Object Management System Concepts: Supporting Integrated Office Workstation Applications

by

Stanley Benjamin Zdonik, Jr.

S.B., Massachusetts Institute of Technology (1970)
S.M., Massachusetts Institute of Technology (1980)
E.E., Massachusetts Institute of Technology (1980)

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

May 1983

Signature of Author . . . . [illegible]
Department of Electrical Engineering and Computer Science
May 13, 1983

Certified by . . . . [illegible]
Michael Hammer
Thesis Supervisor

Accepted by . . . . [illegible]
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Object Management System Concepts: Supporting Integrated Office Workstation Applications

by

Stanley B. Zdonik, Jr.

Submitted to the
Department of Electrical Engineering and Computer Science
on May 13, 1983, in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

The capabilities of a system for storing and retrieving office style objects are described in this work. Traditional file systems provide facilities for the storage and retrieval of objects that are created in user programs, but the semantics of these objects are not available to the file system. Database management systems provide a means of describing the semantics of objects using a single basic paradigm, the record. This model is inadequate for describing the richer semantics of office objects. An object management system combines the advantages of both a file system and a database management system in that it can store arbitrarily defined programming language objects and at the same time maintain a high-level description of their meaning.

This work presents a high-level model of data that can be used to describe office objects more effectively than data processing oriented models. This model (ODM) forms the basis for our object management system. It is shown how this model can be used to facilitate the creation of new office application programs. A language for describing object schemas that is based on the model is presented. The language contains constructs for conveniently describing common office modeling situations.

A prototype system that is based on ODM is described. We discuss the techniques that were used to implement this prototype. The use of some specialized data types (e.g., databases, derivatives) is shown to facilitate the construction of object management system software.

We also provide a methodology for designing object schemas that match the characteristics of the application. Also, given a textual schema, if the user requires specialized representations, there is a procedure to determine which new operation programs must be written in order to provide an object type with the semantics that is described in that schema. Users can choose to ignore this step and have the system use default representations and operation programs.

Thesis Supervisor: Michael Hammer
Title: Associate Professor of Computer Science

Acknowledgments

In 1974, while I was working for a small Cambridge firm, a very lucky thing happened; I enrolled in a seminar that was given by Prof. Michael Hammer. During the course of this seminar, he convinced me that I should apply to the MIT Graduate Program in Computer Science. I entered the program a few years later, and since that time Michael Hammer has been my advisor. He has always managed to have the confidence in me that made it possible to persevere. It is difficult to express my gratitude and appreciation for all that he has done in helping to make this thesis a reality. But, his contribution does not end there. He has taught me a great deal about what it means to be a researcher, a consultant, a teacher, and a critical thinker.

The other members of my thesis committee have also contributed to the overall quality of this work. Irene Greif, as co-director of the Office Automation Group, has helped to provide an intellectual environment in which the free exchange of ideas was possible. She was also responsible for major revisions to the content and organization of this document which have increased its overall quality. Prof. J.C.R. Licklider read several versions of this document with such interest and curiosity that he has provided me with enough inspiration to last for some time to come.

Marvin Sirbu, the group's other co-director, has constantly expressed interest in the ideas that are embodied in this work. He has an incredible capacity to make insightful comments that often spur one on to new ideas and directions.

The members of the Office Automation group have all contributed something to my experience at MIT. Dan Carnese deserves special mention for his patient debugging of my early ideas. His willingness to give his time to assist others as well as his ability to make sound technical comments makes him truly a unique individual. Sunil Sarin added insightful suggestions at many points in this research. Richard Ilson spent many hours trying to convince me of the difficulties involved in trying to describe formally the structure of a document.

Two former members of the group deserve special mention. Dennis McLeod's work on the Semantic Data Model helped to solidify my ideas about data modeling and served as a jumping off point for this research. Jay Kunin's work on an Office Specification Language helped me to understand better what offices in general and office automation in particular are about. Dennis has become a group legend and, I think, will always remain as such in my mind. Jay was an office mate and friend

who shared many of the same problems that I encountered from the beginning to the end our graduate careers.

An Visiting Scientist from Italy, Andrea Aparo, has been a member of our research group for the past year. He deserves special mention for his contribution to the overall quality of life in the group. He has infused the second floor with a renewed vitality and life. His dedication to quality in intellectual pursuits and philosophical underpinnings has been an important force for keeping this research in perspective. I think there are others in the group who would have to agree.

Other members of the Office Automation Group and the Programming Technology Group have helped to make my stay at MIT a happy and productive time. The particularly guilty parties include Brian Berkowitz, John Cimral, Bahram Niamir, Larry Rosenstein, Juliet Sutherland, Tim Anderson, Dave Lebling, Stu Galley, and Chris Reeve.

Several very special friends deserve mention. My dearest longtime friends, Ray and Monique Magliozzi have given generously of themselves through many difficult times. They have also provided me with an environment in which it was possible to laugh and relax. Toby Bloom, an honorary member of the second floor crew, has been a loyal friend through it all and has proved that other computer scientists can share a passion for bluegrass music.

My parents have contributed a great deal to this enterprise. Their encouragement started many years ago and has never failed. They were always there when I needed them.

Erica Zissman has been a constant source of emotional support and caring that has made it possible to endure the sometimes grueling parts of these last several years. She has also made it possible to enjoy fully their successes. Her patience and love has been very much appreciated.

Table of Contents

Chapter One: Introduction	**8**
1.1 Workstation Application Characteristics	9
1.2 Outline of the Thesis	11
Chapter Two: Object Management Systems	**13**
2.1 Comparison to Data Processing	13
2.2 Object Management Systems	16
2.3 Four Simple Scenarios	18
2.4 Brief Description of This Work	29
2.5 Relationship to Previous Work	37
Chapter Three: The Office Data Model	**48**
3.1 Purpose	48
3.2 ODM Fundamentals	50
3.3 The Office Data Model	53
3.4 Linguistic Concepts for ODM	80
Chapter Four: Examples of ODM Use	**118**
4.1 An Extended Example	118
4.2 Example Interactions	127
4.3 How to Do Some Useful Things	132
4.4 Advantages of Approach	142
Chapter Five: Architecture	**148**
5.1 System Architecture	148
5.2 Program Architecture	153
Chapter Six: The Program (ENCORE)	**155**
6.1 Object Repositories	155
6.2 Useful Object Types	160
6.3 Classes	178
6.4 Standard DBMS Services	180

6.5 Memory Use 184

Chapter Seven: Repository Design **186**

7.1 A Design Methodology 186
7.2 The Mapping Level 193

Chapter Eight: Summary and Future Directions **197**

8.1 Meeting the Goals 197
8.2 Workstation Principles 201
8.3 Future Directions 202

Appendix A.: Summary of ODM Operations **207**

Appendix B.: Schema Mapping Methodology **210**

Appendix C.: ODM Reference Manual **213**

C.1 Basic Definitions 213
C.2 Notation 216
C.3 Class Definitions 217
C.4 Aspects 218
C.5 Class Specifiers 236

Appendix D.: Formal Syntax of ODM **242**

Appendix E.: Functional Object-Retrieval Language **248**

E.1 Functions 248
E.2 Set formers 250
E.3 Predicate formers 252
E.4 ODM-Specific Functions 254

References **256**

Table of Figures

Figure 3-1: Hierarchically Structured Objects 56
Figure 3-2: A Linear Version Set 58
Figure 3-3: A Branching Version Set 59
Figure 3-4: An Earlier Snapshot of the Branching Version Set 59
Figure 3-5: Version Sets for a Hierarchical Object 98
Figure 4-1: Schema for the Class of Reports 119
Figure 4-2: Schema for the Class of Reports (cont.) 120
Figure 4-3: Schema for the Final Reports Class 121
Figure 4-4: Schema for the OAReports Class 121
Figure 4-5: Schema for the Chapters Class 122
Figure 4-6: Schema for the Paragraphs Class 123
Figure 4-7: Schema for the Class of Graphs 125
Figure 4-8: Schema for the Class of Axes 125
Figure 4-9: A Schema for the Class of Curves 126
Figure 4-10: A Schema for the Class of Points 126
Figure 4-11: A Schema for the Class of Users 126
Figure 4-12: A Simple Class Specifier 127
Figure 4-13: A Class Specifier 128
Figure 4-14: Environment for Modification Example 131
Figure 4-15: Final State of Conceptual Report 132
Figure 4-16: A View of Version Sets Provided by Subclassing 133
Figure 4-17: An Example of a Comment Form 134
Figure 4-18: Schema Fragment for Outward Appearance Example 136
Figure 4-19: Schema Fragment for Formatting Database Example 137
Figure 4-20: Schemas for the User-Profile Example 140
Figure 5-1: The Basic System Levels 149
Figure 5-2: Example Process Space 151
Figure 5-3: Object Mappings 153

Chapter One

Introduction

The work that is described in this document is a part of an overall effort in office automation. The focus of that effort is on the development of an advanced office workstation that can support office applications. The *workstation* is a vehicle for the introduction of office system technology into the office environment. In order to introduce technology into an office, we feel that it is essential to understand first the purpose of that office within the organization. Once the mission of the office is understood, it is then possible to apply technology to the automation of office functions. The form of this technology is embodied in the design of an office workstation. The design of a powerful and flexible workstation that supports an integrated working environment is the major undercurrent of this work. We believe that such a workstation cannot be designed without a basic understanding of office functions and activities.

A *workstation* is a personal computer on which an office worker can perform functions that are necessary to his business. Current commercial workstation design is a quickly growing and highly competitive field. At this point, most vendors of workstations offer systems that provide facilities for document production, electronic mail, and some rudimentary form of database management. The ultimate goal of this technology is to fit into an existing office environment to facilitate the overall mission of the office.

We believe that a workstation is characterized by the functionality that it provides to the office worker as opposed to the nature of the hardware on which it runs. The common notion that a workstation must have a specific set of hardware characteristics is misdirected. The functional requirements often place requirements on the workstation hardware, but the hardware is the wrong place to start. More importantly, the workstation must provide the kind of facilities that an office worker needs. This set of needs will vary somewhat from office to office; however, we will try to indicate some general functional characteristics that seem to be of general use. One of the most important characteristics of the software environment is that it must support highly interactive programs.

1.1 Workstation Application Characteristics

A workstation is characterized by the collection of application programs that run on it to provide the functionality that is needed by office workers. There are generic applications that are common to all workstations and applications that are specially designed for a particular working environment. The specially designed application programs often make use of the facilities provided by the generic applications. For example, a text editor is a generic application and an admissions processing system is a specific application that would be used by a college admissions office. In the admissions processing system, there would be cases in which the admissions officer would need the ability to insert a textual comment into an applicant's record. This could be accomplished by calling on the facilities of the text editor.

Some of the generic applications that are being included as a part of the workstation effort of the Office Automation Group at MIT are:

1. **A text editor/formatter.** ETUDE is a text editing system that displays to the user a formatted version of the current working document. It creates this formatted text based on knowledge about the ways in which different kinds of textual components are to be displayed. It is also

designed to be as easy to use as possible by providing users with assistance such as help messages and clearly labeled function keys.

2. **A calendar system** PCAL is a system to allow a community of users the ability to maintain personal calendars as well as common shared calendars. A given user can have access to another users calendar for the purpose of scheduling a meeting. Of course, interactions of this sort must not compromise any private information that is stored in one's calendar.

3. **A graphics system.** A facility for producing images on a high-resolution display will be a part of the workstation. This will allow users to create pictures out of shapes, lines, and text. It will also allow users to create graphs (i.e., plots) of data points that are stored in some table or database. The system will be able to pick default values for graph parameters such as the spacing of the tick marks on the axes.

4. **A table system.** The workstation will also include a subsystem for constructing tabular arrays of data easily. Commands will be available for editing tables. A user will be able to move rows and columns around within the table format. The system will also have the ability to define rows or columns that are functionally dependent on some other row(s) or column(s). This is similar to the popular software package, VISICALC.

All of the applications systems that are described above are concerned with creating and modifying different types of objects. The text editor/formatter creates and modifies textual objects (i.e., documents). The calendar system is primarily concerned with updating calendars. All of these applications are fundamentally object-oriented applications in that they are concerned with the creation and update of objects. The programs that support this class of application are basically object editors. This is similar to the approach taken at the University of Washington [61]. The document editor is concerned with the creation and modification of textual objects; the graphical editor is concerned with the creation and modification of graphical objects. All the objects that are created and manipulated by the workstation applications should be stored and managed in a uniform way. The object management system provides this common view of data.

1.2 Outline of the Thesis

The remainder of this document describes a project that addresses the information management needs of an office workstation. This project has analyzed the requirements of such a system, produced a design of a system that responds to these requirements, and looked at a few novel techniques for implementing such a system.

In Chapter 2 we describe how an object management system might look. We sketch the goals of such a system and fit them into a framework of previous related work. The character of the applications that would be handled by an object management system is portrayed by four simple examples.

Chapter 3 is divided into two sections. The first is a summary of the key concepts that are embodied in our object data model, ODM, and the second talks about the linguistic structures that have have been designed to capture the features of ODM. This chapter is not intended to include the actual syntax of the language. That is deferred to Appendix A.

In Chapter 4, we discuss how our data model can be used to describe several different office object types in an extended example of the use of this language in a typical setting. The use of the object management system facilities is illustrated by showing how typical functions in the application environment can be accomplished.

Chapters 5 and 6 discuss issues that relate to object management system implementations. Chapter 5 concerns the basic architecture of our object management system. Chapter 6 describes some programming level constructs that could be used to construct the programs that are required by the object interface.

Chapter 7 is a discussion of how one would go about designing a particular object repository. It includes a methodology that would be used by designers, and states the program requirements that are imposed on the system builder by an object specification (written in the high-level language

Chapter 8 summarizes the project. It indicates how the goals that were stated in Chapter 2 have been addressed by our design. It also indicates some directions for future research. The appendices give summaries of the languages used in this work including a language reference manual for ODM.

Chapter Two

Object Management Systems

2.1 Comparison to Data Processing

We feel that office applications have characteristics that distinguish them from conventional data processing applications. We will try to understand the peculiarities of office applications better by analyzing them in distinction to the applications that have been addressed by traditional data processing systems.

When we use the term *office applications* we do not mean to imply that they are disjoint from what are calling *data processing applications.* Rather, we believe that data processing applications are a subset of office applications. The distinctions that can be drawn between these two application types serve to delineate more clearly the areas in which new systems techniques can make an impact.

2.1.1 Data Processing Applications

The field of data processing has been concerned with the construction of *data intensive* application programs. The structure of the data in these applications dominates the complexity of the processes. If we look at the applications that data processing has addressed, we observe the following general characteristics:

1. They tend to be *highly structured* in the sense that the entire process that is to be automated can be described in detail. The criteria on which the decisions are based can be specified in sufficient detail such that the decisions can be made automatically. In a payroll application, the deduction for medical insurance can be determined by the kind of coverage that the employee has elected and the plan that is providing the coverage.

2. The tasks that are being automated tend to be *repetitive.* A payroll program may run once a week doing the same work that was done last week.

3. The applications are *formal* meaning that they tend to follow a well-defined procedure with very few exceptions. The main-line processing is the rule. Almost all checks cut by the payroll program have a set of standard deductions for each employee.

4. There is a *high transaction volume.* The number of interactions with the system in a given time period is large even though the types of interaction are fairly uniform. The payroll system will cut many checks in any large corporation, while the process involved for each check is the same. Therefore, the efficiency of the system for processing large sets is an important issue.

5. The data that is used in a data processing application tends to be *uniform.* If we are using employee data to process the organization's payroll, we will make use of employee records that are all very much alike. The fields will be the same for each record, and the overall form of the values for these fields will be very much the same.

6. Data processing applications are often characterized by their need for a *large number of objects.* In order to process a report, the application code must often iterate through very large sets of objects (e.g., the set of all employees). The discipline of file design is largely concerned with data structuring tricks that make these iterations more efficient. For example, an index produces the effect of an iteration without actually having to manifest the individual elements.

7. Data processing applications tend to have a *relatively short and predictable time frame.* When a report writing application is submitted to the batch queue for overnight processing, it is clear that the job will run sometime during the night, require roughly five minutes of processor time, and will be completed by the next morning.

8. There is often a *well-defined locus of responsibility.* Normally, a single person (or a very small group) will have the responsibility of seeing that a particular job is completed.

2.1.2 Office Workstation Applications

In contrast to the conventional set of data processing applications, office applications have a somewhat different set of characteristics. As a result of these differences, we believe that a successful object management system must be different from the state of the art in database technology today. Some of these distinguishing characteristics of office applications are listed below:

1. We would characterize most office procedures as *semi-structured.* By this we mean that they are a combination of structured activity as described above interleaved with unstructured activity. An unstructured activity is one that can not be adequately described in sufficient detail to be meaningfully automated. An example of this type of application is a college admissions office. The processing of an application consists of making sure that the proper forms are sent out to the appropriate parties on time, something that can be described and, therefore, automated, and the making of decisions such as who should interview an applicant a process which depends on many subjective criteria and, therefore, cannot be automated.

2. *Occasional access to a small number of objects* is more common in an office than routine access to large numbers of objects as in the data processing environment. An admissions officer will want to see the recommendations of a particular applicant in response to some question that has come up. It is hard to see a case in which it would be appropriate to process all letters of recommendation at the same time in batch mode.

3. The applications that will be run in an office on a given day are *less predictable* than in a data processing environment. Ad hoc use of the applications software is more common than use that can be planned a priori. The admissions office typically does not know that it will need to look at three letters of reference and four grade summaries on a given day. Instead, an admissions officer will access this information only when the need arises based on some unpredictable stimuli.

4. There is a much *lower transaction volume* in an office than in a data processing environment, even though these office transactions are less uniform. There may be more transaction types used in given day, but the total volume will be lower.

5. For formatted data, there are *fewer records* and these records will be *less uniform* than in the data processing case. For non-formatted data, there will also be much less uniformity than one encounters in a data processing application.

6. An office application will occur over a *longer and less predictable time frame.* Consider the application of writing a final report. This can often take a month of effort with a deadline that might slip several times.

7. There is often a *more diffuse (i.e., distributed) locus of responsibility.* The people who are involved in getting a given job done will often include many (if not all) the individuals in the office. The people whose responsibility it is to see that the job is completed will often shift over time.

8. The applications that are run in an office environment will have to be *specified by non-computer experts.* It is not realistic to expect that customized office applications will be created by a large centralized data processing staff. They will, instead, be created by people within the office.

2.2 Object Management Systems

There is a general need in the workstation environment for a tool that can assist users in managing the large number of objects that are created in the course of daily work. We will call this type of tool an *object management system.* An object management system provides many of the facilities that are provided by a database management system. An object management system allows users (and programmers) to describe in high-level terms the behavior of an object. The system would then act on these objects in such a way as to support that behavior. For example, suppose all copies of group reports that have new chapters added to them should be forwarded to the group's manager. An object management system could notice when such a change has occurred and cause the appropriate forwarding action to occur. ENCORE (Extensible and Natural Common Object REsource), the topic of this research, is such a system. It serves as a platform for the creation of other applications.

As a result of the application differences cited above, we feel that office object management systems must possess characteristics that are not present or not emphasized in database management systems of today. This does not mean that object management systems should be a different breed of system unable to support traditional data management chores. On the contrary, we believe that a successful object management system must be able to support both data processing and office applications in an integrated manner. When we speak of new feature requirements and differences between these two application areas, we do not intend to exclude data processing. Instead, we are pointing out a direction for this work and for future system development. Characteristics that would be required of an office object management system in order to fit into the office workstation environment are:

1. The object management system must be able to support the creation of *highly interactive programs.* The nature of office applications is such that the programs that implement them must interact heavily with their users as well as with other programs in the system.

2. The object management system must be able to deal with *multiple modes of data.* In the office environment, there are many different kinds of object that must be treated in a uniform manner. Some examples of object types that must be handled are documents, graphics, calendars, and tables. Although one could possibly shoe-horn these data objects into a records based system, the objects are not inherently records oriented.

3. The system should be able to deal with *non-formatted data.* By this we mean that the interpretation comes from the person using the system. In a conventional database environment, a value of $30K might be stored as a value of an attribute named *salary*, meaning that the employee's salary is 30K dollars. The interpretation of the value was provided by the name of the attribute to which it was attached. The value of a paragraph component of a document is a long text string the interpretation of this string is provided by the reader.

4. In an office environment there will be *no Database Administrator (DBA)* as there is in most large-scale databases of today. The job of organizing, documenting, and maintaining the data must be done by the office worker with a great deal of help from the system.

5. Since the old model of applications development by a team of highly skilled programmers cannot apply in an environment of limited resources, there will be a need for more application development done by the end-user. The object management system must *support the applications development process.*

6. There will be more use of data for *operational decision making* by the office worker. There must be a convenient and flexible facility for the formulation of queries that can retrieve subsets of the many kinds of objects that will be stored on the workstation.

7. The office environment will be *inherently distributed* (decentralized). Workers at different work stations will be creating objects that fit their own model of the world. There must be a facility for coordinating their efforts such that they can share the objects that they produce.

The office object management system will be accessible by all the subsystems. Users of the workstation will also be presented with an object retrieval language interface which they can use to retrieve objects that have properties of interest. The exact form of this interface is not a part of this research. It might be very similar to the interactive query formulation advisor that was built on top of the Semantic Data Model [43]. Such a tool would give the user access to the object semantics supported by the system.

2.3 Four Simple Scenarios

In order to illustrate the kinds of interactions that one can expect from an object management system, we will present a few scenarios of use. These scenarios are intended to illustrate the different kinds of behavior that the object management system might exhibit. They are not intended to explain how the intended behavior is implemented, nor are they intended to describe what a user must do in order to achieve that behavior.

2.3.1 Document Editing Example

In this example, we will look at how an author interacts with the object management system in the process of editing a research group's progress report. The steps in such an interaction are outlined below:

- The first thing that must be done is to retrieve the version of the report that the worker last saw. This reflects the state of the document at the last meaningful point from the worker's point of view. The repository must, therefore, have kept track of which versions of an object have been seen by which users. This does not have to be recorded for all objects, but may be specified for certain types of objects (e.g., objects that several people are working on).

- The worker then inquires about what pieces of the document have changed since this version was created. The system responds with a listing of document component identifiers that correspond to the parts of the document that have been edited. These identifiers have the form *Section 3 of Chapter 2.*

- The worker then wants to know who changed these components, and, in particular, were any changed by the group leader? If the group leader made any changes, the worker is interested to see if these changes reflect any shift in the group's public research posture.

- Now that the current state of the report has been investigated, the worker decides to continue editing the section that describes his work. In order to appreciate the potential impact of further changes, the worker asks the repository what other objects are using this section or what other objects are using paragraphs from this section? The answer is that there are two paragraphs that are also being used in a paper that the worker is writing for publication in a journal. It is determined that any changes that are made to these two paragraphs in the context of the final report should also be made in the paper as long as the basic subject of the paragraphs does not change.

- It is noticed that there are several points in this section that have comments from coworkers attached to them. When the user reads the text of the section, these comments appear in the margin in italics. They are not part of the content of the document.

- The worker then spends a few hours making many editorial changes to this section, responding to the coworkers' comments. Some of the responses are direct modifications to the text of the document, while others are comments that are addressed to the author of the original comment. This illustrates the ability to attach comments to an object that are only visible to a specific user or class of users. At the close of this session, a new version of the section is created.

- The worker would like to insert a figure into the section which contains a graph that appeared in a paper from another research group. This is not allowed because the final report was specified to contain only components that were created by members of this group. He, therefore, composes a message to the group leader to ask for permission to relax this restriction.

- At this point in the session, the worker receives a message that says that the group leader is currently looking at this section. In order to get some immediate feedback, the worker sends a message to the group leader that describes the changes that he is about to make. The group leader responds positively.

- The worker has a question about the results of another research effort that was completed last year. The answer is contained in a paper that was written by Jones and Smith and delivered at the annual symposium last June. A request is, therefore, formulated to *find the paper written by Jones and Smith and given at the Annual Symposium.* The object management system processes this request and produces a reference to the correct paper. The worker reads the section of interest, includes the answer to his question in the progress report, and places the reference to the paper in the bibliography.

- This section is then saved in the repository. As a side effect, a new version of the final report is created as well, since the specification of the final report states that any changes to the constituent sections should cause a new version of the entire report to be produced.

- Copies of the new section are sent to co-workers. The list of co-workers of interest are defined in a database that is stored in the repository. This database is referenced in the specification of who should get copies of the section.

2.3.2 Planning a Seminar Series Example

In this example, a member of a research group is interested in arranging a seminar series on the subject of office automation. It will meet on a regular basis, perhaps once a week, and it will try to coordinate with other major events that are happening within the organization. The organizer has never done this before and would like to understand better how things have been done in the past.

- The worker must first find out what other seminar series have been planned in the recent past. This is necessary to try to insure that there is no overlap in speakers or topics. The worker asks the object management system for seminar series that have occurred in his lab within the last year. The system responds by saying that there were two, one on complexity theory and an other on applications programming. The series on applications programming sounds potentially relevant, therefore, the worker asks for the list of speakers and their topics for that series. The worker decides not to include any of these speakers in the proposed series.

- The worker then needs to know what research projects are going on in the organization that have anything to do with office automation. They would be good sources of speakers. A query is formulated, and the system responds with a list of people whose project has the keyword *office automation* associated with it. This would be a query to the projects database.

- The worker would, then, like to find all people who have sent us a paper or a memo on office automation and who are located in New England.

This query would be processed by looking in the paper file database and the author database for papers whose authors are in New England. This is very similar to forming a relational join of two relations. The system also holds the complete text of some papers that were either written by people in the lab or were sent electronically to the group. For these papers, the system looks for candidates by asking for papers that contain the term *office automation* in it contents. This is an example of members of a class (i.e., Papers) being treated in two different ways, one way for papers whose text is on line and another way for papers that are not.

- The worker remembers seeing a paper recently that had a graph of office workstation sales over time on the first page. He submits a request to the system for any paper fitting this description, and the system responds with the paper that he was thinking of. The author of this paper would make an excellent speaker for this series.

- From the above gathered information, the worker selects a list of ten possible speakers. A form is prepared and sent to each of the attendees asking them to select the five speakers that they would most like to hear. The responses from these forms are automatically collected in a summary document containing the names of the respondents choices.

- In order to schedule this series, the worker asks the repository what other seminar series are planned at present and when do they meet? The list seems to indicate that Mondays at 4PM would be a good proposed time since no other seminar series meets on Monday and the history of seminars for this lab seems to show that the latter part of the day is preferred.

- The worker uses the calendar subsystem to propose this time to the other members of the research group.

2.3.3 Admissions Office Example

This example is drawn from a study [37] of the current operations at the admissions office at MIT. It concerns the processing of undergraduate applications by an office worker as new information is added to the applicants' files. We have recast the example slightly under the assumption that the same functions are being carried out on a workstation with the assistance of an object management system. This example is less ad hoc than the other scenarios presented in this chapter. It is an example of an application that is carried out periodically, although the information that is processed each time can vary considerably.

- The Admissions Office worker decides that it is time to assign reviewers to those applicants whose file is now complete. Documentation for an application arrives slowly over some period of time, causing some applicant's files to be completed before others. The Admissions Office worker performs this task at regular intervals (perhaps once a week).

- The worker invokes the application program that has been created for this purpose. It first locates the class named *Applications-in-need-of-reviewers* which has been predefined by the creator of this application. This class, a subclass of the class of *Applications*, contains exactly those applications that do not have reviewers assigned to them yet, but that are in a state that can accept new reviewers. This last condition is met if the application file is complete (i.e., it contains all necessary support documentation) or if the two initial reviewers ratings differ numerically by more than one. If there is such a discrepancy in the ratings a third reviewer is assigned.

- The object management system produces the set of applications that must be worked on, and the worker proceeds to make decisions about which faculty or staff members would be the most appropriate to review the applications. The facilities of the object management system are, again, available to assist in this decision making process.

- The worker asks the object management system for the names of all potential reviewers who have reviewed fewer than the average number of folders. This is possible because there is a class named *Reviewers* that is made up of the union of the subclass of *Faculty-members* called *Reviewing-faculty* and the Class of *Admissions-office-staff-members.*

- The worker looks through this list of potential reviewers for a good match for a given candidate. Perhaps it is desirable to have people from New York public schools reviewed by someone who has familiarity with that environment. This browsing might be accomplished by actually looking at the contents of the applicants folder and the descriptions of each potential reviewer, or it might be facilitated by producing some ad hoc queries, for example, *Get all potential reviewers who are from New York City.*

- For applicants that need an additional reviewer due to discrepancies in the first two reviewers ratings, the Admissions Office worker might want to read their comments in order to select the third reviewer. The object management system would, of course, be able to manifest the first reviewer's reports on demand.

- When the decision is made, the worker will add references to the appropriate reviewer to the candidate's application, and refile the application in the object management system. The act of filing the amended application will cause a notification to be mailed to the newly selected reviewers. They will be told of their additional responsibility and a time by which to reply. The object management system can also check to see if a reply is received on or before the assigned date, and if it is not, the system can send a reminder to the reviewer.

2.3.4 Gathering Comments Example

This example is concerned with an author who would like to gather comments from his colleagues about a paper that he is about to submit for publication. These comments are to be collected in one central place. The author would like the response summary to be collected automatically as opposed to having to copy (or extract) fields from the returned messages.

- The author of the paper first gets a list of all current members of research projects in the lab that are engaged in work on office automation. These are the people that should submit comments.

- The author sends a message to each person on this list to ask them for comments about the paper. The message is a form that each reviewer fills out and returns to the object management system. The form asks for their name, their office number, the date, and their review.

- The object management system makes the appropriate entries in a summary database from the fields in the returned forms. The fields of interest in the form become individual objects in the object management system. These objects are shared by database object and the form object. It is important to note that the form object does not disappear after it is returned to the system. It is available for reference by the reviewer or anyone else who has access rights for it. If the reviewer retrieves it and changes a field, the change will be automatically reflected in the database.

A high-level description of the objects in this example, comment forms and comment databases, is provided in Section 4-17 starting on page 134. The details of this language will be explained latter.

2.3.5 Object Management System Features

The examples that are presented above suggest some of the capabilities that a user would find useful from an object management system. These features are not adequately supported in most modern data management systems. Some of the specific features from these scenarios that distinguish object management systems from conventional information management systems are:

1. This data model is object-oriented as opposed to records-oriented. In a conventional database system, the basic data structure is a record with fixed structure and fixed length fields. This is adequate in an environment in which the data structures are intended to capture short descriptions of various attributes of objects in the real-world. In the office environment, however, we are interested in dealing with classes of objects that are actually stored entirely in the machine. A document stored in an office database is not a description of some real-world entity, but, rather, is an entity in its own right.

   Here, we would like to support objects which can have variable structure. For example, a document might or might not have sections for some of its chapters. The data model should be able to express this directly as opposed to having a set of empty section fields for each chapter. An object can be built out of other objects that can be of arbitrary size and structure. In the document editing example, the worker was able to ask questions about what pieces of the document have changed since he last saw it. This requires that the system have some model of what the pieces of the document are and how they can be located.

2. The objects should allow *sharing* of components. It often happens that part of one office object (e.g. a document) is another object that has existence of its own (e.g. a graph). In fact, the graph might be maintained by a different person than the worker who is creating and maintaining the document. It would be convenient to have any change to the graph object be automatically reflected in the document object. This can be accomplished by actual sharing of the common object (i.e. the graph) in the object management store.

   In the *gathering comments example*, the individual comment forms and the comment database share the object which is the text of the comment. Changing the text of a comment via a comment form will cause the corresponding entry in the comment database to change.

3. The office environment is characterized by continual modification of the objects in the database. It is essential that there be a way to keep track of the history of a given object. In a conventional file system (e.g. TOPS-20), the entire object that is being modified is copied to create a new version. In an object management system, *version control* should be done at the component level. That is, when a component of a structured object is changed, only that component needs to be copied as long as all objects that contain it as a component are aware of the change.

   Often, an object will be modified by one or many users in several different ways. These different candidates for the next version of an evolving object represent a set of *alternatives* that are all under consideration at the same time. They might be different alternatives to a paper that is being co-authored by several people, or they might be different versions of the same graph that were all produced by a single worker. In both cases, the new set of objects are all derived from the same original object. This is a kind of hierarchical history that is different from the linear form of version history. Once a user has split a version history into several *alternative* paths, it is often necessary to merge these paths at some point in the future. An object management system should give users the tools for expressing this type of object relationship.

In the document editing example, the changes that were made by the office worker were stored as new versions of their respective former versions. A new version of the final report object was also generated. Each of these new objects had the appropriate information (the person who made the changes, the date the changes were made, etc.) stored with it.

4. The *outward appearance* of an object is an object in its own right. It has structure that is different from the structure of the object on which it is based (i.e. the logical object). The outward appearance object (i.e. the physical object) that corresponds to a document object would consist of components such as pages, columns, lines, and characters. There may be several outward appearance objects for a given logical object, one for each output device or format. The outward appearance could provide information to support queries based on visual cues. The user might like to see all reports that have been produced in the last month that had a graph in the upper right-hand corner of one of the first two pages. This is a search through a set of outward appearance objects. A similar kind of request was issued in the above seminar series example.

5. The office data model should provide an *easy to alter schema definition.* A user who is creating a report might want to create an appendix that has not been defined as a legitimate component for this document type. There should be a facility for defining alternative conceptualizations for an object type.

6. There should be a mechanism for specifying a schema for an individual object as well as for an object type. Conventional database systems view statement types to be uniform across all instances of a given type. We feel that there might be a great deal of information about the meaning of an object that is common to all objects of a given type, but that there is also a need to be able to say things about an object that is true for a particular object of that type. For example, a report always has chapters as components, but the 1981 group progress report might have an extra component that is the list of talks given in the previous year by group members.

7. It is often useful to be able to create *aggregations of arbitrary object types* in the same way that one would do with a file folder. The members of such a collection should not have to be of a common type. Each applicant's folder in the Admissions Office Scenario was an example of this type of object grouping. The folders contained letters of recommendation, application forms, and reviewers' reports.

8. One problem with unformatted data is that the interpretation of that information is not captured in the system. An approach that seems to have some utility in trying to capture some interpretation of the unformatted data is to allow users to attach descriptive material to any object. The ability to associate descriptive properties with an object is a useful facility for all object types.

9. An area that seems to place many requirements on the data model is the support of *cooperative processing.* By this we mean the coordination of a single task by many workers, possibly centering around a set of shared objects. An example of this type of activity would be joint document production. Some of the facilities that would be useful for this kind of activity are given below:

   a. There should be a facility to support *editorial control* over the object. That is, each person involved should be able to include comments that are analogous to "red-marks" that one might write in the margin of a document. There must also be a distinguishable way to include suggestions for alternative forms of the original material. This could appear as paraphrasings for existing sentences or sentence fragments of the document. The Document Editing Example showed how a worker might use comments to drive his editing session.

   b. *Concurrency* is an important issue in an environment in which many people are working on the same object. Concurrency should be supported at the component level. If one person is changing a paragraph, others should be prohibited from changing it. Several people, however, could be attaching comments to a paragraph at the same time. The object management system should support the definition of what activities can proceed in parallel and what effects users should see when parallel activities occur. At one point in the document editing example, the worker received a message telling him that the group leader was looking at the section that was being edited. This type of notification of interactions that are in progress is an example of one of the ways that some kinds of concurrent activity should be handled.

c. In the process of joint authorship, workers would at times require their own *private copies* of an object. They could then work on this copy independently of the others and, at some time in the future, could *publish* their version back into the main stream. This process could require facilities for merging changes from each of the competing alternatives.

10. The user interface to an object management system must be simple, easy to learn, and readily applicable. The data model that is presented to the user must be free of computer-oriented complexities. The data models and their associated data manipulation languages that modern database systems are based on are too formal and unnatural.

11. The object management system should support *active objects*, objects that notice when something important has happened and that respond to that occurrence by performing a specified action. For example, when an object of a given type is modified in a particular way, a message should be sent to the custodian of that particular object instance. This facility is very useful to organize the process of alerting users of the state of the office data.

    When the admissions office worker filed the amended application in the repository, the new set of reviewers were automatically sent messages informing them of their appointment. This is an example of an active object.

2.4 Brief Description of This Work

This work is concerned with the design of a system that can store and manage the kinds of objects that one produces on an object workstation. The main theme of such a system is the intelligent management of archival objects*. We believe that some of the techniques that are being developed will be applicable outside the area of office applications* and programming environments), but the primary examples that have driven the design have been drawn from the office domain. The ultimate measure of its success will come from two sources. The first is the degree to which it proves useful to the designers of workstation programs, and the other is the degree to which it assists the user of these programs to manage the ways in which they think about and organize the objects they need.

The workstation environment is characterized by continual change. This change manifests itself by the refinement of workstation objects by an office worker. An object will be created on one day, and modified possibly by a different person on the next. The progression of changes from an initial version of an object to some later version represents an evolution from one state to another. The object management system provides a means for describing these states and the relationships between them.

In traditional database management systems, users are interested in maintaining statements about objects in the applications domain. Most database management systems model objects as a record. A record is a set of statements about an object in the real world. An employee record contains statements about a particular employee. That employee is identified by some unique identifier that is stored in the record. An example of a statement represented by a typical employee record is *Employee number 02358 has a salary of $20K.* In this example the employee is

*An archival object is one that persists from session to session.

*These techniques can be applied to general design environments. An office is an example of a design environment in which the object that is being designed might be a report or a graph; other examples include computer-aided design (CAD certainly not stored in the system. Instead, a set of statements become a shadow of the state of that employee. Most current database management technology is concerned with methods for managing records (i.e., sets of statements).

We believe that there are fundamental differences between the kinds of systems that will adequately support office applications and the records based systems used in data processing today. However, we feel that database techniques can be used quite heavily by the system as an implementation base. We also feel that much of the traditional "database viewpoint" has influenced the direction of this work.

In distinction to current database management systems, we are interested in a system that will store the actual objects of interest. In our view, the system must be able to handle the actual object, not merely a set of statements about it. For example, our system will be able to store a report object which has been created by some application program. There is no report object outside of the machine for which our object is a surrogate. All the information that is necessary to generate the report is available in the system. This is very much akin to a file system. However, the system that we are proposing will be able to express much richer object semantics than a typical file system. It will combine the object storage facilities of a file system with the data modeling facilities of a database management system. The object management system will not only store objects, but it will also contain a large amount of information about the objects that it is storing. This will allow the system to participate in the management activities associated with these objects.

In records-oriented data management systems, the identity of an object is coded into the data itself. This is done by designating some field or set of fields as the *key*. The key field(s) must be unique across the set of records of a given record type. This insures that there is one record that stands for each external object that is of interest to the application. Modifying the key field of a record modifies the correspondence between the the application object and the assertions that are represented by the rest of the record. An object management system should not require that the objects that it stores contain unique identifiers, since the system is handling the actual object. When an object is presented to the system, it generates a unique identifier that is invisible to the user. There is no way for a user to change the identity of an object by modifying its key field. An object's identity cannot change as a side effect of modifying its contents. Altering the contents of an object creates an entirely different object that has separate identity from the original object.

The definition of the objects that are being stored cannot be predicted by the object management system. New object types are often defined by the applications programs. A graphics system may define some new data types that represent different kinds of images (e.g., pie charts, bar graphs). The semantics of these new object types must be provided in a declarative fashion to the object management system by the application designer. The design of the object management system should facilitate this process.

2.4.1 Goals of an Object Management System

The purpose of this work is to develop models for describing the semantics of office objects and for specifying the ways in which they should behave. The ideas that are generated by this work will be captured in the design of a particular object management system that we shall call ENCORE. This system must be constructed in such a way as to be efficient and to have an easily understandable and modular structure. The facilities of our object management system should address the following areas:

1. **Handle different kinds of data.** An office is characterized by the diversity of information that must be managed. Workers routinely deal with reports, memos, graphs of past performance, and accounting records in

order to perform a single task or make a single decision. An object management system must provide a single mechanism for handling all the different office information resources. This mechanism must treat all objects in a uniform manner so that they can be used together effectively. Such a tool will make a big contribution toward the creation of integrated office systems.

2. **Support incremental and segmented development.** By incremental development, we mean that the collection of objects that make up the general information resource are not designed at a single time. This is not the case when one designs a database for a large corporate application. The database administrator (DBA) and his staff create an overall schema that should not change (at least for some time to come). In an office environment, we expect that new object types will be needed and designed on an ad hoc basis. Our object management system must be able to support the distribution of design over time.

   Segmented development is the design of new object object types by different people. The notion of a centralized development team is not appropriate for the office setting. An object management system must be able to allow different people to add new object types without interfering with existing objects. Here the design is distributed across people.

