NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A275 048

DTIC
S ELECTE
JAN 27 1994
A

THESIS

The Covering Property of the Object-Oriented Data Model

-

Design and Implementation Issues

by

Todd Gregory Estes
and
Eric Martin Mueller

September 1993

Thesis Advisor: David K Hsiao

Approved for public release; distribution is unlimited.

94-02576

94 1 26 039

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Master's Thesis

4. TITLE AND SUBTITLE
The Covering Property of the Object-Oriented Data Model-Design and Implementation Issues

5. FUNDING NUMBERS

6. AUTHOR(S)
Estes, Todd Gregory
Mueller, Eric Martin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA. 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Inheritance is a necessary condition for construction of an object-oriented data model (OODM), but it is not sufficient. This is because inheritance applies to only one hierarchy. The covering construct meets this deficiency because covering maps an object in one hierarchy to a class of objects in another hierarchy. To date, covering has not been implemented into an existing OODM application.

This thesis implements the covering construct into a functioning object-oriented database environment. Implementation was achieved through modification of data constructs and the creation of a user-defined relation linking two or more hierarchies. Using the Multi-model Multi-lingual Dababase Supercomputer (MDBS), a sample, working application is described illustrating real world applications.

The results of this thesis show that the covering property can be implemented into an existing OODM without sacrificing the integrity of the data model. The cross-hierarchical mapping afforded by covering is a powerful construct that expands the capabilities of the model beyond pure inheritance. This makes the OODM suitable for a far wider range of applications. Together, inheritance and covering meet the necessary and sufficient conditions of the OODM.

14. SUBJECT TERMS
Object-oriented, object-oriented data model, covering, aggregation, inheritance, hierarchical data model.

15. NUMBER OF PAGES
120

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

The Covering Property of the Object-Oriented Data Model

-

Design and Implementation Issues

by

Todd Gregory Estes
Lieutenant, United States Navy
Bachelor of Arts University of Rochester, 1986
and
Eric Martin Mueller
Lieutenant, United States Navy Reserve
Bachelor of Science, University of California, Davis, 1982

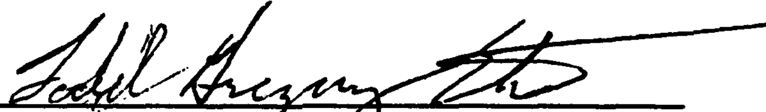
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

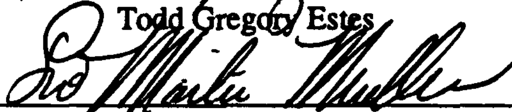
from the

NAVAL POSTGRADUATE SCHOOL
September 1993

Authors:



Todd Gregory Estes



Eric Martin Mueller

Approved By:

Dr. David K. Hsiao, Thesis Advisor

Dr. C. Thomas Wu, Second Reader

Dr. Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Inheritance is a necessary condition for construction of an object-oriented data model (OODM), but it is not sufficient. This is because inheritance applies to only one hierarchy. The covering construct meets this deficiency because covering maps an object in one hierarchy to a class of objects in another hierarchy. To date, covering has not been implemented into an existing OODM application.

This thesis implements the covering construct into a functioning object-oriented database environment. Implementation was achieved through modification of data constructs and the creation of a user-defined relation linking two or more hierarchies. Using the Multi-model Multi-lingual Dababase Supercomputer (MDBS), a sample, working application is described illustrating real world applications.

The results of this thesis show that the covering property can be implemented into an existing OODM without sacrificing the integrity of the data model. The cross-hierarchical mapping afforded by covering is a powerful construct that expands the capabilities of the model beyond pure inheritance. This makes the OODM suitable for a far wider range of applications. Together, inheritance and covering meet the necessary and sufficient conditions of the OODM.

DTIC QUALITY INSPECTED 5

Accesion For		
NTIS CRA&I	☑	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

B. THE OBJECT-ORIENTED DATA MODEL (OODM) 3

1. The Basic Constructs of OODM 3

2. OODM As a Database Model 6

C. THE COVERING CONSTRUCT 7

D. ORGANIZATION OF THE THESIS 10

II. A DESCRIPTION OF MDBS 11

A. THE CASE FOR FEDERATED DATABASES AND THE MULTIBACKEND DATABASE SUPERCOMPUTER WITH THE MULTIMODEL AND MULTILINGUAL CAPABILITIES 11

B. THE MULTIBACKEND DATABASE SUPERCOMPUTER (MDBS) 14

C. THE MULTIMODEL/MULTILINGUAL DATABASE SYSTEM (M^2DBMS) 16

D. SUMMARY 20

III. THE OBJECT-ORIENTED MODEL ON MDBS 21

A. THE INHERITANCE PROPERTY IN MDBS 24

B. THE COVERING PROPERTY ON MDBS 25

IV. IMPLEMENTATION OF THE COVERING PROPERTY 27

A. NAMING THE COVERING RELATION 28

B. ESTABLISHING THE RELATION 29

1. Establishing, originating and terminating objects. 30

2. Searching for objects of the same hierarchy. 30

C. ESTABLISHING COVERING 32

1. Determining the Scope. 32

2. Marking the covered objects. 32

3. Writing the data to the data file. 33

D. ACCESSING THE DATA VIA QUERIES 34

V. ADDITIONAL MODIFICATIONS AND LIMITATIONS OF THE OBJECT-ORIENTED INTERFACE 37

A. ADDITIONAL MODIFICATIONS TO THE OBJECT-ORIENTED INTERFACE 37
B. LIMITATIONS 39
VI. FUTURE WORK 42
VII. CONCLUSION 45
A. COVERING APPLICATION ONE: A NAVAL TASK FORCE 46
B. COVERING APPLICATION TWO: MULTI-LEVEL SECURITY 50
C. SUMMARY 54
APPENDIX A. DATA STRUCTURES OF THE OODM 55
APPENDIX B. SOURCE CODE 59
A. SOURCE CODE FOR THE CREATION OF A COVERING RELATION. 59
B. SOURCE CODE FOR QUERYING VIA THE COVERING CONSTRUCT... 73
C. SOURCE CODE FOR THE OBJECTID GENERATOR 83
APPENDIX C. TUTORIAL FOR THE OBJECT-ORIENTED INTERFACE ON MDBS 86
A. ACCESSING THE OBJECT-ORIENTED INTERFACE 86
B. SELECTING A DATABASE 87
C. LOADING DATA INTO THE DATABASE 90
D. CREATING A COVERING RELATION 95
E. PERFORMING QUERIES ON THE OBJECT-ORIENTED DATABASE 100
F. CONCLUSION 107
LIST OF REFERENCES 109
INITIAL DISTRIBUTION LIST 111

LIST OF FIGURES

Figure 1, An Example of the Inheritance Hierarchy 5
Figure 2, An Example of the Covering Property Through the In-Law Relationship of Two Family Trees 9
Figure 3, The Multibackend Database Supercomputer (MDBS) 15
Figure 4, The Multimodel, Multilingual and Cross-Model Accessing Capability 17
Figure 5, The Multimodel and Multilingual Database System (M^2DBMS) 18
Figure 6, The Model-Language Interfaces on MDBS 19
Figure 7, The Object-Class-Node Data Structure 22
Figure 8, The Object-Oriented Attribute-Node Data Structure 23
Figure 9, The Object-Oriented Superclass-Node Data Structure 23
Figure 10, The Object-Oriented Subclass-Node Data Structure 24
Figure 11, The Modified Object-Class Node 28
Figure 12, Pointers for the Two Family Trees After An Object Search 31
Figure 13, The Scope of a Covered Object 33
Figure 14, Example of a Covering Data File 34
Figure 15, Example of the object-class CARRIER and its component 48
Figure 16, Establishing a Task Force using the Covering Construct BATTLE GROUP BRAVO 49
Figure 17, Using covering to map the "Read-Down" principle 52
Figure 18, Using covering to map the "Write-Up" principle 53
Figure 19, The Object-Class Node Data Structure 55
Figure 20, The Object-Attribute Node Data Structure 56
Figure 21, The Object-Superclass Node Data Structure 57
Figure 22, The Object-Subclass Node Data Structure 58
Figure 23, The Schema Text File for the Family Database: FAMILYooldb 89
Figure 24, Descriptor and Template Files for the FAMILY Database 91
Figure 25, Record File for the FAMILY Database: FAMILY.r 93
Figure 26, Screen Display of the File FAMILY.r Being Loaded Into the Database 94
Figure 27, Request File for the FAMILY Database: FAMILYoolreq 102
Figure 28, Query Menu After Loading Request File FAMILYoolreq 103

ACKNOWLEDGEMENTS

There is a large number of people who contributed to this work both directly and indirectly. Although it would be impossible to acknowledge all of them, we would like to take this opportunity to thank some of the major players. Dr. David K. Hsiao provided endless support and encouragement during this work. Dr. Hsiao never once hesitated to provide technical and moral support whenever we raised the flag. His guidance was invaluable.

We would also like to thank Stan Watkins for his help in navigating through the complexities of MDBS. Additionally we would like to thank Steven R. Zeswitz, Richard S. Smith, and Bill Demers for imparting their knowledge of C on to us.

Eric would like to thank Alessandra A. Aubert for her infinite patience and understanding during the many long weekends and late nights he spent in the lab. Her positive attitude and inspiration helped him to overcome the periodic frustration he encountered while writing this thesis.

