205

SPECIFICATION AND CONTROL OF ROUTING AND SYNCHRONIZATION REQUIREMENTS OF OFFICE FORMS USING PETRI NETS

by

Charles W. Miller

B.S., University of Texas at Austin, 1979

A Master's Report

submitted in partial fulfillment of the

requirements for the degree

Master Of Science

Department of Computer Science

Kansas State University
Manhattan, Kansas

1987

Approved by:

R A McBride

Major Professor

LD
.668
.R4
CMSC
1987
M54
c. 2

A11207 303599

CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Overview 1
1.2 Scope 3
1.3 Justification 3
1.4 Report Organization 4

CHAPTER 2 REVIEW OF THE LITERATURE 5
2.1 Distributed Systems 5
2.2 Synchronization Requirements 6
2.3 Message Management Systems 8
2.4 Petri Nets 9
2.5 Survey of Related Work 14
2.5.1 Specifying and Controlling Routing And Synchronization of Office Forms With Predicate Path Expressions . 14
2.5.2 Execution Mechanisms For Jobs in a Distributed System 19
2.5.3 Logical Routing Specification in Office Information Systems 25
2.5.4 Modeling Jobs in a Distributed System 31
2.5.5 A System For Managing Structured Messages 35

CHAPTER 3 DESIGN SPECIFICATION 39
3.1 Design Objectives 39
3.2 Structure of the Design Specification. . 39
3.3 The Petri Net Specification for the Routing and Synchronization of an Office Form 40
3.4 The Control Net 47
3.4.1 Process Table 47
3.4.2 Transition Input Map 49
3.4.3 Transition Output Map 51
3.5 The Form Definition 52
3.6 The Run Time Design 52
3.6.1 Run Time Data Structures 64
3.6.1.1 Structured Mail Messages . 65
3.6.1.2 Incoming Mailbox 65
3.6.1.3 Sequence Counter 65
3.6.1.4 Form Instance Control Table 65
3.6.2 Run Time Processes 68
3.6.2.1 Form Request Process . . . 68
3.6.2.2 UNIX™ Mail Process . . . 69

3.6.2.3 Form Service Processes . . 69
3.6.2.4 The Controller Process . . 70
3.7 Summary 75

CHAPTER 4 CONCLUSION 76
4.1 Extensions of This Work 76
4.2 Concluding Remarks 77

REFERENCES 78

LIST OF FIGURES

Figure 2-1 Petri Net Structure 11

Figure 2-2(a) Petri Net Before Firing 11

Figure 2-2(b) Petri Net After Firing 11

Figure 3-1 Specification of Predicate Conditions . 44

Figure 3-2 Logical View of Net From Figure 3-1 . . . 45

Figure 3-3 Naming Convention For Nodes 45

Figure 3-4(a) Sequential Transitions 48

Figure 3-4(b) Concurrent Transitions 48

Figure 3-4(c) Decision Construct 48

Figure 3-4(d) Synchronizing Predicate 48

Figure 3-4(e) Halting Transition 48

Figure 3-5 Process Table 50

Figure 3-6 Transition Input Map 50

Figure 3-7 Distributed Evaluation Example 55

Figure 3-8 Distributed Evaluation Solution 56

Figure 3-9 A Net Structured To Benefit From A Hybrid Evaluation Scheme 61

Figure 3-10 Form Instance Control Table 66

Figure 3-11 Firing Table 74

To my wife, Doreen

I wish to thank my major professor, Dr. Richard A. McBride,
for his valuable suggestions and guidance of this endeavor.

CHAPTER 1
INTRODUCTION

1.1 Overview

This paper describes a system for specifying and controlling the routing and synchronization of an automated office form. An office form can be viewed as a preprinted document on which white space has been left for the insertion of requested information, that is, the form's data values. An office can be thought of as a distributed system in which work on a form progresses by having it flow between procedures that operate on its data [McBr85]. These procedures can operate sequentially or concurrently; they can be distributed or centralized. By viewing a form as a set of data values plus a set of operations that can be performed on these values, we see that a form is very similar to an abstract data type [Geha82]. In fact, the electronic form can be represented as an abstract data type containing both data and control information [McBr85]. This control information contains, among other things, the routing and synchronization requirements of the form.

The routing specification of a form identifies all procedures that can operate on the form. The routing can be

conditional or unconditional, depending on such things as the form's current data content or values of globally available variables such as the system clock. In essence, the route is determined from conditions generated as the form progresses through the system. The synchronization specification of a form identifies the allowable sequences of procedures which can operate on the form. The goal of synchronization is to prevent interference among procedures which share data objects.

The particular approach taken in this paper is that the routing and synchronization requirements of a form will be specified via a Petri net. Predicate conditions [Kell76] are allowed to affect the execution of actions in the net.

The implementation of the system is based on the concept of incorporating both the form content and the routing and synchronization control information into a structured message. By using concepts presented in [Maze84] and [Tsic82], a form routing and synchronization system (based on structured messages and the interpretation of the content of the messages) can be developed and implemented. Thus, the mail system becomes the primitive communication mechanism. By allowing a controller process at each site to interpret the message, the message then becomes an

"intelligent" entity guiding itself through the system without explicit user direction.

1.2 Scope

The design presented in this paper addresses the question of routing and synchronization. The form definition, i.e., the data content of the form, and the procedures that manipulate the form's data are not addressed.

1.3 Justification

The original concept and justification for the automation of office forms is given in [Geha82].

An implementation of automated forms in an office system requires two distinct efforts. First, the electronic versions of the paper forms and the associated procedures for manipulating the content of the forms must be developed. This activity would be unique to each system implementation. Secondly, the routing and synchronization system must be developed. This activity need not be repeated each time. The routing and synchronization system, as developed in this paper, is generic in nature. It could be utilized in any office system implementation without a repeated development effort.

1.4 Report Organization

This paper is organized into four chapters. Chapter 1 is an overview and justification of the paper. Chapter 2 addresses concepts pertinent to the design presented in this document and also surveys other related articles from the literature. The detailed design of the proposed forms system is presented in Chapter 3. Chapter 4 discusses the value of this work and suggests several possible extensions.

CHAPTER 2
REVIEW OF THE LITERATURE

2.1 Distributed Systems

A distributed system is a set of loosely or tightly coupled processing elements working cooperatively and concurrently on a set of related tasks [Rama80]. Loosely coupled multiprocessing [Deit83] involves connecting multiple independent computer systems via a communication link. Each of the systems has its own operating system and storage. The systems can function independently. They can also communicate when necessary to access each other's files or to share processing tasks. A tightly coupled multiprocessing system [Deit83] has storage which is shared by the various processors and a single operating system that controls all of the processors and system resources.

In a distributed environment, the applications being performed can be geared towards the parallel execution of processes rather than the traditional serial execution. The concurrent processes may be required to synchronize their operations.

In general, a distributed system can be characterized [Fort85] by the following properties:

dispersion - there exists a physical distribution of processing resources;

interconnection - the processing resources are connected via a communication link and communication occurs by the passing of messages;

resource sharing - resources are spread throughout the system and can be utilized in some way by remote devices;

global control - there is some form of global control mechanism synchronizing the operations of the overall system.

Thus, an automated office, consisting of a group of autonomous work stations interconnected by a communication link to shared resources, is a distributed system.

2.2 Synchronization Requirements

Parallelism can be successfully introduced into the solution of a problem only when the processes involved can cooperate in the sharing of data objects and system resources. This sharing must be controlled (synchronized) in such a way as to ensure proper results.

A popular means of sychronizing accesses to shared data objects is known as mutual exclusion. The idea of mutual exclusion is that at most one process be allowed to operate on a common data item at any time. One widely used technique for implementing mutual exclusion involves building entry and exit routines around code which operates on common data. This protected code is called a critical section. The entry and exit routines allow only one process at a time to enter the critical section. Thus, at most one process can access the common data at any time.