3. **Support Application Development.** Users of an office workstation will need to create applications systems that support the activities and needs of their specific tasks. A college admissions office will need programs that will assist them in evaluating an applicant's qualifications based on the available information.

   a. **Specification of Classes of Objects.** An object management system, much like a database management system, provides a means for describing the properties of objects such that they can be used for retrieval later. In a database management system, users are often interested in performing retrievals such as *Get all employees who make more than $30K.* In an object management system, a user might ask for *all reports that are longer than 20 pages and that have been written by people who make more than $30K.* Being able to specify the objects of interest is one of the most common parts of most office applications.

Users should have an easy to use and powerful language for describing precisely the collections of objects that are of interest to them. These descriptions should make use of a uniform interface to potentially dissimilar object types. They will be used in many different contexts within the object management system for making statements to the system about how a set of objects is to be treated. One might wish to specify that reports created by a member of the office automation group can only be read by other members of the office automation group. The specification *reports created by members of the office automation group* is an example of an interesting collection of objects.

b. **Perform object maintenance.** The object management system itself can have responsibility for performing many of the functions that might otherwise be considered part of an application program. Examples of this type of assistance include alerting a user to new changes in objects and the automatic maintenance of certain properties of objects as they are created and changed (e.g., the date of a change or the person making the change).

4. **Manage an environment characterized by change.** It is a general philosophical theme of this work that an object management system must have features that make it easier for a user to deal with change. The workstation environment is characterized by the constant production of new objects and by the evolution of the state of these objects. The system design should reflect this fact by providing specific facilities for automatically managing object change.

   a. **Side effects to change.** There must be a way to describe what actions are to be taken when some change occurs. To this end, we will need a means for describing side effects on the state of the system that should occur whenever certain operations are invoked.

   b. **History of change.** The system should be able to automatically record and deal with the history of an object, including the alternative formulations of a change that a single user or multiple users may wish to propose.

5. **Control of the use of objects.** A workstation is an environment in which many different objects are being used by many different people, all of

whom have different requirements. The management of the large collections of objects can be a formidable task. We believe that one way in which the object management system can help is by enforcing constraints on how objects are to be used.

a. **Policies Toward Objects.** Many different object types are created in an office to communicate ideas to co-workers, clients, consumers, etc. In order to communicate effectively, numerous conventions are often necessary to facilitate efficiency. An organization often wants to preserve some consistency in the form of their internal as well as external documents. An object management system should make it easy to set and abide by policies concerning objects. It should also make it easy to deviate from certain policies when this is appropriate.

b. **Object structure specification.** It is desirable to have a mechanism for describing the structure of the objects that are being managed by the object management system. A report might be described as containing a set of chapters, a set of appendices, and a bibliography. The object management system should also be able to access any of these object components. If a user stores a report, for example, the system should be able to retrieve at some later time any one of its chapters or its bibliography.

   The system would also accept descriptions of the objects that can appear as legal components of another object.. These object construction constraints would define how an object can be assembled. For example, one could specify that a group project report can only contain chapters that have been written by members of the research group.

c. **Object control.** Object control is concerned with controlling the ways in which an object is used, including who is allowed to perform actions on objects and how concurrent action is to be managed. We feel that the user should have easy to use facilities for specifying these usage constraints. The way in which these constraints are specified for a particular object can be viewed as another kind of attribute that should be available for object retrieval. That is, one might want to see all reports that can have any of their pieces modified by members of the office automation group.

One common type of constraint is the classical restriction on who has read or write access to a given object or set of objects. These security constraints are normally enforced by the system in order to guard against malicious behavior. They are also useful for that reason here; however, we feel that they are useful beyond that in the office environment. They can be used as additional information for managing objects. For example, knowledge about the office automation memos that can be released to the rest of the community can be used in making a decision about how a proposal should be written.

6. **Flexible implementation choices.** The system must not constrain the choice of implementation for system builders. The programmers who are responsible for a subsystem such as ETUDE will not accept the default implementations of the object management system as a constraint on their design. They should be able to select whatever data structures best suit their purpose and still be able to have their objects available to users of the object management system. On the other hand, the system should be able to pick reasonable default implementations for casual users.

7. **Office object semantics.** The system should be able to capture directly and naturally the kinds of information that occurs frequently in an office environment.

   a. **Object hierarchies.** Office objects are, in general, built up from other objects. A report might be a set of chapters, a set of appendices, and a bibliography. The object management system should be able to express this information and be able to incorporate it into all other aspects of its functionality.

   b. **Alternative views of an object.** There should be a way to describe the construction of different objects based on the same underlying information base. An example of a different view of an object is the outward appearance of a document. Different outward appearances can be computed based on the ultimate output device. Changing the logical document changes the outward appearance.

8. **Relationships among objects.** The object management system will store

many objects of many different types. It is essential that the system have a means for expressing arbitrary relationships among objects. A given paper might be related to some meeting object that represents a time when the authors will get together to discuss its content.

There is also a need to be able to describe certain predefined relationships for which the system has built in primitive operations. An example of such a relationship is *latest-version-of* which expresses the fact that one object is the latest version of some other object.

9. **Effective memory utilization.** Another important function that an object management system should perform is the management of the way in which large objects are read into main memory. Since the system will have access to a large amount of information about the semantics of an object, we expect that it can do a good job of retrieving those pieces of a large object that are most likely to be needed.

10. **Generalized information about object types.** There should be a way to describe general information about an object type. An example, is the formatting rules that are used by the document formatting program. This information is associated with an object type, and is probably only useful to a small number (i.e., in most cases one) of programs.

We will return to these goals in the last chapter of this document. The bulk of what follows is intended to give the reader an understanding of the approach to the problem of object management system construction taken by this work. At the end, we will show how this approach addresses these goals.

2.5 Relationship to Previous Work

Although we feel that our conceptualization of object management systems is significantly different from other work, there are some areas of current research that are related. We will describe briefly the similarities and the differences below.

2.5.1 Traditional Database Management

Traditional database management systems are those systems that are commercially available to run on most modern computer systems. They usually embody one of the major data models that take a view of data as graphs or relations. Examples of some of these systems include IMS (IBM), IDMS (Cullinane, Inc.), Total (Cincom, Inc.), System 2000 (Intel, Inc.), and Model 204 (CCA, Inc.).

Traditional database management has produced a technology of records processing. The fundamental data structure provided by a DBMS is the record. A record is a set of short, formatted, fields. This structure is adequate for most data processing applications that have been encountered to date. For the problem of supporting a payroll application, modeling an employee as a record containing his name, social security number, salary, and job title is very reasonable. Some of the characteristics of these applications that spawned this technology have been listed in a previous section. However, if the application is document production, the use of records as the fundamental data structuring tool presents serious limitations.

The records-based systems have achieved a high degree of commercial success over the last decade. The primary reasons for this success are:

1. **Improved applications development.** The most important factor in the success of modern DBMS's is the fact that they have made it much easier for applications programmers to produce and maintain applications code. By providing a high-level description of data semantics, one could write code that used more abstract notions to describe the required data operations.

2. **Data independence.** Another important reason that DBMS's have continued to grow in popularity is that the enforced use of a higher level of description created applications programs that were independent of the underlying file structures. Tuning the file structures to better accommodate database usage patterns required no change to the applications code.

3. **Better control over data.** Database systems provide the facilities to control the use of a shared data resource by multiple concurrent users. It controlled which operations could be performed on a datum by a given user. It controlled the order in which programs could operate on data items such that no unexpected results could be obtained. The DBMS contains these facilities and provides a set of guarantees about the way in which users are allowed to access the data. The application programs were not required to address these concerns.

These general success factors should be retained in object management system designs. They should be used as guidelines that can be used to judge our overall efforts.

Despite the above listed advantages, there are several reasons why a conventional DBMS is inappropriate for dealing with the kinds of objects that we encounter in an office applications environment. These disadvantages for conventional DBMS in the office environment are summarized below:

1. **Wrong level of abstraction.** Arbitrarily defined programming language objects and record structures are difficult to maintain in any sort of integrated manner. Statements that describe the properties of an object must be mixed with the components of an object. There is no easy way for the system to distinguish the attributes of an object from the pieces of the object itself. There is no way to express directly the kind of information that is required in the effective management of office objects.

2. **Inflexible data structuring facilities.** In many systems, the size of a field is restricted to some unreasonably small length. Also, the data type of the field is limited to a small set of choices (e.g., integer or string). The mechanisms for expressing data relationships is also extremely limited.

3. **No concept of the whole object.** If the object is stored in different records, one for each component, there is no way for the system to know about the extent of the object. The system would have no concept of what pieces would have to be retrieved in order to produce a whole object (e.g., a report).

4. **No simple way to restructure complex objects.** The structures that were chosen by the database designer as a representation of the application are relatively frozen. New object structures and relationships cannot be defined dynamically. In an office environment, we would like to be able to look at various pieces of an information base in many different and unpredictable ways.

2.5.2 The Semantic Data Model

The Semantic Data Model (SDM) of Hammer and McLeod [31] was designed as a more expressive alternative to the major data models that are in current use (i.e., hierarchical, network, relational, and entity-relationship). The SDM views a database as a collection of entities that are organized into classes. Entities correspond to the relevant abstract objects of the application, and classes correspond to relevant collections of these objects. Classes can be related by inter-class connections, and entities can be related by means of attributes.

One feature that distinguishes the SDM from other data models is that a given entity can belong to many classes. This is a reaction to the observation that it is often useful to describe a given object in many different ways. Another distinguishing feature is the rich facility for describing derived data. Also, the built-in constructs for describing high-level application semantics such as events is something that other data models do not address.

The direction that was started with the SDM for data processing style databases has been continued in this work for office style databases. There are many aspects of our office data model that resemble the SDM. This is not accidental. The facilities of the SDM have been extended and augmented, however, in order to encompass the type of object semantics that occur in office applications.

2.5.3 Office Information Systems

Current office systems products that are primarily designed around word processing facilities contain only rudimentary versions of information management. The documents that are produced by the word processor must be stored on some permanent medium, usually some form of hard or floppy disk. Most systems provide a means of storing each document as a single unit with some unique name. This is much the same as the file system component of current operating systems. No structure is imposed on these documents making it very difficult for a user to maintain effective control over their use.

Newer products allow the creation of hierarchical directory structures in which each user can have his own directory that is further subdivided into subdirectories, one for each topic of interest, for example. Access to an individual file is achieved by specifying a path from the root of the file directory and including the sequence of directories that are needed. Other systems provide a limited database capability here by allowing access to the directory by means of properties of the entries. An example of this is accessing the files by the "name of the creator" or by the "date last processed". The kinds of information that are available for a given set of files is very limited and is fixed by the implementation of the system.

The other capability that is provided by current office information systems is essentially a records processing tool that is usually much the same as primitive file management packages that have been available as support for current data processing applications. The system supports the creation of a file of records each of which contains a particular set of fields. The fields can contain numbers or short text strings. These systems usually supply some sort of retrieval language for selecting a subset of the records that match some retrieval predicate. These languages are usually based on the select/sort/print model of queries. That is, a set of records is selected by sequentially searching the file, the resulting set is sorted by some sort criterion, and the result is printed on some output device.

In summary, then, the information management capabilities that are included in most current office system products are simple-minded imitations of the technologies that existed in data processing systems. The documents that are produced by the word processing system are stored in the filing system and records-oriented data is stored in the database or file management system. The integration of these two types of data is very difficult. Our research in object management systems is intended to provide more powerful tools that can model office objects more directly in terms that reflect the underlying needs of office work. We are also interested in improving on the ability to integrate a large variety of different types of data.

2.5.4 PIE

The PIE system [22], developed at XEROX PARC, provides an environment for creating a network of nodes that each represent some entity of interest. The network structure indicates relationships between these entities. The original motivation for developing PIE was based on a desire to manage alternative designs for Smalltalk programs. PIE is implemented in Smalltalk and shares much of the Smalltalk design philosophy as well as techniques that are a part of knowledge representation languages like KRL [8].

PIE is primarily a programming environment. It does have a particular view of objects, but that view is not designed to support office functions. It does, however, support the process of incremental design (specifically for programs). PIE provides a mechanism for recording the history of object changes. This is achieved by defining two concepts, *contexts* and *layers*. A context is structured as a sequence of layers, and a layer is a tagged environment in which new values can be assigned to state variables or, in Smalltalk terminology, instance variables. An instance variable can, therefore, have a sequence of values that reflects the state of that variable over the course of the object's evolution. This version history is locked into a given context for an object. Several contexts cannot share the same collection of layers. In our work, a collection of object changes is an object just like any other object and can, therefore, be referenced from arbitrary places.

There is no way to control the way in which additions to a layer will propagate into other objects. For example, if we change a chapter in a report object, there is no way to conveniently cause a new report object that reflects this change to be produced.

PIE also adopts the concept of object classes from Smalltalk. It extends that notion to include such things as multiple inheritance. It's class mechanism differs from the class mechanism of our object management system in several important ways. Most importantly, the class mechanism that we will present allows classes to change their membership as the properties of their constituents changes. In PIE (and Smalltalk) once an object is created as a member of a given class, it always remains an instance of that class. Class membership in our object management system depends on current properties of objects.

2.5.5 Object Filing Systems

The area of object-oriented filing systems has been explored by several research efforts in the field of operating systems. Some of the more prominent research systems are Hydra [68], CAP [67], iMAX [48], and Swallow [52]. These systems represent somewhat different approaches, but they all are concerned with various aspects of the problem of building operating system facilities that go beyond the normal conceptualization of file systems.

HYDRA's main goal is to provide the *kernel* facilities for constructing operating systems, including object-oriented filing systems. This kernel should provide a protection mechanism for controlling access to a shared object resource. Both HYDRA and CAP are similar in that their approach to protection is via a *capability* approach. A capability is a reference to an object in the file system that carries with it a set of access rights. An access right is the privilege to apply a specific set of operations to an object of a given type. In HYDRA, capabilities can be passed to other programs by means of a kernel mediated call mechanism. The kernel checks to see if the the type of the actual parameters match the parameter types expected by the called procedure. It also checks the *rights* of the actual parameters against the rights associated with the corresponding parameter slot in the called procedure. The rights contained in the actual parameter capability must include the rights for that parameter that are specified in the called procedure. If either of these conditions is violated, the call is not allowed.

HYDRA views objects as instances of user defined abstract data types much like those that are definable in a modern programming language like CLU [39]. Beyond this, there is no prescriptive model of data imposed by the system. Our object management system presents a particular view of data that we believe fits the needs of office applications. This view of data relies on the ability to store networks of related objects in the file system but goes further in its addressing of office information needs.

Our approach to protection is also somewhat different. Rather than a capability based approach, we have adopted an approach in which all information about which users (or programs) can apply which operations to an object is contained within the object itself. This type of information is stored in a database that is associated with the object and that is accessed each time any access operation for the object is invoked. The access code for each object type, must enforce the protection constraints. We assume that there is a mechanism for dealing with the problem of authenticating the identity of a user (e.g. passwords).

2.5.6 OBE

OBE (Office Procedures by Example) [74] is a two-dimensional programming language that is designed for non-programmers who wish to specify their applications interactively. It presents a user interface that is an extension of the authors previous work on QBE (Query by Example) [71]. OBE combines the database query aspects of QBE with a simple mechanism for specifying triggers and object distribution (i.e., electronic mail). It also has a facility for specifying an object or set of objects as a view of some database query. For example, a set of form letters could be viewed as a textual template with the names and locations of each recipient extracted from a database and inserted in the right slots. The emphasis here is on user interfaces; whereas, our research centers more on capabilities of an object management system and techniques for building such a system.

This approach is very useful for the specification of simple applications. We expect that a facility such as OBE could be built on top of the kind of object management system that we are proposing, although an object management system should also support the construction of more complex applications such as a text editor or a calendar management system. It is not clear that the tabular paradigm used in OBE (and relational systems in general) is flexible enough to support completely the rich set of applications that one encounters in an office.

2.5.7 Electronic filing cabinets

Some researchers have seen the problem of office data management as replacing conventional filing mechanisms (i.e., filing cabinets) with an electronic counterpart. A typical enhancement to paper storage devices is the ability to locate a document on the basis of its containing a particular word or words. A simple variation of this is the use of user assigned keywords. This is an outgrowth of the field of information retrieval systems [58] that has grown up in the library environment. For document retrieval in a library, this technology has proven to be very useful. It often revolves around the idea of supporting a user's search for documents that are *relevant* to a user's imprecise model of his interests. Many of these systems make use of the user's previous responses and complex probabilistic models to decide which documents to show next.

We feel that this kind of ability would also be useful in an office environment. However, we also feel that it is only a small part of the picture. The information retrieval model uses a model of documents that essentially views them as long strings of tokens. A document can display certain patterns in these strings of tokens (e.g., contains a specific word), but there is no way to express any of the additional semantics that we have for office objects. A report can have a date on which it is due, and it can be made up of chapters. Each of these chapters may have its own author and a sequence of previous versions that led to the current version. This additional semantics can also be used for object retrieval and is essential for managing the tasks that must be performed with these objects.

An example of such a text retrieval system is the NDX-100 Electronic Filing Machine [62]. It is a physically separate piece of hardware that literally takes the place of a filing cabinet. One stores documents into it and retrieves them by partial text matching. It uses a surrogation indexing technique that achieves very rapid response time without devoting huge amounts of space to the storage of indices. This technique works very well for documents that are one to two pages in length. These algorithms would be useful in many text retrieval contexts, and we expect that they could be used to support such a feature, if one decided to build that feature into an object management system.

Another example of an electronic filing cabinet style system is BUSINESS [45]. This system not only mimics the standard filing cabinet paradigm, but also embodies a specific model of an entire office. This system has primitive objects such as desks, bulletin boards, waste baskets, desk calendars, and office aides. Its primary goal is to present a programming environment that is easy to use by being familiar to its users. It is not clear that the office model that is integrally a part of this system is the correct model for all offices. This system provides a particular interface to a particular set of office objects. If this proves to be a satisfactory interface, an object management system could provide a suitable platform for building it.

Chapter Three

The Office Data Model

A central focus of this work has been to develop a single object specification language for describing the kinds of objects that one is likely to encounter in an office application. This language is based on a model of data that encapsulates our theory of how office objects are used. The features of this model of data delimit the functional boundaries of our object management system.

This chapter describes our basic view of data that we will call the Office Data Model (i.e., ODM). To this end, we begin with a discussion of the philosophy upon which the design of the ODM is based. This will be followed by two main sections. The first will discuss the ODM in detail including a description of the fundamental object types and the operations that one can perform on them. The second section will contain a description of the linguistic features of the object specification language. The language supports the basic notions of the data model as well as including constructs that make the data model more convenient to use. In both sections, the basic principles of the design are discussed in some detail. The detailed syntax of the language is deferred to Appendix A, an ODM Reference Manual.

3.1 Purpose

The object management system is designed to store and manage objects of many different types in an integrated fashion. The system puts no restriction on the underlying implementation of these objects. In this regard it is much like a file system. On the other hand, most file systems do not have the ability to capture any interesting semantics about the objects that they are storing. We would like our object management system to be able to provide the semantics that are necessary to support the intelligent management of these objects. To this end, it is much like a modern database system. One of the purposes of an object management system is to marry these two technologies such that users can store objects with arbitrary representations and at the same time be able to represent high-level semantics about these objects.

An *object definition language* is a mechanism for describing the semantics of a broad class of objects in a uniform manner. Each object type is described by a specification in an *object schema.* The total collection of objects managed by the system will be called the *repository* and the collection of all object schemas for each object type in the system will be called the *repository schema.* The repository schema gives a user a uniform view of the total information resource that the system makes available to applications.

Currently, the different software packages that make up an office automation system or an office workstation produce objects of many different types. The systems for storing the objects that are created by these systems have great difficulty in providing uniform access to the total collection of objects. The primary purpose of a data model such as ODM is to create a uniform environment for the management of many different object types. Each object type is described using the constructs of ODM. A user may, then, use the facilities of the data manipulation language to access any of the objects. At the highest level, the same basic structures are used to describe both reports and graphs. Therefore, accessing reports and graphs will involve the same set of primitives.

3.2 ODM Fundamentals

Our approach to object management takes a view that considers data to be central and processes to be secondary. This is similar to the view held by modern database management systems. One of the triumphs of the modern database approach is the elevation of data to the status of an organizational resource. Data is recognized as having existence apart from any single application program that might make use of it. We feel that the same advantages can be achieved in the office environment by taking a data-oriented view of object description.

Once we have a uniform access language for the objects in our application domain, we can construct application programs that are based on these objects. These applications embody the required processes.

3.2.1 Premises

In this section, we will describe the premises that underlie this work. We believe that an understanding of these premises is essential in order to appreciate the rest of the work.

Our first premise is that office applications are fundamentally object-oriented. Office objects are required to support the completion of office tasks that are responsive to the overall function or mission of the office. The objects that are found in offices today (e.g., forms) are a particular manifestation of a system that addresses the function of the office; they are not the only such objects. These objects are only artifacts of the current way of doing things. They could easily change, and, in fact, might be expected to change if we introduce a new technology base. But whatever the current wisdom on how office work should be organized, we believe that there will be a set of objects that are needed to support the applications in a given environment.

A second premise is that office applications require integrated access to a wide variety of objects. It is not sufficient to access objects of only a single type within an application program. If we look at current office operations, it is common to see people using multiple information sources. A worker might have several pertinent reports, a few graphs of business performance, and some catalogues from parts suppliers spread out on his desk all at the same time. A fundamental part of the task at hand is to locate the appropriate information and summarize it or move it into some working document.

Many current office systems tend to be low-level and isolated. Each application program creates data files that are not easily accessible to other applications. Each application is responsible for its own data. There is no view of this data beyond the application at hand. We believe that this leads to a situation in which it is impossible to exploit the potential benefits of computerized office systems.

Our third premise is that the objects that exist in an office environment have structure. Even though objects like reports and graphs do not fit into the records-oriented mold of traditional data management systems, there is, nonetheless, some structure that can be exploited by an object management system and by applications programs in general. The design of ODM has been driven by a desire to provide a tool for capturing that structure.

The final premise is that there is much commonality among the different object types that one encounters in an office workstation environment. It is possible to use the same basic modelling mechanisms for describing objects from different applications. We do not believe that one must invent new modelling primitives in order to accommodate each new style of object that must be handled by the system.

3.2.2 Office Application Style

One of the primary goals of an object management system should be to support the creation of new application programs. In order to do this, we must understand the nature of these applications.

It is our contention that many office applications are inherently interactive. This is a direct result of the fact that these applications are not highly structured as are data processing applications. That is, it is impossible to generate a formal procedure that will perform the task. Instead, the user must intervene at various points in the processing to make decisions about what should happen next. Office applications require human judgment for their successful completion.

Does this mean that the machine cannot be of help in performing office tasks? We believe that the answer to this question is that it can. Office applications tend to be semi-structured in that there are parts of the task that can be completely described and parts that cannot. The computer can perform that part of the application that is formalizable, leaving the other parts for the user.

We feel that the basic paradigm for an office application is an alternation between these structured and unstructured parts. The computer performs some work, and then the user makes some decisions and perhaps issues some further instructions. At this point the machine performs some more work, and the cycle repeats. To the extent that the machine can present the user with exactly the information that is required to make a decision, the more helpful it is toward accomplishing the overall goal.

The primary focus for the structured parts of an application, then, is the retrieval of the objects that are most useful at that particular point in the process. Finding too many objects floods the user with unnecessary detail and obfuscates what is really going on. Insufficient information could lead the user to make an incorrect decision based on erroneous assumptions caused by the inability to see critical information sources. This observation relates to two key measures of success in the field of Information Retrieval (i.e., bibliographic search) [58]. These two measures are *recall*, the percentage of all existing relevant objects that are retrieved, and *precision*, the percentage of the objects that are retrieved that are relevant. The role of an object management system is to allow someone who is building an application to describe as precisely as possible the set of objects that are required at each potential branch in the application code. The system provides an easy to use mechanism for describing the objects of interest in terms of the rich semantics that are recorded in the object repository.

The object management system returns a set of objects to the application. The user looks through these objects and selects the ones of interest or performs some operations on a subset of the objects. For example, the user might look at the content of each object in a set of objects returned by the system and mail copies of interesting objects or sections of interesting objects to a colleague.

3.3 The Office Data Model

We will now present our model of data. This includes the basic modelling primitives and the operations that can be performed on them. The data model expresses our view of how data should be structured. It is distinct from the language that we use to describe objects. The language is treated in a later section of this chapter.

3.3.1 Simple Objects

An *object* is a package of information that is treated as a conceptual unit. It typically has some set of programming language level operations that can be performed on it. This corresponds to the CLU [40] notion of a cluster or the ADA [50] notion of a package. An object has some abstract significance to users of the system; it is something of interest to the application. It can belong to one of the following two categories:

1. **External objects.** An external object is an abstract object in the repository that describes an object in the real world. An external object is a shadow of some object that is external to the computer system. In order to achieve the mapping between the external object and the abstraction of the object that is stored in the machine, one invents a symbol (a unique identifier or key) that stands for the real object. Modern database systems deal with external objects.

2. **Internal objects.** An internal object is an object that exists within the computer system. For example, a report is created and manipulated by the document editor and stored in the repository. The report is defined by the computer system. Any printed copies are merely outward appearances of the internal report and serve as a shadow of the object that resides within the machine. An internal object stands for itself.

Although these two categories are not primitive, this distinction will manifest itself in the object modeling primitives that the object management system supports.

An object is divided into two distinct parts, its *content* and its *attributes*. The content relates to that portion of the object that gives it its identity, and the attributes describe various aspects of the object. This distinction is not drawn at a primitive level in other data management systems. We believe that it is very important in the office domain. The next few paragraphs will present some intuition about this distinction. A more detailed discussion will follow in a subsequent section.

Each major object type (e.g., documents, graphs, and tables) that is dealt with on the workstation has an editor associated with it. The editor is a program (or subsystem) that allows a user to create and modify instances of the associated type. Many of the subsystems of a workstation can be viewed as editors [61]. We can begin to get an intuitive feel for the notion of content if we begin to think of the *content* of an object as that portion of the object that is manipulated by its editor. The *content* of a document is the information that appears on the screen when one invokes the text editor (e.g., EMACS) on that document. For a document, it is the chapters, sections, and paragraphs that one normally thinks of as the content. For a graph, it might be the starting and ending values for the tick marks on the axes or the points that are included in the plot. This is only an approximation, but it begins to give us the flavor of the content.

Objects can contain other objects. The content of an object describes this containment relationship. It is common for a given object to be built out of a collection of other objects, which leads to a hierarchical view of objects. The content is made up of a set of named *components*. Each component can be a single object or a set of objects.

Objects also allow for the sharing of other objects. Two or more objects can both have some other repository object as a shared component. For example, two report objects can incorporate the same chapter object as a component of each. In this way, changes to the shared object will be reflected in the content of all objects that are sharing it.

Figure 3-1 depicts several structured objects. One of them has O1 as its root object and O2 and O3 as its direct components. O2 and O3, in turn, contain objects O4, O5, O6, and O7 as components. The lines in this diagram represent the containment relationship. That is, O1 contains O2 and O3. Each of these seven objects are distinct repository objects. Object O8 is sharing O3 with O1. Any changes to O3, O6, or O7 will be seen from O1 and O3.

```
      01    08
     /   \  /
   02     03
  / \    / \
04  05  06  07
```

Figure 3-1: Hierarchically Structured Objects

An object can also have an arbitrary number of attributes, as well as its content. An attribute makes a statement about the object. Its value has nothing to do with the essence of the object (as does the content), but rather it records some observation about it. An example of an attribute of a report is *Author-of.* This attribute has a value that is the person who has written the report. This is merely an assertion about something that is true about the report.

We make the distinction between three basic types of object. They are:

1. A **primitive object**, a sting or a number. These are the most basic data types in the system. Other objects are built out of groupings of primitive objects.

2. A **simple object**, an object whose state is not dependent on time. An operation on a simple object does not have time as a parameter. A simple object is built out of primitive objects and other simple objects. The objects that we have been describing in this section so far are simple objects.

3. A **conceptual object**, an object whose state is dependent on time. An operation on a conceptual object does have time as a parameter.

The following section will look at this later type of object in more detail.

3.3.2 Conceptual Objects

A *conceptual object* changes over time and keeps track of its previous states. It is a collection of simple objects that represents the version history of some evolving entity. This thesis is a conceptual object. Thinking of it evokes memories of countless previous versions all of which were an instantiation of the concept of this thesis. Each new version of the thesis was derived from a previous one. The collection of these versions makes up a single entity that I label with the single name, *My-thesis*. The name *My-thesis* refers to the concept of my thesis, (a concept that has had many instantiations) and this is how we arrived at the name, conceptual object.

An office environment is characterized by continual change. The process of producing objects such as reports, graphical displays, memos, and slides for a talk all involve the evolution of objects from some early state to a state that is considered to be more appropriate for meeting the worker's ultimate goals. If one is producing slides for a talk, several versions are produced, each one, perhaps, being clearer in its organization than its predecessor.

The conceptual object is a response to an application environments that are characterized by change. They provide the primitives that are needed to capture the history of objects that evolve over time. It provides a place to attach information that concerns the evolving entity as opposed to information about the individual versions.

3.3.2.1 Linear Version Sets

The simplest form of version is a linear time-history. We will call this type of version set a *linear version set*. In this case, the version set is a sequence of simple objects whose order represents the order in which these versions were created in time. Each version except the latest one has exactly one successor, and each version except the first one has exactly one predecessor. A simple linear version set is shown in figure 3-2. Each of the V_i's represents a different version object. The double dashed line represents a link from a version on its left to that version's successor on the right.

```
V1 -- V2 -- V3 -- V4 --V5
```

Figure 3-2:A Linear Version Set

This particular version set contains five versions. New versions can be added at the right-hand end only. That is, the *add-new-version* operation applied to this version set and a new primitive object, V6, will make V6 the successor of V5. The operation *latest-version* applied to any version set will always produce the right-most version. In this case, it would return V5.

3.3.2.2 Branching Version Sets

Version sets can be much more complicated than a simple linear history of versions. Often, several new versions will be proposed as *alternatives* to some one existing version of an object. They can, therefore, incorporate a notion of branching. In other words, the successor of a given object can be a set of objects. In any environment in which many people are working on the creation of the same set of objects, it is often necessary to maintain several competing versions at the same point in time. Each of the authors of a final report will have their private temporary versions of the report. At some point in the future, all competing versions could be merged into a single document that is accepted for publication. This happens after a decision is made about the nature of the final product. This decision can be made by a single individual or a group of individuals. Our system makes no judgment about how the merging decision should be made. The system should, however, be able to support whatever policy the producer(s) of the object deem appropriate.

For object management purposes, it is necessary to support all intermediate versions in their proper relationship to each other . Figure 3-3 shows an example of a version set that manifests branching behavior. We shall call any such version set a *branching version set.*

```
                          V8
                         / \
                V4 -- V6    V10
               /        \  /  \
V1 -- V2 -- V3           V9    V12
               \               /
                V5 -- V7 -- V11
```

Figure 3-3:A Branching Version Set

In this example, two versions, V4 and V5, are both derived from the same predecessor, V3. After this split, these two versions have independently evolving version histories until they are recombined into the single version, V12. V4 is followed by V6 which again splits into V8 and V9 and is recombined into V10.

As this example illustrates, there are times when an application of the *latest-version* operation on a branching version set will produce more than a single primitive object. Consider the state of the version set in figure 3-4. This is a possible previous state for the version set in the figure 3-3.

```
                          V8
                         /
                V4 -- V6
               /        \
V1 -- V2 -- V3           V9
               \
                V5 -- V7
```

Figure 3-4: An Earlier Snapshot of the Branching Version Set

An application of the *latest-version* operation for this version set will produce the set of simple objects, {V7, V8, V9}. All three of these objects are currently active candidates for the role of latest-version. In order to add a new version to this version set, one must identify which of the three current alternatives it is a new version of. For example, if VS is the name of the version set in figure 3-4, then *Add-new-version (VS, V10, V7)* would add V10 as the successor of V7 (as in figure 3-3). In the simple linear case, the third argument to *Add-new-version* is optional, and it is assumed that the new version is to be added to the end of the list.

3.3.2.3 Version Set Operations

In summary, the basic operations that can be performed on the version set of a conceptual object are:

1. **Create-Version-Set**: -> Version-set.
   This causes the creation of a new version set object. Initially, this version set is empty.

2. **Add version**: Version-set X Simple Object X [Set] -> Version-set.
   This adds the given object to a given version set. The third argument is optional (and, therefore, enclosed in brackets). It indicates which objects of the version set are to be the predecessors of the given object. All members of this set must be members of the set of latest versions of the given version set. If the version set is a linear version set, this argument is unnecessary since there can be one and only one latest version.

3. **Latest-version**: Linear-version-set -> Simple-Object
   Latest-version: Branching-version-set -> Set.
   This operation selects those versions from the version set that are the latest versions. They are the versions for which there is no successor. If the version set is linear, there is only a single latest version, and, therefore, this function returns that object. If, however, it is a branching version set, the function returns the set of objects that are the latest versions.

4. **Delete version**: Version-set X Simple-Object -> Version-set.
   This causes the given object to be removed from the given version set. The result is the old version set with the given object missing. Etracting the object does not cause the remaining objects to be renumbered. In this way, the fact that there were once versions there has been retained.

5. **Iterate-over-versions**: Version set X function.
   This operation will cause the given function to be applied to each of the members of the given version set. The function is applied in the time-history order of the versions in a linear version set. In a branching version set, the function f is applied to the versions in an order such that if f is applied to v_i, then all f must have been applied to all predecessors of v_i.

All of these operations are available for conceptual objects, and they all manipulate the version history part of conceptual objects.

3.3.3 The Repository

The *repository* is the place in which objects are stored archivally. An object that has been stored in the repository will not disappear between sessions. The repository is logically partitioned into two pieces:

1. **The immutable repository.** Information in this repository cannot be changed. The only operations that can be performed on these objects is *read* and *delete*. Two read operations on a given piece of information occurring at different times will always produce the same result if the object has not been deleted.

2. **The mutable repository.** Information in this repository can change in place. That is, it is possible to modify objects in the mutable repository such that two read operations on a given piece of information occurring at different times can possibly produce different results.

We will see that it is possible for certain parts of an object to be in the mutable repository while other parts are in the immutable repository. The data model enforces the read-only discipline of the immutable repository.

A *repository object* is any object that has been stored in the repository. The set of all repository objects is the set of all things that the object management system knows about and can manage intelligently. The mechanism provided by the object management system is only available to repository objects. For example, the only objects that will be remembered from session to session are repository objects.

How does an object get to be a repository object? In the course of running application programs, objects of many different types will be created. These objects initially exist within the address space of the program. They can be operated on by other programs, but they are not as yet repository objects. In order for them to attain this status they must be explicitly inserted in the repository by some program. The successful insertion of an object into the repository will be called *committing* the object. The operation of committing an object to the repository is analogous to writing an object to a conventional file system. Once it has been committed, any facilities that have been defined to the object management system for objects of its type are automatically made available.

Each repository object has a unique identifier called a *repository key* associated with it. This repository key is created whenever an object is commited to the repository. When this repository key is presented to the access level routines of the object management system, it is always guaranteed to return either the same object that was stored under it or else some indication that the object has been deleted.

There are four basic operations that can be performed on the repository that involve simple objects. All of these operations except the second one have a side effect on the state of the repository. They are:

1. **Store-object**: Repository X Object -> (Repository, Repository-Key)
   This operation changes the state of the repository to include the given object. A unique identifier (i.e., a repository-key) is also returned. We will say the the object was stored *under* that key.

2. **Retrieve-object**: Repository X Repository-Key -> Object
   This operation returns the object that was stored in the repository under the given repository key. If the object has been deleted, the operation returns an indication to this effect (e.g., *false*).

3. **Delete object**: Repository X Repository-Key -> Repository
   This operation changes the state of the repository such that it no longer includes the object that corresponds to the given repository key. The key, however, will not be reassigned. This means that one can never get an unexpected value as a result for a *retrieve-object* operation.

4. **Modify object**: Mutable-Repository X Repository-Key X Object -> Mutable-Repository
   This operation can only be performed on the mutable repository. It takes an object and an existing unique identifier and alters the state of the mutable repository such that the given new object is associated with the given repository key. The object that used to be associated with that unique identifier is forgotten.

These operations can also have side-effects. For example, the operation *Store-Object* has the side effect of adding the given object to some predefined collections (i.e., Classes. See Section 3.3.7). The side efects will vary depending on the type of the object involved. The side effects of adding a report will be different from the side effects of adding a graph. A report will be placed in the report class while a graph will be placed in the graph class.

3.3.4 References

By making an object into a repository object, one copies it into a shared memory space. Users and programs can access this memory in a controlled way. The basic mechanism for referring to a repository object or a set of repository objects is called a *reference.*

In general, a repository object will contain other repository objects as components. These inter-object connections are made via a *reference.* A reference is an expression that when evaluated will produce some repository object or set of repository objects. We will call this object or set of objects the *referent* of the reference. Retrieving a structured object consists of retrieving the root of that object and then retrieving each of its components by evaluating the corresponding references. This process can be repeated for the components of each of the components, and so forth, until the entire object has been retrieved.

The basic operations for a reference are as follows:

1. **Create-Reference**: Type X Definition -> Reference.
   This operation creates a new reference. The referent can be known, as in the case of a static reference, or it can change over time, as in the case of a dynamic reference. The type of a reference identifies how the definition is to be evaluated. The definition is some expression, which when evaluated, will produce the proper object or set of objects.

2. **Evaluate-Reference**: Reference -> Object or Set of Objects.
   The method for evaluating a reference depends on the type of that reference. When a reference is evaluated, the proper method is used to produce the desired referent.

An example of a reference is a latest version reference. This reference points to the latest version of some version set. Let us call the type of this reference *latest-version*. The definition of this reference is the repository key, K_1, that corresponds to the version set for conceptual object C from which we are extracting the latest version. Applying the operation Create-Reference (Latest-Version, K_1) will produce a reference that will always yield the latest version of C.

3.3.5 Object Content

The *content* of simple repository objects is immutable in that it can never change. In other words, the content of an object is read-only. The content of a simple object is written once and accessed many times in the future. The content, therefore, must be a part of the immutable repository.

We view the content of an object as that information which defines it. The content plays a role much like the logical key in conventional database systems. Changing the content of an object is viewed as creating a new object. This new object is, to be sure, related to the old object from which it was derived, but it is really a different object with separate identity. The way in which we reflect changes to the content is by making an addition of the new object to the version set of the conceptual object.

The defining nature of the content requires that it be used in the equality testing operations. They are:

1. **=Test?**: Object X Object -> {True, False}*
   This operation returns **true** if the the two objects have the same content. The **=Test?** operation for each of the corresponding components of the two objects must also produce **true**. The two objects must have the same form.

2. **==Test?**: Object X Object -> {True, False}**
   This operation returns true if the content of the two objects are in fact the same physical object. That is, the content of the two objects must have the same repository key.

Notice that these operations apply only to the content. The attributes of the objects can be different and the objects can still be the same.

The content is made up of a set of named components. A component must be a reference, a set of references, or a primitive type (i.e., a string, a number, or a sequence of bits): Each component can have any of the following characteristics:

1. **Editable or uneditable.** If a component is editable, then there exist abstract operations that can change the value of that component. The object management system need not know what these operations are. It

*This is like EQUAL in LISP.

**This is like EQ in LISP

is sufficient to know that they exist. If a component is uneditable, then its value can never change. The most common use for this characteristic is for a content that is acting as a logical database key.

2. **Single-valued or Set-valued.** A component can be either single-valued or set valued. If it is single valued, there can be only one object that is the value of that component. If it is set-valued, there can be many.

3. **Ordered or unordered.** A component that is specified as *set-valued* can be further specified to be ordered or unordered. If the set is ordered, it is possible to deterministicly iterate through the members of the set. The default value for this property is *unordered.*

4. **Non-empty or Possibly-empty.** A component can be declared to be *non-empty* which means that there must be a value assigned to that component. The default for a component is *possibly-empty* which means that it is possible to have no assigned value for that component.

All of these characteristics have implications for the operations that are available for objects of the given type.

The operations that are available for the content of a simple object are given below:

1. **Create-Repository-Object**: Repository X Object -> Repository
   This operation is not the create operation that would be a part of the abstract data type semantics for the given object type. It is an operation that is invoked in order to include a new object in the repository. It creates a new repository with the given object as a member. If the object type is not known to the repository or if it violates some basic constraint, the operation fails.

   If there are no components in the content that have the *editable* characteristic, this operation will add a simple object to the repository. In this case, the content would contain a symbol that stands for some external object. If there is at least one editable component, this operation will add a new conceptual object.

2. **Iterate-Over-Components**: Object X Function.

This operation causes the given function to be applied to each of the components of the given object.

3. **Get-Component**: Object -> Object
   If the component is single-valued, this operation is available to get the value of that component.