Finally, we would also like to thank Paulla J. Estes for her help in editing our work, transforming our writing from endless babble into a coherent document. Todd would like to thank Paulla for her patience and love during all the late nights in the database lab, and for taking care of his children, Andy and Samantha during his period in exile. For this, he is greatly appreciative.

Without the help and moral support of the above people and countless others, this work would not have been possible.

I. INTRODUCTION

A. BACKGROUND

Since the invention of the computer, the size and complexity of software applications have grown steadily. Starting from simple repetitious mathematical calculations, these applications have grown to include computer-aided design, artificial intelligence, and a host of others. Perhaps, the most natural use for computers has been in the storage and manipulation of large databases, since databases represent information collections. Database applications touch nearly every use of computers, because every application relies on some sets of data. As computers became more complex, higher-level programing languages were developed to allow software to keep pace with the advances in hardware. Moreover, these high-level languages have also allowed programmers to implement abstract concepts.

Paralleling the growth in programming languages has been the development of specific database models to handle increasingly large databases for database applications. These include the classical relational, hierarchical, and network models. Yet, when programs became increasingly complex, as with the development of computer-aided-design and/or computer-aided-manufacturing (CAD/CAM) applications, these classical models fall short of adequately supporting the more demanding and complex applications. There is a need for a more flexible and powerful model to keep pace with the new demands.

The object-oriented approach to database management is a direct consequence of the following two factors. One is the rapid movement away from imperative-programming paradigms and more towards object-oriented-programming paradigms. Second is the need for more flexibility, power, and abstraction required for modelling sophisticated applications.

Before discussing the Object-Oriented Data Model (OODM) and its constructs, we examine the classical data models and their intended usages:

The Relational Model: Introduced by Codd in 1970, this model stores data in the form of tables. Each attribute is assigned a column of the table. Key attributes are used to link the tables and provide a means of accessing data from multiple tables. This makes the relational model ideal for record-keeping applications. For example, data regarding a company's employees may be grouped by department. Employees of each department may then be represented by a table, with each column of the table corresponding to a particular employee attribute such as age, address, salary and others. Each row in the table corresponds to an individual employee. Queries are made using a relational data language, SQL, which translates relational language constructs into computer recognizable commands.

The Hierarchical Data Model: As its name implies, the hierarchical model has been developed to illustrate the many naturally occurring hierarchies found in the world. Corporate structures, biological classifications, and family trees are logical examples of these hierarchies. The hierarchial data model uses a "parent-child" relationship to describe the one-to-many characteristic of a tree structure. This parent-child relationship is an ideal modelling tool for data related to product assemblies. To illustrate, we can break down a product into its component parts, which in turn, into their sub-component parts. Thus, there exists a natural hierarchical one-to-many relationship corresponding to every level of the assembly.

The Network Data Model: The fundamental constructs of the network model are record types and set types. Data are stored in records; records of same value types of are grouped into record types. The set type describes the relationship between the two record types. Thus, the network model can model one-to-many, many-to-one, one-to-one, and many-to-many relationships. This model is best suited for inventory control applications. For example, an auto parts supplier may have many buyers for a specific part, and at the same time the supplier may have many individual auto parts to sell. Conversely, the buyer may buy different parts from one or many different manufacturers. Queries are written in a network language, CODASYL.

These models can be termed as task-specific. That is, they are best suited for particular tasks, but not flexible enough for the user to easily apply them to alternative tasks and their use. The result is that the database user tends to use the model with which he is most familiar. Furthermore, as different applications are needed, the database user will tend to adapt the model most familiar to him rather than employ the model best suited to the new task.

B. THE OBJECT-ORIENTED DATA MODEL (OODM)

The object-oriented data model (OODM) marks a significant departure from other data models. First, unlike the aforementioned models, OODM is not based on a standard language such as SQL or CODASYL. Rather, it uses conventional languages such as C, C++, and Ada, and borrows many of its concepts from object-oriented programing. In fact, there are many different object-oriented data models, languages, and database systems. Second, the object-oriented data model provides constructs that allow flexibility, modularity, and portability, previously unavailable in other models. Finally, it is powerful enough to model many diverse database applications, in particular abstract modeling such as applications in CAD/CAM. These final two points make OODM less task specific than conventional models.

1. The Basic Constructs of OODM

The basic element of OODM is the "object" and "class" of objects. An object can be defined as "a physical entity, a concept, an idea, an event, or some aspect of interest to database application" [Booc91]. These objects are grouped logically into "classes" and then arranged into a hierarchy. Some examples of objects may be cars, trucks, or vans. Similarly, cars could be grouped by types into sub-classes of sports cars, sedans, and station wagons which are then grouped together into one class, Vehicles.

With the object as its building block, OODM comprises four major elements: Abstraction, Encapsulation, Modularity, and Hierarchy [Booc91]. Abstraction refers to the

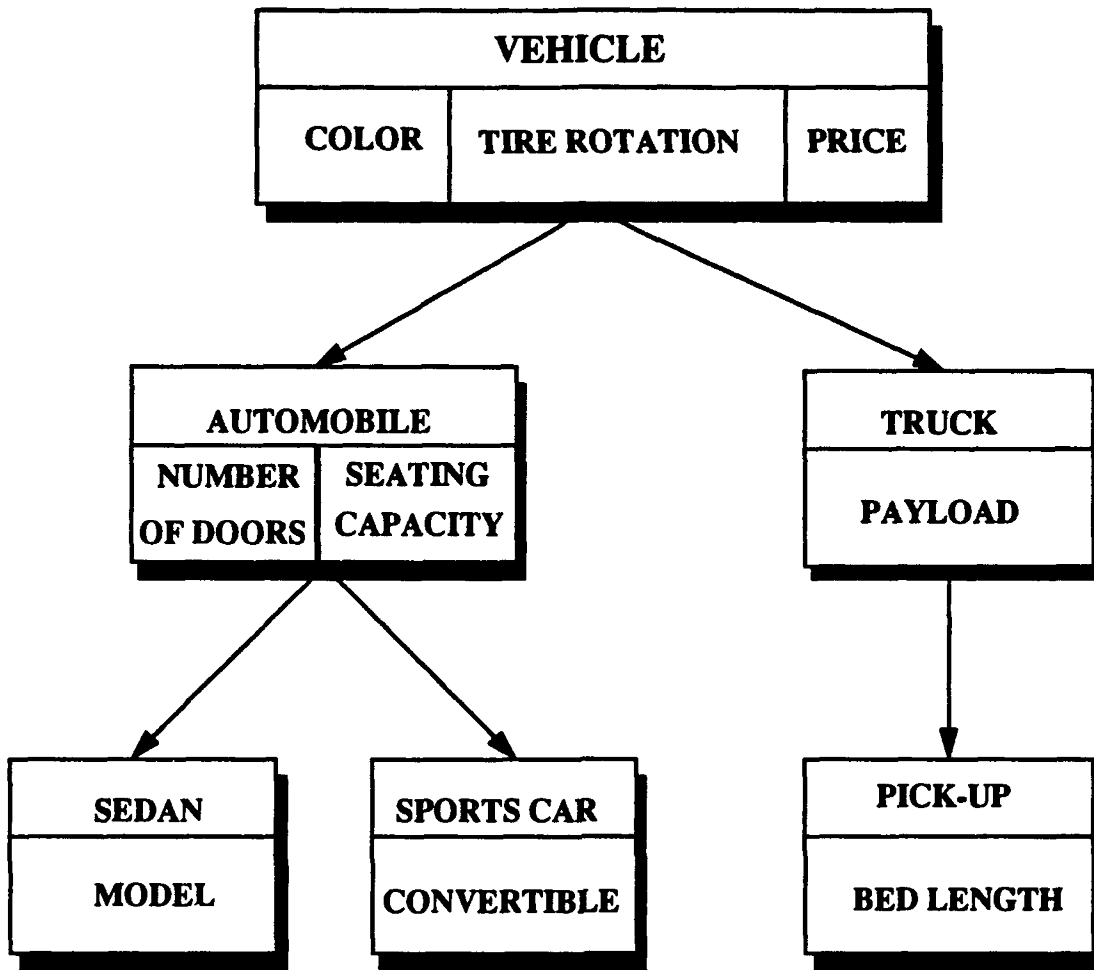
essential characteristics of an object that distinguish it from all other objects. Returning to the vehicle example, the extended passenger/cargo compartment distinguishes a station wagon from a sedan, and an open bed distinguishes a truck from a sports car. Both are examples of abstraction.

Encapsulation marks a revolutionary feature of OODM. Rather than store the data separately from the program, as in conventional database systems, the object-oriented data model stores the data with the program; i.e., these programs, called the methods (or actions), to be applied to the data are stored entirely within the object. This encapsulation of the methods allows only legitimate operations to be performed on that object, thus preserving data integrity.

Modularity is a direct by-product of the object structure and the encapsulation characteristic. Because each object contains the methods necessary to operate on that object, the object can stand alone or be moved to other parts of the database. Modularity and portability go hand-in-hand.

Finally, the hierarchical structure gives rise to the notion of inheritance. Inheritance is the key to object-oriented design and functionality. The object A is said to inherit from the object B if the object A retains the same characteristics,i.e., data or methods, as the object B. Graphically, the object B is located above the object A in the inheritance hierarchy. There are two types of inheritance. *Data inheritance* occurs if A inherits all the attributes of B whereas *operational inheritance* occurs when A inherits all the methods (or actions) of B.