The routing and synchronization system described in this paper allows processes to execute concurrently. However, some processes may operate on common data in the form instance. Consequently, the access of these processes to the form instance must be synchronized. Mutual exclusion of conflicting processes is used as the means of synchronization. The Petri net specification is the mechanism for implementing the mutual exclusion. The Petri net will be constructed in a manner such that two conflicting processes cannot execute concurrently.

It is important to note that the term synchronization, in the context of this document, refers only to the control of processes operating on a particular form instance.

2.3 Message Management Systems

Traditionally, the purpose of message systems is to allow users to send and receive messages. The messages remain uninterpreted by the system. Today, office systems require that the computer take a more active role in controlling and coordinating office procedures. Therefore, an enhanced message system is required; a message management system that not only delivers messages but also manages them. [Tsic82] suggests that in order to enhance their functionality, message systems have to interpret, at least partially, the messages which they handle. This capability can be provided by superimposing some structure on the messages. This structure can then be defined to the system and used for the interpretation of messages. Messages can be structurally typed [Tsic82]. The message type describes the message structure from which instances of the type are created. This structure then guides the interpretation of each message instance.

In a traditional message system, the user explicitly specifies the destination for every message sent. The

routing is single hop - the user specifies the single most immediate destination from which the next user must specify the next destination, and so on. A message management system supports messages that can effect their own processing. By associating routing specifications [Maze84] with message types, the system assumes the responsibility both for evaluating the current message instance to yield the next destination and for forwarding the message instance. The user is no longer required to explicitly direct each instance of a message type.

The routing and synchronization system described in this paper uses the concept of attaching routing specifications to message types. This allows automatic routings of message instances. The emphasis is on messages as independent, "intelligent" entities guiding themselves through the system.

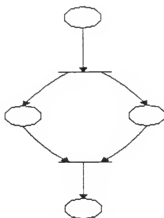
2.4 Petri Nets

A Petri net is an abstract model of information flow. Petri net models were originally intended [Pete81] as a means for a natural representation of the interaction, logical sequence and synchronization among the elementary activities into which the operation of a system or the execution of a procedure can be divided. Petri nets are

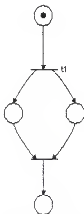
frequently used to model systems of events in which it is possible for some eventa to occur concurrently but there are constaints on the concurrence or precedence of other events.

A Petri net (Figure 2-1) is represented by a bipartite, directed graph which consists of two types of nodes: circles (called places) and bars (called transitions). Places correspond to conditions and transitions to events in the system being modeled. These nodes are connected by directed arcs from places to transitions and from transitions to places. An arc from a place to a transition identifies that place as an input place to the transition. An arc from a transition to a place identifies that place as an output place of the transition.

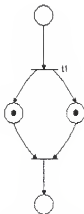
The Petri net graph models the static properties of a system. In addition to these static properties, a Petri net has dynamic properties that result from its execution. The execution of a Petri net is controlled by the position of tokens (markers) in the Petri net. Tokens reside in the places of the net and are represented by small solid dots within the circles representing places. A Petri net containing tokens is called a marked Petri net. The execution of a Petri net is reflected by the movement of



Petri Net Structure
Figure 2-1



Petri Net Before Firing t1
Figure 2-2(a)



Petri Net After Firing t1
Figure 2-2(b)

tokens caused by the firing of transitions according to the following rules:

- a transition must be enabled in order to fire,
- a transition is enabled when all of its input places contain a token,
- the transition fires by removing a token from each of its input places, and placing a new token on each of its output places.

Figures 2-2(a) and 2-2(b) show the results of firing a transition. A Petri net marking is defined to be the distribution of tokens in the net. The state of a Petri net is defined by its marking. Note that the firing of a Petri net results in a new marking.

When transitions in a Petri net represent events in a real system, an additional result of a transition firing is a change to the program variables which were acted upon by the process represented by the transition. R. M. Keller [Kell76] has extended the basic notion of Petri nets to include predicate conditions. Keller associates a unary predicate and a function defined on the program variables with each transition. He then requires that the predicate be true in order for the transition to fire.

Several terms relating to Petri nets that are relevant to this paper are defined here.

1) When there is a choice as to which of several transitions will fire next, and the firing of a transition disables another, the transistions are said to be in "conflict" [Pete77]. The decision as to which transition fires is made in some nondeterministic manner.
2) Petri nets which are constructed such that no more than one token can ever reside in any one place of the net at the same time are called "safe" [Pete77] nets (i.e., a place is either marked or it is unmarked). This stucture is sometimes referred to as a Condition/Event System [Reis85].
3) A Petri net is properly terminating [Rama80] if it always terminates in a well defined manner such that no tokens are left in the net. Note that this definition is not universal. [Pete81] refers to a properly terminating Petri net as one which has one token remaining upon termination and that token is in a "final" place.

In the design presented in this paper, Petri nets are used to specify the flow of a form through a distributed office system, and also to specify the synchronization requirements of procedures which operate on the form.

2.5 Survey Of Related Work

Several articles from the literature which are pertinent to the work presented in this document have been reviewed and are presented in the following sections.

2.5.1 Specifying and Controlling Routing and Synchronization Of Office Forms With Predicate Path Expressions [VanD86].

This paper describes a system designed to manage the process and synchronization of forms in an automated office. The concept of the automated form is reviewed with emphasis on the fact that an office form can be represented as an abstract data type containing both data and control information. The control information specifies the operations that can be performed on the form's data along with sequencing requirements of the operations. The control information is utilized to guide the form through the distributed office environment. It is stated that the routing and synchronization of an automated form should be dynamic, depending on conditions existing at the time of processing.

VanDusen, in this paper, develops a system to specify and control the routing and synchronization of automated forms in a distributed office environment. His design includes three major components. They are :

1) a specification language based on predicate path expressions;
2) a controller process which interprets the specification language and synchronizes the various processes that act on the form; and
3) the service processes that operate on the form's data.

The specification language allows the operations involved in the processing of a form to be identified using a predicate path expression. The operations consist of the name of the service process to be executed. By utilizing a subset of predicate path operators, the routing and sychronization of the service processes can be specified. The following operators are utilized in VanDusen's design:

sequencing (specified by ";") - used to indicate serial operations;

selection (specified by "+") - used to demarcate alternative procedures, exactly one of which is to be chosen;

parallel execution (specified by ",") - used to

indicate parallel operations;

predicates (specified by "[P]") - each procedure can be associated with a predicate condition.

To allow for parsing, the predicate path expression is enclosed within the keywords PATH and END. Thus, a valid routing and synchronization specification might be expressed as:

PATH A;(B,C,D);(E[passed] + F[failed]) END.

This specifies operation A followed by B, C and D in parallel, followed by E or F depending on the evaluation of the predicate. Although the predicate expressions in VanDusen's prototype are limited to one particular field's value, he states that it is desirable to expand the predicate to allow a boolean expression on any combination of data items in the form, or any global data such as date, time, node name, etc..

A copy of the controller process resides at each node in the system. When a form instance is initialized, the controller process accepts the form with an embedded predicate path specification and creates two separate entities, a control table and the form data. The control table is built through the parsing process and represents the routing and synchronization requirements as specified in the predicate path expression. Each controller process

has access to a control table for all instances of all forms which it is currently processing. The controller evaluates the form's control table and data in order to make routing and synchronization decisions. The controller creates and destroys temporary form copies as necessary for processing. For example, if the next operation on the routing exists on another node, the executing controller must forward a copy of the form's data and control information to the remote controller, while at the same time coordinating the overall process. Temporary form copies are also used to allow multiple operations to occur concurrently.

The controller process also acts as the interface to the service processes which manipulate the form's data. This is accomplished by making the form's data as well as appropriate control information available to the service process. The service process manipulates the form's data and returns changed data and control information to the controller process. The changed data might include data objects associated with the predicate values and hence the continued processing of the form by the controller process is dynamically determined. VanDusen's prototype is restricted to one service process, UNIX[1] mail. In an actual

1 UNIX is a Trademark of AT&T Bell Laboratories

implementation, many service processes would be developed to interact with the controller process. These service processes would do such things as split a form into sibling forms, merge multiple copies of a form, and other specialized processing tasks unique to each form type.