4. **Iterate-Over-Component**: Object X Function
   If the component is set-valued, this operation will apply the given function to each of the members of the set of component objects.

5. **Get-Component-n**: Object X Integer -> Object or a Set of Objects
   It the component is set-valued and ordered, this additional operation is available to get the component that is occupying the nth position in this ordered set. If n is greater that the total number of components in the set or if it is less than one, the operation fails.

There are also operations that are analogous to the last three operations that deal with references to components instead of the components themselves. They are *Get-Reference, Iterate-Over-References, Get-Reference-n.* They simply do the same thing that the analogous operation does except with the reference that is standing for the actual component; whereas, the previous operations automatically evaluated these references. Both types of operation are useful. One often needs to get the reference in order to process the definition. For example, we might want to have access to a conceptual object from which the latest version is being drawn.

It should be noticed that there are no value-setting operations provided for the content of an object. This is a direct result of our assumption that the content is read-only. This is also in distinction to attributes as we will see in the next section.

3.3.6 Object Attributes

As we have indicated above, the content of an object is that portion of the object information structure from which the object derives its identity. In contrast to this, it is often useful to record additional information about an object. This information can be conceptualized as a set of statements. Each statement summarizes some property of an object. These statements do not define the object; they merely record observations about it. As we shall see, these observations can be made by users of the system, or they can be derived by the system itself.

An object of some type can have an arbitrarily large set of attributes. All attributes have a name that is unique within the set of attributes for a given object type. The name identifies the attribute for the purposes of applying attribute--oriented operations. An attribute can additionally have any of the following characteristics:

1. **Settable or Derived.** A settable attribute can be directly set to a value by a user of the system. The value of a derived attribute is computed by a procedure that uses some other information in the repository.

2. **Set-Valued or Single-Valued.** When one applies the get operation to a set-valued attribute, the result is a set of values. This also requires that there exist a set operation that will add a new member value to a set valued attribute. A single-valued attribute can only have one value. This means that the set operation for this type of attribute can only replace the old value with a new value.

3. **Ordered or Unordered.** This characteristic only applies to set valued attributes. An ordered attribute requires that there be a well-defined total order on the members of the set. If an attribute is unordered, no such order exists.

4. **Mutable or Immutable.** A mutable attribute can be changed. If it has a value at some point in time, the execution of a set operation will change the old value to the given new value. If an attribute is immutable, it can only be set once (i.e., initialized). Once it has been set, that value will remain in force as long as the object exists.

An attribute that is settable has a value of **null** until it is set to a value. **Null** is a distinguished value for an attribute that indicates that the attribute is defined for the object at hand, but that it has not as yet received a value. A settable attribute must receive its value from a user of the system. There is no way in which the system can determine its value. An example for a report is the attribute *due-date.* In general, there is no way in which the system can determine when the report is due. Only the user can indicate this information.

3.3.6.1 Derived Attributes

A *derived attribute* is one whose value depends on some other information in the repository. It is an expression of some inter-relationship between two or more pieces of data. A derived attribute cannot be set to a value since its value is always dependent on the value of some other piece of data. This correspondence is enforced by a program. ODM allows for the use of an arbitrary program to derive the value of an attribute.

Derived attributes have appeared in other contexts [31]; however, they are especially important in the object management system environment. This is a direct result of the observation that actual objects can be stored in the machine. Therefore, it is often the case that some observations about an object can be made by some program that takes the content of the object as an argument. For example, a graph object might have an attribute called *number-of-curves* whose value is an integer that expresses at any point in time the number of curves that appear in that graph. The number of curves in a graph can be derived from the data structure for a graph by a program that simply counts the number of curves that are represented.

3.3.6.2 Attribute Operations

Since there can be many different attributes, one must be able to iterate over all possible attributes for a given object. The following operation provides that capability:

- **Iterate-Over-Attributes**: Object X Function.
  This operation applies the given function to each of the existing attributes of the object.

If an attribute is set-valued, the following two operations are available for that attribute (they are a convenience, since iteration operations are already provided for all sets):

1. **Iterate-Over-Values**: Object X Function.

2. **Iterate-Over-References**: Object X Function

These operations apply the given function to each of the members of the set of objects that are the value of the attribute for the given object. The side effects of these iterations are the side effects that are the side effects of the given function.

For each attribute of an object, there are operations that will get the value of that attribute. These operations are of the following form:

1. **Get-Attribute**: Object -> Object.
   This operation returns the value of an attribute. The function name used above is the name of an operation for a generic attribute. In reality, there would be a function like this for each attribute. Therefore, we would get functions like, *Get-Report-Chapters* which returns the set of chapters from a report.

2. **Get-Reference**: Object -> Reference
   This operation is very much like the above operation except that it returns the imbedded reference or set of references.

For example, a report that has a bibliography as a component has a corresponding operation called *Get-Report-Bibliography* that will get the value of the bibliography component for a given report. The *Get-Reference* operation is available for cases in which one needs to process the actual reference. One might, for example, need to know what type of reference is contained in a certain component or, in the case of a *latest-version* reference, from which conceptual object the latest version is being extracted.

If the attribute is set-valued and ordered, we provide the following operations for convenience:

1. **Get-Attribute-n**: Object X Integer -> Object
   This operation returns the nth component of a set-valued attribute. It automatically dereferences the references that stand for the components.

2. **Get-Reference-n**: Object X Integer -> Reference
   This operation returns the nth reference to a component of a set-valued attribute.

Attributes that are not derived are settable. A user settable attribute is one whose value must be supplied by a user of the system. An example, is the attribute *due-date* for a report. In general, there is no way in which the system can determine when the report is due. Only the user can indicate this. A settable attribute must have an accompanying operation for assigning values to it. These value-setting operations are very parallel in form to the get operations that were described above. They are:

1. **Set-Attribute**: Object X Object -> Object
   This sets an attribute of a given object to be the other given object. A static reference is constructed and used to accomplish this mapping.

2. **Set-Reference**: Object X Reference -> Object
   This sets a component of the given object to be the given reference.

3. **Set-Attribute-n**: Object X Integer X Object -> Object

This operation and the next are provided for convenience. This one sets the nth member of a set-valued attribute to the given object.

4. **Set-Reference-n**: Object X Integer X Reference -> Object
This operation sets the nth member of a set valued attribute to be the given reference.

Notice that operations like these are not available for the components that make up an objects content. For immutable attributes, these operations will only succeed if the current value of the attribute is null.

3.3.6.3 Parameterized Attributes

Object attributes are functions that map the given object into some other set of values. These functions are of a single argument, the object itself. We also allow attributes to be parameterized. A *parameterized attribute* takes one or more additional argument and maps the cross product of these arguments and the object itself into the set of domain values. This is a generalization of the simple attribute. A simple attribute is a function of a single parameter, the object itself, while a parameterized-attribute is a function of the object itself as well as its additional parameters.

An example of a derived parameterized attribute is *Contains-Word (word)* of a paragraph. The argument *word* is a text string that is supplied to the attribute function in order to compute the value of the attribute as one of **true** or **false.** The value of the attribute is **true** if the given paragraph contains the given word and **false** otherwise. The attribute function would be a function of two arguments, the specific paragraph of which we seek the attribute value and a specific word. Another example of a parameterized attribute is the attribute *outward-appearance* which takes a single parameter, the name of an output device. We would, then, have different values for *outward-appearance ("dover"), outward-appearance ("line-printer")* and so forth.

Parameterized attributes require that the argument lists for all attribute operations be extended to include the additional parameters. An example of an extended operation to handle a parameterized attribute is:

```
    Get-Attribute: Object X Param₁ X ... X Param₂ -> Object
```

The other attribute operations can be extended in a similar way.

3.3.7 Classes

Objects are organized into *classes.* A class is a named collection of objects that share some semantic properties (e.g., the class of Reports, the class of chapters written last Tuesday). A repository is organized as a collection of classes. A given object can be a member of many classes. The objects that make up a class are its *members.* A given member of a class will be referred to as an *instance* of that class.

A class can be either a *base class* or a *non-base class.* Base classes are defined independently of other classes in the repository. Non-base classes are defined in terms of base classes or other non-base classes. Non-base classes are related to other classes by *interclass connections.* The set of all classes and the interconnections between them will be called the *class graph.* A non-base class can be either:

1. **Permanent.** A permanent class is defined by an object schema (see section 3.4.1 on page 80. The collection of objects that is defined by the schema is always available within the repository.

2. **Temporary.** A temporary class is defined by an expression such that the members of the class will be manifested when the expression is *evaluated.* There is no guarantee that this class will be maintained by the system. The class will not appear in the class graph. It can be used, however, in all contexts in which a permanent class name is appropriate.

Whenever a new repository object is created, it must be explicitly inserted into one of the existing base classes. A side effect of this insertion might be that the new object will also be inserted into other non-base classes by the system. Also, once an object is a member of a base class, it remains a member of that base class until it is explicitly removed. The membership of certain non-base classes can change as the properties of its members change. The user is does not have to be aware of this shifting membership until he inspects the content of the non-base class. The system makes sure that the proper class insertions and deletions are made for these non-base classes.

The most fundamental operations for classes are the following:

1. **Create-Class**: Repository X Base-Class-Schema -> Repository.
   This operation takes a *schema** for a base class and adds that class to the repository. The class is initially empty.

2. **Delete-Class**: Repository X Schema -> Repository.
   This operation causes the given class to be deleted from the repository. If the class is a base class, all its members are deleted as well. For any class, the classes that are dependent on the given class (via interclass connections) are also deleted.

3. **Modify-Class**: Repository X Schema X Class -> Repository.
   This operation causes the existing schema for the given class to be superceded by the new schema.

As we have indicated, non-base classes are not defined independently. They are logically linked to related classes via *interclass connections.* Two classes that are related by an interclass connection will be said to be *connected.* The term interclass connection refers to the fact that one of the connected classes was created by means of an operation that involved the other connected classes. This implies certain constraints on the membership of the connected classes. Two classes, C_1 and C_2, can be related to each other if C_1 is a subset of C_2. In this case, we will call C_1 a *subclass* of C_2, and we will call C_2 a *parent class* of C_1.

*A schema is a description of the semantics of the members of a class. They will be described in detail in Section 3.4.1.

There are several different ways in which interclass connections can be specified.

1. **Restrict**: Class X Predicate -> Class.
   This operation defines a subclass in terms of properties of a *parent class.* The members of the subclass are exactly those members of the parent class for which the given predicate is true. A member of the restricted subclass is also a member of the parent class. For example, we could define *Short-Reports* to be the subclass of the class of *Reports* that contains those members of *Reports* that have fewer than three chapters.

2. **Subset**: Class -> Class.
   This operation defines a class to be a user-controlled subclass of another. A subset class contains exactly those members of its parent class that have been inserted into the class by some user. The members of a subset class are not specified in the definition of the class as they are in a restrict. Instead, they become members of the class only when they are identified as such by an external agent (i.e., a user). The class of *Reports-for-Distribution* is a subset of the class *Reports.* Objects are entered into this class by a user of authority who decides which reports should be distributed to people outside of the organization.

3. **Merge-Members**: Class X ... X Class -> Class.
   This operation produces a class that contains the union of the members of the classes in the class list. It is not required that the elements in the list of classes have a common root class (as in the SDM). In this way we can define the class *Workstation-Project-File* to be the **merge-members** of the classes *Workstation-Documents, Workstation-Graphs,* and *Workstation-Messages.*

4. **Extract-Common-Members**: Class X ... X Class -> CLass.
   This operation produces a class that contains the intersection of the members of the classes in the class list.

5. **Extract-Missing-Members**: $Class_1$ X $Class_2$ -> $Class_3$.
   This operation produces a new class, $Class_3$, that contains all those members of $Class_1$ that are not members of $Class_2$. This is like the conventional set difference operation.

   The next two connection types are both defined between two classes of dissimilar type. One of the classes has members that are sets of objects whose members are drawn from the other class that we will call the *underlying class.* The class that contains set-valued members and the underlying class are linked by an interclass connection.

6. **Abstract**: Class X Grouping-Program -> Class.
   This operation defines a new class whose members, called *abstractions,* are subsets of the underlying class. Each abstraction contains elements that are specified by some grouping program on the underlying class. Let us call this grouping program P and the underlying class C. Then, if P: C -> $\{x_1, \ldots, x_n\}$, the abstraction class A defined by P will contain n members, one for each of the possible values in the range of P. All members of C for which P produces x_i will be members of the same abstraction. Some of the members of an abstraction class might also be permanent repository classes while others might not. A member of an abstraction class would not be a permanent repository class if it was not of interest in modeling the application domain.

7. **Aggregate**: Class -> Class.
   This operation defines a class that contains *aggregates* (see above) as members. The members of each aggregate are drawn from the underlying class. Some of these aggregates may be defined as permanent classes in the repository, and others may not be if the members of these classes are not of interest to the application. Notice that unlike an abstraction class, the definition of an aggregate class does not determine the members of each aggregate.

Once the permanent classes of the repository have been defined, the following operations can be performed on base classes and subset classes:

1. **Insert-Member**: Class X Object -> Class.
   Insert-Member makes the given object be a member of the given class. In the case of a base class, the object must be of the proper underlying type. In the case of an subset class, the object must already be a member of one of the parent base classes for the given class.

2. **Delete-Member**: Class X Object -> Class.
   Delete-Member removes the given object from the given class. In the case of a base class, the object is implicitly removed from all subclasses

that currently contain it. This deletes the object from the system. In the case of a subset class, the object is removed from that class and any subclasses of the subset class.

The following operations are always available for any repository class (base or non-base):

1. **Is-a-Member-of?**: Class X Object -> {true, false}.
   This predicate tests to see if the given object is a member of the given class. If it is, the function returns true, otherwise, it returns fallse.

2. **Iterate-Over-Class-Members**: Class X Function.
   Since a class is basically a set of repository objects, it is possible to iterate over the members of any class. This operation applies the given function to each of the members of the given class.

The general class mechanism is one of the major ways in which we support office applications. The application code refers to those classes that are relevant to the task at hand. If the relevant class is not a permanent repository class, the application can make use of the general object semantics and create a dynamic class by means of a class specifier expression (See Section 3.4.10 on page 112).

3.3.8 Access-Specifiers

In an office workstation environment, it is extremely important to be able to control the use of objects. We should be able to specify carefully our intentions about how our objects are to be manipulated. The object management system should support this need by giving us flexible tools for expressing these constraints.

An *access specifier* is an expression of <u>who</u> can perform <u>what</u> operation on a given object. It is created by means of the following operation:

- **Create-Access-Specifier**: Operation X User-Class -> Access-Specifier.

This operation creates an access specifier that states that members of the given user class are allowed to perform the given operation on objects to which that operation applies. For example, Create-Access-Specifier (Get-Report-Bibliography, {Stan, Mike}) would create an access specifier that says that Stan and Mike are allowed to perform the operation that extracts the bibliography component from a report.

The above example used a constant set of user names to specify the set of users who are allowed to perform the operation. In general, we use a scheme that uses a distinguished class of users and a query against that class to specify the user-class. The form of this query is as a class-specifier (See Section 3.4.10 on page 112).

In order to determine whether or not a given user can perform some operation: the system makes use of the following access-specifier operation:

- **Check-Access**: Access-Specifier X User-id -> {true, false}

This operation takes a unique identifier that is associated with the current user and checks to see if that user is allowed to perform the operation of the given access-specifier. We assume that the system has adequate authentication facilities and can always determine who is currently making the request on the object management system.

3.3.9 Triggers

Although ODM is basically a model of data, we include hooks for certain process oriented specifications. The mechanism for this is called a *trigger.* A trigger determines what actions should occur as a side-effect of other actions. It is a procedure that is executed whenever some condition called the *trigger condition* is satisfied.

A trigger condition has three essential parts:

1. An **operation**. Any of the available repository operations (See below).

2. A **user class**. A subset of the distinguished class *Users*.

3. An **object class**. A subset of the objects to which the operation applies.

The trigger condition is *matched* if a member of the given user class invokes the given operation on a member of the given object class. When the trigger condition is matched, a program called the *trigger program* is called. A trigger is, therefore, created by means of the following operation:

- **Create-Trigger**: Operation X User-Class X Object-Class X Program -> Trigger

The first three arguments specify the trigger condition, and the given program is the trigger program.

A trigger is *activated* by the following operation:

- **Activate-Trigger**: Trigger

Activating a trigger involves checking the trigger condition and then invoking the trigger program if the trigger condition was matched. The trigger program is ignored if the trigger condition is not matched.

3.3.10 Repository Operations

Each class in the repository has some underlying abstract programming language type. An abstract type has a set of operations that provide the interface to objects of that type. The object management system must be aware of some of these operations in order to its job, while others do not concern it. The operations that are specified in class schemas are the province of the object management system.

The operations that have been outlined in the earlier parts of this section will be called *repository operations.* For any abstract type that can be stored in the repository, its repository operations are a subset of its abstract operations.

Each repository operation has two things associated with it. For an operation Op they are:

1. an access-specifier, that is returned by Access-Spec (Op) and

2. a set of Triggers, that is returned by Triggers (Op).

In order to execute a repository operation on an object, one must use the following operation:

- **Invoke**: Operation X Object.

The meaning of the Invoke operation is:

```
If [Check-Access (Access-Spec (Operation), User-id),
    Operation (Object),
    For-each-member [
      Triggers (Operation),
      lambda (t) Activate-Trigger (t)]]
```

3.4 Linguistic Concepts for ODM

This section is intended to describe the basic linguistic capabilities that accompany the Office Data Model and the motivation behind them. It is not an extensive treatment of the precise syntax of this language. The syntax is covered in the appendices. This section assumes that the reader is familiar with the basic data structuring primitives of ODM as described in Section 3.2.

3.4.1 Object Schemas

An *object schema* is a textual definition of the semantics of a class of repository objects.* An object schema is written in a language that we shall call The Object Definition Language (ODL). There is one object schema associated with each permanent repository class, and each schema applies to only one class. Since all object classes that are handled by the object management system are descibed in the same language, the schemas provide an excellent documentation of repository object behavior.

Writing a schema implies the existence of certain repository operations. It is a contract between the way users view an object type and the programs that implement the operations on that object type. All repository objects must, at least, comply to the specification that is embodied in the schemas for the classes of which it is a member. If an object belongs to some class, it must be possible to apply any of the operations that are implicitly defined in the schema for that object. For example, if a letter has a component called *Inside-address*, it must be possible to apply the operation *Get-Inside-Address* to a letter yielding the appropriate piece of that letter.

There are two ways in which this schema contract can be fulfilled:

1. **User supplied programs.** The user can write a servies of programs that will implement the semantics specified in the schema in terms of an arbitrary data representation. This allows representations that are highly optimized to the needs of the application.

2. **System compiled defaults.** The user can supply the schema, and the system will compile it into a set of programs that make use of a set of system default implementations. This might result in representations that are not very well suited to the application, but it provides a way to

*There are also implementation level objects called *class schemas* that are used by the programs that implement the semantics of an object. Class schemas are loosely analogous to the object schemas that exist at the user level. When we use the term schema, we will usually mean the user-level object schema. If there is a possibility for confusion we will use the more complete term.

get an application running quickly and with a minimum amount of effort and expertise.

We will have more to say about this mapping of schema-level semantics to program-level implementations latter in this document.

The integration of different object types is achieved by using the object definition language (ODL) to describe all objects. The collection of all high-level schemas provides a uniform view of the contents of the object repository. Since all schemas are expressed in terms of the same language, thereby utilizing the same data modeling primitives, it is possible to exploit semantic constructs that are shared by each object type. For example, a basic assumption of ODM is that many interesting object types are made up of components. The system must be able to manifest the components of an object regardless of its type. It is, therefore, possible to retrieve all reports or graphs that have more than five components. In order to process this request, there must be some mechanism for the system to obtain and count the components of a report or a graph.

3.4.2 Individual Schemas

Based on the above discussion, we can see that the semantics of a class of objects is derived from the definitions in the object schema of that class. An object can belong to many classes. Therefore, the total semantics for a given object is derived from the collection of object schemas that are associated with the classes of which that object is a member. A user would often like to configure the semantics of a single object to be slightly different from the semantics that is specified by its associated object schemas. In order to accommodate this requirement, the object management system provides the notion of an *individual schema*. An individual schema is like an object schema in form. It is associated, however, with an object instead of a class. This implies that the specification in an object schema applies to one and only one object. No other object can now or in the future share that schema by becoming a member of some class. In this environment, an object can be very complicated; it can be thought of as a database in its own right.

The object schema refers to groups of objects and expresses policy decisions about object semantics. The individual schema defines specific object requirements. The notion that all reports must have chapters is a policy decision regarding the manner in which reports will be constructed and can be expressed in the object schema for reports. All objects that belong to the *Report* class inherit this property. On the other hand, a security requirement that says that the group progress report for this year can only be written by one of three trusted people is a requirement for the progress report object only and would be expressed in the individual schema.

It is required that the individual object schema not violate the constraints imposed by its class definition in the general object schema. The individual object schema typically includes two types of semantic requirements.

1. It imposes *tighter restrictions* on the object than was specified by the general object schema. For example, the general schema for a report may specify that part of the contents of a report is a non-empty ordered set of chapters. The individual schema for a particular report may further require that the chapters that are included in that object must have authors who are members of the Office Automation Group.

2. The individual schema can also include the *addition of new object properties* such as a new attribute. This clearly does not violate any previous constraints. It is also possible to add attributes in an individual object schema. One may want to define a report that is exactly like the report definition in the general object schema except that it can contain a title page or it can contain an attribute called *Responsible for* that contains the name of the person who has ultimate sign-off authority for the report.

One way to imitate the behavior of individual schemas is to define a new subclass for each object. For example, we could define a subclass of the class *Reports* which contains reports with the the additional attribute *Responsible for.* This attribute, however, is peculiar to this one report. Creating subclasses for every object is not a very effective solution. A large number of singleton subclasses would be created.

3.4.3 Class Definition

Class schemas are written as a single textual block. Within this block of text, one defines the semantic features of the class using ODL. An ODL class definition can be broken down into following definitional elements:

1. A *class name* that is used to refer to the class. Class names are only assigned to permanent classes, and they must be unique.

2. A *class description* which is a textual description of the class. It is much like comment. This serves to document the nature of the class and is, therefore, optional.

3. A *class definition* which indicates how the members of the class are determined. The class definition can take one of two forms:

    a. It can be a *base class* in which case the members of the class are defined independently of the other classes in the repository.

    b. It can be a *non-base class* in which case it can be defined in terms of one or more other classes in the repository. A non-base class is defined in terms of an expression that specifies an interclass connection. Interclass connections were discussed in the previous section on ODM.

4. Each schema has potentially four *aspects* that describe different portions of the semantics of the class. Each of the aspects has its own sublanguage and appears in a separate textual section of the schema. The four aspects are:

a. The *content* which specifies the components of an internal object. (See Section 3.4.5 on page 86.)

b. The *attributes* which specify properties of the object. The attributes are a set of statements about the object. (See Section 3.4.6 on page 87.)

c. The *history* which specifies how the version history is to be managed as well as any attributes that apply to the collection of versions as a whole. (See Section 3.4.7 on page 96.)

d. The *control* which specifies what actions should occur when certain operations are invoked on members of the class. This includes such things as information concerning access rights for an object. (See Section 3.4.9 on page 3.4.9.)

In these brief characterizations, one should notice parallels between the aspects and certain features of ODM. Each aspect except the content is optional. A base class must have the content defined. We will say more about the details of each of the aspects later in this chapter.

3.4.4 Class Aspects

An object schema consists of a header and up to four major sections. THe header includes the class name, the class description, and the class definition. The other sections contain the definition of the fundamental class aspects. The four class aspects are content, attributes, history, and control. Each of these aspects will be described in more detail below. A schema can contain definitions for any number of the class aspects. It is not necessary to define all four aspects in a given schema. In fact for some derived classes, it is possible to have no new aspect definitions. The schema might simply consist of the class name and the derivation specification. For example, the class of *Long-reports* might only consist an expression that states that members of this class are members of the class *Reports* that have more than six chapters. All of the semantics for members of *Long-reports* is inherited from the class *Reports.*

Another example of a schema definition that does not involve all four aspects might be a class of employee records. This class contains objects that are very much like the objects that one might define using a general purpose database management system. Each record contains a number of statements or facts about a given employee. The nature of these facts is defined in the attribute aspect of the object schema. A record has no editable content since the object that is being described does not exist in the machine.

The next several sections will investigate the functionality of the four class aspects in much greater detail.

3.4.5 Content

The content portion of a schema is the part that describes the content of the object. The content section of a schema is broken down into *component definitions.* A component is a reference to another object that is also a member of the repository. For a report, the components from which it might be constructed are a set of chapters, a set of appendices and a bibliography. The content section of the schema for a report, then, would include three component definitions, one for each of these components.

Each component is assigned a class from which its values can be drawn. We will call this the *value-class* for that component. The value class can be either a permanent repository class identified by the class name or a temporary repository class that is defined by a class specifier (see Section 3.4.10). The value class creates a restiction on what type of objects can be connected to the given component.

Each component can have an access specifier and a list of triggers associated with it. These control specifications are written directly following the component definition to indicate that they are to be associated with that component.

3.4.6 Attributes

The attributes aspect of a schema is the place in which information about the attributes of a class is specified. It occurs within a block of text that is contained within the schema definition for the class.

3.4.6.1 Attribute Characteristics

The essential features of an attribute definition are very similar to the features of a component definition. The differences are in meaning more than in form, although there are some small differences in form. The characteristics of an attribute that are described in an attribute definition are as follows:

1. **A name.** All attributes have a name that is used to identify it. The name must be unique within a class.

2. **A value set.** The value set is a repository class from which values for the attribute can be drawn. This can be a class name or a class specifier (See Section 3.4.10 below).

3. **A set of characteristics.** These include indications of whether or not the give attribute is *single-valued/set-valued*, *mutable/immutable*, *derived/settable*, or *ordered/unordered*.

Each attribute can have an *access specifier* and a set of *triggers* associated with it. This association is established by writing the name of the trigger following a given attribute definition.

3.4.6.2 Attribute Applicability

One of the essential differences in the ways in which attributes can be defined is the locus of applicability of the attribute. Any repository object can have attributes that apply to it and, therefore, attributes can pertain to any one of the following four places:

1. **A simple object.** If the attribute applies to a simple object then each object of this type that is a part of the conceptual object will also have a value for this attribute. In other words, the attribute has a value that reflects the state of a given conceptual object at some point in time. As the object changes, so might the attribute. An example of this type of attribute is the attribute *date-created.* Each version in a conceptual object would have a different value for this attribute. That value would be the date on which the new version was created.

2. **A conceptual object.** If the attribute applies to a conceptual object, it is a statement about the set of versions as a whole. An example for a report is the attribute *related-project.* The value of this attribute is the project entity for which the conceptual report was created. It does not matter how many new versions of this object are created. They will all be for the same *related-project.* The idea of a related project really attaches to the abstract entity that is modeled by the conceptual object. The specifications for attributes of this type reside in the history aspect of the object schema (See Section 3.4.7).

3. **A class** (as an entity). An attribute can also make a statement about the class object as a whole. We will call this a *class attribute.* An example of a class attribute is *number-of-members* for some class. The value of this attribute is the current number of instances of the given class.

An attribute can also apply to a reference to an object. In this case, it makes a statement about the way in which the object is used. Let us assume that there is a *comment* attribute defined for chapters. Suppose we want to create an attribute called *date-connected* to assert the fact that a given comment was connected to a given chapter on a given date. This is not an attribute of the chapter or of the comment. It is really an attribute of the connection (i.e., the attribute). This means that all the attribute operations must work for references as well as repository objects.

3.4.6.3 Derived Attributes

An attribute can be settable by a user, or it can be derived from some other information in the repository. For derived attributes, there are two basic types. They are:

1. derived from content

2. derived from other class aspects

The reason for making the above distinction arises from the fact that changing the content of an object, by definition, creates a new version of the conceptual object. Therefore, the appropriate time to create a new value for an attribute that is derived from an object's content is whenever the content changes, or in other words, whenever an addition is made to a version set. A program can be written that computes the value of the attribute. This program would be triggered whenever a new version for the object is created, if the value of the attribute is to be maintained. Otherwise, this program would be invoked whenever the value is needed.

An attribute can also be derived from the attribute, history, or control aspects of an object. Consider the attribute of a conceptual object called *number-of-versions.* This would be derived from the history aspect. More details on how the derived attribute mechanism works will follow in Section 3.4.6.3.

As was observed above, the derived information can come from several sources. It can be derived from the content of the object or else it can be derived from some other aspect of the object. We have already seen an example above of an attribute that is derived from the content of an object. For an example of an attribute that is derived from another attribute, assume that reports have an attribute called *OKed-by* which has as a value the names of all people who have read and approved the report. Further, assume that a report has an attribute called *ready-for-publication* which has a value, *true* or *false*, expressing whether or not the report can currently be published. The attribute *ready-for publication* for a report will have the value *true* if the value of the attribute *OKed-by* includes the name of the boss; otherwise, it will have the value *false*. Clearly, *ready-for-publication* depends on *OKed-by*.

Derived information for a given object can also come from some aspect of another object in the repository that is related in some way to the first object. For example, a graph object might have an attribute called *Outward-Appearance-for-the-Line-Printer*. The value of this attribute is another repository object that is a representation of the image of the graph as it is printed on the line printer. This outward appearance object has as its content a set of *line* objects that each contain the characters that are printed on a given line of the output. The outward appearance object has an attribute called *printed-by* which has as a value the names of the users who have printed this outward appearance. The original graph object also has an attribute called *printed-on-line-printer-by* which is derived from the corresponding attribute in it's associated outward appearance object. The value of the attribute for the graph is defined to be the same as the value of the attribute for the outward appearance object.

In general, the value of a derived attribute is computed by a program. This program can be invoked at several different times. It can be triggered by a change to the state of the repository, or it can be triggered by a request for the value of the attribute.

The program that computes the value of a derived attribute is stored in the repository. Like any other repository object it can be retrieved by a class specifier. This is an example of a program that is identified by invoking a database query. The program that is used to compute the value of a derived attribute can be specified by any combination of object properties that can uniquely identify a program. This could involve a uniquely assigned name or it could involve the latest version of a program written by a particular user to compute the value of a given attribute. In this case, the user name and the attribute name are attributes of the program objects. The specification of a derived attribute first requires a class specifier that evaluates to a single object from the class of all programs. It also requires the specification of other objects (possibly repository objects) that will be used as arguments to the derivation program. These other objects are typically the things on which the value of the attribute depends. Computing the value of a derived attribute involves evaluating the class specifier to obtain a program and evaluating the argument specifications to obtain a set of arguments. The program is then applied to the arguments to obtain a value or set of values for the attribute.

The Semantic Data Model [31] includes ten different constructs for deriving the value of an attribute from other information in the database. For example, an attribute, A, can be defined to be an *inverse* of some other attribute, B. In this case, the value of attribute, A, for some object y can be computed to be some object x such that B.$x=y$.* These built-in derivation methods have primitive data model constructs defined for their use. The derivation methods, however, are fixed and limited to these ten. In the data processing environment, these ten techniques seem to cover most of the common cases.

In the object management system environment, we believe that it is not sufficient to supply a small set of predicted ways in which information can depend on other information. The ten attribute derivation methods from the SDM might at times be useful, but we feel that there will often be other derivation methods that

*The dot notation is used to denote the application of an attribute to an object. (i.e., B.x means the value of the B attribute of x.)

will be needed. Since we cannot predict what these will be, we allow the use of some general purpose programming language for the specification of new data relationships.

It should be pointed out that the ten methods of deriving attributes in the SDM can each be expressed by a program. For example, we could easily write a program that functions in exactly the same way as the *inverse* derivation method of the SDM mentioned above. This program would take as arguments an object for which the attribute value is being derived, the underlying value class of the derived attribute, and the name of the attribute from the value class for which the given attribute is an inverse. From this information, it can compute the value of an attribute of interest such that the *inverse* constraint is satisfied.

3.4.6.4 Attribute Inheritance

An object can have many components and attributes. These properties derive from the membership of that object in some collection of classes. An object that is a member of a class C has all of the properties defined for C. It also *inherits* properties from other classes that are related to C in the class graph. Inheritance rules are transitive. That is, if class X inherits attributes from class Y which inherits attributes from class Z, then an attribute A that is defined for class Z will also be defined for class X.

The following describes the rules for attribute inheritance via interclass connections.

1. A class C_2 defined as a **restriction** of a class C_1 inherits from C_1 all of its member attributes. For example, since *Progress-Reports* is a restriction class of *Reports*, any attributes that are defined for *Reports* will also be defined for *Progress-Reports*. The value of an inherited attribute for a member of *Progress-Reports* is the value that the object would have as a member of *Reports*.

2. A class C defined by **extract-common-members** on the classes C_1 and C_2 inherits all the attributes from both C_1 and C_2. If C has conceptual object class CC and C_1 and C_2 both have conceptual object classes CC_1 and CC_2, then CC will inherit all attributes from CC_1 and CC_2.

3. A class C defined by **merge-members** on the classes C_1 and C_2 inherits all those attributes that are common to both C_1 and C_2. Attribute A must be defined for both C_1 and C_2 in order for that attribute to be applicable to C.

4. A class C defined by **extract-missing-members** as the difference between C_1 and C_2 (i.e., C_1 - C_2) inherits the member attributes from C_1.

5. A class C that is defined to be an abstraction or an aggregation of a class C_1 inherits all class-determined attributes of C_1 as member attributes of C.

Another way for an object to inherit an attribute is via the *part-of* hierarchy. That is, an object that is a component of some other object can inherit properties from its parent or containing object. If we think in physical terms for a minute, a fender of a car might have no color attribute. The color of a fender before it is installed on a vehicle is the color of the metal that it is made out of. Since this is true of all fenders, it is reasonable to decide not to give fenders a color attribute. However, if a fender is installed on a car, and that car is painted green, it becomes reasonable to ask about the color of the fender. The fender now has a color attribute by virtue of the fact that it is a part of a green car. The same phenomenon can be observed with office objects. If a chapter is connected to a report object and that report object has a *custodian* attribute that indicates who is responsible for the report, then the chapter object should now have a *custodian* attribute regardless of whether or not it had one before.

One must be careful about which attributes are inherited in this way. Consider a report that has an *author* attribute. If we connect a chapter to it that does not have an *author* attribute, we do not want to automatically say that the author of the chapter is the same person as the author of the report. We must have a way to selectively allow only attributes of the whole object that become attributes of the parts to be inherited.

An *extent specification* is a way to indicate which attributes of an object are to be inherited by its components. An extent specification always refers to some set of components or parents of the object at hand. An extent expression can, therefore, be defined to be either upward or downward. A downward extent expression defines a set of objects that extends downward to lower levels of the component tree. It stops at a set of leaves that satisfy a given condition. All nodes between the root and these leaves are said to *belong* to the extent. An upward extent expression defines a set of objects that are connected to the object at hand through component relationships. These objects are all ancestors of the given object in the component hierarchy.

There are two further distinctions that can be made with respect to types of extent expression. They are:

1. **Through** a given predicate. Along each branch, the extent includes all components that satisfy the given predicate up to a component that does not satisfy it. The first object that does not satisfy the predicate along a given branch terminates the extent for that branch. The predecessor of that object (i.e., the last one for which the predicate is true) becomes a leaf node of the extent.

2. **Upto** a given predicate. Along each branch, all components are included in the extent until an object is encountered that satisfies the predicate. This object terminates the extent along that branch, and its predecessor becomes a leaf node of the extent.

Suppose that Y is a component of X. It is possible for Y to inherit an attribute A from a higher level object like X if the class definition for X specifies that A is available to objects at lower levels. This specification is accomplished by an extent expression that accompanies the definition of A in the schema for X. An example of this type of inheritance is contained in the following:

```
Define Class Documents
Definition: Base

Content
  Chapters: Ordered Set of Chapters
  Comments: Ordered Set of Comments

Attributes
  Author: Users
  Co-author: Set of Users
    Attribute Downward Extent Through Chapters
        is Through Chapters
```

The *Co-Author* attribute is made available to all *Chapter* components* of a document up to and including chapter objects. Any chapter object that is connected to the subtree rooted to the *Chapters* component of a document will inherit both the *CoAuthor* attribute and the attribute value from the containing document.

Assume that object X is an ancestor of object Y in the component hierarchy; Y is a component of X. The inheritance of attributes via the component hierarchy can only occur for an object like Y from an object like X. A higher-level object can never inherit an attribute from a lower-level object. Therefore, whenever we define an **Attribute** extent, the direction of this extent is implied to be downward.

One often needs to be able to inherit the value of an atribute from a related object. This is similar to derived attributes, but a derived attribute cannot ever be set to a value. Here, we have an attribute that can be set to a value, but whenever it has not been set, the value is inherited from some containing object. This extent specification occurs in the class definition for the contained object. A **Value** extent is implied to be upward, for a subobject that gets its attribute value from some ancestor.

*The first *through chapters* phrase refers to the component named *chapters* in the content definition.

```
DEFINE CLASS Chapters
Definition Base

Contents
 Paragraphs: Ordered Set of Paragraphs
 Comments: Ordered Set of Comments

Attributes
 Owner: Names
 Co-Authors: Set of Names
   Value Extent is through Documents
   Attribute Extent Through Paragraphs is Through Paragraphs
```

In this example, the attribute *Co-Authors* is available as an attribute of a chapter. If it has not been set, the system will look up the component hierarchy, one-level at a time, to get the value from an object that has both the attribute defined and a value assigned to it. This search will stop at an object of type *Docuement.*

3.4.7 History

The history aspect of an object schema describes the way in which conceptual objects are to be managed. A *conceptual object* is one that can change over time such that the progression of versions in time is maintained. This thesis is a conceptual object that has undergone many intermediate versions before reaching the version that you are reading. All of these intermediate objects are instances of a conceptual object which I call *My-Thesis.*

3.4.7.1 Version Sets

As has been stated previously, one of the most salient features of a conceptual object is its ability to record a version history. A conceptual object is made up of many primitive objects each of which represents a snapshot of the state of the object at some point in time. The conceptual object can have attributes that are distinct from the attributes of the individual versions. The history aspect of an object schema is the place in which these attributes are defined.

THe history aspect is an optional self-contained block of text. The attributes that are defined in the history aspect have exactly the same form as the attributes that are defined in the attributes aspect. The difference is simply that they apply to the conceptual object.

Any object that has an editable portion of its content must have a corresponding conceptual object. This leads to a basic dicotomy in the definition of classes. When we define an object class with an editable content, we are, in effect, defining a pair of classes. One of the classes is a class containing simple objects of the type defined by the schema, and the other is a shadow class that contains conceptual objects whose versions are drawn from the first class. For example, the two classes might be *Reports* and *Conceptual-Reports.* Each member of the class *Conceptual-Reports* will have a version history whose versions are members of the class *Reports.* The class of conceptual objects is very much like an aggregation of the underlying class of simple objects.

3.4.7.2 Proliferation of Versions

In this section, we will discuss what can happen when an addition is made to the version set of a conceptual object, and that conceptual object is referenced as a component of some other object. The discussion will be driven by an example.

As we stated in a previous section, objects and their components are glued together by means of references. There are several different types of references. To review, they are:

1. **Static references.** This is a reference that refers to a particular object.

2. **Dynamic references.** This is a reference that refers to different objects over time. References in this category are very much like database queries. They are expressions that specify a condition on the state of the

repository. The object or objects that match that condition are the referent of the reference.

The most common reference of the latter type is the *latest-version-reference.* Its referent is the newest addition to a specified version set. That is, if VS is a version set, v_i is a simple object, and there is only a single user interacting with VS, then the following relationship is always true:

latest-version (add-new-version (VS, v_i)) = v_i

Evaluating a latest-version reference will produce different objects at different times. Each time a new version is added to a version set, the referent of a latest-version reference involving that version set will change.

The object represented by figure 3-5 is a more detailed view of the object in figure 3-1. In this figure, the links represent *latest-version-references* for the latest version of the parent conceptual object. These links point to the version sets of the components. We see that O1 is really a conceptual object that contains three versions, V1, V2, and V3.

```
                        O1:
                  {V1, V2,  V3}
                   /          \
            O2:                 O3:
        {V4, V5}             {V6, V7, V8}
         /     \              /         \
      O4:       O5:        O6:            O7:
 {V9, V10}  {V11, V12}  {V13, V14}      {V15}
```

Figure 3-5: Version Sets for a Hierarchical Object

The latest version of O1 contains two latest-version references (indicated by diagonal lines). One of these references will evaluate to the newest addition to the conceptual object, O2. In this case, it will contain V5. The other latest-version reference will evaluate to the newest addition to O3's version set, V8.