Inheritance is most naturally thought of as the "a-kind-of" relationship. For instance, a sports car is a-kind-of car, which is a-kind-of vehicle. The benefit of inheritance is that the methods and data of "higher" classes of objects are passed to those "lower" in the hierarchy. If method describing tire rotation is pertinent to all three types of vehicles, i.e.,objects, then it is only declared in the object, Vehicle. Any object further down the Vehicle hierarchy inherits the tire rotation method. This requirement is depicted in Figure 1.



VEHICLE contains the attributes COLOR and PRICE and the method TIRE ROTATION. AUTOMOBILE and TRUCK inherit the attributes and methods of VEHICLE. SEDAN and SPORTS CAR inherit the attributes and methods of VEHICLE and AUTOMOBILE. PICK-UP inherits the attributes and methods of VEHICLE and TRUCK.

Figure 1: An Example of the Inheritance Hierarchy

2. OODM As a Database Model

Abstraction, encapsulation, modularity, and inheritance make OODM the ideal data model for sophisticated database applications. OODM offers a number of advantages when compared to traditional data models. First, OODM, unlike the relational data model, does not rely on the use of foreign keys in the manipulation of data. A foreign key is a key attribute which refers from one relation to another and its use or misuse may violate the referential integrity. Both relations must share the same domain for these attribute values. Instead of foreign keys, OODM uses the operational inheritance and the data inheritance to implement generalization and specialization. These inheritance properties maintain the data integrity constraints within OODM. Thus, not only does OODM eliminate the overhead required to maintain the foreign keys, it also minimizes the "dangling-key phenomenon" which violates the referential integrity of the foreign key. This phenomenon occurs when records referred to by the foreign keys are removed [Hsia92c].

Second, the hierarchical data model and OODM both use a hierarchical data structure. The hierarchical data model differs from OODM because it uses hierarchies that are arbitrary in nature. The hierarchies are arbitrary because they are developed by the programmer when the database is created. This may or may not be based on logical or natural hierarchies, and may change with time from one programmer to the other programmer. OODM, however, uses the inheritance property to create the hierarchies. Thus, they are more natural and less arbitrary than those found in the hierarchical data model [Hsia92c].

Finally, the network data model, like the hierarchical data model, uses hierarchies that are arbitrary in nature. Yet, it differs from the hierarchical model because two hierarchies, rather than one, are used to model a relationship between two record sets. This introduces confusion into the database design making database management more difficult to build and use [Hsia92c].

As discussed above, the traditional data models suffer from serious shortcomings. Furthermore, they are better suited for modelling specific tasks and concepts. OODM is

naturally suited for modelling complex and general concepts. With the traditional data models, the database designer must convert the conceptual design into an actual database. In the traditional procedure, the conceptual design may have some of its features sacrificed in order to meet the specifications of the database. On the other hand, OODM allows the database designer to move directly from the abstract and conceptual design to a functioning database, without corrupting the original concept.

Object-oriented databases can offer advantages in speed. Unlike the relational model, joins are not used in the OODM. This is because the object can be directly found using object IDs rather than using time-consuming search techniques. Also the inheritance property insures that data are passed to lower objects in the hierarchy thus eliminating the need for a join. Because the same data model is used by the database and the database programming language, format conversions at the disk-level are not required. This in turn helps speeding up the reading of data from the disk and the storage of data on the disk.

C. THE COVERING CONSTRUCT

There are various forms of object-oriented databases in existence. The proliferation of the OODM constructs stems from a lack of the standard similar to those found in other data models. The majority of the OODB designers agree that the inheritance is an important characteristic of the object-oriented database. Another construct found in object-oriented data models is the covering property. Covering (or aggregation) is the means by which an object in one hierarchy can relate to a class of objects in another hierarchy. As explained in [Hsia92c] that, "we say A is a cover of B if every object of A corresponds to a subset of objects of B." In mathematical terms, this means that the object class A is related to the power set of the object class B, though no "hard-wired" structures exist between them.

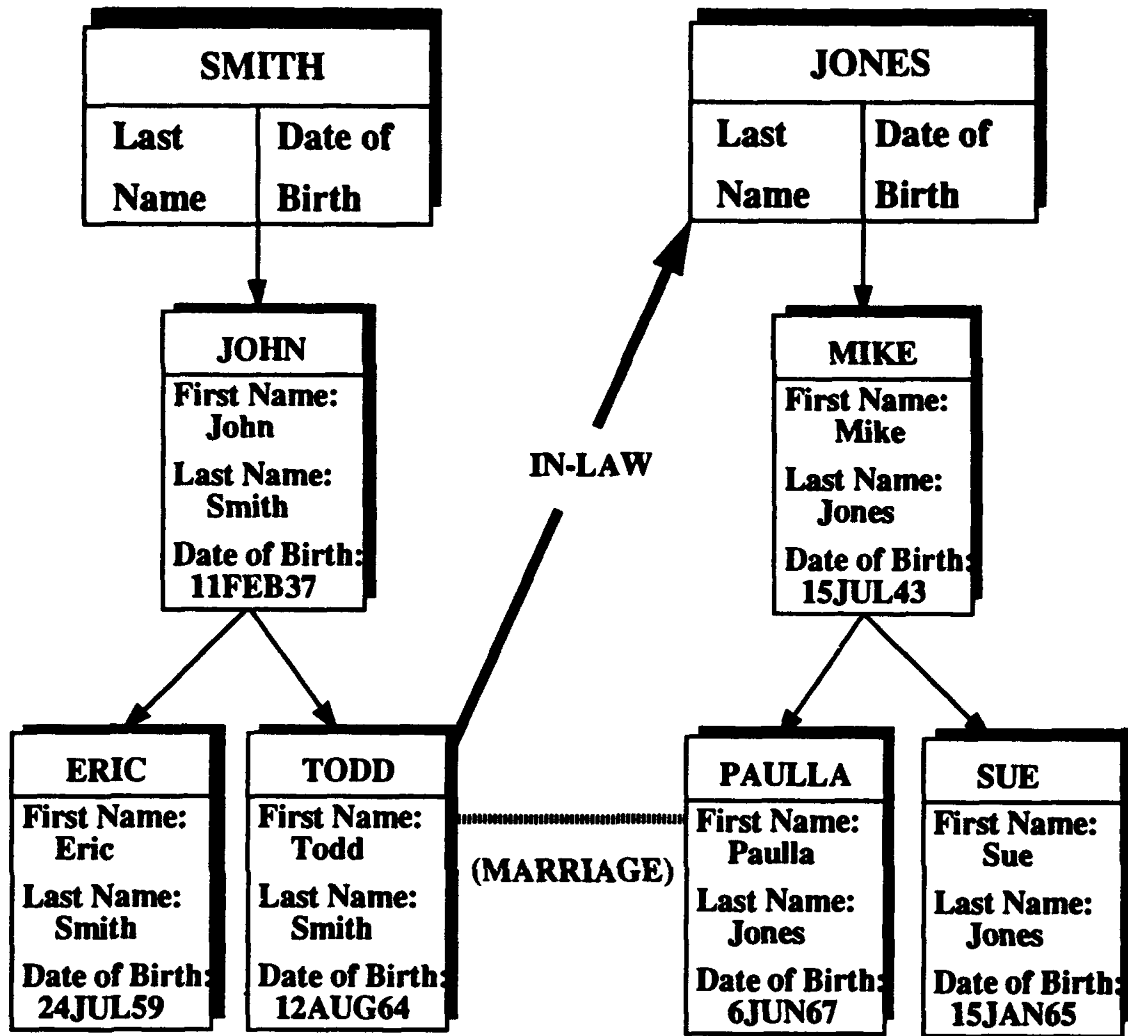
The covering property does not enjoy the same amount of popularity as the inheritance in the object-oriented database community. This is a carry-over from object-oriented programing where programmers did not want interaction between separate object

hierarchies. In programming, encapsulation prevents corruption of an object module by another object module. This concept does not necessarily hold with database applications. Encapsulation can prevent the sharing of data between objects. Covering is designed to overcome this limitation. It will be shown that covering allows more flexibility in accessing and manipulating data within an object-oriented database.

Perhaps the easiest way to visualize the covering property is to use the in-law relationship between two family trees. Two hierarchies (family trees) are related when a son of one set of parents marries the daughter of another set of parents. This marriage illustrates the utility of the covering property. Though not related by blood, there is now a recognized relationship between the groom and the bride's parents which are called the mother-in-law and father-in-law. The same is true for the bride. Similarly, from the parent's perspective, there is a "downward" relationship of the daughter-in-law and son-in-law. If this covering relationship is not introduced into OODM, it will be difficult for the members of two family trees to be related through the marriage.

Now we apply the covering property to a database. Suppose the family tree structure exists as described above. The inheritance alone allows each child to access the birth date of anyone within his family tree. Since the new husband is not part of that hierarchy, he cannot access information on his in-laws. With the covering property, this is not only possible, but other members of the opposite family tree can also do the same. This is a very powerful construct unique to the object-oriented data model. Without the covering property, independent hierarchies within a single database cannot be manipulated as a whole. See Figure 2 for an illustration of two family hierarchies based on the inheritance property and one in-law relationship based on the covering property.

