INTELLIGENT DATA OBJECT MANAGEMENT SYSTEM (IDOMS)

By

Kathy Pedersen Huml

B.A., North Central College

MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Kansas State University
Manhattan, Kansas

1986

Approved By:

Major Professor

CONTENTS

LIST OF FIGURES

1. Chapter 1: Introduction

1.1 Overview

This report describes a prototype design for an Office Information System using an Intelligent Data Object (IDO). In order to define an IDO and describe its potential use within a system one must understand the concept of an abstract data object. An abstract data object has been defined to be a single class or type of object with a set of operations that function against that object. [LIS75] "An abstract data type is not available as a primitive in a programming language but built using the data types provided by the language and other abstract data types". [GEH82] An intelligent data object is a self-contained set of instructions that can be routed within a potentially distributed network in a similar fashion to the routing of a message. The IDO is designed to perform some relevant activity i.e., have a specific function. In particular the function of this IDO will be consistent with activities that generally occur in an office environment.

Almost all of the activities occurring within an office are centered around the use of paper forms. [LAR81] [GEH83] They are the most pervasive element found today in office information systems. A substantial amount of the average office workers time is spent completing all types of forms

and passing them on to other office workers for their input and so on. Many applications of office information systems require that forms be routed to several individuals and/or workstations within an office or network in order to complete the processing of information. Because paper forms seem to be a "way of life" for most office environments there are several advantages to utilizing a "form-oriented" structure for the IDO [GEH82] [TSI82c]:

- forms allow logically related data to be handled as a single entity
- office workers are accustomed to dealing with them
- forms can be traced
- they can be routed
- availability of completed forms can be restricted (provide security)

The report that follows concentrates on the implementation of an IDO as a form-based abstract data object. The first chapter reviews some of the current research into the area of office information systems, particularly form-based systems. Chapter 2 focuses on a general overview of the proposed system, the Intelligent Data Object Management System (IDOMS). The use of the Form Definition Language (FDL) is explored and more clearly defined in Chapter 3 and Chapter 4, respectively. Chapter 5 will deal entirely with the implementation of the language constructs in the FDL that provide the user with the capability of data generation

based on a limited set of inputs. Testing considerations will be discussed in Chapter 6. A final analysis of the design and implementation will be presented in Chapter 7.

1.2 IDO Definition

Although the concept of a system based on the sending of messages is not new, more emphasis is being placed on developing systems which allow users to define complex processing instructions and routing patterns for messages. [VIT81] According to McBride and Unger in order to incorporate this "needed" intelligence into the IDO or message it must have access to the following basic information [MCB83]:

- the data instance (a single data record)
- processing instructions (control Petri Nets)
- token to identify the position in the net
- routing history
- capabilities list (describes files to be accessed and operations for those files)

All of the information is required in order for the IDO to complete its processing as it is routed throughout the network and thus is logically part of the IDO. Not all of this information resides physically in the IDO. The only information physically contained within the IDO is:

- unique key which identifies the form
- the data instance

- list of destinations
- sender identification (originator)
- routing history

The IDO is physically structured to include a header and a body. [TSI82a] The header contains formatted data that is critical to the sending/receiving of a message (e.g., sender identification, destination, form type, date, etc.). The body of the text consists of words (e.g., data instances, list of destinations, routing history, etc.).

The concept of an Intelligent Data Object has some similarity to that of an "actor". An actor is defined to be a communicating object that interacts with other actors via messages. [HEW77] Actors execute asynchronously upon receipt of a message according to a presubscribed "script". [BYR82] An example of an actor based system is Officetalk-D which employs the use of "actors" in mapping people to roles and roles to activities. [ELL80] As the IDO routes itself within the network it will behave according to a predefined set of instructions established at the time the form was created. The difference lies in the fact that the intelligence for an actor-based system is built into the script whereas the intelligence of the IDO is built into the data object and is carried along as it is routed within the network.

1.3 IDO's Use Of A Form

A form has been defined by Gehani as an abstract data type. [GEH82] Treating it as such will facilitate automatic error checking and the enforcing of access to only authorized users. A form can be structurely defined to include the following components:

- field definitions
- display
- operations or processing instructions
- access rights
- routing instructions
- constraints (error checking)
- calculations to be performed to generate new fields

Fields are defined as items or requested information to be placed on a form. Gehani defines several different kinds of fields defined for use within the form [GEH82]. These are listed in Appendix 1. Each field has related to it a type (character, integer, etc.) and a range of allowed values. The definition of the field must be satisfied before the field can be populated. Included in the field definitions are the rules, if needed, to establish the order the fields are to be filled in by the user.

The data-populated fields are viewed by the user through a template. "A form template is a mapping which maps a form

instance into a message in a particular communication medium." [TSI82c] The display area of the form contains the description of the screen image (template) to be provided to the user for a particular form.

Operations or processing instructions act on form instances. A form instance can be regarded as a snapshot of a data record. A list of operators include:

Replace - used to modify the fields in a form

Create - used to add a new instance of a form

Delete - removes a data instance from a form type

Retrieve - displays the form, searches for match of a desired field or value of a field

Mail - sends the form instance to another user

Copy - form can be copied into another form of the same type

Read - allows user to view a data instance

The operations allowed for the IDO should encompass a large variety of office activities. The criteria for defining the processing instructions should be based on the values of the data instance residing in the IDO. Not all instances of the same form will require the same processing instructions. Several systems have been developed around the concept of Information Control Nets and augmented Petri Nets. These systems employ office definition languages that intricately step through every procedure that occurs within a particular

office environment. Utilization of a token to identify the processing state of that data instance is maintained. SCOOP [ZIS77] uses a system along this order. As the data instance is routed and processed the token or pointer is automatically updated and the message is sent to the next processing station.

Official form instances (those residing on the database) should be distinguished from those that are currently being operated against. This capability allows the user the opportunity to view the form instance before officially storing it into the database.

"Users of electronic forms should be allowed to interact with a form according to the prespecified operations supplied by the form designer." [GEH82] Not all users will be allowed to view or modify all of the fields within the form provided the creator of the form has declared the field to be "proprietary". This control will be maintained by modifying the template to be used to view the data and by defining access rights on a per user basis. [GEH82] [TSI82c] Users that are not allowed to view particular data (e.g., payroll information, etc.) will not be aware that the particular data item even appears on the form. The user will not see a blank entry or a field heading.

Form instances will be passed around the network via "send" and "receive" commands. Each form possesses internally defined algorithms to determine the list of destinations for this form instance to be routed. The routing of the IDO must incorporate more intelligence than just being able to behave according to some generic algorithm defining a path for all instances of a particular form. A finite set of routing patterns can be defined for a form type. The selection of the proper routing pattern will be based on a control attribute or what Tsichritzis terms a "routing predicate". Form instances can be grouped according to a "class" of form. Each class is defined for a particular routing pattern not necessarily for the form type. This allows different instances of the same form to follow different routes around the network. "All forms in the same class take the same path of stations." [TSI82c] For example if a voucher of $10 had to be approved it may not follow the same path for approval as a voucher for $5,000. The intent of the algorithm is to define a routing pattern that is dependent on the value of the control attributes. [TSI82c] These algorithms must be defined by the creator of the form to define routing predicates which are mutually exclusive.

Various integrity constraints for the data can be provided to insure the quality of the data on the database. Error checking should consist of three levels of error detection:

type checking, validation of field values within one form, and consistency checking across forms or form instances.

ODIN is a system designed to automate the construction of electronic form entry, processing, and retrieval of records from a relational database. [FER82] It was developed for a system that was extremely data intensive and whose data format was highly prone to major changes in format and continuous update. The integrity of the data is extremely critical due to the fact that the data is utilized as part of a telephone switching network. To insure reliable data ODIN has implemented an extensive, sophisticated system of error checking. [FER82] [DIP83] ODIN incorporates an interactive, full screen editor to enter, modify, and remove data instances. As each field is entered domain checks are executed. Structure checks (fields that are logically grouped together) are made immediately after all the fields of that structure have been entered. List element checks are executed against any repeating groups. Finally form checks are made once the entire form has been created.