Notice that, by this mechanism, as the membership of version sets at the lower levels change, members of version sets at the higher levels effectively change. O1 has three recorded versions. Each one contains latest-version references for each of their components. The structure that is drawn in figure 3-5 indicates the component structure for the latest version of O1 (i.e., V3). The earlier versions of O1 (i.e., V2 and V3) might not contain two components, or, at least, might not contain the same two.

This behavior for versions is often exactly what is required. Notice that with latest-version references one can always be sure that the most current version of a component is retrieved. The user does not have to periodically update containing objects to reflect changes to the components. In this way, updates are propagated to other versions automatically. On the other hand, if all versions of O1 contain latest-version-references, there are no versions of O1 that contain the early versions of the objects at lower levels like V9 or V11. Also, the actual high-level objects from which V3 was derived have now disappeared. Unless we have some other mechanism such as timestamps for the individual objects, we cannot reconstruct the actual objects.

There are other desirable behaviors for version sets. For example, we may want the addition of a new version to a version set to cause the previous versions to freeze their state. This would require conversion of all latest-version references (and other dynamic references) to constant-references. Given an object with embedded latest-version references, it is always possible to convert it to an object with constant-references such that the referents of the constant references are the current latest versions of each component. This is possible because for all objects of a given type, it is possible to find each of its components. If a component is a latest-version reference, then it can be converted and replaced. A version set that freezes earlier versions is declared as such in the history aspect of the schema for the appropriate object class. The *add-new-version* operation for this type of version set performs the conversion operation on the previous latest version before inserting the new latest version.

Another approach to managing the state of version sets is characterized by providing a controlled way to allow changes from one version set to effect other version sets. The way in which new versions of a contained object are propagated to the version sets of containing objects is specified in the object schema for the containing object. For example, the schema for reports would indicate which changes to its components, if any, would cause a new version to be created for the report.

In the history aspect of the schema, there is a section labeled **New Versions.** Specifications in this section effect the way in which versions are propagated. For each named component, in the content aspect of the schema, there can be a corresponding entry in the history aspect. A given component can be specified to cause a new version of its parent if:

1. Any change is made. For example, changes to any chapter object of a report will cause the generation of a new version of the report.

2. Any member of the component set changes and a given condition on that object is satisfied. For example, changes to chapters of a report will cause new versions of the report if it is true that the author of the new chapter object is Mr. Smith, the boss.

3. A particular pre-specified member of the component set is changed. For example, changes to chapter three of the progress report can be declared to cause a new version of the report object. This would have to be specified at run time for a particular instance of an object type.

Specifications at this level for components are optional. If a given component is not included, it is assumed that changes to that component will **not** cause new versions to be propagated to the parent object.

3.4.8 Object State

A special kind of derived attribute is the *state* of an object. Intuitively, an object's state specifies where an object is in it's processing history. That is, a state is a reflection of how the object has been handled in the past and, perhaps, what should be done to it in the future. It's current state corresponds to some condition on the existing properties of the object as well as the object's previous states. A state could be modeled as the value of a derived attribute of a conceptual object. By using the conceptual object, which contains the complete object history, the system could derive the state of each successive version up to and including the latest version. We include in ODL a mechanism for talking about the state of an object for convenience. We feel that it is something that will recur in many different settings.

The states that an object can be in and the ways in which that object can move from state to state are specified by a *state machine* in the object schema. The formalism for expressing a state machine resembles the specification mechanism for finite state machines from automata theory. An attribute for a conceptual object (i.e., an attribute in the history aspect) can be specified to have a value set that is a subclass of the class of *States.* The derivation mechanism for this attribute is specified by a state machine. The state machine definition consists of a set of states (one of which is designated as the initial state) and a set of *transitions.* A transition has an beginning state, a target state, and a condition associated with it. The meaning of a transition is that if the machine is in the source state and the condition is satisfied, then the machine should make a transition to the target state. The condition is a predicate. In this case, it is a class-specifier. Therefore, if (S_s, S_t, CS) is a transition in the machine's transition set, the machine is currently in state, S_s, and the latest version of the conceptual object satisfies CS, then the machine will transit to state, S_t.

The transitions for a state machine can be defined to be either optional or mandatory. If the transitions are optional, then the machine can sit in a given state through an arbitrary number of changes. If the transitions are declared to be mandatory, then each change of the object (i.e., each addition to its version set) must cause the state machine to make a transition to a new state. In this way, the state machine can be used to enforce a set of constraints on the ways in which an object can change. If a new version does not satisfy any of the predicates on the transitions out of the current state, an exception is generated.

For a given conceptual object type, there can be arbitrarily many derived attributes that have states as a value set. An example of an object that has several state-valued attributes is a research report that has an attribute named *Publication-state* and an attribute named *Revision-state*, both of which are defined to have values drawn from the class, *States.*

The *Publication-state* is meant to be a reflection of the group of people that are currently allowed to read the report. States in the machine for this attribute have an attribute of their own called, *Readable-by.* Example values for this attribute are *OA-Group*, *Laboratory-for-Computer-Science*, and *CS-and-AI-LAb.* Each of these groups of potential readers encompasses a wider audience. The first state in the machine definition of this state-valued attribute has its *Readable-by* attribute set to *OA-Group.* Presumably, as the set of people that can read this document grows, the state machine will progress to a new state, each of which has a different value for the *Readable-by* attribute. It should be pointed out that the three example values for *Readable-by* given above might not be defined explicitly in the control aspect of the object type. These values might only be defined in terms of the predicates that define the transition from state to state. These predicates could be expressed in terms of individuals who are allowed to read the report.

In this example, the *Revision-state* attribute is defined by a two state machine. Each state has an attribute called *Condition* that has the value *Acceptable* in one state and *Needs-approval* in the other. If any component of the report is edited, then the *Revision-state* attribute enters the state that has its *Condition* attribute equal to *Needs-approval.* Whenever the group leader reads the latest version and sets the *Approved* attribute for that object to *true*, the *Revision-state* attribute transits to the state for which its *Condition* attribute has the value *Acceptable.*

The two attributes described in the previous two paragraphs exist independently. Other state-valued attributes can be defined for this object type. In this example, a research report object can, therefore, exist in several states at the same time.

It should be pointed out that the notion of an attribute as a state machine can be handled with the derived attribute mechanism that we have previously described. However, since it is such an important technique for managing the current processing state of an object, we have provided some special syntax. This syntax allows us to define a state-valued attribute in a more intuitive form. (See appendix for details.) The language for expressing interesting subsets of the object space will also include primitives for dealing with state machines. For example, it is possible to form the class of all final reports that are one state away from a state that has a condition of *accepted.* This might be an interesting set because a little additional effort expended on these documents might cause them to be released to the waiting organization. Similarly, one might want to form the class of all messages that are in a state that is only one state beyond the initial approval state. Both of these examples make use of the class specifier mechanism that allows us to form a predicate that matches all those objects that are *n* states before or after a state that has some property.

3.4.9 Control

The control aspect of an object describes the ways in which objects can be manipulated by users of the object management system. It specifies who can apply various operations to which objects, and what should happen when they do. Unlike the other aspects, some of the specifications that relate to the control aspect will appear within the textual body of the other aspects. This is merely a notational convenience. For example, we have already descibed how the specification of the access specifier and the triggers for a component or an attribute occur textually within the context of the the component or attribute definition.

3.4.9.1 Access Control

An access control specification defines classes of users that will be granted certain privileges to invoke operations on parts of the object. The basic model for access control is that invoking an operation on an object causes a special security program to be executed. This program checks to see if the current user is allowed to perform the given operation. The operation proceeds only if the user has the right to perform it.

It is very important in an office environment for the system to deal with rapid changes to the accessing requirements of the user base. A query on the database of information about the current users of the system specifies who can perform an operation on an object. As this database changes, so do the sets of users who have access rights for the various object types.

The object management system maintains a distinguished class called *Users*. Members of this class are defined to have attributes that are determined to be relevant to workstation applications. The class of *Users* contais an object for each user for each user of the system. Each user object has a unique identifier (possibly a name) that is used to establish the correspondence between the abstract repository object and some user of the system. Any program can always ask the operating system for the unique-id of the user who is currently associated with a given process.

For each operation on objects of a given class that the object management system supports, there is a set of users that are allowed to perform it. This set of users is a subset of the class *Users.* A pair of the form, (Op, U) where Op is an operation and U is a specification for a subclass of the class *Users* is said to be a *right.* The specification U is in the form of a class specifier. In order to determine which users match the class specifier at a given point in time, it is necessary to evaluate U. This is done with a special expression evaluating function. It is necessary to retain the specification of the class U in the form of an expression (i.e., a query) so that changes to the members of the class *Users* will be included in the rights for an object next time they are needed.

Rights are specified in the object schema for a class. It is always possible to determine if some operation, O, can be performed on a given object, x, by the current process. The operating system can give us the unique-id of the user, u_{id}, and the object schemas can be used to determine the rights (O, U) for x. If u_{id} is a member of U, then the operation O can be executed.

Rights for components and attributes are of three basic kinds. They correspond to the basic operations that are supported for components and attributes. There are:

1. **Read.**[*] A user who has the read privilege can obtain a copy of the object into the address space of the program that makes the read request.

[*]This corresponds to the *Get* operations for components and attributes.

2. **Write.**** A user who has the write privilege can modify the repository object in question. If the object is an immutable conceptual object, then the user must have write privilege on the version set for the conceptual object.

3. **Include.** A user who has the include privilege for an object can know of that object's existence. If a user does not have the include privilege for an object, it is as if that object is not there for that user. This is useful for defining "views" of the repository.

These privileges are related in the following way. If a user has the write privilege for an object, then that user also has the read and the include privilege for that object. If a user has the read privilege for an object, then that user also has the include privilege.

The granting of privileges can occur in the control aspect. This imposes constraints on the basic operations on a repository. If one says that reports "as a whole" can only be read by people in the OA Group, then this places restrictions on the *Repository-Get* operation.

Rights can also be assigned to the individual components and attributes of an object. In this way, it is possible to say that the bibliography of a report can be read by anyone, but that the chapters can only be read by members of the Office Automation Group. Similarly, it is possible to restrict access to an attribute of an object. Perhaps only the leader of a group and the commentator can read the *comments* attribute of a paper.

**This corresponds to the *Set* operatoins for attributes and the *Add-New-Version* operation for a conceptual object. In the later case, the write privilege for a component is a constraint on the things that can be changed by a user from one version to the next.

3.4.9.2 Determining Object Rights

Each object in the repository has some set of users who can access it. Determining this set of users can involve more than looking at the schema for a single class. Access rights are determined in two ways:

1. If the object is being accessed as a member of some class from the top-level of the object management system, the access rights are determined by the class hierarchy.

2. If the object is being accessed as a component of an object that one already has accessed, the rights are also determined by the grant privileges that are specified in the class definitions for the contained and the containing objects.

We will now look at these two methods in more detail.

In the first method, an object will be a member of several repository classes at any point in time. The class of users who can access a repository object from the top-level of the object management system is determined by the membership of the object in the various classes of the class graph. Let us call this set of classes, S_c. The rights associated with the schemas for the members of S_c will collectively determine the access rights for a given object, x. The following algorithm can be used to determine the rights for an object, x. The function *Schema-Rights(C)* returns the rights for objects that are members of C. *Descendants(C)* returns the set of subclasses of the class, C. *And-Expression* and *Or-Expression* take expressions as arguments and return expressions that are the logical **and** and the logical **or** of their arguments, respectively.

```
Define Composite-Rights (C)
  And-Expression [
    Schema-Rights (C)
    Or-Expression [
      Iterate [Descendants (C),
        lambda (c)
          if Member-of [x,c] then Composite-Rights (c)
            else False]]]
```

The result of this algorithm is an expression that when evaluated yields the subclass of users that can access x. Using this algorithm, we can arrive at the rights for an object, x, with respect to a class, C, of which x is a member. This is done by taking the logical **or** of the rights for all subclasses of C of which x is a member and then taking the logical **and** of this result with the rights for class, C. The above computation of the rights for a class can be done once and stored with the class. It only has to be recomputed when new classes are added to the repository.

In an environment in which objects are being constructed out of other repository objects, it is useful to allow rights to an object to be determined dynamically by the current content of the object. This addresses method two in the above list. When a user creates a chapter object, it is difficult for that user to predict a priori who should be able to access it. Certain rights specifications would accompany the definition of the chapter object class; however, other rights might be granted to special users depending on the context in which the user is accessing the chapter. A user who has read access to a given report might reasonably expect to be able to read the chapters of that report. Moreover, the creator of the chapter object may sanction this type of access. The user who connects a chapter to a report may not be able to change the access specifications of the chapter. The creator of the chapter might be very willing to relinquish additional rights to the chapter to anyone who can access the containing report. We provide mechanisms for accomplishing this type of *rights inheritance*, the inheritance of access rights from a contained object.

If rights are to be inherited, there must be a way to carefully control the way in which it takes place. Our mechanism is a kind of "handshaking" procedure. The containing object schema specifies to what objects rights will be granted, and the contained object schema specifies from what objects rights will be inherited. The containing object schema specifies for each component an access specifier that contains a designated operation O_1, a set of users U_1, and a subset C_1 of the value set for that component. The contained object schema also contains specifications for an operation O_2, a set of users U_2, and a subset C_2 of the members of the defined class. This latter specification is given for a particular component of a potentially containing object. A user U will be given access to a component C of an object O if the following is true:

```
Or [
  And [member-of [U,U1], member-of [C,C1]],
  And [member-of [U,U2], member-of [O,C2]]]
```

For example, assume that reports contain chapters as components. There will be a subclass of *Users* specified in the Report schema as those users who can apply the *Get-Component* operation on chapters that match a given predicate. This predicate is optional. An example of this follows:

```
Get-Chapter-n by
  Users where group = "OA"
    for Chapters where date-created > 1/1/83
```

So far, this is no different from what has been described already.

To complete this example, the following kind of specification could appear in the schema for chapters:

```
As Chapters of Reports:
  Get-Chapter-n
    by Users where age > 30
    for Reports where author = "Stan"
```

The first line says that what follows applies to chapter objects that are members of the *Chapters* component of a report. For those chapters, we are making a restiction on the use of the Get-Chapter-n operation. The last two lines indicate that Users who are over 30 years old should be able to apply that operation on reports that have Stan as an author.

The result of these two specifications is that a chapter C can be accessed from a report R by a user U who is applying the *Get-Chapter-n* operation on R if the following is true:

```
Or [
  And [Member-of [U, Users where group="OA"],
       Member-of [C, Chapters where date-created > 1/1/83]],
  And [Member-of [U, Users where age > 30],
       Member-of [R, Reports where author="Stan"]]]
```

3.4.9.3 Class Owners

When a class is created by means of one of the class creating operations (i.e., create-class or interclass connections), the user who has invoked the operation becomes the *class owner* (or owner for short). The owner can expand the definition of the class owner by including in the schema a specification for a class of users who will play the role of owner. We will, therefore, use the term owner as both singular and plural.

Only the owner can modify the definition for a class. That is, the access control specification for the **Modify-Class** operation is always defined to be the class owner. An example of the type of modification operation that would be performed by the class owner is altering the access specification for a component or attribute.

A class schema also contains a definition of who is allowed to create subclasses (thereby becoming a class owner for the new class) for the given class. This is an access specification for the all of the interclass connection operations. Only the members of the specified class of users are allowed to create subclasses by performing these operations on the parent class.

Assume that there are two classes such that one is a subclass of the other. Let us call the parent class P and the subclass S. Assume that the functions *Owner*(C) and *Sublass*(C) return the class of users who are the owners of C and the class of users who are allowed to form subsets of C, respectively. The rules for class owners are as follows:

1. Anyone can become the owner of a base class simply by creating the class. The creater of a base class B becomes Owner(B). Once the class is formed, though, the ownership can only be changed by the current owner.

2. Owner(S) must be a subset of Subclass(P).

3. Subclass(S) must be a subset of Subclass(P).

4. For any class C, Owner(C) must be a subclass of Subclass(C).

As we move down the class hierarchy, the owner/subclass specifications can only get tighter. That is, we cannot admit more users at a lower level than the class of users that have these rights at a higher level since the members of a lower level class are also members of the higher level classes.

3.4.9.4 Triggers

Another part of object control is the specification of *triggers.* A trigger is an action that is to occur whenever it is noticed that some condition on the state of the repository has occurred. Triggers are defined within the block of text that is associated with the Control aspect. They are given a name, and that name is used within the other aspects in order to attach a given trigger to an attribute or a component.

Triggers have three main parts.

1. The *trigger condition.*

2. The *trigger program.*

3. The *trigger arguments.*

The trigger condition, in this case, is based on the reading or writing of a repository object. A trigger condition specifies three things:

1. The *action* that is being performed. For components and attributes, the appropriate actions are **read** and **write**. (i.e., get and set)

2. A subclass of objects (the *object set*) that are being operated on.

3. A subclass of the class *Users* (the *user set*) that are to apply the operation.

If the action occurs to a member of the target set by a member of the user set, the trigger program is executed on the trigger arguments.

The trigger programs will be stored and managed by the object management system. They will, therefore, be repository objects that are retrievable by class specifiers. In this way, one can define a trigger that will execute one program at one point in time and some other program at another, depending on the state of the repository. This same technique for managing programs is used for derived attribute functions.

3.4.10 Class Specifiers

A *class specifier* is a formal description of some set of repository objects. It is very much akin to a database query in conventional terms. The options for forming class specifiers provide the basic mechanisms for the data manipulation language for the object management system. The value of a class specifier is always a set of objects. We will call this set of objects the *target set.*

The basic mechanism involves a powerful set of primitives for constructing predicates that make use of the underlying semantics of repository objects. The class of objects that is specified by the class specifier is exactly those objects that are in the repository and that satisfy the given predicate. An example of a class specifier is *the set of reports that have more than five chapters.* This class is a subclass of the class *Reports.* It contains all members of the class *Reports* that currently have more than five members of its component named *chapters.*

In this section we will describe the basic facilities of the class specifier mechanism. We will give examples in English and also in a functional notation that is defined in Appendix E. The actual syntax for class specifiers will be deferred to Appendix C. The purpose of the functional language for specifying classes is to be more precise about the meaning of a given retrieval. The meaning of the functional language is given in the appendix.

Class specifiers are very important to our object specification language. They are used in many different contexts within ODM to define a class of objects. They are also used by applications programs to specify precisely a small set of entities that is of interest to the application. The following discussion is intended to illustrate the capabilities that are provided by the class specifier mechanism.

3.4.10.1 Simple Class Specifiers

The simplest class specifier is a class name. For example, *Reports* is a class specifier that refers to the class of objects named *Reports.* A class specifier of more complexity is basically some other class specifier qualified by a predicate. We will call the predicate a *qualifier.* Other forms of class specifiers make use of more complex qualifiers.

Qualifiers can be constructed by *simple predicates* on the values of an object attribute. A simple predicate is a predicate that involves a single attribute name, a *relational operator*, and a constant. The relational operators are **equals, greater-than, less-than, greater-than-or-equal-to,** and **less-than-or-equal-to.** An example of a simple predicate involving the attribute *author* is *author* **equals** "John Smith".

Qualifiers can be combined with standard Boolean operators to form other compound qualifiers. The Boolean operators are **and**, **or**, **minus**, and **not**. An example of a compound qualifier is *author* **equals** "John Smith" **or** *date* **greater-than** 1/23/82.

Class specifiers can also be combined with standard Boolean operators to form other class specifiers. The Boolean operator that are available here are the same operators that were mentioned above. If the class of *Short-Reports* is defined by some class specifier, CS_1, and the class of *Good-Reports* is defined by some class specifier, CS_2, then the class of *Good-Short-Reports* is defined by the class specifier, CS_1 **and** CS_2. The other Boolean operations work in the same intuitive manner.

3.4.10.2 Class Specifiers with Containment

Class specifiers can make use of the internal structure of the content of an object. In this way, classes can be created based on properties of the components of an object as well as on properties of an object itself. For example, we might want to specify the class of objects that are Reports that contain chapters that were written by "John Smith". This request involves the information that is stored in the content aspect of the object schema. It requires that the system be able to access the components that are members of the class *Chapters.*

It is also possible to form a class specifier that involves the containment relationship in reverse. That is, we might be interested in all chapters that are contained in reports that were published in the last week. In this case, the containment relationship is also used; however, the objects of interest are the components of some object class. In order to process this type of request, it must be possible to obtain all the parent objects of a given object. We will discuss how this might be implemented in a later chapter.

3.4.10.3 Class Specifiers with Versions

It is also possible to construct a class specifier that restricts the target set to include only those objects whose version set has certain specified properties. This is a qualifier based on the history aspect of objects.

The first way in which we can qualify conceptual objects is by forming simple and compound predicates on the values of conceptual object attributes. A conceptual object attribute is one that applies to the conceptual object as a whole. These attributes can be qualified in the same way that a simple attribute can. For example, assume that the class *Conceptual-Reports* has an attribute called *Person-responsible-for* that has a value that is the person that is ultimately responsible for making decisions about a given report. We might like to specify the subclass of the class *Conceptual-Reports* as all conceptual reports that have the head of the Office Automation Group as the person responsible for them.

We can also form conditions on the membership of a version set. It is possible to create class specifiers that restrict a class to be all conceptual objects of a given type that have all, some or a fixed number of versions that satisfy a given qualifier. For example, we might want all conceptual reports that have some versions that were written by "John Smith". This is equivalent to:

```
Restrict [Conceptual [Reports],
          lambda (cr)
            For-some [Versions-of [cr],
                      lambda (v) written-by (v) = "John Smith"]]
```

We can also ask for all conceptual reports that have all versions that were written by John Smith. The condition on the version set can also be quantified with an integer to yield the class of conceptual reports that have three versions that were written by John Smith.

3.4.10.4 Class Specifiers with Control Properties

One of our basic assumptions about object management systems is that the control information ought to be available to help manage the space of all objects. As a result, we have promoted control information to the same level as other object aspects. We, therefore, have primitives in the class specifier mechanism to form new classes of objects on the basis of their control properties.

In the *control* section of an object schema, one defines various classes of users that can have different access rights to objects of that class. Class specifiers can contain qualifiers that describe conditions on this information. A qualifier can produce a class of objects by specifying only those members of some existing class that have *read*, *write*, or *include* access by some user class. For example, a class specifier could generate the class of reports that were produced by the Lab for Computer Science and that can be read by all members of the Office Automation Group.

```
Restrict [Reports,
          lambda (r)
            and [produced-by (r) = Lab for Computer Science,
                 included-in [OA-Group, read-access (r)]]]
```

3.4.10.5 Class Specifiers with State

We provide specialized syntax for specifying a state machine that describes various interesting states that an object can be in and how it can get there. The purpose of having this mechanism is to be able to access all those objects that are in some particular state at some point in time. Class specifiers, therefore, have special facilities for indicating classes of objects based on their current *state*.

There are three basic ways of specifying object state in a class specifier. They are:

1. **State with property.** Since a state is a repository object, it can have attributes of its own. It is, then, possible to specify all objects that are in a state for which one of its attributes has a given value. This is simply like any other object specifier that restricts a class on the value of an attribute, except, here, the class that is being restricted is the class *States.*

2. **Previous or future states.** It is useful to be able to ask for all objects that are some number of states before or after a state with some specified property. The number is given as some integer.

3. **Waiting for state transition.** A state machine for an object can be waiting for some interesting condition. Since transitions are made on the basis of the object's changing from belonging to one class to another, the interesting condition is given as a class specifier (the one that must be satisfied if a transition is to be made).

Chapter Four

Examples of ODM Use

In this Chapter, we present an extended, annotated example of the description of several object types in terms of the Office Data Model. The example will be presented in the context of a scenario of producing a final report for a technical R&D project within a large corporation. The purpose of this example is not to illustrate every aspect or use of an object management system, but, rather, to provide the flavor of its use in the context of a concrete office application. We, then, demonstrate how the Office Data Model can be used to handle some common office situations. The syntax that is used in this chapter is explained in Appendix C and summarized in Appendix D. We have occasionally used underlining that is not a part of the formal syntax to emphasize certain parts of the definitions.

4.1 An Extended Example

In this example, the Office Automation Research and Development team expects to produce many reports to document the progress of projects that they are pursuing. The group has some conventions that it has developed with respect to report production. In their view, a report consists of a set of chapters, and a bibliography. It might also have an optional set of appendices. There exist many other such views about how their information sources are to be structured. The nature of these information structures will be described in detail in the next section.

4.1.1 Document Related Classes

As we mentioned above, the report is a central information source for the Office Automation Group. The schema for the class of *Reports* is given in Figures 4-1 and 4-2.

```
DEFINE CLASS Reports
Definition: Base

CONTENT
Chapters: Non-empty Ordered Set of Chapters
          Read by Researchers, Write by Chapter-Authors
          Change-Trigger
          Constraint-Trigger
Appendices: Ordered Set of Appendices
            Read by Researchers, Write by Researchers
            Change-Trigger
Bibliography: Bibliography
              Read by Home-Group
              Write by Users where Equals(Author(Thisobject))

ATTRIBUTES
Author: Users
        Write by Author(thisobject)
Date-of-Creation: Dates
Length-in-pages (Output-devices): Integers
    Derived by Length-Program
Contains-word (String): True-or-False
    Derived by Word-Containment-Program
Outward-Appearance: Report-Outward-Appearance

HISTORY
Type: Report-Types
Related-Project: Projects
Topic: String
Date-due: Dates
Done?: True-or-False
Send-copy-to: Set of Users
```

Figure 4-1:Schema for the Class of Reports

The class of reports can be specialized to create interesting subclasses. For example, we would like to be able to speak about the class of *Final-Reports*. A final report is a report that has a value of *final* for its *type* attribute. A final report has a number of additional components and attributes, and its control requirements are somewhat different from those for reports in general. The schema for final reports is given in Figure 4-3.

```
CONTROL
OA-Group = Users where Group="Office Automation"
PL-Group = Users where Group="Programming Languages"
Researchers = OA-Group or PL-Group
Home-Group = Users where Group=Group (author (thisobject))
Home-Group-Leader = Users where Group=Group (author (thisobject))
                      and role="Leader"
Group-Leaders = Users where role="Leader"
Chapter-Authors=Users where
                  Member-of [Image [Chapters(thisobject), author]]

Change-Trigger=
  Define Trigger on Change
    Object Set: Set thisobject to matching-object
    User Set: Not (OA-Group)
              Set OffendingUser to matching-object
    Trigger program: Send-Message
                 On: thisobject, OffendingUser

Constraint-Trigger=
  Define Trigger on Change
    Object Set: Restrict where date-of-creation < 1/1/82
                Set thisobject to matching-object
    Trigger program: Error
                 On: "Object is out of date"

Read-Trigger=
  Define Trigger on Read
    Object Set: Set thisobject to matching-object
    Trigger Program: Latest Version of
                       The Trigger-Program where name="send-read-msg-to"
                 on: author (thisobject), thisobject

Control for whole object:
  Read By Researchers
  Write By Home-Group
  Include By Users
  Read-Trigger
Additional Owners: Researchers
Subclasses by: Researchers
```

Figure 4-2:Schema for the Class of Reports (cont.)

A final report contains an additional component called *Group-Publication-List.* Notice that this component is in addition to the other three components defined in Figure 4-1 for reports. These three components are inherited from the parent class. The attributes that are defined in that object schema (i.e., Figure 4-1) are also inherited by objects in the *Final-Reports* class. The control specification and the trigger program that both attached to the *Group-Publication-List* component involve

```
DEFINE CLASS Final-Reports
DEFINITION: Reports where type="final"

CONTENT
Group-Publication-List: Bibliography
  Read by Researchers
  Change-Trigger

ATTRIBUTES
Available-for-comment: True-or-False
  Write by Leader
Read-by: Users

HISTORY
For-year-ending: Year
Research-Groups: Groups

CONTROL
Leader=Users where group="OA" and role="Leader"
Additional Owners: Group-Leaders
Subclasses by: Users where name="Stan"
```

Figure 4-3:Schema for the Final Reports Class

definitions from the *Reports* schema. These definitions are inherited too, and would only have different meaning if another definition for the same name occurred in the *Final-Reports* control aspect.

```
DEFINE CLASS OAReports
Definition: Reports where Member-of [Author(thisobject), OAGroup]

CONTENT
Abstract: Ordered Set of Paragraphs
  Read by Users, Write by Author (thisobject)

CONTROL
Additional Owners: OAGroup
Subclasses by: OAGroup
```

Figure 4-4:Schema for the OAReports Class

The *OAReports* class is also a subclass of the class *Reports*. The control specification in this class indicates that the owners of this class should be the members of the Office Automation (i.e., OA) Group. This is legal since the OAGroup is a subclass of the class *Researchers*. The definition for the class *Researchers* occurs in the control aspect of the class *Reports*. This definition is inherited since it is not specially defined in the context of this class and this class is a subclass of *Reports.* The users who can construct new subclasses of this class are defined to be just the single user named Stan.

The class *Chapters,* the value class for the *Chapters* component of reports, is defined in Figure 4-5. It is another base class like *Reports.* A chapter has two components, both of which are sets. The first one is defined much like the components that we saw in the *Reports* schema. The second component, *Figures,* is defined to have a value set that is the Merge-members (i.e., Union) of the classes *Graphs* and *Texts.*

```
DEFINE CLASS Chapters
DEFINITION: Base

CONTENT
Paragraphs: Set of Paragraphs
Figures: Set of Merge-members(Graphs, Texts)
  Define for connection:
    [Associated-Paragraph: Paragraphs where Part of thisobject]

ATTRIBUTES
Author: Users
  Write by thisauthor
Date-of-Creation: Dates

HISTORY
Number-of-Versions: Integer
Longest-Version: Chapter where version of thisobject

CONTROL
Thisauthor=author(thisobject)
```

Figure 4-5:Schema for the Chapters Class

The *Figures* component is the collection of figures that are to appear in the given chapter. Each figure can be keyed to a given paragraph in the *Paragraphs* component. This association between figures and paragraphs is accomplished by the additional definition that appears below the definition line for the component, *Figures.* There, we find a definition for an attribute named *Associated-paragraph.* This is an attribute of a connection between a Chapter and a Figure. For each figure in the *Figures* set, there is a connection between parent chapter and the contained figures. Each of these connections will have an attribute called *Associated-paragraph* which has as a value the paragraph to which the given figure is keyed.

```
DEFINE CLASS Paragraphs
DEFINITION: Base

CONTENT
Body: Text

ATTRIBUTES
Creator: User
Date-Created: Date
Contains-Word (word: String): True-or-False

HISTORY
```

Figure 4-6:Schema for the Paragraphs Class

The *Paragraphs* class, another base class, has a single component named *Body* which contains the string of text that is the paragraph. The class *Text* is primitive to the system. It has an empty history aspect. This means that the system will maintain paragraphs as conceptual objects (i.e., in version sets) with no additional restrictions or attributes. If the keyword HISTORY were absent from the schema, paragraphs would not be maintained in version sets.

4.1.2 A Graph

The Office Automation R&D Team also makes use of graphs to communicate business information to their monitoring agencies. They will often plot such things as expenditures over time or system cost as a function of memory size. These graphs are stored in the repository, and certain important aspects of them are described in ODM. This makes it possible to retrieve graphs as well as documents and document components with the same retrieval commands.

The schema for the class *Graphs* is given in Figure 4-7. Notice that it has a state machine defined as a part of its history aspect. This state machine has two states, one that represents a state waiting for the graph to be approved for release and another state that is entered when the graph is approved. The attribute, *Graph-Status* has a value one of the two attributes from the machine *Graph-Status-Machine.* The states in this machine are repository objects, each having an attribute called *Condition.* This attribute can have a value of *done* or *pending,* thereby, partitioning the states of the machine into two sets, the pending-states and the done-states. In this case this dichotomy is not very rich because there are only two states. However, in a more complex state machine, it might be interesting to look at states that are either *done* or *pending.*

A graph is made up of axes, curves, and a title. Of these components, axes and curves are distinct repository objects and, therefore, must have object schemas of their own. The schemas for these two classes are given in Figures 4-8 and 4-9. These schemas are rather simple and should be understandable in the context of the preceding examples.

Curves are, further, made up of points which are simply a pair of real numbers. If we change one of these numbers, we will get a different point, so they are defined to be part of the content. They are also very simple objects, and it was not considered necessary to define any other aspects for them. The *Points* schema is given in Figure 4-10.

4.1.3 Some Record-Like Objects

The object management system should be able to deal with objects that behave like ordinary records in the sense of conventional database management. To model objects of this kind, we must only define the *Attributes* aspect in the object schema.

```
DEFINE CLASS Graph
DEFINITION: Base

CONTENT
Axes: Axes
Curves: Set of Curves
Title: String

ATTRIBUTES
Author: Users
Approved?: One of {true, false}
  Write by Leader

HISTORY
Send-to: Users
Processing-state: Graph-Status-Machine

Graph-Status-Machine=
  Define State-Machine
    State1:
      Condition: pending
      When Entering (Graphs where approved?=true)
      Do Send-Msg-To(Leader)
      Goto State2
    State2:
      Condition=done

CONTROL
Leader=Users where group="OA" and role="Leader"
OA-Group=Users where group="OA"

Control for whole object:
Read by OA-Group
Write by OA-Group
```

Figure 4-7:Schema for the Class of Graphs

```
DEFINE CLASS Axes
Definition: Base

CONTENT
X-Start-Point: Real
X-End-Point: Real
X-Tick-Spacing: Real
Y-Start-Point: Real
Y-End-Point: Real
Y-Tick-Spacing: Real

ATTRIBUTES
Creator: Users

HISTORY
Useful-for: Plotting-goals
```

Figure 4-8:Schema for the Class of Axes

An important class that is record-like is the class of *Users*. The system uses this

```
DEFINE CLASS Curves
DEFINITION: Base

CONTENT
Points: Set of Points

ATTRIBUTES
Increasing?: True-or-False
```

Figure 4-9:A Schema for the Class of Curves

```
DEFINE CLASS Points
DEFINITION: Base

CONTENT
X-Value: Real
Y-Value: Real
```

Figure 4-10:A Schema for the Class of Points

```
DEFINE CLASS Users
DEFINITION: Base

ATTRIBUTES
User-name: String
Group: Groups
Role:Roles
System-status: String
Salary: Integer

CONTROL

Read By Group(thisobject)
Write By Users where System-status="Wheel"
```

Figure 4-11:A Schema for the Class of Users

class to manage the control aspect of other object types. It contains an object (i.e., a record) for each user of the system. This class must be present in all object management system implementations, but the attributes of each user record can be configured to meet the special needs of a particular application environment.

The user class for our Office Automation Group example is given in Figure 4-11. In this case, we have defined five attributes that apply to each user. These attributes are used in defining subclasses of *Users* that are to be given certain privileges.

4.2 Example Interactions

Now that we have described the data that comprises this example, we will demonstrate the ways in which the object management will handle some realistic interactions. These interactions include retrievals and modifications of the basic data that has been described previously in this chapter.

4.2.1 Retrievals

The retrieval language that is provided by the class specifier mechanism can be used to specify interesting sets of objects. These set specifications can be issued to the object management system at top-level in order to retrieve some information that is relevant to making some decision. They can also used in the definition of the data itself to describe relevant sets. We have seen examples of this in the definition of user sets and the definition of restriction subclasses as in Figure 4-3.

Let us begin by looking at a very simple retrieval. We can specify the class of all graphs that have only increasing curves. The class specifier that would produce this set is shown in Figure 4-12.

```
Graphs where
  Contains all Curves where
                Increasing?=true
```

Figure 4-12:A Simple Class Specifier

This class specifier contains a condition on the class *Graphs* which contains a second condition on the set of curves in each graph. The first condition is on a given graph. It requires that the second condition be true for **all** the *Curves* components of the graph. The second condition requires that the *increasing?* attribute have a value of *true.*

Now, let us look at a somewhat more complex retrieval. Suppose that we were interested in the set of all reports that have Stan as an author and that contain graphs that are in a "done" processing state. The class specifier given in Figure 4-13 will select exactly those objects. It contains two conditions on the class *Reports.* The first simply restricts *Reports* to those that have a value of "Stan" for their *Author* attribute. The second restricts *Reports* to those that contain a graph that satisfy an additional condition. That condition is that the *processing-state* attribute of the conceptual graph object should have as a value a state that has a value of "done" for its *condition* attribute.

```
Reports where
  author="Stan" and
  contains Graph where
    Processing-state in
      States where condition="done".
```

Figure 4-13:A Class Specifier

Issuing this class specifier to the top-level of the object management system causes the system to construct a set of reports that satisfy the qualifying condition. The user can access members of this set one at a time. Let us assume that the user who issued the request in Figure 4-13 has a value of *Programming-languages* for the *Group* attribute of the corresponding *User* object (i.e., for that user). This means that from the point of view of the *Reports* class, that user is at least a member of the *PL-Group* and *Researchers* control groups. As the user tries to access (i.e., read) a report that was returned by the given class specifier, the system checks to that that operation is allowed. The control aspect of the schema for reports (see Figure 4-2 says that read access is granted to all members of *Researchers.* The access is allowed.

There is also another effect of the user's reading a given report. The author of that report is sent a message that informs him about the fact that the given user is reading his report. This is a result of associating the *Read-Trigger* with report objects. This trigger matches the reading of any report by any user, and will activate a program on the author of the report and the report as arguments. The author is the value of the *author* attribute for the given report. The message is prespecified except for two fields that are filled in with the name of the author and the name of the report.

One could argue that this read trigger would cause an over-production of message traffic. Every time a report is read, some user gets set a message. If this is not the desired behavior, one can redesign the change trigger to be more selective about the class of objects or the class of users that it matches. The object management system gives workstation users the flexibility to design whatever behavior suites their environment.

4.2.2 A Simple Modification Example

As a simple example of a modification, let us suppose that a member of the Office Automation Group decides that a graph is acceptable for publication. This user, therefore, decides to change the *approved?* attribute to **true**. The system checks the authority of this user and discovers that since his *role* attribute is not equal to *leader*, this change request must be denied.

The user, then, looks at the values of the attributes for his corresponding *user* object. There is no problem with this, because the schema indicates that users can read *User* objects for any members of their group, including their own. The value of the *role* attribute is equal to *group-member.* The user, then, tries to change this value so that his access privileges can be expanded. The system prevents this action since one must have *system-status* equal to *wheel* in order to change the value of an attribute of a user.

The user, then, issues the following class-specifier to the top-level object management system:

```
    Users where system-status="wheel"
```

The result of this request is a list of the users who are system wheels. The user can ask one of these users to make the change for him.

4.2.3 A More Complicated Modification Example

The previous example involved the modification of simple attributes. We will now look at an example that involves changing the value of a component of a structured object.

Let us suppose that there is a report R1 that is a member of the class *Conceptual-Reports.* Further, let us assume that the chapter objects Cl, C2, and C3, are all members of the class *Conceptual-Chapters.* The relationship of these objects is shown in Figure 4-14. The symbol O is used to represent a particular version off an object (i.e., a simple object). The symbol lv that labels the two arcs represents the act that those two components o the latest-version o R1 are latest-version references. Suppose that John, a student in the OA-Group, is using the system. He iterates through the members of the class *Conceptual-Reports* in order to find a report that he needs. He finds the report called R1 that seems interesting. He accesses $R1_3$ by invoking the operation *Latest-Version (R1).* He begins reading it which causes the system to retrieve the first chapter by means of *Get-Chapter-n* ($R1_3$, *1).* When this operation is executed, the system first checks the access specifier for the operation *Get-Chapter-n.* Since John is a member of the OA-Group, the operation is performed. This causes an *Evaluate-Reference* to be performed on the reference that is embedded in $R1_3$. Since this is a latest-version reference, the latest version of Cl is produced (i.e., Cl_2). The trigger for *Get-Chapter-n* is then invoked since John is not the author of the report and the type of the report is "controlled". This trigger

```
                {0,0,0} R1
                     / \
                lv /     \ lv
                   /       \
           C1 {0,0}      {0} C2     {0,0} C3
```

Attribute definitions:

```
  Author (R1) = "Stan"
  Type (R1) = "Controlled"
  Date-of-Creation (C1) = 1/18/83
  Date-of-Creation (C2) = 1/23/83

For Reports, Get-Chapter-n Operation has:

  Access Specifier: Users where group = "OA"

  Trigger:
    Objects: Reports where type="Controlled"
    Users: Users where not(name=author(Thisobject))
    Program: Send-Message
      On: Author (Thisobject),
           "Someone is reading your report"
```

Figure 4-14: Environment for Modification Example

causes a message to be sent to the author (i.e., Stan) telling him that someone is reading his report.