To reiterate, we point out that, unlike the inheritance property, the covering property has not enjoyed the same widespread acceptance within the OODB community. In fact, covering has not been fully implemented into any working databases to date. Consequently, there is a need to prove the viability of this construct and demonstrate its usefulness. This thesis will focus on the implementation of the covering property into the object-oriented



TODD and PAULLA are married and create a marriage relationship. The covering construct, IN-LAW, allows TODD to access the Date of Birth of each member of PAULLA's family. If the IN-LAW property did not exist, TODD could not access this information.

Figure 2: An Example of the Covering Property Through the In-Law Relationship of Two Family Trees

data model of the Multimodel and Multilingual Database System (M^2DBMS). Specifically, the thesis will address the questions of feasibility, practicality, and usefulness of the covering property of OODM. The end product will be a working, demonstrable object-oriented database interface for an application using the covering construct.

D. ORGANIZATION OF THE THESIS

In Chapter II of this thesis, we provide an overview of the hardware and software of the Multibackend Database Supercomputer (MDBS) on which M^2DBMS runs. In Chapter III, we describe the implementation issues regarding the object-oriented data model in MDBS. The details and design of implementing the covering property into OODM is the subject of Chapter IV. Chapter V discusses additional modifications made to the object-oriented interface on MDBS as well as the limitations which exist within the interface. Future work need on the object-oriented interface in MDBS is detailed in Chapter VI. In Chapter VII, we summarize our conclusions. The appendices contain relevant illustrations, program logic, and a tutorial for the object-oriented interface on MDBS.

II. A DESCRIPTION OF MDBS

A. THE CASE FOR FEDERATED DATABASES AND THE MULTIBACKEND DATABASE SUPERCOMPUTER WITH THE MULTIMODEL AND MULTILINGUAL CAPABILITIES

The growth and proliferation of computers within business and industry has resulted in their involvement in nearly every aspect of the business life. Each company, or department within a company, accumulates and maintains different data types according to their own particular needs. This accumulation of data in multiple forms causes redundancy and inefficiency.

To illustrate this point, let us view a company composed of many departments, each using a specialized database for its operations [Hsia92b]. The personnel department, concerned with payroll-and-record keeping, might use a relational database to keep track of employee records. The engineering department might use the software running a hierarchical database because it is most naturally suited for engineered assemblies. Shipping and inventory control is accomplished using the network-model software due to the ease of representing the many-many relationships. The end result is that each department uses a "homogeneous" database for its own particular use; yet together, they form a "heterogeneous" database because they use different models and languages. Naturally, this contributes to needless duplication of data which are common to all departments. Moreover, since each department uses a different model, data cannot be shared among departments. The lack of data sharing among the departments is due to the fact that the modeled data are foreign to the user of another department. Also, as mentioned previously, users in each department will become familiar with their own particular model/ language and thus be limited to locally available data only.

If the same company now wishes to open several other plants and offices around the country, then the ability of each department or company division to share data is crucial for planning, implementing, and meeting corporate strategies and goals. Clearly duplicating all

the databases into different forms on the basis of various heterogeneous data models is inefficient; yet, purely centralized control requires a common type of database and model. The solution to the aforementioned two extreme measures is a *federated* or *multidatabase* structure. In a federated database or a multidatabase an user believes the user is using the user's own database with the user's favorite language, yet the actual data may be located elsewhere, based on a different data model and part of a larger database system. A federated database permits sharing of data, wider availability, and optimal use of assets. Federated database systems are actually a hybrid between a centralized and distributed systems [Elma89].

In order to establish a federated database with efficient data sharing, the following five conditions must be met [Hsia92b]:

1. *Transparent access to heterogeneous databases.* An user should be able to access any data using the model and language most familiar to the user. For example, the director of engineering may want to study the inventory of the most recently designed product. Rather than learn CODASYL-DML transactions to access the network database, he or she may use the more familiar DL/I transaction to access the data maintained by the shipping department.

2. *Local autonomy of each heterogeneous database.* This condition allows an owner to share the database with others without compromising the owners own integrity or security constraints. Continuing with our previous example where the director of engineering may view the data, he or she is not allowed to change the data. The ability to change the data is retained by the owner, who in this case is the shipping and receiving departmental manager.

3. *Federated databases are multimodel and multilingual.* Multimodel means that the database supports many different models. In our company example, the federated database would have to support the relational, hierarchical, and network models. Multilingual means that each transaction made on the database can be made using any language applicable to the models supported by the system. Thus, the company is able to

support all three models while executing transactions written in SQL, CODASYL-DML, and DL/I on their respective databases.

4. *Multibackend Capability.* This requirement is specifically aimed at resource consolidation. Rather than let all of the software and database run in existing computers in a multi-system environment, a specialized computer system, with many parallel *backends* is used to run the multimodel and multilingual software and to store all the databases. These backends are actually special-purpose database computers that are dedicated to supporting the database application. Efficiency is attained through the use of multiple backends linked in parallel. This parallelism induces high speed and great capacity.

5. *Effective and Efficient access and concurrency control mechanisms.* Because the wider access to the database allowed by the multimodel/multilingual capability, there exists a great potential to violate integrity and security constraints in a multi-system environment. However, despite the large number of backends and high degree of parallelism, the multibackend is in a single system or uni-system environment. We know how to safeguard the integrity and security of databases in the uni-system environment.

As can be seen above, these requirements dictate the use of specialized hardware and software. The parallel, multibackend computers are required for capacity and efficiency. Specialized software for interpreting multiple models and languages is necessary to ensure universal compatibility among users while still preserving data integrity and security constraints. The database research laboratory at the Naval Postgraduate School (NPS) combines both features into one system utilizing a multibackend supercomputer with the multimodel/multilingual software.

B. THE MULTIBACKEND DATABASE SUPERCOMPUTER (MDBS)

The Multibackend Database Supercomputer (MDBS) in the NPS database laboratory consists of one controller computer and six backend computers. The backend computers are connected to the controller by an ethernet local area network (LAN). Figure 3 illustrates the MDBS configuration. This system offers two significant advantages. (1) The reduction in response time for a given query varies inversely in proportion with the number of backend computers; and (2) if the number of backends increases proportional with the size of the database, then there is little change in the query response time [Meek93]. Thus, the number of backends is the deciding factor in determining transaction response time. In other words, MDBS is scalable for the desired responses.

The controller computer is an off-the-shelf Sun model 4/110 workstation utilizing the Sparc 4 RISC architecture, with 8 megabytes of RAM and one 373-megabyte hard drive [Meek93]. The controller's primary purpose is to provide communication between the backends and act as the interface between the user and the system. Unlike the backends, the controller does not have any database. However, in case of a backend failure, the controller can provide backup and recovery of the database with the use of magnetic tape [Meek93].

The backends are Sun model 4/280 workstations also using the Sparc 4 RISC architecture. All backends have identical hardware and software. This parallel architecture provides optimum speed and performance through its scalability, in terms of response-time reduction, and the response-time invariance [Bour93]. Each backend, being a self-contained database computer, stores the data on two types of disks. Base data (raw data) is stored on 1000-megabyte moving-head disk while meta data (information about the base data) and paging information are stored on two 96-megabyte Winchester-type disk drives, respectively [Meek93]. Base data are not replicated on its disks. Instead, they are distributed by clusters of data one cluster at a time across the tracks of the disk parallel drives. This clustering (or partitioning) of the data and evenly distributing the clustered data among the backend's disks help to achieve parallel access operations, because the data is