According to Stonebraker (INGRES) integrity control algorithms can be applied to append, delete, and replace statements as they act on data to be stored or removed from the database. These integrity constraints if desired at each update, should be implemented at a high a level as

possible to insure the most efficient update process. "Schemes which append integrity controls at lower levels have considerable difficulty enforcing complex controls...". [STO75] Gehani identifies the concepts of "preconditions" and "postconditions". Preconditions must be satisfied before a field is filled and postconditions must be satisfied after a field is filled. These constraints are placed on the data contained in the form or form instance to assure that the complex relationships existing between the fields is preserved. "In the case of electronic forms, specification of these conditions will enable the verification to take place automatically and will happen while the form is being filled out." [GEH83]

The final area to be discussed within the form-oriented IDO is the area of generating new fields. There are basically two categories of fields that will be generated either by the system or through the calculation of a user specified algorithm. The first category, automatic fields consist of data such as the time of day, user identification, etc. that will be supplied by the system. Virtual fields cannot be filled in by the user. They are calculated according to predefined rules set down by the creator of the form. [GEH83] Forms that generate fields automatically are viewed by Tsichritzis as "smart" forms. The implementation of "smart" forms can be taken one step farther to include the

modifying of values of fields dependent on time of day conditions, state of the system, etc. These are viewed as "smarter" forms. [TSI82c]

1.4 Examples Of Form-Based Systems

Many previously developed systems, based on the concept of form manipulation have been successfully designed: Officetalk-D [ELL82], System for Business Automation (SBA) [DEJ80], and Query By Example and Office Procedures By Example (QBE/OBE) [ZLO81]. Each of these systems employ the use of a relational database and stress the need for a simple, easy-to-use interface for untrained users.

QBE/OBE uses a highly structured, predefined set of "box-like" forms which operate on the database to retrieve, add, delete, or create data instances. It is an interactive system that provides a facility for manipulating data. A mail system to send instances of data within a local or distributed network is available. Security is provided through the concept that forms are owned by their creators and authorization to query or modify data within these forms must be explicitly defined. QBE/OBE provides a "trigger" mechanism that causes the system to act automatically when specified conditions are met. These conditions arise when either predefined modifications are made to the data (modification triggers) or by a time trigger. These

triggers are also defined by the creator of the form. [ZLO81]

SBA, of which QBE/OBE are subsystems, is constructed out of abstract objects also called "boxes". Each of these abstract objects is regarded as an independent entity. Users are allowed to create forms, however, not with much flexibility i.e., the forms must follow a standard format. SBA employs triggering devices, modification and time triggers, and the use of a mail system. Distribution of applications over many users and machines is supported by SBA. [DEJ80]

Officetalk-D like SBA and QBE/OBE uses an electronic display screen to interface with the user. The system provides a separate language for office procedure descriptions. These descriptions are based on the use of Information Controlled Nets (ICN's) [ELL82] which define activities and establish a precedence among the activities. Use of ICN's has promoted the development of more generalized procedures, forms, and aids for management in understanding and tracking the workflow in an office. Emphasis is placed on providing a definition language for forms and office procedure descriptions that can be used by individuals that are not computer specialists. There is considerable flexibility for the user in that he/she may define new forms. Compared to

SBA and QBE/OBE much more effort is concentrated in identifying the office procedures and providing forms to support those activities. [ELL82]

The use of forms promotes a very structured approach to office activity and their use is familiar to most individuals. [TSI82c] In order to streamline some of these activities it is believed that the use of automated, electronic forms that can be routed promotes a much more productive, efficient working environment. The intent of this proposed system is to provide an electronic, form-oriented IDO that contains the built-in intelligence to route itself within the network.

1.5 Environment Description

"Any system or tool that is sufficiently general to be employed without modification in a wide range of office contexts cannot, by definition be oriented toward the particular needs of a specific office." [HAM80b] The activities and procedures are so unique to individual office environments that it is essential to analyze each office information system. By current standards custom-built software is required in order to accommodate each application. Because these customized software packages are very expensive and time consuming to produce an emphasis on developing models for the specification of office procedures

has evolved. Research in this area has produced several promising models which may be of potential use for implementation within the Intelligent Data Object Management System.

The Information Control Net (ICN) model evaluates the "control structure", organization of activities, and the "information structure", i.e., communication or use of information in a procedure. [CO080] It produces a diagram of an office procedure which is very similar in scope to the use of Petri Nets. Construction of an ICN model is an iterative process where initially activities are defined, a precedence order for these activities is established, and repositories of information are identified. After the procedures are detailed streamlining is performed in order to reduce an ICN model to the basic communication and information requirements of an office procedure. [CO080]

The second model employs the use of an office specification language. An office specification language is used to describe in detail the specifics of the functions performed within an office and the processes conducted to realize them. [HAM80b] [KON82] Hammer and Kunin have identified five design criteria for an office specification language:

- formal; limited number of constructs

- able to be read and be understood with a small amount of training
- support process of writing descriptions
- high-level and nonprocedural
- modifiable

The concept behind both of these models is to furnish a language that is "...extremely usable for a large class of applications, rather than minimally adequate for the universal class of applications." [HAM80b] Officetalk-D, an electronic form-based office information system, employs the use of ICN models to provide graphical pictures of all of the activities currently in progress. This system has potentially created the capability of allowing automatically generated activities to be initiated through the use of documented office procedures, however this is not currently part of its design. [ELL82] The System for Computerization of Office Processes, SCOOP, is based on a similar concept of using "augmented" Petri Nets which provide a mechanism for encoding knowledge about office procedures. This system utilizes an event-driver that detects that some office situation exists, determines what procedures apply to this situation, analyzes the procedures to determine what actions are to be taken, and then takes them. [ZIS77]

2. Chapter 2: The Intelligent Data Object Management System (IDOMS)

2.1 Overview

The prototype, IDOMS, has been designed to send a form-structured intelligent data object throughout the network to a set of individuals who are defined to have some interface with it. This interface to the IDO may include actions requiring additions or modifications to the data within the IDO, it may act as a trigger device for an individual to perform some alternate activity, or it may require no action i.e., its function is strictly informational.

The design of IDOMS has been logically divided into the following areas: the dispatcher/station manager, database management, the form model, the screen manager, and the form processing interface. What follows is a general overview of the major elements that compose IDOMS. It is not intended to provide what might be interpreted as high-level design for these areas. Its purpose is to give the reader a general perspective on the basic organization of the Intelligent Data Object Management System.

2.2 The Dispatcher/Station Manager

Generally speaking, a node consists of one station manager and one or more users. This will be referenced as a local network. Two or more nodes connected through a communication link are considered to be a distributed network. Communication between nodes/users must proceed through station managers.

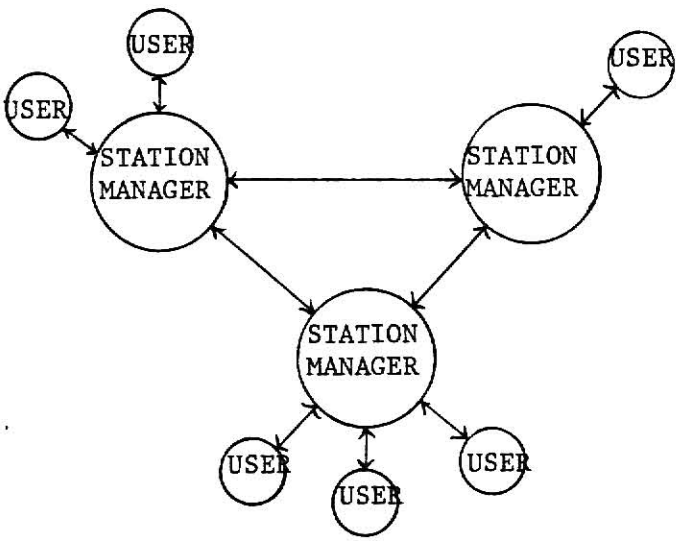

Figure 1. Sample Distributed Network

For this application communication is considered to be the sending and receiving of messages. The term network will imply distributed network.

The station managers regulate and synchronize the movement of each of the IDO's i.e., messages within the network. Each message type whether it be a serial (sent to one

receiver at a time in a predefined order) or parallel (many receivers and no specified order) must be sent by a user via the station manager in order to be routed. Within the IDO will be a list of routing destinations. The station manager will access the list and send it to the next individual/groups of individuals. Each message will be considered an independent entity. The manager will be responsible for maintaining a log history file whereas the routing history of the IDO is resident in the Intelligent Data Object. It is assumed that the manager will update this information. The sender identification, the form key, and the receiver information will be contained in the log history. By examining the log history the message may be located and a trace can be made of the message, i.e., a list of the station managers and users that it has visited. [TSI84] Messages will be sent to the "rje" directory of the station manager or user. This will insure that the message can be sent and received even if the user/station is not currently active. Whenever a message is sent a response will be sent back by the receiver (user/station manager) to acknowledge that a message has arrived. If no response is received the message will be re-sent.

A user accesses his/her messages by a mail-like command which will access the user's/station manager's own rje directory to retrieve messages. A user will be allowed to

operate only on messages within its own directory. A message is only accessible to the individual owning that directory.

2.3 Database Management

The data for each form instance will be retained in a relational database. For the purposes of this paper, a relational database is defined as a table where each row of the table corresponds to a tuple of the relation and a column represents the set of data instances. [MCL76] The database will be strictly a location to "hold" data. It will not provide any kind of type checking or range checking for any of the fields defined within the form.