John decides to change the report by adding C3 to it as the third chapter. He, therefore, executes the following operation:

```
     Set-Chapter-n (R1_3, 3, Create-Reference (Latest-Version, C3))
```

This inserts a latest version reference into the set of chapters of $R1_3$. No triggers or access specifiers are applicable since this is not a repository operation*. John then adds this new report object to the R1 version set by means of the operation *Add-New-Version(R1, $R1_3$)*. This operation will check the structural constraints to make sure that any component that has changed is a member of the proper value set**. After all this, the final structure for R1 is given in Figure 4-15.

*Components do not have a set operation from the point of view of the repository

**Is the new chapter object a member of the class of *Chapters*?

```
{0, 0, 0, 0}  R1
      / \/|\
     /  /\| \
    /  /  \  \
   {0,0} {0} {0,0}
    C1    C2   C3
```

Figure 4-15: Final State of Conceptual Report

4.3 How to Do Some Useful Things

There are many common situations that can be effectively modeled with the facilities described in the previous chapter. Some of those situations are illustrated in this section. Most of these examples have to do with providing alternate views of the information stored in the repository.

4.3.1 Version-Set Views

Suppose that we have a conceptual object, R, that is a report. R will have several versions, $r_1, \ldots, r_5$. Now suppose that only the first, third, and fifth of these versions is available to the public as published versions of R. These three versions have a value of *published* for their *condition* attribute. To the public, R should appear to have only these three versions. This is a view of the conceptual object, R.

How is this view provided using ODM primitives? The definition of a class for conceptual objects (i.e., objects with history) really creates two parallel classes, the class of simple objects and the class of conceptual objects with elements of their version sets drawn from the first class. For example, the schema for the class, *Reports*, implicitly defines a class for conceptual reports. See the top of figure 4-16.

Moreover, defining a subclass of a class that has a parallel class of conceptual objects will implicitly create a subclass of the class of conceptual objects. Suppose that the subclass is formed by restricting the members of the first class by a predicate, P. The subclass of the conceptual object class, CC, is formed by creating one new conceptual object for each member, x, of CC such that the new conceptual object contains just those versions of x such that P(x) is true. Following this procedure, version sets that have no members are not included in the new class. For example, in figure 4-16, the subclass of *Reports* called *Published-Reports* contains those reports that have a value of *published* for their *condition* attribute. The subclass, *Conceptual-Published-Reports* is formed automatically to contain members of *Conceptual-Reports* with versions for which *condition* is not equal to *published* missing. An example version set is indicated in the figure under the class names.

$$\begin{tikzcd}
\text{Reports} \arrow[r, dashed] \arrow[d, dash, "\text{subclass}"] & \text{Conceptual Reports } \{v1,v2,v3,v4,v5\} \arrow[d, dash, "\text{subclass}"] \\
\text{Published Reports} \arrow[r, dashed] & \text{Conceptual Published Reports } \{v1,v3,v5\}
\end{tikzcd}$$

Figure 4-16: A View of Version Sets Provided by Subclassing

4.3.2 Object Dependencies

It is often the case that a given object should be defined to have pieces that are dependent on a piece of some other object or objects. An example of this phenomenon is a set of comment forms and a summary database for those forms. The forms are filled out by users to indicate their feelings about a particular paper. The summary database is a set of records whose fields have values that are some of the responses on the comment forms. Filling in values in the fields of the form will cause the appropriate fields in the database to be filled in.

In Figure 4-17, the comment form for Smith has caused a record to be created

```
                                  CommentDB
CommentForm                            Name  |   Evaluation
  Regarding: Fred's paper          -------------------------
  Commenter: Smith  -------------> Smith  |   Fair
  Evaluation: Fair                 Jones  |   Good
  Comment: "You should
    emphasize the final point."
```

Figure 4-17:An Example of a Comment Form

in the Comment Database. The fields in this record are derived from the response in the Comment Form.

We model this situation using ODM, by defining the attributes of the records in the Comment Database to be derived attributes with values computed from the corresponding Comment Form. That is to say that there is a derivation program stored in the repository that will be invoked whenever the value of an attribute from a Comment Database record is required. This program will take as an argument the value of the *Name* attribute from the record.

A record in the Comment Database will be created by a trigger program every time a new Comment Form object is entered into the repository.

4.3.3 Handling Outward Appearances

An interesting and important dichotomy exists between two broad classes of objects. One is the *logical objects* that are manipulated by the object editors, and the other is the *outward-appearance objects* that are actually "printed" on the available output devices. The logical objects contain components that make sense in terms of the basic object structure. For example, a report has logical pieces that might include chapters, sections, and paragraphs. The outward appearance of an object contains components that make sense with respect to the constraints of the output medium and current printing conventions. For example, an outward appearance object for a report might contain pages, columns, and lines as components.

Moreover, the sizes of these pieces will be dependent on the physical characteristics of the output device. If we print a report on a line printer, the number of words on a line will be different from the number of words on a line if we were to print it on the xerographic printer.

There is, however, a correspondence between the logical report and the outward appearances for that report. Furthermore, there are outward appearances for each of the components of the report. We feel that it is useful to be able to treat these outward appearances as objects in their own right. One often uses visual (i.e., appearance related) cues to remember and retrieve objects. It is desirable to be able to ask for all reports that have a graph in the upper right hand corner on one of the first five pages. The object management system should be able to maintain this correspondence in some convenient fashion, and we should be able to use class specifiers to specify restrictions of the outward appearances.

Let us assume that reports only have chapters as components. The following schema fragments illustrate the main features of one possible schema for keeping the logical reports and their corresponding outward appearances synchronized.

In this example, changing a chapter in the chapter set of a report will cause a trigger called *Reformat-Trigger* to be invoked. This trigger program will recompute the outward appearance object for the report. This is necessary since any editing to a chapter of a report will potentially require that the entire report be reformatted. Adding or deleting text at one point in a chapter can have globally propagating effects in the formatted version.

Another example of a situation that can be modeled using outward appearance objects can be found in the management of *screens* as objects. A screen is an entity that is very much like a form [65, 66] with which a user interacts with a database.

```
DEFINE CLASS Reports
DEFINITION: Base

CONTENT
Chapters: Ordered Set of Chapters
          Reformat-Trigger

ATTRIBUTES
Outward-Appearance: OWAP-for-Reports

CONTROL
DEFINE TRIGGER on Change
Object Set: All
            Set Matching-object to thechapter
Trigger Program: Reformat-Chapter-Set
             on: Outward-Appearance (thisobject), thechapter

DEFINE CLASS OWAP-for-Reports
DEFINITION: Base

CONTENT
Chapter-OWAPs: Ordered Set of OWAP-for-Chapters

ATTRIBUTES
Logical-Report: Reports

DEFINE CLASS OWAP-for-Chapters
DEFINITION: Base

CONTENT
Pages: Ordered Set of Lines

ATTRIBUTES
Logical-Chapter: Chapters

DEFINE CLASS Lines
DEFINITION: Base

CONTENT
Line: String
```

Figure 4-18: Schema Fragment for Outward Appearance Example

The user actually interacts with an outward appearance of a logical form. The logical form does not contain any information about how the fields are laid out on the screen. This additional information is the domain of the screen outward-appearance objects.

4.3.4 Maintenance of Formatting Databases

In a text formatting system such as SCRIBE [55], information concerning the ways in which various document components are to be formatted are stored in databases. The scribe databases are specially named files containing formatted textual format specifications. It is difficult to make coordinated changes to one of these scribe databases such that the effects of the change are propagated in an understandable manner. It is also very difficult to query these databases in any ad hoc way. For example, one cannot ask to see all types of environment types that use a fixed width font.

An object management system should, however, make the above problems easier to deal with. We should be able to design a set of schemas that can deal with textual objects as well as specialized databases to handle their associated formatting information.

```
DEFINE CLASS Paragraphs
DEFINITION: Base

CONTENT
Body: Text

ATTRIBUTES
Formatting-info: Environment-Format-DBs
  Class-determined

DEFINE CLASS Environment-Format-DBs
DEFINITION: Base

ATTRIBUTES
Left-margin: Number
Right-Margin: Number
Spacing: Number
Type-face: One of {Regular, Bold, Italic}
Font-Family: One of {Helvetica10, TimesRoman12}
Blanklines: One-of {Ignored, Break, Kept, Hinge}
```

Figure 4-19: Schema Fragment for Formatting Database Example

The format database for a given document component is determined by its class membership. In this case, paragraphs that belongs to the class *Paragraphs* will all have a particular *Environment-Format-DB* as the value of their *Formatting-info* attribute. The particular database that applies is determined by the class to which the paragraph belongs.

The names of the attributes are standardized and known to the program, SCRIBE. All the attributes that apply to a given database need not be designated in the schema for the object class. They can be inherited from classes of format databases for which the given class is a member. For example, the *Environment-Format-Databases* class might have a subclass named *Numbered-Environment-Format-Databases* that contain formatting databases that describe numbered environments such as enumerated lists. These databases have another attribute called *Numbering-Style* that indicates whether the enumeration is "numbered" with numbers (1, 2, 3,...) or letters (a, b, c, ...). This class will inherit all attributes from its parent class.

4.3.5 Path-Dependent Attributes

A *path-dependent attribute*, a, for an object, x, is an attribute whose value depends on how the object, x, was accessed. Assume that there exists a predicate, Component-of(x,y), that is true if x is a component of y. Let us say that a path to an object, x_n, is some sequence of objects, x_1, ... ,x_n such that for all i between 1 and n-1, Component-of (x_{i+1}, x_i). The value of the path-dependent attribute, a, of object, x, depends on the path that was followed to arrive at x.

Consider a chapter, C_1, that has a set of paragraphs as components. One of these paragraphs, P, might have a comment attached to it that applies to that paragraph from the point of view of the chapter. It might say, "In my opinion, the tone of this chapter does not really fit with the rest of the chapter." Since another chapter, C_2, could be sharing the paragraph, P, it would not be sensible for someone viewing P from C_2 to see that value of the comment. There might be a different value of the *comment* attribute from the point of view of C_2. Then, where does the comment value logically belong? The example comment does not apply to C_1, and it does not apply to the paragraph, P, in general. It also does not apply to the connection between the chapter, C_1 and the paragraph, P. In this case, the comment is the value of the *comment* attribute of the paragraph, P, from the point of view of C_1 only. The comment, then, is a path-determined attribute.

The way that we would model this situation is with a parameterized attribute that takes a set of objects as a parameter. This set of objects represents the path. In most cases this path will be a single object, the immediate parent object of the given object in the containment hierarchy. In the example, we would define an attribute named *comment* for paragraphs. This attribute takes a single argument which is the containing object (i.e., the chapter object). If we use C_1 as the parameter, we will get the example value used above. If we use C_2, we will get some other result. Notice that with this approach the user is responsible for keeping track of how the object with the parameterized attribute was accessed.

4.3.6 User Interface Profiles

In a user-friendly system, a particular user should be able to customize the interface to various application programs based on that user's own style of interaction. Databases can be used to support this capability. The system could provide a set of reasonable defaults that many users would find satisfactory. Those that require changes would be able to set the values of attributes in the databases to indicate their preferred style of interaction. They could set the attributes directly or there could be a special-purpose interface program that would question them about the available options. Sometimes these options would depend on the characteristics of the user's hardware. For example, if the user is running on a bit-mapped display, an iconic interface would be appropriate, whereas, if the user is running on a character oriented terminal, some form of textual interface would have to be run.

The object management system would store a class of objects called *User-Profiles* that would specify the way in which a given user is to interact with a given application. The current state of these objects would determine the way in which the interface programs would behave. The information in these databases is also potentially available to users and programs via the same ODM interface. Many current interface programs capture this type of information in an idiosyncratic form making it inaccessible from other contexts.

The databases that capture the user profile information can be configured by the workstation designer to match the types of interfaces that are available. Our system does not advocate any particular configuration. An example of a possible design for the class of User-Profiles follows:

```
Define Class User-Profiles
Definition: Base

CONTENT
User: Users
Program: Programs where type="subsystem"

ATTRIBUTES
Style: One of {Iconic, Textual}
Selection: One of {Pointer, Menu}
Prompt-line-position: One of {Top, Bottom}
Resolution: Number
```

Figure 4-20: Schemas for the User-Profile Example

In this example, we have a set of objects that are defined by two pieces of information, a user and a program. Each of these objects defines a user interface protocol for a given user for a given program. Each of these objects has a number of attributes that indicate the way in which that interface is defined. For example, the attribute *Prompt-line-position* indicates whether the system prompts appear at the top or the bottom of the screen.

4.3.7 Connections as Objects

We have previously indicated that objects can have components. A component represents a connection between the contained and the containing object. Objects can have a variety of attributes, each one with an object or set of objects as a value. A component or attribute involves a connection between the given object and the value of that component or attribute. In the simplest model, the connections are a reference to some repository object. Given an object and a component name, we can follow the named connection to another object. Here the connection (i.e., the reference) has no additional structure.

In our object management system, we allow much greater structure to exist for a connection. In fact, we would like to be able to treat some connections as objects in their own right. Connections can have attributes and access rights. It is also possible to retrieve a set of references that satisfy a given condition.

A connection is really a repository object that *references* another repository object. We, therefore, have a class called *References*. This class has a schema that describes the semantics of references in the same way that a schema describes the semantics of any other class of objects. Subclasses of *References* can be created to describe various kinds of references. For example, the class of references that can be connected to chapter objects is a subclass of *References*.

It is possible to get the object to which any reference refers. We call this *dereferencing* the reference. Performing the *Get-Component* or the *Get-Attribute* operations on an object causes the reference that is standing for the component or the attribute to be retrieved. This involves dereferencing the reference. The same sort of behavior should occur if we interpose an object from the class *Connections* between the containing and the contained objects. The two operations mentioned above must be aware of this special object class. When one of these object is invoked, the code for the operation gets the connection object and applies its dereference operation to it automatically. This will produce the proper referent.

Alternatively, it is possible to get the reference to the connection object by means of the *Get-Reference* operation. This reference can be manually dereferenced to yield the connection object. We can then access any attributes of the connection. An attribute of a reference makes some statement about the connection. For example, if we had a report with chapters, there might be a reference type that is defined between reports and chapters. These references have an attribute called *date-connected* which makes a statement about when the given chapter was connected to the given report.

4.4 Advantages of Approach

This section looks at the ways in which the approach to object management that has been presented in this thesis differs from the ways in which one might maintain similar information using more conventional approaches.

Let us look at a comparison of the example given in Section 4.1 with the same application handled by a conventional database approach. We will use the relational data model [13] as our point of reference to illustrate these points. In our examples, relations are designated by the relation name followed by a list of domains (contained in parentheses) for that relation. The primary key of each relation is underlined.

One way to represent the report class that was defined in Section 4.1 is by means of the relations that are given below:

```
Reports (Report-Id, Bibliography, Author,
           Date-of-Creation, Length-in-Pages, Contains-Word,
           Outward-Appearance)
Chapters (Chapter-Id, Author, Date-of-Creation)
Appendices (Appendix-Id, Author, Title)
Paragraphs (Paragraph-Id, Body, Creator, Date-Created, Contains-Word)

Report-Chapters (Report-Id, Chapter-Id)
Report-Appendices (Report-Id, Appendix-Id)
Chapter-Paragraphs (Chapter-Id, Paragraph-Id)
Appendix-Paragraphs (Appendix-Id, Paragraph-Id)
```

The *Reports* relation contains one tuple for each report in the *Reports* class. Each report is assigned a unique identifier (i.e., *Report-Id*) that is used to refer to this report from other contexts. The unique identifier for the bibliography of a report is stored in the *Bibliography* field of a *Report* tuple. The relationship between a report and one of its chapters or a report and one of its appendices is captured by a tuple in the either the *Reports-Chapters* relation or the *Reports-Appendices* relation respectively.

In order to retrieve an entire report including all of its components, we must use a program like the following:

```
Get-Whole-Report (X) <-
  Set-Chapters [Answer,
    For-each-member [
      Restriction [Chapters,
        lambda (ch) Member-of [Chapter-Id (ch),
          Project [
            Restrict [Report-Chapters,
                      lambda (c) Report (c) = Report-Id (X)],
            Chapters]]],
      lambda (ch) Get-Whole-Chapter (ch)]]
  Set-Appendices [Answer,
    For-each-member [
      Restriction [Appendices,
        lambda (ap) Member-of [Appendix-Id (ap),
          Project [
            Restrict [Report-Appendices,
                      lambda (a) Report (a) = Report-Id (X)],
            Appendices]]]
      lambda (ap) Get-Whole-Appendix (ap)]]
  Set-Bibliography [Answer, Bibliography (X)]
  Return (Answer)
```

This is a complex piece of code. Its structure depends on the particular way in which the components of a report are associated with that report. In this case, the chapters and the appendices are associated differently from the bibliography. The chapters and the appendices are linked to their containing report by means of additional relations, while the bibliography is linked by a field within the report tuple. Moreover, notice that this piece of code makes use of two additional programs, *Get-Whole-Chapter* and *Get-Whole-Appendix.* These are equally complex programs that depend on the structure of the relations that represent the components of the chapters and the appendices.

With our view of object content, it is possible to write a single simple program that will return an entire object given its root. This code is given below:

```
Get-Whole-Object (X) <-
  Iterate-Over-Components [X,
    lambda (c) If [Set?(c),
                   Set-Component [X, c,
                     For-each-member [c,
                       lambda (r) Get-Whole-Object (r)]],
                   Set-Component [X, c,
                     Get-Whole-Object (c)]]]
```

This program will work for any object type that has been declared to contain components (i.e., have content). It can be supplied as part of the system code making the notion of the entire content of an object something that is primitive to the object management system. This relieves the user from having to write complicated and idiosyncratic procedures as in the example above.

Since the data model captures the component structure of objects directly, the system can make use of this information to support other parts of its functionality. For example, the inheritance of attributes via the component hierarchy is made possible by this feature. The representation of which subparts of an object are its components makes inheritance by components something that the system can deal with.

How are versions of these objects handled? With relations, one might choose to store a version number in each tuple that represents a component of a conceptual object. This version number will become part of the primary key since it is used to distinguish between instances of a single conceptual report (i.e., reports that have the same *report-id*). In this way, the definition of the *Reports* relation becomes the following:

```
Reports (Report-Id, Version-Number, Bibliography, Author,
          Date-of-Creation, Length-in-Pages, Contains-Word,
          Outward-Appearance)
```

In this view, there is no identifiable entity that corresponds to the version set as a whole. There is, therefore, no place to attach attributes of for the conceptual object. If we want to include conceptual object attributes, we must invent another relation that contains domains for each of these attributes. That relation might look like the following:

```
Conceptual-Reports (Report-Id, Type, Related-Project, Topic,
                    Date-due, Done?, Send-copy-to)
```

This separation into two distinct relations is unnatural. The relationship between these two relations is expressed only in their names.

Also, since there is no object that corresponds to the version set, it is impossible to provide a locking schema that allows only one process to add new versions to a version set while other processes are concurrently reading old versions. There is nothing with which to associate the lock. Locking the *Reports* relation is too course a granularity. This prevents other processes from accessing any reports. Locking any one tuple for a report R does not achieve the desired effect since anyone can add new tuples with a Report-Id equal to the Report-Id for R.

Related to this is the problem of maintaining the integrity of the version numbers. There is nothing to prevent someone from entering a new report tuple that has a version number that is completely out of sequence. Let us suppose that version sets are implemented by a version numbering scheme in our object management system. Since the actual implementation of the version set mechanism is hidden from the user the determination of the version number is localized to one specific module, the operation *Add-New-Version.*

It is also impossible to pass a conceptual object as a parameter to programs. This is useful, for example, for query programs that are going to choose the version of an object that has the fewest components. This type of program takes a conceptual report object as an argument and returns the desired element. The program can rely on the fact that the object that is passed to it can have any of some standard set to version set operations applied to it. In the relational example, if we pass the Report-Id to such a query program as an indicator of the conceptual object, then all relations that can store multiple versions of an object must obey this unique identifier convention.

The class mechanism provides several benefits over the relational approach. First, a given repository object can be a member of many classes. In the relational model, it is impossible to have an object that is a member of more than one collection. A report record can only occur in a single relation (i.e., the *Reports* relation). If there is a representation of the same report object as a tuple in some other relation, that is linked by a common identifier such as the report-id, there is no way to enforce the identity of these two tuples. If one is deleted, the system cannot know to automatically delete the other. There is no real notion of the object itself. Consider the additional relation:

```
    Long-Reports (Report-Id, Version-Number, Number-of-Chapters)
```

This represents the additional attribute *Number-of-Chapters* that is defined for long reports. The correspondence between the original report and a given long report is made by the *Report-Id* and the *Version-Number* attributes since they are the keys of both relations. If a given report is deleted by an application program, that program must know that it must also locate the corresponding long report record (as well ass any other similar qualifying relations) and delete that too.

Unlike the relational model, our model supports the notion of access control at the most primitive level. It is not handled by a system that is build on top of the basic data modeling mechanism. This makes it possible to access this information in much the same way that one would access any other information about objects. We can ask the system to produce all reports that can be read by Smith.

Chapter Five

Architecture

5.1 System Architecture

In this chapter we will explain the basic system modules, their functions, and how they communicate. We will also describe some general approaches to building such a set of modules. The descriptions in this chapter will be high-level; detailed descriptions will be deferred to later discussions.

5.1.1 The System Modules

The design of our object management system involves six identifiable modules. Each module has a well-defined, functional purpose that will be described in some detail in this section. The details on how to build each of these modules will be deferred to a later chapter on implementation.

The six modules are listed in Figure 5-1. The lines that connect the modules indicate a dependence relationship. That is, for two modules connected by a line, the module at a higher level makes use of the abstractions presented by the module at the lower level.

We will now sketch the functionality of these six modules:

1. The **Object Filing System** (OFS) forms the lowest level of the object management system architecture. It is similar to the file system component of a conventional operating system. It has the capability of storing objects archivally. It is much like a laundry in that when a user presents an object to the system, it takes possession of the object and gives the user back a ticket that can be redeemed at some future time for the original object. We call this ticket a *repository key.*

```
           Interfaces
                |
         Class Mechanism (CM)
                |
     Predicate Support (DML)
                |
Individual Object Semantics (IOS)
                |
 Object Storage System (OSS)
                |
  Object Filing System (OFS)
```

Figure 5-1: The Basic System Levels

2. The **Object Storage System** (OSS) adds the view that repository objects are made up of other objects and that conceptual objects consist of a set of versions. One of the functions of the OSS is to break a complex, structured object into its constituent pieces and store them each as separate repository objects. For example, a report is often created using a text editor as if it were a single object. When this report is stored in the repository, the OSS will explode it into its component chapters as well as exploding each of these chapters into its component paragraphs. The complete set of objects (i.e., the paragraphs, the chapters, and the report) are then each stored by the OFS.

   The OSS is also responsible for the maintenance of the version sets that are part of a conceptual objects. Whenever a new version of a conceptual object is created, the OSS places in the proper version set. The OSS will also propagate a given change to version sets of containing objects if the object schemas indicate that this is appropriate.

3. The **Individual Object Semantics** (IOS) module is responsible for maintaining the additional object semantics that is described in an object schema. For example, the attributes of a graph or the triggers that are associated with a chapter set in a report are supported by the IOS.

4. The **Predicate Support** module (DML) is akin to data manipulation languages of conventional database systems (thereby, the acronym, DML). This module provides facilities for processing class specifiers and producing sets of objects that satisfy a given class specifier. It makes use of all the semantic capabilities that are supported by the previous three modules.

5. The **Class Mechanism** module (CM) creates an environment in which all the objects in the repository are partitioned into relevant classes. Whenever a new object is placed in the repository, the CM module must decide to which classes the object belongs. The CM module must also be able to manifest all the members of a given class.

6. The **Interfaces** module is a collection of interface programs to assist the user in searching for relevant information from the repository. There are several interface protocols supported by the system at any point in time. Each interface protocol is suited for a different style of interaction.

Users interact with the system at the level of the interface modules. A system designer, however, will often add things at the lower levels. For example, when adding a new object type to the system, a designer may have to write programs that logically belong to the IOS or the OSS level.

5.1.2 Process-Level Architecture

There are several alternatives to the way in which the object management system is configured within the process space of the workstation.

The first and simplest configuration has the object management system as a set of programs residing within the address space of the application program. In this scheme, each application would have its own copy of the object management software. There would be no communication between the object management modules. This works for repositories that are not shared, but makes it difficult to synchronize and control the activities of multiple workstation users. This solution is only good for true personal computers.

The next configuration has a single object management process that runs in its own address space. Each of the application programs reside in their own process and communicates with the object management process via well-defined communication paths. The nature of the communication along these paths is worth some discussion.

We would like the use of our object management system to be independent of the language in which it is implemented. One should be able to communicate with the object management system from programs that are written in different programming languages. This situation is illustrated in Figure 5-2. In this figure, MDL [20], CLU [40], and PASCAL are all general purpose programming languages. Even though our object management system, ENCORE, is written in MDL, we ultimately do not want to restrict our user base to people who write or use MDL programs. This artificially limits the number of potential users as well as limiting the potential sharing of information between different user groups.

```
   MDL              CLU              PASCAL
 Program          Program           Program
    |                |                 |
    |                |                 |
    -------------------------------------
                     |
                     |
                  Object
               Management------->Disk
                  System
```

Figure 5-2:Example Process Space

In order to accomplish this type of information sharing, we establish communications conventions along the paths of Figure 5-2. That is, the external representation for objects that are transferred between the processes must conform to some standard. We assume that all object types for each programming language environment can be converted to this standard. The standard involves predetermined ASCII character sequences to indicate the relationship of the external data representation to the corresponding part of the ODM schema.

In order to make the translation from the internal representation of the application to the external representation of the communication paths, as modular as possible we adopt a scheme that is very similar to Herlihy [32]. In this scheme, every type that is to be communicated must include a definition for an **encode** and a **decode** operation. **Encode** converts from the internal to the external representation, and **decode** converts from the external representation to the internal representation for the receiving environment. The act of transmitting a value of some type will cause the appropriate encode operation to be invoked. the act of receiving a value will cause the corresponding decode operation to be invoked.

The development of a suitable external representation for ODM objects is beyond the scope of this work. It would resemble the protocol given in Herlihy [32].

An alternative to this scheme is to pass the internal representation and the operation programs to the receiving program. The receiving program could then interact with the object by invoking the operation code that it received with the object. This works if the receiving program exists in an environment in which these operations can execute. For example, the receiving environment might have to be running on a particular processor, or a floating point unit might be required. If the machine code for the operations will not run in the receiving environment, we might be able to pass the operation source code. The source code could then be interpreted on any processor, assuming an interpreter exists in that environment.

We reject this solution because it is potentially very inefficient to transmit large amounts of code back and forth and because it potentially compromises the security and integrity of the data. Since we are giving the receiving program the internal representation, it could perform incorrect transformations to the data. It could also discover more information than it had a right to by rummaging through the data structures.

5.2 Program Architecture

The programs that support the semantics of object types can be supplied by the system as a default implementation, or else it can be supplied by the system designer. This basic approach to system building is sketched in this section.

5.2.1 The Basic System Paradigm

The object schemas provide the high-level, integrated interface to the repository. The complete collection of object schemas makes it possible for users to use objects of disparate types together. The object semantics that is expressed in these schemas is supported by a set of abstract data types [39] that include operations that are implemented in some general-purpose programming language (e.g., in this case, MDL).

How does the mapping between high-level object schemas and the underlying programming types occur? We provide an intermediate level that contains programs that express this mapping. This structure is illustrated in Figure 5-3.

```
     Schemas
        |
Mapping Programs
        |
 MDL Data Types
```

Figure 5-3:Object Mappings

These mapping programs can come from two sources.

1. **The system.** The user can construct a set of schemas for the system to process. The system understands the intended semantics and provides default implementations. These implementations are guaranteed to be correct, but may not be as efficient as possible. This provides a way to get new object types working quickly without additional low-level programming. This capability is useful for designers to test some new ideas or for non-experts to get some specialized application running.

2. **The user.** In this case, the user who is a programmer can create programs that implement the functionality that is expressed in some object schema. This implementation can be tailored to the intended use of the object. It can be as efficient as possible. For a given ODM semantic feature, we provide a specification of what programs must be written and how they should behave. This set of specifications will appear in a later chapter.

   It is often the case, that an expert user who is creating a complex new application, like a text editor, must be allowed to choose arbitrary implementations in order to achieve the performance that is demanded by his user community. For this reason, we feel that it is essential to allow this type of capability for customizing the system. The object types that have been created with user provided implementations should be indistinguishable from objects that have their implementations provided by the system. Any aspect of an object that can be described by ODM should be implementable by a user.

With reference to Figure 5-3, this paradigm allows access from the top down by non-programmers or from the bottom up by computer experts. We find that this approach provides the flexibility that is required in a workstation environment.

Chapter Six

The Program (ENCORE)

This chapter is a description of some of the basic structures that are used in a prototype version of the object management system called ENCORE (Extensible and Natural Common Object REsource). The primitive programming language types and their operations out of which the higher-level object management system facilities are built will be described in this section. We believe that this set of facilities forms a base for supporting the construction of object management systems in general.

6.1 Object Repositories

At the lowest level of the program, there is a module that implements a file system for objects that are maintained by the object management system. The file system provides the interface between the higher-level modules and the disk.

6.1.1 Conventional File Systems

Systems that deal with data objects that must be preserved from one working session to another need some form of archival storage facility. In conventional operating systems, this is normally provided by the file system. At the simplest level, a file is a collection of data that can be stored archivally associated with some unique name. During some future session, the file system can produce that collection of data by presenting it with the previously assigned name.

The structure of the file system is usually very limited and inflexible. A file is limited in terms of the degree of interpretation of the data that it contains. It is either considered to contain some special set of objects (e.g., records) that are obtainable by means of specialized access methods, or else it is considered to be an uninterpreted collection of data. The structures that are available in the file system for describing the data in a file are usually very difficult to customize or change. For example, if the date of last access is not part of a file descriptor, it is impossible to add this additional piece of information without making changes to the system internals.

6.1.2 The Object Filing System

We will now describe the overall characteristics of the file system or *repository*. This description will include the general assumptions and programming techniques that will be used to build the system. Specific facilities that will be built on top of this facility have been described in a previous chapter.

We feel that it is very important in the office workstation environment for the file system, or as we shall call it the *repository*, to take an active part in the management of objects. This requires that the repository have some degree of knowledge about the semantics of the objects that it stores. We also feel that a uniform approach to the description of the objects that are in the repository will be a step toward achieving integration of the subsystems that will be running on the workstation.

The repository is the place in which objects that are created by the applications programs are stored. These objects are data structures whose meaning is defined by the programs that implement their abstract operations. These programs are part of the applications packages. For example, the document editor implements the operations that are available for modifying a document.

At the simplest level, the repository allows an application program to store an object that has been created in an application that is written in some arbitrary programming language. The act of successfully storing the object will return to the program a unique identifier, or *repository-key*, that can be used to retrieve the object at an arbitrary time in the future. We will defer, for now, how this key is remembered from session to session. If a program has a key, it can be guaranteed the ability to get the object which was stored under that key if that object still exists. It will always be the case that nothing else will ever be stored under that key. Given a key, the repository will always produce the original object that was stored under it or an indication that the original object has been deleted.

The workstation environment is characterized by the concurrent manipulation of objects by multiple users. If a user could store an object in the repository, and that object could be modified by another user or program, then a key that at one time referred to a given object may at some later time refer to a modified version of that object. There is no stability of reference; there is no way to guarantee that what I stored yesterday under a key that I now possess has retained its state.

In this system, we view the immutable repository as a place in which stable objects are stored. Once an object is stored there under a particular key, it can never be changed. There is a one to one correspondence between repository-keys and objects in the immutable repository. Therefore, if a user gets a key for an object, then that user can safely assume that that object will not change. This is different from most file systems in which a user can over-write a copy of a given file. It is different from current database system technology in which a user is allowed to change the contents of a record that is referred to by some key.

There are three basic operations on this repository. They are:

1. **Repository-put**: Repository X Object -> Repository-Key.

Repository-put takes a repository and some programming language object as an argument and returns a repository-key as a result. As a side effect, the given object is stored in the given repository.

2. **Repository-get**: Repository X Repository-Key -> Object.
Repository-get takes a repository and a repository-key that was the result of a previous *repository-put* operation and returns the previously stored object as a result. *Repository-get* is the inverse of *Repository-put.*

3. **Repository-delete**: Repository X Repository-Key -> Repository.
Repository-delete takes a repository-key as an argument and removes the correspondence between that key and its associated object. The system must not reassign the key to some other object as the result of another *repository-put* operation. If it did, then a user (i.e. some program) that had a key with the expectation of being able to retrieve a particular object could have an undesirable effect. The original object could be deleted, a new object could be reassigned to the old key, and a *repository-get* operation on the old key would get the new object. The desired behavior would be for the system to alert the user that the old object had been deleted. We will explore how this can be accomplished, and how additional information (e.g., who deleted it) can be maintained.

6.1.3 Mutable Repositories

Changes to objects are still possible within the context of the immutable object repository. The concept of object changes is maintained by another type of object called a *version set.* Version sets were introduced in Chapter 3. Version sets are stored in the mutable repository. A mutable repository is just like an immutable repository except that it has the following operation available:

- **Repository-replace**: Mutable-Repository X Repository-key X Object -> Mutable-Repository.

This operation stores the given object in the repository under the given key. The key must already exist with respect to the repository.

There seems to be no benefit from requiring that there be versions sets for version sets, etc. In any case, this process of constructing multi-level version sets must stop somewhere with some object that can actually change in place. We have chosen to have this occur at the first level of version set. Therefore, all version sets will be in the mutable object repository. The members of these version sets will be members of an immutable object repository.

Some objects do not need to incur the overhead of the version set mechanism and, therefore, should be allowed to change in place. That is, for some objects, it should be possible to change a field or a component without having to create a new repository key. Objects of this type are also stored in a *mutable repository.* Mutable objects do not have a system maintained history aspect.

Often an object contains some mutable and some immutable parts. We allow objects to be fragmented across the two repository types. Some portions of a primitive object like its attributes can change in place. These attributes, then, are stored in a mutable repository. The content of an object is by definition read-only and, therefore, stored in an immutable repository.

6.1.4 Object Explosion

Very often objects are produced and manipulated by their editors in such a way that what appears to the user to be a single object is in reality many objects. When a report is created by the text editor, that report is often a collection of chapters which in turn are a collection of paragraphs.

Creating a single report might produce scores of component objects. The repository will store each of these component objects as distinct entities. This implies that the act of writing a complex object to the repository often consists of *exploding* that object into its constituent pieces.

The *Repository-put* operation on an object X calls an *Explode-object* program which recursively calls *Repository-put* on each of the components of X. This recursion continues until objects are encountered that have no distinct* components. As the recursive calls return, they each write out the object that they were called with and return the repository key of that object. This key is used to construct a reference that is inserted in the next higher level object in place of the original component. This requires that the data structure slot for each component must be able to accommodate objects of the basic type for that component or of type reference.

6.2 Useful Object Types

In this section we will discuss some of the primitive types on which the object management system has been built. We feel that they have use beyond this particular implementation. They could be used to construct other examples of object management systems. These types have been created in the context of our MDL program, although we feel that nothing in their definition depends on MDL.

6.2.1 Databases

Databases have traditionally provided a convenient tool for modeling applications environments by presenting the user with a records-oriented data model. The record is a particular mechanism for organizing a set of statements about objects that are relevant to the application. It is important to note that, in this paradigm, the objects exist outside of the system, and the statements that are

*By distinct components, we mean objects that have existence in the repository apart from their parent. For example, a paragraph contains a component called *body* which contains the text string that is its content.

represented by the record are made to track the state of the object by means of updates performed by the system maintainers. The record is a "shadow" of its counterpart real object.

Database objects are one possible mechanism that we use for storing a collection of statements. Although we use the term database to describe this type of object, as we shall see, we need not view these objects as imposing structures with the overhead associated with modern database management systems.

6.2.2 Database Objects

A database object in its most abstract form is considered to be a set of assertions. An assertion is an expression which is considered to be true. Intuitively, it is a statement about the application. Examples of assertions are *the salary of John is $30K* and *the salary of John is greater than the salary of Jack.* An assertion is an instance of a mathematical relation. A relation is a named set of ordered n-tuples in which the values in each position of the n-tuple are drawn from some domain set. A relation defined over the domains $D_1,...,D_n$ is a subset of the set $D_1 X ... X D_n$. The elements of the tuple are expressions that denote members of the domain sets. The relation specifies a particular relationship among the members of the tuple. In a database, all tuples are assumed to assert something about the application.

Binary relations are very common vehicles for asserting properties of objects. A binary relation involves two value sets. Any binary relation which has a single value set that is a unique identifier for some object will be called a *simple assertion.* In a simple assertion, the relation expresses the value of some attribute for the object that is referred to by the unique identifier. The attribute is designated by the name of the relation. For example, the relation *Salary*(X, Y) asserts that Y is the salary of X, where X is an identifier that uniquely determines some person in the environment of the application. A member of that relation might be the assertion *Salary*(John, $30K). This asserts that the salary of John is $30K. It is the job of the database maintainers to insure that any changes in the real world are reflected by updates to the relevant database assertions.

Since a database is a set of assertions, all operations that are available for sets are also available for databases. For example,**iterate** is a set operation that takes a set and a function as arguments and applies the function to each of the members of the set. It is possible to use iterate for sets of assertions in the same way as for other sets. In this way, an application program can manipulate the assertions in the database by using the standard set interface.

In general, however, using the generalized set operations for complex programs involving databases that contain large numbers of assertions could be very cumbersome. Additional interfaces for databases are needed. An example of such an interface is the records-oriented interface that is used by most modern data processing applications. A typical application might be interested in many simple assertions about a single object. This could be modeled by means of a set of simple assertions, all containing the unique identifier for the object of interest. The record is a way of grouping together all of the simple assertions about a given entity. It is a set of related assertions. This higher-level interface can be built on top of the set-of-assertions interface.

A database type has a set of *transactions* (i.e., additional operations) associated with it. The transactions are used for creating, updating, and querying a database. Each database type will have its own set of transactions. Each transaction would be written using the standard set-of-assertions interface.

Let us look at an example of a simple set-of-assertions database and a single retrieval transaction (i.e., a query). Suppose that the *Salary-Database* is a set of assertions of the form Salary(X,Y) which states that the salary of employee X is Y. Further, suppose there exist two selectors, *name* and *emp-salary*, such that for a given assertion A = Salary(X,Y), name(A) = X and emp-salary(A) = Y. A transaction to retrieve the salary of someone with name N would be written as follows:

```
Transaction (N)=
  [Iterate [Salary-Database,
          lambda (a) If [name(A)=N,
                         Return salary(A)]]
  Return Null-Assertion]
```

This particular transaction program requires, in the worst case, looking at all assertions in the database.

All transactions must have an implementation in terms of the set-of-assertions interface. We, further, allow each transaction to have an optional *specialized implementation*. The specialized implementation is a program that must perform the same action as the transaction that is written in terms of the set-of-assertions interface; however, it is not required to use this interface. It can take advantage of the underlying implementation.

Suppose that our Salary-Database has an associated data structure called a name-index. Applying the operation *Get-Assertion* to the name-index and a given name returns the assertion containing the given name. This is done by using an index structure like perhaps a B-tree. The above transaction would have a specialized implementation as follows:

```
Transaction (N)=
  Salary [Get-Assertion [Name-Index, N]]
```

If using the index is fast, executing this program will also be fast.

To execute a transaction, one must use the procedure:

- **Apply-Transaction**: Transaction X Arg_1 X ... X Arg_n.

This procedure simply uses the normal definition for the transaction (i.e, in terms of the set-of-assertions interface) unless a specialized implementation exists. If such a procedure exists, *Apply-Transaction* uses it instead of the normal one.

One class of databases that is very useful is *pattern-matching databases.* A pattern is an assertion for which one or more of its slots is unbound. A pattern is said to match an assertion if the slots that are bound in the pattern are the the same as the corresponding values in the assertion. For example, the pattern *Salary[X, $30K]* will match the following two assertions: *Salary[John, $30K]* and *Salary [Mary, $30K]* A pattern-matching database is one for which there exists a transaction called *match-pattern* that takes a pattern as an argument and returns all assertions in the database that match that pattern. This behavior matches the behavior of a large class of useful database queries. It is much like a simple QBE retrieval.

It is important to point out that when we use the term database, we do not mean a system that is supported by hundreds of thousands of lines of code like most commercial database systems. We use the term to refer to any data structure and set of operations that implement the set of assertions interface that was described above. The implementation of this interface may be chosen by the database designer to optimally match the expected use of the system. The designer has the full facilities of the programming language available to produce an efficient implementation of any database. Therefore, we do not view databases as requiring unnecessary overhead based on the generality of conventional database system storage structures. If there is going to be a database associated with each conceptual object, accessing a database must be very efficient. The programming language class mechanism will assist us in building database classes with different implementations.

