

Student Work

12-1992

An Object Based Approach Towards the Automation of Office Procedures Using Intelligent Messages

Robert L. Palumbo

Follow this and additional works at: <https://digitalcommons.unomaha.edu/studentwork>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Palumbo, Robert L., "An Object Based Approach Towards the Automation of Office Procedures Using Intelligent Messages" (1992). *Student Work*. 3074.

<https://digitalcommons.unomaha.edu/studentwork/3074>

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.



**An Object Based Approach Towards the
Automation of Office Procedures
Using Intelligent Messages**

A THESIS

**Presented to the
Department of Computer Science
and the
Faculty of the Graduate College
University of Nebraska**

**In Partial Fulfillment
of the Requirements for the Degree**

MASTER OF ARTS

University of Nebraska
Omaha, Nebraska

by

Robert L. Palumbo

December 1992

UMI Number: EP73447

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP73447

Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code

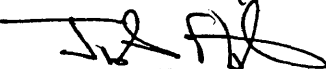
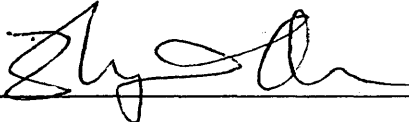


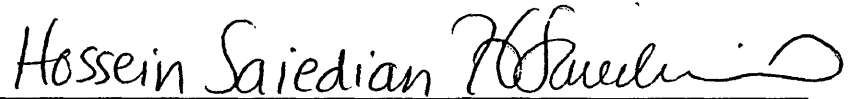
ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

THESIS
ACCEPTANCE

Acceptance for the Graduate College, University of Nebraska, in partial fulfillment of the requirements for the degree, **MASTER OF ARTS**, University of Nebraska at Omaha.

Committee

Name	Department
Stanley A. Wilman Jr.	MATH/CS
	ISQA
	Math/CS



Chairman

11-30-92

Date

Acknowledgements

The author would like to acknowledge the assistance of those individuals whose support and contributions made this thesis possible. A very special thank you is extended to Dr. Hossein Saiedian for his guidance, support, and generosity during this study. A sincere thanks is also extended to committee members: Mr. Stanley Wileman, Dr. Zhenqin Chen, and Dr. Justin Stolen.

The author would also like to acknowledge the faculty and staff of the Computer Science Department and the Graduate Studies Department for their valued assistance and support.

Finally, the author would like to thank his daughter Sarah who may have been too young to understand the meaning of this undertaking but nevertheless, provided the motivation and incentive to complete the work. My deepest and most loving thanks are extended to my wife Ann whose undenying love, understanding, and support have contributed in ways that words cannot express. The long nights have finally paid off.

Abstract

Office support systems are used to automate routine office tasks. Since office tasks often require the cooperation of several office workers who may be physically dispersed, it is important to develop advanced communication systems that better facilitate collaborative and cooperative office work among office workers.

In this thesis, we propose an approach to the construction of advanced communication systems in which messages are represented as *objects* that are “intelligent” and “active” and can therefore perform certain activities (such as interacting with various entities to collect data) and decisions (such as dynamically deciding which user to go to next) on their own. The system is called *Intelligent Message System* IMS. The major components of IMS are as follows:

1. *Intelligent Message Objects* (IMO). These objects represent the actual intelligent objects of the system.
2. *System Mail Manager* (SMM) which provides the users an interface to create, send, receive, and maintain IMOs.
3. *Intelligent Message Script Language* (IMSL) is a language used to *program* IMOs.

Office workers can delegate the responsibility for certain routine office tasks to an IMO and can therefore spend their time on more important activities. Thus, in addition to its theoretical contributions, this thesis provides a framework for building an advanced computer-based message system that increases productivity in an office environment.

Each component of the IMS is explained in detail in the thesis. A variation of BNF formalism is used to define the syntax of IMSL while VDM is used to formally define the semantics of major functions of intelligent message objects. A number of examples are provided to illustrate the effectiveness of the IMS in automation of certain office tasks.

Contents

Table of Contents	ii
List of Figures	v
List of Tables	vi
1 Introduction	1
1.1 Background	1
1.2 Office Procedures	2
1.3 Office Automation and Communication	3
1.4 Computer-Based Message Systems (CBMS)	4
1.4.1 Information Management	5
1.4.2 Office Task Management	5
1.4.3 Time Management	6
1.4.4 Discussion	6
1.5 Limitations of CBMS	6
1.6 Problem Definition	7
1.7 Organization of Thesis	9
2 Literature Survey	10
2.1 Foundation	10
2.1.1 Object-Orientation	10
2.1.2 Active Objects	11
2.2 R2D2 - Research-To-Development-Tool	12
2.3 Imail	14
2.3.1 Imail Prototype System	15
2.3.2 Imessage Routing	17
2.4 MMS	19
2.4.1 Routing Model	20
2.4.2 Routing Specification Language	20
2.5 KNOS	23
2.5.1 KNO Environment	24
2.5.2 KNO Structure	24
2.6 Conclusions	26

3	IMS - Intelligent Message System & Operational Semantics	28
3.1	IMS	28
3.2	IMO - Intelligent Message Object	29
3.2.1	Global Data Object	31
3.3	Route Control Object (RCO)	32
3.3.1	RCO Data Space	33
3.3.2	RCO Route Event Table	35
3.3.3	RCO Manager	38
3.3.4	Summary	38
3.4	Message Control Object (MCO)	38
3.4.1	MCO Data Space	39
3.4.2	MCO Methods Table	41
3.4.3	MCO Message Script	41
3.4.4	MCO Manager	42
3.4.5	Summary	42
3.5	IMSL	43
3.5.1	IMSL Variables	43
3.5.2	IMSL Command Set	43
3.6	System Mail Manager	46
3.6.1	Interprocess Communication	47
3.7	IMO Administration Environment	47
3.8	IMO Creation Environment	49
3.9	IMO Transfer Environment	50
3.10	Operational Semantics	51
3.11	IMO Creation	51
3.12	IMO Routing	53
3.13	IMO Receipt	54
3.14	System Configuration File	54
3.15	Summary	55
4	Formal Definition	57
4.1	VDM	57
4.1.1	Data Types	58
4.1.2	Set Types	58
4.1.3	List Types	59
4.1.4	Record Types	60
4.1.5	Mappings	61
4.1.6	Operations and Functions	61
4.2	IMS Specification	63
4.2.1	INIT-IMS	65
4.2.2	CREAT-MCO	66
4.2.3	CREAT-RCO	66
4.2.4	CREAT-IMO	67
4.2.5	ARCHIVE-IMO	68
4.2.6	PURGE-IMO	70
4.2.7	SEND-IMO	70

4.2.8	RECV-IMO	71
4.2.9	ADD-RDEST	74
4.2.10	DEL-RDEST	74
4.2.11	PROCESS-SCRIPT	76
4.2.12	EVT-PROCESS	76
4.2.13	IMO-INITIATE	80
4.3	Formal Definition of IMSL	81
4.3.1	BNF Overview	81
4.3.2	BNF Specification of IMSL	81
5	IMS Applications	86
5.1	A Weekly Status Report Example	86
5.1.1	Task Description	86
5.1.2	Automating Status Report Processing	88
5.1.3	Event Table Specification	90
5.1.4	Message Script Specification	91
5.1.5	IMO Execution	91
5.1.6	IMO Message Script	93
5.2	Scheduling a Meeting	94
5.2.1	Task Description	95
5.2.2	Automating the Scheduling Process	95
5.2.3	IMO Message Script Definition	96
5.2.4	IMO Message Script Description	100
5.3	Summary	101
6	Conclusions and Further Research	102
6.1	Contributions	102
6.2	IMS Overview	103
6.3	Relation to Other Models	104
6.4	Future Research	104

List of Figures

2-1	Example imessage script.	18
2-2	Route Specification Language Syntax	21
2-3	General Kno class form.	25
2-4	KNO Production Rule.	25
3-1	Intelligent Message Object Internal Structure.	31
3-2	Route Control Object Internal Structure.	33
3-3	Message Control Object Internal Structure.	39
3-4	System Mail Manager Functional Representation.	48
3-5	IMS System Configuration File.	56
4-1	IMO Record Type Specification	64
4-2	VDM Schema for IMS-INIT().	66
4-3	VDM Schema for CREAT-MCO().	67
4-4	VDM Schema for CREAT-RCO().	68
4-5	VDM Schema for CREAT-IMO().	69
4-6	VDM Schema for ARCHIVE-RCO().	69
4-7	VDM Schema for PURGE-IMO().	70
4-8	VDM Schema for SEND-IMO().	72
4-9	VDM Schema for RECV-IMO().	73
4-10	VDM Schema for ADD-RDEST().	75
4-11	VDM Schema for DEL-RDEST().	77
4-12	VDM Schema for PROCESS-SCRIPT.	78
4-13	VDM Schema for SEND-IMO().	79
4-14	VDM Schema for IMO-INITATE().	80

List of Tables

2.1	KNO Actions by Type.	26
3.1	Global Data Object Internal Structure.	32
3.2	RCO Data Space Internal Structure.	34
3.3	RCO Route Event Table.	36
3.4	RCO Event Specifications.	37
3.5	RCO Route Event Table Actions.	37
3.6	MCO Data Space Internal Structure.	40
3.7	IMSL Predefined Variables.	44
3.8	IMSL Command Set.	44
3.9	IMSL Relational, Arithmetic, and Logical Operators.	46
4.1	VDM Set Operations.	59
4.2	VDM List Operations.	60
4.3	VDM Mapping Operations.	61
4.4	VDM Predicate Operations.	62
5.1	Telecom Software Development Team	88
5.2	Thesis Committee Members	96

Chapter 1

Introduction

Office support systems are developed to automate office tasks. Office tasks are generally communication intensive and involve a large volume of document exchange between office entities. Thus, office communication tools, and in particular, *computer-based message systems* have a major impact on office automation. In this chapter, we describe the role and limitations of computer-based message systems in the automation of office tasks and propose an *intelligent* message system that allows office task automation to become more simplified and natural.

1.1 Background

An office can be described as the central knowledge and information base within an organization. Information is received, stored, processed, and generated from within the office. The resulting knowledge that has been accumulated from this information is disseminated among the entities that interact with the office. The functional organization within an office is directed by a complex set of office procedures. These procedures define a relationship between how information is represented and ultimately processed. Office procedures can be grouped together to form an office task.¹ Collectively, these procedures and tasks encapsulate the fundamental office function of information processing. In any environment that supports information processing as a primary function, a mechanism must exist that allows information to be reliably and effectively communicated throughout the environment.

¹Procedures and tasks will be used interchangeably in this discussion.

This is particularly important in an office environment where task execution can require dissimilar input from multiple entities.

Most organizations function within a distributed processing environment. In such an environment, the ability to execute an office task can quickly evolve into a time-consuming and inefficient effort. Such a situation can arise when an office task requires input from several office workers. Each worker may be responsible for a specific procedure within the task. The organizational structure of the task may require that some procedures be executed sequentially while others can be executed in parallel. This necessitates a cooperative group effort among the office workers to ensure that information is exchanged in an efficient and cost effective manner with respect to the completion of the task.

The ability to provide support for cooperative office work environments has come to the forefront of research in office automation. Technological advancements have given rise to more sophisticated and powerful computer systems. As a result, applications that run on these systems are expected to provide a more open system architecture with increased functionality and flexibility. This trend has created a demand for intelligent office tools that can provide support for distributive communication and procedural processing. Tools such as traditional computer-based message systems, which function as a primary source for office communication, only provide for simple message transfers. We believe that these systems can be expanded to provide advanced capabilities which would serve to promote a more cooperative work environment.

1.2 Office Procedures

Woo (Woo & Lochovsky 1986) defined office tasks in terms of *structured* and *unstructured* tasks. Structured tasks are described as routine tasks that can be solved using a predefined step-by-step procedure. In other words, the task can be defined as a sequence of actions that can be expected to remain constant over time. Typically these are viewed as routine tasks and can usually be replaced using an automated tool.

Unstructured tasks are those tasks for which no step-by-step solution exists and as such, cannot be easily automated. With these tasks, there is a *logical* sequence of actions that do not remain constant over time. These tasks, by definition, are harder to solve than

structured tasks (Woo & Lochovsky 1986, Woo & Lochovsky 1987). This can be illustrated by the fact that since a predefined solution to a task does not exist, an individual may not have all the information required nor be aware of the necessary information to perform the task. Furthermore, information tends to be dynamic in nature and thus the content, as well as, the location of the information can change from one moment to the next. Unstructured tasks typically require a cooperative group effort on the part of office workers to complete the task. Therefore, communication plays an important role in the exchange of knowledge and information within the office environment.

As an example of structured and unstructured tasks, consider the process of applying for a bank loan. The tasks of performing credit checks and calculation of payments and interest can be automated, and as such, are considered structured tasks. On the other hand, determining whether an applicant is granted the loan is considered an unstructured task because an applicant may not satisfy all requirements pertaining to the loan. Thus, loan officer will have to decide whether the loan should be awarded. Clearly, this type of decision is based on other intuitive factors that introduce a level of complexity that can not be easily automated.

1.3 Office Automation and Communication

Guiliano (Giuliano 1982) defined *office mechanization* as the process of replacing structured office tasks by automated tools. In the most strict sense, this phrase defines the concept of office automation. In the past, the focal point of this automation has been the automation of routine business applications exemplified by accounting, inventory, word processing. However, office automation has encompassed a much broader spectrum than the automation of routine applications. With the advances in telecommunications, office workers can communicate world-wide using a wide range of communication mediums such as video-conferencing, cellular telephones, and voice and fax systems.

The ability to easily communicate office knowledge and information has led to an increase in a more distributive and cooperative work environment. In the context of this type of environment, the office infrastructure can be described as a *message* driven environment. Each task that takes place within the office can be defined as a series of messages which

are exchanged between the entities executing the task. Given the fact that an office is a communication intensive environment, a primary attribute for any automated office tool should be the ability to provide support for communication between office entities.

Traditionally, office workers deliver messages through the use of communication systems known as *computer-based message systems* (CBMS). The importance of these systems and their role in managing office tasks cannot be overstated. However, in many ways these systems are limited in their capabilities and do not offer the kind of flexibility that is required in a dynamic office environment. We elaborate on the role of the CBMS and its limitations and provide a perspective of the goal of this thesis in the next sections.

1.4 Computer-Based Message Systems (CBMS)

Office workers carry out local and distributive communication primarily through the use of computer-based message systems or electronic mail systems. These systems provide the conduit that allows two entities to communicate electronically without regard to geographic locality. With such systems, users can deliver and collect information with minimal effort. They can create, send, read, route, and delete messages with the actual transmission of the message handled explicitly by the system and the underlying communication subsystem.

The Advanced Research Project Agency (Robert 1970) is credited with sponsoring the research effort that led to the development of this type of message system. *MSG* (Vittal 1976) is noted as being one of the first successful CBMS that resulted from this research effort. The advantages of CBMS are summarized as follows (Bruder et al. 1981):

- Fast, reliable transportation of messages.
- Asynchronous communication between parties.
- Removal of geographic restrictions.
- Optimal use of sender's and receiver's time.

A CBMS can be viewed as a third party which acts as an intermediary between the originator and recipient of a message. The originator creates a message for delivery and passes it to the CBMS. Once the control of the message is transferred to the CBMS, the CBMS becomes solely responsible for maintaining the integrity and delivery of the message

to the recipient. Each user of the system maintains a logical mailbox where messages can be placed upon delivery. As messages are placed in the mailbox, the recipient must periodically check for any newly arrived messages². When a message arrives in a mailbox, the recipient is then allowed to take possession of the message. It is important to note that actual ownership of the message shifts between the entities involved as the message is transferred through the system.

Within an office, a CBMS provides several key functions which are enumerated as follows (Mackay 1988):

- Information Management.
- Office Task Management.
- Time Management.

An overview of each of these functions is provided below.

1.4.1 Information Management

One of the keys to success within an organization is the ability to effectively manage large amounts of information. As the information flows through the organization it must be processed in an effective and reliable way. A CBMS can provide the facilities to allow an office worker to perform administrative functions such as information analysis, data collection, filing, and retrieval in a very efficient and productive manner. Since the information is transferred in the form of messages, a process of information categorization can be performed by the receiving node. This process allows the information to be automatically organized based on factors such as the source and priority of the information.

1.4.2 Office Task Management

Since many office procedures require the cooperative effort among a group of office workers within a distributed environment, a CBMS is a logical extension for the management of the task. This type of effort requires effective communication between those involved in

²More recent CBMSs can indicate arrival of new mail by displaying an icon on the display terminal when a new message arrives.

the task. Each member of the group performs certain tasks based on the requests of other members in the group. It is much more convenient and efficient to utilize the resources of a CBMS to transfer the request to the recipient office worker. In this manner, each member can continue execution of other subtasks while the request is being carried out.

1.4.3 Time Management

Office workers are required to perform a considerable number of tasks each day. Since many of these tasks arrive in the form of messages, the office worker must be able to make some assemblance as to which tasks are more important. This is necessary to minimize the amount of time spent performing a single task and to ensure that a critical task is processed before a non-critical task. Thus, a CBMS can be used in the form of a time management tool by allowing this prioritization of tasks. Also, from a strictly physical standpoint, it is by far more time and cost effective for an office worker to send or reply to a message by using an electronic message as opposed to other methods such as the telephone, regular mail, or to just physically carry out the request or task manually.

1.4.4 Discussion

Based on the office functions enumerated above, it is clear that CBMSs have become an integral component in the office environment. The functions provided by the systems serve to increase and enhance the efficiency and management with which daily office activities are carried out. Therefore, development of new forms of CBMSs must preserve this level of functionality, while at the same time providing additional capabilities that will further extend the system's usefulness. Our model extends the functionality of the CBMS by allowing messages to have a more active role in the execution of certain office activities.

1.5 Limitations of CBMS

Although technological innovations have resulted in the development of computer-based message systems that offer greater flexibility and functionality than those of the past, there are several key limitations to these systems within the office environment (Tsiehrizis 1985).

The fundamental limitation of these systems is their passive nature whereby users are

required to initiate all the actions. The system defines a static relationship between the users. That is, the scope of user communications is very much limited by the underlying functionality inherent within the system itself.

Furthermore, messages in these systems are considered passive entities. That is, messages are composed strictly of data and possess no processing capabilities. Users are required to know all of the intended recipients and must specify the full routing path of the message. Once a message is sent to a destination, the originator of the message loses jurisdiction over the message. When the message arrives at the destination, the recipient is free to act on the message. The recipient can modify the message and forward it to another destination without approval or knowledge of the originator. Users are also required to perform all message management functions as necessary.

In general, these systems only provide a framework for simple point-to-point communication between entities. They do not provide the enhanced functionality required to support a cooperative and distributive work environment. For example, since a message is a passive entity it cannot locate a piece of information and then return back to the originator with the results of the search. Nor can a message be programmed to perform a manual office procedure, thus allowing an office worker to concentrate on more important tasks. Ideally, a CBMS should support a more dynamic environment which would allow the creation of intelligent messages that could be delegated the authority to perform a task or set of tasks on behalf of the office worker. In the next section we present the problem definition and introduce the requirements for an intelligent message system.