Operations on forms are issued from users where a user is identified as an individual and not a process. Some of the defined operations include:

- Create - create a new relation(form)
- Retrieve - get a particular instance of data
- Append - add a new data instance to an existing relation
- Replace - allows user to modify or add values to existing data instances
- Delete - removes a data instance from a relation (form); archives data instance

(Deletion operations of form instances must be carefully controlled. Once a form instance has been created it cannot

be deleted. If a delete request is made the data instance is archived and removed from the relation.) [TSI82c]

Concurrency of the processing of data within a distributed system is of paramount importance. Use of a locking algorithm will guarantee that the same data on form instances cannot be processed simultaneously by different users. The database management system will incorporate a lock on a given data instance.

2.4 Data Dictionary

The data dictionary contains all of the data definitions for the forms and the fields contained therein. Its purpose is to provide descriptions about the data entities and the relationships of office activities to the other areas of IDOMS. It should be the overall physical documentation of each and every relation existing in the database.

In order for the data dictionary to be used by the other areas of IDOMS a formalized data definition language must be implemented for use with the data dictionary. Officetalk-D based its data dictionary entries on the ICN models defined for all office activities. A module to translate the ICN or map the elements to tuples and relations was incorporated. [ELL82] The major benefit of mapping back Information Control Nets is that all relationships existing within the

office environment are represented. A consideration might be to drive the data dictionary from the Form Definition Language and from a streamlined ICN which is to be discussed in section 2.5. This area has been deemed as necessary by IDOMS, however, there are no current plans to implement this in the immediate future.

2.5 The Form Model

The emphasis on "paper forms" in offices has promoted increased development of electronic form based systems. The prototype, IDOMS, is one of those systems. Each of the electronic messages or IDO's routed throughout the network is form based. The form can be divided into the following units:

- field definitions
- display
- processing instructions
- access rights
- constraints (error checking)
- routing instructions
- calculations to be performed

Because the intent of this form is to provide some "intelligence" the structure becomes somewhat complex and the need for specific, detailed information about how the IDO is to be processed must exist in the form definition. Complexity coupled with the knowledge that no one form can

handle all activities effectively encourages the idea of allowing the user the flexibility of creating his/her own forms. This poses a problem in that most individuals within an office environment do not have the programming background to define workable forms. Other systems such as SBA and QBE/OBE have defined new forms based on the use of simplified language constructs that non-programmer types have successfully used with little or no training to create new forms. [DEJ80] "The main innovation of OBE is that end users themselves create (define) the objects on the two dimensional displays in much the same way they create these objects manually on paper." [ZLO81]

The Form Definition Language (FDL) which has been created for IDOMS is designed to be used by an individual with little or no programming background and runs on a electronic display. The FDL will provide the language constructs to support the above seven areas of the form.

2.6 The Screen Manager

The electronic forms will be accessible through a CRT (Cathode Ray Tube) device. The screen manager will provide a menu-driven, interactive dialogue that will allow the user access to the database, creation of a form, creation of the IDO, sending of the IDO, receiving of messages, and possibly access the data dictionary. Actual execution of these

activities will be handled by the forms processing interface where the screen manager's responsibility includes the mechanism to request specific actions. A template will be used to view the data and add or modify data instances. Not all users will be allowed to view the forms. Restrictions may range from entire sets of forms to items within a form.

2.7 The Forms Processing Interface

The forms processing interface is responsible for execution of error checks and calculations of virtual and automatic fields, determining the routing list, verifying access rights and processing instructions. Its major function is to create the IDO and the respective data and send it to the station manager for routing. The FDL provides the language constructs to allow the designer to create forms whereas the forms processing interface will execute the program defined as the form.

3. Chapter 3: Form Definition

3.1 Overview

Significant strides have been made in the recent past regarding the managing of information in the office environment. Improvements to the mode of interfacing with data has evolved to a state where use of computer systems in small businesses is as prevalent as use within larger corporations. Because these computer systems have become so common-place emphasis has been placed on developing systems that incorporate their use with everyday activities. This move to office automation coupled with the fact that a majority of the office activity surrounds the use of forms has led to the development of systems that are based on form management.

3.2 Basic Design Principles

All operations within the Intelligent Data Object Management System are based on the use of forms. The intent of this system is to provide a generalized form definition such that a user is capable of defining forms to handle any or all office activities. Not all forms will require action by receivers i.e., they are considered passive. Some are active forms in that they require the receiver to perform some activity. By keeping the form definition requirements as basic as possible the user should be able to create forms

that cover the whole gamut of office activities.

The form will be divided into seven sections: fields, display, operations, access rights, routing, constraints, and calculations. To provide the reader with a better understanding of the form and its components an example has been included in Appendix 2. The example provided depicts a sample order form for equipment. This form is included to exemplify the principles and the component parts of the form not the actual, implemented language constructs to be provided by the Form Definition Language.

For a general overview of each of the seven components of the form refer to section 1.3, "The IDO's Use Of A Form".

3.3 Form Definition Language Requirements

In order to provide flexibility to the user a Form Definition Language will be provided. The purpose of the FDL will be to allow the user to produce new forms using simple "English" and "Math" like constructs. By preserving simplicity in the language the creator will be able to successfully design a form with minimal training. This is one of the most critical factors to be considered regarding the implementation of the FDL because most office workers do not have extensive programming backgrounds. The following guidelines adapted from Zloof [ZLO81] and Larson [LAR81]

should be adhered to as closely as possible during the design phase of the form definition language.

Forms Oriented.

The language is used to define and access data in a similar fashion to paper forms.

Minimum Knowledge Required To Start.

The user must not be required to spend a significant amount of time being trained in order to start using the language.

Minimum Syntax.

The language syntax should be simple but complete.

Adaptability.

The language must provide some element of flexibility for the user. It should be able to effectively handle activities that occur in various types of office environments.

Not Sensitive To Change.

The language must allow for change in the form specification without requiring unnecessary levels of effort on the user's part.

Uniform Language.

A similar type syntax should be used for all language constructs.

In addition to these general objectives of the language there are some specific requirements of the FDL. These requirements are itemized here because their implementation or lack of implementation potentially affects the calculation area of the form. These are beyond the scope of requirements for the calculation unit.

- In an effort to keep the number of created forms to a reduced level and to keep related data together the concept of multipaged forms should be implemented. Many forms in an office environment are extended over more than one page. It is preferable to keep the electronic form consistent with the paper form. Retaining the use of a multipaged form should simplify or reduce the number of relations that would be accessed during the calculating of new fields.

- Situations will exist where other forms must be accessed in order to read or retrieve data. The implementation of the form will be required to allow not just access to local data but to global data as well. S. Bing Yao has referred to this concept as "form linking". "The form designers can specify the

linking as the form is being designed, and new links can easily be added to a form by modifying the specification." [YAO84] Effectively this means that a form can activate more than one form for processing.

- An overall objective in defining a form specification is to require the creator to identify all the essential fields, i.e., all those that appear on the form, however, a means to distinguish the fields that will be generated for the user must be stipulated. Because of the complexity of forms and the fact that some of the fields will be calculated for the user a somewhat sophisticated screen manager is needed. The user of each form should be led through the fields in a specified order and those fields that the user in not required to fill should be skipped over.

- Many forms within an office environment require approvals, letters require signatures, etc. In order to support this the form must employ a mechanism to affix signatures to itself. Once a signature has been attached to the form selective locking of fields must be provided. A locking algorithm guarantees that the critical values in a form instance cannot be manipulated.

- Repeating groups are defined to be data fields that

have a potentially unlimited number of entries within one data instance. They are identified by both Gehani [GEH83] and Tsichritzis [TSI82c] as necessary for real world applications of form definitions. The implementation of this feature has a potential impact on the calculation portion of the form. Because both authors have recognized its need and have declined to implement the capability, it will be also expressed here as a needed facility. The language constructs designed for the calculation unit support the concept of repeating groups.

The following paragraphs deal specifically with the requirements that directly relate to the calculation unit of the form definition language.

- As a form is routed throughout the network individuals may be required to add, modify, or delete data in order to complete the processing of the data instance. As these changes are applied new fields may be calculated and added to the data instance and those already represented in the data instance may necessitate recalculation. In order to preserve the consistency of the data and eliminate possible side effects a means to identify the changes and take action to recalculate the fields must be available.

- Situations may occur or conditions may come to exist in which specific actions should take place. These procedures (calculations) should be initiated whenever certain preconditions are met. These will be referred to as "triggers". There are basically two types of triggers, modification and time triggers. These circumstances can occur because of the dependencies on data residing within a form or between two or more forms. For example if fields were added to form.x based on a calculation using fields on form.y and the fields on form.y are modified a mechanism to automatically regenerate the fields on form.x are needed. Triggers can also be activated because of time or date changes. For example a weekly time sheet for employees may be generated and sent to the payroll department.

