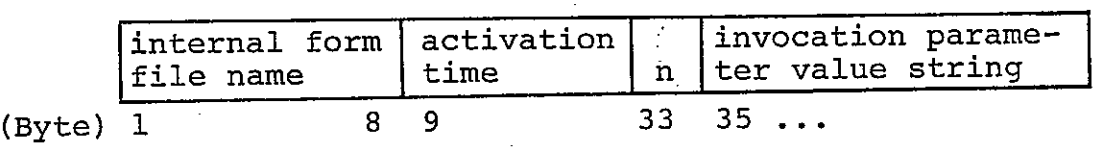
where,

message type: message type code, e.g., MSG_MT, MSG_MIP, etc,

message text: contents of message, variable-length.

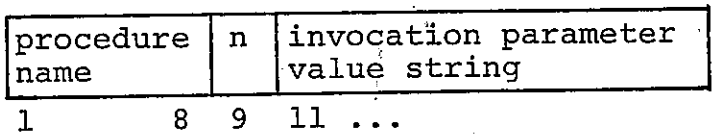
The message text part of each message type is detailed below.

MSG_MIP (Monitor initialization)



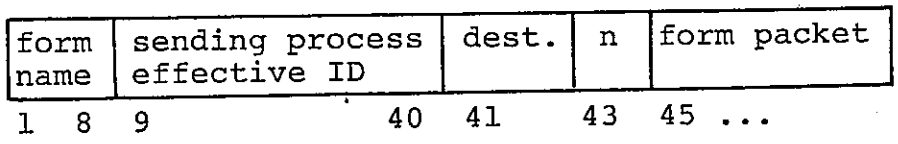
n: length of invocation parameter value string

MSG_PI (Procedure invocation)



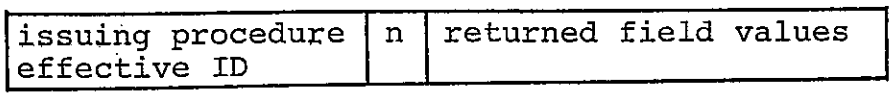
n: length of invocation parameter value string

MSG_FP (Form packet)



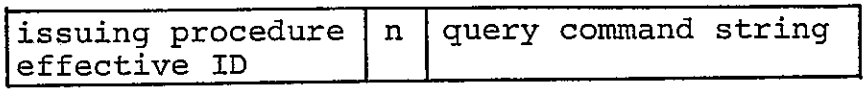
n: length of form packet

MSG_FFR (Returned form fields)



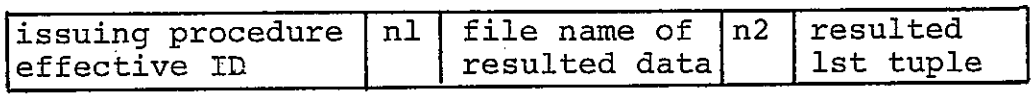
n: length of field value or form manager/mail manager error code

MSG_DBO (Database operation)



n: length of query command string

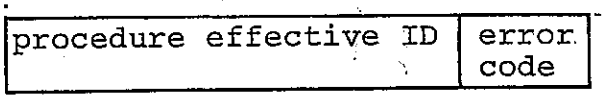
MSG_DBR (Database operation result)



n1: length of file name or data manager error code

n2: length of lst tuple

MSG_MER (Monitor error report)



Appendix E. Procedure Internal Form

1. Procedure Contral Block (PCB)

Field name	Type	Descrittion
PRC_ID	char*8	procedure ID
PRC_AT	char*24	activation time, in VMS ASCII time format
PRC_STA	char*1	execution status
PRC_CMK	int	current marking pointer to TMT entry
PRC_CTR	int	current executing transition pointer (to TDT)
PRC_IPL	int array	list of pointers to ODT entries used as invocation parameters

2. Marking Table (MT)

MK_NAT	int	no. of active transitions in this marking
MK_FAT	int	lst active transition, pointer to ATL

3. Active Transition List (ATL)

AT_TP	int	tran. internal form pointer (to TDT)
AT_OMK	int	output marking pointer (to MT)