1.6 Problem Definition

The office environment has been characterized as a knowledge and information based center of activity. Given the fact that these entities have become communication intensive environments, the role of the computer-based message system has increased to the point where these system have become an integral part of the organization. However, we argue that these systems do not fully serve the dynamic requirements that exist within an office.

As an example, consider the problem of locating a piece of information within a large organization (Woo & Lochovsky 1986). As information continually flows from one location

to the next, it is not possible to guarantee that the information in question can be found even after visiting each location within the organization. Also, as the organization grows, the complexity of locating the same information increases to the point where productivity can diminish past a reasonable level of tolerance. The success of the organization is very much dependent on the organization's ability to manage the flow of information in an efficient and effective manner.

We believe that the evolution of CBMSs should encompass a role that parallels the dynamic nature of the office. By this we mean that a CBMS should serve a far more important role in the office than just a system for editing and transporting of messages. It should comprise a system that supports a more dynamic entity relationship more closely related to a real world environment.

What we envision is an advanced message facility in which messages are viewed as *active* and *intelligent* objects³. By active we mean that a message exists as a functioning entity capable of performing its own tasks. The intelligence within a message reflects the capabilities that a message can interact with a recipient and make dynamic routing decisions to facilitate the execution of the task assigned to the message. This is in direct contrast to messages in traditional CBMSs. Thus, we are proposing the conceptual framework for a message system in which messages encapsulate the following properties:

- Can perform a sequence of actions at a recipient location based on a message script.
- Can collect responses from their recipients.
- Can make dynamic routing decisions based on external events.
- Can return to the originator if required.
- Allows the originator to maintain jurisdiction over the message.

A system such as this would allow messages to be programmed to carry out a task that would normally require manual intervention on the part of the both the originator and recipient. Although our model is predicated towards automation of routine office procedures, in general, this system provides the framework for modeling any routine task in which a logical sequence of actions are required to complete the task.

³The concepts of active and intelligent objects will be discussed in Chapter 2.

We philosophize that the object-oriented⁴ (Nierstrasz 1989, Shriver & Wegner 1987, Kim & Lochovsky 1989) paradigm provides the theoretical foundation for which we base our model. Our view of active message objects follows from this paradigm and is a natural mechanism for modeling an advanced message system of this type.

This thesis is dedicated towards proposing a conceptual model for such an advanced intelligent message system. Our model is called **IMS**, for *Intelligent Message System*. This model is primarily based on the object-oriented paradigm, and *active messages* (Vittal 1980).

1.7 Organization of Thesis

The organization of this thesis as follows:

- Chapter 2 provides an overview of the concepts and applications of CBMSs with respect to the automation of office tasks. Specifically we research the areas of object-orientation and active objects. The chapter ends with a study of several related models and their application of these concepts.
- In Chapter 3 we define the components of **IMS**. The structure of each component and its function within the system is described. We also define the syntax of a high level script language which is used for programming message scripts.
- In Chapter 4 we continue with our description of **IMS**. We define the major functions of the system and specify a formal definition for each using a formal specification language. We then provide a formal grammar for our script language.
- Chapter 5 offers two examples which illustrate the concepts and functionality of the system defined in Chapters 3 and 4.
- In Chapter 6, the results of this research effort are summarized and areas for future research are identified.

⁴A brief overview of the object oriented paradigm is provided in Chapter 2.

Chapter 2

Literature Survey

Throughout the evolution of Message Management Systems (MMS), many models have been proposed and developed. This chapter presents a survey of the MMSs which are related to the areas of active objects and active messages. The chapter begins with an overview of the foundation on which these models have been based. We then present a survey of several models based on these concepts.

2.1 Foundation

2.1.1 Object-Orientation

The current trend in software development has been a gradual shift towards a methodology known as *object-orientation*. In the object-oriented paradigm, systems are viewed as a collection of objects. Each object is considered a self-contained, autonomous entity that can represent either a physical or abstract entity. Objects are defined in terms of *class* and *sub-class*. An object class contains the set of attributes that uniquely define the characteristics of an object. The ability to define object-classes allows for a hierarchical structure in which a sub-class definitions can inherit the attributes of a super-class. This single characteristic is a fundamental property for which objects can be used to model real-world entities. Objects communicate through *message* passing. If one object needs information from a second object, it can only request the information via a message to the second object.

The object-oriented paradigm provides a mechanism for developing computer systems that maintain a relationship between the objects in the system and the real-world entities

represented by the objects. Much literature has been written in recent years with respect to this topic (Kim & Lochovsky 1989, Shriver & Wegner 1987, Nierstrasz 1989). A summary of the fundamental concepts that this paradigm promotes is enumerated below (Nierstrasz 1989):

1. **Object-class:** the set of attributes (i.e. data and operations) that define the characteristics of a specific object.
2. **Inheritance:** the ability to define object-class structures whereby an object can inherit attributes of a similar object.
3. **Reusability:** the ability to re-use objects without modification to solve new problems.
4. **Polymorphism:** the ability to define a generic operation that can be applied uniformly across a range of objects.
5. **Rapid-Prototyping:** the ability to quickly generate systems by combining and reusing previously defined objects and classes.

The object-oriented approach has been applied across a wide range of applications such as compilers, databases, and graphical-user-interfaces (GUI). As the acceptance of this paradigm increases over time, more creative and innovative applications of these concepts will evolve that will enhance the usefulness of the paradigm. Message systems based on objects are a natural extension to which the object-oriented paradigm can be applied.

2.1.2 Active Objects

As alluded to in the previous section, active objects are an extension of the object-oriented paradigm. Many definitions of an active object have been given due to the fact that many authors interpret the idea of active objects in different ways. In this study an active object must possess the capabilities for autonomous behavior. An active object is defined by Ellis & Gibbs (1985) as:

An object in which a high degree of autonomous responsibility and control is vested. The active object is considered as an independent agent and frequently a source of knowledge and activity.

With this definition, active objects have the ability to initiate asynchronous actions without having first received a message from another object. These objects facilitate a

modularization that encapsulates both control and data within the object. In this manner, a more distributive level of processing can be achieved by delegating a certain level of responsibility to an object.

Hewitt (Hewitt & Baker 1977) provided the earliest theoretical work in the area of active objects. Hewitt characterized the behavior of an active object system in his *actor* model. The actor model emphasizes message passing communication between entities involved in distributive computations. In this model, Hewitt defined two fundamental concepts: *actors* and *events*. Actors are designated as the computational agents within the system while events are used to indicate the arrival of a message at an actor. Actors communicate with each other via message passing. The sender and receiver can proceed in both an asynchronous and parallel manner during and after the message transfer process. In this model, a message itself was considered an actor or active object. Agha (Agha 1986) provided a formal definition for the actor model in which an actor consisted of a mail address and a *behavior*. The behavior was used to specify the set of message types an actor could receive, as well as, the operations to perform on receipt of a message. The actor model is considered one of the forerunners of active object systems because of its ability to encapsulate both control and data into a functioning entity.

2.2 R2D2 - Research-To-Development-Tool

Vittal (Vittal 1980) is credited with performing the initial work in the area of active message systems. Vittal characterized active messages as:

Messages with a mission that the message knows about.

The mission can be described by the intent for which the message was originally created, such as to display a message or collect some information from a recipient. The mission of the message is in the form of a procedure or set of instructions that are executed when the message arrives at a destination. The execution of the mission can take place before or after delivery to a recipient. In general, a message must be aware of the types of valid processing that can be performed on it and not allow unknown actions to be performed. A message must also be able to alter itself at intermediate locations during its' mission to account for changes in its environment.

Vittal developed an experimental message system called the Research-To-Development-Tool (R2D2) which utilizes active messages for office communication. His goal was to develop a message system that was more dynamic and flexible than the traditional message systems. The objectives of the R2D2 system are summarized below:

- Messages are active, executable objects.
- Senders can specify actions that both the message and a recipient can perform on the message after it has been sent.
- Users can customize system functionality, adding new functionality if required.
- Users can tailor the user-interaction style of the system.

The R2D2 system is comprised of the following three major components:

1. **Messages:** Messages are considered to be those that have been received or are in the process of being sent.
2. **Instructions:** These make up the message processing language used to implement the active messages.
 - **Selectors** which are instructions for selecting subsets of messages.
 - **PInstrs** which are instructions for transcribing messages.
 - **CInstrs** which are instructions for composing messages.
3. **User-interface:** This defines the presentation mechanism and the set of commands utilized by a user to perform operations within the system.

In the R2D2 system, all messages are viewed in a strictly textual manner. The fundamental property of this system is that messages are capable of performing certain actions on their own. Messages can also alter their interactions with a user depending upon the responses received by the user. Each message is viewed as a *messenger* with its routing specification stated in the form of a distribution list within the message. The instruction, *Circulate-Next*, is used to determine the message's destinations.

As indicated above, PInstrs are the instructions for transcribing (i.e. printing) a message. PInstrs include operations for output formats, control structures, testing for the existence of a specific field or the contents of a field, and the ability to invoke a CInstr. These instructions also allow the ability to obtain the contents of a field or other information

that is not explicitly stated within the message itself. Transcription of a message can occur in one of two ways. When a message arrives at a recipient site a default PInstr instruction is invoked. This instruction searches the message for the existence of an *Instructions* field. If this field is located, the contents of the field will be used to transcribe the message. Otherwise, the instructions within the recipients environment will be used.

CInstrs are the instruction for composing (i.e. creating) a message. These instructions allow users to specify prompting information, such as the text and placement of each prompt on the display screen, and placement of input data, and data types such as text, date, and numeric.

Also as previously indicated, the user-interface to the R2D2 system is comprised of a user-level command set and a presentation style. The command set defines the way that a user can direct system operation. Presentation style includes how commands are invoked, but also the way in which data appears on the display.

Vittal's system is very important in that it provided validation that the active message concept could be envisioned. Although the system achieved its intended goals, the simplistic nature of the system limits its usefulness in a real-world office environment. The system provides for only a simple routing specification and makes no provisions for decision making or dynamic routing criteria. In the next section, we survey another system that expands on the capabilities of the R2D2 system

2.3 Imail

The Imail system of John Hogg (Hogg 1985) addressed the issues of decision making and dynamic routing. In the Imail system, a message can interact with its recipient, and based on the recipient responses, can decide whether it should route itself to other recipients or terminate itself.

In Hogg's Imail system, a message is referred to as an *imessage*. An imessage is essentially a program. The imessage is composed of a list of questions in the form of a script. In this system, the imessage script is translated into C shell source and it is this source which actually gets executed at the recipient site. An *interaction* is defined to be the running of the script at a single recipient site. During an interaction, information may be collected and

used to determine the next recipient to receive the message. The lifetime of an imessage is called its *execution*. Thus, the execution of the imessage may require many interactions. After an interaction, an imessage is *shipped* to other recipients. Upon termination, the imessage will return to its sender with any information collected during its execution.

As indicated above, the imessage script is composed of a series of questions. Execution of the script is in the form of a *query-response-process* sequence. Each question is printed in the form of a query on the recipient's display terminal. A response is then collected from the recipient after which a list of commands is displayed. This command set provides the capabilities for:

- Processing the responses.
- Modifying the content of the imessage script itself.
- Shipping the imessage to another destination.
- Terminating the imessage.

The notion of intelligence within an imessage is encapsulated in what is termed the imessage *state*. The initial state is set by the originator of the imessage but may be altered at any time during an interaction. The state of an imessage affects the execution, interaction, and shipping of the imessage. That is, during an interaction, questions in the script may be skipped or repeated and afterwards, depending on the input collected, the imessage may be shipped to other destinations or terminated. This is a dynamic form of routing since the imessage can determine its next destination either directly or indirectly from the responses of the recipient.

2.3.1 Imail Prototype System

A prototype Imail system was implemented within a UNIX¹ environment. An imessage is created by creating an imessage script using a special Imail language. The command set for this language is defined as follows:

¹UNIX is a trademark of AT&T

- `>` is used to indicate a question
- **get** is used to obtain a recipient response
- **ship** is used to add a login to the list of imessage destinations provided that the imessage has not already been to the destination.
- **reship** will send the imessage back to a destination regardless of whether it has already been there or not.
- **terminate** immediately terminates an imessage.
- **next** takes a number or question label as an argument and transfer control of script execution to that location.
- **if** is used to conditionally perform commands. *if* commands may also be nested.
- **print** is used to print a message.
- **set** is used to set the values of variables.

Each question in the script begins with a “>” and may contain a one word label which allows for loop control if a question needs to be repeated. The text of the query follows and begins in the leftmost column. Each question line, except the last, must be followed by a *response collection* line. The last line may or may not have a response collection. This response collection line is of the form:

```
get <number> <type>
```

`<number>` specifies an upper and/or lower bound on the number of items in the reply and `<type>` indicates the type of reply which may be *numbers*, *words*, *text*, or *logins*². Examples of this format would be “get 2 numbers” and “get 1 login”.

As indicated above, the set command is used for setting variables. There are three types of variables utilized in the Imail system: *response*, *local*, and *global*. A single response variable is associated with each question in the script and is used to hold the response to that question. These variables are indicated by a “#” followed by the number or label of the question to which it is assigned. As examples, “#1” refers to the response for question 1, “#-2” refers to the response for the question preceding the previous question and “#getname” to the response for question labelled “getname”.

²UNIX user-ids

Local variables are indicated by words prefixed with “!” and global variables are indicated by words prefixed with “?”. Both sets of variables can be initialized at the beginning of an imessage script. However, local variables are reset to their initial value at the beginning of each interaction while global variables retain their values between interactions. Variables appear in the *set* commands, but may also appear in *if*, *ship*, *reship*, and *print* commands, as well as, in the text of queries.

Figure 2.3.1 is an example of an imessage script as specified in Hogg’s paper *Intelligent Message Systems* (Hogg 1985). This script will ask a list of recipients to predict the inflation rate for the following year. After it has collected forty responses, it calculates the average and variance of the responses. If the variance is less than 0.1, the imessage will terminate. Otherwise, it will reroute itself to each of the recipients and repeat the process.

2.3.2 Imessage Routing

To send an imessage to a destination, the originator simply sends the imessage as input to the Imail system along with a list of initial recipients. An optional subject line may be included as part of the imessage. This line appears in a recipient’s header line when the message is received. A timeout value can be specified to indicate a time at which the imessage should automatically terminate. Termination of an imessage may also occur if the list of destinations is exhausted or by explicit termination after executing some task. In any termination, the imessage will return any collected information to the sender .

Notification of imessage arrivals are placed in a recipient’s mailbox. To receive an imessage, the recipient is presented a list of imessage headers. Each header indicates the imessage number, sender, sending date, and subject. An imessage is selected by number and execution of the imessage script can then take place. At this point an interaction with the recipient occurs. When the interaction terminates, the imessage remains in the recipients mailbox. An imessage is only deleted when explicitly requested by a user.

The Imail system provides capabilities that allow a message to interact with a user and to collect information. At each destination, the user is allowed to alter the recipient list which allows for dynamic routing. The system can also perform computations and make routing decisions based on the results of these computations. However, the usefulness of this system over traditional mail system is very much limited. Since an imessage is a

```
number ?n=0
number ?sum = 0
number ?sqsum = 0
number ?maxvar = 0.1
number ?itreps = 40
number ?avg = 4.0
number ?var = 0
>
What do you think the inflation rate for next year will be?
The last average prediction was ?avg.
  get 1 number
  set ?sum = ?sum + #1
  set ?sqsum = ?sqsum + #1 x #1
  set ?n = ?n + 1
  if ?n = ?itreps
    set ?avg = ?sum / ?n
    set !var = ?sqsum / ?n - ?avg x ?avg
    set ?n = 0
    set ?sum = 0
    set ?sqsum = 0
    if !var > ?maxver
      reship
      next last
    print Thanks. Goodbye!
    terminate
>last
Thanks!
```

Figure 2-1: Example imessage script.

program, it has to wait to be executed at each recipient site. An imessage can not initiate its own execution. Because of this shortcoming, timely processing of information can not be guaranteed. With this system, the responsibility of message management remains with the users.

2.4 MMS

The *Message Management System* (MMS) presented by Mazer & Lochovsky (1984) integrates the facilities of a computer-based message system with those of a database management system to create an office information system for managing structured messages. A structured message is associated a specific *message type*. Each message type incorporates a *logical routing specification* that allows the system itself to determine the next destination to forward the message to based on both the contents (fields) of the message and the current system state. This mechanism allows conditional routing as opposed to a predefined, static form of routing and frees the user from having to explicitly defining each route in the path of the message. A *routing specification language* is used to describe the routing specifications for a specific message type.

As previously indicated, each message is defined in terms of a message type. The message type specifies the basic structure of the message and includes both structured (formatted records) and unstructured data (text, video, etc). An instance of a message is stored in a *communication base* which provides the medium for end-user communication. A *communication base administrator* (CBA) is responsible for managing and maintaining the communication base.

A message instance is routed according to the specifications associated with the message type. The routing specification is considered the basic property of the message type. Therefore, if two messages differ only by their respective routing specifications, then they are considered separate types. Recipient destinations of a message are referred to as *sites*. Each site is represented by an *agent* that has been assigned a *role* within the system. A role can refer to either an individual user or a specific function.

2.4.1 Routing Model

The routing of a message is specified in terms of the sites through which the message may pass. Three types of sites may be specified in the routing:

1. Origin site α . This refers to the source of the message which must be created by a role using either manual or automatic procedures.
2. Processing sites σ_i ($2 \leq i \leq m - 1$). These refer to intermediate sites where roles process a message and then forward it.
3. Terminal site ω . This refers to the site where automatic routing of a message terminates.

Each site has an *in-tray*, where messages are placed upon arrival, and an *out-tray*, where messages are placed for automatic routing.

Three forms of routing exists:

1. Type Routing. This type of routing is specified by the CBA during message type design. It becomes part of the message type and thus, applies to all message instances of that type.
2. Instance Routing. This type of routing is specified by the user at message instance creation time and only applies to that message instance.
3. Override Routing. This type of routing is applied when the normal routing specification must be suspended and replaced by manual routing.

Routing can be both conditional and unconditional based upon various criteria such as the values in the fields of the message, the current state of the system, and time constraints. Actions that can be performed include sequential or concurrent routing to other destinations, routing termination, alerting, and rerouting. In sequential routing, the message is sent from recipient to recipient in the order specified. In concurrent routing, the message is sent to all recipients at the same time.