- The form definition language must provide the user with the capability of creating new fields based on complex mathematical calculations. In order to guarantee that simple and complex calculations can be made a comprehensive set of operations must be supported. These should include operators for the basic arithmetic functions, a mechanism for adjusting the order of calculations, some specific functions, logic operations, and relational operations.

- There is a perceptible difference between automatic and virtual fields. Automatic fields are data items that are provided by the system e.g., time, date, etc. Virtual fields are calculated by a user supplied algorithm. The FDL constructs must provide for both of these types of fields.

In summary, the FDL has to provide a reasonable interaction with the user. The ability to create additional information and still provide flexibility is essential.

3.4 Input/Output Requirements

The input to the Form Definition Language is the actual "form definition" created by the designer of the form. The form definition will eventually produce or output a C-Language program that can be compiled and executed by the Forms Processor Interface. There are several functions available on the UNIX operating system that will assist in this process. They are the Lexical Analyzer (Lex) and Yet Another Compiler Compiler (Yacc).

4. Chapter 4: Form Definition Language

4.1 Design Strategy

The target implementation system for IDOMS is the UNIX operating system running on a VAX Processor available at Kansas State University. The design is intended to incorporate as many available functions from the UNIX environment as possible. This strategy of utilizing existing functions is to basically maintain portability of the final implementation to other UNIX-based systems and to promote continuity among the individual pieces that comprise this project. The implementation of the FDL has utilized a general top-down development approach using no formalized methodology. Concentration was placed on developing a set of complete requirements prior to the planning of the design phase. The task of designing required additional research into some of the available tools on UNIX. Development of both high and low level designs were completed before the implementation was begun.

4.2 Design Overview

Each form can be thought of as a program that is executable for a particular instance of data. The Form Definition Language provides the creator of the form with the language constructs needed to create this program i.e., the form. Once the form has been created it is input to several tools

residing within the UNIX operating system, Lex and Yacc. These two tools together produce a C-Language program that then must be compiled in order to provide an "executable" program. Creation of this program for the form signals a user that he/she may create data instances for the form. Once a data instance is available it acts as input to the "executable" program.

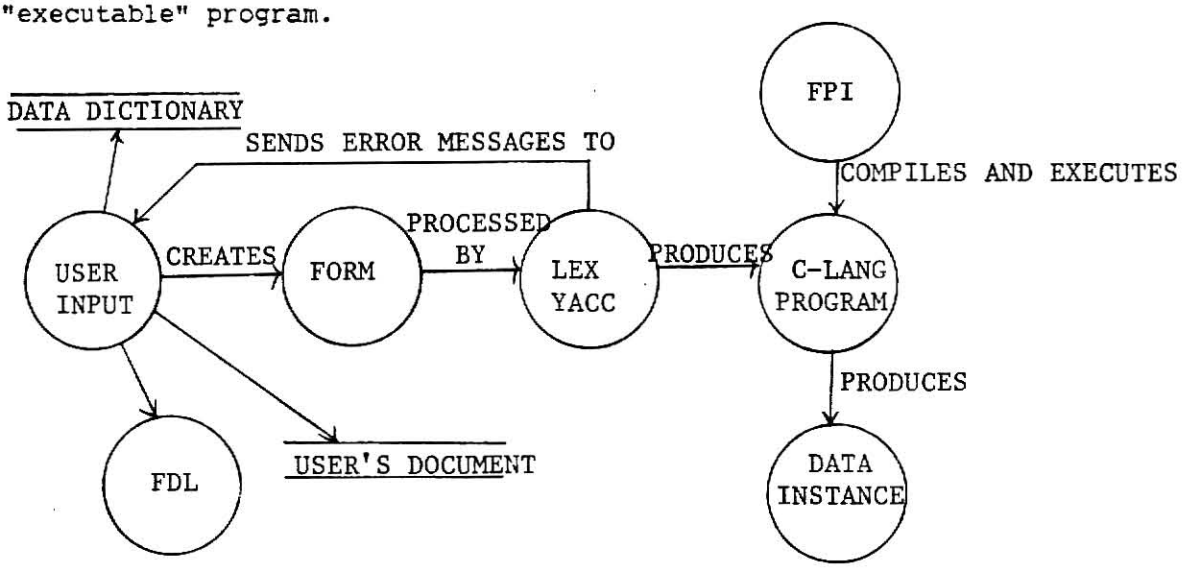

Figure 2. User's View Of The Creation Of A Form

4.3 Lexical Analyzer (Lex)

The Lexical Analyzer Generator was developed by AT & T- Bell Laboratories. It is designed for lexical processing of character input strings i.e., it recognizes regular expressions which are specified by the user and partitions the input stream into strings matching the expressions. [LES81]

Lex is not considered a complete language, it provides an additional feature to the host language which in this case is C-Language. Source refers to the user-defined regular expressions which are input into Lex to produce a generated program, yylex. The yylex program will recognize regular expressions (set of strings to be matched) in a stream and perform the specified actions as each expression is detected. Input processed through yylex should produce a partitioned stream of expressions based on the rules supplied by the user.

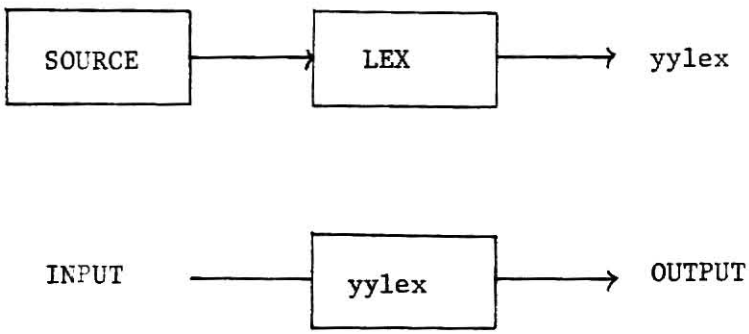

Figure 3. Lex Overview

"Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize

input tokens." [LES81] Lex and Yacc, respectively can provide a combined program to partition the input stream and a parser generator to assign structure to the resulting pieces. The interface between Yacc and Lex is somewhat simplified in that Yacc will recognize the name yylex as its input file. In fact, it assumes that the lexical analyzer is named yylex and if the user chooses not to use the Lex tool he/she is forced to name the input file yylex.

4.4 Yet Another Compiler Compiler (Yacc)

Yacc is a tool used to describe the input to a computer program. The user is responsible for specifying the structure of the input and the code to be generated as each structure is recognized.

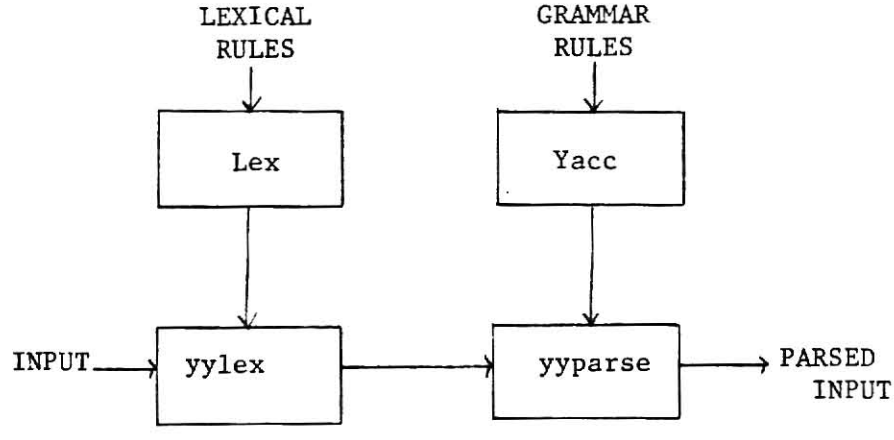

Figure 4. Lex With Yacc

Yacc produces a function to control the input process called

a parser. The parser function, yyparse, which can be user-supplied or provided by Lex calls the input routine yylex. The tokens are organized according to the input structure rules or grammar rules. When one of the rules has been recognized the user-supplied code for this rule is activated. The input to Yacc is fundamentally a set of grammar rules. [JOH81] The lexical grammar rules specified by the user must follow a rigid format. Both Lex and Yacc have a predefined specification language that must be adhered to.

4.5 Detailed Design

The design for the Form Definition Language will rely heavily on the usage of Lex and Yacc to produce the C-Language code for each form. The language constructs to be provided for the user will be English-like for cases where that application is more appropriate than a mathematical-oriented language.

```
$OrderTotal = $Quantity * $Cost;
```

or

```
$OrderTotal equals $Quantity multiplied by $Cost;
```

Math-oriented constructs will predominate within those provided for the calculation area.

