The Design and Implementation of the SEAU Procedure Management System

A Thesis Submitted For The Degree Of Master of Science

by

Gary Stephens BSc.

School of Computer Applications
Dublin City University
Dublin 9

October 1992.

Supervisor: Renaat Verbruggen

This thesis is based on the candidate's own work, and has not previously been submitted for a degree at any academic institution.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Renaat Verbruggen, for his guidance and assistance during my research.

I am also very grateful to IBM Ireland Information Services Ltd. for their generous sponsorship of my work and for the facilities made available to me, and I wish to thank Mark Sweetnam, Paraic Sweeney and Aidan Clarke of IBM IISL for their advice and assistance.

Gary Stephens
October 1992.

Table of Contents

Chapter 1 : Procedure Management Systems 4
1.1 Introduction 4
1.2 Existing models & systems 9
1.2.1 OSIRIS 9
1.2.2 Electronic Circulation Folders 16
1.2.3 XCP 22
1.2.4 OTM 25
1.2.5 Augmented Petri Nets 28
1.2.6 VPL 32
1.3 Summary 36

Chapter 2 : Procedure Management System Design Issues . . . 38
2.1 Introduction 38
2.2 Structured and unstructured procedures 38
2.3 The need for a low-level model 39
2.4 Objects and their routing 40
2.5 Support and automation 40
2.6 Attributing status to objects 42
2.7 Making parts of procedures optional or mandatory . 43
2.8 Separate organisational database 46
2.9 PMS design guidelines 46
2.9.1 Support for structured procedures 46
2.9.2 Low-level model 47
2.9.3 Separate organisational database 48
2.9.4 User discretion 48
2.9.5 System should deal with generic objects . . 48
2.9.6 Robustness, efficiency 49
2.9.7 Support and automation 49
2.9.8 Multiple platforms 49
2.10 Summary 49

Chapter 3 : Procedure Representation 51
3.1 Introduction 51
3.2 Low-level model 52
3.3 Advantages/disadvantages of the low-level model . . 54
3.4 Post-conditions 55
3.5 Modifying executing procedures 55
3.6 Examples of high-level model features 56
3.6.1 Translating a high-level model into a low-level model 57
3.6.2 Procedures within procedures 61
3.6.3 User discretion 61
3.6.4 Concurrent access to objects 62
3.6.5 Roles 62
3.7 Summary 63

Chapter 4 : The SEAU System 65
4.1 Introduction 65
4.2 System architecture 66
4.2.1 Effects of having to design for AIX and VM/CMS 67
4.3 System components 69
4.3.1 *Execution* component (server) 69

4.3.2 *Submit* component 73
4.3.3 *Allocate* component 74
4.3.4 User Interface components 75
4.3.5 Organisational database 78
4.4 Limitations of the *SEAU* system 79
4.4.1 High-level procedure specification component 79
4.4.2 High-level procedure translation component 79
4.4.3 High-level procedure monitoring component . 80
4.4.4 Procedure automation component 81
4.5 Application architecture 82
4.5.1 Precondition program architecture 83
4.5.2 Action program architecture 83
4.5.3 Procedure element architecture 83
4.6 Procedures within procedures 84
4.7 Summary 85

Chapter 5 : The Application of the SEAU System 86
5.1 Introduction 86
5.2 Implementing high-level procedures using the low-level model 86
5.2.1 OSIRIS 86
5.2.2 XCP 87
5.2.3 Electronic Circulation Folders 88
5.2.4 Augmented Petri Nets 94
5.2.5 VPL 98
5.3 Summary 100

Chapter 6 : Conclusions 102
6.1 Introduction 102
6.2 Conclusions 102
6.2.1 Structured procedures 102
6.2.2 Procedure representation 103
6.2.3 Workflow management is only an application of a PMS 104
6.2.4 Separate organisational database 104
6.2.5 PMS should ignore the contents of objects . 104
6.2.6 Degrees of procedure support / automation . 105
6.2.7 Procedure types 105
6.2.8 Allocation and submission components . . . 107
6.3 Future work 107
6.3.1 Robustness, efficiency 108
6.3.2 Additional components 108
6.3.3 Complete groupware system 108
6.4 Overall summary 109

Appendix A : References 1

Appendix B : Rule Definition Language 1

Abstract

The Design and Implementation of
the *SEAU* Procedure Management System

by Gary Stephens

An emerging requirement across a range of industries is to be able to quickly and efficiently automate an organisation's official, and also more ad-hoc, policies and procedures. A Procedure Management System is a system which assists users in carrying out these procedures.

The purpose of the research presented in this thesis has been to

- define a low-level model for the representation of procedures
- construct a platform-independent prototype Procedure Management System (PMS) (the *SEAU* system) which supports this model
- experiment with the use of this PMS for representing and enacting procedures defined using other high-level models
- assess the suitability of the model as a low-level model for the representation of procedures.

We begin by explaining what a Procedure Management System is and by examining some existing Procedure Management Systems and the models used in them for procedure representation.

We then discuss some important issues in the design of a Procedure Management System, particularly the models used for representing procedures. A number of guidelines are outlined which should be followed in the design of a model for representing procedure and for the design of a prototype Procedure Management System.

A low-level rule-based model for the representation of procedures which may be used as the basis for a PMS is then presented. Also given are some important features of high-level procedure specification models and it is shown how these might be implemented in a PMS.

The architecture of the *SEAU* (Submission, Execution, Allocation, User-Interface) PMS, and the individual components which make up this system, are described. The criteria that must be conformed to by programs which are to be used with the *SEAU* system are also given, and the way in which the system assists in the execution of sub-procedures is described.

The use of the SEAU system for the implementation of example procedures, defined using a number of different high-level models, is then examined. It is explained how some of the features of these models may be implemented using the low-level model used in the *SEAU* system, and features of the example procedures which caused some difficulty during implementation are highlighted.

Finally, we summarise the conclusions reached as a result of this research and outline some possible future research directions, including ways in which the *SEAU* system might be enhanced.

Chapter 1 : Procedure Management Systems

1.1 Introduction

Conventional software applications perform specific pre-defined tasks in a stand-alone computing environment and are usually executed by a single user, or where they involve more than one user, each user must be specifically registered to the application. An emerging requirement across a range of industries is to be able to quickly and efficiently automate an organisation's official policies and procedures (and also more ad-hoc procedures), a requirement which conventional software applications are incapable of supporting.

Groupware [Ellis91] is a term applied to hardware/software which facilitates groups of users in performing cooperative work. The commonest example of a groupware system in use today is electronic mail. *Procedure Management Systems (PMS)* are another example of groupware.

Groupware may be categorised into two basic types :

- **Information sharing**, where the system manages information and helps users share and update that information

- **Workflow**, where the system manages the flow of work and the content of the objects being worked on is left up to users.

These two basic types can of course be mixed. The data relating to the current status of a piece of work (in a **workflow** type system) could be stored using the same information management system used for office

documents (**information sharing** type system), and in this way these two basic types may be interrelated.

Procedure Management Systems fit into the **workflow** category. The terms *workflow*, *process*, and *task* have also been used to describe what shall be referred to here as a *procedure*. By *procedure* we mean a set of steps designed to achieve a certain goal. One can view the work that goes on in an organisation as consisting of a number of procedures (whether formalised or ad-hoc), each of which can be decomposed into a number of sub-procedures, and so on, resulting in a hierarchical structure of procedures. At the lowest level of the hierarchy, procedures are decomposed into elementary activities, or *procedure elements*. These elementary tasks might be implemented through the use of a computer program. Each elementary task would be allocated to a particular user to be performed.

The distinction has been made between structured and unstructured procedures [Mazer87]. Very "structured" procedures are those which can be described using structured algorithms, e.g. any typical data processing activity, such as producing payroll cheques. Very "unstructured" procedures are those that are difficult to specify algorithmically and involve a large degree of problem solving, e.g. deciding whether to accept or reject a merger offer.

A PMS may be used to support / automate an organisation's standard procedures (as defined for the whole organisation), or more ad-hoc procedures created by individual users, which, in time, may become adopted by the whole organisation. By the term *support* we mean that the system should tell users what tasks they are supposed to perform and facilitate them in performing them by having all the necessary data ready to be worked upon. By the term *automate* we mean that the

system should actually carry out the task without the user's involvement being required.

To illustrate the kind of requirements that a PMS should be able to support, consider the following example. An insurance company's procedure for issuing life assurance policies is :

- A clerk receives the original proposal, and verifies that all required information and documentation has been provided by the applicant.

- The clerk determines which company approved doctor is geographically nearest the applicant and sends a letter to the applicant asking him/her to arrange a medical with that doctor. The doctor is sent a copy of that letter.

- If a response is not received from the doctor within a predefined time a reminder is send to the applicant.

- If a response is not received from the doctor within another predefined time the application is cancelled.

- If the response from the doctor is negative a refusal letter is sent to the applicant and the procedure terminated.

- If the response from the doctor is positive an actuary calculates the premium.

- A proposal is sent to the applicant.

A number of other requirements might apply :

- The company's target is that, excluding the time between the letter requesting a medical being sent and the response from the doctor being received, that the entire procedure be handled within three working days. The manager of the clerk concerned is to be notified if any such procedure is not completed within that time.

- Although, with the manual procedure, an actuary calculates the premium, the company wants to automate that step by obtaining the premium from an actuarial database.

More general requirements of a PMS include the ability to :

- Support procedures where the people involved have full discretion over the completion of the procedure as well as those where they do not. A PMS it should be able to ensure that specific steps in a procedure are performed exactly as specified, while allowing other steps to be carried out only if the user taking part in the procedure sees fit.

- Allocate work appropriately to individuals or groups of people.

- Balance the workload of individuals and groups of people.

- Enable the current status of a procedure to be queried.

- Generate reports containing statistics about procedures.

- Allow a single procedure to involve people in different organisational or geographic areas and on different computing platforms.

- Allow integration with existing applications.

- Allow the integration of data from and about completed and in-progress procedures into existing company databases.

- Allow the easy creation of 'one-off' procedures by end-users in addition to an organisation's standard procedures. An example of such a one-off procedure is a user wanting to send a filled-in form to another user, and when she/he has approved the form to forward it to a particular department for processing.

- Allow procedure to be specified using different models. Different models may be used for defining different types of procedures (each office might want to use it's own model). One would therefore require a different execution engine to execute each different type of model. A preferable solution would be if one were to have a single low-level model onto which one could map different high-level models (as used in different types of office) (both structured and un-structured), thereby only requiring a single execution engine.

Based on these requirements to support procedures such as the example procedure presented and the more general requirements of a PMS, the goals of this research are :

- to define a low-level model for the representation of procedures, onto which many different high-level models, for the representation of different type of procedures, may be mapped. These high-level models may be of a structured or un-structured nature.

- to construct a prototype PMS which supports the use of this low-level model for procedure representation. This prototype system should not be tied down to any specific computing platform - to this end it has been developed on two different operating systems (AIX and VM/CMS).

- to experiment with the use of the prototype PMS and the low-level model for representing and enacting procedures defined using other high-level models and to examine problems which occur.

- and to thus assess the suitability of the model as a low-level model for the representation of procedures.

In the remainder of this first chapter some of the models and systems which have influenced the design of the *SEAU* model and system will be examined and analysed with a view towards the development of a low-level model which will support these models.

1.2 Existing models & systems

1.2.1 OSIRIS

[Maiocchi87] describes the *OSIRIS* model, a model for the specification of office systems, which may be used for the specification of office procedures. The OSIRIS model incorporates elements of two earlier models; the Semantic Office System (SOS) model [Bracchi84] and the Information Control Nets (ICN) model [Ellis79].

The OSIRIS model seeks to model the flow of control and information in an office by modelling such concepts as documents, workers and tasks. [Maiocchi87] also presents a mechanism which allows the 'learning' and 'refinement' of procedures specified using the OSIRIS model, but this is outside the scope of this thesis.

The basic elements of the OSIRIS model are :

- **Documents** (collections of elementary data)
- **Dossiers** (collections of documents)
- **Agents** (single employees and groups or categories of employees)
- **Activities** (descriptions of work that is to be done)

An *activity* is a description of a task that is to be performed. Each activity may be decomposed into lower level activities. At each level of refinement, each activity is connected to other activities through control structures. At the lowest level an activity is decomposed into *elementary activities*, which correspond to the invocation of one of the tools actually available in the system, such as editors, database programs, spreadsheet programs. At the top level an activity is referred to as a *procedure*.

Control structures specify the relationships between the activities that make up a procedure in terms of synchronisation rules, and indicate the documents which are exchanged, which agents are performing which activities, and what documents are used, created or deleted by activities.

Agents are grouped into *classes*. A class is characterised by a set of procedures that the agents belonging to that class can execute. Classes are disjoint, i.e. a procedure cannot be in more than one class and an agent must belong to at least one class.

The system notifies the user of each activity (complex or elementary) which he/she must perform. When the user has performed a given (complex or elementary) activity, she/he notifies the system and it presents the next activity to be performed in that procedure. At any time the user can modify the procedure by adding new items to the list of activities to be performed.

When a step in a procedure which must be performed by a different user is reached, the user currently in possession of the procedure must pass the procedure to that user in order for execution to continue. It would be more helpful if the system automatically passed each procedure element to the user who was to perform it.

OSIRIS provides a notation for specifying the synchronization of activities using *path expressions*. The control structures used are as follows :

;	sequence	Activities are listed in the order they are to be executed, separated by the *;* sign.
|	selection (XOR)	A selection from a set of activities permits only one to occur. A selection condition *c* is usually associated with a selection.
$+_c$	iteration	An iteration permits an activity to be performed one or more times, depending on the value of condition *c*.

&	parallelism (AND)	Parallel execution permits the activities specified to be executed in conjunction.
( )	parentheses	Parentheses are used to group activities into more complex blocks.

OSIRIS also provides a graphic representation of all the elements in the model. Flow graphs show the precedence relationships between activities (complex precedence relationships involving AND or XOR are also catered for), and show which documents are used by each activity.

1.2.1.1 Example OSIRIS procedure

The model is illustrated with both a path expression representation and a graphical representation (see Figure 1.1) of a procedure which forms part of a task for hiring a person for a job.

The procedure is as follows :

> Three employees are involved : a secretary, an interviewer, and a chief clerk. The chief clerk examines applications (i.e. covering letter and the curriculum vitae of the applicant) as they arrive to the office. The chief clerk knows the needs of the company, and can then decide to refuse the application, or to accept it temporarily.
>
> The secretary notifies the candidate either that his/her application has been rejected or accepted. In the latter case, the secretary must also notify the candidate of the

date fixed for the interview and must update the schedule of the interviewer.