A database contains assertions about objects. These objects can be objects in the real world or they can be other objects in the repository. For real world objects, we have seen how a unique identifier (i.e., the key) must be invented as one of the fields of the assertion to maintain the correspondence. An employee might be identified by his social security number. For repository objects, the assertion can contain a reference object that will serve the same purpose as the key. The reference object might contain the repository key for the referent. Objects of type reference, then, are analogous in the way they are used to the key field of a database record. A reference object is more powerful, however, since the referent is not necessarily fixed. It is possible to make statements about things like *the latest version of the progress report.*

6.2.3 The Use of Databases

Databases have some specialized uses in the object management environment. A number of specialized database types would be predefined in the context of any object management system. There are two uses of these database types:

1. Databases will be useful as a resource for users who have need to build object types that can maintain statement-like information. The most important example of this is the attributes of a repository object. For these users, there will be a few database types available for direct use.

2. Databases will also be used by the object management system to keep track of information that is required for the proper management of repository objects. For example, the concept of a version set that was described earlier may be implemented by a special database type called a *version-set-database.* Below, we describe this further.

As we have just pointed out above, version sets are implemented by databases. These databases will contain information about which objects are new versions of which other objects. Assertions of the form *New version of(object$_1$, object$_2$)* will be stored in this database to indicate that object$_2$ is a new version of object$_1$. This is not the only kind of assertion type that will be stored in the version set database. Any other type of information that is useful in managing the conceptual object could also be stored here. For example, the information about who would like to be notified whenever another user accesses a particular conceptual object might be stored in that object's version set. It would be stored by a set of assertions of the form *Inform on Read Access ["SBZ"].* This information would be used by any operations on a version set that retrieve a member object.

Many other types of information could be stored in the version set objects. An implementation for a particular object type will reflect the choices that were made for that object type. Consider the attribute *author-of* for a conceptual report. This could be implemented directly as a field in the content of the object. On the other hand, if the object does not have this designed into it, or if the designer decides that it is not something that should cause a new version to be generated, then it could be implemented by making it an assertion type to be stored in the version set database. We would then have assertions of the form *Author-of(object, person).*

6.2.4 References

The above approach to storing object components limits their use in shared contexts. For cases in which actual sharing of the same component is necessary, there must be some way to provide access to an entity in the repository from several places. This is accomplished by a reference mechanism that allows an object to contain a virtual pointer to other objects in the file system. This should not be confused with a physical pointer to a location on secondary (or primary) storage. A reference can take many forms.

The reference mechanism requires that there be a way to get the referent from the repository given the reference. It is special type that is distinguished by the fact that it has the operation **evaluate-reference** defined for it. There will be many subtypes of the type *reference*, one for each form or reference that is useful to applications programs. A reference subtype corresponds to a specific way of referring to repository objects.

Each reference type has a slot that is used to store the definition of that type. The definition can be an object of any type but is of a uniform type for all references of a given subtype. The **evaluate-reference** operation for the specific subtype knows how to obtain the referent from the definition.

We implement this with a single MDL type called *reference*. One slot in this data type contains an atom that identifies the reference subtype. This atom has a property named *eval* on its property list. The value of this property is the **evaluate-reference** function for that subtype.

So far, the program-level reference appears to be very much like the ODM-level reference. The lower-level reference has some additional features. These additional features provide general facilities for storing information about the referent with the reference. In this way, we can often make judgments about the referent without having to incur the overhead of dereferencing the reference.

One of these features is the a slot that can be used to store the type of the referent. From this information, for example, we can tell whether or not the object pointed to by a given reference is a paragraph or a graph. We also have the ability to store a list of names for the referent in the reference. These names and the object type can be printed by the system interface at a point at which the user is being asked to determine whether or not a given reference is of interest and should, therefore, be followed. An example of the type of message that could be printed, consider the following: "Here there is a section named section 2 of chapter 1 of my report". This message uses first the type (i.e., section) and the name (i.e., section 2 of chapter 1 of my report).

Another useful part of a reference object is a slot that can hold a database. This database can contain assertions about the referent or about the object inter-object connection that is being expressed by the reference. For example, we could store an assertion of the form *Date-Connected(1/1/83)* indicating that the referent was established on the given date. This assertion could correspond to an attribute of an object at the ODM level, or it could be something that is useful for the system to perform its functions.

As an example, a *name reference* has a definition that contains the name of the object that is referenced. This name would be used by the **evaluate-reference** operation for references of type *name-reference.* This operation will look up the given name in some directory and return the object that is associated with the name in the context of that directory. The directory that is used could be specified as an additional slot in the reference definition, or it could default to the directory that is connected to the current job.

There are two broad categories of references that are of interest. They are:

1. **Constant references** These references always evaluate to the same repository object at any point in time. A constant reference is useful when the intention is to freeze the value of a reference. If one wants the first chapter of a report to be the version of that chapter as it appeared on June 30, then a constant reference would be used.

2. **Variable references** A variable reference can evaluate to different repository objects at different times. There is something about the reference that depends on the current state of the repository. This is useful if one is interested in constructing objects that are in some way automatically kept current. An object containing a variable reference as a component will logically contain different objects as the object or set of objects that satisfy the definition of the reference changes.

We will now give some examples of both of these reference types. These example reference types are all needed in an office workstation environment.

Each reference subtype has a creation operation that, when invoked with the proper arguments, will produce a reference object of the proper type. In the following examples, we will use these operations to illustrate the basic nature of these reference types. The operation name will be followed by an argument list that contains symbols that are taken in pairs; the first element of a pair is the name of a component and the second is the type of that component.

Let us first look at an example of a constant reference. It would be defined by the following:

```
    create-constant-reference (key: repository-key)
```

This will create an object of type reference with a subtype slot that contains the atom *constant-reference* and a definition slot that contains the given *key* the repository key for the object that is referenced. The **evaluate-reference** operation for this type will get the value of the *key* field and invoke the *repository-get* operation on it. The result of this will always be the same object since the repository is guaranteed to maintain a correspondence between a key and an immutable object.

The simplest and perhaps the most useful kind of variable reference is a reference to the latest version of some conceptual object. In this case, the definition would be as follows:

```
    create-latest-version-reference (version-set-key: repository-key)
```

This subtype is created with a single argument which is the repository key of an object of type *version-set*. The **evaluate-reference** operation for this subtype will get the value of the *version-set-key* from the reference definition and will get that version set object from the repository. It will then apply the *latest-version* transaction to the version set database which will produce a reference to an object in the repository. Presumably, this object will be of the type with which that version set is associated. It will then evaluate that reference to obtain the actual latest version object. This referent will change in time as new versions are added to the version set making this a variable reference.

Another example of a variable reference is one involving a query on some database in the file system. Let us suppose that there is a database that contains information about the office automation group's working papers. This database contains statements of the form *WP015 is a working paper*, *WP015 was written by Ilson*, *WP015 is about ETUDE*, and *WP015 is dated November, 1980.* A working paper database also presents a set of transactions as an interface. One of these transactions (i.e. a query) is **find-newest-paper-about-subject** which takes a working paper database and a subject as arguments and returns a single conceptual object, the newest paper about the given subject. The transaction determines this by means of the statements about the date and the subject of the papers. The office automation group might, then, maintain a repository object which is the set of papers that are distributed to people that inquire about the group. This object would contain a set of references, some of which might be variable. An example of one of these variable references is as follows:

```
Create-newest-paper-on-subject-reference
    (paper-database: repository-key, subject: string)
```

This reference definition contains two components, a repository key that corresponds to the working paper database and a string that describes the subject of interest. Evaluation of this reference will cause the **find-newest-paper-about-subject** transaction to be executed with the database and the subject string as arguments. This transaction will return a *latest-version-reference* to a conceptual working paper object. The generic **evaluate-reference** operation will, then, take this reference and apply the **evaluate-reference** operation for *latest-version-references* to it to yield the latest version of the version set. Notice that the latest version is retrieved since the reference that is contained in the paper database is of type *latest-version-reference.*

The manner in which a particular reference is dereferenced depends on the type of the reference. If one wanted the referent of the *Newest-paper-on-subject-reference* to be a keyed reference to the version set that contains the paper, this could be done by the code for the generic **evaluate-reference** for that reference type. It would not evaluate the *latest-version-reference* but rather simply return its definition.

All references should also have a generic **display** operation defined for them. When **display** is applied to a reference, it prints some meaningful description of what this reference is. It might print the name and/or the type of the reference. It might also print selected statements from the reference's associated database. This allows a program to communicate to a user the important aspects of a reference that occurs as a component of some other object.

6.2.5 Alternatives

As objects evolve, there can often be several alternative versions of a conceptual object at any point in time. We observed this phenomenon in our earlier description of ODM. These *alternatives* represent a set of evolving development histories from which the user will choose a final version. These alternatives might have been produced by one or several users. The version set structures should have some means of managing this forking of the normally linear version chain.

One approach is to use a special object type called an *alternative* to construct the data structures that will represent a set of concurrently active versions within the simple linear version history. An alternative has two components. The first is a reference to the original object from which each of the alternatives derive, and the second is a set of version sets. Each of these version sets represents the version history for a single branch of the alternative tree. Alternative objects occur as members of a version set in order to indicate that a branching occurs at that point. If additional objects occur in a version set after an alternative object, then the alternatives represented by the alternative object must have been merged. Since alternatives occur in version sets and version sets are used to indicate the history of each of the choices of an alternative, this process can be repeated recursively to construct arbitrary branching structures.

Let us look at a simple example. Suppose that we have a paper that is being worked on by two co-authors. The paper goes through two revisions to produce p_2 and p_3 from p_1. The version set that represents this development is as follows:

```
{p1, p2, p3}
```

The two authors decide that they would like to each produce their own versions of the paper on their own. They will meet in a few days to discuss the results and merge the new suggestions into a single paper again. Each author creates two new versions of the paper on their own as they edit in their changes. The first author creates p_4 and p_5, while the second author creates p_6 and p_7. When they get together to talk about their results, they collectively create p_8. The version set for the paper will at this point be as follows:

```
{p1, p2, p3, A1, p8}
where
  A1 = {p3,
               {{p4,p5}, {p6, p7}}}
```

A_1 is an alternative object that indicates that there is a fork in the version history. The alternative object has two components. The first is p_3 to indicate that p_3 is the object from which the alternatives are generated, and the second is a pair of version sets, each of which records the history of the two chains of development produced by each of the co-authors.

This would be sufficient if all alternatives that were created at a given time are merged at the same time. If any subset of the active alternatives can be merged, we need another kind of object type to represent the joining of arbitrary paths. Let us call this object type a *merge.* A merge has two components, a set of objects that are latest versions of some alternative path, and a merged object. The set of latest version objects are those objects that are being merged, and the merged object is the new merged version of the set of alternatives. As an example, consider:

$$\{p_1, p_2, p_3, A_1, M_1\}$$

$$\begin{aligned} \text{where } A_1 &= (p_3, \{\{p_4, p_5\}, \{p_6, A_2\}\}) \\ A_2 &= (p_6, \{\{p_7, p_8\}, \{p_9, p_{10}\}\}) \\ M_1 &= (\{p_5, p_8\}, p_{11}) \end{aligned}$$

Here, at one point three alternatives are active. At some point, however, the p_5 and p_8 versions are merged into the p_{11} object. This coalesces two of the three paths. One of the three path still remains as an active alternative. It's latest version object is p_{10}. The overall version set now has two latest versions.

6.2.6 Derivatives

In the definition of version sets, there is no requirement that a new version of an object bear any structural relationship to any of the previous versions. This can, in fact, be useful for cases in which the previous version is obsolete or incorrect and a completely new version of the conceptual object is needed. However, the most common version-creation process is by incremental evolution of one version into another. A user will make a few modifications to an existing object and call the new object the next version.

To take advantage of this observation, we will make a distinction between a *version* and a *derivative.* As we have said before, a *version* is related to its predecessor because a user has stated that it is, but a *derivative* is related to its predecessor because of some structural similarities. A special data structure has been designed to store an object that is a derivative of some existing object. Essentially, the derivative object contains both a pointer to the object from which the new object is derived and the modified components of the new object. The members of a version set will, in most cases, be derivative objects since new versions are most often created by modifying previous versions. It is important to note, however, that this is not a requirement.

There are two main reasons for having derivatives (i.e., change objects). They are:

1. Derivatives are useful to save space in the repository. By storing only those components that have changed, the amount of space that is consumed by the version set mechanism is minimized.

2. Derivatives also provide easy manifestation of what has changed. A typical use of the version set mechanism is to support a user's questions about what has changed between the current version and the previous version. By storing versions as derivatives, this information is easy to manifest since it is maintained directly.

The above discussion has described a derivative as the set of changes that have taken some old object into a newer object. In this model, in order to recover the latest version of an object, by far the most common form of retrieval, the system would potentially have to search the entire version chain to get components that have not been modified. Alternatively, the latest version of the object could be stored in its entirety and the previous version could be stored as a derivative from the most current object. In this way, the penalties of reconstructing versions from derivatives would be incurred when looking at older versions. Both ways of structuring the derivatives within a version set are available to the programmer. It is felt, though, that the technique of storing derivatives from the current version would be the one of choice in most applications.

If the approach of using derivatives that are changes from the original object is taken, it is harder to retrieve the entire latest version object. However, the components that have been modified recently are easy to get. They would be part of the current derivative object (or perhaps the previous one). If one believes that components that have changed recently are the ones that are most likely to change again, this might be the preferred choice. With this approach, the components that have been recently modified, and the skeleton of the rest of the object will be read in first.

6.2.7 Front-end Databases

For some object types, it might be inconvenient to store assertions about individual objects in the version set. Consider an assertion that describes the date on which a given version was created. Further, let us suppose that this information is almost always accessed at the same time that the content of the version is accessed. It might be better to store this assertion in a data structure that was easy to obtain once we have the content of the object in hand. For these cases, we provide the notion of a front-end database. This is a database for each repository object that is associated with the object. This approach has the advantage of grouping all assertions about an object together, thereby, eliminating the need to search a larger collection for all versions of a conceptual object in the version set. The decision concerning the appropriate choice of where to store an assertion will depend on how the objects are going to be accessed. If one tends to access the object assertions only when one accesses the object as a whole, then using a front-end database might make more sense. If many queries (i.e. class specifiers) involving a particular object type are processed that involve some assertion types, then these assertions might better be stored in the version set. That way, the object itself would not have to be retrieved in order to answer the queries.

The above discussion about having a special database for every object in the repository at first sounds inherently very expensive. How can such a proliferation of databases be handled with any kind of efficiency? We have said nothing about how these front-end databases should be implemented. They might be very low overhead items. If the number of assertions in a front-end database is small, the database could become nothing more than a *header* that is attached to the main object. This header would be a small block of storage that would contain the assertions of interest. The front-end database operations would interpret this header to present the proper set-of-assertions interface.

6.2.8 Associators

There is a general need to be able to relate two objects to each other. For example, The content of an object is related to a database that contains the current values of the attributes for that object. The mechanism that we use to accomplish this is an *associator table.*

There is one associator table connected to each repository. An associator table contains a set of *associators.* Each associator contains two repository keys. In the case of the object and the database, one of the keys corresponds to the object and the other corresponds to the database. The following operation:

- **Get-Associated-Object**: Association-Table X Object -> Object.

finds the other object that is associated with the given object. In most cases, this is the attribute database.

6.2.9 Program-Level Schemas

We have already encountered schemas as a means for describing the high-level semantics of object classes. The notion of a schema at the program level is loosely analogous to the higher-level concept. A program-level schema contains much of the information that is contained in the higher-level form, but it also includes information about the implementations of repository operations. In this section when we use the word schema we will be referring to the program-level kind.

An example of a program-level schema is the following:

```
<SETG REPORT-SCHEMA
     <SEQUENCE
      <ASSERTION COMPONENT REPORTS CHAPTERS>
      <ASSERTION COMPONENT REPORTS APPENDICES>
      <ASSERTION COMPONENT REPORTS BIBLIOGRAPHY>
      <ASSERTION COMPONENT-DEF CHAPTERS
                   SET SETS-OF-CHAPTERS>
      <ASSERTION COMPONENT-DEF APPENDICES
                   SET SETS-OF-APPENDICES>
      <ASSERTION COMPONENT-DEF BIBLIOGRAPHY
                   NON-SET BIBLIOGRAPHY>
      <ASSERTION COMPONENT-IMPL CHAPTERS
                   GET-CHAPTERS-IN-REPORT
                   SET-CHAPTERS-IN-REPORT
                   SINGLE-VALUED-COMPONENT>
      <ASSERTION COMPONENT-IMPL APPENDICES
                   GET-APPENDICES-IN-REPORT
                   SET-APPENDICES-IN-REPORT
                   SINGLE-VALUED-COMPONENT>
      <ASSERTION COMPONENT-IMPL BIBLIOGRAPHY
                   GET-BIBLIOGRAPHY-IN-REPORT
                   SET-BIBLIOGRAPHY-IN-REPORT
                   SINGLE-VALUED-COMPONENT>
     <ASSERTION ATTRIBUTE AUTHOR
                  GET-AUTHOR-FROM-REPORT
                  NOT-SETTABLE>
     <ASSERTION ATTRIBUTE DATE-CREATED
                  GET-DATE-CREATED-FROM-REPORT
                  NOT-SETTABLE>
     <ASSERTION ATTRIBUTE HOURS-CHARGED
                  GET-HOURS-CHARGED-FROM-REPORT
                  SET-HOURS-CHARGED-IN-REPORT>>>
```

This example shows the MDL data structure definition of a schema for the class *Reports.* As one can see, it is a set of (i.e., sequence) assertions. Each of the assertions are coded in a special way. The first atom following the atom ASSERTION is the type of the assertion. The remaining fields in each assertion indicate some specialization of the assertion type. For example, the very first assertion in the example sequence states that one component of a report is called chapters.

6.3 Classes

A repository is actually a data structure that is used to access any of the objects that are stored in that repository. One of the crucial parts of that that data structure is a class that we will call the *root class*. The root class is the top class in the object class hierarchy. It contains all objects in the repository; it is normally called *Objects*. All other classes are subclasses of *Objects*.

A class has six subparts. They are:

1. A name. This name must be unique within a repository.

2. A class definition. This is either the atom BASE or a predicate that determines how to derive the members of this class from its parent classes.

3. A schema. A schema at this level is a database object that contains assertions about how the semantics of the members of the class as well as assertions about the programs that implement these semantics.

4. A list of parent classes. The members of the given class must be subsets of the parent classes. The definition subpart applies to the parent classes.

5. A list of subclasses. The subclasses are subsets of the given class.

6. A list of members. The members of the given class are indicated by a list of references to the corresponding repository objects.

The operation *Members-of-Class* applied to a class C produces the set of objects that are the members of C. This could either simply return the list of members if that list is complete, or it could involve a more complex procedure if it is not. For example, each object in the repository might be stored in only the lowest level of each applicable path in the class hierarchy even though it is a member of many other classes. *Members-of-Class* would, then, have to look through all of its subclasses to see if there are any objects stored at the lower level that are not stored at the higher levels.

ENCORE uses a multi-processing scheme for inserting objects into the class hierarchy. There is a *pending list* that has an entry for objects that are still being worked on. When a new object O is inserted into the class hierarchy, O is placed into its base class and an entry is made on the pending list. This entry indicates that O has not been fully placed into all its containing classes. O will be called a *pending list object.*

The existence of a pending list has implications for the implementation of the *Members-of-Class* operation. When this program is run on class C, it must now return all objects on C's list of members as well as any objects on the pending list that might potentially be members of C. The pending list contains a notation about the base class of each of the objects that it contains. There is conceptually one process for each entry on the pending list. When one asks for the objects that are the members of C, one must run to completion all of the processes that could potentially contribute members to C. That is, if the base class that is mentioned in the pending list entry is an ancestor of C, the pending list object could possibly belong to C. Once all of these processes have been run, the list of members of the class will contain all its real members.

Whenever the user is not making use of the workstation CPU by running an application program*, these processes can be run. This approach uses otherwise wasted resources to accomplish work that must eventually be done, but that can in many cases be deferred to a more opportune time. The net result is that the load on the system is smoothed out over time.

*This can occur often whenever the user is thinking about what to do next.

This approach makes adding an object to the class hierarchy very fast by delaying a great deal of the work. The price that must be paid is that for certain classes, the *Members-of-Class* operation will be slower. Notice, however, that after some time has passed since objects have been added to the class hierarchy, the system will become quiescent in a state that is indistinguishable from the state that we would be in if we followed the brute force approach*.

Objects are initially inserted into some base class. The base class into which it is placed is determined by the programming language class of the object. There is a database that contains a set of assertions that state the correspondences between a programming language type and a base class name. The MDL type *Report* is inserted into the base class named *Reports.*

6.4 Standard DBMS Services

In any commercially viable DBMS, there is a level of service that must be offered in to support the typical operations in a way that is not disruptive of business. These services relate to the ways in which concurrently executing transactions are handled. They must be present in order to guarantee that unexpected occurrences in this environment do not cause the data resource to become incorrect** The unexpected events that we will deal with in this section are potentially interfering transactions*** and failures of the basic system components.

*The brute force approach refers to the insertion of an object into all of its appropriate subclasses at the time that the insertion operation is invoked.

**In this context, incorrect is usually taken to mean inconsistent. That is changes to related values should be made reliably to all such values, not just some of them.

***That is, transactions that are both trying to access the same piece of data at the same time, and at least one of these transactions is trying to access the data in order to write it.

The first type of event will be termed *concurrency* and the second will be called *recovery*.

The main thrust of this project has not been to study these two areas. We feel that, at least for the present, the techniques that are used in current practice will suffice. The remainder of this section will place these issues in the context of our object management system.

6.4.1 Concurrency

Concurrency relates to the handling of potentially conflicting transactions. Two transactions are said to conflict if they are both accessing the same piece of information and at least on of them is trying to change it. In the immutable object repository, the only way that the state of the world can change is by making additions to a version set object. This has implications about how the problem of concurrency control is handled in this environment. Since objects cannot be overwritten in the immutable object repository, once a user has access to a set of objects that represents a consistent state, there is no need for that user to worry about someone else's overwriting these objects. The only place that conflicts can occur is in adding new entries to a version set. This means that concurrency control can be localized to the control of who has access (by setting a lock) to the version sets. If a set of transactions or changes are to happen atomically, then several version set objects will have to be locked. Notice that the lock on a version set does not have to exclude all users from using it. Previous versions can be read without interfering with the change that is being made.

For the mutable repository, there is no version set discipline imposed by the system. Concurrency control, however, is still an issue. In this case, we use some conventional concurrency control mechanism as embodied in current database technology. This would be some scheme like 2-Phase Locking [18, 24] involving read locks and write locks at the repository object level. In these schemes, read locks may be shared by several readers, and a writer must wait on a read lock.

We feel that the issues that arise from concurrent access to shared resources in an office environment extend beyond the traditional problems of guaranteeing serializability of transactions. An object management system should also address other problems that arise in an environment of concurrent access involving the cooperative sharing of resources among several co-workers. This includes facilities for keeping track of which objects have changed and in which ways since a user has last seen it. A given user might want to be notified about such changes. Of course, the user should have some means of defining exactly what is meant by a *change* for a given object or object class.

This type of change control information would be stored in the version set object or the front-end databases. The transactions (i.e., operations) for these databases will be sensitive to their content and will issue the proper actions when a change is detected. These actions should be user configurable.

6.4.2 Recovery

Recovery relates to the ability of the system to recover the database itself from various forms of catastrophic failure. It is crucial that there be some mechanism to restore the database to some state that is known to be consistent whenever something has happened to make the state of the database suspect. Without consistent information most organizations cannot conduct their business. The technical area of recovery management is something that has received a great deal of attention in commercial database implementations.* There are several classes of potential causes for failure. They are:

1. Transaction failures. The individual user programs can fail by attempting to do something that is illegal. An example of this would be an arithmetic overflow or an attempt to divide by zero. When this happens the system must abort the transaction.

2. System failures. The general system facilities can fail in a way that causes all current transactions to be affected. An example of this would be a software problem or a CPU failure. In this case, the database itself remains intact.

3. Media failures. It is possible for the physical device that stores the database to fail in such a way that the damage to the database occurs. An example of this type of failure is a disk head crash.

In order to achieve recoverable behavior from a failure-prone system, one must rely on redundancy. It must be possible to reconstruct data values from other values that are stored redundantly somewhere else in the system. The generally accepted techniques all involve some version of *checkpointing* and *transaction logs.* A checkpoint is a dump of the database (or a part of the database). It provides a point of reference at which a stable state of the database has been captured. A transaction log is a record of all transactions that have been executed on the database since some time in the past. For each transaction, the log stores an indication of the data value along with a "before image" and an "after image" for that value.

With this mechanism in place, transactions can be rolled forward or rolled backward to some consistent state. We can use a scheme similar to this in the object management system. A checkpoint is basically the same for our purposes. The log

*It is estimated that over half of the code in IMS is devoted to recovery and concurrency issues. In IBM's System R product, roughly ten percent of the code is for recovery.

contains entries for each repository operation the is performed. Each entry contains:

1. A transaction-id.

2. An indication of the repository operation.

3. A before image for the value that changed.

4. An after image for the value that changed.

A transaction that is interrupted because of failure can be rolled back to a consistent state by applying the before images.

6.5 Memory Use

For large structured objects, it is desirable to be able to read in pieces of the structure without having to incur the overhead of first having to read the whole object. For example, if one is interested in looking at a small part of a 1000 page proposal, it would be unfortunate to have to read the entire 1000 page of text from the disk and search through it to the proper point. The file system should have knowledge about the structure of the object such that if a user (or a program) specifies a single low-level component of an object (e.g., paragraph 6 of section 3 of chapter 2 of FinalReport) for retrieval, essentially, only that component need be read.

This is accomplished in ENCORE by making each of the low-level components of an object into separate repository entities, each with its own repository key. The lower-level objects can be linked together to form hierarchical objects that contain references to their descendants. The way in which an object type should have its components exploded into separate objects can be specified in the object schema definition for that object type. This information is stored in a schema database to be used by the repository-write operation.

Editing a reference that occurs within an object can cause the original reference to be superseded. Consider a reference to the chapter with the largest number of pages. This is a variable reference that will refer to a particular chapter at any given point in time. If this chapter is retrieved via the reference and then edited, the intent of the user is probably to supersede the original referent with a new object, the result of the edit. There is no guarantee that object will still satisfy the condition of the variable reference. Therefore, a constant reference to the new object will be substituted for the original reference.

Another issue that relates to the use of memory and to objects that contain references is the ability to know what objects in main memory have been changed and, therefore, need to be written out to the repository. We will adopt the use of an additional object type that we will call a *virtual key.* A virtual key will contain at least two slots, one for the real repository key and another for the actual object that is read from the repository. The invariant that must hold for this object type is that if the key slot is empty, then the object that is contained in the object slot has been changed. This means that the first change to the object must delete the repository object that is in the key slot.

We will also provide type specific *virtual-read* and *virtual-write* operations. These operations will be used to access an object from the repository. They will know about the structure of that object type and will take care of reading any associated structures (e.g., indices) for an object type.

Chapter Seven

Repository Design

This chapter covers issues involved in the design of a repository for a given application environment. We begin by presenting a methodology for a system designer who is approaching the design of a workstation information resource. The methodology provides a set of guidelines to assist in the design process. It is not a magic formula. Instead, it is an application of common sense practices that has a good chance at leading to a reasonable first cut design. This procedure must be mitigated by the designers good sense and knowledge of the application.

In previous chapters, we have presented the details of two general topics: 1. a data model (i.e., ODM) and language (i.e., ODL) for expressing high-level object semantics and 2. programming structures that could be used as implementations for a schema that is written in ODL. In the second half of this chapter, we discuss some techniques for putting these previous two topics together. We introduce the reader to our basic approach to the problem of mapping high-level semantic structures to a set of underlying programs. In other words, we are interested in ways of mapping concepts in the object schema into particular implementations.

7.1 A Design Methodology

The steps in the creation of an office workstation repository are analogous to the steps that would be followed in the design of any database. There has been much literature recently on the subject of logical database design for data processing applications [12]. Our methodology is good for designs that include data processing-style as well as office-style applications and data.

This methodology is not intended to be followed linearly from beginning to end. As we shall see, there are places in the procedure that require iteration. The result of this procedure is a set of object schemas that describe the characteristics of the repository. The examples that are used in the discussion of the methodology are all drawn from the design of a document-oriented repository, much like the one in the extended example in Chapter 4. The basic steps in our repository design methodology are as follows:

1. The designer prepares an initial list of types of objects (i.e., classes) that are interesting to the applications at the present time. There is no need to worry about redundancy since a given object may appear in many classes. The name of each class should be chosen carefully to be suggestive of the function of that class. If the name does not convey a precise enough characterization of the class, the designer should also generate a short textual description.

   Examples of classes that might appear on an initial class list are REPORTS, REPORT-OUTWARD-APPEARANCES (representations of reports in terms of physical characteristics such as pages, columns, and lines), LONG-REPORTS (reports that have an outward appearance of more than twenty pages on the line printer), INTERNAL-REPORTS, PUBLISHED-REPORTS, GRAPHS, CALENDARS, OA-GROUP (the members of the office automation group), PROJECTS (the projects that researchers are engaged in), GOV-PROJECTS (projects supported by government agencies), and PRIV-PROJECTS (projects paid for by the private sector).

2. The list that was developed above is examined to determine which classes on the list are subclasses of other classes on the list. Often, in doing this, the designer will discover that some subclasses on the class list suggest other similar subclasses that were initially missed. These new classes are added.

   In the above list, the class LONG-REPORTS is a subclass of REPORTS. The realization of this relationship might cause the designer to realize that the class SHORT-REPORTS is also of interest.

3. The classes on the list are determined to be base or non-base classes. This is easy to determine from the previous two steps. Any classes on the class list that are not subclasses of any other class will be base classes unless the designer discovers that some additional classes have been omitted. For example, REPORTS is a base class while LONG-REPORTS is not.

4. The base classes are determined to contain internal or external objects. If the members of a class have editable content, then they will be internal objects, otherwise, they will likely be external objects. It is possible to have a read-only internal object. The designer must decide whether the read only content is an object that is stored in the machine or whether it is a symbol (i.e., a key) that stands for something that is outside of the context of the machine.

   An example of an internal object class is the class REPORTS or the class GRAPHS. The basic stuff* of which reports and graphs are made are objects in the repository. An example of an external object class is the class USERS or the class PROJECTS. A user is a person that is not physically a part of the repository, while a project is an abstract concept that cannot be completely captured in the world of computers.

5. The base classes are determined to contain user implemented or system default implemented objects. A user implemented class is one for which some system programmer (possibly the designer) determines the underlying data structures that will be used to support it. This is a decision of policy and resources and has little to do with semantics. It is, however, an important distinction to make at this level since it will impact the activities of the next level of design.

   If there exists a special editor for textual objects on the workstation, it is likely that textual objects will require a specialized implementation. The class REPORTS, then, might fall into the category of a user implemented class. The class of PROJECTS might be an information resource that is accessed infrequently and, therefore, can tolerate the implementation chosen for it by the object management system.

*In the case of reports, this stuff is in the form of chapters and appendices, while in the case of graphs, it is in the form of axes and curves.

6. Any class that contains editable internal objects is indicated to be a pair of classes: the class containing primitive objects and the class containing conceptual objects whose versions are drawn from the class of primitive objects. This is because the version sets are objects, too.

   For example, the objects in the class REPORTS are versions of the conceptual objects in the class CONCEPTUAL-REPORTS. The class conceptual reports was not listed in the original class list. It is added now, and the steps from step one to this point are repeated for it.

7. The class list is scanned to determine if any of the members of the list are abstractions or aggregations of other members of the list. These classes are often of interest because they might have attributes of their own.

8. The designer begins to add detail to the class list. Any base classes that are supported by internal objects are examined to determine the next level of structure of the object content. These objects have an editable content that could be composed of other objects in the repository. For example, a member of REPORTS is composed of chapters, appendices, and a bibliography. Each of these components is an object or set of objects in the repository.

9. For each component of this content, the component characteristics* and a value class are determined. If the value class for a component is not already a member of the class list and the designer determines that it is of general interest to the application, then the class is added to the list and all the steps from step one to here are repeated for that class.

   The names of the components of a report are Chapters, Appendices, and Bibliography. The Chapters component is determined to have the characteristics Set-valued, Editable, Ordered, and Possibly-empty. The value set for this component is the class CHAPTERS. This class was not included in the initial class list, therefore, it is included now and all previous steps are repeated for it.

*These are single-valued/set-valued, editable/uneditable, ordered/unordered, and non-empty/possibly-empty

10. For each class, the designer determines what attributes are required by the application. These attributes can be attributes for the members of the class or attributes of the class itself. Member attributes should be assigned to the lowest possible class in the class hierarchy for which that attribute is applicable. For example, suppose there is a class called *Reports* and a class called *Long Reports* that is a subclass of *Reports.* Further, suppose that only long reports have cost of publication recorded. Then, the attribute *Cost-of-Publication* should be associated with *Long-Reports* only and not with *Reports.* Attribute association should be as specific as possible.

11. Any of these attributes whose values can be derived from other information in the repository is indicated as such. The method by which the attribute value can be derived is also recorded. As an example, consider the attribute *number-of-chapters* for a report. The value of this attribute is dependent on the content of the report. The value can be derived by counting the number of chapters that make up the component named Chapters.

12. The designer determines the characteristics* and the value classes for these attributes. If a value class is not already in the class list, it is added and the previous steps are repeated for that class. The value class should also be as specific as possible. Even though it is correct to say that a very high-level parent class is the value class for an attribute, indicating a lower-level class as the value class provides the system with more information.

    For example, the *outward-appearance* attribute for members of the class GRAPHS has a value in the class GRAPH-OUTWARD-APPEARANCES which was omitted from our original list. This attribute is single-valued, derived, and mutable. The attribute *related-projects* is set-valued, unordered, settable, and mutable. It has a value class of PROJECTS.

13. For each attribute, the designer must decide if that attribute can be inherited through the *part-of* (i.e., the component) hierarchy. If it is

*The characteristics for attributes are settable/derived, set-valued/single-valued, ordered/unordered, and mutable/immutable determined that it can, the extent of this inheritance is defined. For example, the *Author* attribute of a report can be inherited by all components of that report as long as the component is a member of the class DOCUMENT-COMPONENT. DOCUMENT-COMPONENTS are the union of all document component classes like CHAPTERS, SECTIONS, and PARAGRAPHS. This new class is also added to the class list.

14. For each component and each attribute determined above, a set of class specifiers is invented, one for each restricted operation involving the given subpiece. The easiest way to do this is to make a list of the possible operations that can be performed on a component or an attribute (See Appendix A). For each attribute and each component, ask who can perform each of the operations. In many cases, the result will be either "anyone can perform that operation" or "the user class U can perform most all the operations" thereby simplifying the process.

    For example, the *bibliography* component of a report can have the **get-component** operation performed on it by anyone who is in the same research group as the author of the report. New versions of a report object that contain a changed bibliography can only be added by the author of the report. (See the example in Figure 4-1).

15. For each component and each attribute determined above, a set of appropriate triggers is invented. For many of the components and attributes, there will be no triggers. Again, for each of the possible operations that can be performed on each of the components and attributes (use the same list that was used in the previous step), the designer asks whether any special side-effect is in order. For any operation for which a trigger is required, the trigger pattern is written down by using class specifiers on the class of *Users* and the class of objects at hand. If the side effect is one of a standard library of programs (e.g., Send-Message), then the trigger is written by referring to this program. If it is not, a description of the program is written down and added to the list of programs that must be written by the designer or an applications programmer.

    The example in Figure 4-1 shows an example of a trigger called *Change-Trigger* that is associated with both the *Chapters* and the *Appendices* components. The specification for this trigger was created when the designer realized that it was necessary to send a message to an unauthorized user telling him that he is not allowed to change these parts of a report object.

16. Derivation specifications are now determined for each of the subclasses determined in step two. This involves providing detail for the interclass connections. Some of the subclasses will be Restrictions of their parent classes, others will be formed by operations such as Merge-Members. The class LONG-REPORTS is specified to be a restriction of the class REPORTS where the number of pages of the reports is greater than twenty.

17. State machines are useful for keeping track of objects that have progressed to some point of significance. For each class, the designer should think about the normal progression of states might be in the life of a typical member of that class. Once these states have been articulated, the designer tries to write down a state machine that captures the order in which the states can occur. A characterization of the properties that define an objects transition from one state to an other is written next to each state change. These properties are expressed in terms of the schema descriptions of the object class. If the current schema does not provide enough semantic detail, new attributes might have to be invented.

    Many of the classes in an office environment might have two principal states: one that indicates that the object is still under development and another that indicates that it finished and ready for release. In this case, the transition between these two states could occur when the value of the *approved?* attribute becomes equal to **true**. If this attribute was not defined previously, it is defined at this point.

18. The designer now has a complete class structure diagram and a notion of how the components of a given object type are put together. He, then, asks himself if altering any component of an object should cause a new version of the parent object to be created. If the answer is yes, any condition on the component that must be satisfied is recorded in the history aspect of the object schema.

7.2 The Mapping Level

The object management system cannot impose a default implementation for objects on the designers of the applications that manipulate these objects. The design of the object management system should not effect the implementation choices of other systems programmers. The designers of workstation applications can make the best implementation decisions about their own programs. The only requirement for them is that if they want the objects that their programs create to be handled by the object management system, they must supply some programs that provide the proper interface to their objects.

This requirement drives the next level of repository design. The designer must write* a set of programs for each class that contains objects with specialized implementations. These programs provide the interface that is specified in the schemas. For example, the content of an object is structured as a set of components that are each available via a name. Components can be retrieved (i.e. read) by name and, therefore, there must be some program that can treat the components of an object as a set with names as selectors. This corresponds to the **Get-Component** operation. If the content is defined for some class C that has a specialized implementation, then someone must insure that this operation is provided.

It is of no consequence to the object management system interface how these operations are implemented. It is the job of the subsystem designers and implementors to write the code for these two programs in terms of the implementation that they have chosen. Suppose the designer has chosen to

*It is more often the case that the designer of the overall repository does not write these programs. Instead, the designer will enlist the services of the individual subsystem implementors. A list of programs that must be written by them as well as a specification of how these are to behave will facilitate this process. The designer, then, will provide such a list.

implement a report in terms of a list pairs, the first element of which is the index and the second element of which is the value. The document editing system designer would then write the put and get operations to search the list for the correct slot. The designer would have to provide programs that map each feature of the schema-level definition to the implementation that has been chosen for the object.

It is important to realize that if the application designers' conceptualization of an object type is too far askew from the facilities provided by the object management system, it might not be possible to interface the two. This could occur for an object like a digitized photograph that is simply a large collection of gray-scale intensities. In this case, it might be difficult to model the object as a hierarchical collection of components. However, as more structure is imposed on pictures, and individual items in the scene are identified, one begins to see ways in which it could be handled. The scene might consist of physical objects which might each have a name, dimensions, and a location in space.

It is not desirable to require all classes of user of the object management system to write programs that map their definitions into structures. It should also be possible to define a new class of object that does not need a specialized implementation. We, therefore, would like to have a default implementation for each schema concept. When a default is chosen, it might not be the most efficient one possible, but it is guaranteed to work. This is useful for getting applications working quickly.

One approach to managing these mapping programs would be to have a database that stored assertions such as *Implementation-for-type (T, F, P).* The meaning of this assertion is that for an object of type T, the schema feature F is realized by program P. In our implementation of ENCORE, these assertions about implementation are merged into the schema database in which the features for a particular class are defined (See Section 6.2.9). This mechanism is completely invisible to the object management system interface. It is an implementation choice.

The last part of our design methodology helps a designer prepare the list of programs that either he or an application builder will write. This process is guided by the set of tests that are contained in Appendix B. The way in which this Appendix is used will be described in the rest of this Section.

Appendix B contains a procedure that indicates what to do with each feature that is defined in an object schema. The best way to illustrate how it is used is to present an example. Consider the following schema:

```
Define Class Chapters
Definition: Base

CONTENT
Intro-Paragraphs: Ordered Set of Paragraphs
Sections: Ordered Set of Sections

ATTRIBUTES
Author: User
Date-Created: Date
```

Since this schema has a defined content, the procedure tells us that there must be an **Iterate-Over-Components** operation defined. This procedure will apply a given function first to the set of introductory paragraphs and then to the set of sections.

For each of the two components, there must be a procedure to test if it is a member of its value set. Therefore, there must be a procedure that indicates whether or not the introductory paragraphs are really a set of paragraphs and a procedure that indicates whether or not the sections are indeed a set of sections. These procedures might be called **Set-of-Paragraphs?** and **Set-of-Sections?**, respectively.