2.4.2 Routing Specification Language

A routing specification for a message type is created using the *Routing Specification Language*. Figure 2-2 shows the general format for the language (Mazer & Lochovsky 1984):

```

SITE <site name> <source/nosource>
  TIME-CASE <condition>:<time constraints>:<actions>
    ...
  :
  TIME-CASE <condition>:<time constraints>:<actions>
  CREATION {valid only if SOURCE specified in SITE line}
    TIME-CASE <condition>:<time constraints>:<actions>
    ...
  :
    TIME-CASE <condition>:<time constraints>:<actions>
  ROUTE-CASE <conditions> TO <next site>
  :
  ROUTE-CASE <conditions> TO <next site>
  FIRST {to be checked upon first visit to this site}
  :
  SECOND {to be checked upon second visit to this site}
  :
  [THIRD,FOURTH,etc]
  OTHER {optional default case}
  :
  ERROR {exception handling}
  :
END-SITE
  :
  [more sites]

```

Figure 2-2: Route Specification Language Syntax

The language allows users to describe message routing specifications that the system can use to route the message. Keywords are presented in all uppercase, while nonkeywords are in lowercase. Meta-symbols are enclosed in "<>". A specification consists of a set of subspecifications, one for each possible site in the routing. For each site, a set of possible destinations is defined and may include specific decision criteria for selecting a destination. <source/nosource> indicates whether the current site is allowed to create other instance of the message type. If the site is defined to be "source", then it is allowed to create a message of that type.

Zero or more TIME-CASE entries may be specified. These constructs are used for specifying time constraints such as time limits on message processing at a site. Each entry specifies a specific time constraint and actions to be performed if the constraint is met. For example,

```
TIME-CASE TIME-LIMIT 7 DAYS:ALERT
```

indicates that the user of the station should be alerted if no action has been taken after 7 days.

The keywords (CREATION,FIRST,SECOND,...) are *visit* keywords and apply to each visit of a message instance at a site. These must appear in ascending order. These keywords are used in association with the ROUTE-CASE keyword for specifying routing conditions. Zero or more ROUTE-CASE entries can be specified. These constructs specify the conditions that must hold true in order for the message to be forwarded to the next site. For example,

```
FIRST
  ROUTE-CASE? status='rejected' T0 rejection-file
  ROUTE-CASE? status='accepted' T0 accepted-file
SECOND
  ROUTE-CASE? status='accepted' T0 accepted-file
ERROR
  ALERT 'Illegal message instance; Contact CBA'
```

specifies that on the first visit to a site, if the value of the status field of the message is "rejected", then route the message to the "rejection-file". If the value is "accepted", then route the message to the "accepted-file". If on the second visit the value is "accepted", then

route the message to the “accepted-file”, otherwise an error condition has resulted and the CBA will be notified. To indicate sequential routing to a site, a “>” is placed in front of the agent name to receive the message (*e.g.* >committee-members). Concurrent routing is indicated with a “&” (*e.g.* &committee-members).

A *selection list* may be used with the “TO” keyword which allows the message to select a qualifying site from the list of potential sites. The general format is:

TO <system qualifier> <selection list>

The system qualifier must be one of (MOST-MESSAGES,FEWEST-MESSAGES,LEAST-LOADED,MOST-LOADED) where the LOADED conditions refer to the CPU load of a site. As an example,

TO LEAST-LOAD <site-1> OR <site-2> <site-3>

indicates that the message should route to the next site, either site-1, site-2, or site-3, that has the lowest CPU load.

Mazer’s MMS system provides the framework for the logical routing of messages. The system is based on structured messages in which a routing specification can be associated with the message type. A route specification language is utilized for creating the routing specifications.

2.5 KNOS

In D. Tsichritzis’ *KNO* model (Tsichritzis & Gibbs 1987), an object-oriented environment is proposed in which active objects, called *K*nowledge *O*bjects, are utilized to acquire and disseminate knowledge. In this model, KNOs are highly intelligent objects that migrate to different environments and learn from their surroundings. The goal of the KNO model is provide the facilities for defining an advanced tool for the office environment that can automate certain office tasks that involve cooperation, negotiation, and apprenticeship learning. Tasks that have these characteristics are typically very difficult to model. However, the concept of KNOs and the KNO environment provides the facilities to effectively model such tasks. The next section describes the KNO environment.

2.5.1 KNO Environment

Each KNO exists as a self-contained and highly autonomous entity that maintains complete control over its own behavior. KNOs exist within the framework of a *context*. A context in the KNO environment is defined to be a set of cooperating KNOs. Physically however, a context is usually associated with a workstation. Communication within a context is facilitated through a *blackboard*. A blackboard is a shared construct that exists within each context. If a KNO wishes to communicate with another KNO, it simply posts a message to the blackboard with the information for the KNO to read and awaits a response. This type of indirect communication allows a KNO to maintain its autonomy as it moves between contexts. KNOs are capable of spawning other KNOs called *limbs*. Limbs must always stay in contact with the parent KNO or *head* KNO. This is achieved through the use of agent objects described below. A head KNO combined with all its limbs defines a *complex* KNO. A complex KNO can execute a distributive task.

Each context is administered by an *object-manager*. The role of the object manager is to manage the use of the bulletin board and to oversee the migration of KNOs to and from the context. It is also the responsibility of the object-manager to decide if it should acquaint itself with object-managers from other neighboring contexts.

Two special forms of KNOs exist: *user* and *agent*. A user KNO is one whose behavior can be controlled or manipulated by an external user. For example, a user KNO could be directed to monitor and report the activity within the KNO environment. An agent KNO functions as a representative for the object-manager of a particular context. The behavior of an agent KNO can be indirectly controlled by KNOs from other contexts. Agent KNOs are responsible for managing the migration of KNOs and forwarding messages to other environments, such as messages from a limb KNO to its head KNO.

2.5.2 KNO Structure

The structure of a KNO is defined in terms of *class*. A KNO class defines both the *behavior* and *structure* of a KNO. The behavior of a KNO is defined by the set of operations that the KNO is allowed to perform while its structure defines the set of instance variables (data structures) utilized by the KNO. Figure 2-3 represents the general form of a KNO class.

The class *basic-kno* is a predefined KNO class from which all KNOs are derived. This

```
(kno-def (Kno class name)
  ((instance variable list))
  ((inheritance list))
  ((kno-operations
    (Kno class name) (Kno operation name))
    ((parameter list))
    ((body))
  :
  )
```

Figure 2-3: General Kno class form.

```
(rule (rule-name)
  (trigger (trigger condition))
  (action (action series)))
```

Figure 2-4: KNO Production Rule.

class defines the basic structure and behavior that is common to all KNOs. When defining a new KNO class, the structure and behavior of a previously defined class can be inherited by the new class. Thus, the new class will contain all the instance variables and operations of the base class, inherited class, and those defined within the new class.

Each KNO class definition defines a specific set of operations. Each operation defines sequences which messages are communicated between KNOs through the local bulletin board of a context. Operations in the KNO model are specified in the form of *production rules* (Patterson 1990) as shown in Figure 2-4.

Each production rule is specified by a *rule-name*. The *trigger condition*, also known as the *antecedent*, specifies a simple boolean expression. The *action series*, or *conclusion*, specifies a list of actions to be executed. When a trigger condition for a rule evaluates to true, all action in the action series list are executed. Table 2.1 lists the set of allowable

Action Type	Actions
Local	Put, Get
Communication	Import, Export
Existential	Spawn, Die, Move, Freeze, Unfreeze
Learning	Act, Learn, Unlearn
Limb	Grow, Kill, Ship, Teach, Unteach

Table 2.1: KNO Actions by Type.

actions according to the type of action.

- Local actions allow modification or inspection of instance variables.
- Communication actions allow KNOs to interact with each other via the local bulletin board.
- Existential actions allow KNOs to migrate, create other KNOs, terminate, and to change to and from a static representation during migration.
- Learning actions allow KNOs to teach, learn, and unlearn operations to and from other KNOs.
- Limb actions allow a KNO to create and manage representative KNOs when attempting to gather information that is distributed throughout multiple environments.

The KNO model is a very important in that it provides a powerful framework for modeling tasks that are often non-repetitive or require negotiation and cooperation. In the next section we begin our discussion of several messages systems that are based on the concepts of active objects. In particular, we will focus on message management systems that employ the use of *active messages* to carry out certain office tasks.

2.6 Conclusions

Each of the models presented above share many similar characteristics but at the same time are quite different in terms of providing a functional mechanism for utilizing active messages to perform routine and non-routine tasks. However, the fundamental characteristic

that each model has in common is the use of an electronic message as a functional entity that can actively participate in the execution of a task. Our model can be characterized in a similar manner, however, we enumerate two important distinguishing characteristics with respect to our proposed model.

First, a message is as a collection of independent and autonomous objects that cooperate together to perform the task assigned to the message. Each component object is responsible for performing a specific task during the lifetime of the message. This task is referred to as the *function* of the object and is defined by the creator of the message. Objects can communicate with other objects, as well as, external entities such recipients of the message. Second, each message object exists as a intelligent functional entity. A message can react to external events during the routing process and take appropriate actions as specified by the creator of the message. Further, a formal language is defined which provides a flexible set of commands which are be used for programming message tasks. This language provides a wide array of options for modeling office tasks. In the next chapter we introduce our model for an advanced computer-based message system which we call **IMS** for Intelligent Message System.

Chapter 3

IMS - Intelligent Message System & Operational Semantics

In this chapter we present our model which we have called **IMS** for *Intelligent Message System*. We begin with a brief overview of the components which define the **IMS** system. We then provide a detailed description of the structure of each individual component along with its function within the system.

3.1 IMS

The goal of the **IMS** system is to extend the domain of the computer based message system by providing the resources and facilities which allow a user to program a message to execute certain tasks that would normally be manually performed. The system is predicated on several constructs that allow timely and efficient flow of information through the system. We begin our presentation of **IMS** with an overview of the system components.

IMS is an object-oriented based model in which messages, called *intelligent message objects*¹, can be programmed to perform certain tasks. Structurally, each component within **IMS** is viewed as an object, and thus, possess the structural abstraction provided by the object-oriented paradigm. **IMS** is designed to operate within a networked-based environment that consists of one or more processing sites. This environment must support a

¹Intelligent message object and message object will be used interchangeably.

communication subsystem that allows individual sites to communicate and provides reliable and efficient transmission of messages. Each individual site must also provide the facilities for concurrent execution of multiple processes. The fundamental components of **IMS** are listed below:

- **IMO** - Intelligent Message Object defined in section 3.2.
- **IMSL** - Intelligent Message Script Language defined in section 3.5.
- **SMM** - System Mail Manager defined in section 3.6.

The System Mail Manager defines the physical environment for the system. Each site that wishes to utilize **IMS** must have a SMM executing within its domain. The SMM provides the facilities for managing intelligent message objects, as well as, establishing an end-point of the system which allow users to communicate. SMM allows for the creation of intelligent message objects. Each IMO logically exists as a functioning entity within the system through which the SMM provides the conduit that allows the IMO to carry out its assigned task. Each IMO is a composition of objects which consists primarily of the RCO and MCO. The RCO is responsible for routing the IMO through the system while the MCO is responsible for carrying out the assigned task at each recipient site. The IMSL is a high-level interpretive language which allows a user to program a task that an IMO can execute. The IMSL command set provides a basic set of constructs for simple logic flow and user interaction. Interpretation of an IMO script is a function of the SMM at a specific site.

3.2 IMO - Intelligent Message Object

As previously defined, an IMO is the end product of the SMM and exists in the form of a multi-object entity. An IMO is essentially an abstract data type (Gutttag 1977) with instructions to route a message, process a message, and user interactions embedded in the message object's *script* (i.e., the executable part of message). As a result, an IMO can perform a role analogous to a processor executing instructions of a program and thus it

can directly execute the actions which are needed to perform the task assigned to it by its creator². For example, when a message object arrives at a given station, it can execute the code associated with that recipient station. The actions may also have time-dependent constraints, for example, if there is specific time difference between the creation time of the message and the current time, the message object would terminate its mission and archive itself in the originator's station. Also, since a message object may have routing information in itself, it can deliver itself to a next station if the current station doesn't interact with it after some period of time. This approach would provide a better user interface since the users are released from burden of details for routing procedures.

An IMO is defined in terms of having two distinct responsibilities. The first, referred to as the *IMO primary function* is to carry out the task that the message has been assigned (i.e. programmed) when it arrives at each of the intended destinations. The IMO script dictates what the task entails. This could be as simple as displaying a textual message or retrieving data from the recipient. The task of an IMO will be referred to as the *IMO mission*. The second responsibility, referred to as the *IMO secondary function*, is to successfully route itself to a set of intended destinations performing any tasks necessary to achieve the routing requirements.

The internal structure of an IMO is shown in Figure 3-1. It defines three internal objects that collectively form the *IMO space* which represents an *instance* of an IMO. The three objects are defined as follows:

- Message Control Object (MCO)
- Route Control Object (RCO)
- Global Data Object (GDO)

Each of these objects function as an independent and autonomous entity. The Message Control Object maintains the activities associated with the primary IMO function while the Route Control Object maintains the activities associated with the secondary IMO function. The Global Data Object maintains a shared data structure which allows data that is global

²The Intelligent Message Script Language (see Section 3.5) is used for programming IMOs

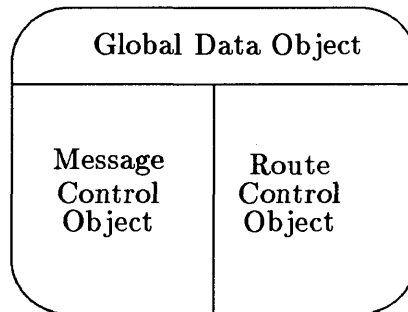


Figure 3-1: Intelligent Message Object Internal Structure.

within the IMO to be processed and stored. It also serves as a medium which allows the RCO and MCO to communicate. Each object is governed by an *object manager*. The object manager is responsible for managing the activities within the object, as well as, any external interactions with other entities. Each of these objects will be defined in detail in the following sections.

3.2.1 Global Data Object

The internal structure of the Global Data Object (GDO) is shown in Table 3.1. The GDO is a shared static structure that is used for maintaining specific global data with respect to the IMO. Access to data in this structure is accomplished via the GDO *Object Manager* (OM). The GDO OM is responsible for managing access and modification of the data in this structure. The OM will determine if the request is valid and will either accept or reject the request. No other object can directly access this data. The function of each field is as follows:

- **IMOID** is a system generated identifier that uniquely defines the message with respect to the originator.
- **CreateTstmp** is the creation time of the message.

Global Data Object		
Field Name	Field Type	Configured
IMOID	Int	System
CreateTstamp	Time	System
OrgId	Char	System
OrgAddr	AlphaNum	System
CurrDest	AlphaNum	System
CurrDestAddr	AlphaNum	System
NextDest	AlphaNum	System
MsgBlkPtr	Pointer	System

Table 3.1: Global Data Object Internal Structure.

- **OrgId** identifies the originator of the message.
- **OrgAddr** identifies the network address of the originator.
- **CurrDest** indicates the current recipient identifier.
- **CurrDestAddr** indicates the current recipient network address.
- **NextDest** identifies the next destination of the message.
- **MsgBlkPtr** pointer to the internal data store.

As an IMO arrives at a recipient site, interactions with the user, SMM, and other objects will require that fields within this structure be updated. The OM will ensure that the updates are performed atomically to preserve integrity of the data. In Section 3.5 we show how these fields can be accessed during IMO script processing through constructs specified within the Intelligent Message Script Language (IMSL).

3.3 Route Control Object (RCO)

The RCO is an independent and autonomous object within the IMO. It encapsulates all the data and functionality for determining the logical routing path for the IMO which is referred to as the *secondary IMO function*. This object can be configured either statically by the user upon creation of an IMO or dynamically during the actual routing of the IMO

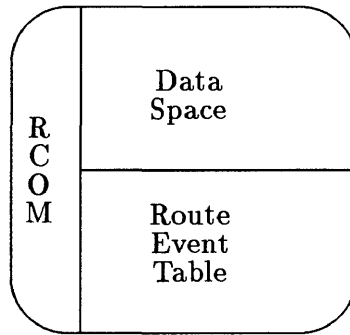


Figure 3-2: Route Control Object Internal Structure.

through the system. The internal structure of the RCO, shown in Figure 3-2, contains the following internal structures:

- Data Space
- Route Event Table

Each of these structures will be described in following sections.

3.3.1 RCO Data Space

Table 3.2 defines the internal structure of the RCO Data Space. This region is composed of a set of data structures which are used for maintaining routing context as the IMO moves to each destination. The function of each field within this structure is defined below:

- **Priority** indicates the priority of the message. Valid priorities are:
 - **NORMAL** (default)
 - **URGENT**
- **DestList** indicates the intended recipient(s) of the IMO.
- **DestListType** indicates type corresponding to the DestList field. Valid list types are:
 - **USER** (single entry)

RCO Data Space Fields		
Field Name	Field Type	Configured
Priority	Enum	User/System
DestList	Char	User
DestListType	Enum	User
DlvConfirm	Boolean	User/System
AccessConfirm	Boolean	User/System
RetToOrg	Boolean	User/System
Spawn	Boolean	User/System
ITV	Int	User/System
ArriveTstamp	Time	System
DLAccess	Enum	User/System

Table 3.2: RCO Data Space Internal Structure.

- **GROUP** (list of entries)
- **ALIAS** (resolves to **USER** or **GROUP**)
- **DlvConfirm** indicates if the sender should be notified when a message arrives at a destination. Valid entries are:
 - **Y** (Yes)
 - **N** (No - Default)
- **AccessConfirm** indicates if the sender should be notified when a message has been accessed by a recipient. Valid entries are:
 - **Y** (Yes)
 - **N** (No - Default)
- **RetToOrg** indicates if the message should return to the sender. Valid entries are:
 - **Y** (Yes)
 - **N** (No - Default)
- **Spawn** indicates if the message can create a copy of itself. Valid entries are:
 - **Y** (Yes)
 - **N** (No - Default)
- **ITV** (interrupt-timeout value) specifies the amount of time the message will wait for a recipient response before proceeding with its next action. Values must be numeric and greater than or equal to zero.

- **ArriveTstmp** indicates the arrival time of the message at each destination.
- **DLAccess** indicates if a recipient is allowed to modify the DestList entries. Valid values are:
 - **NONE** - cannot access the list.
 - **DELETE** - can delete entries from the list.
 - **ADD** - can add entries to the list.

These fields describe the attributes which the RCO OM utilizes during the IMO routing process. In order to increase the efficiency of route configuration creation, **routing templates** can be utilized. A routing template is a predefined routing configuration that can be retrieved from a template database and used for the current routing requirements. If needed, the configuration can be modified as needs dictate. An example would be a routing template that is used for sending stock information out to company stockholders. Any route configuration can be archived for later re-use. Default values can be specified in the IMS system configuration file.

As indicated above, the *DLAccess* field defines the access privileges that a recipient has with respect to the destination list. Since the originator maintains jurisdiction over the IMO, permission must be given to the recipients to allow modification to this list. It may be the case that a recipient knows another individual that might be interested in the IMO or there may be a recipient specified in the list that should not receive the message. This provides a mechanism for allowing dynamic modifications to the destination list.