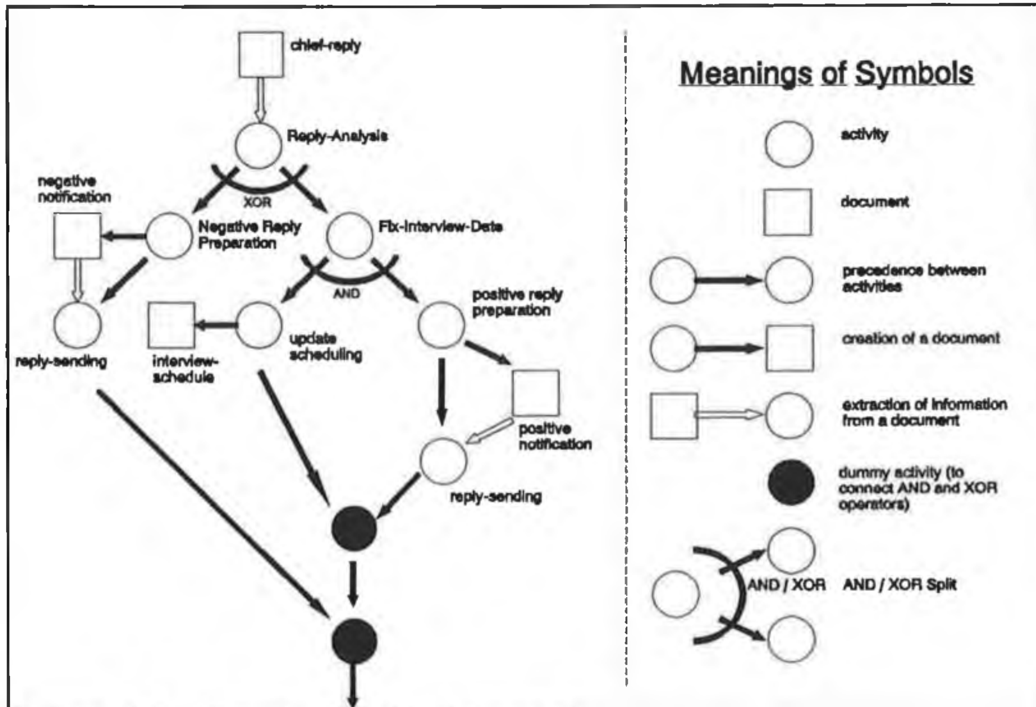

Figure 1.1 Example OSIRIS Procedure

The path-expression representation of this procedure is :

Prepare-Reply =	Reply-Analysis; (Negative-Reply-Preparation; Reply-Sending) | (Fix-Interview-Date; ((Positive-Reply-Preparation; Reply-Sending) & Update-Scheduling))

...which may be structured as follows to make it more readable :

```
Prepare-Reply =

      Reply-Analysis;
      (
            Negative-Reply-Preparation;
            Reply-Sending
      )
      |
      (
            Fix-Interview-Date;
            (
                  (
                        Positive-Reply-Preparation;
                        Reply-Sending
                  )
                  &
                  Update-Scheduling
            )
      )
```

It is also possible to specify the procedure as a hierarchy of complex activities :

Prepare-Reply =	Reply-Analysis; Negative-Reply | Positive-Reply
Negative-Reply =	Negative-Reply-Preparation; Reply-Sending
Positive-Reply =	Fix-Interview-Date; (Reply & Update-Scheduling)
Reply =	Positive-Reply-Preparation; Reply-Sending

The outcome of the Reply-Analysis step controls whether the positive reply path or negative reply path is followed.

The negative reply path consists of two steps in sequence (Negative-Reply-Preparation and Reply-Sending).

The positive reply path consists of Fix-Interview-Date followed by two actions in parallel (the Positive-Reply-Preparation followed by Reply-Sending, and the Update-Scheduling actions).

1.2.1.2 Comments

The OSIRIS model allows the modelling of office tasks in a high-level form which might be used by non-programmers, and the resulting task specifications may be verified and refined.

Such concepts as different categories of employees are not central to the issue of controlling the flow of work and could be kept separate in a lower level model which concentrated on the precedence of activities.

The concept of the role played by a user is a useful one which allows tasks to be allocated to anyone in a set of users who is capable of performing that task.

1.2.2 Electronic Circulation Folders

The *Electronic Circulation Folder* (ECF) system [Karbe90a,Karbe90b] was produced as a result of the ESPRIT project ProMInanD (Extended Office Process Migration with Interactive Panel Displays). The authors claim that there is a lack of understanding of what the characteristics and underlying concept of office work are. They describe office work as being carried out be office workers playing office roles. Human beings are involved in this work, and due to judgements, faults, or unexpected reactions may sometimes behave in an unpredictable or non-deterministic way. The authors' analysis shows that both routine and non-routine work is carried out in the same offices. Further results show that office work is full of exceptions, and that, in the long run, changes take place with respect to organisation structure, assignments, office roles and tasks themselves. The ECF system is designed to cope with these changes.

An ECF is the electronic equivalent of the "circulation folder", a common conventional tool for supporting the processing of office tasks. An ECF, like a real circulation folder, contains a number of task-related documents. Each procedure has an ECF associated with it which contains the documents related to that procedure. Each ECF has a *migration specification* which describes the steps that make up the procedure, what sequence they must be performed in, and the type of user (i.e. a user playing a particular role) who must perform each step.

The authors point out that both the exact steps and the role played by the user who will perform those steps are not always known in advance, and that therefore there is often a need for exception handling (i.e. deviations from predefined migration routes). In fact, there are some tasks which require completely unformalised migrations, in other words the procedure must be made up by the participants as it is carried out (with each participant indicating what the next step should be). They claim that all these problems are solved through the use of ECFs.

An ECF consists of a description and some contents. The 'description' contains :

- migration specification
- system-wide unique I.D.
- relationship to other ECF's
- state of progress
- history of all steps performed so far

The 'contents' contain :

- documents which are required in order to perform the steps of the ECF
- optional folder slip which can be edited by a worker
- optional appendix where documents can be added at worker's discretion

The migration specification defines the possible migration routes in terms of steps to be carried out, where each step has associated with it :

- the role played by the worker who is to perform the step (e.g. secretary, senior manager)
- documents which are affected
- application programs which are used

The ECF system does not concern itself with the contents or types of documents inside ECFs or with the details of the programs which process them (other than simply the identity of the program and documents).

Allowing parallel paths to use the same document is accomplished by putting a copy of a document into a ECF of its own which then migrates independently of the original.

The system allows for the late resolving of addresses, e.g. where a step has to be performed by the manager of the worker who carried out the previous step, the identity of the manager is not hard-coded into the migration specification, or even resolved as soon as the identity of the person who is to perform the previous step is resolved, but is resolved at the last possible moment, just before that step in the procedure is allocated to the manager.

The system allows for the designating of certain steps as optional or

mandatory. The system also handles exceptions, by providing commands such as :

- *Not Me* A worker may claim that he/she is not the one who has to work on the current step, so this command allows her/him to send the ECF back to the preceding worker.

- *Append* Using this command a worker can add a step after the current step. Using this facility it is possible to have a "free-style" ECF, where the ECF starts with only one step, and all the following ones are appended, whereby many kinds of unformalised tasks can be supported.

- *Refer Back* A worker may use this command if she/he needs some further information from the worker who previously handled the folder (the request for this information should be described by the user in the folder slip). By 'referring back' the ECF it is sent back to the preceding office worker.

- *Delegate* This operation appends two steps after the current one. The first step has the delegate worker carrying out the task and the second step allows the worker doing the delegating to check the work performed by the delegate. The delegating worker thus gets the results back and takes over responsibility for the folder.

The system uses an *Electronic Organisational Handbook* to store a description of the organisation which is kept separate from the migration system, which makes it easy to respond to changes of the organisation such as restructuring, changing temporary or permanent assignments, etc. This handbook keeps information about, for example :

- the office workers employed in the organisation
- the roles established in the organisation
- the organisational units and relationships between them (e.g. superiority)
- assignments of workers to roles and posts
- assignment of posts/roles/workers to work places and workstations

By using this organisational handbook, the description of the organisation is kept apart from the migration system and changes in the organisation will not adversely affect the migration of ECFs.

The similarity of the electronic circulation folder to a conventional circulation folder means that office workers familiar with circulation folders should have less difficulty coming to terms with their electronic equivalent. Of course, any limitations and disadvantages of circulation folders are inherited by ECFs.

1.2.2.1 Example ECF procedure

The procedure, a diagram of which is shown in Figure 1.2, is taken from [Karbe90a] and is as follows :

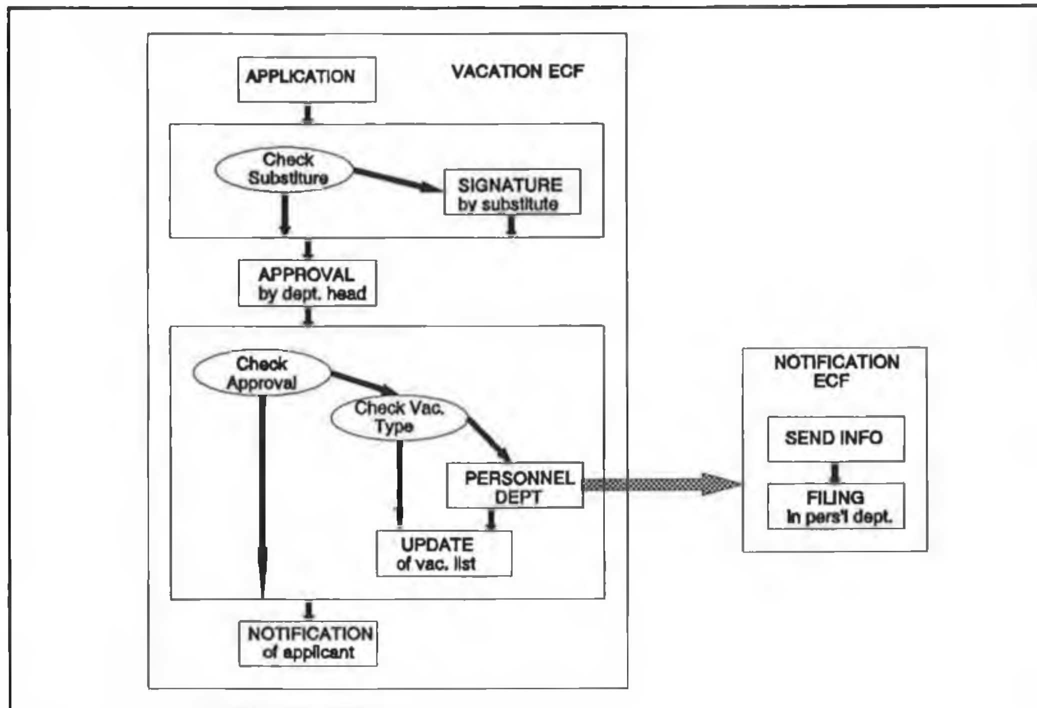

Figure 1.2 Example ECF Procedure

"An office worker applying for vacation starts this task in step APPLICATION by filling in the vacation form. In the step SIGNATURE the colleague entered as substitute confirms his taking over. Then, the head of department approves the application in step APPROVAL. The secretary will then enter the dates of the office worker's leave of absence into a vacation list during step UPDATE. Eventually, the applicant is informed of the application's success in the final step NOTIFICATION. There are some cases which have to be differentiated: if for some reason no substitute is entered the step SIGNATURE is to be skipped, if the application is not approved the step UPDATE is to be skipped too and if the vacation is of type "special leave" a copy has to be sent to the personnel department for some FILING purposes."

1.2.2.2 Comments

[Karbe90a, Karbe90b] list a number of types of exceptions (deviations from pre-specified procedures) which are supported by the ECF system. It should be possible to implement each of these deviations using a low-level model.

The concept of an electronic circulation folder, which is forwarded from user to user as each completes his/her step in the process falls down somewhat when one tries to implement parallel processing, i.e. where, say, two office workers are to be working on different, or even the same, documents simultaneously. Obviously, they can not both be in 'possession' of the circulation folder at the same time. Indeed, if they are both working with the same document (presumably, each having only read access) they can both not be is possession of the document at the same time either. This problem illustrates a disadvantage of using the electronic circulation folder metaphor - the circulation folder concept is a familiar one to users, but the electronic circulation folder inherits the limitations of the real circulation folder.

1.2.3 XCP

The XCP tool [Sluizer84] was developed by the Intelligent Systems Technologies Group of Digital Equipment Corporation. XCP supports cooperative activity by interpreting *protocols* which implement and enforce office procedures. A protocol defines the tasks of which an office procedure is comprised.

Earlier work by Zisman had claimed that when a procedure is driven by personnel reacting to a request for service, problems often arise from

not recognising the need to perform a particular task; the difficult lies in knowing when a task should be done rather than in actually doing it. An individual may be assigned to work on more than one task at a time. Such an individual must keep track of the functions and responsibilities of each task, and can be easily overwhelmed by complexity as the relationship among tasks becomes more intricate.

The authors claim that communication, whether formal or informal, is essential to the success of office procedures. They argue that people find it difficult to absorb large amounts of information, and to coordinate actions and resources to implement those decisions and that these "transaction costs" of communicating, coordinating and deciding have been enormously underestimated. As the number of employees who need to interact to get tasks done increases, these costs suffer an explosive rise.

Attempts have been made to build office tools that support procedures requiring a high degree of human involvement. This requires a tool which assists people in coordinating their actions to achieve a desired result. The authors make the assumption that the information flow and activity coordination aspects of such a procedure can be formalised and then executed by a tool.

The goal of their research was to reduce these "transaction costs" of communicating, coordinating and deciding. This is accomplished by formalising and automating protocols using the XCP model and tool.

The XCP model consists of the following fundamental concepts :

- Roles are the parts played by users of the system (e.g secretary, manager, project leader).

- An **actor** is a person who has assumed some role.

- A **document** is the symbolic representation of some paper form.

- A **protocol** is a *plan of cooperative activity* (i.e. it defines the tasks of which an office procedure is comprised). It coordinates the actions of the office staff, and supports them in carrying out the office procedure.

Protocols can be represented using graphical constructs in a similar way to the OSIRIS and ECF models.

1.2.3.1 Example XCP procedure

This procedure is taken from [Sluizer84] and a graphical representation of it is shown in Figure 1.3.

The procedure involves three roles: CLERK, ADMIN and SHIPPER. One or more users perform each role.

A CLERK creates an order (ORD in Figure 1.3), which is sent to someone in the ADMIN role (but cannot be sent to any specific person in that role). The ADMIN sends an acknowledgement (ACK) to the originating CLERK. The ADMIN then sends the order either to a specific SHIPPER (based upon criteria such as work load or specific thing ordered) or to the SHIPPER role where any person in the role can take charge of it. The SHIPPER fills the order, ships it C.O.D., and then sends the ADMIN a shipped-notice (SHP). The ADMIN then sends the originating CLERK a done-notice (DONE).

1.2.3.2 Comments

As with the OSIRIS and ECF models, XCP allows the modelling of roles played by different workers in the office, which gives the system flexibility in assigning tasks to workers (e.g. a particular task can be assigned to anyone playing the SHIPPER role).