4. Transition Detail Table (TDT)

TR_SN	char*20	symbolic name of transition
TR_FIRE	char*1	in-firing flag
TR_PEP	int	pred. expression pointer

Field name	Type	Description
TR_NAC	int	no. of activities
TR_FAC	int	lst activity pointer (to AL)
TR_CAC	int	activity counter

5. Predicate Expression List (PEL)

PE_PEL	int array	predicate expression list

6. Predicate Component List (PCL)

PC_OPR	char*1	predicate operator
PC_OPN1	int	pointer to operand 1
PC_OPN2	int	pointer to operand 2

7. Activity List (AL)

AC_COD	char*1	activity code
AC_NAG	char*1	no. of arguments
AC_FAG	int	pointer to lst argument

8. Actual Argument List (AAL)

AR_AAL	int array	storage for argument list of activites

9. Objects Description Table (ODT)

OD_TYPE	char*2	type information
OD_LEN	char*2	length information
OD_XD	char*2	an extra descriptor
OD_STA	int	data address in ODA

10. Object Data Area (ODA)

OD_ODA	char array	storage area for objects (in characters)

Appendix F. Formats of Activity Internal Form

1. ASSIGNTO object expression

```
AC_COD = AC$ASG
AC_NAG = 1
AAL    = pointer to PEL for prefix Polish string
```

2. INVOKE proc-name WITH obj-list

```
AC_COD = AC$PI
AC_NAG = n (# invoking parameters)+1
AAL    = procedure-name invoked
         p1 \
         p2  |
          .  |  n parameters
          .  |
          .  |
         pn /
```

3. SENDTO dest form WITH field-list

```
AC_COD = AC$SND
AC_NAG = 2 + # form argument fields
AAL    = form name
         destination code
         a1 \
         a2  |
          .  |  argument fields, ai=0, no related source/sink
          .  |                    >0, source/sink objects
          .  |
         an /
```

4. DB operation:

```
AC_COD = AC$DB
AC_NAG = 4+n*2
AAL    = code (1-SELECT, 2-UNION, 3-MINUS)
         rel1 (pointer to relation name string in ODA)
         rel2 (  0, pointer to relation name in ODA;
                 0, pointer to group item in ODT)
         n    (# of attribute/value pairs)
         a1   (attribute 1)
         v1   (value 1)
          .
          .
          .
```

Appendix G. Characteristics and Coding of Objects

Objects	OD_TYPE		OD_LEN		OD_XD		OD_STA
	byte 1	byte 2	byte 1	byte 2	byte 1	byte 2	
Single Item	"010	A* 9	# characters # int. digits	#frac. digits	-	-	ODA address
Group Item	"20	A	#char		#comp.	items	"
Comp. Item	"21	A 9	(same as "10)		ptr to	group	"
Form	"30	A	line dim.	col dim.	#field	inter. flag	addr of chain proc. name
literal field	"31	A	#char		line loc.	col loc.	ODA address
ID field	"32	A	"		"	"	"
unpro. field	"33	A 9	(same as "10)		"	"	"
protect. field	"34	A 9	"		"	"	"

* A: alphabetic, 9: numeric.

Appendix H. Implementation of OPAS Modules

Monitor

1) 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 = proc effective ID).
6. Start Petri net driving (the following).

2) Transition selection

1. Start with the first 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.

3) Transition firing

1. Fire the selected transition by performing activities associated with it one by one.
2. After done, go to 4).

4) Marking advance

1. Set the current marking to the output marking of the current transition.

2. Go to 2).

Form Manager

1) Initializations

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

2) 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" 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 go to step 1.

3) 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 first request in the queue, put the corresponding terminal into "start I/O chain".

3. Wake up form manager from hibernating.

4) 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

1) Initializations

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

2) Mailbox reading handler

1. If the form is to be returned (having input fields), register and save it into "Pending Form Table".
2. Print out a hardcopy of the form on printer (with registration number).

3) 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 a MSG_PI message to the supervisor.
4. go to step 1.