Since both of the components are set-valued and ordered they must both have an **Iterate-Over-Component** and a **Get-Component-n** operation. These might be called **Iterate-Over-Intro-Paragraphs-of-Chapter**, **Get-Intro-Chapter-n-of-Chapter**, **Iterate-Over-Sections-of Chapter**, and **Get-Section-n-of-Chapter**.

The two attributes, *author* and *date-created*, are single valued and settable. There must, therefore, be operations called **Get-Author**, **Set-Author**, **Get-Date-Created**, and **Set-Date-Created**.

This example is admittedly simple, but it gives the flavor for how the methodology in the Appendix works. The other aspects of the methodology should be self-explanatory.

Chapter Eight

Summary and Future Directions

Our work on object management systems has focused on the capabilities of an information management system in an office environment. We have demonstrated the ways in which this type of system is substantively different from conventional database management systems. Our major goal has been to provide rich definitional tools for describing objects such that an application program (or office worker) can manifest the set of objects that are required to perform some task.

8.1 Meeting the Goals

We have described the general approach to object management that has been be adopted in this work. In this chapter, we will describe how this approach satisfies the goals listed in Section 2.4.1 on page 32. This recap of the broad problem areas and our approaches to their solutions is intended to focus more sharply the large number of techniques that were delineated in the previous pages. We feel that the following list will close the circle by tying the analysis of office information system needs in the first chapter with the technical approach of our object management system design.

The goals that were listed in the first chapter were:

1. **Handle different kinds of data.** The ability to use a common language for describing many different types of objects goes a long way toward allowing users to treat data uniformly. The capability for mapping the constructs of the high-level object description language onto programmer chosen implementation structures also encourages a

common view of data. The operations for manipulation of data in terms of the high level model will work the same for all object types that have been specified in terms of the high-level data model.

2. **Support incremental and segmented development.** The process of writing schemas in our object management system paradigm does not require that someone create all database classes at one time. It is possible to create new classes at any point in the life of the system. Moreover, we have made it possible for a user who is creating a new class to make use of existing structures and constraints. The attribute inheritance mechanism makes it possible to create a new class that is exactly like some older class with some additions. The access specifier mechanism and the value classes for components and attributes all require that the definition of new class based on an older class at least maintain the restrictions that are defined at the higher level. THe new class can make these constraints tighter but it cannot loosen them. In this way it is impossible for someone to inadvertently (or otherwise) subvert the intentions of another user.

3. **Support application development.** The process of workstation application development is served by the fact that the object management system can support the creation of the *working set* of objects. The fact that it also presents a single uniform language for referring to objects of different types makes the use of the workstation for office applications more tractable. Also, the ability to define objects as having structure that is automatically derived from other objects simplifies application development. Many of the chores that would have to be handled by the individual applications can be handled by the object management system.

    a. **Specification of Classes of Objects.** Object class specification is possible with the *class-specifiers* that were discussed earlier. The class specifier provides a powerful means of grouping objects based on common properties. These properties can be defined in terms of any of the object semantics that is captured in the schema definition language. This includes things like the history or control aspects. One can therefore create a class of all objects that can be modified by Smith but were created by Jones.

    b. **Perform object maintenance.** The trigger mechanism provides the hooks to achieve this. The ability to generate calls to arbitrary programs whenever a change occurs (i.e., an object is added to a version set) gives the system the power to perform many functions that are related to maintenance of information across object changes.

4. **Manage an environment characterized by change.** The system that was outlined above has many facilities for dealing with objects that are changing.

   a. **Side effects to change.** The trigger specifications that are a part of the control aspect of an object allow a user to perform arbitrary actions when a new object is added to a version set for that object. For the immutable repository this is the only way that objects can change.

   b. **History of change.** The version set, derivatives, and alternatives are all designed to help workstation users deal with objects that are evolving over time. These capabilities allow for the efficient representation of the way in which new versions of objects relate to each other. The control mechanisms for defining triggers are another facility that enhances a users ability to manage the continual change that occurs in a workstation environment.

5. **Control of the use of objects.** The above discussion has outlined different ways in which the constraints can be placed on the usage of objects.

   a. **Object structure specification.** The content of an object can be specified as a set of components each of which is drawn from some class. The ability to describe arbitrary subclasses with class specifiers allows a user to restrict the legal values of a component to be a member of any class that can be described with a class specifier.

   b. **Object control.** The control aspect of an object class definition describes the way an instance of that class will behave when users try to access it in various ways. The object might deny access to some users or send a message to another user. The exact behavior is user configurable. The information on which this behavior rests is accessible as a part of a class specification (or query). This allows a user to easily find all objects that behave in a particular way.

6. **Flexible implementation choices.** Our object management system allows system builders to use any implementation choice that they feel is best for their needs as long as the behavior that they would like to display at the higher level is describable in terms of the required operational interface. The only requirement is that the system builder produce the set of programs that implement the operational interface for the new class of objects.

7. **Office object semantics.** The high-level object description language has been designed with an eye toward the kind of semantics that can be observed for many types of objects that are used in an office setting.

   a. **Object hierarchies.** Office objects tend to be constructed from other objects. The content aspect of an object is specifically for the purpose of expressing this type of object hierarchy.

   b. **Alternative views of an object.** The ability to define an object's contents as being dependent on some other object or objects allows us to have several views of the information that is stored in the repository.

8. **Relationships among objects.** The attribute aspect of a class specification allows a user to express arbitrary relationships between a given object and any other object in the repository. Some system supported relationships between objects have also been explained. For example, the *version-of* and the *component-of* relationship have a great deal of machinery built into the system to support them.

9. **Effective memory utilization.** The reference mechanism allows one to define objects that do not physically contain their contents. In this way, one only reads in (i.e., dereferences) the parts of an object that are needed by the program.

10. **Generalized information about object types.** We provide general database object types for storing arbitrary assertions about the application or about repository objects. Using a reference object type in

an assertion gives us a way to handle general statements about repository objects.

8.2 Workstation Principles

This project has produced a number of tangible results, a model of data (ODM), and a working prototype system, to name just two. Another less obvious result has been the articulation of a number of general principles for the design of workstation applications. Some of these principles have manifested themselves in the design that we have described earlier in this document and others have been born out of our experience with workstation applications. We feel that these principles have applicability to many areas of workstation application design.

Principle 1: When building workstation programs, one should strive to make use of primary data structures that can be updated with minimal overhead. These data structures are always available but are not necessarily efficient. There should also be auxiliary data structures to support rapid access. In order to make the effect of an update visible to the auxiliary data structures there should be a background process that updates the auxiliary data structures on the basis of changes to the primary data structure. After the initial update to the primary data structure, a process is scheduled to update the auxiliary data structure. At some time in the future, access will be through the auxiliary data structure, and, therefore, efficient.

Principle 2: One should always try to construct workstation programs such that they anticipate the types of operations that might reasonably occur next. The programs can then preprocess the data in a way that eliminates the need to always perform actions only on demand. Sometimes, when a users requests that a certain operation occur, the system might already have performed it.*

Principle 3: User interfaces for workstation applications that involve

*This idea became a principle during a private conversation with Marvin Sirbu.

the selection of objects should incorporate the notion of "browsing"**. A good basis for an interface that browses is our class hierarchy. The user can travel down paths of the class hierarchy inspecting the contents of classes. Each class that is visited represents a narrowing of the domain presented by the previous class. The choice of a path represents a semantic property of interest to the user. It should be possible to back up and move arbitrarily around the class hierarchy.

Principle 4: User interfaces must suit the working style of each user if there is to be wide-spread acceptance of office workstation technology. People tend to have individual preferences concerning how they interact with the basic tools of their job. If we use statistical methods to design user interfaces, we will end up with systems that are best used by average people. If we aspire to systems that will amplify the talents of all office workers, we must understand many different interface techniques and allow each user to customize the interface to each program by selecting options from this set* .

Principle 5: Many information sources that are required by the object management system itself can be stored in the object repository and treated as any other object type. Examples of this type of object include the classes *Users, Programs, Attribute-Derivation-Programs,* and *User-Interface-Profiles.*

8.3 Future Directions

This work has been an initial study into the conceptualization of office databases. It has been an experiment in a new area of data management. We expect that it could form the basis for many future projects that will study each of the aspects of an object management system that we have described in this document in more detail.

**This is like the Xerox notion of a browser [22, 23]

*This could be done by parameterizing user interface profiles that are stored in the repository and accessed whenever a user runs a new program.

8.3.1 Efficient Access Structures

We feel that, given the kinds of logical structures that are presented by an object management system and the patterns of use that they will experience in an office, there is a great deal of potential for the invention of specialized data structures that will optimize this use. This is very much like what has happened over the years in the field of database management systems. It is also reasonable to expect that some of the techniques from traditional database management systems implementations will carry over to the object management system environment.

Studies of how an object management system is used and where the bottlenecks occur would be fruitful as a first step in determining where effort should be expended. It is unclear at present where the maximum benefit can be derived from a new system data structure.

8.3.2 User-friendly Interfaces

In order for an office worker to use an object management system, easy to use interfaces must be made available. The office worker is not prepared to invest large amounts of time learning how to use arcane computer systems. The interface must communicate in terms of metaphors that are already familiar to the typical office worker.

The office data model that has been presented in this work is not intended for computer naive users. We expect that it would have the most utility to an office system designer or office data administrator. There is great utility in making some of the facilities of the system directly available to the users of a workstation. Future projects in the area of interfaces for object management systems could potentially have enormous impact in terms of the ability for all levels of users to have uniform access to the data resource. This relates to the use of an object repository as a part of a decision support system.

8.3.3 Distributed Environments

Most current proposals for office system architectures involve a collection of distributed workstation each with its own local storage facility as well as more centralized file server machines all connected together in some form of network. In this model, data does not reside at a single site. Instead it can be located at various physical sites and can possibly have multiple copies scattered around the network. The application code will be simpler to the extent that the location of this data is transparent to the users of the system.

The distribution of data around the system raises a new set of issues. Can components of objects (e.g., the different chapters of a report) be allowed to reside on different machines? Should repositories (as defined previously) be allowed to straddle multiple machines? If so, how are facilities such as the locking scheme affected? Can one effectively write interface programs for a repository operation on another machine in much the same way that one would write such a program for the single machine case, the only difference being the remote procedure call to a repository server on another machine? How can one guarantee consistency of data across multiple machines? These and other issues form the basis for another line of potential development.

8.3.4 Study of Impacts

Once an operational version of the object management is available that exhibits acceptable performance, we feel that a project that tracked its use in a real-world environment would be of enormous utility. This kind of study would be interested in which features of an object management system were used most often and in what ways. A study of this sort could demonstrate which areas of object management are most vital and which have a less immediate payback.

One could also study the impact of object management systems on the ways in which work gets done. Do people structure their tasks differently if they have the capability to access various types of information by means of a single facility? What things that were impossible to do before become important to the functioning of the office. This would be interesting in order to determine the ultimate niche of object management system technology.

8.3.5 Use in Other Environments

Although this work has been driven by studying office workstation applications, we believe that some of these techniques could be fruitfully applied to other disciplines. For example, many of the data handling problems that occur in the field of CAD/CAM resemble problems that we have encountered in the office. Engineering drawings often go through many revisions. Each of these versions of a drawing is in reality a structured object. The components of a computer circuit schematic, for example, might be the CPU, the memory unit, and the I/O bus. Each of these pieces are also broken down into smaller pieces at the next level of detail.

Program maintenance in any large software project poses a large number of problems related to version control. Software designers need to be able to assemble many modules to produce a consistent overall version of the system. In order to facilitate this process, it would be useful to make assertions about each of these pieces. With such a facility, one could, for example, find the version of the disk driver module that has been checked out by the head of software quality control. This should be very reminiscent of some of the examples that we encountered in the world of office objects (i.e., documents).

We feel that the ideas embodied in our object management system could be applied to any application which is concerned with the perturbation of large structured objects. More of these applications should be ferreted out and their suitability for object management system solution should be determined.

Appendix A.

Summary of ODM Operations

The following summarizes the operations types of modeling primitives that are available in ODM. For each type of primitive object, we list the basic operations that are available for them. A more detailed discussion is available in Section 3.3 from which the following list was excerpted.

Version Sets:

Create-Version-Set: -> Version-set.
Add version: Version-set X Simple Object X [Set] -> Version-set.
Latest-version: Linear-version-set -> Simple-Object.
Latest-version: Branching-version-set -> Set.
Delete version: Version-set X Simple-Object -> Version-set.
Iterate-over-versions: Version set X function.

The Repository:

Store-object: Repository X Object -> (Repository, Repository-Key).
Retrieve-object: Repository X Repository-Key -> Object.
Delete object: Repository X Repository-Key -> Repository.
Modify object: Mutable-Repository X Repository-Key X Object ->
Mutable-Repository.

References:

Create-Reference: Type X Definition -> Reference.
Evaluate-Reference: Reference -> Object or Set of Objects.

Object Content:

=Test?: Object X Object -> {True, False}.
==Test?: Object X Object -> {True, False}.

Create-Repository-Object: Repository X Object -> Repository.
Get-Component: Object -> Object.
Iterate-Over-Components: Object X Function.
Iterate-Over-Component: Object X Function.
Get-Component-n: Object X Integer -> Object or a Set of Objects.

Object Attributes:

Iterate-Over-Attributes: Object X Function.
Iterate-Over-Values: Object X Function.
Iterate-Over-References: Object X Function.
Get-Attribute: Object -> Object.
Get-Reference: Object -> Referece.
Get-Attribute-n: Object X Integer -> Object.
Get-Reference-n: Object X Integer -> Reference.
Set-Attribute: Object X Object -> Object.
Set-Reference: Object X Reference -> Object.
Set-Attribute-n: Object X Integer X Object -> Object.
Set-Reference-n: Object X Integer X Reference -> Object.

Classes:

Create-Class: Repository X Base-Class-Schema -> Repository.
Delete-Class: Repository X Schema -> Repository.
Modify-Class: Repository X Schema -> Repository.
Restrict: Class X Predicate -> Class.
Subset: Class -> Class.
Merge-Members: Class X ... X Class -> Class.
Extract-Common-Members: Class X ... X Class -> Class.
Extract-Missing-Members: Class_1 X Class_2 -> Class_3.
Abstract: Class X Grouping-Program -> Class.
Aggregate: Class -> Class.
Insert-Member: Class X Object -> Class.
Delete-Member: Class X Object -> Class.
Is-a-Member-of?: Class X Object -> {true, false}.
Iterate-Over-Class-Members: Class X Function.

Access Specifiers:

Create-Access-Specifier: Operation X User-Class -> Access-Specifier.

Check-Access: Access-Specifier X User-id -> {true, false}.

Triggers:

Create-Trigger: Operation X User-Class X Object-Class X Program -> Trigger.
Activate-Trigger: Trigger.

Repository Operations:

Invoke: Operation X Object.

Appendix B.

Schema Mapping Methodology

The procedure contained in this appendix serves as a guideline for a system designer who is involved in the process of creating the set of programs that supports the semantics of an object class (as contained in a given object schema). In many cases, these programs will already be available as some of the operations of the abstract data type that has been created to underly the given class. For cases like these, this methodology will provide a checklist to insure that there is a one-to-one correspondence between the high-level schema features and the required program level operations.

For the content of an object:

There must be a program called *Iterate-Over-Components*

Components

For each component of an object's content:

There must be a program that tests to see if this component
is a member of its value set.

If Single-valued -> Get-Component
If Set-valued -> Iterate-Over-Component
If Set-valued and ordered -> Get-Component-n

If component has:
1. Access specifier for Read -> Predicate for all above operations
2. Access specifier for Write -> Predicate for Add-New-Version

If component has triggers, one must write 2 programs for each one.
1. A matching program

2. The trigger program

Attributes

For each attribute of an object:

```
If Single-valued -> Set-Attribute
            -> Get-Attribute
If Set-valued -> Iterate-Over-Attribute
If Set-valued and ordered -> Set-Attribute-n
                  -> Get-Attribute-n
All set operations must check to see if the value to which the attribute is
  being set is a member of its value class.
```

If attribute has:
1. Access specifier for Read -> Predicate for Get-Attribute operations
2. Access specifier for Write -> Predicate for Set-Attribute operations

If attribute has triggers, one must write 2 programs for each one.
1. A matching program
2. The trigger program

```
If an attribute has an extent specification,
  there must be a way of making this information available
  to lower level objects.
    For any object:
      Superiors-That-Could-Contribute-Attributes: Object -> Set of Objects
```

State Machines

```
If a schema definition has a state machine associated with it,
  For each transition in the state machine,
    1. There must be a program that checks to see if
         the transition specification is satisfied.
    2. Triggers that use this program to see if the change has occurred.
       These triggers do not have to be described at the schema level.
```

Version Propagation

For Add-New-Version for the version set of the component there must be

a program that gets parent objects that get new versions.
This might require program to notice when a dependent connection is made.
(i.e., Add-New-Version at the higher-level might notice that such a
connection has been made and record this fact somewhere.)

Appendix C.

ODM Reference Manual

This chapter will serve as a guide for someone who is interested in creating ODM object schemas. It is assumed that the reader is familiar with basic ODM concepts as described in Chapter 3. In a few cases, some of the more central concepts have been recast here in order to facilitate the reading of this manual.

C.1 Basic Definitions

In order to provide a setting for the more detailed discussion of the object management system interface, we will begin with some basic definitions of the most central concepts. These concepts are the basic semantic units for an office information system, and they form the conceptual framework for our object management system. They are useful for describing objects of any type.

C.1.1 Objects

An **object** is any collection of information that is thought of as a conceptually distinct unit. The decision concerning what constitutes the boundaries of an object is a part of the overall database design. For example, the final report of our research group might be considered to be an object. This report is an entity that has identity of its own. Members of the group perform various operations on it such as reading, editing, and mailing copies to colleagues.

In our view, objects can be constructed out of other objects. We call the pieces of an object its **components**, and we call the collection of all its components the **content**. As an example, the final report object might be built out of several chapter objects and an appendix object. Objects can share components. That is, a given object can appear as a component in more than one parent object. In this way, changes to a single object can be reflected in more than one containing object.

An object is also the locus of change. It is the smallest unit of information that can be edited. When one modifies the content of an object, a new version is created, and the correspondence between the old version and the new version is recorded. The content portion of an object, therefore, is read-only.

C.1.2 Classes

The entire collection of objects that the object management system knows about will be called the **repository**. This collection of objects is broken down into smaller sets of objects called **classes**. A class is a meaningful collection of objects that all share some property. Sometimes this shared property can be a complex predicate or sometimes the property can be as simple as the fact that the user considers them to be members of the same class.

Classes do not, in general, exist independently of each other, but rather are related to each other by means of *interclass connections*. For example, one of the most common ways to establish an interclass connection is by defining one class to be a *restriction* of the members of some other class. A restriction class contains those members of the parent class that satisfy a given *restriction predicate*. A restriction class is, therefore a *subclass* of its parent class in that all members of the subclass are also members of the parent. A given object must be a member of at least one class, its **base class**, but it may be a member of arbitrarily many other classes. A base class is a class whose existence does not depend on any other class. Every object in the repository has a base class.

At any point in time, an object will belong to some set of classes. As the conditions upon which class membership is based change, so will the class membership patterns. A given class can have different member objects over time. This is in distinction to most programming language type systems in which an object's type is fixed.

We do not require that the members of a class be of a uniform type. That is, they do not all have to be members of a common base class. In this way, for example, we can make a class imitate the behavior of a file folder. It can accumulate a set of objects of dissimilar type that share some common purpose (e.g., relate to the management of the Office Automation project).

C.1.3 Schemas

A **schema** exists for each class in the repository database. The schema is a set of statements that describes the objects belonging to that class. There are two levels of schema information about members of a class.

1. The schema specifies the high-level semantics of objects that belong to a class. This is the interface that users of the object management system see. A given user will interact with the objects in the repository by making use of the semantic constructs from the various object class schemas. The rest of this appendix deals with the mechanics of creating this level of schema.

2. At the implementation level, the schemas specify the ways in which the high-level semantics of the object class are to be implemented. They provide the mechanism for indicating the mapping between a given semantic feature and a program that implements that feature. This is described more fully in Chapter 5.

A schema is associated with each class in the database, but a schema can also be associated with an individual object. In this case, it describes the high-level semantics of that particular object. This is useful when a given object deviates in important ways from the other members of its class. For example, a class of *final reports* might be defined to have a set of chapters as components. Another user of the system might like to create a final report object which has an additional component called a bibliography. This new component would be defined in the individual schema for the new final report object.

C.1.4 Class Specifiers

A **class specifier** is an expression that evaluates to a repository class. It might be a class that already exists in the repository or it might be a brand new class that is derived from one or several existing repository classes. A class specifier is analogous to a database query in that it selects a set of objects as its result. It consists of a predicate P and a class C. The result of evaluating the class specifier is the set of objects from C that satisfy P. Class specifiers are often used as the method of defining a new, permanent repository class.

C.2 Notation

In the descriptions of the linguistic features of the Office Data Model (ODM) that appear in this document, we will make use of the following conventions:

- Language literals or constants (i.e. reserved words) will appear in lower case letters, sometimes with beginning capitals. (e.g., Define Class)

- Class names will be presented in lower-case with initial capital letters. (e.g., Reports)

- Attribute names and component names will appear in a lower-case type face. (e.g., author-of-report)

- Symbols that represent a set of possible values drawn from some syntactic class are presented completely in upper case (e.g., COMPONENT-DEFINITION). These symbols correspond to the non-terminals in a grammar.

- Curly brackets are placed around items that are optionally present. The items within the brackets may or may not be present. If there is more than one choice for the optional item, the options are separated by semicolons (e.g., {Any; All; INTEGER}. The example indicates that one of the following may optionally appear: the literal Any, the literal All, or an integer.

- Square brackets are used to enclose a list of items of which exactly one must be chosen (e.g., [Upward; Downward]).

- Double curly brackets (i.e., {{ }}) mean that zero or more of the enclosed can appear, appended linearly.

- Double pointed brackets (i.e., << >>) mean one or more of the enclosed can appear, appended linearly.

- If the example is a production (as used in BNF), the left and right sides of the production will be separated by a "<-".

C.3 Class Definitions

Many different types of object can be stored by an object management system. An object can be a member of one or more named collections called *classes.* The objects in a class are homogeneous in the sense that they all share some property or set of properties. One of the principal ways in which two objects can be distinguished from each other is by their membership in different predefined classes. A class definition is a description of the high-level semantics that members of the class share. It also provides a specification for a set of procedures that must exist in order for the class members to be managed by the object management system.

Users of the object management facilities can specify new object classes by means of the following construct:

```
Define Class CLASS-NAME
Definition : CLASS-DEFINITION
{ Content
   CONTENT-SPECIFIER-LIST }
{ Attributes
   ATTRIBUTE-SPECIFIER-LIST }
{ History
   HISTORY-SPECIFIER }
{ Control
   CONTROL-SPECIFIER }
```

The syntactic categories used above (e.g., CONTENT-SPECIFIER-LIST), will be defined later in this document.

Within a repository, each class must have a unique name that is specified in the first line of its definition. The class name is a string of characters of arbitrary length. The characters can be any printing ASCII character except *space.*

The class definition is a predicate that can serve as a membership test for elements of the class. It is either expressed in terms of the class-specifier forms that are describe below, or it is the literal BASE. In the later case, the class is a *base class.* A base class is one that is defined independently of the other classes in the repository. A newly created object becomes a member of a base class because someone or some program tells the system that it should belong. The system can determine the base class of an object by the underlying programming language type of the object.

C.4 Aspects

Four optional definitions, one for each of the potential *aspects* of an object type, complete the definition of an object class. Each of the aspects of an object is a way of grouping together related semantics that objects in the class share. Each of the four aspects that we will investigate in this work displays unique requirements for a linguistic mechanism that adequately expresses the kinds of information that we feel is necessary. We, therefore, have provided distinct sublanguages for each of the class aspects. Not all object types will have all four aspects defined. An object type can be defined to behave like a record type in traditional data processing. This object type would have attributes (i.e., the fields in the record), but no content. Textually, we group, within the schema, the definitions that belong to each aspect.

C.4.1 Content

The content aspect of an object schema describes the physical structure of the objects that belong to the corresponding class. The specific functions that are provided by the content aspect of an object schema are as follows:

1. **Defines the object.** Objects that have editable content do not require an artificial unique identifier (i.e., key) to identify the object. The object here stands for itself. If two objects have the same content, then they both represent the same object.

2. **Defines the containment relationship.** If an object type is defined in terms of a collection of other object types, this is reflected in the content specification. The question of whether a particular report has been edited this week can now be processed by looking at each of the components of the report and restating the question for each of them. This process is continued recursively until we reach only objects whose components are not repository objects. They will be primitive objects instead. In the case of a report, we will continue following components until we encounter objects such as paragraphs whose components are strings (i.e., primitive).

3. **Defines structural constraints.** The content section of an object schema describes the structural form of legal objects. Suppose the content aspect of a report object were defined as follows:
```
Chapters: Set of Chapters
Bibliography: Bibliography
```

This would indicate that a *report* must be a set of *chapter* objects followed by a single *bibliography* object. If a user tried to create a *report* object with a *section* object or with more than one *bibliography*, the object management system would notify the user of the error at the moment when he tried to commit the object to the repository.

4. **Defines characteristics of components.** The characteristics *set-valued/single-valued*, *mutable/immutable*, *derived/settable*, and *ordered/unordered* are indicated in each of the component definitions.

5. **Defines object explosion.** It is possible to indicate which components of an object are to written out to the repository as separate objects and which are not. This is accomplished by placing the word *Repository* before the defining class specification. This means that the object component has its own identity and can, therefore, change without causing all containing objects to change directly.

The *content* aspect is defined directly following the literal **Content** in a class definition. It is specified by a content-specifier that is a list of definitions for each of the *components* of the object type. In the simplest case, each component specification has the following form:

```
COMPONENT-NAME :
 {{ [Mutable; Immutable]; [Ordered; Unordered]; Set Of }}
    CLASS-SPECIFIER
```

The central item here is the class-specifier. Any object that is an instance of this named component must be a member of the specified class. This class need not exist explicitly in the overall repository schema. It might instead be a derivation of some existing class. The legal forms for a class-specifier will be discussed below. If the literal, **Set Of**, is present in front of the class-specifier, then the value of this named component can be a set of objects drawn from the stated class.

C.4.2 Attributes

An attribute is used to make a statement about some object. For example, the salary attribute of an employee can be used to state that the salary of Jones is $20K (i.e., salary(Jones)=$20K). The object to which the attribute attaches can be physically either in the repository (as in the case of a report) or in the application environment outside of the machine (as in the case of an employee). The later case requires that this external object be represented by some surrogate object in the repository.

C.4.2.1 Regular Attributes

The simplest version of an attribute specification is:

```
ATTRIBUTE-NAME: CLASS-SPECIFIER
```

All attributes are defined in the *attributes* aspect of a schema. There is no limit to the number of attributes that can be defined for objects of a given class. The definition for each of the attributes appears in a list of attribute definitions beneath the keyword **Attributes** in the schema definition. Each attribute has a name that is unique within the attribute names for that class. The attribute definition begins with the attribute name, followed by a colon, followed by a value class specification. The value class is the set of objects from which the values of the attribute can legitimately be drawn. The value class is always some repository class. It can be the name of an existing class, or it can be a class specifier (see below) that will dynamically produce a new class of objects. We do not actually need to manifest the members of this class. It is sufficient to be able to test an object's membership in the class.

The data manipulation language includes facilities for getting and setting the value of an attribute. It also includes constructs for adding a new attribute to a schema definition.

C.4.2.2 Derived Attributes

A derived attribute is one whose value is dependent on other information about the object. The form of the dependency is expressed in terms of some program. This program expresses a relationship between the derived attribute and other information about the given object or related objects. An expression like the following expresses how an object is derived.

```
Number-of-pages: Integer
  Derived by (The Derivation-Program where name = "Count-pages")
    on thisobject
```

The attribute *Number-of-pages* is defined like any other attribute by its name, followed by a colon, followed by class specifier that indicates the set of legal values that the attribute can acquire. In this case, *Number-of-pages* can have a value that is drawn from the class of integers. The next line in the specification indicates that the value for this attribute is derived. That is, it depends on the value of other data and, therefore, cannot be manually set to a value. The program that computes the value of this attribute is indicated by the class specifier in parentheses. The parentheses are optional. Programs are repository objects and, therefore, can be manipulated by the facilities of the object management system (e.g., retrieved by class specifiers). The last line specifies the arguments to the derivation program. The name **thisobject** is used throughout this work to refer to the current object. If we are looking at the *number-of-pages* attribute for the thesis object that you are now reading, the value of *thisobject* will be this thesis.

The general form of an attribute derivation specification is:

```
Derived by OBJECT-SPECIFIER on ARGUMENT-LIST
```

where an object specifier is a class specifier that is required to return a single object, and the argument list is a sequence of expressions, separated by commas, each of which evaluates to a repository object or set of objects.

C.4.2.3 Parameterized attributes.

A *parameterized attribute* is declared just like a regular attribute except that it has a list of parameters between the attribute name and the colon. The general form of a parameterized attribute is:

```
    ATTRIBUTE-NAME (PARAMETER-LIST): CLASS-SPECIFIER
```

The parameter list is a list of expressions that each evaluate to a repository object. The list of parameters are arguments that are used by the function that computes the value of the attribute. Simple attributes are functions of one parameter, the object itself. A parameterized attribute is a generalization of that idea by allowing the attribute to be a function of many arguments, only one of which is the object itself. This is very useful in an office system environment. It can be used to allow attribute definitions to vary based on other properties of an object.

The parameters in the parameter list can be optionally given names to describe the purpose of the parameters. These names have no fundamental semantics to the system. The most common example of a parameterized attribute for a report is the following:

```
    Contains-word (word : String) : One of {True, False}
```

This represents a function of two parameters, the report object and a particular word (i.e., string). The result is either **true** or **false.**

C.4.2.4 Attribute Extent Specification

The repository contains many objects, some of which contain other objects as components. This creates an environment in which objects can be connected through long chains to arbitrarily many other objects. It is often useful to be able to describe some portion of this network that emanates from the object in question. For example, from the point of view of a report object, we may wish to indicate all components that are members of the class *Document Components.* This would exclude objects such as graphs or tables that might be connected as components to some report component. The normal reason for needing this type of expression is to indicate the extent of applicability of an attribute for purposes of attribute inheritance.

In order to provide this control over the locus of an object, we provide the *extent expression* construct. It is defined in its most general form by the following:

```
EXTENT-SPECIFIER <-
  [Attribute; Value] Extent
    {Via <<COMPONENT-NAME>> }
    {Is CLASS-SPECIFIER}
    [Through; Upto] CLASS-SPECIFIER
```

Suppose that *component-of(x,y)* is a predicate that is true if object x is a component of object y. That is, object x is defined to be part of the *content* of object y. A *component path* is a sequence of objects, $O_1,...,O_n$, such that for all i greater than or equal to 1 and less than or equal to n, **component-of** (O_{i+1}, O_i).* An extent expression for object O defines a set of objects that lie on some component path beginning with object O.

The choice of the modifier, **Attribute**, indicates that we are defining an extent over the descendants of the given object in the content hierarchy. This is because attributes of objects in a class C can be specified to apply to objects that are attached as components to members of C. If we were to use the modifier **Value**, we would be specifying an extent over the ancestors of the given object. This upward extent reaches up in the content hierarchy to objects that contain the given object in order to supply a value for the attribute that is defined for the lower level object.

The first phrase beginning with the literal *Via* indicates which components of the object are allowed to produce component paths. A list of component names

*__Component-of__ (X,Y) is a predicate that is true if X is a component of Y.

follows the literal. All of these component names must be defined as part of the content of the object schema in which the given extent expression is found. This phrase is only applicable to downward extents since only components and not parent objects have names.

The first class specifier in the above definition is optional. It indicates which object types along a component path are to be included in the extent. For example, if it is *Paragraphs*, then only objects that are members of the class *Paragraphs* will be considered to be part of the extent.

The choice of the modifier, **Through**, in the final phrase of an extent expression indicates that the component paths are defined to be all those component objects that are members of the specified class. As soon as an object is encountered that does not belong to this class, the extent is terminated, and the the object that does not belong is excluded from the result. If the modifier, **upto**, is used, the component paths reach out through all components until an object shows up that is a member of the specified class. This object terminates the extent and is not included in the result.

A simple example of an extent expression is:

```
Downward Extent Is Paragraphs Through Document-Components
```

The set of objects that is being defined by the above expression extends recursively, through components of components of the object, as long as all objects encountered are members of the class *Document-Components*. As soon as an object is found along a given component path that is not a *Document-Component*, the extent is terminated. Of the objects that appear along this component path, only objects that belong to the class *Paragraphs* are selected as members of the *extent*.

C.4.3 History

The history section of an object schema appears between the keyword **History** and the beginning of the next aspect definition. We use the term history since this aspect describes properties of version histories. The history aspect of a schema for class C really describes conceptual objects that are members of another class parallel to C. The parallel class contains collections of objects that are drawn from C. If the given schema defines a class with name T, the history aspect defines properties of the version set objects that are members of a class called *Conceptual-Ts.*

C.4.3.1 History Attributes

An important part of the History aspect is the definition of another set of attributes that pertain to the associated conceptual objects. These attributes follow the rules of construction of the attributes that are defined in the Attributes aspect. They are defined by a list of attribute specifications appended vertically. Each attribute specification contains a name and a value class separated by a comma, as in the following history attribute for a graph:

```
    Related-project: Projects
```

In this example, the attribute *related-projects* is defined to have a a value in the class of *Projects.* This attribute describes the related project for a conceptual object. It makes a statement about the version set object as a whole. That is, the related project does not change from version to version.

C.4.3.2 Parameterized History Attributes

A history attribute can take additional parameters, like any parameterized attribute, as in:

```
    Latest-version-seen (User-name : String): Report
```

The additional parameters for the attribute are specified in a list following the attribute name. The elements of this list are separated by commas and the entire list is enclosed in parenthesis. In this example, the value of the attribute is dependent on the value of the supplied parameter, *User-name.* If user name, U, is used as a parameter, *latest-version-seen* for a given conceptual report will have a value that is the last version of the report that U has read.

It is possible to assign a name to a parameterized history attribute in exactly the same way that one assigns names to parameters of regular attributes. These names are optional. They are specified by prefixing the value class for the parameter with the name followed by a colon, as in the example above.

C.4.3.3 Multiple-valued Attributes

Just as with any other attribute, if the value can be a set of objects, the keyword *Set-of* is inserted before the definition of the value class, as in:

```
Related-papers: Set-of Papers
```

In this example, if this attribute definition appears in the context of a schema for reports, then a given report can have many papers to which it is related.

C.4.3.4 Derived Attributes and Access Specifiers

Each of the attributes in the history aspect can also have a derivation specifier and an access specifier. This can be seen in the following:

```
Longest-version: Report
  Derived by: Number-of-Chapters-Comparator
          on: thisobject
  Read by Users where group="OA"
```

In this case, *Number-of-Chapters-Comparator* is the program that compares a set of reports and selects the one with the largest number of chapters. It should be noticed that the syntax for a derived attribute and the syntax for an access specification are exactly the same in the history aspect as they are in the attributes aspect.

C.4.3.5 State Attributes

In the history aspect, we can define a new type of attribute called a *state attribute.* This type of attribute has a value that is a *state.* A state is a special kind of object that is related to other state objects by means of *transitions.* A transition specifies the requirements for a state attribute to change its value from a current state to some new state. State attributes behave much like conventional finite state machines.

A state attribute is defined by indicating that the value set of an attribute is some machine specification. The name of the machine, then, occupies the position of the class-specifier in the normal attribute specification. We, therefore, get attributes of the form:

```
    Processing-status: Status-machine
```

where the name *Status-machine* corresponds to a state machine that is also defined with in the history aspect of the schema by a machine specification.

A machine specification is a list of state definitions. The heading **Define State Machine** is followed by the name of the machine. This line is then directly followed by the state definitions. Each state definition is a state name, followed by a colon, followed by a class-specifier that is a subclass of the base class of *States.* The states of a machine are repository objects, and, therefore, belong to repository classes. This class specifier in the state definition indicates which states can comprise the states of the given machine.

A state specification contains a number of components. First, there is a list of state attributes and their values for this state. This can be useful to indicate properties of the state (i.e., a *completed* state). Then, a list of transition specifications follows. Each transition corresponds to one of the directed arcs that emanates from a state in a finite state machine. At any point in time, the machine is said to be in one of these states. The machine will be said to *follow* one of the transitions from the current state whenever a pattern associated with one of the transitions is matched.

A transition specification contains three components. They are:

1. **When entering CLASS-SPECIFIER.** This is used to indicate the pattern that must be matched in order to cause the machine to *follow* this transition when it is in the current state.

2. **Do** ACTION. This indicates a side effect that will occur whenever the transition is followed. This component of a transition is optional. If it is not present, the only side effect of *following* the transition is possibly to cause the machine to be in some other state. The action is some program that will be executed as a side-effect. The program is indicated by an *object-specifier.*

3. **Goto** STATE-NAME. This indicates which state will be the new current state of the machine after the transition has been *followed.* The state name must be one of the state names that appears in one of the state specifications in the given machine definition.

As an example of the definition of a state machine, the Status-machine that was alluded to above could be defined as follows:

```
Define State Machine Status-Machine
  State1: States-with-Condition
    Condition: Pending
    When Entering (Reports where publishable?=true)
    Do Add-to-Publication-List
    Goto State2
  State2: States-with-Condition
    Condition: Done
```

In this example, we have defined a machine with two states. Both of these states are drawn from the class *States-with-Condition* that contains states that have the attribute *Condition* defined. The first state, *State1*, has a value of *pending* for its *condition* attribute. This state has one transition that is followed whenever the report object becomes a member of the class (Reports where publishable?=true).

Whenever this transition is followed, the *Add-to-Publication-List* procedure is activated, and the machine enters *State2.*

By convention, we will assume that all machines for a new conceptual object (if any exist) will be initialized to the state that is listed first in the machine definition. This state will be called the machine's *initial state.*

C.4.4 Control

The control aspect of an object schema defines features of an object class that relate to the ways in which members of that class can be used and to the things that should happen when an object of that type is used. The fundamental definitions that support the control aspect are textually contained between the keyword **Control** and the next aspect or object definition. We will call this section of textual definition the *control block.* Unlike other aspects, however, control information can textually pervade all of the other aspect definitions. One specifies general definitions within the control block and uses these general definitions within other aspects.

C.4.4.1 User Group Definitions

The first part of the control block contains the definition of groups of users (i.e., subclasses of the class *Users*) that are useful in specifying control constraints for other aspects. A user group is given a name. That name is used to refer to this user group in other contexts. The name appears to the left of an equal sign and the class specifier that defines the group appears to the right of the equal sign. All of these class specifiers must be qualifications of the class *Users.* An example of a typical user group follows:

```
OAGroup = Users where group="Office Automation"
```

One may define arbitrarily many user groups. A user group that is defined within the control aspect of an object schema is known only within the context of that object schema. If one wants to define user groups that have global existence, one must create schemas for these groups.

C.4.4.2 Trigger Definitions

A trigger is a program that is run at a well-defined time in response to the matching of a *trigger pattern.* A trigger pattern involves three elements: an operation P, a set of users U, and a set of objects B. If any member of U tries to perform P on any member of B, the trigger is activated.

The syntax for a trigger begins with the header line:

```
Define Trigger on P
```

This indicates that the trigger is to be associated with the performance of the operation P. The complete definition of P is determined by the context in which it is used. We can define a trigger to be applicable on *read*, but that does not completely specify the matching operation. *Read* for which objects? If the trigger is attached to the *Chapters* component of reports, the operation for the trigger becomes *read* for a chapter.

The object set is indicated next by a definition of the form:

```
Object Set: B
```

The set B is specified by a class-specifier. This class specifier must be a qualification of the class of objects to which the operation P pertains. For example, if the trigger is attached to the *Chapters* component of reports, the qualification must be on Chapters, since it is really the chapters that are being read. If this object set is not specified (i.e., this part of the trigger definition is absent), the object set defaults to all objects (e.g., all chapters).

The user set is defined via a textual component that begins with the string *User set:* followed by a user class specification (i.e., a class-specifier involving the class *Users*). This is illustrated by the following:

```
User Set: Users where years-of-service > 3
```

Finally, a program is invoked by the trigger when the condition specified by the trigger pattern is satisfied. This program is specified by the preface *Trigger-program:* followed by a specification of the program. In general, this specification is an object-specifier, a class-specifier that must return a single object.

A complete example of the syntax of a trigger definition follows:

```
Define Trigger on Write
  Object Set: Set thisobject to matching-object
  User Set: Not (OA-Group)
            Set OffendingUser to matching-object
  Trigger program: Send-Change-Warning-Message
                On: author of thisobject, OffendingUser
```

This trigger is defined on any **write** operation. If the object type with which it is associated is changed by someone who is not a member of the OA-Group, then a program that sends a message is invoked with the user who is the author of the object and the user who is changing the object as arguments. Presumably, that program will send a message to the author warning him that someone is tampering with his object.