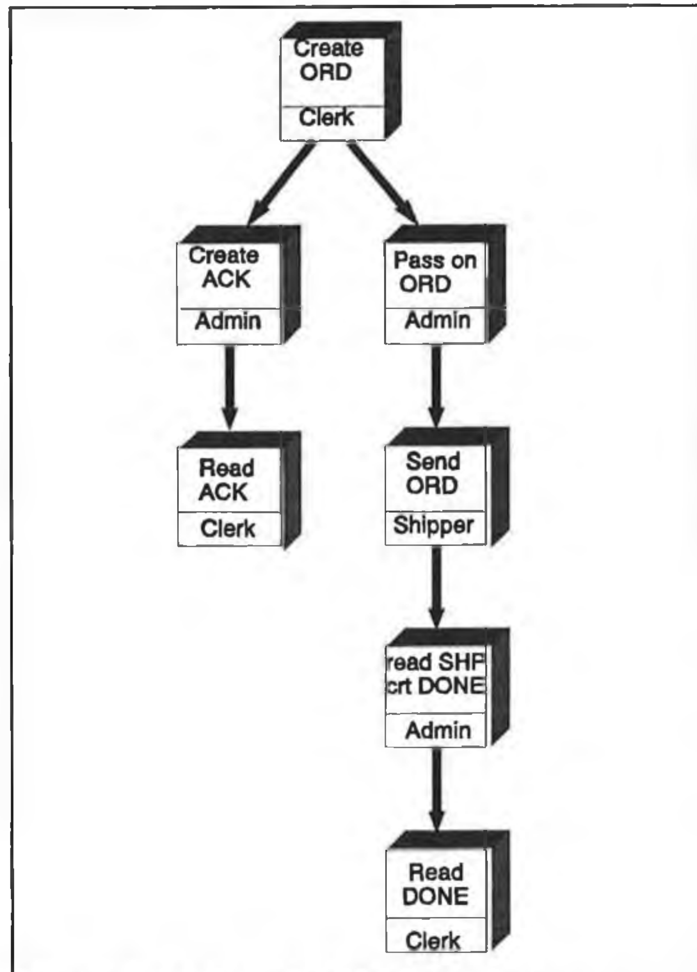

Figure 1.3 Example XCP Procedure

The authors make the assumption that it is possible to completely formalise a procedure in advance. Work done by Karbe et al [Karbe90a, Karbe90b] has shown organisational work to involve many exceptions to predefined procedures so this is an unreasonable assumption and some provision for exception handling should be provided.

1.2.4 OTM

The OTM (Office Task Manager) system [Lochovsky87,Lochovsky88] was developed at the Computer Systems Research Institute at the University of Toronto. The OTM project attempted to provide office workers with a programming language which may be used to specify tasks. To this end, a programming-by-example method was used. An underlying system

support for this task specification facility was also developed. The objective of the OTM project was to address the problem of supporting structured office tasks.

OTM task specification is based on the concept of the folder model, where an office task in constructed by assembling a folder of relevant documents and specifying the actions that are to be performed on those documents. A graphical environment, called *OfficeAid*, is provided, and a *Programming-By-Example* (PBE) method is provided for the specification of tasks by office workers. PBE specifications are translated into a lower-level model for execution. This low-level model is a concurrent, object-oriented programming language called *OTM*. Underlying this OTM language is an object-oriented database system called *OZ+*, which provides support for the storage of office data and office tasks. OTM bears some similarity to Smalltalk. The *task* concept corresponds to the *method* concept in other object-oriented languages, and each task is composed of a set of parameters, a set of temporary variables, and a (compound) statement, all of which are similar to other object-oriented languages.

A block of statements may be executed sequentially, or in parallel. Standard programming languages constructs such as **if**, **while**, **foreach** and **parforeach** (parallel form of **foreach**) are provided, which may be used to implement the graphical constructs of the PBE graph-based language.

As in the models described above, the role concept is used. A **role** object represents an entity that executes an office task, and can be either an office worker or an 'electronic substitute'. One office worker can play many roles, and one role can be played by many workers. Knowledge which is specific to a particular office worker is

encoded in a special role object called an **agent**, which is an electronic representative of that worker.

A **document** object is the "basic information carrying entity". Each document object consists of contents and behaviour. The contents contain static information, stored in fields. The behaviour of a document specifies what actions may be performed on it (like methods in an object-oriented programming language). These actions can affect only the contents of the document object for which they are invoked.

1.2.4.1 Comments

The OTM system has both a high-level (OfficeAid) and low-level (OTM) model, with consequent advantages as describe earlier in section 1.1.

But the low-level model is of a procedural nature (it is a procedural programming language) and is therefore orientated towards supporting structured procedures. It would be of limited use in supporting less structured, perhaps rule-based, procedure specifications.

The concept of roles played within an organisation are modelled in the OTM language. While the 'role' concept is a useful one to have in a high-level model, such a concept should not be included in a low-level model as it is desirable to keep a low-level PMS and an organisational database separate.

1.2.5 Augmented Petri Nets

In [Zisman78] the author claims that office procedures may be represented as systems of asynchronous, concurrent processes (each *process* being a task, perhaps complex or elementary) which may be modeled using a combination of production systems (i.e. a set of rules) and Petri nets [Reisig85]. The combination of a Petri net and a production system is called an augmented Petri net.

Previous work [Davis76] suggests that production systems (PS) are most useful in problem domains that are generally modeled by a large number of independent states, with independent actions, and where the knowledge base is best encoded declaratively as opposed to procedurally. They also suggest that a fundamental characteristic of PSs is their restriction on the interaction between rules. To produce a degree of interaction between rules requires the introduction of indirect communication through the short-term memory (STM). This results in the STM being used for both data and for complex control mechanisms.

Zisman investigates the possibility of introducing a separate explicit control structure for PSs where there is a need for substantial interaction between rules. He develops a formalism for modelling a system that is composed of a collection of asynchronous concurrent events, the particular problem domain of interest being office procedures. He is interested in modelling procedures that exist in office environments and chooses to view instances of these procedures as asynchronous concurrent processes. This is because an office can be viewed as an environment in which a large number of independent tasks are in progress and these tasks tend to be primarily event driven. Such a combination of a PS and an explicit control structure is called an augmented Petri net.

A Petri net is a directed graph which has two different types of nodes; *places* (represented by circles) and *transitions* (represented by bars). Places can hold *tokens*. Places that have arcs directed into a transition are called *input* places. Similarly, places that have arcs directed out of the transition are called *output* places. If all the input places for a transition contain a token, then the transition is said to be *active*, and may therefore *fire*. Firing involves the removal of a token from each input place and the placing of a token in each output place.

In an augmented Petri nets, each process in a system of asynchronous, concurrent processes is modeled as a set of rules. A Petri net can then be used to structure these processes by having each transition in the net represent a process. Therefore each transition in the Petri net will have a rule associated with it. The transition may fire when its input place contains a token, and the rule associated with it evaluates to true.

1.2.5.1 Example augmented Petri net procedure

[Zisman78] describes a journal editing procedure as a pair of augmented Petri nets, as shown in Figure 1.4.

Note that the rule for T2 of the *editor* net, instantiates the *referee* net for each referee selected by the editor.

The set of rules, one for each transition, are as follows :

T1: If a paper is received => send acknowledgement letter to author and request names of referees (any number) from editor.

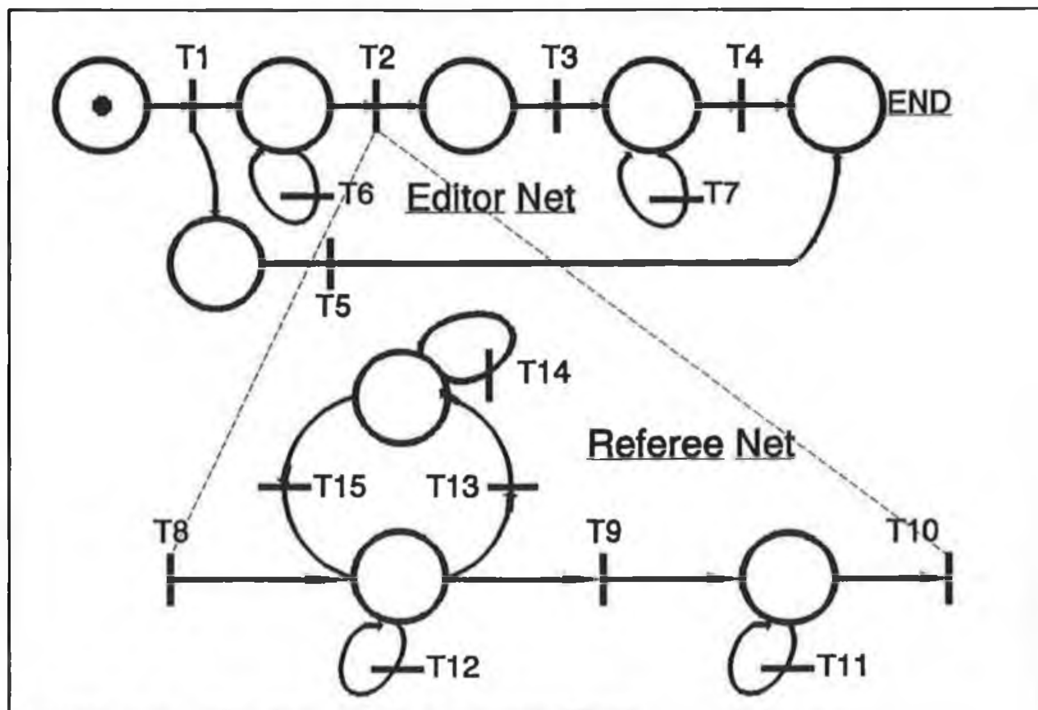

Figure 1.4 Example Petri Net Procedure

T2: If the editor supplies names of referees => instantiate the referee process for each referee

T3: If all of the referee activities terminate (i.e. fire T10) => request that the editor make a decision.

T4: If the editor supplies a decision on the paper => generate final documentation to author and editor-in-chief.

T5: If the author withdraws the paper => instantiate termination procedure.

T6: If the editor does not respond within two weeks of T6 enabling => send reminder letter to editor.

T7: If the editor does not make a decision within two weeks from T7 enabling => send reminder letter to editor.

T8: If (null condition, fires upon instantiation) => send letter to referee requesting services.

T9: If the referee returns postcard and can review the paper, => allow one month for report.

T10: If report is received => send thank-you letter to referee.

T11: If referee does not send report within one month from enabling of T11 => send reminder to referee.

T12: If referee does not return postcard within two weeks from enabling of T12 => send reminder letter to referee.

T13: If referee returns postcard and cannot review paper => request that editor supply another referee.

T14: If editor does not respond within two weeks from enabling of T14 => send reminder letter to editor.

T15: If editor does supply referee name => send letter to referee requesting services.

1.2.5.2 Comments

Production systems have many advantages, as outlined above, for modelling office procedures. The addition of Petri nets as a control structure help to separate some of the complexity from production systems.

Augmented Petri Nets might therefore form the basis of a good low-level model. Nevertheless, it would still be possible to simplify this model further, e.g. by representing the Petri net structure itself as a set of rules. Such simplification might be desirable so that the low-level model could support as many high-level models as possible.

1.2.6 VPL

The VPL (*Visual Process Language*) model [Shepard92] was developed at the Royal Military College, Ontario, Canada. VPL is a formal programming language designed to visually represent and permit enaction of software development processes.

The author envisages a Process Programming Language (PPL) as being an essential element of the next-generation Integrated Project Support Environments (IPSE). Such a PPL would be used to create a model of the software development process being used, and must be flexible enough to model all potential processes, yet detailed enough to be of real use to the IPSE. There are a number of software process models, in fact every software developer uses a unique process (which would normally be based on, and bear much similarity to, a standard process). Therefore IPSE developers should base their environments around a general model - called a metamodel - which is capable of emulating any of these processes. A process programming language is a formal enactable metamodel of the software process. Programs written in such languages implement particular processes.

A software process model will necessarily represent activities performed completely by the computer (e.g. compiling), activities performed completely by people (e.g. creation of a modular

decomposition), and activities that are a combination of human expert decisions aided by computerised tools (e.g. text editing).

A software process model is enacted by using a mechanism to use a supplied process program to monitor the progress of the many concurrent streams of a development effort. It proposes the invocation of tools at various times, enforces the process model, gives guidance to the users, keeps management informed of the status of the various streams, and executes the completely automated activities in the process model.

A process specified using the VPL software process model is defined to be a rooted connected directed acyclic graph of nodes and edges which satisfies certain constraints, such as :

- the graph must be fully connected
- each node must be one of the 9 VPL node types listed below
- each input of a node is connected to one and only one output of another node
- there is no path from the output of any node to its input except through a branch node
- the graph must contain exactly one start node and one finish node.

Procedures may be defined in a hierarchical top-down fashion. At the top level, a procedure is called a process program. This process-program may consist of tasks (elementary actions) and procedures (each of which may contains further tasks and sub-procedures).

The VPL model is very similar to other graph-based models as described above, with the notable exception of the **Decompose/Recompose** and **Split/Merge** constructs.

The 9 VPL node types are :

- A **Start** node is the entry point for objects from outside a process program or a procedure. This is the node at which execution of the process program starts.

- When the **Finish** node is reached the process program or procedure has finished.

- A **Procedure** node is a way of representing a graph (with the same rules as a process program) that forms part of the overall program (i.e. it is a sub-procedure).

- A **Task** node represents an action performed by an automated tool or by a user using a tool (i.e. a task is an elementary activity).

- A **Branch** node causes only one of the leaving arcs to be followed.

- A **Decompose** node causes an object passing through it to emerge as a family of objects, each of which possesses some subset of the parent objects information.

- A **Recompose** node causes an family of objects passing through to emerge as a single object, which is some combination of the family of objects. (The path between a **Decompose / Recompose** pair is followed by each of the objects in the family, in

parallel, and only when they **all** complete their respective paths are combined at the **Recompose** node).

- A **Split** node creates duplicates of an input object and emits one along each output arc, so a different process is followed by each copy.

- A **Merge** node acts as a rendezvous point for the concurrent streams from the **Split** node. When a complete family of objects reaches a **Merge** node, some of them may be combined into a composite object, or perhaps one may be selected as the best and the others rejected.

Associated with each graph are two tables, which store information regarding objects and roles :

- An object represents all the artifacts associated with a currently active individual work assignment.

- A role is a label that is attached to every system user to indicate the functions that user will perform.

It is interesting to note that the paper describing the VPL software process model [Shepard92] does not seem to be based on any of the work underlying the other systems described above, despite the similarity between the models.

1.2.6.1 Comments

VPL is a good example of a process model which, though not part of office procedure modelling research, bears much similarity to the kind of models used for representing office procedures.

The **Split/Merge** and **Decompose/Recompose** operations are good example of operations which a low-level model should be able to support. In other words, it should be possible to convert these constructs into an equivalent form in the chosen low-level model.

1.3 Summary

In this Chapter the concept of a Procedure Management System (PMS) has been explained and a number of existing models / systems which allow the support and automation of procedures have been described and analysed.

The models described above contain many similarities, but some also have unique features. For example, the support for exception handling in the ECF system and the use of a high-level and low-level model in the OTM system are features that are desirable in a PMS. Augmented Petri nets are more general than the low-level model used in the OTM system and would be able to support less structured procedures than those supported by the OTM system. The similarity of the VPL software development process model to the office procedure models presented illustrates how the support and automation of procedures is not limited to office procedures.

In Chapter 2, we will examine in more detail the lessons can be learnt from these models / systems. Also, the important issues involved in

the design of the models used for representing procedures and in the design of a prototype Procedure Management System will be discussed.

Chapter 2 : Procedure Management System Design Issues

2.1 Introduction

In this chapter some important issues in the design of the PMS and particularly the models used for representing procedures, are described.

A number of guidelines are outlined which should be followed in the design of a model for representing procedures and the design of the prototype PMS.