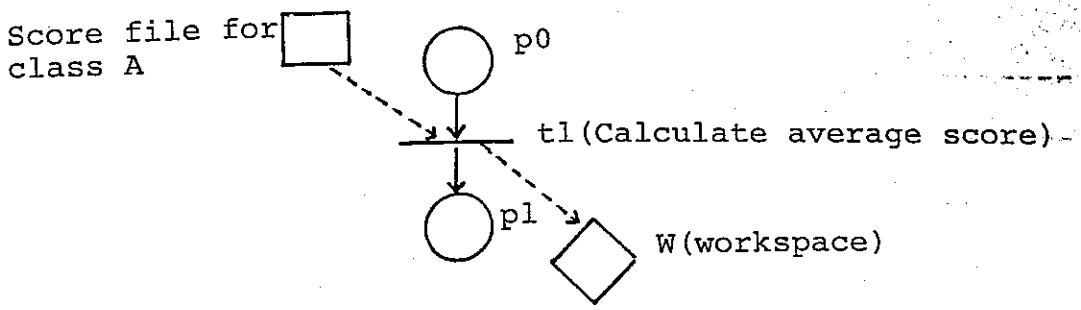


Figure 2-1 Illustration of data graph in the PNB model

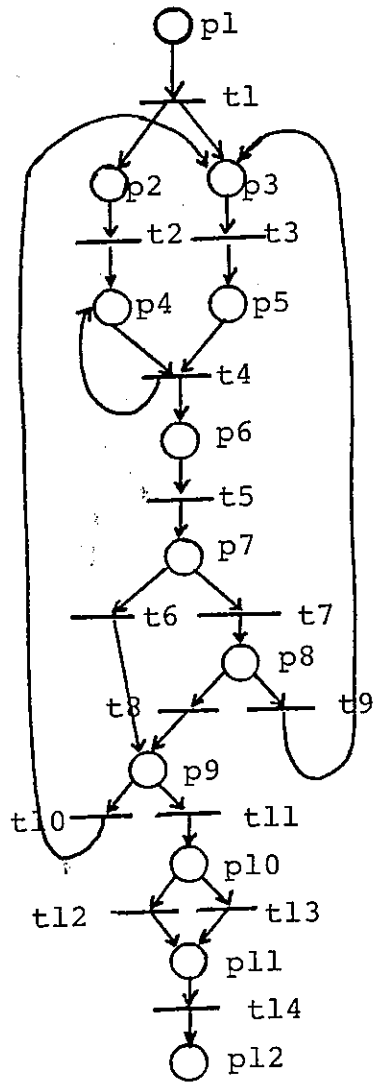


Figure 2-2 The PNB model for journal editing procedure (continued on next page)

Transition	Predicate	Action
t1	-	Log submission information in file PAPERS.
t2	-	Send acknowledge form F2 to author.
t3	-	Prompt F3 to the editor for reviewer selection.
t4	-	Insert a review record into the file REVIEW.
t5	-	Send letter (F4) and a copy of paper to the reviewer.
t6	F4* arrived	-
t7	2 months elapsed	Send letter F5 to inform the reviewer.
t8	F4* arrived	-
t9	5 days elapsed	Inform the editor.
t10	Review rejected	-
t11	F4* is review opinion	Promt form F7 to the editor for final decision.
t12	Paper rejected	Delete PAPERS record; Inform the author.
t13	Paper accepted	Update status of PAPERS record; Inform the author.
t14	-	Delete REVIEW record.

Figure 2-2 (Continued)