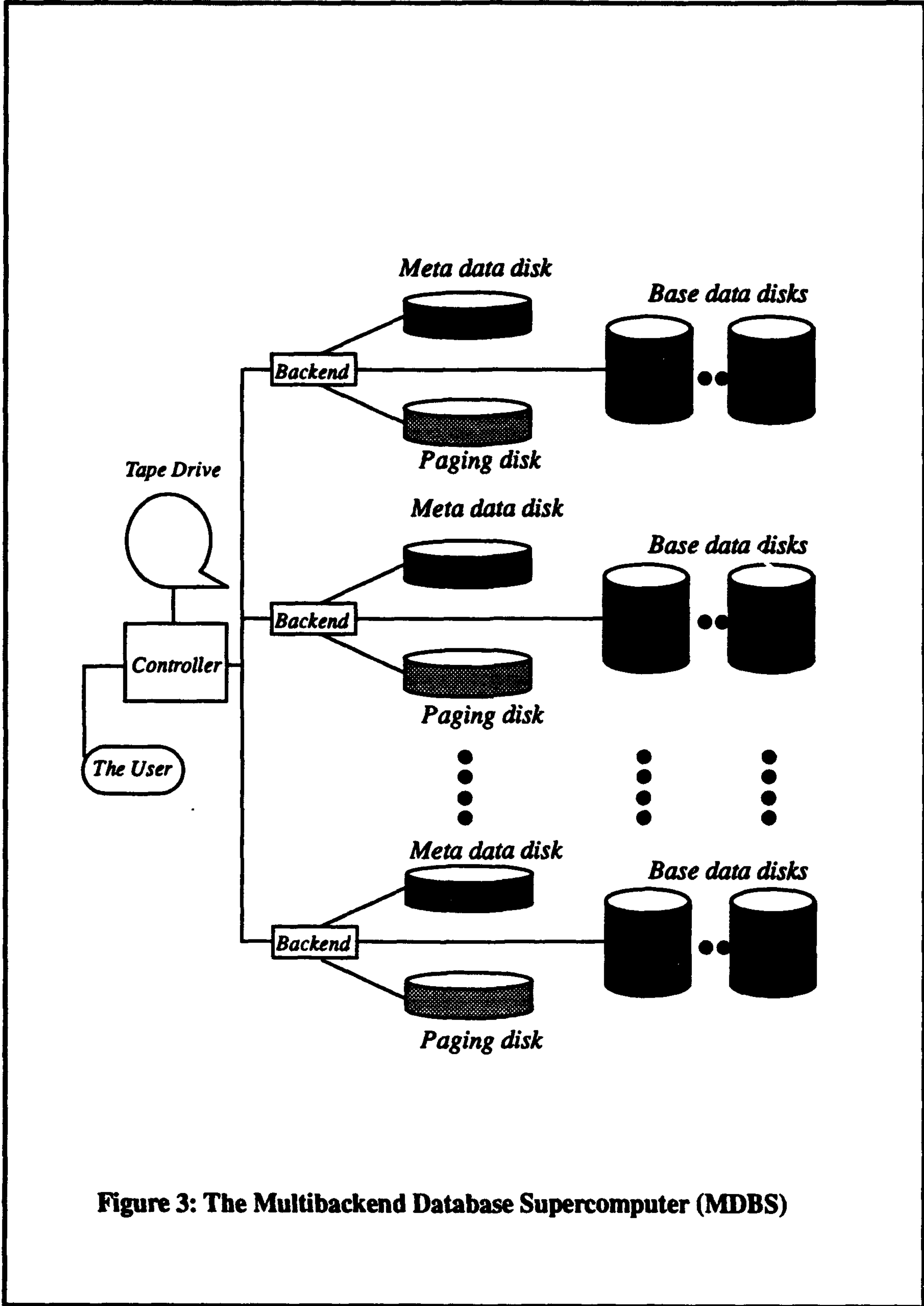


Figure 3: The Multibackend Database Supercomputer (MDBS)

evenly distributed among the disks of all the backend computers [Hsia92b]. Each track is accessed serially, while all the tracks for a cluster are accessed in parallel.

C. THE MULTIMODEL/MULTILINGUAL DATABASE SYSTEM (M^2DBMS)

To meet the multimodel/multilingual requirement of efficient federated databases, at present MDBS uses the software capable of accessing a consolidated database in any one of four data models. These models are the relational, hierarchical, network, or object-oriented data models. Further, each of the first three traditional models can be accessed using their corresponding languages, i.e., SQL, CODASYL, or DL/I. The object-oriented model does not have an associated language. The primary feature of this software is the ability to use any of the above languages to access the database. Furthermore, some cross-model accessing capabilities are possible on MDBS. Figure 4 describes the system in detail.

The heart of the MDBS software is the kernel data model (KDM) and the kernel data language (KDL). MDBS uses the attribute-based data model (ABDM) and its associated attribute-based data language (ABDL) for the kernel data model. The ABDM supports the five primary database operations of INSERT, UPDATE, DELETE, RETRIEVE, and RETRIEVE COMMON [Bour93]. All data is stored on the disk in the kernel format. Each transaction, regardless of model or language, is translated into the ABDL equivalent where it is processed and then returned to the user in the original language. Thus the users language is merely a convenient interface between him and the kernel data model.

When the user logs on to the system, he chooses a model with which to access the database. The chosen model is called the user data model (UDM) and the associated language is called the user data language (UDL). The UDL is how the user communicates with the kernel data model.

Each data language requires four software modules to accomplish the translation into the KDM equivalent. These modules are the language interface layer (LIL), the kernel mapping system (KMS), the kernel formatting system (KFS), and the kernel controller (KC) [Meek93].The relationship between the software modules and the kernel system is

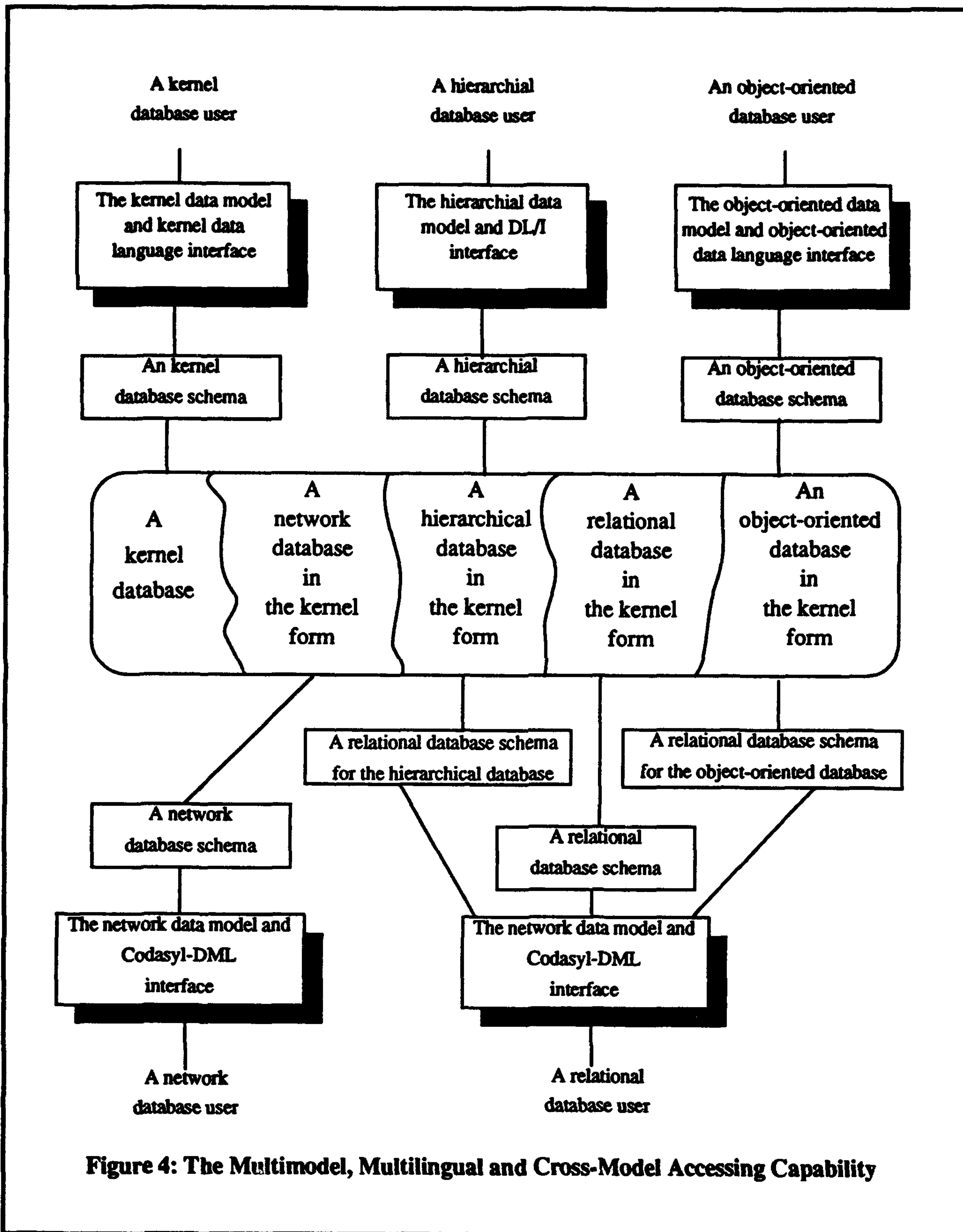
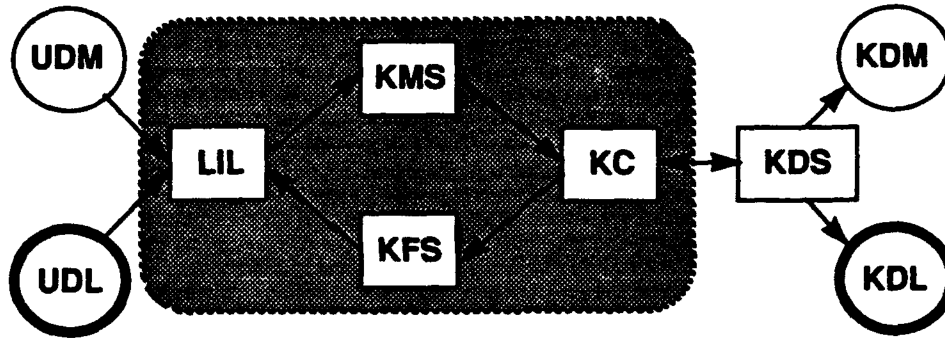


Figure 4: The Multimodel, Multilingual and Cross-Model Accessing Capability



UDM : User Data Model
UDL : User Data Language
LIL : Language Interface Layer
KMS : Kernel Mapping System
KFS : Kernel Formatting System
KC : Kernel Controller
M/LI : Model/Language Interface
KDS : Kernel Database System
KDM : Kernel Data Model
KDL : Kernel Data Language

Figure 5: The Multimodel and Multilingual Database System (M^2DBMS)

described in Figure 5 . Figure 6 illustrates the relationship between the different language interfaces. Each module is described in the following paragraphs [Karl93].