VanDusen's work is similar to the work presented herein. The basic design components are the same (i.e., a controller process at each node and form service processes). The method of implementation is the essential difference. VanDusen's design is based upon a specification language which utilizes predicate path expressions. The design described herein uses Petri nets as the building block. Also, this paper expands upon VanDusen's work in the area of predicates. Whereas, VanDusen allows only one field to determine the predicate value, the design described herein will allow for an arbitrary number of fields to be evaluated. This paper will also expand upon the service processes which are allowed. VanDusen's prototype design is limited to one service process, UNIX™ mail. The design described in this paper will allow for an expanded set of service processes to be available. UNIX™ mail will continue to be used as the transport mechanism for communications between controller processes. Another significant difference is the inclusion in this design of a

single source library for office form definitions. An Office Systems Administrator will coordinate the creation, modification and deletion of form definitions. New or changed form definitions will be provided to each applicable site as appropriate. In VanDusen's design, the form's source remained at the node where it was initially developed and the entire routing specification was included in each message representing a form instance. The design presented herein allows for a copy of the form's definition (which includes the routing specification for the form type) to reside at each appropriate site. Instead of passing the entire routing specification as part of the message, only status information will be passed. The controller process will use the status information contained in the message, in conjunction with the copy of the form's definition which resides at the site, to evaluate the routing and synchronization requirements for each form instance.

2.5.2 Execution Mechanisms for Jobs in a Distributed System [McBr85].

This paper addresses the task of modeling the execution of jobs in a distributed system and the requirements of implementing the model. The perspective of this paper is

one of viewing the automated office form as an example of a job execution in a distributed environment. In particular, McBride views the office as a distributed system in which work on a form progresses by having some sequence of procedures operate on the form's data. He views the electronic form as an abstract data type containing both data and control information. This control information must contain, among other things, the routing and synchronization requirements of the form.

McBride then describes the use of Petri nets as a modeling tool. Keller has extended Petri nets to allow predicates to be attached to transitions [Kell76]. McBride states that this extended version of Petri nets can be used to model an automated form in a distributed office. He refers to a Petri net which is used in such a fashion as a control net.

McBride discusses three possible implementations for his control net in a distributed environment. In the first method, the control net would be created and remain at the site at which the form instance which it is controlling was created. In this case, the control net would determine the proper operations to be invoked on the form instance. The control net would then send the form instance to the node which provides the first operation. After receiving the

form instance back from the first operation, the control net would determine the next operation to be performed and again send the form off to the appropriate node. This process would continue until all operations had been performed. McBride notes that this method is wasteful in terms of communications processing and bandwidth. The second implementation method is to embed the control information for a form type within the procedures that operate on the form. This means that each procedure would have knowledge of where to send each processed form (i.e., the control net does not exist as a separate entity but the equivalent control information is embeded within the procedures themselves to form a pipeline). A disadvantage to this method is that the procedures are no longer generalized but instead must be specialized for each form type. In the third implementation strategy, the control net would accompany the form instance through the sytem while acting as its guide. This requires a controller process at each node to execute the control net. McBride states that this third implementation method is the most flexible and efficient, and hence the most desirable.

McBride notes that it is the responsibility of the controller processes to interpret and update the control net of a form instance to accurately reflect the processing

that has been applied to the form instance. The controller processes must cooperate to only update a form's control net in a manner consistent with the firing rules for Keller's Petri Net Model [Kell76]. McBride proposes the following steps to ensure this coordination:

1) A controller process (referred to as the Executing Controller) is sent a form instance which is to have procedures performed on it that are available at the Executing Controller's site. This is reflected in the form's control net which is marked as having tokens present in the input places for the transitions corresponding to the local procedures.
2) The Executing Controller invokes local procedures to operate on the form. A possibly modified form is returned to the Executing Controller when the local procedures are complete.
3) When the procedures corresponding to a transition have completed, a token must be removed from each of the input places. This is accomplished by the Executing Controller sending an acknowledgement message to each of the controllers which sent a form instance used as input to the transition that just fired.

4) After a transition has fired, the Executing Controller must place a token at each of the transition's output places. This causes the Executing Controller to route the form (and its control net) to distant controller(s) where procedure(s) are available to carry out the next transitions.
5) When all acknowledgements corresponding to a transition's output places have been received, the Executing Controller's locally held copy of a form is destroyed.

In addition to the above processing rules, McBride mentions several other considerations. First he notes that each form instance must be uniquely identified with a form type and sequence number. He states that for greater efficiency, concurrent processing should be allowed. This requires that each controller process be allowed to make copies (either full or partial) of a form instance. At some point, the copies of a form may be merged by a transition. All copies of the same form instance are required to have identical form type and sequence numbers. McBride also comments on the problems of delayed and duplicate messages in a distributed system. He states that by keeping a copy of the form until all acknowledgements have arrived, a controller

can try an alternate routing if a transition associated with a timeout occurs. The problem of duplicate messages can be handled by having each controller record the arrival time of each form instance.

McBride indicates that an office form can be viewed as an abstract data type containing both data and the Control Petri Net which represents both the operations required to operate on the data and the synchronization requirements of those operations. One way to implement the abstract data type's control net is with predicate path expressions [Andl79]. McBride states that predicate path expressions can provide the same information as the control net in his model. The predicate path expression operations required for this model include :

- sequencing;
- selection;
- parallel path;
- predicate.

These operations are described in Section 2.5.1 of this document in the review of [VanD86].

McBride has extended the original definition of a predicate to also include decisions based on data values contained in the form itself, and the availability of required data

files. The predicates associated with the operations in the form's predicate path expression are evaluated by the controller process as part of the determination of where to send the form next.

Several of the ideas and concepts presented in McBride's paper have been utilized in the development of the design presented herein. In particular, the concept of an automated form being a job process in a distributed environment which can be viewed as an abstract data type containing both data and control information is basic to the design presented in this paper. Likewise, the concept of a control net modeled with a Petri net is carried forth. McBride's recommendation that a controller process reside at each node to execute the control net as a form instance travels throughout the system has been followed, as has his proposed processing steps regarding firing rules. The primary difference between McBride's work and the work presented in this paper is the implementation mechanism. McBride proposed that the control net be implemented with predicate path expressions. In the design presented here, the control net is implemented as a Petri net.

2.5.3 Logical Routing Specification in Office Information Systems [Maze84].

This paper introduces both a framework and language for the specification of logical routing for messages in an office information system. By associating routing specifications with message types, the system can evaluate the routing requirements of each instance of a message type. Thus the user is freed from explicitly directing the routing of each message instance.

The authors first discuss the concept of a message management system which is an integration of computer-based message systems and database management systems. The message management system not only delivers messages but manages them as well. The messages in the system are structurally typed. A message type describes the basic message structure from which instances of the type are created. Users manipulate instances of message types. These instances are stored in a communication base which is the medium by which users communicate. This communication base may be distributed or centralized. A communication base administrator (analogous to a database administrator) is responaible for the creation, maintenance, security and integrity of the communication base.

The structure of message types is used by the system to enable manipulation of the contents and routing of

messages. A message type definition includes message fields, each of which has a type and properties; authorizations on access to fields and instances; value and action constraints on fields; and a specification of the message type's routing.

In the framework described by Mazer et. al., routings specify the next destinations for each message type according to current message and system state. For example, the routing could be dependent on such criteria as the value of data fielda in the message, system characteristics such as current load at a site, values of queries to a database, etc.. Each instance of a message type is routed according to the specification associated with that type. The emphasis is on messages aa independent, "intelligent" entities guiding themselves through the system without explicit user direction. The routing specification for a message type indicates the logical paths to be taken by the message instances. The authors describe three types of routing specifications:

- type routing is specified at message type design time and applies, in general, to all instances of the message type;
- instance routing is specified by the user at message instance creation time and applies only to

that instance;

- override routing is used for exceptional situations to temporarily alter the normal routing specification.