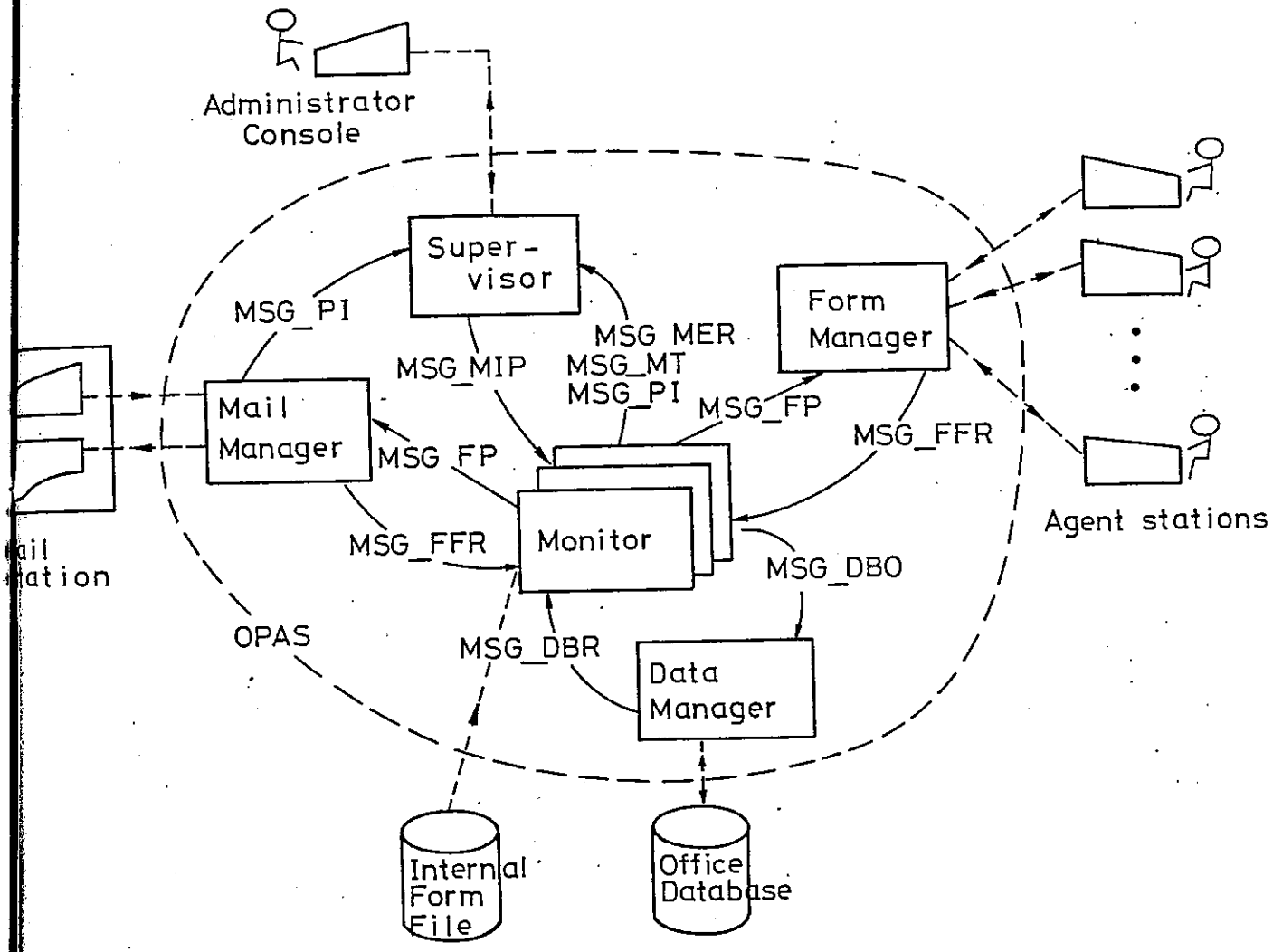
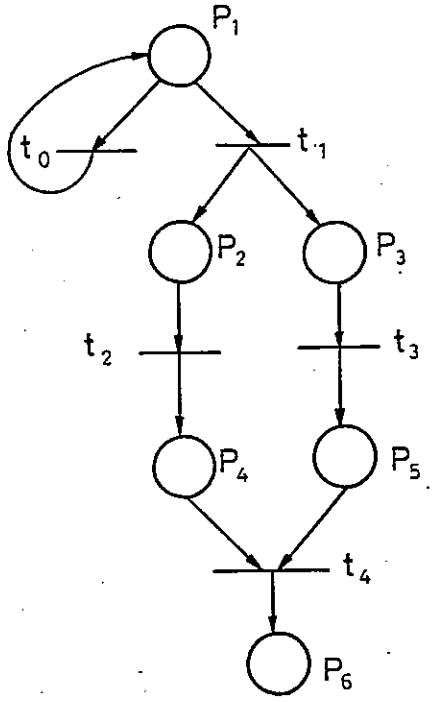