2.2 Structured and unstructured procedures

In Chapter 1, the distinction between so-called structured and unstructured procedures was described. These are not two distinct types, but rather just two ends of a scale. They both have rules to decide what must be done next, the only difference being that "structured" procedures have quite simple rules, whereas "unstructured" procedures have more complicated rules. Unstructured procedures require a problem-solving approach, and have to be specified in terms of the goals of the procedure so that the system may determine what is the best course of action.

Implementing a system to support structured procedures would, due to their simpler nature, be less difficult than implementing a system to support more unstructured procedures.

2.3 The need for a low-level model

Just as many different high-level languages such as C and Pascal can be translated into a low-level representation (machine language) for execution, it should be possible to have one underlying representation onto which many different high-level procedure models (which would be used for the specification of procedures) may be mapped. One could have one high-level model which is orientated towards representing office procedures, another which is oriented towards supporting software development processes, and so on.

If one was not able to translate many high-level procedure specifications into one underlying model it would be necessary, for each high-level procedure specification model, to have a separate component for executing that model. By having one underlying model which all high-level models map onto, it is only necessary to implement one procedure execution component.

Of course it is then necessary to write a separate translation component (which will translate a procedure specification from the high-level model into the low-level model) for each high-level model. But the complexity of the part of a system which executes procedures would typically be greater than that of the part which translates a high-level specification into a low-level specification, so it is preferable to implement one procedure execution component and a number of translation components.

2.4 Objects and their routing

Some existing office systems, such as Lambda [Oyanagi85], are form-based, i.e. they use the concept of a *form* as their basic unit of data. Other systems, such as ECF [Karbe90a] and OTM [Lochovsky[87], make the important distinction between the routing of data from user to user for processing and the contents of the data.

While it is possible to implement many typical office procedures using forms, reliance on the form concept is limiting and it is desirable to develop systems which deal in the routing of generic objects. There is no added functionality to be gained by making the procedure execution system aware of the different types of data it is dealing with. It is therefore sufficient, and indeed desirable, to have a PMS deal simply with generic *objects* (e.g. forms, documents, graphical images) without concern for the types of those objects.

2.5 Support and automation

A distinction can be made between the support of procedures and the automation of procedures. *Support* involves assisting the user by showing him/her what is to be done and by having available all the objects necessary in order for the user to perform the task. *Automation* involves executing the task without requiring the involvement of the user.

Therefore the fundamental differences between support and automation

are that a task may only be *supported* if the program that carries out the task :

- requires user interaction

  OR

- may not be run without the user's explicit approval.

For example, if a task involves the filling in an electronic form, then this operation must be performed by the user. Or, for example, if a task involved the deletion of a number of files it may be necessary to obtain the user's approval before this action is carried out. In both of these situations it is not possible to totally automate the task.

In the case of *automation*, the program that carries out the task may be automated (i.e. invoked automatically) if :

- it does not require user interaction

  AND

- the user's explicit approval is not required to run it.

For example, if a task involves the conversion of a file from one word processor format to another, and a program which performs this conversion is available, then this action may be carried out without the user being required to oversee, or even be aware of, the operation. Or, for example, if a task only involves the creation of some new file, and not the modification or deletion of existing files,

then it may be deemed safe to allow the execution of that operation without the user's explicit approval being required.

2.6 Attributing status to objects

The facility to attribute a particular status to an object is often required in a PMS e.g. *approval*, where a form is approved by a particular user, which in a paper-based system might be implemented through a signature on the form.

One way to implement *approval* would be, when an object is to be approved, mark it as such in the PMS and do not allow its contents to be altered from then on, or if the contents are altered, remove the approved status. This would require the PMS to be aware of the status of all such objects and to keep track of their status.

A different solution is to let procedure elements attribute a status to an object by altering that object in some way. For example, one could use a digital signature method (e.g. using public key cryptography) to allow a user to put an electronic "signature" on an object in order to indicate approval.

Of these two methods for implementing the approval of objects, the second is preferable because it is independent of the PMS and allows the functionality of the PMS to be limited to only those features that are necessary.

A low-level model, therefore, does not need to contain any concept of approval, or any other status, which may be applied to a document.

2.7 Making parts of procedures optional or mandatory

There is often a need for particular steps in a procedure to be specified as, for example, optional (the step may be skipped), or mandatory (the step must always be performed). One could include in the low-level model a facility for the person defining a procedure to specify a step as optional or mandatory, and/or allow that person to include in the procedure specification a definition of the ways in which particular users may modify that procedure during execution.

The alternative approach is to allow a user who is allocated a step in a procedure to do whatever he/she wants with that step (e.g. ignore it, replace it with some other action). Of course, if the user were to replace the step with another action, that other action could only be performed on a sub-set of objects which the initial task was to be performed on (i.e. it would not be possible for the user to, say, edit a file using editor B instead of editor A unless that file had been allocated for editing in the first place).

In [Fikes80] the authors argue that the domain with which office systems must deal is open-ended and therefore a procedure which implements a task is an inadequate description of all the actions which must be done to achieve that task's goals. So, at the time the procedure is being defined, one cannot predict the range of situations that will be encountered during execution of the task. Hence, for any given procedure, situations may occur in which the procedure does not indicate what is to be done, or that which is indicated in the procedure cannot be done.

The procedure specification designed to carry out a particular task should serve only as a guide in that it indicates one way of doing the task under a particular set of assumptions. The office worker should

have the responsibility of deciding in each particular situation whether the procedure's assumptions are satisfied and whether he/she wants to carry out the task in the way specified by the procedure.

The authors therefore argue that users should be able to exercise options in carrying out their scheduled tasks. For example, users should be able to choose to :

- ignore some of the requirements of a task
- renegotiate the requirements of a task
- get someone else to do a task
- create and follow a new procedure for doing a task

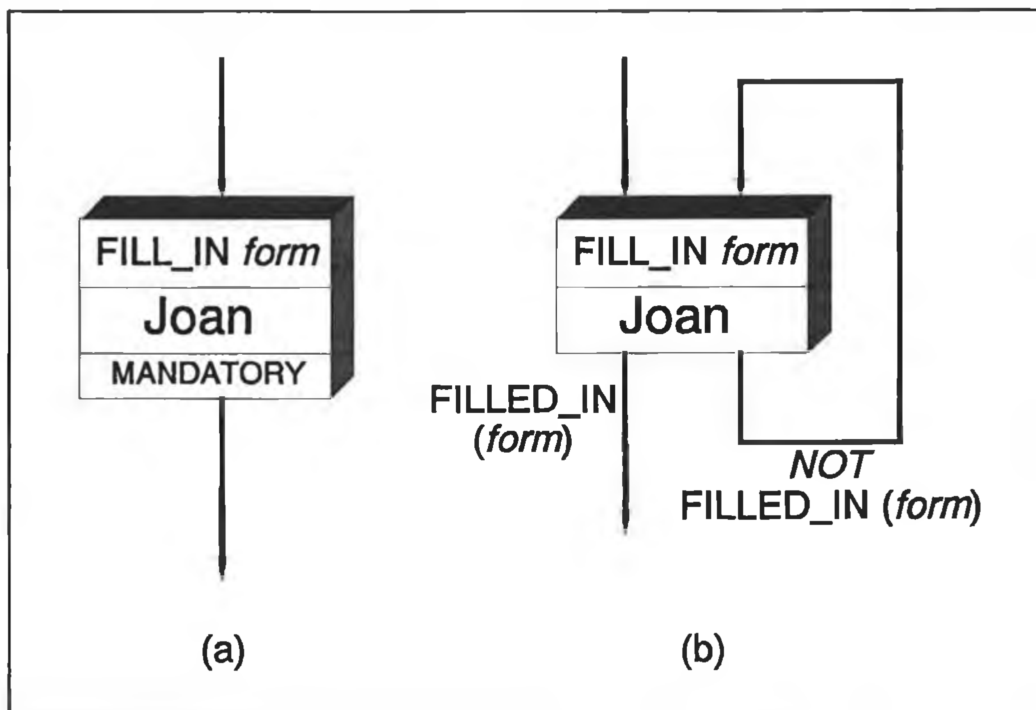

Figure 2.1 Example of a *mandatory* procedure element

It is possible to simulate a model where a step may be marked as mandatory using a model in which it is not possible to explicitly mark a step as mandatory. Figure 2.1 shows a procedure segment (a), which contains an element which is marked as *mandatory* (the procedure would be defined in terms of a high-level model which supported the concept

of a mandatory procedure element). The element consists of the task of filling out a form to be allocated to Joan (we assume that the *mandatory* tag indicates that Joan must fill in **all** the fields in the form).

This procedure segment may be translated into the procedure segment (b). In (b) the element is not marked as mandatory (so the model used to define (b) does not need to support the concept of a mandatory procedure element). The element is allocated as before, but when Joan has finished filling in the form, a check is made to ensure that all the fields in the form have been filled in. If any of the fields are empty, the task of filling in the form is re-allocated to Joan so that she may fill in those fields. Once all the fields have been filled in, the high-level *mandatory* element is complete.

This approach mirrors real life, where if you ask someone to do something, you normally do not stand over them watching them do it, but rather wait until they say they are finished and then perhaps check that they have actually carried out the requested task.

It should be noted that it is not always possible for the system to check that a task has been carried out by the user. For example, the system can determine that all the fields of a form have been filled in, but not necessarily that they have been filled in correctly. It can check, for example, that the value in a particular field is within a certain range, but not that it is the correct value. It the system could determine if the value in the field was the correct value then there would be no need for the user to have to enter that value, since the system would already know what value the field ought to contain.

2.8 Separate organisational database

The Electronic Circulation Folder system incorporates an *Electronic Organizational Handbook* which is a description of the relationships and roles within an organisation.

Such an organisational database fits into the *information sharing* category of groupware as outlined in section 1.1 and should be kept separate from a PMS. But the concept of the role played by a user is central to many PMSs, and the data relating to user roles should be stored in an organisational database. It would therefore be necessary to construct at least a simple organisational database alongside a prototype PMS.

By having a separate organisational database, the description of the organisation is kept apart from the PMS and changes in the organisation will not adversely affect the execution of procedures.

2.9 PMS design guidelines

Based on the above considerations, the design of the SEAU system and the model it uses for procedure representation should adhere to the following guidelines.

2.9.1 Support for structured procedures

Since structured procedures are easier to deal with, the first step should be to implement a system which supports their execution.

[Fikes80] asserts that systems which provide no information to the

user regarding the result expected from a step, or the use of the result, limit the ability of the user to make a wise deviation from the predefined procedure. It is therefore important that information such as the function of the element and the expected result of the element be provided to the user who is allocated an element.

Such information can be used by the user to decide whether or not to deviate from the pre-defined procedure specification. A system which supports unstructured procedures would perform this kind of deviation from the pre-defined specification automatically.

Therefore by designing a PMS which supports structured procedures and allows user discretion in the execution of those procedures, one relies on the system to blindly follow a procedure specification (something computers are good at) and the user to decide when any deviations from that specification are necessary.

2.9.2 Low-level model

The model used for the representation and execution of procedures should be simple enough to allow the mapping of many different high-level models (such as those described in Chapter 1) for representing structured procedures.

The OTM system uses a procedural programming language as a low-level model. Such a model would have limitations supporting unstructured procedures (e.g. those specified as a rule-base). The low-level model used in the SEAU system should be able to support both structured and unstructured high-level models.

The low-level model should also be able to support the exceptions to predefined procedures that the ECF model provides.

2.9.3 Separate organisational database

The prototype PMS should include an organisational database, or rather an organisational database should be developed alongside the PMS, since an organisational database ought to be a separate entity in itself. The organisational database need not be a full organisational database - the only function it has to supply, in order to support the PMS, is to allow the storage of information regarding the roles played by the users of the system.

2.9.4 User discretion

As explained in section 2.7, there is no need to include in the low-level model a concept of how much discretion a user has in the execution of a procedure element (i.e. whether the element is mandatory or optional).

2.9.5 System should deal with generic objects

As explained in section 2.4, the prototype system should simply concentrate on the 'routing' of objects, i.e. allocating objects to users so that they may perform work on those objects, and should not concern itself with what type of data those objects contain.

2.9.6 Robustness, efficiency

Since the system to be developed is only a prototype PMS, it is possible to ignore issues such as robustness and efficiency, the absence or presence of which will not affect the evaluation of the prototype system.

2.9.7 Support and automation

The distinction between the support and automation of procedures has been highlighted. The prototype PMS should certainly be able to **support** procedures, and at least have the potential to **automate** them through the addition of extra system components.

2.9.8 Multiple platforms

It should be possible to use a PMS to support/automate the procedures/policies of a complete organisation. Such organisations typically make use of many different computing platforms. It is therefore important that a PMS should not be tied down to a single operating system.

2.10 Summary

In this chapter important issues in the design of a PMS and the design of the model(s) used for representing procedures have been discussed, and a set of guidelines for the design of the SEAU system and the model used for procedure representation have been given.

Based on these guidelines, a low-level model for the representation of procedures has been developed. This model is general enough to support many different high-levels models (such as those presented in Chapter 1) for representing structured procedures, but should also be capable of supporting unstructured procedures, and allows for the modelling of generic objects. This low-level model is presented in Chapter 3.

Chapter 3 : Procedure Representation

3.1 Introduction

The model used in a PMS for the representation of procedures should be a low-level model onto which one may map different users conceptual models, which are used for the specification of procedures. Through a process of simplification, it is possible to reduce many of the features of the models described in Chapter 1 to their basic components. Thus we arrive at a low-level model which is general enough to allow most of the features of those models to be easily mapped onto it.
Described below is a rule-based low-level model which is orientated towards supporting high-level models of a structured nature, but which, due to its rule-based nature should also have the ability to support less structured high-level models. The model deals in terms of generic objects and does not concern itself with the types of those objects.

Since this low-level model is quite elementary, a program which implements it (a rule interpreter) is of little use on its own. Consequently, some of the important features of high-level models are described. These features can indicate the types of components that are needed, in addition to a rule interpreter, in order to support the execution of procedures defined using the model described below.

3.2 Low-level model

The low-level model is a rule-based model, where a procedure is specified as a set of rules whose order is unimportant except from the point of view of efficiency of execution. Or rather the order of the rules <u>ought</u> not to be important, but it is possible to construct a set of rules which, if <u>listed</u> in two different orders, will be <u>invoked</u> in two different orders, leading possibly to two different end results. One can recommend that this ought never to be the case, but one can not stop a person from designing a rule-set which does not obey this rule.

In addition to the set of rules, there exists a working memory (as in a rule-based expert system) which consists of a set of objects. (I use the term *object* not in the sense of an abstract data-type, but rather an item of data with no associated methods). The decision whether or not to invoke a rule is based on the contents of these objects, and the invocation of a rule may cause the contents of some objects to change.

Each rule has two parts :

Pre-condition

> This consists of a predicate (some function of the value of the objects in working memory) which must evaluate to **True** in order for the rule to be invoked.
>
> When a stage is reached where the pre-condition predicate of a rule evaluates to True, it does not necessarily follow that it will immediately be invoked. Since only one rule may be invoked at a time, another rule that is also

eligible for invocation may be invoked before it, and the invocation of that rule may cause the pre-condition predicate of the first rule to no longer evaluate to True (or if you prefer, the first rule missed its chance for invocation, and is no longer eligible).

Action

This consists of an action (or list of actions) to be carried out when the rule is invoked. The action *STOP* would signify that execution of the rule-set is to end.