The design of the language constructs for the calculation unit surround the concept of mathematic operators. These

operators have been divided into six major classes: primary, arithmetic, relational, logical, functional, and control. The primary operators are those that basically establish the order of the operations.

```
$OrderTot = ($Quan * $Cost) + ($Discount * ($Quan + $Tax));
```

The arithmetic operators provide fundamental mathematical operations such as addition, subtraction, multiplication, and division. Relational operators are used to compare two or more objects.

```
                If $OrderTotal > $MaxOrder
        Then $OrderTotal = $OrderTotal * $Discount
                            End;
```

Logical operators provide analytic or deductive functions such as and, or, not, etc.

```
  If $OrderTotal < $Inventory And $OrderTotal < $MaxOrder
            Then $OrderTotal = $OrderTotal + 1
                            End;
                             or
   If $OrderTotal < $Inventory && $OrderTotal < $MaxOrder
            Then $OrderTotal = $OrderTotal + 1
                            End;
```

Control operators include the looping capabilities and the if-then-else clause.

```
          Dowhile $OrderQuantity < $InventoryTotal
          Loop $OrderQuantity = $OrderQuantity + 1
                            End;
```

Finally, the functional operators provide the capability of calculating specific kinds of mathematical actions: square root, minimum, and maximum are typical examples. These are listed with more detail in Appendix 4.

As mentioned in the requirements virtual and automatic fields are defined to be separate types of fields in so far as their actual definitions. In reality the constructs required to support them are basically equivalent. Automatic field calculation will necessitate no additional language constructs outside of those required for the virtual fields.

Identification of the operators and the language constructs attacks the "how to calculate", the next step to pursue is the "how to manage changes" to these calculated fields. As the data instance is routed throughout the network, users have the opportunity to add data to or modify the existing data contained within that instance. The calculation of some or all of the fields that appear on that form may potentially be dependent upon the new or modified data. In order to assure that the data contained within that instance is consistent with the new data, each time an addition or modification of a field is entered on the form all of the calculable data items are regenerated and the new items replace the old items in the data instance.

This design plan handles data changes that occur within one form or what is termed intraform calculation of generated fields. In order to provide for dependencies among other forms or interform calculations, separate files must be set up for each type of form. Further this means that the calculation units for each of these forms must exist in independent files so that field generation can exist in as small a unit as possible. A mapping back of dependencies (form links) among forms is required. This mapping is not planned for the immediate implementation and will be regarded as an enhancement. In order to provide for this future development the calculation of individual files for each form will be incorporated as part of the implementation.

5. Chapter 5: Implementation

5.1 Implementation Details

The Form Definition Language constructs were designed and developed on the UNIX operating system available at Kansas State University. Because the implementation utilizes two separate tools, Lex (Lexical Analyzer) and Yacc (Yet Another Compiler Compiler) available through UNIX the FDL should basically be portable to other UNIX-based applications.

The FDL consists of two separate files or modules, form.lex and form.yacc, implemented using the language facilities provided by Lex and Yacc. Generally speaking, these two tools use the same language facility but their input formats differ slightly. The general format for input to Lex is:

```
{definitions}                names to be translated
%%
{rules}          table of users control decisions
%%
{user subroutines}                    included code
```

The output of Lex is a program named yylex() which is required by Yacc for its analyzer. The Lex output file is to be printed as part of the Yacc output file.

The input specification file of Yacc is formatted according to the following prescription:

```
declarations                    definition of tokens
%%
rules                                 grammar rules
```

```
%%
programs                                    included code
```

The output produced by Yacc will be a file called y.tab.c (on most systems). The routine yyerror prints a message when a syntax error is detected. This message will include the input line number and the error message itself.

5.2 Language Constructs and BNF Diagrams

The object of the Form Definition Language is to provide the user with a precise, easy to use language. The syntax of the FDL is based on a context-free grammar. A grammar is considered to be a set of rules which specify the sequence of characters within a language. A grammar includes four basic elements: terminals, nonterminals, a start symbol, and productions. Terminals or tokens are the basic symbol of which strings are comprised of. Nonterminals are symbols that represent strings. The start symbol is a designated nonterminal which denotes the language. The productions are rules that define the ways in which nonterminals may be built from other nonterminals or terminals. A context-free grammar is a grammar that contains a finite state set of rules in which productions may be applied repeatedly to expand the nonterminals in a string of nonterminals and terminals.

The best-known type of formal grammar is the Backus-Naur

form. The terms BNF and context-free grammar forms are used in the computer industry interchangeably. Appendix 5 contains the BNF diagrams for the Form Definition Language. These diagrams are intended to define and document the syntax of the FDL.

5.3 Parsing And Precedence Order

The parser produced by Yacc consists of a finite machine with a stack. It is capable of reading and remembering the next input token (look ahead of 1). It is an LALR(1) parser. [JOH81] The current state is always the one resident on the top of the stack. The machine has basically four types of actions that it may follow: shift, reduce, accept, and error. According to Johnson the parser moves according to the following rules:

1. Based on the current state, the parser decides whether it should look ahead to decide what action to take; if it does it calls yylex to obtain the next token.

2. Using the current state and the look ahead token the parser decides the next action and carries it out. (This action includes states being pushed on to the stack, popped off the stack, or processing look ahead or left alone.)

The shift action allows the next input symbol to be placed on the stack and a look ahead to occur. The reduce action

occurs when the parser recognizes a grammar rule and replaces it with a nonterminal. An accept action occurs when the parser has successfully completes the parsing. The error action signifies that a syntax error has occurred.

When a shift/reduce conflict or a reduce/reduce conflict occurs Yacc follows one of these two rules (disambiguating rules):

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (earliest in the input sequence).

Sometimes situations occur where the above rules are not sufficient. These situations occur most often in the parsing of arithmetic expressions. Yacc provides for user defined precedence levels and right and left associativities for operators. Because of the eventual use of C-Language and the desire to insure consistency, the precedence order for operators will be based on the precedence order defined for C-Language. Appendix 6 documents the precedence order for the Form Definition Language.

5.4 External Dependencies

The IDOMS Form Definition Language implementation is dependent upon the following commercial products: UNIX, YACC, LEX, and C - Language. The IDOMS implementation will be installed on a VAX 11/780 using a 4.2 BSD (Berkeley Software Distribution) version of a UNIX operating system. YACC and LEX which are also products of AT&T-Bell Laboratories will be utilized to parse the Form Definition Language.

6. Chapter 6: Test Strategy

6.1 Testing Of The FDL

In order to insure reliable development of the FDL each of the individual modules, form.lex and form.yacc must be tested separately. This testing can be done by creating a simplified form specification. This test bed should utilize as many different mathematical constructs as possible. The only examination that can be made on these two files individually is to manually verify that the expected transformations occurred and occurred in the appropriate places.

After each module has been tested individually they are to be tested in a combined fashion. The final output from Yacc is to be compiled through the C-Language compiler and executed using the sample form defined for the individual testing. The following files are to be produced when an entire form is defined:

- form.fields
- form.calculations
- form.routing
- form.display
- form.operations
- form.errorchecks

- form.accessrights

In order to complete this testing a file containing input data for each of the fields on the form must be created. This file will be referred to as form.fieldvalues. It should then be used as input to the C-Language program. After the program has executed the calculated fields must be verified to assure that the values have been computed correctly.

6.2 Overall Testing Of IDOMS

The testing of the Intelligent Data Object Management System should be done incrementally, i.e., integrating one unit at a time until the entire system has been verified. The tests should be executed against an all-inclusive set of conditions. Several examples of different forms should be used to verify the following capabilities:

- create a new form
- use a menu via a CRT screen to select an activity
- display the new form on the screen
- create a data instance for the new form using good and bad data
- verify that the error checks identify the incorrect data
- retrieve and modify a data instance
- calculate new fields
- create a routing list

- route the IDO within the local and distributed networks
- verify that the routing list and the routing history match
- send and receive the IDO
- trace the IDO

7. Chapter 7: Conclusions

7.1 Evaluation Of The Implementation

The Form Definition Language has been designed to meet the specified requirements.

Implementation of the FDL using Lex and Yacc simplifies the task enormously. Using UNIX tools that have already been developed and have proven capabilities has aided in shortening the development and planning time.

7.2 Design Alternatives

Some sophistication is scheduled to be built into the screen manager to handle complicated forms. Simultaneous development of the screen manager and the form could allow the intelligence needed for the display and access rights sections to be incorporated into the screen manager rather than the form. This would assist in keeping the form itself simplified and possibly reduce to a minimum the amount of unnecessary or repeated information.