3.3.2 RCO Route Event Table

The RCO Route Event Table, as shown in contains the “routing intelligence” within the RCO. The structure of this table is shown in Figure 3.3. Each row in the table contains an *event* and *action* entry. An event corresponds to a particular state that may occur during message routing such as destination not available. The action associated with that event is the function that is to be taken should that event occur. For example, if a destination is unavailable then an event will be signaled to the OM indicating the condition. If that event is specified in the table then the associated action will be performed. The action could, for instance, specify that the OM route the IMO to its next destination and update its destination list to return to the unavailable node at a later time. An event can be entered

Event Table	
Event	Action
E_1	A_1
E_2	A_2
\vdots	\vdots
E_n	A_n
$E\alpha$	$A\alpha$

Table 3.3: RCO Route Event Table.

without an associated action, however, it is illegal to enter an action without an associated event. The event $E\alpha$ and action $A\alpha$ are special case entries that must be included in the table. These entries will be derived based on the following rules:

1. If an event E_i is entered without an associated action, then $A\alpha$ will be derived as the action for any occurrence of that event.
2. For all other events E_j not entered in the table, $E\alpha$ and $A\alpha$ will be derived as the default event and action respectively.

Events can correspond to either standard events or non-standard events. *Standard* events are those events that arise as a result of normal IMS processing. *Non-standard* events are those events that are common to network routing. Table 3.4 indicates the valid events defined in the IMS system.

If a specific event is not entered in the event table then $E\alpha$ and $A\alpha$ will be derived as defined above. As a specific event occurs during routing, the routing table is scanned to determine how to handle the event. When a match to an entry in the event column is found, the action will be carried out. The set of valid actions that can be specified are defined in Table 3.5. Upon creation of an IMO, the user must specify one of the valid actions for each event that is defined within the system. For example, the sender may specify the action RETURN to be performed for the event ITVTimeout which indicates that the IMO should return to its sender if a timeout occurs while the IMO is awaiting a response from a recipient.

RCO Events		
Event ID	Type	Indication
ITVTimeout	Std	ITV timer expiration
AccessInit	Std	IMO access initiated
NoDest	Std	SMM no destination
RecvDeny	Std	SMM receive denial
NoResource	Std	SMM no resources
DestUnavl	NonStd	Dest. not available
DestNoExist	NonStd	Dest. does not exist
NetwUnavl	NonStd	Network unavailable
NetwErr	NonStd	Network error

Table 3.4: RCO Event Specifications.

RCO Event Actions	
Action ID	Function
NOTIFY	Send notification to sender
DELETE	Delete current destination
FORWARD	Proceed to next destination
TERMINATE	Terminate mission
RETURN	Return to sender

Table 3.5: RCO Route Event Table Actions.

3.3.3 RCO Manager

Each RCO is governed by an *Route Control Object Manager* (RCOM) which serves as the administrator of and interface to the RCO. All communication to the RCO is directed through the RCOM. The main functions of the RCOM are listed as follows:

- Manages access to its internal data structures.
- Coordinates dynamic modification of the internal event table.
- Communicates with the local SMM.
- Monitors event status within a recipient message queue.
- Relinquishes control to the SMM after arrival at a destination

The RCOM institutes the routing mechanisms within the message. It is called upon to route the message to the specified recipients in accordance with the event table specifications. When an IMO arrives at a recipient destination, the SMM will place the IMO on a receive queue and the recipient will be notified of a new message arrival. If an ITV was specified, a timer will be started and the RCOM will be notified if the timer expires. If a timeout occurs, the event table action for the event will be performed. Once the recipient takes action upon the message (i.e. to read the message), control will be transferred to the MCO whereby the message mission can then be executed.

3.3.4 Summary

The RCO exists as an active object within an IMO. It is responsible for logically routing the IMO to each destination specified in its destination list. A route event table is maintained within the RCO which specifies an event/action 2-tuple. Events are signaled to the RCOM when an activity takes place that can potentially interrupt the mission of the IMO. The action specifies how to handle the event if this should occur.

3.4 Message Control Object (MCO)

Like the RCO, the MCO is an independent and autonomous object within the IMO. It encapsulates all the data and functionality for executing the *primary IMO function* or

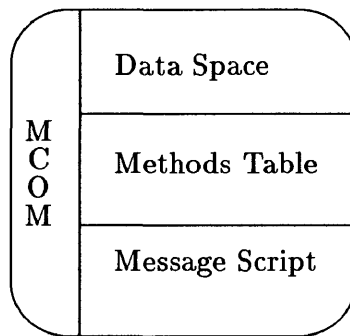


Figure 3-3: Message Control Object Internal Structure.

message mission that has been assigned to the IMO. This object is configured statically by the user upon creation of an IMO. The internal structure of the MCO, shown in Figure 3-3, contains the following internal structures:

- Data Space
- Methods Table
- Message Script

Each of these structures is defined in the following sections.

3.4.1 MCO Data Space

Table 3.6 defines the internal structure of the MCO Data Space. The region is composed of a set of data structures which are used for maintaining context associated with the execution of the mission of the message at a recipient site. The function of each field within this structure is defined below:

- **Subject** specifies a short description of the content of the message which is displayed to the recipient.
- **AccessAttr** indicates the access attributes that a recipient has with respect to the message. Valid attributes are:

MCO Data Space Fields		
Field Name	Field Type	Configured
Subject	Char	User
AccessAttr	Enum	User/System
AccessTstmp	Time	System
MTPtr	Pointer	System
RspStore	Pointer	System
MScriptPtr	Pointer	System

Table 3.6: MCO Data Space Internal Structure.

- ARCHIVE
 - FORWARD
 - MODIFY
 - VIEW
 - PURGE
 - PRINT
- **AccessTstmp** indicates when the message was accessed by a recipient.
 - **MTPtr** pointer to a table of user-defined methods that can be invoked through message script processing.
 - **RspStore** data store for managing recipient response information.
 - **MScriptPtr** pointer to the message script that defines the IMO message mission.

These fields define the context which allows the message mission to be executed at the recipient site. The data section is composed of *private* and *public* data regions. The private data region contains data that cannot be accessed by any recipient of the message other than the originator while the public data region is accessible by any recipient of the message. The purpose of the private data region is to provide a means to store system generated data, such as access timestamps, and user supplied responses to any queries that the originator might provide.

As indicated above, the *AccessAttr* field defines the access attribute that a recipient has with respect to a message. These attributes define the set of standard operations that

the user can apply on the message. Standard operations are made available by the SMM and exist at each recipient site. With this approach these methods do not need to be encapsulated within the message thus reducing the size of the message. As an example, if an IMO has the **ARCHIVE** attribute enabled then the standard operation for archiving messages will be available when the user accesses the message. If this field is left blank, then all operations will be made available. This strategy allows the originator to maintain jurisdiction over the message throughout its lifetime.

3.4.2 MCO Methods Table

The MCO Methods Table consists of a set of user-defined support functions, called *methods*, that can be called from the MCO message script during script processing. These are typically generic routines that provide a particular function such as retrieving a name and address from a recipient. Each routine is written using IMSL. Access to these routines is via the *MTblPtr* entry in the MCO Data Space. Each entry in this table points to either a single method or a set of methods that have been combined into a method *library*. A master index is maintained for each table entry that maps the method(s) that are located within each entry. During message script processing, a method in this table can be invoked by accessing the appropriate index into the table for the desired method.

3.4.3 MCO Message Script

The MCO Message Script contains the “intelligence” within the MCO. The message script is a set of user-defined instructions that defines the actual mission of the message. The script itself is composed utilizing the *Intelligent Message Script Language* (IMSL). IMSL is a high-level language that provides a simple command set which can be used to program an IMO task. The command set allows for user I/O, simple logic flow, and manipulation of data fields within the GDO. Section 3.5 defines the constructs of this language in greater detail. The message script is interpreted as a function of the SMM at the recipient site. Each instruction is carried out in turn with user-defined operations accessed as described above. Since the potential exists for a script to become quite large in size, an interpretive language is used as opposed to an actual executable script image. This significantly reduces the size of the message which in turn increases the efficiency of message processing throughout the

system.

3.4.4 MCO Manager

Each MCO is governed by a *Message Control Object Manager* (MCOM) which serves as the administrator and interface to the MCO. All communication to the MCO is directed through the MCOM. The main functions of the MCOM are listed below:

- Manages access to its internal data structures.
- Communicates with the SMM.
- Provides access to the message script and response store for script execution.
- Manages recipient supplied input.
- Relinquishes control to the SMM upon completion of the message mission.

The MCOM is called upon to carry out the mission of the message once a target destination is reached and the recipient selects an IMO for processing. When an IMO is selected, the MCOM passes pointers to the message script and response store to the script processor within SMM. The script processor will then interpret and carry out the associated script instructions. The script itself may vary in complexity ranging from simply displaying textual message to stepping the recipient through an automated survey of some topic while collecting and storing responses to the questions. When the mission of the message is completed, the MCOM returns control back to the SMM which in turn allows the RCO to evaluate its destination list to select the next destination to transfer to.

3.4.5 Summary

The MCO exists as an active object within an IMO. It is responsible for carrying out the mission that the active message has been assigned. The MCOM provides the interface to the MCO and is responsible for managing its internal processing. The message script provides the intelligence within the MCO. User defined functions can be created and included in the methods table of the MCO. These functions can be accessed through the message script processor.

3.5 IMSL

In this section we define the Intelligent Message Script Language, IMSL. As indicated in previous sections, IMSL is a high-level interpretive script language that is used for programming IMO tasks. The language consists of a relatively simple command set but allows for the creation of very powerful scripts that can perform complex tasks. The language allows for user-definable functions, simple logic flow, variable assignments, and access to data within the GDO. No explicit declaration of variables is required. A variable is instantiated at the time of its use during the execution of the script. Its type is determined by the context in which it is used. The script is interpreted from top to bottom with the first command in the script being the entry point. After a script has been generated an optimization is performed to reduce the size of the script. This is necessary to reduce the overall size of the IMO to improve the efficiency of IMO throughput.

3.5.1 IMSL Variables

Variables by default are initialized to an “empty” value. The value of a variable can be determined by prefixing a “\$” to the variable name as in “\$<variable>”. For example, if *COUNT* is a variable that has been set to the value 10, then \$*COUNT* represents its value. Global variables are specified by prefixing the variable name with “@” as in “@<variable>”. The values assigned to global variables are persistent throughout the lifetime of the IMO. For example, “@LASTDEST” represents a global variable that maintains the value of the last recipient of the IMO.

A set of predefined variables exists that can be used within a script to access certain attributes with respect to the IMO. Table 3.7 shows these variables and the function of each. These variables are accessed in the same manner as described above.

3.5.2 IMSL Command Set

The IMSL command set is shown in Table 3.8. The syntax of each command is defined as follows:

- **SET** (variable) (value)
This command sets (variable) equal to (value). If a value is not specified the variable

Predefined Variables	
Variable Name	Output
IMOID	IMO Identifier
DESTS	Destination List
DEST	Current destination
NEXTDEST	Next destination
SUBJECT	MCO subject line
SENDNAME	Name of the sender
RECNAME	Name of the recipient
DATE	Current date
TIME	Current time

Table 3.7: IMSL Predefined Variables.

IMSL Commands	
Command	Function
SET	Assign a value to a variable
INPUT	Accept input from a recipient
DISPLAY	Display a message or variable
STORE	Store a value
PROMPT	Prompt a query and receive input
CASE	Selective control flow mechanism
IF	Evaluate an expression
GOTO	Control flow mechanism
FUNCTION	Define a new method
CALL	Call a user defined method
RETURN	Return from user defined method
EXIT	Terminate script processing

Table 3.8: IMSL Command Set.

is initialized to “NIL”.

- **INPUT** (variable) (type)
This command accepts and stores recipient input into the specified variable. If a variable is not specified, an internal variable is used which is referenced by “\$0”. The type of input corresponds to **CHAR** | **LINE** | **PARA** | **NUM** | **DT** where **CHAR** is a single character, **LINE** (default) is a single line, and **PARA** is multi-lined input, **NUM** is numeric, and **DT** is Date and Time (mm-dd-yy:hh:mm).
- **DISPLAY** [(variable) | [text]]
This command displays the contents of the specified variable or all text and variables enclosed between the opening and closing brackets.
- **STORE** (variable)
This command allows the value of a variable to be stored in the response store of the current recipient.
- **PROMPT** (query text) (response options)
This command will prompt the recipient with the specified query text and then wait for a response. If the response is one that is listed in the (response options) the input will be accepted, otherwise the query will be prompted over. An internal variable referenced by “\$\$” holds the index into the response options that corresponds to the accepted value. The query text must be enclosed in quotes.
- **CASE** (evaluator) (id : commands) **ENDC**
This command works in conjunction with the **PROMPT** command. The value of “\$\$” is used to select the id and commands to perform. An indirect jump is then made to the id corresponding to “\$”.
- **IF** (expression) **THEN** command [**ELSE** command]
This command allows conditional execution of commands. **ELSE** is optional.
- **GOTO** (label)
This command allows direct branching within the message script.
- **FUNCTION** (function identifier) (args) (body) **ENDF**
This command allows for the creation of a user-defined function.
A set of related functions can be combined into a single unit that can be included in an **IMO**. These functions can be referenced through the methods table in the **MCO**.
- **CALL** (user-defined function)
This command allows a user-defined function to be executed.
- **RETURN**
This command allows a return from a call to a user-defined function.
- **EXIT**
This command allows immediate termination of message script processing.

IMSL Operators				
Relational		Arithmetic		Logical
==	Equal	+	Addition	AND
<>	Not Equal	-	Subtraction	OR
<	Less Than	*	Multiply	NOT
>	Greater Than	\	Division	
≤	Less or Equal			
≥	Greater or Equal			

Table 3.9: IMSL Relational, Arithmetic, and Logical Operators.

Each command may be preceded by a label identifier which can be used for branching using the GOTO and CASE statements. Labels can be up to eight alpha-numeric characters in length and must be preceded with a semi-colon (:). For example, “:LABEL” and “:LABEL001” are valid labels. In addition to the above command set, IMSL supports the use of the standard arithmetic operators and relational operators as shown in Table 3.9.

Data for each recipient site is maintained within the MCO RspStore. This structure maintains responses by destination identifier so the when the IMO returns to the originator, that data can be processed and interpreted as required. Termination of message script processing is done directly through the EXIT command or indirectly after the last command in the script, the script to interpret. As with route specifications, IMSL scripts can be archived in a IMSL database for re-use at a later time. This expedites the setup of IMOs which share similar characteristics.

3.6 System Mail Manager

Each end-user node within the message system has associated with it a *System Mail Manager* (SMM). The SMM is the interface to IMS and provides the user with the capabilities for creating, sending and receiving, and administration of messages within the system. Another function of the SMM is to instantiate resource objects that carry out other tasks on a periodic basis. For example, the SMM will instantiate a *cleanup* object to purge all

messages that have been marked for deletion by the user. The SMM is also responsible for providing communication channels for the various objects.

3.6.1 Interprocess Communication

The SMM maintains a shared memory block, called the *message center* at each site. The purpose of this region is to provide a mechanism for IMOs to communicate with each other. This region is maintained as *passive* structure. It is the responsibility of each object that accesses the region to be considerate and respectful of other objects that are currently utilizing the region. IMOs may read any of the posted notifications from other IMOs as there are no security measures in this region. For example, a message may notify other messages of its arrival at a destination by posting its message ID to the block for other IMOs to read.

The SMM is associated with three support environments that provide all the necessary facilities for end-user operation of the message system. Within each environment, the SMM will create an instance of a *resource object* that will be responsible for performing the function of that environment. Figure 3-4 represents the architecture of the SMM whose components will be described next.

3.7 IMO Administration Environment

The *IMO Administration Environment* provides a collection of support routines that allow a user to maintain those messages that are currently stored in the users *IMO folder*³. The IMO folder is simply a storage facility that provides the recipient a location to save messages. This environment provides the following support facilities for IMO folder entries:

- **View** execute message script processor.
- **Print** allows printing of selected entries.
- **Purge** allows purging (i.e. deleting) of selected entries.
- **Cancel** cancel pending action.

³The only messages that can be stored in the message folder are those that were received with the ARCHIVE MCO access attribute enabled.

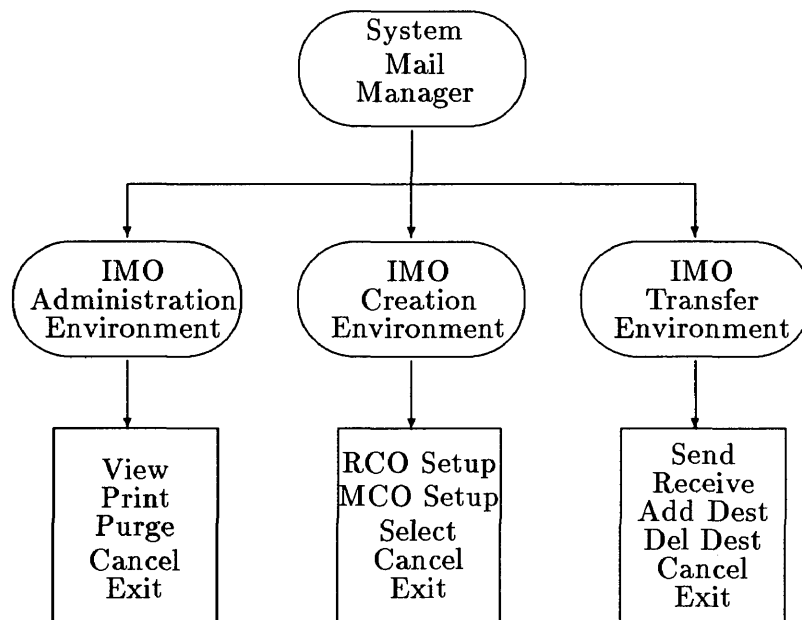


Figure 3-4: System Mail Manager Functional Representation.

- **Exit** exit environment.

In this environment, the user first selects and marks those messages in which processing is desired. Once selected, the user may perform any of the valid operations as indicated through the MCO access attribute. The MCOM is instantiated for each selected message one at a time and the message script is executed for the current MCOM. Processing in this manner is analogous to a newly arrived message.

3.8 IMO Creation Environment

The *IMO Creation Environment* provides the capabilities for creating intelligent message objects. When the user enters the this environment, all the facilities necessary to create a new IMO (or modify an existing IMO) are accessible. The following support facilities for the creation of IMOs:

- **RCO Setup** Setup the RCO specifications.
- **MCO Setup** Setup the MCO specifications.
- **Select** Select an existing IMO for re-use.
- **Cancel** cancel pending action.
- **Exit** exit environment.