The function of LIL is to take the users transaction and route it to the appropriate module within KMS. The user can access LIL by sending transactions via the terminal or from a mass-load file. LIL also controls the order in which other modules are called.

The KMS module supports two functions. First, it identifies whether or not the user is creating a new database. If so, then a *data-model transformation* must occur. The UDM-database definition is then transformed by KMS into the KDM-database definition so that data can be processed and the database can be created. With the data-model transformation complete, the request can be sent to the kernel controller (KC). The KC sends the newly created KDB-database definition to the kernel data system (KDS) for processing. KDS then

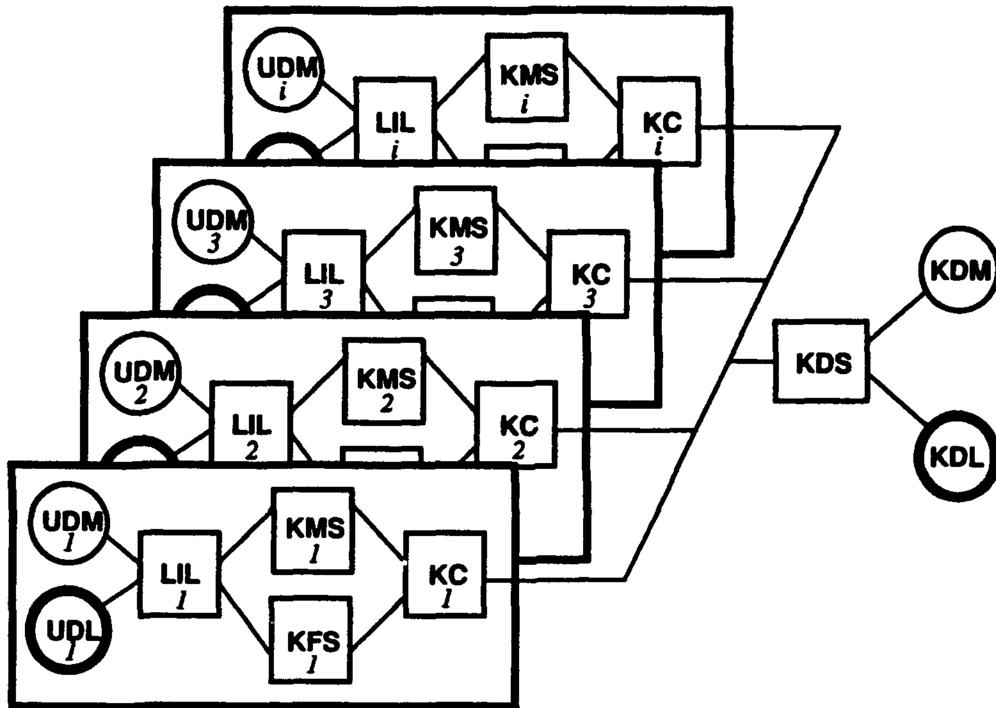


Figure 6: The Model-Language Interfaces on MDBS

notifies the MDBS controller that a new database has been created in the form of UDM. Data can now be entered by the user as well as queries against that data.

The second function of the KMS concerns the *data-language translation*. Here the UDL transaction is transformed into an equivalent KDL transaction by KMS. Once the transaction has been translated, it is sent to KC and then on to KDS for execution. Thus, KMS performs the vital functions of data-model and data-language translation, which is essential to achieving the multimodel/multilingual capability.

KFS is concerned primarily with returning the results of the query to the user. Once the transaction is executed in KDS, KFS takes the results and accomplishes the translation from the KDM form back to the UDM form. KFS then routes the transaction back to LIL for display to the user.

KC takes all ABDL transactions and passes them to the kernel system for execution. The nature of the transaction determines where it is sent by KC once query processing is

complete. If the transaction concerns an update, insertion, or deletion to the database then control reverts to LIL upon completion of the transaction. If the transaction is a retrieve request, then KC routes the request to KDS, which processes it and then sends the result back to KC. KC then takes the result and places it in a buffer for KFS which then displays the result to the user [Karl93].

D. SUMMARY

The need for accurate and rapid access to large heterogeneous databases is an universal problem facing business, industry, and government. Efficient management of vast quantities of data calls for the use of a federated database structure. In order to meet this challenge, specialized computers and software are necessary. MDBS with M^2DBMS offers one effective and efficient way of satisfying these requirements. The use of multiple backends in a supercomputer dedicated for the database provides speed and efficiency in database access and storage. The specialized software providing multimodel/multilingual capabilities allows wider use and greater access among the heterogeneous database users.

III. THE OBJECT-ORIENTED MODEL ON MDBS

The object-oriented model is implemented on MDBS. This implementation strategy has two advantages. First, an entire database system is not require to be written from the scratch. Only the object-oriented interface is required. This results in a savings of time and resources. Second, by implementing the object-oriented database into multilingual/ multimodel system, we preserve the interoperability of the heterogeneous database. This allows data sharing among the installed interfaces.

The interface design for the object-oriented model on MDBS closely parallels the design for the relational model [Roll84]. As in the relational implementation, the primary data structure used is the linked list. This structure is used because it dynamically connects related objects together and makes a hierarchical (tree) structure possible.

Each object being created is a record called an object-class node (**ocls_node**). These object-class nodes are connected via pointers to form a linked list of other objects. Within each object of an object-class node, there are attributes specific to that object and pointers connecting a list of attributes and pointers relating that node to other class nodes in the list. Refer to Figure 7 for the layout of the object class data structure.

The attribute **ocn_name** contains the name of the object, while the attribute **ocn_num_attr** holds the total number of attributes for the object. The attributes **ocn_supcls** and **ocn_subcls** contain the number of the object's superclass and subclass respectively.

The six pointers in **ocls_node** form a connection between the object's attributes as well as its relationship between its superclasses and subclasses. The pointer **ocn_first_supcls** points to the object's immediate superclass while **ocn_curr_supcls** points to the next superclass in the list of superclasses for that object. Similarly the pointers **ocn_first_subcls** and **ocn_curr_subcls** point to the object's subclasses. **Ocn_first_attr** and **ocn_curr_attr** point to the object's first attribute and subsequent attributes respectively. See Figure 19 in Appendix A. for a diagram of **ocls_node**.

```
struct ocls_node
    {
    char                    ocn_name[RNLength + 1];
    int                     ocn_num_attr;
    int                     ocn_supcls;
    int                     ocn_subcls;
    struct o_supcls_node    *ocn_first_supcls;
    struct o_supcls_node    *ocn_curr_supcls;
    struct o_subcls_node    *ocn_first_subcls;
    struct o_subcls_node    *ocn_curr_subcls;
    struct oattr_node       *ocn_first_attr;
    struct oattr_node       *ocn_curr_attr;
    struct ocls_node        *ocn_next_cls;
    }
```

Figure 7: The Object-Class-Node Data Structure

All attributes are contained in the object-attribute node. For objects with multiple attributes, these nodes are connected to form a linked list of attributes. Special nodes called object-superclass nodes and object-subclass nodes are created to form a hierarchy within the linked list. These nodes form the basis of the inheritance construct and are also the basis for developing the new covering construct. The object-class node is the primary node, which contains pointers to other secondary nodes that form the hierarchical linked list. These secondary nodes are discussed in detail below [Karl93].

1. Object-attribute node. **Oattr_node** forms the elements of a linked list connecting all the attributes relevant to a particular object. Each object attribute, such as name, birth date, salary, and so on is individually located in a different object-attribute node. Within this node are three attribute fields and one linking pointer. The attribute name (for example, birth date) is located in attribute field **oan_name**. The data type (i.e., character, integer, etc.) of the particular attribute name is located in the attribute field **oan_type**. The length of the attribute name is contained in the attribute field **oan_length**. The linking pointer in the node, called **oan_next_attr**, points to the next attribute in the linked list of attributes. This is used for objects with multiple attributes. Figure 8 shows the layout of the attribute-node data structure.

```
struct oattr_node
        {
        char                    oan_name[ANLength + 1];
        char                    oan_type[RNLength + 1];
        int                     oan_length;
        struct oattr_node       *oan_next_attr;
        }
```

Figure 8: The Object-Oriented Attribute-Node Data Structure

2. Object-superclass node. This node is called **o_supcls_node**. It is the connecting point for each object to its superclass. The only attribute within this node is the name associated with the superclass, **osn_name**. Two pointers are contained within this node. One pointer, **osn_supcls**, connects the node to a superclass, while another pointer, **osn_next_supcls**, points to the next superclass in the linked list of superclasses pertaining to that particular node. The ability of the object to have more than one superclass allows for the multiple inheritance. The superclass node data structure is described in Figure 9.

```
struct o_supcls_node
        {
        char                    osn_name[RNLength + 1];
        struct ocls_node        *osn_supcls;
        struct o_supcls_node    *osn_next_supcls;
}
```

Figure 9: The Object-Oriented Superclass-Node Data Structure

3. Object-subclass node. This node is called **o_subcls_node**. It is similar in structure to the superclass node. The attribute **osn_name** lists the name of the node, while the pointers **osn_subcls** and **osn_next_subcls** link the object with its subclasses. See Figure 10. Figures 20 through 22 in Appendix A. illustrate how these three nodes are linked together.

```
struct o_subcls_node
        {
        char                    osn_name[RNLength + 1];
        struct ocls_node        *osn_subcls;
        struct o_subcls_node    *osn_next_subcls;
}
```

Figure 10: The Object-Oriented Subclass-Node Data Structure

A. THE INHERITANCE PROPERTY IN MDBS

Inheritance in the object-oriented model is accomplished through the object-superclass node and the object-subclass node and their interconnections with object class node. When an object is first created, pointers within these nodes are initialized as null. As subsequent objects are created, these nodes will point to objects located higher or lower in the hierarchy. For example, if object B is to become a member of the subclass of object A, then the following pointers will be set. The first subclass pointer (**ocn_first_subcls**) within object A's object class node will point to the newly created object subclass node (**ocn_subcls_node**). This node will in turn point to the object-class node (**ocls_node**) belonging to B. Similarly, the corresponding superclass nodes and pointers will connect object B up to object A.