C.4.4.3 Access Specifiers

An access specifier determines who can perform a given operation to an object. Access specifiers are textually associated with the subparts (i.e., the components and the attributes) of an object class definition. One achieves this association by writing the access specifier following the definition of the appropriate component or attribute.

An access specifier contains two essential parts: an operation and a user class. The operation is one of the basic repository operations that are specified in Appendix A. By writing an access specifier following a component or an attribute definition, one establishes a connection between the operation and that subpart of. An access specifier which specifies a read operation and appears in the context of the definition of a *Chapter* component of a report specifies a constraint on who can read the chapters of a report. If the keyword **read** is used, the access specifier refers to any operation that reads a value*. If the keyword **write** is used, the access specifier refers to any of the operations that write (i.e., change) values**.

The second part of an attribute specifier is a class of users that can perform the given operation. This is indicated by a class specifier on the distinguished repository class named *Users*. This class contains one object or each potential user of the system. The class specifier is like a query on this class. It returns a set of users who satisfy the condition part of the class specifier. The class specifier is evaluated each time the operation is invoked.

The form of an access specifier is the operation name, followed by the keyword **by**, followed by the user class specification, as in:

```
Chapters: Ordered Set of Chapters
  Read by Users where years-of-service > 5
```

This schema fragment indicates that members of the *chapters* component can only be read by users who have more than five years of service.

*This might be any of the following operations: *Get-Component, Get-Component-n, Get-attribute, Get-Reference, Get-nth-Attribute-Value,* or *Get-Reference-n.*

**This might be any of the following operations: *Set-attribute, Set-Reference, Set-Component-n,* or *Set-Reference-n.*

C.4.4.4 Attaching Control to Other Aspects

In our model of control, an access specifier or a trigger can be associated with any operation that is available for an object type. For a report that is defined to have a set of chapters as a component, there is an implicit operation of reading a particular chapter from a report. An access specifier and a set of triggers can be attached to this operation for reports. If the operation is ever invoked, the access specification is checked, and the triggers are invoked.

Components and attributes are the basic *subparts* of an object. Let us call that part of the schema that defines a component or an attribute the *subpart definition.* This includes the name of the subpart, the value set, and any properties of the subpart (e.g., set-of). Syntactically, the access specifiers and the trigger names are attached to a given subpart by writing it following the subpart definition. We, therefore, have:

```
Chapters: Set-of Chapters
  Read by OAGroup
  Msg-trigger
```

The first line is the subpart definition; the succeeding two lines are an attached access specifier and trigger name, respectively. By writing them following the definition of the *Chapters* component, they become associated with that component. The access specifier indicates which users can read chapters from the chapter set. The trigger, *Msg-Trigger* will be invoked whenever an operation is invoked on the chapter set. Of course, there is an operation specified as a part of the pattern that must be matched in order for the trigger program to be invoked. If the given operation is not the one specified, the trigger pattern is not matched and the trigger is aborted.

C.4.4.5 Control for Whole Object

The triggers and access specifications that were described above attach to the components and attributes of an object. It is also possible to specify triggers and access specifiers for the repository and for repository objects independent of their subparts. One should also be able to attach triggers and access specifiers to these operations as well as to operations on the subparts.

The operations that can be performed on a repository are **read**, **write**, and **delete** an object. The user may wish to define additional, non-standard operations for repository objects. An example of such an operation is **mail** which sends a copy of an object to a given user. All of the control specifications of this later type are attached to the repository or to individual objects in a section of the control aspect that is prefaced with the string *Control for the whole object:*.

By including under this heading a trigger or an access specifier for one of the three boldface operations listed above, one associates that control specifier with the repository. Access specifiers or triggers for other operations are associated with the members of the class for which the control aspect is defined.

The following illustrates the use of this type of control definition.

```
Control for the Whole Object:
Read by Secretaries
Trigger-X
```

The syntax for a control specifier is the same as in other contexts, and the triggers are indicated by a trigger name, as in other contexts. In this case, objects that belong to the class that is defined by a schema in which the above fragment occurs can be read from the repository by someone who is a member of the class of *secretaries.* If the trigger pattern for *Trigger-X* is matched, its trigger program is invoked. This trigger pattern includes the specification of an operation.

C.5 Class Specifiers

Class specifiers are the basic mechanism for forming predicates for partitioning the space of objects (i.e., the repository) into interesting subsets. The syntax for class specifiers is simply "syntactic sugar" for the functional language that is presented in Appendix E. At any point in the ODM syntax that a class specifier can be used, one may use an equivalent formulation in terms of this functional language.

We will now discuss the syntax of the class specifier mechanism. There are several different forms of class specifier to accommodate the various semantic constructs in the Office Data Model (ODM).

C.5.1 Class Names

The simplest form of class specifier is a class name. This is the name of a class that has been defined in an object schema. The name is the string that follows the keywords *Define Class* in the first line of a schema definition. The name is often descriptive of the content of the class as in:

```
    Long-Chapters
```

This name may be used as a class specifier if it has been previously defined as the name of a class. It might, for example, have been the name of class containing chapters that contain more than 1000 words.

C.5.2 Qualifiers

Beyond this, class specifiers are built up recursively by qualifying existing class specifiers with a clause that we shall call a *qualifier.* The general form of a class specifier is:

```
    CLASS-SPECIFIER Where QUALIFIER
```

Therefore, a class name followed by a qualifier is also a qualifier. Similarly, this new class specifier can be further qualified by a second qualifier, and so forth.

A *qualifier* is a predicate that can be applied to a set of objects (i.e., a class). A qualifier Q is applied to a set S by applying the qualifier to each member of S and then forming the set of all objects x in S for which Q(x) is **true** (i.e., {x in S | Q(x)}).

C.5.2.1 Qualifiers on Attributes

The simplest form of qualifier presents some condition on the value of an attribute. An example of this type of qualifier in the context of a class specifier is:

```
Reports where (author = "Smith")
```

The expression in parenthesis is the qualifier. It is used here to select those members of the class *Reports* that have a value of "Smith" for their *author* attribute. The qualifier in this example compares the value of the attribute with a constant. Qualifiers can be made more complex by comparing their value with attributes of other objects.

```
Reports where (due-date >
  Due-date of The Report where name = "1981-Progress-Report")
```

In this case, the expression on the second line evaluates to a single value, the due date of the 1981 Progress report. This value is compared with the due date of other reports within the qualifier (i.e., the expression in parenthesis). By nesting expressions within expressions, as in the example, we can build up more complicated qualifiers.

C.5.2.2 Qualifiers with Containment

One of the reasons for distinguishing between ordinary attributes and object components is to provide a semantic base that will allow one to retrieve objects based on properties of their components. Qualifiers with containment are a direct result of this goal.

A qualifier can specify that all, some, or an integral number of components are members of some other class specifier, as in:

```
Reports where contains some (chapters where creation-date < 1/1/81)
```
The expression in parenthesis is another class specifier. The overall expression evaluates to the reports that have at least one component which is a chapter that has been created before January 1, 1981.

The indicators *all*, *some*, or an integer follow the keyword *contains* in a qualifier with containment. These indicators are followed by a class specifier or the keyword *components*. A class specifier in this position indicates that all, some or an integral number of components of an object that satisfies the qualifier must belong to the specified class. The word components indicates that there is no further qualification on the components. This is useful if there is no qualification at this level, but there is further qualification in terms of the version sets (See below).

C.5.2.3 Qualifiers with Versions

When the word *components* is used, it is often qualified by a *version specifier.* A version specifier expresses a condition on the members of a version set. It begins with the keyword *with*, followed by one of the indicators: *all*, *some*, or an integer, followed by the keyword *versions.* We, therefore, have:

```
1. with all versions
2. with some versions
3. with INTEGER versions
```

This prefix to the version specifier is followed by the keyword *where* followed by a qualifier. This qualifier must match the quantification of the versions from the prefix. As an example, we have:

```
with all versions where (author="Stan")
```

which matches a conceptual object that has a value of *Stan* for the *author* attribute of each of its versions. The parentheses are optional. They are used here to emphasize that the last part of this expression is simply another qualifier. Any qualifier is legal in this position.

It is often useful to be able to indicate a subsequence of the version history for which the conceptual object was in a particular state. This is accomplished by using a *state descriptor.* There are three basic forms of state descriptor. They are described in the next section. They are used by appending the keywords *with versions while in* followed by a state descriptor to a version specifier. For example,

```
with 3 versions where author="Stan"
  with versions while in STATE-DESCRIPTOR
```

would match conceptual objects that have three versions with author="Stan" and that are all in some state such that the State-descriptor is satisfied.

C.5.2.4 Qualifiers with State

State machines that are specified as a part of an object definition can be used to retrieve repository objects. An object's state is used to qualify a class of objects by means of a state-descriptor. There are three basic types of state descriptor. They are:

1. **State with STATE-PROPERTY RELOP VALUE.** This qualifies an object to be in a state that has an attribute that is related by the given relational operator (RELOP) to the specified value.

2. **State INTEGER States [Before; After] STATE-DESCRIPTOR.** This qualifies an object to be in a state that is some number of states before or after a specified state. In other words, within the given finite state machine, there exists a path of length INTEGER between the current state of the given object and a state that matches the STATE-DESCRIPTOR.

3. **State Waiting [For CLASS-SPECIFIER$_1$] [to Become CLASS-SPECIFIER$_2$].** This qualifies an object to be in a state that was arrived at by entering CLASS-SPECIFIER$_1$ and is waiting to make a transition by entering CLASS-SPECIFIER$_2$. Either of these two class specifiers can be missing. This requires that one be able to reconstruct the path that was used to arrive at the current state (at least one state back).

This state specifier mechanism is intended to capture the most commonly used uses of the state mechanism for conceptual objects. The state specifier is the optional last clause on a version specifier.

The state specifier can also be used as a simple qualifier by using the following construct:

```
Object in STATE-DESCRIPTOR
```

This leads to class specifiers like:

```
Reports where Object in
               State Waiting to Become
               Reports where status="approved"
```

This class specifier matches reports that are in a state that has a transition emanating from it that will be taken if the object becomes a member of the class Reports where status="approved". In other words, it is looking for all reports that are waiting to become approved.

C.5.2.5 Qualifiers with Control Specification

In ODM, the information about object control is available to the the class specifier mechanism. This is different from the conventional approach in that normally, this information is only available to the database system or the operating system. We feel that this information is important for understanding the full dimensionality of objects and the ways in which they are used.

A qualifier involving access control information can be formed in the following ways:

```
OPERATION By USER-CLASS
```

or

```
OPERATION [Granted-to;
          Denied-to] USER-CLASS
          via COMPONENT-NAME of CLASS-SPECIFIER
```

The OPERATION is any member of the class *Operations* which contains programs that are the fundamental operations on the abstract data types that provide the implementations for the members of repository classes. For most objects and subparts of these objects, the set of legal operations will include a basic **read** and **write** operation.

The most common form of this qualifier is the one that uses the keyword *by.* This qualifies the access specifications for the object itself. An example of this type of qualifier is contained in:

```
Graphs where (Read by Users where group="OA")
```

This indicates those graphs that can be read by users who are in the OA group. The expression in parentheses is a qualifier on an access specification.

As we have described earlier (See Chapter 3), an object schema for a class C can grant rights to objects that are connected to members of C. In order to form a qualifier based on this information, we use the *Granted-to* form described above. For example,

```
Chapters where Read Granted-to Users where role="secretary"
                  via Chapters of Reports
```

This finds all chapters for which secretaries have been granted read access through the chapters component of reports.

Appendix D.

Formal Syntax of ODM

This appendix contains a formal grammar of the syntax of the Office Data Model (ODM). An informal description of this language and motivation for its design were presented in Chapter 3. In this appendix, its syntax is described in a form that resembles BNF with the following metasyntactic conventions:

- The left and right sides of a production are separated by a "<-".

- Symbol in all capital letters is a syntactic category.

- { } means the enclosed item is optional.

- [ ] encloses a set of items, one of which must appear. The choices are separated by semicolons (i.e., ";"). When used with "{ }", one of the choices may appear optionally.

- {{ }} means zero or more of the enclosed can appear, appended linearly.

- << >> means one or more of the enclosed can appear, appended linearly.

- " " means that the enclosed character or characters are to be interpreted literally. This is used, to indicate that some of the metasyntactic characters that are described in this list are to be included as a part of the ODM language.

- * * encloses a textual description of some syntactic category. This textual description is an informal explanation. It is used in places in which the meaning is clear.

The meaning of these constructs is explained in Appendix C from the point of view of a user who wants to write some ODM schemas. Here, we merely summarize the syntax of the language. The complete language syntax follows.

```
CLASS SCHEMA <-
  Define Class CLASS-NAME
  Definition: CLASS-DEFINITION
  Content
    {{ CONTENT-SPECIFIER }}
  Attributes
    ATTRIBUTE-SPECIFIER-LIST
  History
    HISTORY-SPECIFIER
  Control
    {{ CONTROL-SPECIFIER }}

CLASS DEFINITION <-
  CLASS-SPECIFIER | Base

CONTENT-SPECIFIER <-
  {COMPONENT-NAME : COMPONENT-SPECIFIER
      EXTENT-SPECIFIER}

COMPONENT-SPECIFIER <-
  {{Ordered; Non-Empty; Repository}}
  [Set of {Repository} CLASS-SPECIFIER;
    CLASS-SPECIFIER]
        ACCESS-SPECIFIER-LIST
        TRIGGER-SPECIFIER-LIST

EXTENT-SPECIFIER <-
  [Attribute; Value] Extent
    { Via << COMPONENT-NAME >> }
    Is { CLASS-SPECIFIER }
    [Through; Upto] CLASS-SPECIFIER

ATTRIBUTE-SPECIFIER-LIST
  Default for attributes:
    ACCESS-SPECIFICATION-LIST
    TRIGGER-SPECIFICATION-LIST
  {{ ATTRIBUTE-SPECIFIER }}

ATTRIBUTE-SPECIFIER <-
  [SIMPLE-ATTRIBUTE-SPECIFIER;
   PARAMETERIZED-ATTRIBUTE-SPECIFIER]
     {{DERIVATION-SPECIFIER;
       EXTENT-SPECIFIER
       ACCESS-SPECIFIER}}

SIMPLE-ATTRIBUTE-SPECIFIER <-
  ATTRIBUTE-NAME : {Set-of} CLASS-SPECIFIER

PARAMETERIZED-ATTRIBUTE-SPECIFIER <-
  ATTRIBUTE-NAME (PARAMETER-LIST) : {Set-of} CLASS-SPECIFIER
```

```
DERIVATION-SPECIFIER <-
    Derived By OBJECT-SPECIFIER
      With ARGUMENT-LIST

HISTORY-SPECIFIER <-
  Defaults for history attributes:
    ACCESS-SPECIFICATION-LIST
    TRIGGER-SPECIFICATION-LIST
  {{ HISTORY-ATTRIBUTE-SPECIFIER }}

HISTORY-ATTRIBUTE-SPECIFIER <-
  [SIMPLE-ATTRIBUTE-SPECIFIER;
   PARAMETERIZED-ATTRIBUTE-SPECIFIER;
   STATE-ATTRIBUTE-SPECIFIER]
     {{PROPAGATION-SPECIFIER;
       DERIVATION-SPECIFIER;
       ACCESS-SPECIFIER}}

STATE-ATTRIBUTE-SPECIFIER <-
  ATTRIBUTE-NAME : MACHINE-SPECIFIER

MACHINE-SPECIFIER <-
  Define State Machine
    {{ STATE-NAME : STATE-CLASS-SPECIFIER
         STATE-SPECIFIER }}

STATE-CLASS-SPECIFIER <-
  States Where QUALIFIER

STATE-SPECIFIER <-
  { {{ ATTRIBUTE-SPECIFICATION }} }   ;"Attributes of the given state"
    {{ TRANSITION-SPECIFIER }}

TRANSITION-SPECIFIER <-
      When Entering CLASS-SPECIFIER  ;"A predicate"
      { Do ACTION }
      Goto STATE-NAME}

CONTROL-SPECIFIER <-
  USER-CLASS-DEFINITIONS
  TRIGGER-DEFINITIONS
Control for whole object:
  Read By USER-CLASS-NAME
       EXTENT-SPECIFIER
  Write By USER-CLASS-NAME
       EXTENT-SPECIFIER
  Included By USER-CLASS-NAME
       EXTENT-SPECIFIER

ACCESS-SPECIFIER <-
  OPERATION By USER-CLASS
```

```
ACCESS-SPECIFICATION-LIST <-
  [ACCESS-SPECIFIER;
   ACCESS-SPECIFICATION-LIST, ACCESS-SPECIFIER]

USER-CLASS <-
  Users Where QUALIFIER

USER-CLASS-DEFINITIONS <-
  {{ USER-CLASS }}

TRIGGER-DEFINITIONS <-
  {{ TRIGGER-DEFINITION }}

TRIGGER-DEFINITION <-
  { NAME = }
    Define Trigger on OPERATION
    Object Set: CLASS-SPECIFIER
    User Set: USER-CLASS
    Trigger-Program: OBJECT-SPECIFIER
                 on: ARGUMENT-LIST

TRIGGER-SPECIFICATION-LIST <-
  [TRIGGER-NAME;
   TRIGGER-SPECIFICATION-LIST, TRIGGER-NAME]

INHERITANCE-SPECIFIER <-
  Inherit Rights From CLASS-SPECIFIER
  { But Not For USER-CLASS }

GRANT-SPECIFIER <-
  Grant Rights To CLASS-SPECIFIER
  { But Not For USER-CLASS }

CLASS-SPECIFIER <-
  [CLASS-NAME;
   CLASS-SPECIFIER Where QUALIFIER;
   ATTRIBUTE-NAME of CLASS-SPECIFIER;
   Not CLASS-SPECIFIER;
   CLASS-SPECIFIER And CLASS-SPECIFIER;
   CLASS-SPECIFIER Or CLASS-SPECIFIER;
   CLASS-SPECIFIER Minus CLASS-SPECIFIER]

QUALIFIER <-
  [Contains [All Some INTEGER] [CLASS-SPECIFIER Components]
     VERSION-SPECIFIER;

  Part Of CLASS-SPECIFIER;

  ATTRIBUTE-NAME (PARAMETER-LIST) RELOP SINGLE-VALUE;
  ATTRIBUTE-NAME (PARAMETER-LIST)
    Member Of [CONSTANT-SET; CLASS-SPECIFIER];
  Object In STATE-DESCRIPTOR;

  Read By USER-CLASS;
  Write By USER-CLASS;
  Read Granted To CLASS-SPECIFIER;
  Read Denied To USER-CLASS;
  Write Granted To CLASS-SPECIFIER;
  Write Denied To USER-CLASS]

VERSION-SPECIFIER <-
    [With [All; Some; INTEGER] Versions
       Where QUALIFIER;
     With Versions While In STATE-DESCRIPTOR]

STATE-DESCRIPTOR <-
  State With STATE-PROPERTY RELOP VALUE
  State INTEGER States [Before; After] STATE-DESCRIPTOR
  State Waiting For CLASS-SPECIFIER To Become CLASS-SPECIFIER

RELOP <-
  <= | >= | < | > | =

VALUE <-
  [NUMERICAL-CONSTANT;
   STRING-CONSTANT;
   CONSTANT-SET;
   CLASS-SPECIFIER]

OBJECT-SPECIFIER <-
  [ The CLASS-SPECIFIER;
    ATTRIBUTE-NAME of OBJECT-SPECIFIER;
    OBJECT-SPECIFIER.ATTRIBUTE-NAME ]

ARGUMENT-LIST <-
  [VALUE; ARGUMENT-LIST, VALUE]

PARAMETER-LIST <-
  [ {NAME :} VALUE;
    PARAMETER-LIST, {NAME :} VALUE]

ACTION <-
  PROGRAM where QUALIFIER

CONSTANT <-
  [STRING-CONSTANT; NUMERICAL-CONSTANT]

CONSTANT-SET <-
  "{" CONSTANT-LIST "}"

CONSTANT-LIST <-
  [CONSTANT; CONSTANT-LIST, CONSTANT]
```

```
CLASS-NAME <- NAME

COMPONENT-NAME <- NAME

ATTRIBUTE-NAME <- NAME

STATE-NAME <- NAME

USER-CLASS-NAME <- NAME

NAME <-
  *string of lowercase letters beginning with a capital*

INTEGER <-
  *an integer*

NUMERICAL-CONSTANT <-
  *a number constant*

STRING-CONSTANT <-
  *a string constant*

SINGLE-VALUE <-
  [ CONSTANT;
    OBJECT-SPECIFIER ]

USERS <-
  *a class of objects that contains one object for each of the users
   who are known to the system*

PROGRAM <-
  *a class of objects that contains application specific programs*
```

Appendix E.

Functional Object-Retrieval Language

This appendix describes the functional object-retrieval language that is used in the body of this thesis to describe the semantics of object specifiers. The form of this language does not imply anything about the way in which a given retrieval would be implemented. It is used simply to map a given object specifier into a set specification with known semantics.

We will present each of the constructs of this language along with a narrative description of what each one means. This intuitive specification will be couched in terms of set theoretic notions.

E.1 Functions

The language describe in this appendix consists of a set of functions that have primitive values and sets of these primitive values as their domains. Each function maps a set of arguments into a primitive value or a set of primitive values. A function can have a name like the function *plus* or it can be construct from other functions as described below.

New functions can be formed by creating a *lambda expression.* The lambda expression here is like the lambda expression in the lambda calculus or the programming language, LISP. A lambda expression consists of three parts: a list of bound variables (or arguments) and a *body* which is a function that is defined by the composition of previously defined functions. An example lambda expression is the following:

```
Lambda (x) plus (x, 3)
```

This is a function that adds 3 to its argument, x. Plus is a function with two formal parameters that returns the sum of those parameters as a result.

An important class of functions is the class of *predicates*. A predicate is a function that maps its arguments into the set {**true**, **false**}. Predicates can be constructed as lambda expressions as long as the function that is the top-level function of the body of the lambda expression is a predicate. An example of a predicate is:

```
lambda (x) greater-than (x,3)
```

E.1.1 Attribute Application

Functions can be applied to arguments. The built-in function *apply* is available for that purpose. The first argument to *apply* is a function, and the other arguments are the objects to which the function is to be applied. The following:

```
Apply (plus, 3, 4)
```

will compute the value 7 by applying the function *plus* to the remaining arguments. We can also use the equivalent notation, **plus(3, 4)**. An attribute, such as *author*, of an object is a function that returns the value of that attribute for object, x, when applied to x. We can write this as **Author (x)**.

E.1.2 Iterate

In order to apply a function f to each of the members of a set of n elements, S, (i.e., $S = \{x_1, x_2, \dots, x_n\}$), we use the built-in function *iterate*. The meaning of:

```
Iterate [S, f]
```

is to apply f to each of the elements of S. The result of the above application of the iterate function is a new set containing the results of the individual applications of f. That is, the result of the above function is:

```
{f(x_1), f(x_2), ... , f(x_n)}
```

E.1.3 If

The function *If* is used to achieve conditional application of a function. It takes either two or three arguments. In the case of three arguments, if the first argument evaluates to **true**, the second argument is evaluated, and the third argument is ignored. If the first argument evaluates to **false**, the second argument is ignored, and the third argument is evaluated. In the case of two arguments, the second argument is evaluated if and only if the first argument evaluates to **true**.

E.2 Set formers

The functional language deals with sets of objects. The simplest way to refer to a set is by some name. Examples of named sets might be *Employees* or *Reports.* Sets can also be constructed by functions that take other sets as arguments. This section describes the functions that are available to create new sets from old sets.

E.2.1 Restrict

The *Restrict* function takes two arguments: a set and a predicate. The value of *restrict* applied to a set, S, and a predicate, P, is all those members of S for which P is true. The result of a *restrict* function is always a subset of the set, S.

An example of a *restrict* is the set of all Reports that have been written after September 1. This would be expressed as follows:

```
Restrict [Reports, lambda (r) date-written (r) > 9/1]
```

The function *date-written* is an attribute of reports.

E.2.2 Image

The *image* function takes two arguments: a set, S, and a function, F. The function must be defined for all members of the set. That is, S must be a subset of the domain of F. The result of **Image (S, F)** is a set of objects, R, in the range of F, such that for each x in S, F(x) is a member R.

An example is the set of all employees who are managed by the mangers who live in Boston. This would be expressed as follows:

```
Image (
  Restrict (Mangers, lambda (m) lives-in (m) = "Boston")
  lambda (x) managed-by (x))
```

This example also illustrates how one function can be nested within another function.

E.2.3 Boolean Operators

Given two or more set expressions, we can use the standard Boolean operators to combine them into a single set expression. The *Union* function takes an arbitrary number of set expressions as arguments and returns the set of objects that is the union of the sets produced by each of the individual set expressions. The *Intersection* function takes an arbitrary number of set expressions as arguments and returns the set of objects that is the intersection of the sets produced by each of the individual set expressions. The *Difference* function takes two set expressions as arguments and returns all those elements that are members of the set produced by the first argument but are not members of the result produced by the second argument.

E.2.4 Dot Notation

We can use *dot notation* as a shorthand for attribute (a kind of function) application. The application of an attribute, A, to an object, x, can be abbreviated as x.A. In this way we can abbreviate the application of the *author* attribute to a report, r as: **r.author.** Dot notation makes it convenient to cascade the successive application of several attributes to an object. Consider, **r.author.group**. This first applies the *author* attribute to the report, r, to get a particular user of the system, and then applies the *group* attribute to that user to get the group of that user. The result of the compound expression, then, is the group affiliation of the author of a report.

E.3 Predicate formers

There are several functions that form predicates form other predicates and sets. These functions are described in detail in this section.

E.3.1 For-some

The next two functions, *for-some* and *for-all* relate to quantification over a set of objects. *For-some* takes two arguments: a set, S, and a predicate, P. The value of *for-some* will be **true** if the predicate, P, is true for at least one member of S. If P is true for no members of S, the value of the function is **false.**

```
    For-some (Reports, lambda (r) length-in-pages (r) > 50)
```

This predicate is true if there is at least one report in the set (i.e., class) named *Reports* has a value for its *length-in-pages* attribute that exceeds 50.

E.3.2 For-all

The predicate *For-all* takes two arguments: a set, S, and a predicate, P. The value of *for-some* will be **true** if the predicate, P, is true for every member of the set, S. If P is false for any member of S, the value of the predicate is *false.*

```
For-all (Reports, lambda (r) date-created (r) > 1/1/76)
```

This predicate will be true if every report in the set, *Reports* was created after 1975.

E.3.3 Member-of

Member-of is a predicate that tests to see if a given object is a member of a given set. It, thereby, takes two arguments: an object, x, and a set, S. If x is a member of S, the value of this predicate will be **true**, otherwise it will be **false**.

```
Member-of (1981-Group-Final-Report, Reports)
```

This predicate is true if the group's final report for 1981 is a member of the set named *Reports*, otherwise it is false.

E.3.4 Contained-in

Contained-in is a predicate that takes two arguments: a set S_1 and a set S_2. If S_1 is a subset of S_2, then the value of the predicate is **true**, otherwise it is **false**.

```
Contained-in (
  Restrict (Reports, lambda (r) author="Fred"),
  Reports)
```

The above expression is always true since the restriction of a set is always a subset of that set regardless of the predicate (i.e., the lambda expression in the *restrict*).

E.3.5 Comparison Operators

The standard arithmetic comparison operators (i.e., equals, less-than, greater-than, less-than-or-equal-to, greater-than-or-equal-to) are available to form predicates. They each take two arguments and are true whenever the expression that results from substituting the corresponding arithmetic operator between the two arguments is true.

These operators are supported for numbers as described above; however, they are also supported for several other built in types that have some accepted ordering. One date, d_1, is less than another date, d_2, if d_1 occurred before d_2. Similarly, a text string, t_1, is less than another text string, t_2, if t_1 occurs lexicographically before t_2. The other comparison operations work in similar ways for dates and strings.

E.4 ODM-Specific Functions

The functions that were introduced above have general usefulness as a language for manipulating sets. In order to use this language in the domain of object management systems, we have added some additional functions that pertain to the basic mechanisms of such a system. These additional functions are described in the following sections.

E.4.1 Components-of

The function *components-of* takes a primitive object as an argument and returns a set of object that are the components of its argument. It can also take an optional second argument which is the name of one of the components of the type of its first argument. In this case, it will only return those components that belong to the named component. The following two examples illustrate this. Assume that R is a report object.

```
1. Components-of (R)
2. Components-of (R, Chapters)
```

The first application of the function will return a set containing all the components of the report, R. This will include all of its chapters, all of its appendices, its bibliography an so forth. In the second expression, the result will only be the set of chapters which are components of the report, R.

E.4.2 Versions-of

The function *versions-of* takes a conceptual object, x, as an argument and returns all versions of x as a result. That is, it returns the members of its version set.

```
Versions-of (1981-Final-Report)
```

This expression will return a set containing all of the versions of the 1981 final report.

References

1. Abe, Koji, Toshihiko Kuzushima, Hiroshi Ishii, and Shinich Kanbara. A Document Processing System for Office Automation. Proceedings of the Fourth Annual Office Automation Conference, Philadelphia, Pennsylvania, February, 1983, pp. 247-253.

2. Alegria, J. A. S. and T. C. Bylander. Towards a Personal Research Notebook and Library. 1981 Proceedings of the International Conference on Cybernetics and Society, Atlanta, Georgia, October, 1981.

3. Arens, Gail C. Recovery of the Swallow Repository. Tech. Rep. TR-252, MIT, Laboratory for Computer Science, January, 1981.

4. Barber, Gerald. Supporting Organizational Problem Solving with a Workstation. Technical Memorandum TM-224, MIT, Laboratory for Computer Science, July, 1982.

5. Barber, Gerald and Carl Hewitt. Foundations of Office Semantics. Technical Memorandum TM-225, MIT, Laboratory for Computer Science, July, 1982.

6. Baroody, A. James, Jr., and David J. DeWitt. An Object-Oriented Approach to Database System Implementation. *ACM Transactions on Database Systems 6*, 4 (December 1981), 576-601.

7. Barron, John. Dialogue and Process Design for Interactive Information Systems Using Taxis. Proceedings of the Conference on Office Information Systems, ACM/SIGOA, University of Pennsylvania, Philadelphia, Pennsylvania, June, 1982.

8. Bobrow, D.G. and T. Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science 1*, 1 (1977).

9. Botterill, J.H. The Design Rationale of the System/38 User Interface. *IBM Systems Journal 21*, 4 (1982).

10. Cattell, R.G.G. Integrating a Database System and Programming/Information Environment. Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling, ACM, Pingree Park, Colorado, June, 1980.

11. Curry, Gael, Larry Baer, Daniel Lipkie, and Bruce Lee. Traits: An Approach to Multiple-Inheritance Subclassing. Proceedings of the Conference on Office Information Systems, ACM/SIGOA, University of Pennsylvania, Philadelphia, Pennsylvania, June, 1982.

12. Curtice, Robert and Paul Jones. *Logical Data Base Design.* Van Nostrand, 1982.

13. Date, C.J. *An Introduction to Database Systems, Vol. 1, Third Edition.* Addison-Wesley, 1981.

14. Date, C.J. *An Introduction to Database Systems, Vol. 2, Third Edition.* Addison-Wesley, 1983.

15. Ellis, Clarence A. An Office Information System Based on Intelligent Forms.

16. Ellis, C. and G. Nutt. Computer Science and Office Informaion Systems. Tech. Rep. SSL-79-6, Xerox Palo Alto Research Center, June, 1979.

17. Ellis, Clarence A. and Marc Bernal. OfficeTalk-D: An Experimental Office Information System. Proceedings of the Conference on Office Information Systems, ACM/SIGOA, University of Pennsylvania, Philadelphia, Pennsylvania, June, 1982.

18. Eswaren, K.P., J.N.Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM 19*, 11 (November 1976), 624-633.

19. Ferrans, James C. SEDL -- A Language for Specifying Integrity Constraints on Office Forms. Proceedings of the Conference on Office Information Systems, ACM/SIGOA, University of Pennsylvania, Philadelphia, Pennsylvania, June, 1982.

20. Galley, S.W. and Greg Pfister. *MDL Primer and Manual.* MIT, Laboratory for Computer Science, 1977.

21. Gibbs, Simon J. Office Information Models and the Representation of Office Objects. Proceedings of the Conference on Office Information Systems, ACM/SIGOA, University of Pennsylvania, Philadelphia, Pennsylvania, June, 1982.

22. Goldstein, Ira P. and Daniel G. Bobrow. A Layered Approach to Software Design. Tech. Rep. CSL-80-5, Xerox Palo Alto Research Center, December, 1980.

23. Goldstein, Ira P. and Daniel G. Bobrow. An Experimental Description-Based Programming Environment: Four Reports. Tech. Rep. CSL-81-3, Xerox Palo Alto Research Center, March, 1981.

24. Gray, J.N., R.A. Lorie, and G.R. Putzolu. Granularity of Locks in a Large Shared Data Base. Proceedings of the First International Conference on Very Large Data Bases, ACM, September, 1975.

25. Gray, J. N. Notes on a Data Base Operating System. In *Operating Systems: An Advanced Course*, Springer-Verlag, Berlin/Heidelberg, 1979, pp. 394-481.

26. Greif, Irene. PCAL: A Personal Calendar. Tech. Rep. TM-213, MIT, Laboratory for Computer Science, January, 1982.

27. Greif, Irene. Computer Support for Cooperative Office Activities. Proceedings of the 1982 Office Automation Conference, AFIPS, San Francisco, California, April, 1982.

28. Hammer, Michael and Michael Zisman. Design and Implementation of Office Information Systems. Proceedings of the NYU Symposium on Automated Office Systems, New York University Graduate School of Business Administration, May, 1979, pp. 13-24.

29. Hammer, Michael and Marvin A. Sirbu. What is Office Automation? Proceedings of the National Computer Conference Office Automation Conference, AFIPS, March, 1980, pp. 37-49.

30. Hammer, Michael and Jay S. Kunin. Design Principles of an Office Specification Language. AFIPS Conference Proceedings, 1980 National Computer Conference, AFIPS Press, Arlington, Virginia, May, 1980, pp. 541-547.

31. Hammer, Michael and Dennis McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems* (September 1981), 351-386.

32. Herlihy, M. and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems 4*, 4 (October 1982), 527-551.

33. Hollaar, L. A. Text Retrieval Computer. *Computer 12*, 3 (March 1979), 40-50.

34. Ilson, Richard. An Integrated Approach to Formatted Document Production. Master Th., MIT, August, 1980.

35. Kaehler, Ted. Virtual Memory for an Object-Oriented Language. *Byte* (August 1981), 378-387.

36. Krasner, Glenn. The Smalltalk-80 Virtual Machine. *Byte* (August 1981), 300-320.

37. Kunin, Jay S. Analysis and Specification of Office Procedures. Ph.D. Th., Massachusetts Institute of Technology, February, 1982.

38. Lebling, P. David. *The MDL Programming Environment.* MIT, Laboratory for Computer Science, 1980.

39. Liskov, Barbara and S. Zilles. Programming with Abstract Data Types. Proceedings of a Symposium on Very High Level Languages, Santa Monica, California, March, 1974.

40. Liskov, Barbara, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU Reference Manual. Tech. Rep. TR-225, MIT, Laboratory for Computer Science, October, 1979.

41. Malone, Thomas P. How Do People Organize Their Desks? Implications for the Design of Office Information Systems. *ACM Transactions on Office Systems 1*, 1 (January 1983).

42. Martin, James. *Computer Database Organization, Second Edition.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979.

43. McLeod, Dennis. A Semantic Database Model and its Associated Structured User Interface. Tech. Rep. TR-214, MIT, Laboratory for Computer Science, August, 1978.

44. McLeod, Dennis. On Conceptual Database Modelling. Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling, ACM, Pingree Park, Colorado, June, 1980.

45. Miller, Peter B., Sergey Tetelbaum, and Kincade N. Webb. BUSINESS - An End-User Oriented Application Development Language. *SIGMOD Record 12,* 1 (October 1981), 38-69.

46. Minsky, N. Files with Semantics. Proceedings of the 1976 Symposium on the Management of Data, ACM SIGMOD, 1976.

47. Oddy, R. N. Information Retrieval Through Man-Machine Dialogue. *Journal of Documentation 33,* 1 (March 1977), 1-14.

48. Pollack, Fred J., Kevin C. Kahn, and Roy Wilkinson. The iMAX-432 Object Filing System. Proceedings of the 1981 Symposium on Operating Systems, ACM, 1981.

49. Purvy, Robert, Jerry Farrell, and Paul Klose. The Design of Star's Records Processing: Data Processing for the Noncomputer Professional. *ACM Transactions on Office Systems 1,* 1 (January 1983).

50. Pyle, I.C. *The ADA Programming Language.* Prentice-Hall International, 1981.

51. Reed, David P. Naming and Synchronization in a Decentralized Computer System. Tech. Rep. TR-205, MIT, Laboratory for Computer Science, September, 1978.

52. Reed, David and Liba Svobodova. SWALLOW: a Distributed Data Storage System for a Local Network. Local Area Network Workshop, IFIP Working Group 6.4, North-Holland, Zurich, Switzerland, 1980.

53. Reid, Brian K. and Janet Walker. *SCRIBE: Introductory Users Manual.* Second Edition edition, 1979.

54. Reid, Brian K. and Janet Walker. *SCRIBE: Format Designers Guide.* First Edition edition, 1979.

55. Reid, B.K. A High-Level Approach to Computer Document Formatting. Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, ACM, January, 1980, pp. 24-31.

56. Robson, David. Object-Oriented Software Systems. *Byte* (August 1981), 74-86.

57. Rosenstein, Larry S. Display Management in a Integrated Office Workstation. Tech. Rep. TR-278, MIT, Laboratory for Computer Science, January, 1982.

58. Salton, G. Automatic Information Retrieval. *Computer 13*, 9 (September 1980), 41-56.

59. Saltzer, Jerome H., and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE 63*, 9 (September 1975), 1278-1308.

60. Sarin, Sunil K. Supporting Remote Meetings through Interactive Sharing of Computer-Based Information. PhD Thesis Proposal, M.I.T. Lab. for Computer Science, Office Automation Group, 1982.

61. Scofield, J. Editing as a Paradigm for User Interaction: a Thesis Proposal. Tech. Rep. 81-11-01, Department of Computer Science, University of Washington, November, 1981.

62. Slonim, Jacob, L.J. MacRae, W.E. Mennie, and Norman Diamond. NDX-100: An Electronic Filing Machine for the Office of the Future. *Computer* (May 1981), 24-36.

63. Svobodova, Liba, Barbara Liskov, and David Clark. Distributed Computer Systems: Structure and Semantics. Tech. Rep. TR-215, MIT, Laboratory for Computer Science, March, 1979.

64. Svobodova, Liba. Management of Object Histories in the Swallow Repository. Tech. Rep. TR-243, MIT, Laboratory for Computer Science, July, 1980.

65. Tsichritzis, D. OFS: An Integrated Form Management System. Proceedings of the Conference on Very Large Databases, ACM, 1980, pp. 161-166.

66. Tsichritzis, D. Form Management. *Communications of the ACM 25*, 7 (July 1982), 453-478.

67. Wilkes, M.V., and R.M. Needham. *The Cambridge CAP Computer and its Operating System.* Elsevier North Holland, 1979.

68. Wulf W., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernal of a Multiprocessor Operating System. *Communications of the ACM 17*, 6 (June 1974), 337-345.

69. Wulf, William A., Roy Levin, Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill, 1981.

70. Yemini, Yechiam and Boaz Misholi. Integrated Voice-Data Systems. Proceedings of the Fourth Annual Office Automation Conference, Philadelphia, Pennsylvania, February, 1983, pp. 143-149.

71. Zloof, Moshe M. Query-by-Example: The Invocation and Definition of Tables and Forms. Proceedings of the Internaional Conference on Very Large Databases, Framingham, Mass., September, 1975, pp. 1-24.

72. Zloof, M.M., and S.P. deJong. The System for Business Automation (SBA): Programming Language. *Communications of the ACM 20,* 6 (June 1977), 385-396.

73. Zloof, Moshe M. A Language for Office and Business Automation. 1980 Office Automation Conference, Atlanta, Georgia, 1980.

74. Zloof, Moshe M. QBE/OBE: A Language for Office and Business Automation. *Computer* (May 1981), 13-22.

75. Zloof, Moshe M. Office-by-Example: A Business Language That Unifies Data and Word Processing and Electronic Mail. *IBM Systems Journal 21,* 3 (1982), 272-304.