The authors state that a routing can be unconditional or conditional depending on various criteria such as data values in the message instance itself or values of queries to a database. The authors also allow copies of messages to be created to support concurrent activities.

Mazer et. al., next present a routing specification language which allows the users to describe to the system the routings desired for message types in the system. The language consists of the following constructs:

- SITE identifies the site and indicates whether or not it can be the source for creation of a form instance of the given type;
- TIME-CASE is used for identifying time constraints on message instance processing;
- CREATION, FIRST, SECOND, etc. identify subspecifications that apply to the corresponding visit of the message instance to the site;
- ROUTE-CASE specifies conditions that must hold true for the instance to be forwarded to the next site;

- ERROR allows for exception handling at run time;
- END-SITE delimits the site routing specification.

The authors next discuss an implementation of a prototype message management system. The prototype system includes a communication base design system through which message type definitions are designed, a user interface to the system and a routing system. The routing system includes facilities to allow the user to:

- specify routings,
- trace a message instance,
- check an instance's state in the system,
- override and edit routings.

The question of when and how message type definitions are distributed throughout the system is addressed. In general, a routing specification may either be associated with each instance as it flows through the system or be associated with each appropriate site. The first method implies that the routing specification is associated with the origin site only and that instances include the routing specification in addition to the actual message, a significant use of bandwidth. The second method inmplies increased storage requirements at each site but smaller instances flowing through the system. The authors state that since bandwidth is generally more precious than

secondary storage, the latter method of binding routing specifications statically to sites (as part of message type definitions) is more appealing. Mazer et. al., also address the question of routing evaluation. They discuss three possible methods for the evaluation of routing specifications for a message instance. Central evaluation requires a central authority to evaluate the routing after each site completes its processing of the instance. This involves much wasted communication to and from the central station. The advantage is that only one copy of the evaluation software need be maintained. Origin evaluation involves the originating site making all routing decisions at the time of instance creation. This is not feasible if dynamic routing decisions are desired. Distributed evaluation requires that each site, after completing its processing, evaluate the routing specification and send the instance on to the next site. The authors state that this type of evaluation is most suitable for the requirements of a distributed system.

The design presented herein is similar in approach to that presented by Mazer et. al.. In particular, the concept of a message as an independent "intelligent" entity guiding itself through the system without explicit user direction is the basis of the design presented in this paper. Like

Mazer et. al., the design developed here associates routing specifications with message types so that the system can determine the routing requirements of each individual instance of a message type. Likewise, the actual routings may depend upon current message and system state. The design developed in this paper incorporates type routing only. Instance routing and override routing are not included. The guidelines presented by Mazer et. al. for the distribution of message type definitions and also the evaluation of routings have been followed here. In particular, a copy of the message type definition will reside at each appropriate site in the system. Also, the routing evaluation will be distributed with each site performing an evaluation of the routing specification and forwarding the instance on to the next site.

2.5.4 Modeling Jobs in a Distributed System [McBr83].

In this paper, McBride and Unger identify five major components that are necessary to model the processing of a job in a distributed environment. These include:

- a structural model for each procedure in the system,
- a structural model of the control program that directs the processing of a job,

- knowledge of the current control and data states of a job,
- global information available to all jobs and procedures in the system,
- non-global data files.

The authors then proceed to describe a model which can be used to depict this control and information flow of a job in a distributed system. Their model utilizes individual Petri nets to describe each procedure comprising a job, along with a control Petri net which oversees the execution of the job in total. A Petri net is used to model the static (structural) properties of a job , whereas an entity called a "token" records dynamic information regarding the current state of the job. McBride and Unger allow this token to be split into sibling tokens so that concurrent activities of a job can be modeled as well.

The authors next discuss the general Petri net model noting, in particular, the extension by R. M. Keller [Kell76] to associate a predicate condition with each transition. A true predicate is a necessary condition for the transition to fire.

McBride and Unger then expand upon the concept of the token which represents a job in their model. The token contains

the following components:

- "local data" available only to the job itself,
- a control Petri net which describes the manner in which the local data is to be processed by procedures available in the system,
- the current position of the token in the control net (note that this token can also appear in the marking of a Petri net corresponding to a procedure which is being carried out on behalf of the token so that its position in this net must be recorded too),
- the token may carry a capabilities list describing the token's access and authorization rights to data files,
- the token may also carry within it a history of all transitions which have been executed on behalf of the token.

In the model presented, each procedure associated with a transition in a control net is identified by a procedure name that is global throughout the system. A global network directory maps each procedure name onto the Petri net representation for it.

McBride and Unger also note that the concept of a Petri net can be expanded so that data files can be used in a fashion similar to the way that places are used. The data file access requirements could be graphically represented with arcs between the data file representation and the transition utilizing the data file. In this way, the dependence of transitions on data files is brought into prominent view.

The authors note two variations that can be made to the control net. First, they state that it is possible to have the control net that is associated with a job type globally available rather than replicating it with each token instance of that job type. A second variation can be constructed in which the original control net is split into siblings, each of which receives only a portion of the original control net. Each of the siblings can then be processed independently of each other. Each of the completed siblings would finally be merged back into a completed copy of the original control net.

McBride and Unger also present a description of how a control net can cope with the problem of lost tokens. They describe the data fields and transitions necessary to allow the control net to recover from a lost token.

The design presented herein utilizes many of the concepts described by McBride and Unger. However, the design presented in this paper is limited to the processing of an office form rather than looking at an office system as a whole. Consequently, this design is concerned only with the routing and synchronization control net associated with an office form and not with all procedures available in the system. This design does use the concept of a control Petri net to oversee the execution of a job, where the idea of a job refers to the routing of an intelligent office form throughout the system. Likewise, the concept of a "token" being associated with the control Petri net to record current status information is utilized. The suggestion by McBride and Unger that the control net associated with a job type be globally available rather than being replicated for each instance of the job type is followed. The control net for each form type will be defined once and made globally available throughout the system (where needed). Thus, only the dynamic information associated with each individual instance of a form type will be forwarded through the system as a part of the "token".

2.5.5 A System for Managing Structured Messages [Tsic82].

In this paper, Tsichritzis et. al. present a prototype system which integrates the facilities of message systems and database management systems. This combination allows the system to manage structured messages according to their content. Traditionally, message systems have delivered the message but have not managed the message. The authors' intent in this paper is to enhance the functionality of message systems by adding to them the ability to interpret, at least partially, the messages which they handle. By so doing, users can query the message system to find messages, accumulate data contained in the messages, or specify automatic processing and routing procedures which make use of the contents of the messages. The authors' system superimposes a structure on the message types. This structure is known to the system and is used for the interpretation of the message.

The basic structure of the system presented by Tsichritzis et. al. is composed of a number of logical units called stations. Stations may be grouped together on physical units called nodes. A control node performs synchronization and control activities. The remaining nodes are known as satellite nodes. Each node supports a number of processes which are either associated with the node itself or a particular station on the node.

Tsichritzis et. al. assume that each user of the system operates a single station. A user interface process resides at each station. A user defines a message type, via the user interface process, by creating a display template. Once the message type has been defined, instances of the message type can be created by filling in the template. When a message instance is created, the system generates a globally unique identifier for the new message instance. This identifier, known as the message key, is permanently attached to the message instance. As messages are mailed from station to station, an entry is made to a log file. This file records the time of the operation as well as the source and destination stations. Thus, it is possible to locate a message by examining the message's most recent log entry. Similarly, it is possible to construct a trace of a message from the log file. The authors' note that the log file and counters from which message keys are dispensed are located at the control node.

Tsichritzis et. al. state that there may be some activites which require information from messages which are spread over more than one station. In such cases, queries for information present in messages are useful. The authors' have therefore provided for message queries in their

system. The user specifies both the content and the scope of the query.