A procedure defined as a rule-set would be executed by invoking each rule as it is eligible for invocation, until the *STOP* instruction is reached.

The alternative to terminating execution when the *STOP* instruction is reached would be not to have any *STOP* instruction and to terminate execution when a stage is reached when none of the rules are eligible to be invoked. The reason the former is chosen is so that it is possible to have a predicate in the pre-condition part of a rule which accesses an object other than one of the objects specified as one of its parameters (e.g. a program which queries an external database). Because of this requirement, it would not be possible to have the procedure finish executing when no more rules are eligible to be invoked, since the fact the no rules are eligible at one point in time does not imply that one or more of the rules will not be eligible at some later stage (e.g. when a value in an external database changes).

If predicate programs only accessed objects passed to them as parameters, then once the state was reached where no rules were

eligible to fire, it can be guaranteed that this state will not change, since a rule will only become eligible to be invoked if an object is changed in some way, and that can be only be done if a rule is invoked. If predicate programs can access external databases or files then it is necessary to have a *STOP* instruction.

3.3 Advantages/disadvantages of the low-level model

The following are advantages and disadvantages associated with the use of a rule-based model as a low-level model for the representation of procedures.

Advantages :

- The rule-based model supports the execution of many different high-level structured models, so that only one execution component is needed, rather than one for each high-level model.

- The rule-based model may potentially be suited to supporting the execution of less structured procedures (e.g. of a rule-based nature).

Disadvantages :

- It has been noted [Georgeff83] that procedural knowledge *can* be represented declaratively, but that in some domains it cannot be easily or naturally represented, e.g. in the case of a system which uses both procedural and less-structured knowledge (a *procedural expert system*) the construction of such a system can be complicated by this fact, and the explanatory capability of the system reduced.

- [Gallanti85] states that dispersing procedural knowledge into a declarative (e.g. rule-based) representation can create a heavy burden on the inference mechanism (i.e. deducing the next rule to be invoked would involve a large search process).

3.4 Post-conditions

In the low-level rule-based model described above each rule consists of a precondition and an action. It would be possible to include a post-condition in each rule, which would define what effect the invocation of that rule would have on the state of the objects associated with that procedure. The advantage of having a post-condition in each rule would be :

- It would be possible to prove certain characteristics of a rule-set (e.g. that it will always terminate).

The disadvantage would be :

- Having to include in each rule both a pre-condition and a post-condition would result in added difficulty in defining a rule-set.

3.5 Modifying executing procedures

The user who initiates a procedure should have the ability to monitor and modify that procedure during execution (in much the same way as a person debugging a program may examine and alter the values of the programs variables during execution).

The monitoring of a procedure effectively allows the user to *read* the status (i.e. the *objects*) of that procedure. Similarly, the modification of a procedure is effectively allows the user to update (or *write to*) the status of that procedure, and also allows the user to modify the procedure specification (i.e. add, delete, modify some of the rules).

A high-level model modification facility should be provided which would be used to modify executing procedures in terms of the high-level model that the user deals with. It would be the responsibility of this part of a PMS to ensure that any modifications that are made will leave both the modified procedure and the modified procedure status in a valid state (the rules for a valid state being part of the high-level model).

In a similar way, it should be possible for a set of designated users to monitor the execution of a procedure (e.g. the members of a committee should be able to monitor the execution of a procedure which was initiated by one of the members of that committee on behalf of the committee), which effectively means they should be granted *read-only* access to the status of the procedure.

3.6 Examples of high-level model features

Described below are some of the features of a high-level model. This is done in order to indicate the types of components (in addition to a rule-interpreter) that are needed to produce a practical PMS.

3.6.1 Translating a high-level model into a low-level model

Consider a model where procedures are represented as directed graphs where the nodes represent procedure *elements* (programs which must be executed by a user who plays a specific role) and the arcs represent the precedence relationship between elements. An arc may have a predicate associated with it (e.g. that a particular object contains a certain value) which must evaluate to **True** before that arc may be followed.

If an arc connects node **A** to node **B**, this means that node **B** has two preconditions that must be true before it may execute, i.e. that node **A** must have finished executing and that the predicate on the arc must be true.

In the low-level representation, each node in the high-level procedure graph is represented by a rule where the pre-condition is a predicate which may consist of one or more predicate programs ANDed, ORed, or XORed together. The action part consists of a list of programs (with parameters) to be executed.

For example, consider the graph shown in Figure 3.1.

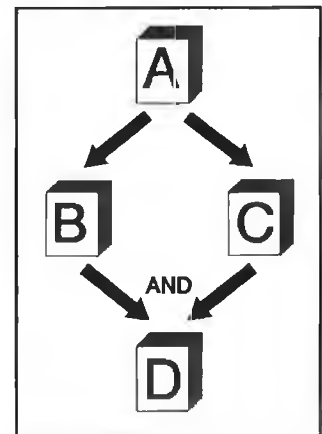

Figure 3.1

The object *Start* is created before execution of the procedure starts in order that execution will start at **A**. **B** and **C** may execute (possibly simultaneously) once **A** has completed. Similarly, **D** may execute when both **B** and **C** have finished executing.

A

Precondition	EXISTS (*Start*)
Action	DELETE (*Start*) ALLOCATE (*elementA*,*roleX*) CREATE (*AtoB*) CREATE (*AtoC*)

B

Precondition	EXISTS (*AtoB*) **and** (objects affected by *elementA* exist) **and** (predicate on A->B arc evaluates to *TRUE*)
Action	DELETE (*AtoB*) ALLOCATE (*elementB*,*roleY*) CREATE (*BtoD*)

C

Precondition	EXISTS (*AtoC*) **and** (objects affected by *elementA* exist) **and** (predicate on A->C arc evaluates to *TRUE*)
Action	DELETE (*AtoC*) ALLOCATE (*elementC*,*roleZ*) CREATE (*CtoD*)

D

Precondition	EXISTS (*BtoD*) **and** EXISTS (*CtoD*) **and** (objects affected by *elementB* exist) **and** (objects affected by *elementC* exist) **and** (predicate on B->D arc evaluates to *TRUE*) **and** (predicate on C->D arc evaluates to *TRUE*)
Action	DELETE (*BtoD*) DELETE (*CtoD*) ALLOCATE (*elementD*,*roleX*) CREATE (*DtoE*)

E

Precondition	EXISTS (*DtoE*) **and** (objects affected by *elementD* exist)
Action	*STOP*

The programs used in the rules perform the following functions :

- EXISTS (*filename*) : Returns **True** if the specified file exists, otherwise **False**.
- CREATE (*filename*) : Creates an (empty) file with the specified name.
- DELETE (*filename*) : Deletes the specified file.

- ALLOCATE (*element*,*role*) : Passes the specified element (which consists of a program with parameters) to a user playing the specified role for execution.

Note that rule **B** is eligible to be invoked when both *AtoB* exists and the objects affected by element **A** exist (we will ignore that third part of the precondition, since it is not relevant to this explanation).

For example, lets say that element **A** involves the user editing a letter. When the ALLOCATE program passes the letter to that user for editing, it deletes the *LETTER* object from the object base. The *LETTER* object will be restored to the object base when the (edited) letter is returned by the user.

The fact that *AtoB* exists (i.e. EXISTS(*AtoB*) evaluates to TRUE) implies that procedure element **A** has been allocated to a user for execution, since *AtoB* is created when A has been allocated. However, the fact that *AtoB* exists does not imply that the results of the execution of that element have been returned to that user. If both *AtoB* <u>and</u> the object(s) affected by the execution of the element exist, then element A must have been allocated <u>and</u> returned.

It should be noted that for some rule-sets when the *STOP* instruction is reached there may still be procedure elements which have been allocated to users for execution, but that have not been executed. If a rule-set was being generated by translating a high-level procedure specification, then, provided this translation was done correctly, such premature termination would not occur, since the high-level model

should not allow it. If the rule-set is being manually designed, this consideration must be borne in mind.

The above example shows how a graphical procedure involving **parallelism** can be implemented using a set of rules. It is also possible to implement other programming constructs such as **sequence**, **selection** and **iteration** in a similar manner.

3.6.2 Procedures within procedures

It is of course possible to implement a procedure within a procedure, through making the element that is allocated to a user actually the execution of a sub-procedure. Therefore everything that applies to procedure elements can also be applied to sub-procedures, since they are elements that just happen to be procedures, and are treated like any other element.

3.6.3 User discretion

When an element is allocated to a user for execution, it does not imply that the user must execute that element. The user may decide not to execute the element, or replace it with an alternative element or sub-procedure.

Allocating a single element to a user gives that user discretion over the execution of only that element. But one could allocate to a user a sub-procedure which consists of all of the elements that make up the rest of the overall procedure. This gives that user discretion over the execution of the remainder of the procedure, (e.g. the user can

add extra objects to the procedure definition, can add or remove procedure elements, etc.)

3.6.4 Concurrent access to objects

In the example shown in Figure 3.1, elements **B** and **C** execute concurrently. A number of elements that are executing simultaneously might require access (read or read/write) to the same document. The following rules might typically be imposed, in a high-level model, to cater for that situation :

- If an element is executing that has an object as a **read/write** parameter, then no element that has that object as a **read** or **read/write** parameter may start executing (since only one program should be able to write to an object at a time).

- If one or more elements that have a specific object as a **read** parameter are executing, then no element that has that object as a **read/write** parameter may start executing. Any element that has that object as a **read** parameter may start executing (since any number of programs can read the same object at the same time).

3.6.5 Roles

In the example shown in Figure 3.1, rather than directly associate a particular user with a given procedure element, we use the concept of roles, which are similar to the concept of agents in the OSIRIS model [Maiocchi87], and roles in the ECF [Karbe90a,Karbe90b], XCP [Sluizer84] and OTM [Lochovsky87,Lochovsky88] models.

Each user can play any number of roles, e.g. John can play the roles of both *Clerk* and *Secretary*. The same role may be associated with any number of users, e.g. Joan, David and Helen could all play the role of *Manager*.

In order to remove the distinction between users and roles, each user would normally be allocated the role of themselves, i.e. Mary would be allocated the role of *Mary*. Thus the system does not have to make a distinction between, for example, a procedure allocated to a specific user and a procedure allocated to any user out of those playing a particular role.

It is also possible to store the name of the role which is to execute an element in an object (rather than hard-code the rolename into the rule-set). This would allow one step of a procedure to involve writing to an object the name of the role who is to perform the next step in the procedure (e.g. the manager of the user who performed the previous step).

3.7 Summary

A low-level rule-based model for the representation of procedures which may be used as the basis for a procedure management system has been presented. Also described were some important features of high-level procedure specification models (roles, procedure elements, etc.) and we have shown how these might be implemented in a PMS.

Procedures specified using this model may be enacted using a rule-interpreter, but this alone is not enough to make a useful PMS. It is necessary to implement a number of other components in order to produce a practical PMS.

These components together make up the *SEAU* Procedure Management System. The SEAU system will be described in Chapter 4.

Chapter 4 : The SEAU System

4.1 Introduction

Described below is the *SEAU* Procedure Management System, which uses the rule-based model described in Chapter 3 as the model for the representation of procedures. The system is implemented in C on AIX 1.2 (IBM's version of the Unix operating system) using C 1.1 and also on VM/CMS 5 (an IBM mainframe operating system) using IBM C/370 and consists of approximately 3500 lines of code.

The system is named *SEAU*[1] (pronounced *so*) after the four basic components of the system. the :

- Submission
- Execution
- Allocation and
- User-interface components.

The SEAU system has been developed on both the AIX and VM/CMS operating system for the following reasons :

- Since it should be possible to use a PMS to support/automate the procedures/policies of an organisation (e.g. an insurance company, etc) and such organisations typically make use of many different computing platforms and it is therefore important that the design of the prototype PMS should not be tied down to a single operating system.

---

[1] No significance should be attributed to the fact the *seau* means *bucket* in the French language.

- The computers used in such organisations range from large mainframes to desktop computers. The two operating systems chosen reflect the diversity in the operating systems used in such organisations. VM/CMS is a mainframe operating system and AIX is a version of the Unix operating system, an operating system which is used on computers of various sizes, from desktop computers to super-computers.

4.2 System architecture

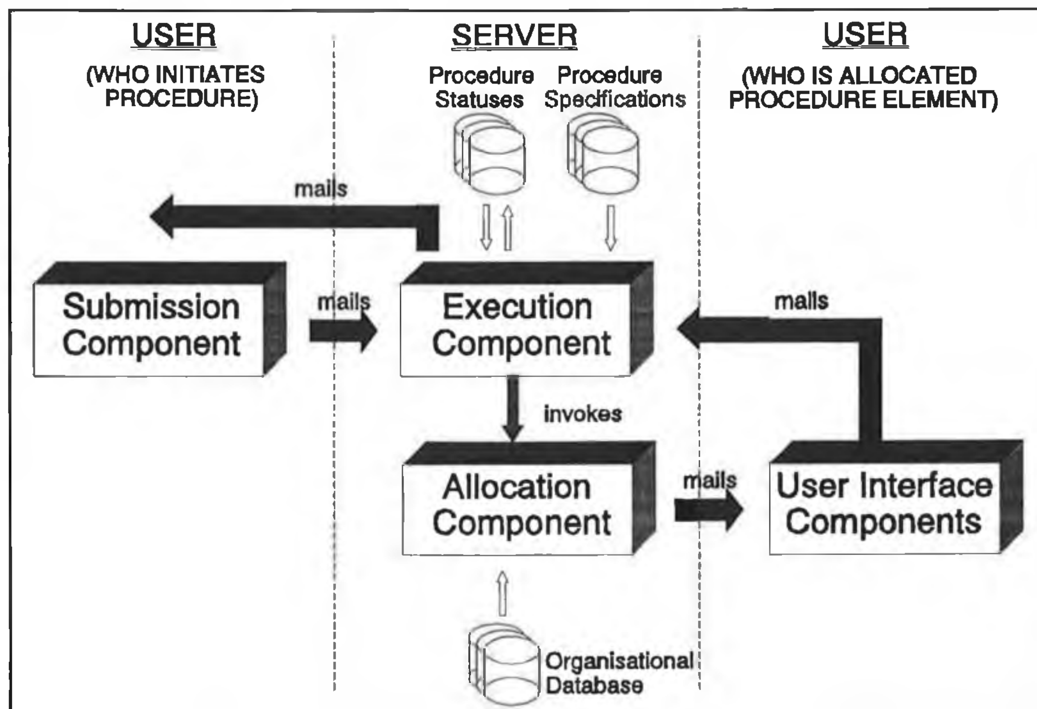

Figure 4.1 System Architecture

Figure 4.1 shows the architecture of the system, with its 4 main components. The system has been fully implemented on both AIX and VM/CMS, and operates in a similar fashion on both operating systems. The **Submission** component is used by a user to submit (via electronic mail) a rule-set (representing a procedure) to the execution component for execution. The **Execution** server receives rule-sets and executes them. At any one time, the **Execution** server may be executing a number

of procedures, each of which has a procedure definition (rule-set) and a procedure status (objects) associated with it. The execution server reads the procedure definitions and the objects, and based on the contents of each updates the objects in some way.

Rule-sets may use the **Allocation** component to send procedure elements to a user (who plays a specified role) for execution. The **Allocation** component accesses the **Organisational Database** in order to convert a role name into a user name.