In this environment, the user can either create an entirely new IMO or modify an existing IMO that closely represents the current user requirements. Once an IMO is created it can be stored in the form of a *IMO template*. An IMO template database is utilized for maintaining the templates. IMO templates offer the feature of being able to reuse existing information and declarations to reduce the amount of time it takes to create and send a message. All information that is pertinent to the message is entered and the message script is constructed. Upon the completion of creating the message, it is placed in a message distribution queue and assigned a unique ID that is used to identify the message for later processing.

3.9 IMO Transfer Environment

The *IMO Transfer Environment* provides the user the facilities for sending and receiving active messages. Within this environment the following facilities are provided:

- **Send** send an IMO.
- **Receive** receive a IMO.
- **Add Dest** add a destination.
- **Del Dest** delete a destination.
- **Cancel** cancel pending action.
- **Exit** exit environment.

In this environment, a user of the system can send and receive IMOs. Two queues are maintained by the SMM for this purpose. The first is an IMO send queue in which IMOs are placed when they are ready to be sent. The RCOM notifies the SMM of its initial destination and waits for the indication of delivery. The actual delivery mechanism is managed by the underlying communication subsystem.

For each IMO that is to be sent, the SMM instantiates a *Message Transfer Object* to handle the request. This allows the SMM to perform other tasks as necessary. The MTO inherits the identity of the RCOM which allows it to process events and determine which action to perform.

At the receiving end, the SMM is notified by the communication subsystem of the arrival of a message. The SMM instantiates an MTO which inherits the identity of the RCOM of the newly arrived IMO. Each IMO is placed on a receive queue and the recipient is notified of the new IMO arrival. The MTO then starts the ITV timer, if specified, and awaits a response from the recipient. If a timeout occurs, the action specified for that event is performed by the MTO. If the recipient responds to the IMO, the MTO notifies the SMM which then instantiates a *User Agent Object*. The MTO relinquishes control to the UAO. This object then inherits the identity of the MCOM for that IMO. At this point the UAO has access to all the internal structures of the MCO and can then carry out the message mission by executing the IMO script.

3.10 Operational Semantics

In this section we describe the operational semantics of IMS. The functionality of the system can be categorized into the following distinct events:

1. IMO Creation
2. IMO Routing
3. IMO Receipt

Each of these events is discussed in the following sections.

3.11 IMO Creation

To create an IMO, the user would enter the IMO Creation Environment. To access this environment, the SMM will create a resource object that will perform the functions specified by the user. At this point the SMM can return to monitoring other tasks. All further activities will then take place between the user and the resource object until the user is ready to send the IMO or terminate the session.

The user can now either select a previously defined IMO from the IMO database or create a new IMO. If a previous IMO is used, the MCO and RCO would be checked to insure that the configurations within these object correspond to the user's intent for the IMO. If changes are necessitated, the user simply updates the appropriate configuration. If a new IMO is desired then the user will have to specify a new configuration for each object. Each IMO is given a system generated identifier that can be used to track the message through the system. IMOs are also timestamped with the current date and time. Many of the configuration parameters can be defaulted using the using the IMS system configuration file. In general, the user must define the following characteristics of the IMO:⁴

⁴Although the initial setup of a new IMO requires much effort, later IMOs can use the characteristics of an archived message thus requiring minimal effort.

- MCO Script and Method Entries.
- MCO Subject.
- MCO Access Attributes.
- RCO Recipient Destinations.
- RCO Events and Actions.
- RCO Attributes (ITV, Delivery Confirmation, etc.)

The MCO access attributes that are assigned to a message determine the set of operations that can be performed on a message at a recipient site. These attributes effectively limit the level of access a recipient has to a given message much like the permission settings of files in the UNIX file system. The list of valid access attributes was listed in Section 3.4.1.

These attributes can be combined to allow multiple operations on a message. For example, if the originator requires that a recipient of the message be able to add comments to the message (i.e. MODIFY), forward the message to other destinations, but not archive the message, then the attributes would be set as:

MCO.AccessAttr \Leftarrow MODIFY | FORWARD | \neg ARCHIVE

The recipient destination(s) can be specified in one of two ways. The user can specify the recipients individually using explicit addressing (eg. USER@NW_1) or by user alias (eg. USR1), or the user can specify a group distribution list. The distribution list is given a name such as USRGRP1_DL, and corresponds to a list of individual destinations. Distribution list are appropriate in an office environment whereby a distribution list can be created and associated with a single department within the office. Alias and distribution lists resolution can be achieved by storing the corresponding values in a named database and performing a database lookup to retrieve the data. The *DestListType* field in the RCO Data Space is used to distinguish which destination specification was used.

3.12 IMO Routing

Once the IMO has been constructed or retrieved, the SMM is notified and a message transfer object is instantiated (MTO). The MTO will then be responsible for delivery of the IMO to the specified destination. The MTO inherits the identity of the RCO OM and thus has access to the internal structures of the RCO. The initial destination is determined via the DestList field and the communication subsystem is called upon to transfer the IMO to the destination.

If the transfer is not successful, the MTO will utilize the event table to determine the appropriate action for the event that caused the failure. The valid events and actions were listed in Tables 3.4 and 3.5 respectively in Section 3.3.2. If the user desires, the message can be put into a send queue where the MTO can retry the transmission at a later time. In general, the default action for an unsuccessful transmission from a remote destination would be for the message to be sent back to the originator.

When an IMO arrives at a destination, the SMM will again instantiate a MTO to process that inherits the identity of the RCO. The IMO is then placed into a receive queue. At this point the MTO will post the IMO and sender identifiers to the local message center to indicate the new arrival to other IMOs. Each entry in the message center can have an associated action. The MTO must periodically scan the message center entries to see if a message has been posted for it. If one is found, the action specifies how the IMO should proceed. The actions specified here are the same as those list in Table 3.5. This mechanism allows IMOs to communicate. For example, if a user has sent an IMO but wishes to have it terminated, another IMO can be sent to the same set of destinations in an attempt to locate the first IMO. It will simply post messages at each destination indicating that the IMO should terminate itself immediately when it arrives at one of the destinations.

While in the receive queue, the MTO will monitor its environment. The ITV value allows the MTO to take action if the user has not responded to the message within a specified amount of time. If the ITV value has been specified, the MTO will start timing its wait time in the queue. If the ITV time limit expires, the MTO will execute the event table action associated with the timeout event. There must be a ITV action specified to handle the timeout event if an ITV is to be used. The user may require the message to

instantiate a copy of itself with the copy staying at the recipient and the original moving on to the next destination, or the MTO may simply modify its destination tables to skip the current recipient and come back at a later time. In the former case, it is not necessary for the copy message to further route itself so its destination list will be cleared.

3.13 IMO Receipt

When a recipient wishes to receive a given message, the SMM spawns a resource object to allow the user to access the message. The user is presented a list of active messages currently in the receive queue. The recipient may select any message at any time. Once a message has been selected, the resource object notifies the MTO for that message and instantiate a user agent object (UAO) to interface the IMO with the recipient. The MTO transfers control to the UAO. The UAO then instantiates a script processor to execute the message script (i.e. the message mission).

Once the script processor has been instantiated, it can begin interpreting and then executing the message script. User-defined methods are accessed via the method entry point table in the MCO Data Space. The appropriate data is collected and stored in the response data store for the current destination.

When script processing is terminated, control is returned to the UAO. The UAO will guarantee that the only operations allowed on the message by the recipient are those operations specified by the originator using the AccessAttr field. For instance, the user will only be allowed to store a local copy of the IMO if the message has archive capabilities. When the mission of the message has been completed, the UAO will return control to the MTO. The MTO will then proceed to transfer the message to the next destination . When the destination list is exhausted the message will execute the termination event action associated with the NoDest event. Typically, this action is set to RETURN, which allows the IMO to return to the original sender of the message.

3.14 System Configuration File

When IMS initializes, system parameters, as well as, IMO configuration parameters are loaded with system defaults. These typically include such parameters as the maximum

number of queued pending messages to be read, interrupt timeout value, etc. These system default values can be overridden through the use of the *system configuration file*. The configuration file is composed of three sections:

System Declarations: these are parameters that define system wide features unrelated to message objects.

Route Control Declarations: these are parameters that are used to configure the RCO.

Message Control Declarations: these are parameters that are used to configure the MCO.

Figure 3-5 shows the format of the system configuration file. At system initialization, the configuration file will be parsed and error checked for invalid configuration options (i.e values out of range, invalid parameter_id, etc). If no errors are found in the file, the configuration values will be populated to the associated data structures for each specific system category specified.

3.15 Summary

In this chapter we have defined the architecture and the operational semantics of IMS. The fundamental entities in the system are the System Mail Manager (SMM) and the Intelligent Message Object (IMO). The system provides the functionality for allowing a user to assign a task to a message object using the IMSL language, as well as, specifying certain routing characteristics in order to increase the efficiency with which the message carries out its task. In the next chapter we provide a formal specification of the major functions of IMS.

```
[System Declarations]
<configuration values>
:

[Route Control Declarations]
<configuration values>
:

[Message Control Declarations]
<configuration values>
:

where: <configuration values> is of the form:
        <parameter_id> = <parameter_value>

and:    <parameter_id> is the tokenized label of
the system variable that is to be configured.
        <parameter_value> is the value that will
populate the system variable.
```

Figure 3-5: IMS System Configuration File.

Chapter 4

Formal Definition

In this chapter we continue our description of **IMS** from Chapter 3 by defining the functional architecture of the system. The functions which are defined in this chapter represent the basic set with which a user can utilize the system. Each function is defined in part by an informal description of the internal processing followed by a more formal definition using the *Vienna Development Method* (VDM) (Jones 1989). An overview of the VDM formal specification method is given in Section 4.1.

4.1 VDM

VDM is a powerful software development tool which allows systems to be formally specified in terms of both data structures and the operations on those structures. VDM is a model-based approach whose foundation is based on set-theory. Each specification of a system is given in the form of a model which represents the input, outputs, internal state, and functions that operate on the data within the system. VDM emphasizes both *functional specification* (i.e. what the system does) and *design verification* (i.e. will it work). For the purposes of this thesis we will address the functional aspects of **IMS**.

Nomenclature within VDM is straightforward. Variables, type definitions, and functions are specified by giving the name of the object followed by a semicolon (;) and its type. Type names start with the first letter in upper case and the following letters in lower case. Variables and functions are specified in all upper case. Constant names are specified in all upper case italics. Constant values are fixed and do not change.

4.1.1 Data Types

VDM supports both scalar and structured data types. Scalar types can either be built-in or user defined types. Built-in data types can be of the form shown below:

- Character: indicating any alphabetic character.
- Int: indicating the set of integers.
- Nat: indicating the set of natural numbers.
- Nat0: indicating the set of non-negative numbers.
- Real: indicating the set of real numbers.
- Bool: indicating *TRUE*, *FALSE*

New types can be created as necessary. For example, *Alphanum* can be defined to represent the set of alpha-numeric character sequences. From this definition we can then define a variable to be of this type. VDM also supports a special data type called *NIL* which represents undefined values. It is unique in that the type represents both the name and value of the type.

VDM also supports the structured types listed below which are described in the following sections:

- Set Types
- List Types
- Record Types
- Mapping Types

4.1.2 Set Types

Set types are useful for grouping similar entities such that operations can be applied on the set as a whole. The set of operations that can be applied to sets are listed in Table 4.1. Let $R = \{U_1, U_2, U_3\}$ and $S = \{U_2, U_5\}$. Then the following relationships exist:

- $R \cup S = \{U_1, U_2, U_3, U_5\}$

Set Operations	
Operator	Function
\in	Set Member
\cup	Union
\cap	Intersection
<i>subset</i>	Union
$-$	Difference
card	Cardinality

Table 4.1: VDM Set Operations.

- $R \cap S = \{U_2\}$
- $R - S = \{U_1, U_3\}$
- $S \subset R = \text{FALSE}$
- $U_2 \in R = \text{TRUE}$
- $\text{card}(R) = 3$

If a set name ends with the extension “-set” then that set represents the powerset of the named base set. For example, *S-set* represents the powerset $\{\{\}, \{U_1\}, \{U_2\}, \{U_1, U_2\}\}$.

4.1.3 List Types

List types are useful for defining ordered collections of values. Similar to set types, if the extension “-list” is appended to the list name, then a powerlist is created which represents all possible lists that can be created from the base list. Table 4.2 defines the set of operators that can be applied to lists. Again let $R = \{U_1, U_2, U_3\}$ and $S = \{U_2, U_5\}$. Then the following relationships exist:

- $\text{hd } R = \{U_1\}$
- $\text{tl } R = \{U_2, U_3\}$
- $R \parallel S = \{U_1, U_2, U_3, U_2, U_5\}$
- $R(3) = \{U_3\}$

List Operations	
Operator	Function
hd	head of list
tl	tail
 	Concatenation
l(num)	List Member
elems	List Elements

Table 4.2: VDM List Operations.

- **len** R = 3
- **elems** S = {U₂, U₅}

4.1.4 Record Types

VDM allows the definition of record structures which can be used to represent the internal entities of the system being defined. Each record type is given a name with the specification of the fields within the record following the name as in the example given below:

```
Employee::      NAME:      String
                ID:        Int
                POSITION:   String
                SSN:       Int
```

In the above example *Employee* is the name of the record type with *Name*, *ID*, *Position*, and *SSN* as fields within the record. VDM allows a special function to be called that will create and initialize an instance of a particular record type. These functions are prefixed with *mk* as in “mk-Employee(John Doe, 100, Programmer, 123-456-7890)”. If *EMP* is a variable of type *Employee*, then the fields within *EMP* can be accessed as in “NAME(EMP) = John Doe” or “SSN(EMP) = 123-456-7890”.

Mapping Operations	
Operator	Function
dom	Domain
rng	Range
m(x)	Element
/	Restrict to
\	Restrict by
†	Overwrite

Table 4.3: VDM Mapping Operations.

4.1.5 Mappings

In VDM, a special function exists that maps the elements of one set onto the elements of a second set. The first set is referred to as the *domain* and the second set is referred to as the *range*. A mapping is denoted by using the “ \rightarrow ” such as in $R \rightarrow S$. Table 4.3 defines the set of mapping operators. Let $R = \{U_1 \rightarrow D_1, U_2 \rightarrow D_2\}$. Then the following relationships exist:

- **dom** $R = \{U_1, U_2\}$
- **rng** $R = \{D_1, D_2\}$
- $R(U_1) = D_1$
- $R / \{U_2\} = \{U_2 \rightarrow D_2\}$
- $R \setminus \{U_2\} = \{U_1 \rightarrow D_1\}$
- $R \dagger \{U_2 \rightarrow Z_1\} = \{U_1 \rightarrow D_1, U_2 \rightarrow Z_1\}$

The restrict to operator (/) creates a mapping that restricts the domain to the specified set while the restrict by operator (\) creates a mapping where the elements of the specified set are not in the domain of the resulting set.

4.1.6 Operations and Functions

VDM utilizes predicate logic to construct predicates which represent operations and functions. Table 4.4 defines the set of predicate operations within VDM.

Predicate Operations	
Operator	Function
\neg	Negation
\wedge	Conjunction (AND)
\vee	Disjunction (OR)
\forall	Universal Quantification.
\exists	Existential Quantification.

Table 4.4: VDM Predicate Operations.

VDM supports three additional constructs. The first is the *let ... in* construct allows a mechanism for text substitution when specifying a lengthy predicate. For example, *let* $W = X(Y(Z))$ *in* $A(W)$ is a simplification for $A(X(Y(Z)))$ which can become confusing to read in complicated predicates. The *while ... do* construct is used for repeated predicates and the *if ... then ... else* construct is used for conditional predicates. Each VDM specification is composed of the four parts listed below:

1. **Operation name and parameters.**

Input and output parameters may be specified. For example, in the specification “ $FOO(X : \text{Int}) Y : \text{Real}$ ”, FOO represents the name of the operation, with input variable “ X ” and output variable “ Y ”.

2. **ext:.**

This section defines the set of external variables and their types which the operation must have access to. Read-only variables are specified with the keyword **rd** while read-write variables are specified with **wr**.

3. **pre:**

This section defines the set of preconditions that must hold true for the operation to be applied. These conditions are applied to the input and external variables in the form of predicate relationships.

4. **post:**

This section defines how the values of the output and external variables have been changed as a result of the operation.

Having provided a brief introduction to the VDM formal method, we will now begin our formal specification of *IMS*. We begin our specification by first defining the sets, data

elements, and data mappings that represent the system. We then proceed with the formal specification of the major functions within IMS

4.2 IMS Specification

As previously indicated, **IMS** functions within a networked environment of end-user sites. Each end-user site represents both the recipient destination and the user-id for that location. Let

$$U_{IMS} = \{U_1, U_2, \dots, U_n\}$$

$$D_{IMS} = \{D_1, D_2, \dots, D_n\}$$

correspond to the set of all valid users and recipient destinations of **IMS**. Then

$$S_{U \rightarrow D} = \{U_1 \rightarrow D_1, U_2 \rightarrow D_2 \dots U_n \rightarrow D_n\}$$

represents the set of one-to-one logical mappings of user-ids to destination-ids. Given S , then S -set is the powerset of S which represents all possible combinations of logical routing paths within **IMS**.

The following defines the fundamental data types of **IMS**.

1. Evt: {ITVTimeout, AccessInit, NoDest, RecvDeny, NoResource, DestUnavl, DestNotExist, NetwUnavl, NetwErr}
2. Act: {NOTIFY, RETURN, TERMINATE, DELETE, FORWARD}
3. Dlaccess: {NONE, DELETE, ADD}
4. Dltype: {USER, GROUP, ALIAS}
5. Priority: {NORMAL, URGENT}
6. Accessattr: {ARCHIVE, MODIFY, VIEW, PRINT, PURGE}
7. Alphanum: sequence of numeric and alphabetic characters

Having defined the previous data types, we declare the set of **IMO** record structures as shown in Figure 4.2. Here we specify **VDM** record types for the Global Data Object, Message Control Object, Route Control Object, and the Intelligent Message Object.

We now define the following set of mapping types specific to the system. The map type names will be subscripted with “M” to distinguish them as mappings.

Gdo::	IMoid	Int
	CREATETSTMP:	Alphanum
	ORGid:	String
	ORGADDR:	Alphanum
	CURRDEST:	Alphanum
	CURRDESTADDR:	Alphanum
	NEXTDEST:	Alphanum
	MSGBLKPTR:	Int
Rco::	PRIORITY:	Priority
	DESTLIST:	S-set
	DESTLISTTYPE:	Dltype
	DLVCONFIRM:	Bool
	ACCESSCONFIRM:	Bool
	RETTOORG:	Bool
	SPAWN:	Bool
	ITV:	Int
	ARRIVETSTMP:	Alphanum
	DLACCESS:	Dlaccess
Mco::	SUBJECT:	String
	ACCESSATTR:	Accessattr
	ACCESSTSTMP:	Alphanum
	MTPTR:	Int
	RSPSTORE:	Int
	SCRIPTPTR:	Int
Imo::	GDO:	Gdo
	MCO:	Mco
	RCO:	Rco

Figure 4-1: IMO Record Type Specification

1. $EVT_M: Evt \rightarrow Act$
Mapping of an event to an action such as specified in the Route Event Table. For example, [NoDest \rightarrow RETURN].
2. $IMSUSR_M: UID \rightarrow DEST$
Mapping of a user-id to a destination. For example, [100 \rightarrow user1@netw1].
3. $SQ_M, RQ_M, MF_M, IMO_M: IMOID \rightarrow IMO$
Each is a mapping of an IMO identifier to an IMO. RQ represents a receive queue, SQ represents a send queue, and MF represents a message folder.
4. $E_M: Evt \rightarrow Int$
Mapping of an event to an integer number. For example [NoDest \rightarrow 0].
5. $A_M: Act \rightarrow Int$
Mapping of an action to an integer number. For example [NOTIFY \rightarrow 0].