Tsichritzis et. al. also provide for automatic procedures to operate on messages in the system. The specification of an automatic procedure indicates to the system that it should look for certain messages and act on them as specified.

The design presented herein, utilizes one very important concept presented by Tsichritzis et. al.. The idea of imposing a formally defined structure on a message type so that the system can manipulate the content of the message is basic to the design presented in this paper. Also, the design presented here greatly expands the idea of a message definition to include routing and synchronization information. Although Tsichritzis et. al. touched on the idea of message routing being somehow dependent on the message content, the idea of a message as an "intelligent" entity guiding itself through the system was not conveyed.

CHAPTER 3
DESIGN SPECIFICATION

3.1 Design Objectives

The purpose of this specification is to describe a system which can coordinate the routing and synchronization of an office form automatically as it flows through an office system. The design is generic in the sense that it addresses the question of routing and synchronization of forms, but it does not address the implementation details of any specific form. The intent is that this system has the ability to be used as a high level routing and synchronization control mechanism for automated forms in any office system implementation. The individual form definitions then become a substructure of this overall control system.

3.2 Structure of the Design Specification

The design specification which follows is organized into three parts. The first part describes the external Petri net representation of the routing and synchronization requirements of a form. This Petri net is generated by the form's designer. The second part presents the design of the

control net. The control net is an internal data structure which captures the semantics of the external Petri net specification. It is used to ensure that a form's automated processing meets the routing and synchronization requirements identified by the form's designer. The third part of the design discusses the data structures and processes required to support the run time execution of a form instance.

3.3 The Petri Net Specification for the Routing and Synchronization of an Office Form

The Petri net structure as described in this section is the tool used by the designer of a form to specify the routing and synchronization requirements of the form. This design imposes some requirements and restrictions on both the form definition and the Petri net structure. There are two general requirements of the form definition. They are:

1) each form instance will have an initial state which assigns initial values to all control variables and data variables;
2) concurrent processes in the Petri net structure are not allowed to operate (except where all are read only) on any common data elements.

Additionally, the Petri net structure has the following

requirements:

1) the Petri net must be safe;
2) the Petri net must be properly terminating;
3) a predicate condition may be associated with any transition in the net.

There are two cases of predicate usage [Kell76]. The first case involves using predicates as decision points in the net. A decision always involves two alternative transitions. One or the other may be enabled, but not both. This design requires both alternatives to be fully specified in the Petri net. Predicates can also be used as a synchronizing mechanism. In this situation, the predicate may prevent a transition from firing even though there is no alternative to choose from (i.e., waiting on the availability of a system resource). This design makes the assumption that transitions dependent on a synchronizing predicate will eventually either fire or be resolved in some other manner (e.g., through timeout processing).

The data variables involved in the evaluation of a predicate can be either local or global. Local variables are contained in the structured message associated with a form instance. Local variables can only be changed as a result of the firing of a transition in the control net of

the form instance. This design assumes that a local variable is potentially modifiable by every transition in the control net. Consequently, all predicates involving local variables must be re-evaluated after each transition fires.

Global variables can also potentially be changed by a transition firing. However, in addition, global variables may be changed by processes external to the control net. This could cause some inconsistencies in the execution of the control net.

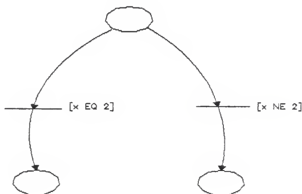
For example, first suppose that a predicate evaluation indicates that a certain global resource is available and this enables a particular transition to fire. Now suppose that between the time that the predicate was evaluated and the process associated with the transition executes, some other process external to the control net consumes the resource. The results are not reliable. For this reason, this design restricts the use of global variables which can be associated with a predicate to those whose truth is not dependent on processes active in the system. In particular, this implementation restricts predicates to the use of only one global data item - the system clock. Predicates involving global data must be re-evaluated periodically to

handle situations where a synchronizing predicate is dependent on a global variable (i.e., Is time > 6:00 P.M.?). They must also be re-evaluated after each transition fires.

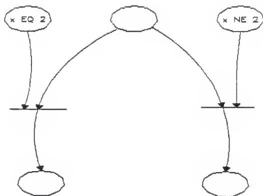
A full implementation of this system would include a user interface to allow the form designer to define the Petri net representation to the system via a specification language. This form definition would then be translated mechanically into the internal control structures. In the prototype implementation described here, the translation is a manual activity. Consequently, in the prototype system, it is the responsibility of the form designer to enforce the requirements mentioned above. In a full implementation, these requirements could be enforced by the system.

It should be noted that an implementation of this system would require an Office Forms Administrator to administer the system. There is one central administration site. The Office Forms Administrator will coordinate the creation, modification, deletion and distribution of form type definitions. Additionally, the Office Forms Administrator will tune the system, control security, and be the interface to end users of the system.

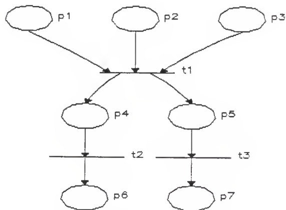
Predicate conditions are specified in the Petri net as conditional statements contained within brackets as shown in Figure 3-1. The actual content of the predicate statement (i.e., x EQ 2) is not of importance to the discussions in this paper. Therefore, for the sake of brevity and clarity, predicates hereafter will be represented only by their truth value (i.e., [T]). Recall that the predicate must be true in order for the associated transition to fire.



Specification Of Predicate Conditions
Figure 3-1



Logical View Of Net From Figure 3—1
Figure 3—2



Naming Convention For Nodes
Figure 3—3

Before proceeding, it seems appropriate to discuss the way in which predicates are treated internally in the system. A predicate appears logically to the system as an additional conditional input place to the transition associated with the predicate. This "conditional place" will have a token present whenever the predicate is true. The token will be removed when the predicate becomes false. As an example, the systems's logical view of the net from Figure 3-1 is depicted in Figure 3-2.

An additional note is made concerning the convention to be followed in determining the internal representation of the Petri net specification. All places (including conditional places) and all transitions in the net will be uniquely named as shown in Figure 3-3. The numbering convention for the nodes is top-to-bottom, left-to-right.

The final items to be presented relative to the Petri net specification are the constructs available to be used in the design of a net. Five constructs are allowed. They are depicted in Figures 3-4(a) through 3-4(e). As per Figure 3-4, these constructs are:

(a) sequential - t1 occurs sequentially, before t2;

(b) concurrent - t1 occurs sequentially, before t2 and t3; t2 and t3 can then occur

concurrently;

(c) decision - either t1 or t2 will fire, but not both;

(d) synchronizing predicate - t1 cannot fire until the predicate condition becomes true;

(e) halting transition - t1 has no output places; when t1 fires, a token is removed from each of its input places.

3.4 The Control Net

The control net is an internal data structure which records the static properties of the external Petri net specification. There is one control net corresponding to each form type definition. A copy of the control net will reside at each site capable of processing the form type. A control net's structure consists of three data components as described in the following sections.

3.4.1 Process Table

A process table, Figure 3-5, is a data structure which:

a) maps the transitions of the Petri net to executable processes which can carry out the action needed for that transition;

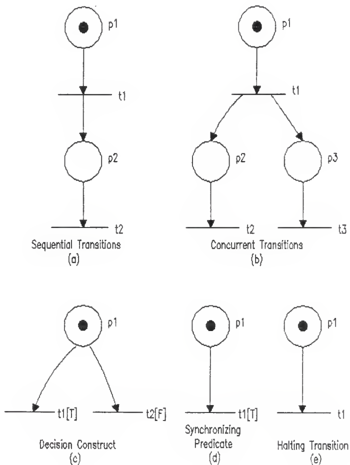


Figure 3-4

b) maps processes to their execution sites in the office system.