A large portion of the form is designed to handle the field definitions. The form could be simplified if this effort were moved to the data dictionary. It would require additional, simultaneous development but would encourage the use of consistent data definitions because the forms

designer could rely on previously existing data fields.

7.3 Recommended Improvements

In order to make effective use of an Intelligent Data Object the following enhancements are suggested:

1. There are currently no plans to utilize an Office Specification Language, however, this potentially could be used to drive the data dictionary along with the FDL. It would also be an excellent source of documentation of the activities occurring within the office environment.

2. The use of "scripts" to define activities required to take place during the processing of the IDO would assist in the automation of the decision making.

3. Changes in the form field values may cause rippling effects among other forms. A means to mechanize the automatic processing of dependent forms should be implemented. This should include time and modification triggers.

4. Modify the IDO to include not just the routing of paper-like forms but graphics and voice representations.

7.4 Acknowledgements

I would like to extend my thanks to Dr. Elizabeth A. Unger who served as the Major Professor for this implementation project and report. The countless hours she has spent in discussions regarding the concept of an IDO during the preparation interval(from the fall of 1984 to the spring of 1985) and the summer session are deeply appreciated. Dr. Unger provided the original concept of the IDO, the division of the project, and the initial list of important references.

I would also like to thank the other members of the committee, Dr. Virgil E. Wallentine and Dr. Richard A. McBride for their interest in the project and for reviewing the document.

Finally, I would like to extend a thank-you to the other members of this project for the numerous discussions and ideas shared during the evolution and development of this project.

8. Appendix 1: Field Classifications

Personalized	- filled in automatically from the user profile and system variables
Required	- must be entered by the user
Unchangeable	- entry is optional but cannot be changed
Unrestricted	- entry is optional and can be changed
Automatic	- filled in by the system
Virtual	- filled in automatically according to some computation algorithm specified by the user
Tag	- value of the field determines the selection of or value of other fields
Variant	- the filling of these fields is prevented until the corresponding tag field is filled in (only one variant is selected per tag field)
Dependent	- these fields can be filled with data that satisfy some constraint
Ordered	- can be filled only after other fields have been filled
Lock	- field which disallows any modification to certain other fields within this form
Conditional	- data is to be filled in by the user on the condition that it is not available in the database or to be changed
Invisible	- this field will be invisible to users who do not have access rights to read or update the field
Variable Length	- the amount of text to be filled in is unlimited
Repeated	- number of instances is unlimited

[Note: Gehani does not distinguish between automatic fields and virtual fields.]

9. Appendix 2: Sample Form

Equipment Order Form

Date:_____

Manufacturer:_____
Street Address:_____
City,State,Zip:_____

Item No.	Item Name	Quantity	Cost	Subtotal
1.				
2.				
3.				
4.				
5.				
6.				
7.				
8.				
9.				
10.				

Page Total:_____
Sales Tax:_____
Order Total:_____

Date Needed:_____

Ordered By:_____ Date:_____
Extension:_____

Signature:_____ Date:_____

Approval:_____ Date:_____
Title:_____
Approval:_____ Date:_____
Title:_____

10. Appendix 3: Form Definition Example

```
form EquipmentOrderForm is

     Fields
          $Date1:  char(12); automatic
          $Manufacturer: char(45)
          $Address1:  char(45)
          $CityStateZip: char(45)
          $ItemNo1:  real(25)
          $ItemNo2:  real(25)
          $ItemNo3:  real(25)
          $ItemNo4:  real(25)
          $ItemNo5:  real(25)
          $ItemNo6:  real(25)
          $ItemNo7:  real(25)
          $ItemNo8:  real(25)
          $ItemNo9:  real(25)
          $ItemNo10: real(25)
          $ItemName1:  char(15)
          $ItemName2:  char(15)
          $ItemName3:  char(15)
          $ItemName4:  char(15)
          $ItemName5:  char(15)
          $ItemName6:  char(15)
          $ItemName7:  char(15)
          $ItemName8:  char(15)
          $ItemName9:  char(15)
          $ItemName10: char(15)
          $Quantity1:  real(6): initial 0
          $Quantity2:  real(6): initial 0
          $Quantity3:  real(6): initial 0
          $Quantity4:  real(6): initial 0
          $Quantity5:  real(6): initial 0
          $Quantity6:  real(6): initial 0
          $Quantity7:  real(6): initial 0
          $Quantity8:  real(6): initial 0
          $Quantity9:  real(6): initial 0
          $Quantity10:  real(6): initial 0
          $Cost1:  real(10): initial 0.00
          $Cost2:  real(10): initial 0.00
          $Cost3:  real(10): initial 0.00
          $Cost4:  real(10): initial 0.00
          $Cost5:  real(10): initial 0.00
          $Cost6:  real(10): initial 0.00
          $Cost7:  real(10): initial 0.00
          $Cost8:  real(10): initial 0.00
          $Cost9:  real(10): initial 0.00
          $Cost10: real(10): initial 0.00
```

```
$Subtotal1:  real(10):  virtual; initial 0.00
$Subtotal2:  real(10):  virtual; initial 0.00
$Subtotal3:  real(10):  virtual; initial 0.00
$Subtotal4:  real(10):  virtual; initial 0.00
$Subtotal5:  real(10):  virtual; initial 0.00
$Subtotal6:  real(10):  virtual; initial 0.00
$Subtotal7:  real(10):  virtual; initial 0.00
$Subtotal8:  real(10):  virtual; initial 0.00
$Subtotal9:  real(10):  virtual; initial 0.00
$Subtotal10: real(10):  virtual; initial 0.00
$PageTotal:  real(10):  virtual
$SalesTax:  real(7):  virtual
$OrderTotal:  real(10):  virtual
$DateNeeded:  char(12)
$OrderedBy:  char(25)
$Extension:  int(4)
$Date2:  char(12):  automatic
$Signature: char(25)
$Date3:  char(12):  automatic
$Approval1:  char(25)
$Date4:  char(12)  automatic
$Title1:  char(25):  virtual
$Approval2:  char(25):  tag
     case of $Approval2
     if $OrderTotal > 5000
     end
$Date5:  char(12)  automatic
$Title2:  char(25):  virtual: tag
     case of $Title2
     if $OrderTotal > 5000
     end


$Subtotal1 after $ItemNo1, $Quantity1, $Cost1
$Subtotal2 after $ItemNo2, $Quantity2, $Cost2
$Subtotal3 after $ItemNo3, $Quantity3, $Cost3
$Subtotal4 after $ItemNo4, $Quantity4, $Cost4
$Subtotal5 after $ItemNo5, $Quantity5, $Cost5
$Subtotal6 after $ItemNo6, $Quantity6, $Cost6
$Subtotal7 after $ItemNo7, $Quantity7, $Cost7
$Subtotal8 after $ItemNo8, $Quantity8, $Cost8
$Subtotal9 after $ItemNo9, $Quantity9, $Cost9
$Subtotal10 after $ItemNo10, $Quantity10, $Cost10
$PageTotal after $Subtotal1
$PageTotal after $Subtotal2
$PageTotal after $Subtotal3
$PageTotal after $Subtotal4
$PageTotal after $Subtotal5
$PageTotal after $Subtotal6
$PageTotal after $Subtotal7
$PageTotal after $Subtotal8
```

```
$PageTotal after $Subtotal9
$PageTotal after $Subtotal10
$SalesTax after $PageTotal
$OrderTotal after $SalesTax, $PageTotal
$Approval1 after $Signature lock all above
$Approval2 after $Signature, $Approval1 lock all above
$OrderedBy after $Signature, $Approval1, $Approval2
$Title1 after $Approval1
$Title2 after $Approval2
$Date3 after $Signature
$Date4 after $Approval1
$Date5 after $Approval2
$Date2 after $OrderedBy
```

Display

Equipment Order Form

Date: $Date

Manufacturer: $Manufacturer

Street Address: $Address

City,State,Zip: $CityStateZip

Item No.	Item Name	Quantity	Cost	Subtotal
1.$ItemNo1	$ItemName1	$Quantity1	$Cost1	$Subtotal1
2.$ItemNo2	$ItemName2	$Quantity2	$Cost2	$Subtotal2
3.$ItemNo3	$ItemName3	$Quantity3	$Cost3	$Subtotal3
4.$ItemNo4	$ItemName4	$Quantity4	$Cost4	$Subtotal4
5.$ItemNo5	$ItemName5	$Quantity5	$Cost5	$Subtotal5
6.$ItemNo6	$ItemName6	$Quantity6	$Cost6	$Subtotal6
7.$ItemNo7	$ItemName7	$Quantity7	$Cost7	$Subtotal7
8.$ItemNo8	$ItemName8	$Quantity8	$Cost8	$Subtotal8
9.$ItemNo9	$ItemName9	$Quantity9	$Cost9	$Subtotal9
10. $ItemNo10	$ItemName10	$Quantity10	$Cost10	$Subtotal10