The **User-Interface** components are used to receive procedure elements for execution. When a user has executed an element, he/she uses the **User-Interface** components to send the results back to the execution server. When a rule-set finishes executing, the results of the execution of the rule-set are returned to the user who originally submitted the rule-set for execution.

By splitting t%w system up into a number of separate components with well defined interfaces, the straight-forward addition of additional / alternative components is catered for.

4.2.1 Effects of having to design for AIX and VM/CMS

If the system was designed for AIX alone, one could develop an execution component which would execute a single procedure, and the user could run it as a background process. VM/CMS does not support multi-processing, therefore the execution component has to run on a separate virtual machine (or a separate user in AIX). It is not possible to arbitrarily create a new virtual machine every time a user wants a procedure executed, therefore it is necessary to have a fixed

number of virtual machines (in this case, 1) which will simultaneously execute any number of different procedures.

Since this is a prototype PMS, the simplest method of communication, which is common to both AIX and VM/CMS has been chosen, i.e. electronic mail. Procedures to be executed are submitted to the execution server using the mail system on AIX and using spool files on VM/CMS. If an organisation, though using many different types of computers and operating systems, already had an electronic mail system in place allowing communication between users on all of those systems, then such an electronic mail system could be used as the communications mechanism for a PMS.

The communication between the VM/CMS and AIX versions of the SEAU system allows for any number of AIX users to communicate with a single VM server, and therefore allows for the following scenarios :

- AIX clients communicating with an AIX server
- VM/CMS clients communicating with a VM/CMS server
- AIX clients communicating with a VM/CMS server
- AIX clients communicating with both an AIX server and also a VM/CMS server

It is unlikely that communication between a VM/CMS client and an AIX server would be of much practical use.

The development on two different operating systems resulted in a number of effects on the C code design :

- There are some very minor differences in the C compilers used and in the #include file structures.

- Differences in the operating systems meant that some C libraries could not be used because they are AIX dependent.

- The main coding problem was the fact that a Unix filename is simply a sequence of characters, whereas a CMS filename consists of 3 separate parts :

    - file name (e.g. 'PROGRAM')

    - file type (e.g. 'C')

    - file 'mode' (i.e. the disk the file is on) (e.g. 'A')

But overall, the effects on the C code design were merely inconveniencing, whereas the effect on the overall architecture of the system caused by the development on both the AIX and VM/CMS operating systems, as described above, was quite significant.

4.3 System components

4.3.1 *Execution* component (server)

The execution component uses the rule-based model described in Chapter 3, with some modifications. It runs continuously and accepts procedures for execution via electronic mail. It is capable of simultaneously executing any number of procedures - this is achieved by the server performing a primitive form of multi-tasking.

The definition of the grammar of the language used for the description of procedures is given in Appendix B.

4.3.1.1 Changes to the model

The following changes were made to the low-level model defined in Chapter 3 for implementation in the *Execution* component :

- Each of the parameters of the programs in the *Action* part of a rule must be specified as being either **read-only** or **read-write**. This means that the precondition for a rule need only be re-evaluated when one of the objects referenced by the precondition program(s) is changed, i.e. when a rule which has a read-write access to that object in its *Action* part has been invoked. (The alternative would be to re-evaluate every pre-condition of every rule after any rule has been invoked.)

- Each rule has associated with it a time-out period. If a period of time equal to this time-out period expires without the precondition of the rule being evaluated, then the precondition is evaluated, and if it evaluates to *True*, the rule becomes eligible for invocation. (This feature is included in order to support preconditions which use files other than the files specified as parameters when they are invoked.)

- The programs which make up the pre-condition part of a rule are ANDed together. ORing and XORing are not supported.

4.3.1.2 Execution strategy

For each procedure, the execution server performs the following actions :

- when the invocation of a rule is complete (and therefore the contents of some objects may have just changed), it re-evaluates the predicates of any rules which make reference to objects which the rule that had just been invoked had **read/write** access to, and it invokes any rule which is now eligible to be invoked.

- every time the time-out period for a rule expires, it re-evaluates the predicate of that rule, and if it is eligible for invocation, it is invoked.

- when the *STOP* instruction is reached, it returns the results of the execution of the procedure (i.e. all the objects passed as read/write parameters) through electronic mail to the user who originally submitted the procedure.

4.3.1.3 Additional function of execution component

The execution component is simply a rule-interpreter, with an additional feature. This feature is necessary so that the execution component can be used to execute *procedures*, which involve the allocation of tasks to users via electronic mail.

When a user finishes executing a procedure element and returns the results of the execution of the element to the server, the server must recognise the objects being returned, what procedure they relate to, and it must mark those objects as *dirty* (i.e. their contents may have

been changed) so that any preconditions which are based on those objects will be re-evaluated.

This is the only function carried out by the execution component other than simply executing rule-sets.

4.3.1.4 Re-evaluation of predicates

As soon as a rule becomes eligible for invocation, (i.e. as soon as its precondition evaluates to *True*) the execution component should recognise this and invoke the rule. The execution component could continually re-evaluate all the preconditions and whenever it finds one that evaluates to *True*, cause the action part to be followed, but this would be very inefficient. It is sufficient to only re-evaluate a precondition when an action which has one or more of the documents in that precondition as a read/write parameter finishes executing. Whenever an action finishes, it might have altered the contents of one of those documents it had read/write access to, and so all preconditions which access those objects should be re-evaluated.

It is possible for predicate programs to access files other than those passed to them as parameters (e.g. external databases). Since a value in, say, an external database, can change at any time, a facility has to be provided to allow the preconditions of rules to be re-evaluated periodically (to check for changes in such external data).

Therefore, each rule has a time period (of *N* seconds) associated with it (a value of -1 indicating that the precondition need not be periodically re-evaluated). Once a period of time equal to the specified time period has passed since the last evaluation of the precondition, the precondition will be automatically re-evaluated.

4.3.1.5 Precondition programs

Precondition programs are executed by the execution component. The execution component has access to some standard precondition programs (such as EXISTS(*filename*) in the example in Chapter 3) but other precondition programs may be, for example, written by the user and reside in the users directory, and the execution server may not be able to execute them. Therefore when the user is submitting a procedure for execution, she/he has two options :

> The user may submit the program as a read-only parameter of the procedure. The execution component will store the program as an object and can execute it when required. When the procedure finishes, the program will be deleted (because it is a read-only parameter).
>
> The user may specify in the procedure definition file the full path of the precondition program, so that the execution component knows where to find it. The precondition program's access privileges would of course have to be set so that the execution component could execute it. (Note that this second option is only possible if the execution server can access the AIX filesystem where the precondition program is stored)

4.3.2 *Submit* component

Usage : **submit** *serverfile procdeffile paramfile1 paramfile2 ... paramfileN*

The *Submit* component takes as its parameters :

- a file containing the address of the execution server (i.e. where the procedure definition and parameter file are to be sent)
- the name of the file containing the procedure definition which is to be executed
- the name of the files which are parameters of the procedure (there should be one filename for each parameter in the procedure definition)

It performs the simple task of combining the procedure definition file and the parameter files into a single large file, and sends that large file to the execution server at the specified address.

4.3.3 *Allocate* component

Usage : **Allocate** *rolename programname parameter1 parameter2 ... parameterN* | *Comment*

The allocation component simply passes the name of a program to be executed, and the contents of the files to be used as parameters to that program, to a user who plays the specified role.

A comment, which describes the function of the procedure element, may be placed after the "|" character, and may consist of any number of words. Through this facility, information may be provided to the user regarding which function the element is to perform, the result that is expected from that element, the use of the result, etc.

If more than one user plays the specified role it is simply sent to the first user in the list of users who play that role. A more

sophisticated system could be used, e.g. to send it to the user, out of those who play the role, who has the smallest current workload.

If a parameter is preceded by the percentage character (%), then that parameter is a *read/write* parameter, otherwise it is a *read* parameter. Only those files that have a % preceding them (i.e. only those files whose contents might have changed) will be returned to the execution server when the program has been executed.

The allocation component actually consists of a number of small programs. A translation program has been written which takes a rule-set file and replaces any invocations of the *Allocate* program with the appropriate invocations of these small programs which carry out the allocation. The reason that the allocation component is composed of a number of smaller programs is so that, if one wants to write an alternative allocation component, one may re-use any of these smaller programs. For example, An alternative version of the *Allocate* program has been written which instead of requiring a rolename as its first parameter, requires the name of a file which contains the name of the role who is to execute the program. This means that roles do not have to hard-coded into procedure definitions.

4.3.4 User Interface components

The following components provide the interface between users (who execute elements of procedures) and the server (which controls the allocation to users of the elements that make up procedures).

The *List*, *Do* and *Finish* components have the following architecture.

Each is separated into a *front-end* and one or more *back-ends*. Each different role played by the user may have a different set of 3 back-ends (one for each of *List*, *Do* and *Finish*) associated with it (although one back-end set may be used for any number of different roles). Each different back-end set could implement a different method of queuing procedure elements for execution.

Only one set of back-ends has been written for these components, but it would be easy to develop other sets of back-ends which would implement alternative queuing methods (e.g. in order of priority, etc.).

When a user is allocated a procedure element, he/she should be free to do whatever he/she wants with that element (i.e. do nothing and simply return the files unchanged, use a different program on the files, etc.). The *Do* component, as it currently stands, does not provide a facility for the user to use, say, an alternative program to the one indicated.

The user-interface components as a whole, while not facilitating a user who wishes to perform a different action to that specified, do not prevent the user from doing this (i.e. the ability to alter the specified action could be incorporated in the *Do* component, or in an alternative *Do* component).

4.3.4.1 ***Receive*** component

Usage : **Receive**

This program is run to receive a file sent by the execution component from mail (in AIX) / reader (in VM/CMS).

If the file contains a procedure element which is to be executed then the *Receive* program will add that procedure element to the list of elements to be performed. If the file contains the results of a finished procedure, then it is extracted from the mail into a file which may later be unpacked into its constituent *object* files.

4.3.4.2 *List* component

Usage : List *rolename*

List, when provided with the name of a role played by the user, will provide a list of procedure elements to be executed by that user in that role. Listed with each element will be a number which may be used with the *Do* component to specify which element is to be executed. Also listed with each element is a comment on the function of the program (as specified in the *Allocate* program).

This program invokes the *List* 'back-end' component which is appropriate for the role specified.

4.3.4.3 *Do* component

Usage : Do *rolename element_number*

Do, when provided with the name of a role played by the user and the number of a element to be executed in that role, will invoke the appropriate *Do* 'back-end' for that role which will cause the specified element to be executed.

4.3.4.4 *Finish* component

Usage : **Finish** *rolename element_number [file]*

Finish, when provided with the name of a role played by the user and the number of a element to be executed in that role, will invoke the appropriate *Finish* 'back-end' for that role which will cause the results of the execution (i.e. all the object files marked as *read/write*) of the specified element to be returned to the execution component.

The optional *file* parameter is the name of a file which contains the results of the execution of a sub-procedure. Its use is explained below in section 4.6.

4.3.5 Organisational database

As described in Chapter 2, an organisational database ought to be a separate entity in itself, rather than part of a PMS. For a PMS which deals with the roles played by users, which ought to be stored in an organisational database, it is necessary to construct a simple organisational database in order to store data relating to users and the roles they play.

The organisational database simply consists of a list of records, each containing the following fields :

- role name
- user name
- node name

Each record gives the name of role and the user name and node name of a person who plays that role. The organisational database could of course be expanded to include such data as relationships between roles (e.g. role1 'is the manager of' role2).

4.4 Limitations of the *SEAU* system

The *SEAU* system lacks the following components, which would be desirable in a comprehensive Procedure Management System.

4.4.1 High-level procedure specification component

A high-level procedure specification component might perhaps be :

- simply an editor, if a traditional programming language is used to specify procedures in the high-level model.

- a graphical procedure editor, if a graphical language is used to specify procedures.

4.4.2 High-level procedure translation component

A high-level procedure translation component would convert the procedure specification in the high-level form into a rule-set which could be submitted to the execution component for execution.

4.4.3 High-level procedure monitoring component

As described in Chapter 3, a monitoring component would allow the user (or a set of users) to monitor the execution of a procedure - typically in terms of a particular high-level model, though it could allow the monitoring of the procedure in terms of the low-level model. It would also allow the user to modify the status of the procedure (i.e. modify the contents of objects related to the procedure) and/or modify the procedure definition itself.

To describe how a monitoring component would work, one may use an analogy with debugging a program written in a conventional high-level programming language. If one wants to monitor the execution of a program written in a language such as C or Pascal, one uses a debugger.

The debugger makes use of :

- source code
- status of executing assembly language program (in assembly language terms)
- symbol table

The symbol table allows the debugger to relate what it sees happening at the assembly language (low) level with the program written in the high-level programming language, and to show the programmer what is happening, not in terms of the <u>low-level</u> model (assembly language), but in terms of the <u>high-level</u> model (high-level programming language).

Similarly, if one was to implement a procedure monitoring component in a PMS, that component would make use of :

- the high-level procedure description
- ability to query status of executing procedure (in terms of the low-level model)
- an equivalent to a *symbol table*

Since the ability to query the status of an executing procedure is required, the execution component would have to be extended to provide this facility.

For each different high-level procedure specification language, one would need a separate monitoring component (in addition to separate specification and translation components) which would allow the monitoring and modification of the procedure in terms of that high-level model.

4.4.4 Procedure automation component

The system as it currently stands, supports the execution of procedures, as opposed to automating their execution. In order to achieve automation, it would be necessary for each user to have a program which would continuously monitor incoming mail for elements being assigned to that user and which would execute some or all of these procedure elements and return the results.

Only procedure elements which do not require user interaction may be executed in this way or to be more precise, a program which requires user interaction cannot be successfully completed in this way. Such a program could be started by the server, but once user interaction was required the user must take over, at least temporarily.

There might only be a particular type of procedure element which the user wants to be executed automatically on her/his behalf. For example, a user would not want a procedure element to be executed automatically if the function of that procedure element was to delete all the files belonging to the user executing it ! So there would have to be some way for users to maintain, for example, a list of programs which may be executed automatically, without their explicit approval (or perhaps a set of predicates defining what programs may be run with what parameters).

A server program which automatically executed particular procedure elements would be straight-forward to implement on AIX, but on VM/CMS where a user may have only one process running at a time, it would only be possible to have the server running when the user is not using her/his *virtual machine*.

Such a server program would operate completely independently of the rest of the system (execution server, allocation programs, etc.) so there would be no need for the rest of the system to know of the existence of such a server.

4.5 Application architecture

Programs may be used with the SEAU PMS as either precondition programs, action programs, or procedure elements.

4.5.1 Precondition program architecture

A program which is to be used in the *Precondition* part of a rule must satisfy the following conditions :

- It must return either 1 (*True*) or 0 (*False*) depending on the values of the parameters passed to it.

- It must not attempt to alter the contents of any objects.

4.5.2 Action program architecture

A program which is to be used in the *Action* part of a rule must satisfy the following conditions :

- It must return 0 upon successful completion, 1 otherwise (in order that an unsuccessful execution can be detected by the execution component).

- It must not attempt to alter the contents of any objects that it is supposed to read but not write.

4.5.3 Procedure element architecture

A program which is to be used as a procedure element must satisfy the following conditions :

- It must return 0 upon successful completion, 1 otherwise (so that if execution is un-successful, the *Do* component will recognise this and may take appropriate action).