Note that the mapping of processes to their execution sites is by form type. Generally, in other works from the literature, this mapping has been global across all form types in the system. The mapping by form type, as presented here, has some distinct advantages. First, the Office Forms Administrator has an increased capability to tune the system. Each individual form type can be specifically routed not only to a process, but to a particular site as well. This allows the system load to be tuned in a manner which is based on the processing requirements and volume of each individual form type.

Another advantage exists from a security perspective. The designer of a form type can be restricted to some subset of the total processes and sites available in the system. In this way, the access rights of the form type to sensitive processes or sites can be easily controlled.

3.4.2 Transition Input Map

A transition input map is a data structure (bit map) which identifies the input places for each transition in the external Petri net specification.

	Process Name	Sites Where Process Resides				
t1	process 1	A	G	F		
t2	process 2	B				
.						
.						
.						
.						
.						
tn	process n	A	C			

Process Table
Figure 3—5

	p1	p2	p3	p4	p5	p6	p7
t1	1	1	1	0	0	0	0
t2	0	0	0	1	0	0	0
t3	0	0	0	0	1	0	0

Transition Input Map
Figure 3—6

Figure 3-6 represents the transition input map for the net depicted in Figure 3-3. The bit map includes one bit position for each place in the net. Each bit position, b(i), corresponds to a unique input place, p(i). If bit position b(i) is set in row j, then the corresponding place p(i) is an input place for transition j. This table identifies the set of input places for each transition in the net. A mapping also exists from the transition input map to the process table; the i-th entry of each structure corresponds to the same transition. The process table is accessed by the system to retrieve the process information associated with the transitions in the transition input map.

3.4.3 Transition Output Map

A transition output map is a data structure which identifies the output places for each transition in the external Petri net specification. It is structurally identical to the transition input map. The difference is interpretation. The marking of each row in this structure reflects the set of output places which are to receive tokens upon completion of the firing of the associated transition. The transition output map corresponds to the

process table in the same manner as the transition input map.

3.5 The Form Definition

The implementation of a specific office form would require a definition of the form's data component. This might include such things as field identifiers and attributes, editing instructions, initial values, access rights, etc.. The form definition is not addressed here since it lies outside the scope of this document.

3.6 The Run Time Design

In this design, a form is viewed as a job in which work progresses by having it flow between procedures that operate on its data. These procedures can operate sequentially or concurrently on a form, and the procedures can be distributed over several machines or local to one. The design is based on the concept of incorporating both the form content and the routing and synchronization control information into a structured message. The mail system then becomes the mechanism for transporting the form (contained in the structured message) through the office system. Defining the structure of a message type to the system allows a controller process at each site to

interpret the control and data information contained in the message. This enables the system to coordinate the routing and synchronization of each individual form instance of that type automatically without explicit user direction.

The run time environment is supported by four data structures and four processes. Briefly described, the four run time data structures are:

1) structured mail messages which contain both control information and the form's data;
2) an incoming mailbox used to receive mail messages;
3) a sequence counter which maintains a sequence number used to identify form instances;
4) a form instance control table which represents the control and data states of each form instance active at a site.

Likewise briefly described, the four run time processes are:

1) a form request process at each site which introduces new form instances to the system;
2) the UNIX™ mail process which serves as the transport mechanism for messages in the system;
3) multiple form service processes at each site, each of which provides some sequence of actions

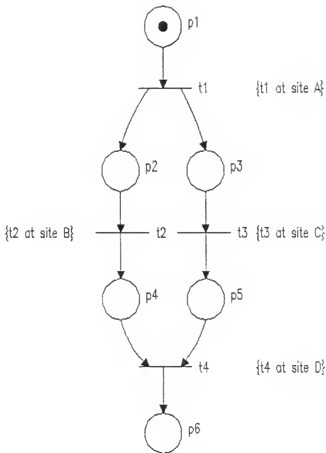
necessary to process the form;

4) a controller process at each site which coordinates the execution of each form instance at the site.

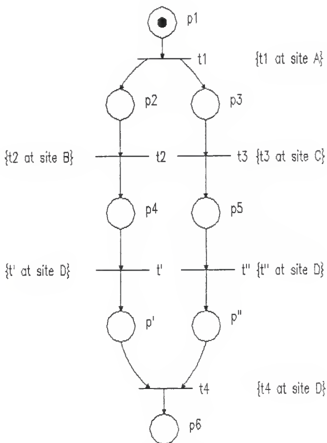
Before presenting the run time data structures and processes in detail, it is appropriate to review the run time design which these data structures and processes must support. The control net evaluation scheme will be described first.

Two basic schemes for control net evaluation are discussed in the literature, centralized and distributed. A central evaluation approach involves one controller process and a single copy of the control net at a designated control site. The central controller process evaluates the control net and sends the form instance to the appropriate execution sites to be processed. Once these processes have acted on the form instance, it is returned to the central site where the controller process once again evaluates the control net and continues the cycle. In this approach, all evaluation of the control net is done at one central site.

In a distributed evaluation scheme, each processing site in the office system has a copy of the controller process. A copy of the control net is included as part of the message



Distributed Evaluation Example
Figure 3–7



Distributed Evaluation Solution
Figure 3-8

which travels through the system. As each site completes its processing of a form instance, the controller process at that site evaluates the control net and forwards the form instance (and control net) to its next processing site. An advantage to this approach is a more efficient utilization of the network.

A problem with using distributed evaluation for concurrent processes is illustrated with Figure 3-7.

In Figure 3-7, after t1 fires at Site A, transitions t2 and t3 will both be enabled concurrently. In a distributed evaluation, a copy of the control net and form instance would be sent to both Sites B and C, where t2 and t3 would fire respectively. Once fired, the resultant control nets would be evaluated at each site. Neither evaluation would identify transition t4 as being enabled. In essence, deadlock occurs, with each site holding a resource (token) the other needs. Of course, this situation could be remedied by making some adjustments to the Petri net specification as shown in Figure 3-8. Places p4 and p5 could enable two intermediate transitions, t' and t'', at Site D, which in turn could enable t4.

The approach taken in this document is that an artificial extension of the Petri net specification, such as in

Figure 3-8, is unnatural and would be inconvenient for the user to apply in a form type definition.

Another solution to the problem of distributed evaluation of concurrent processes, which has been presented in the literature [McBr83], requires that a subspecification (i.e., some subnet of the original net) travel with a form instance sibling (copy) along each concurrent path. Each of the siblings would use its subnet to process independently of the others. At some point, the control and data information from each of the siblings would be merged. This approach, although possible, quickly leads to complexity in the application software. Therefore, a hybrid approach to control net evaluation is presented in this document which is a practical attempt to keep the Petri net specifications and the processing requirements straightforward.

The hybrid evaluation scheme presented here is neither centralized nor distributed - it is a combination of the two. In this approach, the evaluation of sequential processes is fully distributed. When handling concurrent processes (unless all of the processes reside at the same site), the evaluation resembles a central evaluation scheme.

The hybrid evaluation approach requires a form instance master (hereafter referred to as the master) to be in overall control of the execution of a form instance. The master may dispatch form instance siblings to facilitate concurrent activities, but these siblings must report back to the master upon completion of each process so that the current Petri net state can be reflected in the master at all times. An advantage to having a master always in control (as opposed to siblings processing independently), is that timeout processing capabilities are enhanced. The master has total knowledge of all processes being executed on behalf of the form instance. Consequently, timeout conditions can be easily identified and the associated responses can be coordinated by the master.