Attributes are inherited downward within the hierarchy. Lower objects contain all of the attributes ascribed to objects located above it, as well as those attributes specific to that particular object. The object-oriented model on MDBS accomplishes this by using nested linked lists. Each object node contains a pointer to a linked list containing all of the attributes pertaining to that object. Because each object is connected to other objects in the hierarchy via the superclass and subclass pointers, the objects themselves form a linked list. Thus, as the hierarchy is traversed downward, the linked list of attributes grows longer as each linked object is scanned. This is best illustrated by way of example.

Object A contains the attributes name, address, and birth date. Object B, in a subclass of A, contains the attributes salary and employee number. As mentioned before, each of

these attributes are individually located in a special object attribute node (**oattr_node**). These nodes are then linked together into a linked list. Thus, the attribute linked list for A will include Name, Address, and Birth Date. The pointer called **ocn_first_attr**, located within object A's object class node, will point to the first attribute on that list.

Object A's first subclass pointer (**ocn_first_subcls**) will point to the object-class node of B. A query requesting the name, address and salary of B will first traverse the list of A's attributes, and retrieve the attribute nodes of name and salary. Then it will drop to B via the subclass pointer, traverse the list of B's attributes, and retrieve the salary attribute node The linked list of retrieved attribute nodes for B consists of Name, Address, and Salary. The result is that the name and address attribute *fields* are inherited from A and linked with the attribute fields of B. Values for each of these fields are then found by accessing the ABDL record for object B.

The inheritance feature is ideally suited to take advantage of the hierarchical tree structure. Attributes of any object can be accessed by searching the tree for that particular object, then using the subclass pointers to access the attribute linked list. This method will *not* work, however, when a query involves objects from two different hierarchies. Traditional inheritance will not work because there is no defined linkages from one the hierarchy to another hierarchy. Without such a mechanism, the cross-hierarchical query processing is impossible. In other words, inheritance is a necessary but not a sufficient feature of the object-oriented data model. Hence the object-oriented model, as initially implemented on MDBS, is inadequate for realistic database scenarios. The covering property solves this problem by providing a mechanism for an object in one hierarchy to access data in another hierarchy.

B. THE COVERING PROPERTY ON MDBS

Implementation of the covering property of the object-oriented model into M^2DBMS on MDBS is the subject of this thesis. The details and specific methodology are included

in Chapter four. For continuity purposes, however, a brief overview of the implementation is given below.

The covering property is implemented into the existing object-oriented model. It uses the same constructs as inheritance, such as pointers and linked lists, to achieve cross-hierarchical mapping. Once the user logs onto the system and chooses the object-oriented model, he is presented with an option of "establishing a covering relationship." Once selected, the initial menu cascades into the covering menu. Here, the user is asked to name the covering object and select the class of objects to be covered. The user is then queried as to the number of levels in the hierarchy to which the covering relationship will apply. The number of levels determines the scope of the covering relationship onto the other hierarchy. Our new software then establishes the relationship and maps the covered objects to a database.

The database contains a listing of the covering construct's name, initial object, the target object, followed by all of the covered objects that have been mapped. Subsequent queries by one object of one hierarchy to an object of another hierarchy are first checked to see if they belong to the same hierarchy. If so, then pure inheritance will suffice to satisfy the query. If the objects belong to different hierarchies, then the data file is checked to see if a covering relationship exists that covers the other object. If such a relationship exists, then pointers are used to access the relevant hierarchy. The query is answered as usual in the object-oriented interface. If no covering relationship exists, then the query is refused and the user is notified that the requested query cannot be executed. The user then has the option of establishing a covering relationship or exiting the menu.

IV. IMPLEMENTATION OF THE COVERING PROPERTY

The covering property is implemented in the object-oriented data model on MDBS. Consequently, it is meant to supplement and enhance OODM. Every effort was made to use the existing data constructs and original code as written by Karlider and Moore [Karl93]. This was done in order to preserve continuity and data integrity. Where changes became necessary in the constructs, notations were made in the accompanying documentation describing the purpose and author of the code.

This chapter will detail the implementation of the covering property in two ways. First it will describe the changes to the data structure and then outline the additional procedures that make the covering work. Second, it will take the reader sequentially through the process using an illustrative example. Source code for the covering implementation is provided in Appendix B. A user's guide to OODM on MDBS with the new covering feature is provided in Appendix C.

The basic building block of the hierarchy is the object-class node (**ocls_node**). As described in the chapter III, this node contains the variables and pointers necessary to form subclasses and superclasses within the hierarchy. Two changes have been made to the **ocls_node** in order to form the covering relation. A new variable called **ocn_marked** and a new pointer called **cover_cls** have been added. Figure 11 illustrates these changes.

Ocn_marked is a boolean variable and is used when traversing the hierarchy. This switch, originally set to false, is set to true whenever an object in the hierarchy is visited in the search. The **ocn_marked** variable is set when a member of the covered class of objects is encountered in a hierarchical search. Once the search is complete or the system is turned off, **ocn_marked** returns to false. This allows additional or new covering relations to be formed using a "fresh" hierarchy.

The **cover_cls** pointer is used to point from the object initiating the covering relationship to the highest object in the hierarchy which falls within the scope of the covering relation. The actual height in the receiver's hierarchy is determined by the scope

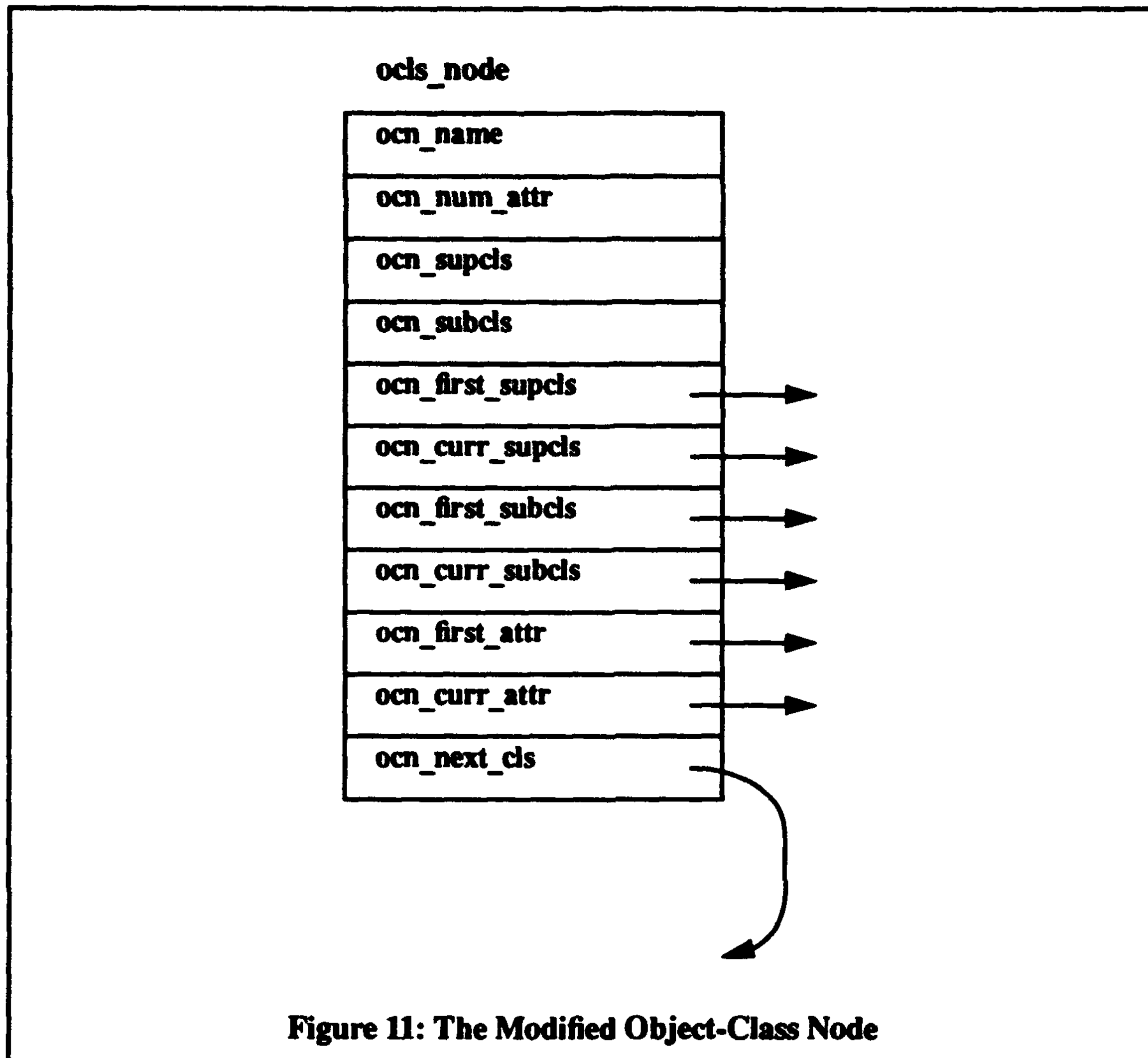


Figure 11: The Modified Object-Class Node

of the covering construct, which is specified by the user. The **cover_cls** pointer can only point to another node of the type, **ocls_node**.