- It should not attempt to alter the contents of any objects that it is supposed to read but not write.

4.6 Procedures within procedures

In order to implement a procedure within a procedure, an element must be allocated to a user, the element consisting of an invocation of the procedure submission component to initiate that sub-procedure. The user must then wait until that sub-procedure terminates, before returning the results of that sub-procedure (which is an element of the larger procedure).

The sub-procedure may take many days/weeks to execute and that user might be executing many different procedures (which are sub-procedures of larger procedures) at the one time. So when a sub-procedure does terminate and the results of its execution are returned to the user, the user has to know which procedure element in that users list of *'elements to be executed'* corresponds to the terminated sub-procedure, so that she/he may return the results of that sub-procedure as the results of the appropriate procedure element.

It is possible for the user to match a field in the sub-procedures results file with a field in the corresponding procedure element (i.e. the element that consisted of the submission of the sub-procedure). The user may then issue the *Finish* command, with the name of the file containing the results of the sub-procedure as an extra parameter. The results of the sub-procedure will automatically be unpacked and returned to the execution server which is executing the outer procedure.

4.7 Summary

The architecture of the *SEAU* Procedure Management System, and the individual components which make up this system, have been described. The components that are missing from the *SEAU* system (high-level specification, translation and monitoring components and a procedure automation component) have been listed. Since the focus of my research is to investigate the low-level representation of procedures, these facilities have not been provided in the prototype system. The criteria that must be conformed to by programs which are to be used with the *SEAU* system have been given, and the way in which the system assists in the execution of sub-procedures has been described.

The *SEAU* system uses a low-level model, which is designed for the **representation** and not the **specification** of procedures. A high-level model such as one of those presented in Chapter 1 should be used for procedure specification, and the procedure then translated into the low-level model for execution. In Chapter 5 the use of the SEAU system to implement example high-level procedures given for some of the systems described in Chapter 1 will be examined.

Chapter 5 : The Application of the SEAU System

5.1 Introduction

In this chapter, the application of the SEAU system, and its rule-based model to the implementation of some high-level procedures will be examined. These procedures are example procedures given for some of the systems described in Chapter 1, and are defined in terms of the procedure specification models defined for these systems. By examining the implementation of these example procedures, it is hoped to show that procedures defined using a number of different high-level models may be implemented using the *SEAU* system and its rule-based model, and to describe any difficulties involved in their implementation.

5.2 Implementing high-level procedures using the low-level model

For each of the following high-level models, the implementation of the example procedure as described in the paper which describes the model is examined.

5.2.1 OSIRIS

The OSIRIS model contains a set of the features which may be easily mapped onto the rule-based low-level model used in the *SEAU* system. It was therefore possible to implement the example procedure [Maiocchi87] described in Section 1.2.1.1 with little difficulty.

In order to implement the OSIRIS procedure in terms of the low-level model, it was necessary to write a program which implemented the

precondition which checked whether the reply-analysis outcome was positive or negative, and also to write programs to implement each of the procedure elements.

The translation of the procedure graph into a set of rules was straight-forward, with each node in the graph being represented by a single rule. The precondition of such a rule would simply check that all the rules relating to the nodes directly preceding it in the graph had been invoked, and also that any other preconditions (other than precedence conditions) for the node (e.g. that a file contained a particular string) were true.

5.2.2 XCP

The example given for the XCP system [Sluizer84], and shown in Section 1.2.3.1, is quite a simple one, as are the high-level model features it requires. For example, none of the arcs in the example procedure graph have predicates associated with them, therefore the preconditions of the rules only have to implement the precedence relationships. The process of translating the XCP protocol into a rule-set that could be executed using the *SEAU* system was therefore even more straight-forward than with the OSIRIS model.

One potential problem with the implementation of the example procedure is as follows. The procedure definition states that when Admin is sending an acknowledgement (ACK) to the Clerk who sent the order (ORD), Admin should send it to the clerk who sent the order, rather than just any clerk.

The *Allocate* component <u>always</u> allocates a procedure element to the **first** user in the list of those performing the role, but only for the

sake of simplicity. In a typical non-prototype PMS the *Allocate* component might not act so predictably. It might allocate the element to the user with, say, the smallest current workload. The workloads of users would change over time, and therefore the identity of the user with the smallest workload would also change over time.

So as the system currently stands, the acknowledgement would be returned to the same Clerk (provided the list of users playing the Clerk role was not changed during execution of the procedure), but this might not always be true.

It would be possible to force the system to send the ACK to the same clerk who sent the order by writing an alternative *Allocate* component which stored, in an object, the identity of the user who sent the order, so that the name of this user could be later retrieved and the ACK could be sent directly to him/her.

5.2.3 Electronic Circulation Folders

As with the OSIRIS model, the Electronic Circulation Folder [Karbe90a, Karbe90b] contains a set of the features which may easily be mapped onto the low-level model, and the example procedure taken from [Karbe90a] and described in Section 1.2.2.1 was straight-forward to implement as a set of rules. As with the OSIRIS example, it was necessary to write a number of predicate programs and element programs.

Examples of exceptions to the given procedure, which the ECF system can handle, are given in [Karbe90b] and the difficulty involved in supporting each of these exceptions using the *SEAU* system is analysed below.

<u>As long as the folder is not forwarded an office worker may wish to revise work on a step.</u>

In the *SEAU* system a user who is allocated an element (in the form of a program with parameters) may re-execute that program any number of times before returning the results to the server.

<u>After work on a step is finished an office worker may wish to get the folder back for some updates.</u>

In theory it is possible, though the *SEAU* system does not provide that facility, to modify a procedure specification as it is executing (by adding, changing, deleting rules). If this feature was available a user could contact the user who initiated, and is therefore in overall control of, the procedure, and can re-organise the procedure as it is running so as to insert an extra procedure element further along in the procedure which consisted of the user being allocated whatever objects he/she was interested in order to update them.

Or this may also be done by contacting a user further along in the chain of those performing the procedure who would be able to allow the first user to do some work on some objects before passing them on to the next user in the chain, (but in that

case, it is possible that the second user might not have been allocated the objects which the first user is interested in).

<u>An office worker may wish to interrupt work and put the folder on the pile.</u>

This is possible through the feature of the AIX C-Shell which allows the suspension of a process (i.e. an executing procedure element) and it's later resumption.

<u>In order to get his manager involved, the applicant may forward the form and a note to him and ask him to continue normal processing.</u>

The *SEAU* system allows the user to replace a procedure element allocated to her/him with any other program, or indeed, a sub-procedure. To add an extra element after the current one, a user may replace the current element with a sub-procedure which consist of the following two elements :

> The first element would be a copy of the element as originally allocated to the user.
>
> The second element, which will be executed after the first element, is an element which allocates the appropriate documents to the manager for viewing, approval or whatever.

Once this sub-procedure has finished executing, the procedure will continue as normal.

The applicant may wish to add appendices to the folder's content in order to give more evidence in support of the application.

If the applicant has only been allocated a single element for execution, then it is not possible for him/her to add extra objects to the set of objects worked on by the procedure.

But if the applicant is allocated a sub-procedure for execution, it is possible for her/him to change that procedure specification before submitting it for execution. The applicant can therefore add any object he/she wants to the set of objects worked on by the procedure.

The applicant may decide to cancel the vacation at any time after having finished the step "Application".

Though the *SEAU* system does not allow it, it would be possible to included a facility whereby the user who initiated a procedure may terminate that procedure at any time.

Therefore, if the applicant submitted the procedure for execution, he/she could terminate its execution at any time. If the applicant did not initiate the procedure, then he/she may have it terminated, by contacting the user who did initiate the procedure and asking her/him to do so.

The substitute selected by the applicant may refuse to take over. Thus, he sends the folder back possibly with a slip on it giving some information.

This may be accomplished, as with one of the exceptions above, by the substitute replacing the procedure element allocated to her/him with a sub-procedure which consists of an element which allocates an element to the applicant, which requires her/him to read the slip. The applicant can then examine the slip and decide what to do (e.g. he/she can replace the element allocated to him/her with any other element or sub-procedure).

The head of the department may be on a business trip. Thus, the step 'Approval' should be performed by his substitute.

The procedure would be constructed so that when it comes to the step where the form has to be approved by the head of the department, the organisational database is checked to see who is the head of the department in which the applicant works. The approval step would then be allocated to this person.

The head of the department, before going on the trip, would indicate in the organisational database that another user was to be her/his substitute until her/his return. Thus when the organisational database is checked to see who the head of the department is, it should return the name of the substitute, who will then be allocated the 'Approval' element.

The head of the department may want to make his decision dependent on the opinion of the manager of the project team of which the applicant is a member. To that end, he may forward this question to the project manager with a request for an answer.

The head may do this by replacing the element allocated to him/her with a sub-procedure, the first element of which allocates the appropriate objects to the project team manager in order to get his/her opinion. The second element would then allocate the appropriate objects and the project team managers answer to the department head for her/him to make the final decision.

The head of the department may want to defer the decision. Therefore, he postpones the work for a later resubmission.

In the *SEAU* system, the department head would simply leave the element in her/his list of elements waiting to be performed until he was ready to execute it.

The *List* component could be enhanced, for example, so that, if a user issued a 'defer' command in relation to an element, it would hide that element from view from a user for a certain length of time.

In order to inform on the success of the application the office worker making the application may forward a copy of the finished form to the substitute.

This could, of course, be done by simply using electronic mail to mail the file to the substitute. It could also be done through the *SEAU* system by replacing the element allocated to the applicant with a sub-procedure which would consist of an element which would allocate a copy of the finished form to the substitute for viewing, filing, etc.

It has been shown that it is possible to implement the above exceptions using the low-level rule-based model, and using the *SEAU* system (or, in a few cases, a slightly enhanced version of the *SEAU* system).

5.2.4 Augmented Petri Nets

Since each Petri net transition has a rule associated with it, the process of converting the procedure specified as an augmented Petri net, and shown in Section 1.2.5.1, into a set of rules is very simple indeed.

A number of the transition rules involve re-sending a letter if a reply is not received within certain period of time. The *SEAU* system does not inherently support this kind of time-out feature, but it is quite straight-forward to implement it. All that is required is to record (in an object) the time at which a rule (e.g. send letter to referee) was invoked. Then create another rule whose precondition is that a certain amount of time has passed since the time specified in

that object (this rule would be periodically re-evaluated). As soon as the difference between the current time and the time specified in the rule is greater than a certain amount, that rule will be invoked to send a reminder to the referee.

The example procedure contains some rules that are troublesome to implement in the *SEAU* system. An example of this is the rule which says "*if, at any time, an author withdraws a paper then end the procedure*", or in general, any rule that waits for an event which may happen at any time, but might never happen at all.

In order to cater for this, it is necessary to allocate a procedure element to a user which states "*if at any time the following event occurs, 'Finish' this element*" (which will result in the output of the element being returned to the server). The server would then be able to respond to the occurrence of the event. Note that the periodic predicate re-evaluation feature of the SEAU system would be required in order to allow such a rule to be invoked, as soon as (or rather, relatively soon after) the result of the execution of the element is returned to the server.

Although this approach would work, it is an awkward way of supporting this requirement, since if the event never occurs, the user is left with a procedure element that never needs to be executed.

If one ignores this sort of situation (i.e. one where the user may or may not have to execute a procedure element allocated to her/him), it is possible for the user to think of the elements allocated to him/her as a 'to-do' list (i.e. a list of tasks for the user to perform). But in this type of situation, this 'to-do' list would contain a task which might never need to be performed and would just sit there (either forever, or until the user, or the system, explicitly removes

it), which means that it is no longer strictly a 'to-do' list (i.e. all the items in the list do not have to be performed).

In order to implement this "*if, at any time, an author withdraws the paper then end the procedure*" rule, it is necessary to create a rule which says "*if return received from 'wait for event' element, then end procedure*". The ending of the procedure would then involve sending a notification to users who have been allocated elements that they no longer need to perform those elements. It would be preferable if the elements allocated to those users could be withdrawn automatically.

Some of the rules in the example procedure have predicates that require user-interaction (i.e. the predicate program would not be able to work out for itself whether something is true or false, but would have to ask a user). Since the *SEAU* execution component requires predicate programs to be devoid of user interaction, any arcs in a high-level model, for example, which have predicates which require user interaction have to be implemented by converting the single arc into two rules, the first of which would allocate the task of deciding on the value of the predicate to a user and the second of which would proceed based on the answer given by that user.

While it is possible to have a lower-level model than this model (consisting of rules only), it would appear that there is no advantage to be gained from this, since a model consisting of Petri nets and rules ought to be able to do everything a model consisting simply of rules can do (since if one were to decide not to use the Petri net aspect of the model, they are equivalent). Indeed, it should therefore be possible to model the example procedures given above using the Petri net / rule-based model.

The question then arises of whether one should draw the line at a combination of Petri nets and rules, or whether one should include another formalism in the model to make it yet more versatile. Of course, the more complicated the model gets, the more complicated the system which must execute that model becomes.

The Petri net can be used to indicate which rules (out of a potentially large set of rules) in a rule-set may be currently eligible to be invoked. In a very similar, but less explicit way, the SEAU system only checks a small sub-set of rules after each rule invocation, since (ignoring the periodic rule evaluation feature) it only re-evaluates the precondition of a rule when the contents of one (or more) of the objects accessed by the precondition of that rule are potentially altered.

Consider a typical rule-based representation of a graph-based procedure, such as that given in Chapter 3. In it objects are used to represent the precedence relationships between procedure elements. When a rule is invoked, both the values of the objects involved in the execution of the element and objects representing precedence rules change. Any rule precondition which makes reference to any of those objects will then be re-evaluated.

Imagine that the procedure consists of three elements; A followed by B followed by C. The B rule may be invoked when the object AtoB exists and some other condition (based on the contents of some other object called X) is true. Similarly, the C rule may be invoked when the object BtoC exists and another condition (based on the object X) is true. When the A element executes the value of X changes and AtoB is created. Since the preconditions of both B and C depend on the contents of object X they will both be re-evaluated, although C cannot proceed since BtoC does not yet exist.

If a Petri net was used to model the precedence relationships between elements, the evaluation of the C precondition would not be performed, since the Petri net would clearly indicate that C was not yet ready to be executed. Time would therefore not be spent evaluating the part of the precondition of C involving the object X.

Therefore the use of a Petri net to represent the precedence rules, though adding complexity to the procedure execution component, might lead to increased efficiency of execution.

As the procedures one is trying to represent become less structured, the importance of the Petri net diminishes and the importance of the rules increases, since the rules involved in less structured procedures deal with rather more complicated relationships than simple precedence. However, for a system which is to support quite structured procedures, a combined Petri net / rule-based model might be the most suitable.

5.2.5 VPL

The VPL model is a graph-based model and contains similar constructs to those found in models such as OSIRIS, ECF, and XCP.

However, it does contain two special constructs which are of particular use in software process modelling - **Decompose/Recompose** and **Split/Merge**. Describe below are methods which allow these constructs to be dealt with in the *SEAU* system.

5.2.5.1 Decompose / Recompose

An object entering a **Decompose** node is split up, in some way, into a family of objects and these 'children' proceed in parallel along the same process. These 'children' later enter a **Recompose** node and are combined in some way to produce a single object.