In the following sections we will formally define the major functions within **IMS**. These functions are classified as either *primary* or *auxiliary*. The distinction is made for clarification purposes. Primary function are those functions that are executed through direct intervention on the part of the user. For example, by selecting the Send option from the IMO Transfer Environment, the SEND-IMO function will be invoked. An auxiliary function is one that is executed indirectly as a result of the internal processing of the system. Except where indicated, these functions will be classified as primary.

4.2.1 INIT-IMS

The *INIT-IMS* function is invoked upon the initialization of **IMS**. All entities within the system are initialized and the system configuration file is read and the values specified are loaded into the appropriate locations. Figure 4-2 shows the VDM specification for this function. The only parameter specifies a user identifier which is used as a validation mechanism which prevents non-**IMS** users access to the system. The precondition specifies that the user must be a valid user of **IMS**. The postcondition indicates that upon initialization, the Send and Receive queues are cleared of any IMOs and the state of any IMO is also cleared. The state of the IMO Message Folder is cleared only during the very first time the system is initialized at a given site.

```

/* Function that initiates the IMO processing mechanism. */

IMS-INIT( U : Uid )

ext:   rd IMSUSR:   IMSUSRm
       wr SQ:      SQm
       wr RQ:      RQm
       wr MF:      MFm

pre:   U ∈ dom IMSUSR

post:  IMO' = □
       SQ'  = □
       RQ'  = □
       MF'  = □

```

Figure 4-2: VDM Schema for IMS-INIT().

4.2.2 CREAT-MCO

The *CREAT-MCO* function is invoked when the user wishes to create and initialize an instance of a Message Control Object. The VDM specification for this function is shown in Figure 4-3. This function utilizes several auxiliary functions which are used for obtaining the user state specifications for the Data Space, Method Table, and Message Script. The function accepts two parameters: the IMO identifier for which the MCO is being created and the identifier for the MCO instance. The precondition simply checks that the IMO identifier is valid. The postcondition indicates that the current instance space of the MCO is overwritten with the user state specification inputs for each region of the MCO. The identifier of the MCO instance is set and the function returns the MCO instance.

4.2.3 CREAT-RCO

The *CREAT-RCO* function is invoked when the user wishes to create and initialize an instance of a Route Control Object. The VDM specification for this function is shown in Figure 4-4. This function utilizes several auxiliary functions which are used for obtaining

```

/* Auxilliary Function to create and initialize an MCO. */

CREAT-MCO( I : IMOid, M : MCOid ) MCO: MCO-type

ext:   wr IMO:   IMOm

pre:   I ∈ dom N

post:  MCO'(IMO(I) = MCO(IMO(I) †
      [M→ mk-mco-ds( MCOds(MCO(IMO(I))) ) ∪
      mk-mco-mtbl( MTbl(MCO(IMO(I))) ) ∪
      mk-mco-script( MScript(MCO(IMO(I))) ) ])

```

The parameters to **mk-mco-ds**, **mk-mco-mtbl**, and **mk-mco-script** correspond to the fields of the MCO Data Space, Method Table, and Script respectively. These functions initialize each structure as specified by the user.

Figure 4-3: VDM Schema for CREAT-MCO().

the user state specifications for the Data Space, Route Event Table. The function accepts two parameters: the IMO identifier for which the RCO is being created and the identifier for the RCO instance. The precondition simply checks that the IMO identifier is valid. The postcondition indicates that the current instance space of the RCO is overwritten with the user state specification inputs for each region of the RCO. The identifier of the RCO instance is set and the function returns the RCO instance.

4.2.4 CREAT-IMO

The *CREAT-IMO* function is invoked indirectly whenever a user enters the IMO Creation Environment. An IMO identifier is created upon entry and is associated with the IMO instance to be created. Figure 4-5 shows the VDM specification for this function. Parameters to the function include the IMO, MCO, and RCO identifiers. The precondition validates the integrity of the IMO identifier and checks to make sure that a unique instance of the IMO has been created. The postcondition simply maps the state specifications for the MCO

```

/* Auxilliary Function to create and initialize an RCO. */

CREAT-RCO( I : IMOid, R : RCOid ) RCO: RCO-type

ext:   wr IMO:   IMOm

pre:   I ∈ dom N

post:  RCO'(IMO(I) = RCO(IMO(I) †
        [R → mk-rco-ds( RCOds(RCO(IMO(I))) ) ∪
          mk-rco-ret( RET(RCO(IMO(I))) ) ]

```

The parameters to **mk-rco-ds** and **mk-rco-ret** correspond to the fields of the RCO Data Space and RCO Route Table respectively. These functions initialize each structure as specified by the user.

Figure 4-4: VDM Schema for CREAT-RCO().

and RCO into the IMO.

4.2.5 ARCHIVE-IMO

The *ARCHIVE-IMO* function is invoked whenever the user wishes to archive (i.e save) an IMO that has arrived from another destination. However, this function can only be invoked if the IMO has the ARCHIVE attribute set in the MCO AccessAttr field. This attribute can only be set by the creator of the message. The VDM specification for this function is shown in Figure 4-6. The function accepts the IMO identifier as the only parameter. The precondition checks to ensure that this identifier is a valid IMO id and that the ARCHIVE attribute has been set for the IMO to be saved. IMOs are archived in the users IMO Message Folder that exists at the current site. The postcondition specifies the addition of the new IMO to this folder.

```

/* Auxilliary Function to create and initialize an IMO. */

CREAT-IMO( I : IMoid, M : MCoid, R : RCOid )

ext:   wr IMO:    IMOm
       rd SQ:    SQm

pre:   I ∈ dom RQ
       ∧ let sq = dom SQ in
         I ∈ ∪I∈sq (IMO(I) ∪ SQ(I))

post:  IMO(I)' = IMO(I) † [I → mk-imo(I,M,R)]

```

Figure 4-5: VDM Schema for CREAT-IMO().

```

/* Function to archive an IMO to the message folder. */

ARCHIVE-IMO( I : IMoid )

ext:   rd IMO:    IMOm
       wr MF:    MFm

pre:   let ACCESS = AccessAttr(MCods(IMO(I))) in
       I ∈ dom MF
       ∧ allow(ACCESS,ARCHIVE) = TRUE

post:  MF' = MF ∪ {I}

allow: → Bool allow(S,E) is a function that determines if E ∈ S and returns
TRUE or FALSE based on the outcome.

```

Figure 4-6: VDM Schema for ARCHIVE-RCO().

```

/* Function to purge an IMO to the message folder. */

PURGE-IMO( I : IMoid )

ext:   rd IMO:   IMOm
       wr MF:   MFm

pre:   let ACCESS = AccessAttr(MCOds(IMO(I))) in
       I ∈ dom MF
       ∧ allow(ACCESS,PURGE) = TRUE

post:  MF' = MF † [MF \ {I}]

allow: → Bool allow(S,E) is a function that determines if E ∈ S and returns
TRUE or FALSE based on the outcome.

```

Figure 4-7: VDM Schema for PURGE-IMO().

4.2.6 PURGE-IMO

The *PURGE-IMO* function is invoked whenever the user wishes to purge (i.e delete) an IMO that has been archived in the users IMO Message Folder. Like the previous function, this function can only be invoked if the IMO has the PURGE attribute set in the MCO AccessAttr field. This attribute can only be set by the creator of the message. The VDM specification for this function is shown in Figure 4-7. The function accepts the IMO identifier as the only parameter. The precondition checks to ensure that this identifier is a valid IMO id and that the PURGE attribute has been set for the IMO to be deleted. The postcondition specifies that the message folder is updated by the removal of the specified IMO from the folder.

4.2.7 SEND-IMO

The *SEND-IMO* function is invoked whenever the user wishes to send an IMO to a recipient destination. This function directly activates the internal processing of the selected IMO. An IMO can be sent by direct intervention of the user, or the expiration of a temporal

specification, or as the result of an external event which requires the IMO to move to another destination. The VDM specification for this function is shown in Figure 4-8.

The function takes the IMO id to be sent as the only parameter. The precondition makes several assertions before an IMO can be sent. The send queue must contain at least one IMO and a validation is indicated to ensure that the specified IMO exists on the queue. The precondition also specifies the conditions for an IMO to be sent due to an indirect activity. The postcondition checks for the specification an empty destination. If this is the case, the event “NoDest” is processed. Normally, this mechanism is utilized to terminate IMO processing with the action for this event being “RETURN” to creator.

If a destination is indicated, the internal tables in the IMO are updated. This includes updating the destination list in the RCO by removing the current destination from the list and setting the next destination in the list appropriately. The CurrDest and NextDest fields in the GDO must be updated to contain the appropriate values. The auxiliary *xfer* function is then invoked which transfers the IMO to the specified destination and returns the result of the transfer.

4.2.8 RECV-IMO

The *RECV-IMO* function is invoked whenever the user wishes to receive an IMO that is currently pending in the users receive queue. When an IMO is selected for receipt, it is possible that the sender of that IMO has set the AccessInit event to be triggered. In this case, the IMO must initiate a reply back to its originator indicating the time that the recipient acted upon the IMO. It may also be the case that an IMO in the queue is an IMO that has returned from its mission and is awaiting notification to its creator. The final possibility is that an IMO has exceeded its ITV interval while awaiting a response from the recipient. It must then process the “ITVTimeout” event according to its route table entry for the event.

The function accepts an IMO id as the only parameter. The precondition checks that the receive queue has at least one entry in it and that the IMO id is a member of that queue. It then asserts the conditions listed in the previous paragraph as possible candidate IMOs to receive. It must be the case that either the IMO id is the one selected by the user, an ITV has expired for an IMO and must be processed, or an IMO has returned from its

```

/* Function to transfer an IMO to another destination. */

SEND-IMO( I : IMOid )

ext:   rd IMO:   IMOm
      wr SQ:    SQm

pre:   SQ ≠ {}
      ∧ ∀ I' ∈ SQ
        ∃ J: IMOid(GDO(IMO(J))) = I
          ∨ SendTime(GDO(IMO(J))) ≥ currtime()

post:  let RDS = RCOds(IMO(I))
      and GDO = GDO(IMO(I))
      and DEST = NextDest(GDO) in
      if DEST = NIL then
        evt-process(I,NoDest)
      else let DL = DestList(RDS)
          and CD = CurrDest(GDO)
          and ND = NextDest(GDO) in
          DL' = DL † tl DL
          ∧ CD' = CD † DEST
          ∧ ND' = ND † hd DL
          ∧ Evt' = Evt † xfer(I,DEST)
          ∧ evt-process(I,Evt)

xfer: → Event xfer(I,D) is a function that sends IMO→I to Destination→D
and returns the outcome event of the transfer.

```

Figure 4-8: VDM Schema for SEND-IMO().

```

/* Function to receive an IMO off the receive queue. */

RECV-IMO( I : IMOid, CD : DESTid )

ext:  rd IMO:    IMOm
      wr RQ:    RQm

pre:   $\forall I \in \text{dom RQ}$ 
      /* An IMO selected by the recipient manually */
       $(\exists J: \text{ITV}(\text{RCods}(\text{IMO}(J))) > 0$ 
         $\wedge \text{IMOid}(\text{GDO}(\text{IMO}(J))) = I$ 
         $\wedge \text{evt-process}(I, \text{AccessInit}))$ 

      /* An IMO in which the ITV expires */
       $\vee (\exists J: \text{ITV}(\text{RCods}(\text{IMO}(J))) \leq 0$ 
         $\wedge \text{evt-process}(J, \text{ITVTimeout}))$ 

      /* An IMO that has returned back to Creator */
       $\vee (\exists J: \text{ORGAddr}(\text{GDO}(\text{IMO}(J))) = \text{CD}$ 
         $\wedge \text{evt-process}(J, \text{NOTIFY}))$ 

post:   $\text{RQ}' = \text{RQ} \uparrow [\text{RQ} \setminus \{\text{RQ}(J)\}]$ 

```

Figure 4-9: VDM Schema for RECV-IMO().

mission and is ready to disseminate the information collected from its mission. The VDM specification for this function is shown in Figure 4-9. The postcondition asserts that the selected IMO is removed from the receive queue.

4.2.9 ADD-RDEST

The *ADD-RDEST* function is invoked when the user wishes to add a recipient destination to the selected IMO. Destinations can be added manually through the **Add Dest** option in the IMO Transfer Environment or through IMO script processing where the recipient is prompted for additional destinations and the destination list of the RCO is automatically updated. In either case, the IMO to be modified must have the ADD attribute set in the DLAccessAttr field to allow destinations to be added. The IMO to be updated must exist in either the users' send or receive queue.

The function accepts two parameters: the first is the user-id of the recipient to be added and the second is the IMO id in which to add the recipient to. The precondition asserts several conditions. First of all, the selected IMO must exist in either the send or receive queue. Further, the destination to be added must be valid within IMS. Finally, that the destination to be added must not already exist in the destination list and the user must have the appropriate permission to add the destination. The postcondition specifies that the destination is mapped from the user-id to be added and that the destination list of the IMO is properly updated by adding the new destination to the end of the destination list. The VDM specification for this function is shown in Figure 4-10.

4.2.10 DEL-RDEST

The *DEL-RDEST* function is invoked when the user wishes to delete a recipient destination from the selected IMO. Destinations can be deleted manually through the **Del Dest** option in the IMO Transfer Environment or through IMO script processing where the recipient is prompted for as to which destinations should be removed and the destination list of the RCO. In either case, the IMO to be modified must have the DELETE attribute set in the DLAccessAttr field to allow destinations to be deleted. The IMO to be updated must exist in either the users' send or receive queue.

The function accepts two parameters: the first is the destination to be deleted and the

```

/* Auxilliary Function that adds a recipient destination */
/* to the RCO destination list.                               */
ADD-RDEST( U : Uid, I : IMOid )

```

```

ext:   rd IMSUSR:    IMSUSRm
         wr IMO:      IMOm
         rd RQ:       RQm
         rd SQ:       SQm

```

```

pre:   U ∈ bf dom IMSUSR
         ∧ (I ∈ bf dom RQ ∨ I ∈ dom SQ)
         ∧ let RDS = RCO_ds(IMO(I))
           and DL = DestList(RDS)
           and ACCESS = DLAccessAttr(RDS)
           and DEST = m(IMSUSR(U)) in
           DEST ∈ elems DL
           ∧ allow(ACCESS, ADD) = TRUE

```

```

post:  let DL = DestList(RCO_ds(IMO(I))) in
         DL' = DLK † [DL || m(IMSUSR(UID))]

```

allow: → Bool **allow(S,E)** is a function that determines if $E \in S$ and returns TRUE or FALSE based on the outcome.

Figure 4-10: VDM Schema for ADD-RDEST().

second is the IMO id in which to delete the destination from. The precondition asserts several conditions. The first is that the selected IMO must exist in either the send or receive queue. The second condition asserts that the destination to be deleted must exist in the destination list of the IMO and that the user must have the appropriate permission to delete the destination. The postcondition specifies that the destination is removed from the destination list and updates the state of the GDO object to reflect the deletion. The VDM specification for this function is shown in Figure 4-11.

4.2.11 PROCESS-SCRIPT

The *PROCESS-SCRIPT* function is an auxiliary function that is automatically invoked when control of an IMO is transferred to the user agent object representing the IMO. The message script of the MCO is interpreted and executed at each recipient site. Information that may be collected at a given destination is stored in the response store for that destination in the GDO. The function accepts an IMO id as its only parameter. The precondition checks that the id is valid and that there is a message script associated with the IMO. The postcondition asserts that the script will be executed until either the recipient terminates the script manually or there are no more lines in the script to carry out. If any user input was collected, the information is stored in the appropriate location in the response store of the GDO. The VDM specification for this function is shown in Figure 4-12.

4.2.12 EVT-PROCESS

The *EVT-PROCESS* function is utilized for accepting and processing events that arise as the IMO moves between destinations. Figure 4-13 shows the VDM specification for this function. The function accepts an IMO id and an event id as parameters. If the event exists in the route table for the IMO then the creator has indicated that the event should be handled according to the action that has been mapped to that event. If the event has not been specified then the default event/action 2-tuple will be utilized. This 2-tuple exists for any IMO with the default action specifying that the IMO return to the creator.

The precondition specifies that the event must be a valid event and that the IMO exists within the users receive queue. The postcondition asserts that for all of the events specified within the route table, if the input event exists in the table then the corresponding action

```
/* Auxilliary Function that deletes a recipient */
/* destination from the RCO destination list.  */
```

```
DEL-RDEST( D : DESTid, I : IMoid )
```

```
ext:  wr IMO:    IMOm
      rd RQ:    RQm
      rd SQ:    SQm
```

```
pre:  (I ∈ bf dom RQ ∨ I ∈ dom SQ)
      ∧ let RDS = RCO_ds(IMO(I))
        and DL = DestList(RDS)
        and ACCESS = DLAccessAttr(RDS)
        elems DL > 0
        ∧ D ∈ elems DL
        ∧ allow(ACCESS, DELET) = TRUE
```

```
post: let DL = DestList(RCO_ds(IMO(I)))
      and CD = CurrDest(GDO(IMO(I)))
      and ND = CurrDest(GDO(IMO(I))) in
      DL' = DL † [DL - D]
      ∧ if D = CD then
        CD' = CD † hd DL
      ∧ if CD' ≠ NIL bf then
        ND' = ND † hd tl DL
```

```
allow: → Bool allow(S,E) is a function that determines if E ∈ S and returns
TRUE or FALSE based on the outcome.
```

Figure 4-11: VDM Schema for DEL-RDEST().