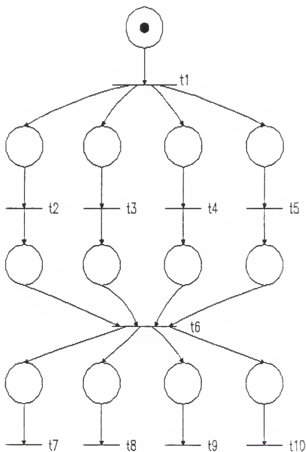
This hybrid scheme of control net evaluation would process the net in Figure 3-7 as follows:

1) t1 fires at Site A;
2) the controller process at Site A evaluates the resultant control net and discovers that t2 and t3 are now enabled;
3) the controller process at Site A (now acting as in a central evaluation scheme) forwards a copy of the form instance to both Sites B and C;
4) t2 and t3 fire at their respective sites;

5) the updated form instances from Sites B and C are returned to the controller process at Site A;
6) the controller process at Site A merges the form instance copies with the master, updating the control state appropriately;
7) the controller process at Site A now re-evaluates the control net and discovers that t4 is now enabled;
8) the controller process at Site A (now acting as in a distributed evaluation scheme) forwards the master copy of the form instance to Site D where t4 is fired.

It should be noted that form definitions in the system can be designed to exploit this hybrid evaluation approach. For example, groups of related processes could reside together at one site, with the routing and synchronization specification structured accordingly as in Figure 3-9.

In Figure 3-9, transitions t2 through t5 might all be purchasing processes which act on a requisition in the purchasing department at Site B. Transitions t6 through t10 might all be accounting processes which act on the requisition in the accounting department at Site C. The



A Net Structured To Benefit From
A Hybrid Evaluation Scheme
Figure 3-9

evaluation of the net in Figure 3-9 would be totally distributed.

Now that the hybrid control net evaluation scheme has been examined, the question of form instance copies can be addressed. Form instance copies will be created under two conditions.

1) The current site has completed its processing of a form instance and must now forward the form instance to the controller process at a new site where the next process(es) reside(s). In this case the current controller process relinquishes control of the form instance and the master is forwarded to the new controller process. In order to avoid the possible loss of a form instance, a copy of the master is retained at the sending site pending an acknowledgement from the receiving site. Only one controller process can be in control (i.e., possess the master) of a form instance at any given time. Note that for a controller process to relinquish control of a form instance, under the hybrid evaluation scheme, implies that all of the "next" (i.e., currently enabled) processes reside at the same (but new) site.
2) If all of the "next" processes do not reside at the same site, a form instance copy will be created for each

enabled transition, and forwarded either to a local form service process or the controller process supporting a remote transition. This case implies that concurrent transitions are enabled. In this situation, the current controller process retains control (i.e., keeps the master) of the form instance.

The purpose for making copies of a form instance is twofold. First, temporary copies of a form instance allow concurrent activities to take place. Secondly, copies reflect the state of the form instance at the point in time that a message was generated and can therefore be used to retransmit messages should timeouts occur. The master has potentially been updated by one or more processes which have completed in the interim, and thus the master cannot be used for retransmission. For the prototype implementation of this system, the network is assumed to be perfectly reliable and that all messages arrive without error and in the order that they are sent.

Form instance copies are retained until they are acknowledged (with an ACK). ACK's are generated under two conditions.

1) When a master is being forwarded to a new controller process, the new controller process responds with an

ACK. When the ACK is received, the former master (which is now a copy) is deleted.

2) When a form service process completes, it returns an ACK to its site's controller process in the shape of a copy of the form instance for the given process. The form instance copy at this executing site can fall into one of two cases:

   (a) the copy was created at this executing site (i.e., the master resides at this site);

   (b) the copy was created at some remote site and forwarded to this site for execution (i.e., the master resides at some remote site).

   In case (a), the (potentially) updated form instance copy is merged with the master and the copy is deleted. In case (b), the (potentially) updated form instance copy is returned (as an ACK) to the remote controller process which initiated the copy. When the ACK is received by the remote controller process, the form instance copy is merged with the master and the copy is deleted.

3.6.1 Run time Data Structures

The run time data structures will now be presented in detail.

3.6.1.1 Structured Mail Messages

A structured mail message is a data structure which contains the routing and synchronization control information for a form instance along with the form's data. The structure of the message varies by form type. The structure is specified in the form type definition which is generated by the form's designer.

3.6.1.2 Incoming Mailbox

An incoming mailbox is used to receive the structured mail messages generated by controller processes and form service processes. This is the same mailbox as implemented in the UNIX™ mail system.

3.6.1.3 Sequence Counter

A sequence counter is maintained at each site to store the last sequential number which has been assigned to a form instance. This sequential number is a sub-part of a unique identifier which is assigned to every form instance in the system. The identifier has the following format:

site::form type::sequence number.

3.6.1.4 Form Instance Control Table

A form instance control table, Figure 3-10, is a data structure which represents the current control and data states of each form instance active at a site. There is one form instance control table per site.

form instance 1	form instance identifier (a)	source site (b)	dest site (c)	time stamp (d)	process name (e)	global flag (f)	Petri net marking (g)	form's data (h)
form instance 2								
.								
.								
.								
.								
.								
form instance k								

Form Instance Control Table

Figure 3-10

The structure can be thought of as a table of individual form instance states. The information fields contained in the form instance control table are defined as follows.

(a) The form instance identifier uniquely identifies the particular form instance. This identifier is assigned when the form instance is initialized.

(b) The source site identifies the site sending a message. This field is populated whenever a message is forwarded to a local form service process or another controller process.

(c) The destination site identifies the site receiving a message. This field is populated under the same conditions as (b).

(d) The time stamp records the time at which a message is initiated. The time stamp is used to identify time out situations. This field is populated under the same conditions as (b).

(e) The process name field identifies the process for which a message is destined. This field is populated under the same conditions as (b).

(f) The global data flag is set when a form instance is initialized. This flag indicates whether any of the predicates associated with the form type are dependent

on global variables (i.e., the system clock). This flag facilitates a periodic re-evaluation of predicates involving global variables.

(g) The Petri net marking is a bit map (one bit per place) which reflects the current state of the Petri net execution (i.e., identifies all places which contain a token).

(h) The form's data fields reflect the current data state of the form instance.

3.6.2 Run Time Processes

The run time processes are presented in the following sections.

3.6.2.1 Form Request Process

The form request process provides a user interface which allows new form instances to be introduced to the system. The process basically allows the user to send a form request message to the controller process at a site. The request message contains the message type (form request) and the form type. Upon receipt of this message, the controller process will:

- create a master entry in the form instance control table;

- assign a unique identifier to the master;
- set the global data flag in the master;
- set initial control and data values in the master;
- evaluate the control net based on the initial values.

A copy of this process will reside at each site.

3.6.2.2 UNIX™ Mail Process

The UNIX™ mail process will serve as the transport mechanism for the structured mail messages communicated between:

1) form service processes and controller processes residing at the same site;
2) controller processes at different sites.

3.6.2.3 Form Service Processes

A form service process will be invoked when required as a background job from a controller process. A structured message which represents a form instance will be passed as an input parameter. Each form service process will provide some sequence of operations necessary to process the form instance. These processes will be capable of operating on both local (contained within the message) and / or global data elements. Once the form service process has completed

its work on the form instance, it formats an ACK and mails the updated message back to the controller process which invoked it. There can be multiple form service processes at each site.

3.6.2.4 The Controller Process

The controller process is the hub of activity at a site. Each controller process has two primary tasks:

- as a traffic controller, determining which traffic can flow (both when and where to) and which traffic must wait for conditions to change;
- as a data handler with the function of merging both the control and data portions of form instance copies with the master.

The development of the first task, the traffic controller, is the primary focus of this paper. In order to accomplish this task, the controller process must perform the activities as specified in the following pseudocode.

```
cycle
if there is mail then
   process the next mail message
endif
if it is time to re-evaluate global variables and
        timeout conditions then  (done periodically)
   begin
   evaluate all predicates associated with global
        variables and mark the associated
        conditional places;
   evaluate and process all control nets whose
        markings change as a result of this
```

```
            activity;
      check all form instances for timeout conditions
            and take appropriate actions
      end
   endif
   if there is no mail then
      sleep 60    (60 is arbitrary)
   endif
   endcycle
```

Note that the above pseudocode really represents a daemon process which is triggered periodically by a timing event or by the arrival of mail.