Figure 4-1 OPAS System Architecture



(a) petri-net

MT

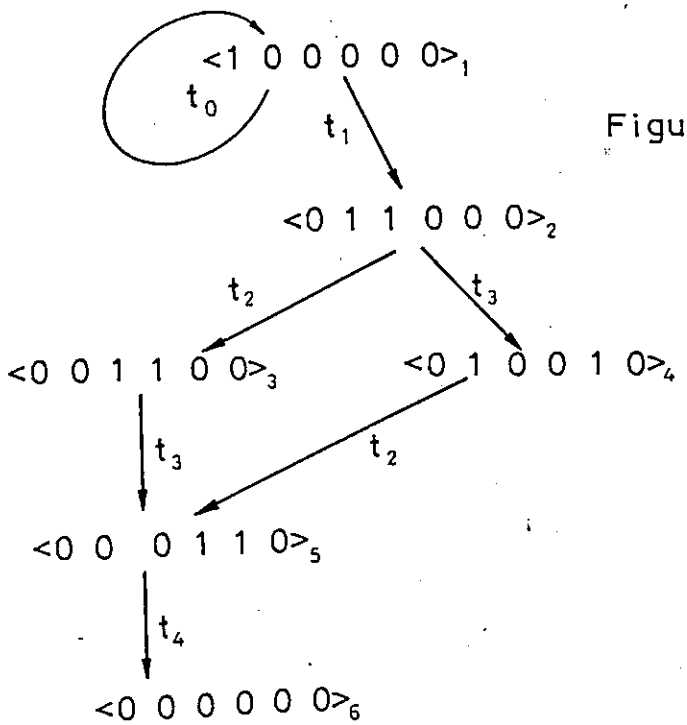
	MK_NAT	MK_FAT
1	2	1
2	2	3
3	1	5
4	1	6
5	1	7
6	0	0

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

Figure 4-2 Petri net driving data structures MT and ATL

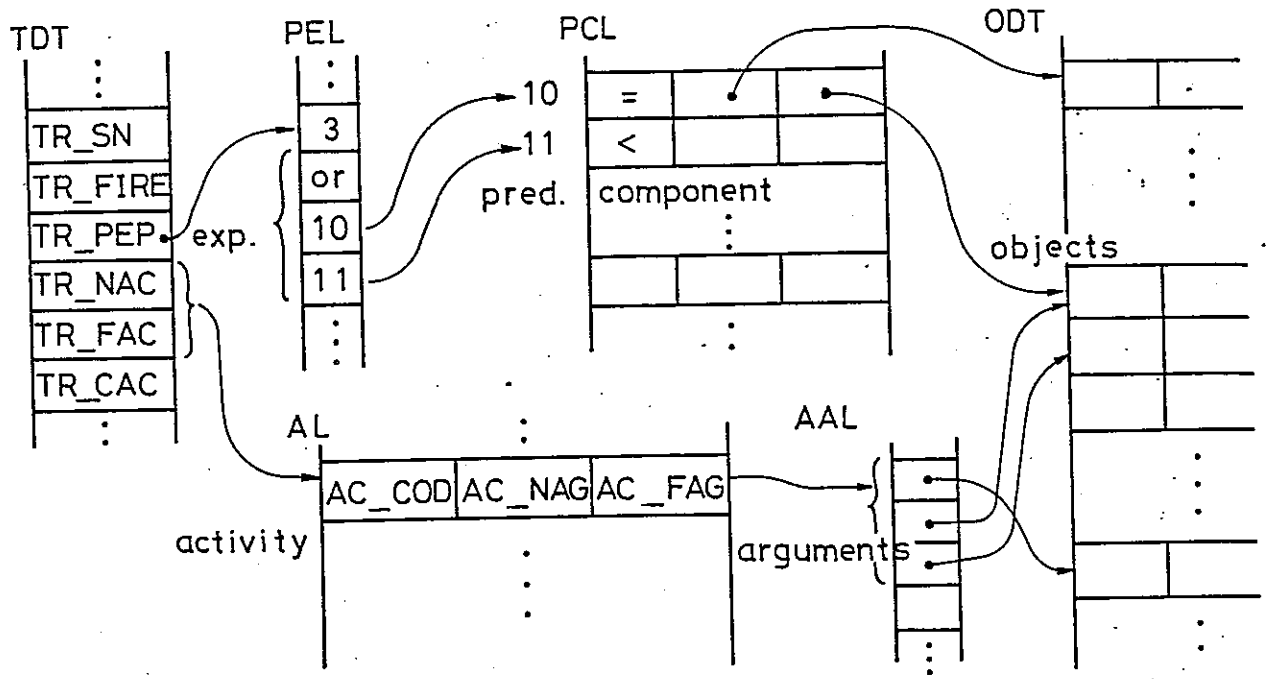
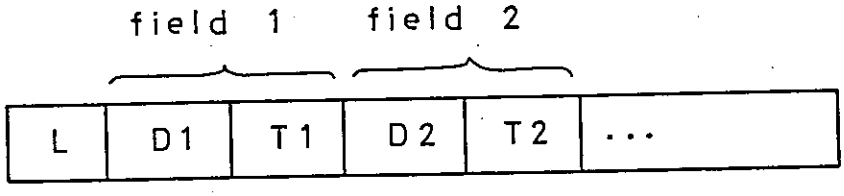