Page Total: $PageTotal

```
                                                  ----------
                                    Sales Tax: $SalesTax
                                                  ----------
                                  Order Total: $OrderTotal
                                                  ----------
                                  Date Needed: $DateNeeded
                                                  ----------

Ordered By: $OrderedBy                  Date: $Date2
           ------------------                ----------
Extension: $Extension
           ----------

Signature: $Signature                   Date: $Date3
           ------------------                ----------
Approval: $Approval1                    Date: $Date4
          --------------------              ----------
          Title: $Title1

Approval: $Approval2                    Date: $Date5
          --------------------              ----------
          Title: $Title2
```

Operations

create
modify
copy
delete
read
mail

Access Rights

Fields

level = technical update all except $Approval1, $Approval2
level = supervisor update $Approval1
level = manager update $Approval2

Operations

none

Display

none

Error Checks

Intraform

If $Quantity1 <> blanks
then $Quantity1, $Cost1 = positive
If $Quantity2 <> blanks
then $Quantity2, $Cost2 = positive
If $Quantity3 <> blanks
then $Quantity3, $Cost3 = positive

```
        If $Quantity4 <> blanks
          then $Quantity4, $Cost4 = positive
        If $Quantity5 <> blanks
          then $Quantity5, $Cost5 = positive
        If $Quantity6 <> blanks
          then $Quantity6, $Cost6 = positive
        If $Quantity7 <> blanks
          then $Quantity7, $Cost7 = positive
        If $Quantity8 <> blanks
          then $Quantity8, $Cost8 = positive
        If $Quantity9 <> blanks
          then $Quantity9, $Cost9 = positive
        If $Quantity10 <> blanks
          then $Quantity10, $Cost10 = positive
        $Subtotal1 >= 0
        $Subtotal2 >= 0
        $Subtotal3 >= 0
        $Subtotal4 >= 0
        $Subtotal5 >= 0
        $Subtotal6 >= 0
        $Subtotal7 >= 0
        $Subtotal8 >= 0
        $Subtotal9 >= 0
        $Subtotal10 >= 0
        $PageTotal > 0
        $SalesTax > 0
        $SalesTax <= "2000.00"
        $OrderTotal <= "1000000.00"
        $Signature <> blanks
    Interform
        none

Calculations
    $Subtotal1 = $Quantity1 * $Cost1
    $Subtotal2 = $Quantity2 * $Cost2
    $Subtotal3 = $Quantity3 * $Cost3
    $Subtotal4 = $Quantity4 * $Cost4
    $Subtotal5 = $Quantity5 * $Cost5
    $Subtotal6 = $Quantity6 * $Cost6
    $Subtotal7 = $Quantity7 * $Cost7
    $Subtotal8 = $Quantity8 * $Cost8
    $Subtotal9 = $Quantity9 * $Cost9
    $Subtotal10 = $Quantity10 * $Cost10
    $PageTotal = $Subtotal1 + ... + $SubTotal10
    $SalesTax = $PageTotal * $IllTaxRate
    $OrderTotal = $PageTotal + $SalesTax
    If $Date1 = blanks
       then $Date1 = $SystemDate
    If $OrderedBy <> blanks and $Date2 = blanks
       then $Date2 = $SystemDate
    If $Signature <> blanks and $Date3 = blanks
```

```
            then $Date3 = $SystemDate
         If $Approval1 <> blanks and $Date4 = blanks
            then $Date4 = $SystemDate
         If $Approval2 <> blanks and $Date5 = blanks
            then $Date5 = $SystemDate

   Routing
         If $OrderTotal >= "5000.00"
            then route to $Supervisor, $Manager, $PurchasingClerk
         If $OrderTotal < "5000.00"
            then route to $Supervisor, $PurchasingClerk

end EquipmentOrderForm.
```

[Note: Field names are preceded by a "$". Not all of the fields are defined within this form. Several are assumed to be global or system defined.]

11. Appendix 4: Required Operators For The FDL

Type	Operators	Description
Primary		
	(	All of these operators
	)	direct the order of
	[	operations
	]	
	.	
Arithmetic		
	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
	Mod	Modulo
Relational	>	Greater than
	>=	Greater than or equal to
	<	Less than
	<=	Less than or equal to
	==	Equal to
	<> or !=	Not equal to
Logical		
	&	And
	|	Or
	Not	Logical negation
	-	unary minus
	=	assignment
Functions		
	%	Percent
	Count	Count of
	Max	Maximum value
	Min	Minimum value
	Sum	Sum of
	Avg	Average of
	Sqrt	Square Root
Control		
	If...Else	If statements
	Dowhile	Loop function
	Repeat until	Loop function

12. Appendix 5: Backus-Naur Form Diagram

```
        ::=     means   "is defined as"
        |       means       "or"
        <       >         means       "category       name"
```

```
<form> ::= form <identifier> is
           <field-statements>
           <display-statements>
           <operation-statements>
           <access-statements>
           <error-statements>
           <route-statements>
           <calculation-statements>
           end <identifier>;

<field-statements> ::= field <field-list>;

<field-list> ::= <field>
               | <field> <field-list>
               | <field> after <field-list>;

<display-statements> ::= display <display-list>;

<display-list> ::= <identifier> : $<identifier>
                 | <spacing> <identifier> : $<identifier>
                 | <spacing> <identifier> : $<identifier>
                   <display-list>;

<operation-statements> ::= operations <operations-list>;

<operation-list> ::= <operation> |  <operation>  <operation-
list>;

<access-statements> ::= access rights <access-list>;

<access-list> ::= fields <access-fields>
                  operations <access-operations>
                  display <access-display>;

<access-fields> ::= level = <level> <operation-list>
                    $<identifier-list>
                  | level = <level> <operation-list>
                    $<identifier-list> <access-fields>
                  | none;

<access-operations> ::= level = <level> <operation-list>
```

```
                    | level = <level> <operation-list>
<access-operations>
                    | none;

<access-display> ::= $<identifier> level <level>
                   | $<identifier> level <level> <access-
display>
                   | none;

<error-statements> ::= error checks <error-list>;

<error-list> ::= intraform <statement-sequence>
                 interform <statement-sequence>;

<calculation-statements> ::= calculations <statement-
sequence>;

<route-statements> ::= routing <statement-sequence>;

<statement-sequence> ::= <statement>
                       | <statement> <statement-sequence>
                       | none;

<statement> ::= <assignment-statement>
              | <if-statement>
              | <loop-statement>
              | <compound-statement>
              | <input-statement>
              | <error-statement>
              | <routing-statement>
              | <output-statement>;

<assignment-statement> ::= $<identifier> = <expression>;

<if-statement> ::= if <expression> then
                      <statement-sequence>
                   end
                 | if <expression> then
                      <statement-sequence>
                   else
                      <statement-sequence>
                   end;

<loop-statement> ::= dowhile <expression>
                       loop <statement-sequence>
                     end
                   | repeat
                     loop <statement-sequence>
                     until <expression>
                     end
                   | for <expression> , <expression> ,
```

```
<expression>
                        <statement-sequence>;

<input-statement> ::= get $<identifier-list>;

<output-statement> ::= write $<identifier-list>;

<error-statement> ::=  write  error  <string>  $<identifier-
list>;

<routing-statement> ::= route to $<identifier-list>;

<compound-statement> ::= begin <statement-sequence>
                         end;

<expression> ::= <factor>
               | <expression> <relational-operator> <factor>
               |     <expression>     <multiplying-operator>
<factor>;
               |     <expression>      <arithmetic-operator>
<factor>;

<factor> ::= <identifier>
           | ( <expression> )
           | Not <factor>
           | [ <expression> ]
           | [ <expression> .. <expression> ]
           | [ <factor> , <factor> ]
           | <function> <expression>;

<operand> ::= <integer> | <identifier> | ( <expression> );

<logical-operator> ::= & | | | Not | - | =;

<arithmetic-operator> ::= + | - | * | / | Mod;

<relational-operator> ::= > | < | >= | <= | == | <>;

<multiplying-operator> ::= * | / | mod;

<functional-operator> ::= <integer> % of ( <operand> )
             | count ( <operand> )
             | max ( <operand> )
             | min ( <operand> )
             | sum ( <operand> )
             | avg ( <operand> )
             | sqrt ( <operand> );

<field> ::= $<identifier> <field-datatype>
          | $ <identifier> <field-datatype> initial
            <initial-value>
```

```
          | $<identifier> <field-datatype> <field-type>;

<field-type> ::= automatic | required | unchangeable | tag
                | virtual | ordered | lock | conditional
                | invisible | variable length | repeated;

<field-datatype> ::= integer| character | real
                   | binary | constant;

<initial-value> ::= <integer> | <real> | <character> |
<boolean>;

<identifier-list> ::= <identifier> | <identifier>
                      <identifier-list>;

<identifier> ::= <letter> | <letter> <digit>
                | <letter> <identifier>
                | <identifier> _ <identifier>;

<string> ::= <letter> | <letter> <string> | <space>
<string>;

<spacing> ::= <space> | <space> <spacing>;

<space> ::=

<boolean> ::= <true> | <false>;

<character> ::= <letter> | <letter> <character>;

<integer> ::= <digit> | <digit> <integer>;

<real> ::= <integer> | .<integer> | <integer>.<integer>;

<operation> ::= create | modify | copy | delete | read |
mail;

<level> ::= creator | technical | clerical | supervisory
          | managerial;

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L
           | M | N | O | P | Q | R | S | T | U | V | W | X
           | Y | Z;

<true> ::= 1;

<false> ::= 0;
```