It seems appropriate at this point to elaborate on the handling of messages received by the controller process. There are four types of messages which must be handled. They are:

1) a form request;
2) receipt of a master forwarded from some remote controller process;
3) receipt of a form instance copy forwarded for processing from some remote controller process;
4) ACK messages from either a local form service process or a remote controller process.

When a form request is received, the controller process will create a master entry in the form instance control table and assign a unique identifier to the master. A subroutine unique to the form type will be invoked to set

the initial control and data values. The controller process will then evaluate the control net based on the initial values.

Receipt of a master is straightforward. A master entry is created in the form instance control table and the corresponding control net is evaluated. An ACK is also generated and returned to the sending controller process.

Receipt of a form instance copy is likewise straightforward. An entry for the copy is created in the form instance control table and the process identified in the "process name" field will be invoked as a background job. Notice that no control net evaluation is required here.

Three cases exist for handling ACK's. The first case is when the ACK is confirming recipt of a master which was forwarded to another controller process. Here, the controller process simply deletes its copy of the master from the form instance control table. The second case occurs when the form instance copy associated with the ACK was created at this site. In this situation, the controller process will:

a) delete the copy's entry in the form instance control table;

b) merge the form instance copy with the master;

c) mark the output places associated with the process which originated the ACK as now containing tokens;

d) evaluate the control net and fire any transitions which are now enabled.

The third case occurs when the form instance copy associated with the ACK was not created at this site. In this situation, the controller process will delete the copy's entry in the form instance control table and forward the ACK to the originating controller process. Notice that when the originating controller process receives the ACK, it will be treated as in the second case above.

The control net execution will now be described more fully. The first step in executing the control net is to re-evaluate all predicate conditions in the net. This is necessary in order to mark the conditional places in the master to reflect the current environment. A subroutine unique to each form type will be called by the controller process to perform this activity.

The second step of the control net execution involves a comparison of the Petri net marking recorded in the master with the transition input map. The master is compared sequentially with each entry of the transition input map.

When an entry is discovered in which all of the input conditions are satisfied, the process name and destination site are retrieved from the process table and stored by the controller process in an internal firing table as depicted in Figure 3-11. A token is removed from each input place "used up" as a result of firing the transition. The procedure then continues in the same fashion through the rest of the transition input map.

Once the processing of the transition input map is complete, the third step of the control net execution is to evaluate the internal firing table. If all of the enabled

Enabled Processes	Destination Site
process x	A
process y	A
process z	C

Firing Table
Figure 3-11

processes reside at the same (but new) site, the controller process will relinquish control of the form instance and forward the master to the controller process at the destination site. In this case, the original marking of the master is first restored. If multiple sites are involved, the controller process will invoke local form service processes (as background jobs) for those processes residing at the local site, and forward form instance copies to the controller process at each remote site in order to initiate remote processes.

3.7 Summary

The design presented in this chapter utilizes various concepts discussed in the literature. However, the idea of a hybrid control net evaluation scheme is unique to this paper. The hybrid approach combines the best features of centralized and distributed evaluation schemes to provide an effective and efficient practical solution to control net execution.

This design supports a functional prototype implementation. The intent is that the prototype can then be used as a basis for an expanded implementation.

CHAPTER 4

CONCLUSION

4.1 Extensions of This Work

The design presented in this document is a prototype. A full implementation could potentially support some additional enhancements. For instance, a user interface to allow the form designer to translate the external Petri net specification into an intermediate specification language would be helpful. This intermediate specification could then be mechanically converted into the internal data structures.

Another possible enhancement is to expand the information contained in the process table to include such things as timeout limits for processes and standard default actions to take if timeouts or run time errors are encountered (e.g., notify the Office Forms Administrator).

Timeout considerations would have to be addressed in greater detail in a full implementation. The prototype design assumes that the network and form service processes are perfectly reliable.

Network management could also be enhanced in the areas of revision control and status reporting. Procedures are needed to allow the introduction of revised form definitions to the system while permitting old versions of form instances to finish execution. Status reporting could be provided by maintaining a central log of all activity in the system. Potentially, the Petri net for a form instance, with its current marking, could be graphically displayed on the screen.

4.2 Concluding Remarks

A system for specifying and controlling the routing and synchronization of office forms has been proposed. The design is generic in that it can be applied to any office system implementation. Thus, the implementors of an office system are able to concentrate their efforts on the actual form definitions at hand.

A unique feature of this design is the use of a hybrid control net evaluation scheme. This hybrid approach offers an effective and efficient method of implementing an office system.

REFERENCES

[Andl79] Andler, S. "Predicate Path Expressions", "Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages", pages 226-236. ACM, San Antonio, Texas, January, 1979.

[Deit83] Deitel, H. M. "An Introduction to Operating Systems", Addison-Wesley, Inc., 1983.

[Fort85] Fortier, P. J. "Design and Analysis of Distributed Real-Time Systems", McGraw-Hill, Inc., 1985.

[Geha82] Gehani, N. H. "The Potential of Forms in Office Automation", "IEEE Transactions on Communications", Vol. COM-30, No. 1, pages 120-125, January, 1982.

[Kell76] Keller, R. M. "Formal Verification of Parallel Programs", "Communications of the ACM", Vol. 19, No. 7, pages 371-384, July, 1976.

[Maze84] Mazer, M. S. and Lochovsky, F. H. "Logical Routing Specification in Office Information Systems", "ACM Transactions on Office Information Systems", Vol. 2, No. 4, pages 303-330, October, 1984.

[McBr83] McBride, R. A. and Unger, E. A. "Modeling Jobs in a Distributed System", ACM 0-89791-123-7/83/012/0032, 1983.

[McBr85] McBride, R. A. "Execution Mechanisms for Jobs in a Distributed System", Draft Working Paper, Department of Computer Science, Kansas State University, March, 1985.

[Pete77] Peterson, J. L. "Petri Nets", "ACM Computing Surveys", Vol. 9, No. 3, pages 223-252, September, 1977.

[Pete81] Peterson, J. L. "Petri Net Theory and the Modeling of Systems", Prentice-Hall, Inc., 1981.

[Rama80] Ramamoorthy, C. V., Ho, G. S. "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets", "IEEE Transactions on Software Engineering", Vol. SE-6, No. 5, pages 440-449, September, 1980.

[Reis85] Reisig, W. "Petri Nets, An Introduction", Springer-Verlag, 1985.

[Tsic82] Tsichritzis, D., Rabitti, F. A., Gibbs, S., Nierstrasz, O. and Hogg, J. "A System for Managing Structured Messages", "IEEE Transactions on Communications", Vol. COM-30, No. 1, pages 66-73, January, 1982.

[VanD86] VanDusen, D. "Specifying and Controlling Routing and Sychronization of Office Forms With Predicate Path Expressions", Master's Report, Department of Computer Science, Kansas State University, 1986.

SPECIFICATION AND CONTROL
OF ROUTING AND SYNCHRONIZATION REQUIREMENTS
OF OFFICE FORMS USING PETRI NETS

by

Charlea W. Miller

B.S., University of Texaa at Auatin, 1979

An Abstract Of A Master's Report

aubmitted in partial fulfillment of the

requirements for the degree

Maater Of Science

Department of Computer Science

Kansaa State University
Manhattan, Kanaas

1987

This paper describes a system for specifying and controlling the routing and synchronization requirements of an office form. An office is viewed as an instance of a distributed system. An office form is viewed as a job in which work progresses by having it flow between procedures that operate on its data. These procedures can operate sequentially or concurrently; they can be distributed or local.

The design of the system is premised on the idea that the external representation of the routing and synchronization requirements are specified via a Petri net. Predicate conditions are allowed. The external Petri net representation is translated into internal control structures which support run time activities.

The implementation of the system is based on the concept of incorporating both the form content and the routing and synchronization control information into a structured message. The mail system then becomes the primitive communication mechanism. By allowing a controller process at each site to interpret the message, the message becomes an "intelligent" entity guiding itself through the system without explicit user direction.