Figure 4-3 Activity realization data structures



L: total length

Di: 6-byte field descriptor
(consisting of OD_TYPE, OD_LEN, OD_XD)

Ti: field value

Figure 4-4 Form packet format

References:

1. Bailey, A. D., Gerlach, J., McAfee, R. P. and Whinston, A. B., "Internal Accounting Controls in The Office of The Future," IEEE Computer, May 1981.

2. Baumann, L. S. and Coop, R. D., "Automated Workflow Control: A Key to Office Productivity," Proc. AFIPS Office Automation Conference, March 1980.

3. Chang, S. K. ,"Knowledge-Based System," Chapter 14 (to be published).

4. DEC, 'VAX/VMS System Services Reference Manual," 1978.

5. Ellis, C. A., "Office Information Systems and Computer Science," ACM Computing Surveys, 12, 1, 3, 1980.

6. Ellis, C. A., "Information Control Nets : A Mathematical Model of Office Information flow," Conference on Simulation, Measurement and Modeling of Computer System, 1979.

7. Hammer, M., Howe, W. G., Krushal, V. I., and Wlandawsky, I., "A Very High-Level Programming Language for Data Processing Applicaions," CACM 20, 11, 4, 1977.

8. Ho, C. C., "Office Systems Modeling, Analysis and Design," MS Thesis, National Taiwan University, Taipei, 1982.

9. Ho, c. C., Hong, Y. c., Ho, Y. W., and T. S. Kuo, "An Office Workflow Model," Proc. NCS, Taiwan, Taiwan, Dec. 1981, pp.354-368.

10. Ku, Y. M., "A Tool for Designing Information Transfer Systems," MS Thesis, National Chiao-Tung University, Hsin-Chu, Taiwan, 1981.

11. Peterson, J. L., "Petri Nets," ACM Computing Survey 9, 3, 3, 1977.

12. Tsichritzis, D., "OFS : An Integrated Form Management Systems," Proc. VLDB, 1982.

13. Tsichritzis, D., "Integrating Data Base and Message Systems," Proc. VLDB, 1982.

14. Uhlig, P. R., Farber, D. J., Bair, J. H., "The Office of The Future," ICCC, 1979.

15. Wohl, Amy, "A Review of Office Automation," Datamation, February 1980.

16. Zisman, M. d., "Representation, Specification and Automation of Office Procedures," Ph.d. Dissertation, Wharton School, University

of Pennsylvania, 1977.

17. Zloof, M. M., "QBE/OBE : A Language for Office and Business Automation," IEEE Computer, May 1981.