/ Function to transfer an IMO to another destination. */*

PROCESS-SCRIPT(I : IMOid)

ext: wr IMO: IMO_m

*pre: I ∈ dom IMO
 ∧ MScript(MCO(IMO(I))) ≠ NIL*

*post: let GDO = GDO(IMO(I)
 and DEST = CurrDest(GDO)
 and RS = RspStore(GDO)(DEST)
 and SC = MScript(MCO(IMO(I))) in
 while ¬terminate() do
 L ← inputLine(SC)
 ∧ L' ← interpret(L)
 ∧ R ← execute(L')
 ∧ if R = TRUE then
 RS' = RS † [RS || (DEST || R)]*

terminate: → Bool **terminate** is a function that returns TRUE when the script is finished executing either by manual or automatic intervention.

inputLine: → IMSL comand line

inputLine(SC): is a function that reads and returns a line of input from an IMSL script.

intepret: → String

interpret(L) is a function that interprets in IMSL input line and outputs a alternate line to be executed.

execute: → Bool

execute(L) is a function executes a IMSL input line formatted by

Figure 4-12: VDM Schema for PROCESS-SCRIPT.

```

/* Function to process an IMO event/action. */

EVT-PROCESS( I : IMOid, E : Event, D : DESTid )

ext:  rd IMO:    IMOm
      rd EVT:    Em
      wr RQ:     RQm
      wr SQ:     SQm

pre:  E ∈ dom EVT
      ∧ I ∈ dom RQ

post: let RET = RET(IMO(I)) and GDO = GDO(IMO(I))
      and RDS = RDOds(IMO(I)) in
      /* Process default event/action */
      if E ∉ dom RET ⊃ ACTION ← Action(RET(α))
      ∧ if E ∈ dom RET ⊃ ACTION ← m(RET(E))
      ∧ let CD = CurrDest(GDO)
        and ND = NextDest(GDO) and DL = DestList(RDS) in
        /* Handle NOTIFY Action */
        ∨ ( if ACTION → NOTIFY then
            if ORGADDR(GDO) = D then
                DL' = DL † ORGADDR(GDO)
                ∧ SQ' = SQ † [SQ / {I}]
                ∧ SEND-IMO(I) )
        /* Handle TERMINATE Action */
        ∨ ( if ACTION → TERMINATE then
            RQ' = RQ † [RQ \ {I}] )
        /* Handle RETURN Action */
        ∨ ( if ACTION → RETURN then
            DL' = DL † ORGADDR(GDO)
            ∧ SEND-IMO(I) )
        /* Handle DELETE Action */
        ∨ ( if ACTION → DELETE then
            DL' = DL † [DL - D]
            ∧ CD' = CD † hd DL'
            ∧ if CD' ≠ NIL then
                ND' = ND † hd tl DL )
        /* Handle FORWARD Action */
        ∨ ( if ACTION → FORWARD then
            DL' = DL † [tl DL || hd DL]
            ∧ SQ' = SQ † [SQ / {I}]
            ∧ SEND-IMO(I) )

```

Figure 4-13: VDM Schema for SEND-IMO().

```

/* Function that initiates the IMO processing mechanism. */

IMO-INITIATE( I : IMOid )

ext:  rd IMO:  IMOm

pre:  successful arrival at NextDest.

post:  process-script(I)
       ∧ let DL = DestList(RCOds(IMO(I)))
         and Dest = hd DL
         and CD = CurrDest(GDO(IMO(I)))
         and ND = NextDest(GDO(IMO(I))) in
       if Dest ≠ NIL then
         DL' = DL † tl DL
         ∧ CD' = CD † hd DL
         ∧ if CD' ≠ NIL then
           ND' = ND † hd tl DL
         ∧ R' = R † xfer(I, Dest)
         ∧ R'' = R' † evt-process(R')

```

Figure 4-14: VDM Schema for IMO-INITIATE().

will be carried out. If the action is to notify or return to the creator, the IMO is placed on the send queue for delivery. If the action is to terminate, the IMO is removed from the receive queue. If the action is to delete the current destination, then the destination is removed from the destination list and the state of the GDO is updated. If the action is to forward to the next destination, then the current destination is placed at the end of the destination list and the IMO is forwarded to the next destination.

4.2.13 IMO-INITIATE

The *IMO-INITIATE* function is an auxiliary function that represents the logical processing of the IMO. After arrival at a destination the execution of the message script takes places and information is collected from the recipient. The IMO then attempts to move to the next destination with the resulting event processed according to the route table entries. Figure 4-14 shows the VDM specification for this function.

The function accepts an IMO id as its only parameter which is validated in the precondition. The postcondition asserts that the message script is executed and for each destination in the destination list, the IMO is transferred to that location and the internal state of the GDO is updated to reflect the transfer.

4.3 Formal Definition of IMSL

In this section we formally define the Intelligent Message Script Language using the Backus-Naur Format. Section 4.3.1 summarizes the notation that will be used to represent the IMSL grammar.

4.3.1 BNF Overview

The notation used to define the IMSL grammar in Section 4.3.2 is based on an extended form of Backus-Naur Format (BNF). (Lewis & Papadimitriou 1981). A short summary of the notation is provided below.

1. A grammar consists of a series of rules which define the syntax of the language to be represented.
2. The left hand side of the rule is separated from the right hand side by a “ $::\Rightarrow$ ”.
3. Alternatives within a rule are separated by a “ $||$ ”.
4. Member of the same alternative are separated by a “ $,$ ”.
5. An empty alternative is represented by “ \emptyset ”.
6. Each rule end with a “ \blacksquare ”.
7. Keywords are written in “bold” type.
8. Literals are enclosed in single quotes (‘’).

The formal definition of IMSL is presented next.

4.3.2 BNF Specification of IMSL

IMSL-Syntax $::\Rightarrow$
IMSL-Statement-Block ■

IMSL-Statement-Block::⇒

IMSL-Stmts |
'[, IMSL-Stmts,]' ■

IMSL-Stmts::⇒

SET-Stmt |
INPUT-Stmt |
DISPLAY-Stmt |
PROMPT-Stmt |
STORE-Stmt |
CASE-Stmt |
IF-Stmt |
GOTO-Stmt |
CALL-Stmt |
FUNCTION-Stmt |
RETURN-Stmt |
EXIT-Stmt ■

SET-Stmt::⇒

SET, Variable, Value ■

INPUT-Stmt::⇒

INPUT, InputVar, InputType ■

DISPLAY-Stmt::⇒

DISPLAY, '[', DispSeq, ']' ■

PROMPT-Stmt::⇒

PROMPT, PromptText, ResponseList ■

IF-Stmt::⇒ ::⇒

IF, Expression, **THEN**, Statement |
IF, Expression, **THEN**, Statement, **ELSE**, Statement ■

CASE-Stmt::⇒

CASE, \$\$, CaseList, **ENDC** ■

STORE-Stmt::⇒

STORE, Variable ■

GOTO-Stmt::⇒

GOTO, Label ■

CALL-Stmt::⇒

CALL, FunctionName, OptParamList ■

FUNCTION-Stmt::⇒
FUNCTION, FunctionName, OptParamList,
FunctionBody,
ENDF ■

RETURN-Stmt::⇒
RETURN ■

EXIT-Stmt::⇒ **EXIT** ■

Statement::⇒ IMSL-Statement-Block ■

CaseList::⇒
Response, :, IMSL-Statement-Block ■

Response::⇒
AlphaChar ■

Expression::⇒
SimpleExpression |
SimpleExpression, RelOp, SimpleExpression ■

SimpleExpression::⇒
Term |
Sign, Term |
SimpleExpression, TermOp, Term ■

Term::⇒ Factor |
Term, FactorOp, Factor ■

Factor::⇒ Variable | Number | NOT, Factor ■

FactorOp::⇒ * | / | AND ■

TermOp::⇒ + | - | OR ■

FunctionBody::⇒ IMSL-Statement-Block ■

FunctionName::⇒ <function identifier> ■

OptParamsList::⇒ '(, OptParams,)' | ∅ ■

OptParams::⇒ Params | ∅ ■

Params::⇒ Param, Params | Param ■

Param::⇒
 <variable identifier> ■

PromptText::⇒ CharSeq ■

ResponseList::⇒ Response, ResponseList | Response ■

Response::⇒ AlphaChar ■

DispSeq::⇒ DispValue, DispSeq | DispValue ■

DispValue::⇒ DispVar | AlphaChar ■

DispVar::⇒ \$VariableType ■

InputVar::⇒ UserVar[!] ■

InputType::⇒
 CHAR |
 LINE |
 PARA |
 NUM |
 DT ■

Label::⇒ :LabelD ■

LabelID::⇒ <label identifier> ■

SystemVarType::⇒ \$SystemVars ■

SystemVars::⇒
 IMOID |
 DESTS |
 DEST |
 NEXTDEST |
 SUBJECT |
 SENDNAME |
 RECNAME |
 DATE |
 TIME |
 NIL |
 \$\$ |
 \$0 ■

UserVarType::⇒ UserVar | @UserVar | ■

UserVar::⇒ <variable identifier> ■
VariableType::⇒ UserVarType | SystemVarType ■
Variable::⇒ VariableType | \$VariableType ■
Constant::⇒ CharSeq | Number | Date ■
Date::⇒ DT ■
Number::⇒ Integer | Real ■
Integer::⇒ Digit | Digit, Integer ■
Real::⇒ Integer.Integer
Digit::⇒ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ■
RelOp::⇒ = | <> | < | > | ≤ | ≥ | in ■
Sign::⇒ + | - | ■
CharSeq::⇒
 AlphaChar, CharSeq | AlphaChar ■
AlphaChar::⇒
 <ascii character> ■

Chapter 5

IMS Applications

In this chapter we provide two examples of how **IMS** can be used to automate routine office tasks. The first example describes how the manager of a team of software developers can utilize an **IMO** to obtain the weekly activity status reports from each member of the team. In the second example, an **IMO** is used to coordinate the scheduling of a meeting so that the time and date of the meeting is acceptable to all attendees. Selected routing scenarios will be utilized to show the typical events that can occur and how they are handled. General functionality of the system is described in the first example while emphasis of the functionality of the **IMSL** language is presented in the second example.

5.1 A Weekly Status Report Example

In this example we provide a description of how **IMS** can be utilized to automate the task of collecting weekly activity status reports. The goal is to show the process that is required to create an **IMO** to perform its task and the general processing that takes place during the execution of the task. We begin with a description of the task which we intend to model.

5.1.1 Task Description

In this example, a large telecommunications software development company will serve as the operating platform for the **IMS**. The company employs approximately 500 software developers and design engineers. Each department in the organization is divided into individual teams with each team managed by a team leader. The team leader is responsible

for providing upper management with monthly status reports pertaining to the individual members of the team, as well as, the team as a whole. This is primarily accomplished through the use of status reports which comprise the following information:

- Employee name
- Employee number
- Current date and time
- Status of tasks assigned to employee
- Description of delays and issues related to assigned tasks
- General Comments.

The current procedure for obtaining team member status information is for the team leader to route a memo to the team members reminding them to submit their status reports by the end of the week so the monthly report can be updated in a timely manner. Each team member then creates a status report document detailing the current week's activities and milestones and submits the report to the inter-office mail department for delivery. Mail is scheduled for pickup and delivery twice a day: once in the morning and once in the afternoon. When the mail has been picked up it is brought to a sorting station where it can be sorted for later delivery. The turnaround time for this process is approximately one day.

There are several problems that arise in such a scenario. First, in the typical office environment, manually routing memos and other mail can be a slow process in and of itself. A team member may complete and submit a status report on time but actual delivery to a destination can easily be delayed due to priorities of other office activities. Second, a team member may misplace the report and fail to submit it on time resulting in a disgruntled team leader. Third, the method increases both the flow of paper through the office and the amount of paper work that must be attended to by the team members.

The solution to the problem is to automate the procedure so that:

1. It requires minimal effort by each member to perform the task.
2. It can be performed in a timely manner.

TDS Team			
User	Job Title	Location	Alias
User1	Team Leader	User1@TDST_ws1	U1
User2	Sr. Design Engineer	User2@TDST_ws2	U2
User3	Design Engineer	User3@TDST_ws3	U3
User4	Sr. Software Engineer	User4@TDST_ws4	U4
User5	Software Engineer	User5@TDST_ws5	U5
User6	Software Engineer	User6@TDST_ws6	U6

Table 5.1: Telecom Software Development Team

3. It reduces paper flow through the organization.

The next section will detail how the IMS can be used to automate this process making it more efficient and reliable. The example will utilize the Telecom Software Development Team (TDST) as the users of the system. An IMO will be given the task of collecting status report information from each team member and returning to the team leader so that the information can be processed and submitted to management.

5.1.2 Automating Status Report Processing

The TDST is a five member development team whose team leader is responsible for developing micro-computer telecommunications applications. Relevant information for each team member is listed in Table 5.1 for convenience.

Each member of the team utilizes the IMS at his workstation and is a registered user of the system. A remote name server database, NS1 has been configured that maintains user aliases for ease of destination specifications. The team leader has also created a destination list alias, called TDTS_DL, in which each member of the team has been included.

The team leader wishes to create an IMO that can route itself to each team member, collect that members' status report input, and return back when the task has been completed. To accomplish this task, the SMM is accessed to create an IMO instance which will

be referred to as *IM*.¹ Upon creation time, the system automatically populates reserved and user-definable message fields with system and default values respectively. The global data space fields are populated first and have been given the following values:

```
IM.GDO.MsgID           = 000001
IM.GDO.CreateTstamp    = 921012:0830
IM.GDO.OrgID           = U1
IM.GDO.OrgAddr         = User1@TSDT_ws0
IM.GDO.CurrDest        = User2
IM.GDO.CurrDestAddr    = User2@TSDT_ws0
IM.GDO.NextDest        = User3
```

As shown above, the message is given a message identifier used for tracking the message through the system. It is also timestamped with the current date and the originators' identification and return address are included.

The user-defined message fields are populated with defaults values as specified in the system configuration file. For this example, the following RCO fields have been populated with values from the team leaders' configuration file:

```
IM.RCO.Priority        = NORMAL
IM.RCO.DlvConfirm      = N
IM.RCO.RetToOrg        = Y
IM.RCO.ITV             = 500
IM.RCO.SPAWN           = N
IM.RCO.DLAccess        = NONE
```

Thus, the default routing parameters for this message indicate:

¹Messages can be reused at any time to minimize creation effort.

- Normal delivery priority.
- No delivery confirmation.
- The message should return to the originator.
- An interrupt timeout value of 500 seconds.
- The message can not duplicate itself.
- The recipient can not perform operations on the routing tables.

The team leader manually sets the destination list type, `DestListType` to **GROUP** and the destination list, `DestList`, to `TDST_DL`. Based on this information the system will retrieve the actual values for the specified group list from the name server database, `NS1`. Thus, `DestList` will resolve to the following list of recipient destinations:

```
IM.RCO.DestList = {User2@TSDT_ws2,User3@TSDT_ws3,
                  User4@TSDT_ws4,User5@TSDT_ws5,
                  User6@TSDT_ws6}
```

The only MCO Data Space field that has a default value is:

```
IM.MCO.AccessAttr = ARCHIVE | PURGE
```

This indicates that a recipient is only allowed to archive (save) and/or delete the IMO. Any other user-defined message fields that have not been populated with defaults must be defined manually. In this example, the team leader defines the **SUBJECT** field to “Weekly Status Report”.

5.1.3 Event Table Specification

Having created an IMO instance, the event table must be defined. As with the definitions of the other message fields, the event table can also be configured with default values again supplied through the system configuration file. Through the configuration file the following event table defaults have been specified:

```
let RET = IM.RCO.RET in
  Event [0] = NIL (default)
  Action [0] = RETURN
  Event [1] = DestUnavl
  Action [1] = FORWARD
  Event [2] = DestNoExist
  Action [2] = RETURN
  Event [3] = ITVTimeout
  Action [3] = FORWARD
  Event [4] = NetwUnavl
  Action [4] = NOTIFY
```

As shown above, the default configuration for the event table specifies that the IMO should perform the following actions:

- Forward itself to the next destination if the current destination is unavailable or the ITV expires.
- Return to the originator if a destination does not exist
- Notify the current user if the network is unavailable when the network is reestablished.
- For all other unspecified events, return to the originator

5.1.4 Message Script Specification

As described earlier, the message script is used to define the events that will take place once the message reaches an intended destination. Since the team leader is interested in retrieving status reports from each member of the group, he creates a script that will route itself to each team member and interactively collect the week's status report activity from each member of the team. The message will return to the team leader when the information has been obtained. The following section outlines the execution of the message through the system.

5.1.5 IMO Execution

When the team leader has finished creation of the message, the SMM is notified that a message is ready for delivery. A MTO is instantiated and the message is put on the send

queue. The first destination as determined from the **DestList** is User1, so the MTO submits the message to the communication sub-system for delivery.

In this example, User1 is active on the network so the message successfully arrives at the destination. The SMM at the location receives the message and instantiates a MTO to process the incoming IMO. The IMO is placed on the receive queue and the ITV is started. User1 is then notified that a message has arrived and is awaiting processing. User1 accesses the awaiting message through the MTO. At this point, a UAO is instantiated and control is relinquished to the UAO.

The UAO then begins executing the message script for the current recipient destination. The actual script is shown in Section 5.1.6. The script first automatically stores the recipient name, current date, and time to the response store for the current destination. The subject of the message is displayed which indicates the message is requesting the weekly activity report of the recipient. The script then prompts the recipient to enter their employee number and the weeks activities status. The contents of the input variable "task-stat" are checked to make sure the recipient responded to the question. If not, the question is asked again until the user responds. The next question is a yes or no response question that asks the recipient if there are any concerns that should be made known. If the answer is "YES" the input is collect and stored. Any final comments are collected in the last question. The script thanks the recipient and terminates processing at that point. The SMM is notified that processing has completed and that the MTO should now forward the message to the next destination.

At this point, control is given back to the MTO and the next destination, User2, is selected and User1 is removed from the list. Again the SMM is notified that the message is ready for delivery. User2 is also active on the network so the message is delivered to that destination. Message receipt at User2, is handled the in the same manner as User1, however in this case, User2 is in a meeting and does not respond to the message. A response timeout occurs in this case. The MTO consults its event tables and determines that the message should forward itself to the next destination. Thus, the message places User2 at the end of its destination list and selects the next destination.

User3 is selected but is on vacation and thus this destination is not active on the network. When the MTO attempts to send this message, it will be notified that the destination is not

available. The MTO will be notified of the event and according to the event table action, the message will be forwarded to the next destination. Since the current destination is not available, the entry is removed from the destination list and the MTO will log an internal message in the response store for this destination indicating the condition.

User4 and User5 are the final unvisited destinations. Both users are active on the network and the message successfully routes itself to each user and collects the desired information. The destination list still contains the User2 destination. User2 has since returned from the meeting. The message routes itself back to User2. User2 accepts the message and enters the requested information.

The MTO updates its destination list by removing User2. Since all destinations have been visited the destination list is empty. However, the **RetToOrg** flag has been turned on which indicates that upon completion of the message mission the message should route itself back to the originator. The MTO determines this destination by accessing the GDO to obtain the return address of the originator. The message is then routed back to the originator following the same procedures for the other destinations.

When the message arrives back to the originating destination, the team leader is notified. Like any other message that arrives, this message is accessed in the same manner. The team leader retrieves the message. In this case however, since the message originated from this destination, the team leader is not prompted to enter the status report information but is allowed to retrieve the information gathered by the message. The team leader extracts the information and begins preparing his report for management.