13. Appendix 6: Operator Precedence Order

	Precedence Order	Operators
Highest	1	. ( ) [ ]
	2	- Not
	3	/ % * Mod
	4	+ -
	5	> < >= <= == =
	6	&
	7	|
	8	,

14. Bibliography

[BYR82] Byrd, Roy J. and Smith, Stephen E. and deJong, S. Peter, "An Actor-Based Programming System", ACM 0-89791-075-3/82/006/0067,1982

[CHA76] Chamberlin, and D.D. Astrahan, M.M. and Eswarah, K.P. and Griffiths, P.P. and Lorie, R.A. and Mehl, J.W. and Reisner, P. and Wade, B.W., "SEQUEL 2: A Unified Approach To Data Definition Manipulation and Control", Publication Unknown, November 1976

[COO80] Cook, Carolyn L., "Streamlining Office Procedures - An Analysis Using The Information Control Net Model", National Computer Conference, 1980

[DEJ80] deJong, S.P., "The System for Business Automation (SBA): A Unified Application Development System", Information Processing 80, North Holland Publishing Co., 1980

[DIP83] DiPirro, J.E. and Ferrans, J.E. and Juszczak, C., "A Form Management System For Switching Database Administration", Proceedings IEEE International Conference On Communications, Boston, MA, June 1983, pp. A4.1.1-A4.1.6 (pp. 125-130), Vol. 1

[ELL80] Ellis, Clarence A. and Nutt, Gary J., "Office Information Systems and Computer Science", Computing Surveys, Vol. 12, No. 1, March 1980

[ELL82] Ellis, Clarence A. and Bernal, Marc, "Officetalk-D: An Experimental Office Information System", ACM 0-89791-075-3/82/006/0131, 1982

[FER82] Ferrans, J. C., "SEDL - A Language For Specifying Integrity Constraints On Office

Forms", Proceedings ACM-SIGOA Conference On Office Information Systems, Philadelphia, PA, June 21-23, 1982, pp. 123-130

[GEH82] Gehani, Narain, "The Potential Of Forms In Office Automation", IEEE Transactions On Communications, Vol COM-30, No. 1, Jan 1982, pp. 120-125

[GEH81] Gehani, Narain H., "An Electronic Form System: An Experience In Prototyping", Bell Laboratories Research Report, June 1981

[GEH83] Gehani, N.H., "High Level Form Definition In Office Information Systems", The Computer Journal, Vol. 26, No. 1, 1983

[GIB82] Gibbs, Simon J., "Office Information Models and the Representation of Office Objects", ACM 0-89791-075-3/82/006/0021, 1982

[GRI77] Gries, David and Gehani, Narain, "Some Ideas on Data Types in High-Level Languages", Communications of the ACM, June 1977, Vol. 20, No. 6

[GUT80] Guttag and Horowitz and Messer, "The Design Of Data Type Specifications", Current Trends In Programming Methodology, Vol IV, Data Structuring, Prentice Hall, 1978

[HAM80a] Hammer, Michael, "What is Automation?", Office Office Automation Conference Digest, pp. 37-49, AFIPS Press, 1980

[HAM80b] Hammer, M. and Kunin, Jay S., "Design Principles Of An Office Specification Language", Proceedings AFIPS Office Automation Conference, Mar 1980, National Computer Conference

[HEW77] Hewitt, Carl and Baker, Henry Jr., "Actors and Continuous Functionals", Library For Computer

Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139, MIT/LCS/TR-194, 1977

[HOG84] Hogg, John and Gamvroulas, Stelios, "An Active Mail System", SIGMOD Record, Vol. 14, No. 2, 1984

[JOH81] Johnson, Stephen C., "YACC - Yet Another Compiler Compiler", Bell Laboratories, Murray Hill, New Jersey, January 1981

[KON82] Konsynski, B. R. and, Bracker, L. C. and Bracker, W. E., "A Model For Specification Of Office Communications", IEEE Transactions On Communications, Vol. Com-30, No. 1, January 1982, pp. 27-36

[LAR81] Larson, James A., "A Data Manipulation Language For Electronic Forms", Proceedings COMPSAC 81, pp. 348-354, 1981

[LEB82] Lebensold, J., and Radhakrishnan, T. and Jaworski, W.M. "A Modelling Tool for Office Information Systems", 1982 ACM 0-89791- 075-3/82/006/0141

[LES81] Lesk, M.E. and Schmidt, E., "LEX - A Lexical Analyzer Generator", AT&T Bell Laboratories, Murray Hill, New Jersey, 1981

[LIS75] Liskov, B. H. and Zilles, S. N., "Specification Techniques for Data Abstractions", IEEE Transactions On Software Engineering, Vol. SE-1, No. 1, March 1975

[MCL76] McLeod,D.J., "High Level Domain Definition in a Relational Database System", IBM Research Report, San Jose, CA, January 20, 1976

[MAZ83] Mazer, Murray S. and Lochovsky, Fredrick H., Routing Specification In A Message Management

System", Proceedings of the 16th Annual Hawaii Int'l Conf. on System Sciences, 1983, Vol. 1

[MCB83] McBride, R.. and Unger, E. A., "Modeling Jobs In A A Distributed System", 1983 ACM 0-89791-123-7/83/012/0032

[SOR79] Sorenson, P.G., Wald, J.A., and Gustafson, J.C., "A Distributed Database Query System Based On A Forms Interface Processor", Proceedings of CIPS Session 79, Quebec City, June 1979

[STO75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification, Proceedings 1975 ACM-SIGMOD Workshop of the Management of Data, San Jose, CA, pp 65-78, May 1975

[TSI82a] Tsichritzis, D. and Christodoulakis, S., "Message Files", ACM 0-89791-075-3/82/006/0110, 1982

[TSI84] Tsichritzis, D., "Message Addressing Schemes", ACM Transactions on Office Information Systems, Vol. 2, No. 1, January 1984, Pages 58-77

[TSI82b] Tsichritzis, D.C. and Rabitti, F.A. and Gibbs, S. and Nierstrasz O.M. and Hogg, J., "A System For Managing Structured Messages", IEEE Trans. Commun., COM-30, Jan 1982, pp. 66-73

[TSI82c] Tsichritzis, D.C., "Form Management", Commun ACM, July 1982, pp. 453-478

[VIT81] Vittal, John, "Active Message Processing: Messages As Messengers", North-Holland Publishing Company, IFIP, 1981

[YAO84] Yao, S. Bing, Hevner, A.R., Zhongzhi, Shi, Luo, Dawei, "Form Manager: An Office Forms Management System", ACM 0734-2047/84/0700-00235, 1984

[ZIS77] Zisman, Michael D., "Representation, Specification, and Automation of Office Procedures", A Dissertation In Decision Sciences, 1977, University of Pennsylvania, Business Administration

[ZLO81] Zloof, M.M., "QBE/OBE: A Language For Office And Business Automation", IEEE Computer (May 1981), pp. 13-22

INTELLIGENT DATA OBJECT MANAGEMENT SYSTEM (IDOMS)

By

Kathy Pedersen Huml

B.A., North Central College

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Kansas State University

Manhattan, Kansas

1986

During the last decade the use of computer systems designed to assist in the handling of everyday activities occurring in an office environment has promoted the rapid development of office information systems. Applications of information systems range from the handling of a very limited set of activities, e.g., the updating of databases via structured form systems to the automation of office activities through triggering devices.

The system described in this report supports the concept of the routing, within a distributed network, of an intelligent data object (IDO), which contains the knowledge to route itself to a correct set of destinations to complete its processing. The IDO provides a form oriented structure which lends itself to a common set of office activities.

This project is being developed by a team of six people from AT & T - Technologies, Inc. The implementation uses the UNIX operating system, the database manager INGRES, LEX and YACC (a lexical analyzer and a compiler program), C - Language, and UNIX shell all of which are available at Kansas State University.