A procedure which is bounded by a **Decompose/Recompose** pair may be implemented by having the decomposition and recomposition carried out by a specialised program. A procedure element would consist of this program, which would :

- split the object up into a number of separate objects
- initiate the sub-procedure for each of these objects
- when all the sub-procedures finish, combine the objects in some way to form a new object
- return the new object

The above technique only applies to <u>procedures</u> bounded by a **Decompose/Recompose** pair. If it is a <u>set of tasks</u> (or even one task) which is bounded by the **Decompose/Recompose** pair, then a sub-procedure may be created which consists of that set of tasks and the above technique applied.

5.2.5.2 Split / Merge

A **Split** node creates duplicates of an object, each of which follows a different process. At the corresponding **Merge** node, these objects are combined in some way to produce a single object.

A procedure (or set of tasks, as above) bounded by a **Split/Merge** pair may be implemented in a similar way to a **Decompose/Recompose** pair, i.e. a procedure element would consist of a specialised program which would :

- make a number of copies of the object
- initiate a sub-procedure for each of these copies
- when all the sub-procedures finish, combine the objects in some way to form a new object
- return the new object

So there should be little difficulty implementing a VPL process using the *SEAU* model. Of course, just because there is no difficulty implementing a VPL model using the rule-based model used by the *SEAU* system does not necessarily mean that the architecture of the *SEAU* system is suitable for supporting the software development process. Typical software processes are liable to be of a different nature to, say, office processes and it might be possible that the nature of the *SEAU* system's architecture would not suit the support of software development processes.

5.3 Summary

In this chapter the implementation of example procedures, defined using a number of different high-level models, on the *SEAU* system has been examined. It has been shown that many of the features of a number of high-level models may be implemented using the SEAU system and it's associated low-level model. In addition, difficulties involved in the representation of high-level concepts using the low-level have been described.

It has been explained how some of the features of these models may be implemented using the low-level model described in Chapter 3, and features of the example procedures which caused some difficulty during implementation have been highlighted.

In Chapter 6, overall conclusions reached as a result of this research, and in particular the analysis of the SEAU low-level model described in this chapter, will be presented.

Chapter 6 : Conclusions

6.1 Introduction

In this final chapter, the conclusions reached as a result of this research and some possible future research directions following on from this research are outlined.

6.2 Conclusions

Listed below are some general conclusions reached as a result of this research.

6.2.1 Structured procedures

The SEAU system supports the execution of structured procedures. It also allows users discretion over the execution of the procedures they are allocated.

This means that it is possible for the PMS to follow a structured procedure specification, and the user may introduce deviations from that procedure specification. Clearly, it would be desirable for the PMS to deduce for itself when deviations from pre-defined procedures are required, but for procedures which are in the most part structured it is adequate to rely on the user to introduce such deviations when they are needed.

6.2.2 Procedure representation

The rule-based model presented in Chapter 3, and used in the SEAU system is general enough to allow the implementation of the features of a number of different high-level models, such as those described in Chapter 1. Through the use of the *SEAU* system to implement various example high-level procedures it has been shown that the low-level rule-based model used is simple enough to allow the features of a number of high-level models to be straight-forwardly implemented.

The low-level model used by the OTM system is a procedural programming language. It is doubtful whether such a language would be able to facilitate the implementation of "unstructured" procedures. The SEAU low-level model, or the Augmented Petri Nets model, provides a more suitable base for representing both structured and unstructured procedures.

The use of augmented Petri nets (Petri nets in conjunction with rules) [Zisman78] adds extra complexity to a system for managing the execution of procedure, but does have the advantage of providing some structure to the rules, with a consequent increase in the efficiency of procedure execution. As the procedures one is trying to represent become less structured the importance of the Petri net diminishes and the importance of the rules increases. However, for a system which is to support quite structured procedures, a combined Petri net / rule-based model might be the most suitable.

As noted in Chapter 1, the paper describing the VPL software process model and the system which implements it [Shepard92] does not appear to be based on any of the work that the other systems described are based on. This could possibly be taken as an indication of a lack of overlap between research into the representation of office procedures

and software development processes, where one would expect some overlap, especially considering the similarity between the VPL model and some of the office procedure models described.

6.2.3 Workflow management is only an application of a PMS

Some existing systems are known as workflow management systems. The term *procedure management system* was chosen to describe the *SEAU* system. The term *workflow management system* seems to imply the use of the system for controlling the flow of work of users. But workflow management (or office procedure management or business process management) is only an application of a procedure management system, the term *procedure* being used in this case to describe a set of steps designed to achieve some goal.

6.2.4 Separate organisational database

While the presence of an organisational database is required for a PMS, it should not be part of the PMS, but rather a separate system in itself, which would have many other uses than simply those required by a PMS.

6.2.5 PMS should ignore the contents of objects

A PMS does not need to, and should not, concern itself with what types of objects it is routing from user to user.

6.2.6 Degrees of procedure support / automation

The components that make up a procedure management system can be classified into 4 layers which provide different degrees of support/automation.

- Use of electronic objects (e.g. files) rather than real objects (e.g. pieces of paper), and tools which work with those electronic objects, with manual routing of objects via electronic mail (i.e. no procedure management system).

- Use of Allocate and User-Interface components to allow users to allocate tasks involving electronic objects to other users.

- Use of Execution/Submission components, in addition to all of the above, to support the execution of procedures.

- Use of Procedure Automation component, in addition to all of the above, to automate the execution of procedures.

6.2.7 Procedure types

A Procedure Management System is required to support a wide range of procedure types.

At one end of the scale is a procedure, fully specified in advance, each step of which must be carried out precisely as specified (i.e. no deviations or exceptions are allowed).

At the other end of the scale is a kind of 'make it up as you go along' procedure. Such a procedure might contain only one element when it is started. When the user who executes that element finishes doing so, she/he may add another element to the procedure and specify who is to perform it. And so it may continue, in this fashion, each user involved in the procedure adding an extra element to the procedure after his/her element. Eventually, some user will choose not to add an extra step to the procedure and it will finish.

The use of a Procedure Management System to aid the execution of the first type of procedure is quite similar the use of a conventional multiple-user information system as used in, say, an office environment, which indicates to users what processing they must perform.

And the use of a Procedure Management System to aid the execution of the second type of procedure is quite similar to the approach that would be taken if a Procedure Management System did not exist (i.e. each user simply forwarding all the relevant objects through electronic mail to the next user in the chain, with a request to carry out a particular action on those documents and forward them to a another user).

A Procedure Management System should allow the support of both of these types of work, rather than being orientated towards supporting one at the expense of the other.

6.2.8 Allocation and submission components

The *Submit* component is used to submit a procedure to the server for execution. The *Allocate* program is used to allocate a procedure element to a user for execution.

It would be possible to implement a program which would run continuously on behalf of a user and which would automatically execute some of the procedure elements allocated to that user (without the user's interaction required) and return the result to the execution server. Such a program would be very similar to the procedure execution server, which accepts a procedure for execution, executes it, and then returns the result of the execution of the procedure to the user who submitted that procedure for execution.

Since these programs perform very similar functions, it would seem suitable to combine them into a single program (perhaps called *allocate*) which would allocate a task (whether a whole procedure or just a procedure element - procedures and procedure elements are treated in the same way using the SEAU system) to a "user" (whether a human user, or a server acting on behalf of a human user) for execution. This would eliminate the unnecessary distinction between procedure and procedure elements, and between procedures/procedure elements carried out by users and those carried out automatically by a program.

6.3 Future work

Outlined below are some possible future work which could follow on from the work presented in this thesis, including some ways in which the *SEAU* Procedure Management System could be enhanced.

6.3.1 Robustness, efficiency

Attributes such as robustness, efficiency, etc. were not addressed during the design of the prototype system, so these are clearly areas in which the SEAU system could be improved.

6.3.2 Additional components

Component could be added to the SEAU system to perform the following functions :

- high-level specification of procedures
- translation of high-level procedures into low-level representation
- monitoring of executing procedures (high-level and/or low-level)
- element automation

In addition, the use of objects which encapsulate both data and methods (as in the OTM system [Lochovsky87, Lochovsky88]) would be another way of enhancing the *SEAU* system.

6.3.3 Complete groupware system

As explained in Chapter 1, a procedure management system fits into the workflow category of groupware. The work presented in this thesis is concerned with the support the flow of work. As described in Chapter 1, a complimentary type of groupware is the information sharing type.

One could implement an information sharing system, which along side the *SEAU* PMS would make a comprehensive *groupware* system.

6.4 Overall summary

A low-level model for procedure representation has been developed, which has been shown to support structured high-level models. Due to its rule-based, non-procedural nature, the low-level model should also be able to support less structured procedures.

A prototype Procedure Management System, the SEAU system, has been implemented on two different platforms and has been used to experiment with the low-level representation of procedures defined using a number of existing high-level models. Issues which arise from the implementation of these procedures have been examined and resulting conclusions presented. The model used in the SEAU system has been shown to be a suitable model for the low-level representation of procedures.

Appendix A : References

[Beslmuller88] E. Beslmuller, "Office Modelling Based on Petri Nets", *Esprit '88 - Putting the Technology to Use, Part 2 (Proceedings of the 5th Annual ESPRIT Conference)*, pp. 977-987, 1988.

[Bracchi84] G. Bracchi and B. Pernici, "SOS: A conceptual model of office information systems", *Data Base*, vol. 15, no. 2, pp. 149-168, 1984.

[Cortese84] G. Cortese and F. Sirovich, "A Daemon Based Programming System for Office Procedures", *Proceedings of the ACM SIGOA Conference*, 1984.

[Croft84] W. B. Croft and L. S. Lefkowitz, "Task Support in an Office System", *Proceedings of the ACM SIGOA Conference*, 1984.

[Davis76] R. Davis and J. King, "An overview of production systems", *Machine Intelligence*, vol. 8, 1976.

[Ellis79] C. A. Ellis, "Information Control Nets - A Mathematical model of office information flow", *Proceedings of the ACM Conference on Simulation, Measurement, and Modelling of Computer Systems*, pp. 225-239, Bondler, Colorado, 1979.

[Ellis85] C. A. Ellis, "Office Information Systems Overview", *Languages for Automation*, ed. S. K. Chang, pp. 3-26, 1985.

[Ellis91] C. A. Ellis, S. J. Gibbs and G. L. Rein, "Groupware: Some Issues and Experiences", *Communications of the ACM*, vol. 34, no. 1, January 1991.

[Fikes80] R. E. Fikes and D. A. Henderson, Jr., "On Supporting the Use of Procedures in Office Work", *Proceedings of the First International Conference on Artificial Intelligence*, pp. 202-207, 1980.

[Fisher87] W. Fisher and J. Gilbert, "FileNet: A Distributed System Supporting Workflo; A Flexible Office Procedures Control Language", *Proceedings IEEE Computer Society Office Automation Symposium*, pp. 226-233, 1987.

[Gallanti85] M. Gallanti, G. Guida, L. Spampinato, and A. Stefanini, "Representing Procedural Knowledge in Expert Systems: An Application to Process Control", *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, vol. 1, pp. 345-352, 1985.

[Georgeff83] M. Georgeff and U. Bonollo, "Procedural Expert Systems", *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, vol. 1, pp. 151-157, 1983.

[Gunther85] K. D. Gunther, "Logic Programming Tailored For Office Procedure Automation", *Languages for Automation*, ed. S. K. Chang, pp. 27-66, 1985.

[Kishimoto85] K. Kishimoto, K. Onaga, and H. Utsunomiya, "OPAL: An OPA Language for LAN Environments via Active Mailing and Program Dispatching", *Languages for Automation*, ed. S. K. Chang, pp. 67-94, 1985.

[Oyanagi85] S. Oyanagi, H. Sakai, T. Tanaka, S. Fujita, and A. Tanaka, "A Form-Based Language for Office Automation", *Languages for Automation*, ed. S. K. Chang, pp. 107-122, 1985.

[Karbe90a] B. H. Karbe and N. G. Ramsperger, "Influence of Exception Handling on the Support of Cooperative Office Work", *Proceedings of the ACM SIGOIS Conference on Office Information Systems 1990*, pp. 2-15, 1990.

[Karbe90b] B. Karbe, N. Ramsperger and P. Weiss, "Support of Cooperative Work by Electronic Circulation Folders", *Proceedings of the ACM Conference on Office Information Systems '90*, pp. 109-117, 1990.

[Lochovsky87] F. H. Lochovsky, "Managing Office Tasks", *Proceedings IEEE Computer Society Office Automation Symposium*, pp. 247-249, 1987.

[Lochovsky88] F. H. Lochovsky, J. S. Hogg, S. P. Weiser, and A. O. Meldelzon, "OTM: Specifying Office Tasks", *Proceedings of the ACM Conference on Office Information Systems '88*, ed. R. B. Allen, pp. 46-54, 1988.

[Maiocchi87] R. Maiocchi and B. Pernici, "Verification and Refinement of Office Procedures", *Proceedings IEEE Computer Society Office Automation Symposium*, pp. 206-216, 1987.

[Mazer87] M. S. Mazer, "Exploring the Use of Distributed Problem Solving in Office Support Systems", *Proceedings IEEE Computer Society Office Automation Symposium*, pp. 217-225, 1987.

[Oyanagi85] S. Oyanagi, H. Sakai, T. Tanaka, S. Fujita, and A. Tanaka, "A Form-Based Language for Office Automation", *Languages for Automation*, ed. S. K. Chang, pp. 107-122, 1985.

[Reisig85] W. Reisig, *Petri Nets : an Introduction*, 1985.

[Shepard92] T. Shepard, C. Wortley, and S. Sibbald, "A Visual Software Process Language", *Communications of the ACM*, vol. 35, no. 4, pp. 37-44, April 1992.

[Sluizer84] S. Sluizer and P. M. Cashman, "XCP: An Experimental Tool for Supporting Office Procedures", *Proceedings IEEE First International Conference on Office Automation*, pp. 73-80, 1984.

[Zisman78] M. D. Zisman, "Use of Production Systems for Modelling Asynchronous, Concurrent Processes", *Pattern-Directed Inference Systems*, ed. D. A. Waterman and F. Hayes-Roth, pp. 53-68, 1978.

Appendix B : Rule Definition Language

There follows a grammar, in extended Backus-Naur form, which defines the rule language used in the *SEAU* system to define procedures.

<u>Note :</u>

[ ] = optional (0 or 1 occurrence)

{ } = 0 or more occurrences

( ) indicates precedence

| means OR

procdef :- *#PROCEDURE*
procedure_name
#READ
{ variable }
#WRITE
{ variable }
#LOCAL
{ variable }
#END
rule { rule }
#ENDRULES

procedure_name :- alphanumeric {alphanumeric}

variable :- filename <NewLine>

```
             rule :-  #RULE rule_period [ rule_name ]
                      precondition { precondition }
                      #ACTION
                      action { action }
                      #END

      rule_period :-  [ - ] digit {digit}

        rule_name :-  alphanumeric {alphanumeric}

     precondition :-  program_name  {  read_parameter  }
                      <NewLine>

           action :-  program_name  {  (read_parameter  |
                      readwrite_parameter) } <NewLine>

     program_name :-  filename

   read_parameter :-  % filename

readwrite_parameter :-  & filename

     alphanumeric :-  A | B | ... | Y | Z | 0 | 1 | ... | 9
```

filename is operating system dependent.