5.1.6 IMO Message Script

The IMSL message script created for the example described above is provided.

```
STORE $RECNAME  
STORE $DATE
```

```
DISPLAY $SUBJECT
```

```
DISPLAY [ $RECNAME, Please answer the following questions for this  
weeks status report. Thanks. ]
```



```

DISPLAY "Enter employee number:"
INPUT emp-num
STORE emp-num

:Q1
DISPLAY "Enter the status of the tasks to which you have been assigned:"
INPUT task-stat PARA

IF $task-stat = $NIL THEN
    GOTO Q1

STORE task-stat

PROMPT "Do you have concerns or issues that you would like to express?" yn
CASE $$
    y : DISPLAY "Go ahead."
        INPUT concerns PARA
        STORE concerns
        GOTO :comments
    n : GOTO :comments
ENDC

:comments

DISPLAY "Enter any additional comments you wish to make."
INPUT comments PARA

IF $comment = $NIL THEN
    SET comments "No Comments"

STORE comments

DISPLAY "Thanks, $RECNAME"
EXIT

```

5.2 Scheduling a Meeting

In this example, we focus on the functionality of the IMSL language and how an advanced message script can be created to perform a somewhat more complex and time consuming task. The task that we will model is that of determining the time of a meeting such that all attendees agree on the scheduled time. We begin the example with a more formal description of the task.

5.2.1 Task Description

Many times, a meeting must be scheduled in which it is mandatory that the specified attendees of the meeting be present. The actual time of the meeting is dependent on the attendees agreeing on a specific time and date. The task of scheduling this type of meeting can often be a frustrating and time consuming event. For instance, an individual must take on the role of contacting each of the possible attendees to determine the times that they are convenient for that individual to attend the meeting. The times that are convenient for each attendee must then be analyzed to see if there is potential time that the meeting can be scheduled such that everyone can attend. If there is a conflict then the process must be repeated until a time can be arranged. The task is further complicated due to the fact that it may be difficult to contact an individual to obtain a possible time in which case the whole process can be significantly delayed.

One solution to the task described above is to assign the task to an individual and let that individual consult with each attendee until a possible time can be scheduled. This solution is acceptable if the meeting can be scheduled in a short amount of time. However, if the task begins to take a considerable amount of time, then other tasks assigned to the individual may start to be delayed which can lead to further delays and problems. A second approach is to schedule a conference call with the expected attendees. In this manner, each attendee can state the times which are convenient for them and a meeting time can quickly be arranged. However, with this approach the scheduling problem still manifests itself in a slightly different format. That is, when should the scheduling of the conference call take place? If each possible attendee must be present during the conference call then the same set of problems still exists. In the next section we will describe how the IMS provides a solution for this problem that requires minimal effort for those involved in the task.

5.2.2 Automating the Scheduling Process

We believe that this problem can be modeled using IMS to create an IMO to carry out the task of scheduling an appropriate time for the meeting. The IMO can route itself to each individual and request a time in which the individual can attend. If it is determined that a conflict exists, the IMO can reroute itself back to each member and request a new time until it determines a time which is acceptable for all the attendees. At this point the

Thesis Committee	
Faculty Member	Location
FM_1	FM1@ws1
FM_2	FM2@ws2
FM_3	FM3@ws3

Table 5.2: Thesis Committee Members

meeting can be scheduled.

We use an the following description for this example. A graduate student is preparing for the defense of a thesis. The faculty members that have been selected to serve of the student's thesis committee must be contacted to arrange a time for the student's thesis defense. Since faculty members are often teaching classes or attending seminars, it is quite possible that difficulties may arise while trying to arrange the meeting. The graduate student employs the use of **IMS** to create an **IMO** that is assigned the task of contacting each member of the committee to determine a convenient time to schedule the defense. When the time is determined, each member of the committee and the graduate student will be notified of the time. The faculty members and the student can then meet at this time for the defense. Table 5.2 represents the members the thesis committee.

Again we emphasize that the message script will be the emphasis of this example so we will assume the following configuration for the **RCO** and **MCO**. The destination list is initialized to include each member of the thesis committee. The **ITV** timeout value is also set such that if no response is obtained before the timeout, the **IMO** will move on to the next member. Recipients will not be allowed to alter the destination list. The event table specifications will remain as indicated in the previous example.

5.2.3 IMO Message Script Definition

The **IMO** message script which is used to perform the described task is shown below with a description of its functionality following the example:

The following is the **IMSL** script for the scheduling of a graduate student thesis defense

meeting as describe in Section 5.2 of Chapter 5.

```
SET @max-retry 0

SET @dests $$DESTS
SET @first-run true

SET @fm1 "FM_1"
SET @fm2 "FM_2"
SET @fm3 "FM_3"

SET @fm1-datetime NIL
SET @fm2-datetime NIL
SET @fm3-datetime NIL

SET @gs-datetime NIL

IF $@first-run = true THEN
  [
    DISPLAY [ $RECNAME: I am in the process of scheduling a meeting time
              between the members of my thesis committee for the defense of
              my thesis. I am open for any date and time during the week of
              November 23. Please let me know what day and time is convenient
              for you during this time period. ]

    CALL get_time ( RECNAME )

    DISPLAY [ I will check with the other members and let you know
              if this time is acceptable. Thank you.]
    SET $$DESTS := $$DESTS + $$DEST
    IF $@fm1-datetime <> $NIL AND
       $@fm2-datetime <> $NIL AND
       $@fm3-datetime <> $NIL THEN
      [
        CALL verify_time
        SET @first-run false
      ]
    EXIT
  ]

IF $@agree = true THEN
  [
    DISPLAY [ $RECNAME: The agreed upon time for the thesis defense will
              be at $@gs-datetime. Thank you for your input. ]
```

```

IF $$NEXTDEST = $NIL THEN
    SET $DESTS $$ORGADDR
EXIT
]

DISPLAY [ $RECNAME: Scheduling conflicts have arisen in the times specified
        by the committee members. Please check the times indicated and
        provide an alternate time that you can attend if appropriate.
        Thank you. ]

CALL display_time( $fm1, $fm1-datetime )
CALL display_time( $fm2, $fm2-datetime )
CALL display_time( $fm3, $fm3-datetime )

CALL get_alttime ( RECNAME )

CALL verify_time

EXIT

FUNCTION get_time ( REC )
    IF $@first-run = true THEN
        DISPLAY "Please enter a time that is convenient for you to attend:"
    ELSE
        [
            PROMPT "Do you want to enter an alternate time?" yn
            CASE $$
                y : DISPLAY "Enter a new time:"
                   GOTO :input
                n : RETURN
            ENDC
        ]
    ]

:input

    INPUT date DT

    IF ( $REC = $@fm_1 ) THEN
        SET @fm1_datetime date
    ELSE
        IF ( $REC = $@fm_2 ) THEN
            SET @fm2_datetime date
        ELSE
            SET @fm3_datetime date
        ]
    ]

ENDF

```

```

FUNCTION display_time ( REC, DATE )

    IF $DATE <> $NIL THEN
        DISPLAY $REC has suggested @fm1-datetime as a possible date.

    ENDF

FUNCTION verify_time
    IF @$fm1-datetime <> $@fm1-datetime <> $@fm3-datetime THEN
        [
            SET @max-retry := @max-retry + 1

            IF @max-retry = 5
                SET $DESTS $$ORGADDR
            ELSE
                SET $DESTS $dests
            EXIT
        ]
    ELSE
        [
            SET @$gs-datetime $fm1-datetime
            SET $DESTS $dests
            SET @$agreed true
            EXIT
        ]
    ENDF

```

In this script, several features of the IMSL language are used. Global variables are included that maintain certain values throughout the existence of the IMO. These include the current date and time input by a recipient, the recipient name, and control flags which are described later. Several functions are also used which simplifies the processing of the which will be defined as necessary.

The global variables are initialized with defaults only upon instantiation of the IMO. The original destination list is saved in a global variable. After times have been collected from each member, the times will be analyzed to determine if there is an agreement on the meeting date. If so, each member will then be notified of the agreed upon time and the IMO will return and inform the graduate student. If there is a conflict, then the destination list will be reset, and the IMO will return to each member and allow an alternate time to be entered by that member. This process will continue until a time is agreed upon or

the number of iterations of the process exceeds a predefined maximum that is set by the graduate student. This limit is imposed so the process will terminate if it appears that a date can not be agreed upon.

5.2.4 IMO Message Script Description

On the first iteration of IMO processing, each committee member is consulted. The subject of the message is displayed followed by a greeting message from the student requesting the recipient to enter a time that is convenient during the time frame indicated by the student. The function “get_time” is called that will request a time from the the recipient. Based on the current recipient, the time is stored in the appropriate global variable.

Since each member must be contacted, the destination of the current recipient is added to the end of the destination list in the RCO. This allows the current recipient to remain active in the list of destinations in case the first iteration fails to obtain an agreed upon time. A message is displayed indicating that the other members of the committee will be consulted. Only after all member have submitted initial input times will the analysis take place.

After the first iteration, each member will have input a time convenient for them. The function “verify-time” is called that evaluates whether each member has agreed to the same time. The flag “@first-run” is set to false to indicate in the remaining iterations, if needed, that the processing logic should now input alternate times.

If all times are in agreement then the global variable “@gs-datetime” is set to the agreed upon time. This variable is simply used to indicate to the graduate student the time of the meeting. The destination list of the RCO is set back to the original list that was stored in the global variables “@dests”. The flag, “@agreed” is set to indicate that a time has been agreed upon by all the members. The IMO then routes back to each member and indicates the time of the meeting to the recipient. An explicit check is then performed such that if no more destinations exist, the destination list is set to return back to the graduate student.

If there is a conflict in times, then a check is made to see if the IMO has reached the maximum number of iterations for determining the time of the meeting. If it has, the destination list is reset to return to the graduate student, otherwise, the destination list is set to the original list.

On the succeeding iterations the recipients are notified of a scheduling conflict and alternate times are required. The current time input by each member are displayed using the function “display_time”. The recipient is then prompted to enter an alternative time if necessary. The “get-time” function is again used. However, since this is not the first iteration a different message is displayed. The recipient can either elect to keep the time entered previously or to enter an alternate time. In either case, the time will again be verified using “verify-time”. If all times are agreed upon then the members will be notified and the IMO will return to the graduate student with the time. If not, then the next iteration will be attempted.

5.3 Summary

In the preceding examples we have demonstrated how **IMS** can be used to model certain routine office tasks and then provide the mechanism for executing the tasks in the form of a message object. We have shown how information flow can be increased while at the same time optimizing a users time for performing other important tasks while the message object perform tasks which can be tedious and time consuming at best. In the conclusion of this thesis we expand upon the usefulness and the need for a system such as **IMS**.

Chapter 6

Conclusions and Further Research

Since communication is a primary function of many office tasks, it is natural to envision a system whereby messages can provide advanced capabilities which would allow the automation of certain tasks. In this thesis we have presented a model for an advanced computer-based message system (CBMS), called the Intelligent Message System (IMS), to achieve this goal. Our system is based on concepts of objects and active messages. The goal of this system is to expand the functionality of traditional CBMSs by allowing messages to be programmed to perform a specific task. Using intelligent messages to automate routine tasks can increase the productivity within the office environment by allowing office workers to concentrate on other non-routine tasks. In the remaining sections, we will present:

1. Contributions of this research.
2. An overview of IMS.
3. Relationship of IMS to other models.
4. Future areas of research.

6.1 Contributions

- Identification of the characteristics of an office environment with respect to the automation of office tasks and procedures.
- Definition of the role of traditional computer-based message systems in the automation of office tasks and procedure.

- A survey of research work related to the automation of office procedures via messages.
- Specification and design of an advanced computer-based message system, called **IMS**, which allows a message to be programmed for performing a specific task which make routing decisions predicated on the occurrence of certain events.
- Formal specification of the syntax for the Intelligent Message Script Language using an extended form of BNF.
- Formal specification of the major functions of **IMS** using the Vienna Development Method specification language.
- Description of examples which illustrate the functionality offered by the system.

6.2 IMS Overview

IMS is a model for an advanced computer-based message system in which messages are viewed as active and intelligent objects that can be programmed to perform a specific task. A message in this system is called an *IMO* for Intelligent Message Object. An *IMO* is composed of three internal objects. The *GDO*, or Global Data Object, maintains global data that is used during routing and execution of the message mission. The *RCO*, or Route Control Object, is responsible for routing the message to the intended destinations. This responsibility is referred to as the *secondary message function* of the message. The *RCO* maintains a route event table that specifies a set of events and corresponding actions to execute should a particular event occur during routing. The *MCO*, or Message Control Object, is responsible for executing the “mission” or task which has been assigned to the message. This responsibility is referred to as the *primary message function*. The *MCO* maintains a methods table which allows the inclusion of *IMSL* methods that can be used during execution of the message script. Each object also maintains an internal data space for data specific to that object.

Each task is programmed into an *IMO* using the *IMSL*, Intelligent Message Script Language. *IMSL* is a simple interpretive language that provides a flexible command set for programming tasks. Commands exist for performing user I/O, control flow, data storage, and computations. Each script is interpreted at a recipient site

Each site in the **IMS** environment must utilize a System Mail Manager (*SMM*). The *SMM* provides all the facilities for *IMO* administration, creation, and transfer. Each of these functions exists as a separate environment within the *SMM*. *RCO* and *MCO* configurations

may be saved for reuse at a later time to minimize IMO creation time. A Message Transfer Object is utilized for sending and receiving IMOs. A User Agent Object is utilized for performing the message mission.

6.3 Relation to Other Models

Several related models were surveyed in Chapter 2. Each of these models provides a specification for using messages as a medium for automation of certain tasks. These models share many similar characteristics while at the same time offer unique approaches with respect to these messages. The models which were surveyed include:

- Vittal's R2D2 system (Vittal 1980).
- Hogg's Imail System (Hogg 1985).
- Mazer's Message Management System (Mazer & Lochovsky 1984).
- Tschritzis' *KNO* model (Tschritzis & Gibbs 1987).

Our model also shares many similar characteristics with the above models. However, our approach has introduced several distinguishing features which are enumerated below.

1. A message, called an IMO, is composed of multiple objects. Each object represents the encapsulation of a specific function of the message.
2. A Route Control Object (RCO) manages the routing of the IMO through the system. A route event table is utilized by the RCO to facilitate efficient routing. This process is characterized as the "secondary function" of the message.
3. A Message Control Object (MCO) manages the execution of the message task at each recipient site. This process is characterized as the "primary function" of the message.
4. A System Mail Manager (SMM) provides the user interface to the system at each site. All facilities for sending, receiving, and managing IMOs are provided.
5. A message script language called IMSL was defined for programming message tasks.

6.4 Future Research

The research work presented in this thesis represents a detailed specification only. As is the case with most research work, issues are raised will require further study. Clearly the

opportunity exists for developing a prototype of the **IMS** system. Implementation would allow a more thorough understanding of the run-time issues and performance requirements for such a system. We identify the areas which require further research below:

1. Reducing the complexity of IMO creation. Many steps are required during the process of creating a new IMO. Attributes must be defined, the Route Table must be configured, and the Message Script task must be written. Allowing the reuse of previously defined IMOs will help in this effort.
2. Minimizing the size requirements of an IMO message script. The amount of space required for an IMO must be minimized to increase the efficiency of IMO routing through the system. A method for tokenizing a message script would significantly reduce the size of an IMO.
3. Message Script verification. A method for verifying a message script must exist to ensure the integrity of the system is not violated.
4. Support for non-textual data. **IMS** provides support for only textual (ASCII) data. However, a multi-media IMO can be envisioned that would allow inclusion of other types of data such as facsimile, audio and visual data, and graphical images.
5. Non-heterogeneous environment support. This requirement would allow an IMO to transfer to locations which support different underlying computational domains. Thus, the constructs supported by **IMS** must be flexible enough to allow IMO execution within these environments.

To fully realize a system of this type will require much effort, however, it is believed that the benefits and/or drawbacks encountered will provide a beneficial influence on the practical implementations of other intelligent message systems.

Bibliography

- Agha, G. (1986), *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press.
- Bruder, J. et al. (1981), User experience and evolving design in a local electronic mail system, in R. Uhlig, ed., 'Computer Message Systems', North-Holland Pub., pp. 69–78.
- Ellis, C. & Gibbs, S. (1985), Active objects: Realities and possibilities, in D. Tschritzis, ed., 'Office Automation', Springer-Verlag.
- Giuliano, V. (1982), 'The mechanization of office work', *Scientific American* pp. 149–165.
- Guttag, J. (1977), 'Abstract data types and the development of data structures', *Communications of ACM* **20**(6), 396–404.
- Hewitt, C. & Baker, H. (1977), Actors and continuous functionals, in E. Neuhold, ed., 'Formal Description of Programming Concepts', North-Holland.
- Hogg, J. (1985), Intelligent message systems, in D. Tschritzis, ed., 'Office Automation', Springer-Verlag.
- Jones, C. (1989), *Systematic Software Development using VDM*, Prentice-Hall International.
- Kim, W. & Lochovsky, F., eds (1989), *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley.
- Lewis, H. & Papadimitriou, C. (1981), *Elements of the Theory of Computation.*, Prentice-Hall.
- Mackay, W. (1988), 'Diversity in the use of electronic mail: A preliminary inquiry', *ACM Trans. on Office Information Systems* **6**(4), 380–397.
- Mazer, M. & Lochovsky, F. (1984), 'Logical routing specification in office information systems', *ACM Transactions on Office Information System* **2**(4), 303–330.
- Nierstrasz, O. (1989), A survey of object-oriented concepts, in W. Kim & F. Lochovsky, eds, 'Object-Oriented Concepts, Databases, and Applications', Addison-Wesley.
- Patterson, D. (1990), *Artificial Intelligence and Expert Systems*, Prentice-Hall.
- Robert, L. (1970), Computer network development to achieve resource sharing, in 'Proc. of Spring Joint Computer Conference', ACM, pp. 543–549.

- Shriver, B. & Wegner, P., eds (1987), *Research Directions in Object-Oriented Programming*, MIT Press.
- Tsichritzis, D. & Gibbs, S. (1987), Messages, messengers, and objects, in 'Proceedings of the IEEE Symposium on Office Automation', pp. 118-127.
- Tsichritzis, D., ed. (1985), *Office Automation*, Springer-Verlag.
- Vittal, J. (1976), *MSG Manual*, Cambridge, Mass.
- Vittal, J. (1980), Active message processing: Messages as messengers, in 'Proceedings of the International Symposium on Computer Message Systems', Vol. IFIP TC-6, Ottawa.
- Woo, C. & Lochovsky, F. (1986), 'Supporting distributed office problem solving in organizations', *ACM Trans. on Office Information Systems* 4(3), 185-204.
- Woo, C. & Lochovsky, F. (1987), 'Viewing communication as a problem solving activity: An enrichment towards supporting cooperative office work', *IEEE-CS Office Knowledge Engineering* 1(3), 18-22.