A. NAMING THE COVERING RELATION

When the user decides to form a covering relation, the system will prompt him to name the relation. The name serves two purposes. First, it gives a logical meaning to the linking of the two separate hierarchies. Second, it allows for the formation of multiple covering relations, each with different scope, purpose, and name. An example of such might be In-law, Business, and Task Group. In-law would list those objects grouped by virtue of the

marriage relationship; Business would list objects grouped by means of a partnership or other business relationship; Task Group would list objects, such as particular ships, that have been grouped together to form a naval task group.

However, several relations may have the same name. There may be a need for multiple relationships; each with a different grouping of objects for the same named business partnership. This feature is possible by resetting the **ocn_marked** variable to false after each covering relationship is built.

Once the name is entered by the user, it will become the first item in a list stored in the data file called **<FILE NAME>.cover**. Subsequent items on the list are the "from-object", the "to-object", followed by all other objects covered in the scope of the newly formed covering relation. This list is maintained in a text file for future access during query processing. Each item of the list is separated by spaces to facilitate parsing during the query phase.

B. ESTABLISHING THE RELATION

The basis for the covering construct is the *defining relation*, which links the two separate hierarchies together. This defining relation is what was named by the user as described above. In order to create this relation, the following pointers are used:

```
struct ocls_node     *from_ptr,
*to_ptr,
*from_rel_ptr,
*to_rel_ptr,
*temp1,
*temp2;
```

1. Establishing, originating and terminating objects.

After naming the relation, the user is prompted to enter the name of the object from which the covering will be originated. This is called the from_obj. After entering the object's name, the user is prompted for the name of the object to which the covering relationship will apply. This is called the to_obj.

The **ocls_node** linked list is then traversed. At each node, the first attribute, called **ocn_name**, is examined. If the name matches **from_obj**, then the pointers **from_rel_ptr** and **temp2** are both assigned to that particular **ocls_node**. If the name does not match, then the search moves to the next node via the **ocn_next_cls** pointer. The search continues until either the name is matched or until the end of the list is reached. If no match is found, the pointers are nullified and the user is prompted to create a new covering construct.

If **from_obj** is found, the list is again traversed from the beginning in order to search for **to_obj**. The same method is used in the search. Once the node is found, the pointers **to_rel_ptr** and **temp1** both point to **to_obj**. To illustrate, suppose **to_obj** was named Paulla and **from_obj** was named Todd. After the search, the pointers are positioned as shown in Figure 12.

2. Searching for objects of the same hierarchy.

Once the two objects have been identified, they are checked to see if they belong to the same hierarchy. If they share a common hierarchy, there is no need for a covering relationship because inheritance will suffice to satisfy the query. To determine this, the pointers **to_ptr**, **from_ptr**, **temp1**, and **temp2** are used.

Beginning at **to_obj**, the hierarchy is traversed "upward" via the node's **ocn_first_supcls** pointer. If the object has a superclass, then **to_ptr** gets assigned to the superclass. The pointer **temp1** trails one level below **to_ptr** throughout the upward traversal of the hierarchy. This continues as long as there remains a superclass to the object being visited. By definition, the top of the hierarchy contains no superclass so that object's

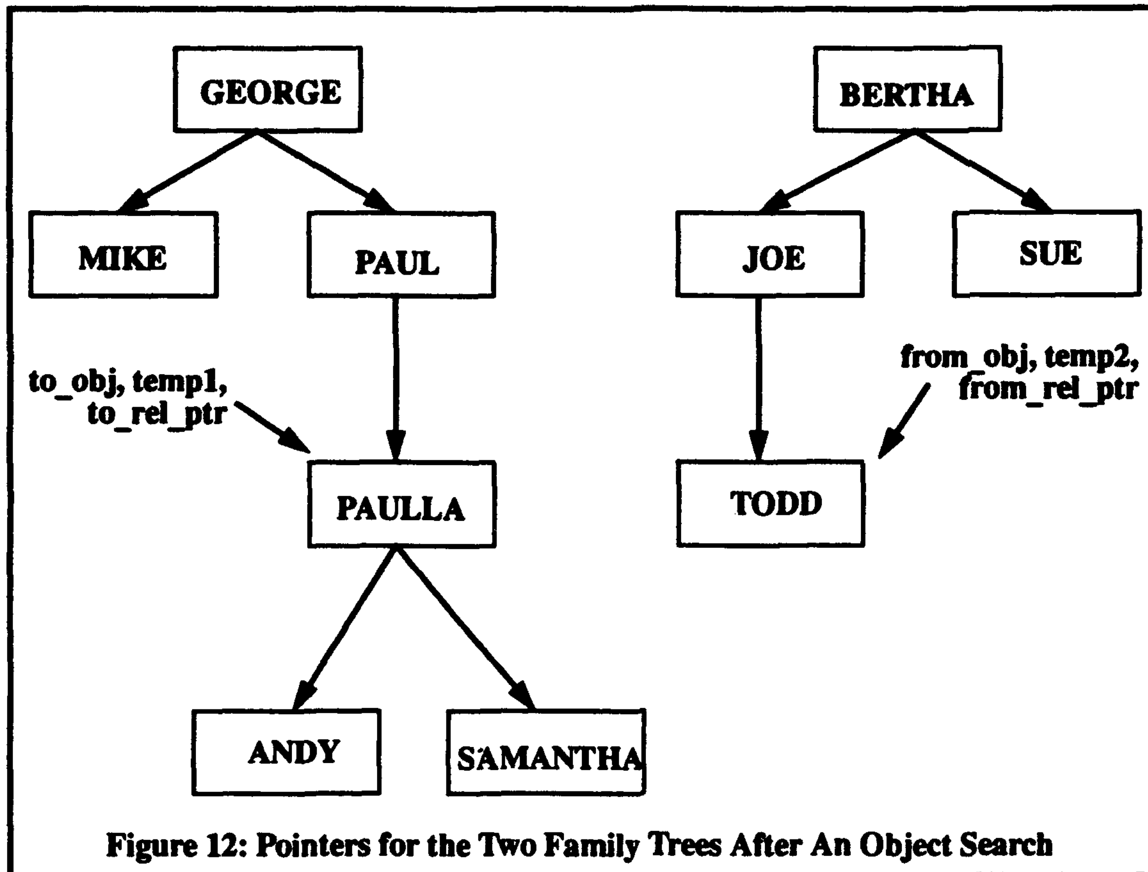


Figure 12: Pointers for the Two Family Trees After An Object Search

ocn_first_supcls pointer is set to null. Because of this, **to_ptr** will also eventually be set to null, leaving **templ** pointing to the top-most node.

An identical upward traversal of the tree containing from_obj is then accomplished. At the end of the hierarchy, temp2 will also point to the top-most object in the hierarchy. A comparison is then made between templ and temp2. If they are identical, then both to_obj and from_obj lie within the same hierarchy and a covering relationship is not required. The user is prompted as such and asked to begin anew. If they are not identical, then it forms the basis for a legitimate covering construct and the process continues.

At this point, to_rel_ptr and from_rel_ptr each still point to the objects defining the relation. Using these pointers, the user is then asked to confirm that he wants the covering relationship to be formed between the two objects. If the answer is yes, then, using the pointers to access the nodes, the ocn_marked variable is set to true in each of the two

nodes. This prevents adding the objects to the list of covered objects once the hierarchy is traversed. The two variables, from_obj and to_obj, are then sent to the covering data file and stored as the second and third item in the list respectively. If the user selects "no", then he does not wish to continue to create the covering relation. In this case, all pointers are returned to null and the covering menu is re-displayed.

C. ESTABLISHING COVERING

Although conceptually a link now exists between the two hierarchies in the mind of the user, in M^2DBMS no physical link exists. As such, inheritance is still restricted to one hierarchy. The final process of creating the covering relation involves determining the scope of the covering and accessing the relevant objects and establishing the physical links for it in M^2DBMS.

1. Determining the Scope.

The user is then prompted to determine the scope of the covering relation. The scope is defined as the number of levels in the hierarchy above and below **to_obj** which are to be encompassed by the covering relation. Figure 12 illustrates the concept of scope. In this illustration, the scope of the covered object includes one level above and one level below the covered object. The user is first prompted for the number of levels above **to_obj**, followed by a prompt for the number below **to_obj**. Any number of levels, including zero, can be chosen by the user for each entry. The system will automatically adjust for situations where the object is linked to fewer subclasses or superclasses than that specified in the scope.

2. Marking the covered objects.

The pointer **to_ptr** is then used to traverse the hierarchy above **to_obj** in accordance with the level specified by the user. When the traversal reaches either the upper bound of the scope or the last superclass, the pointer **cover_cls** is set to that node. The

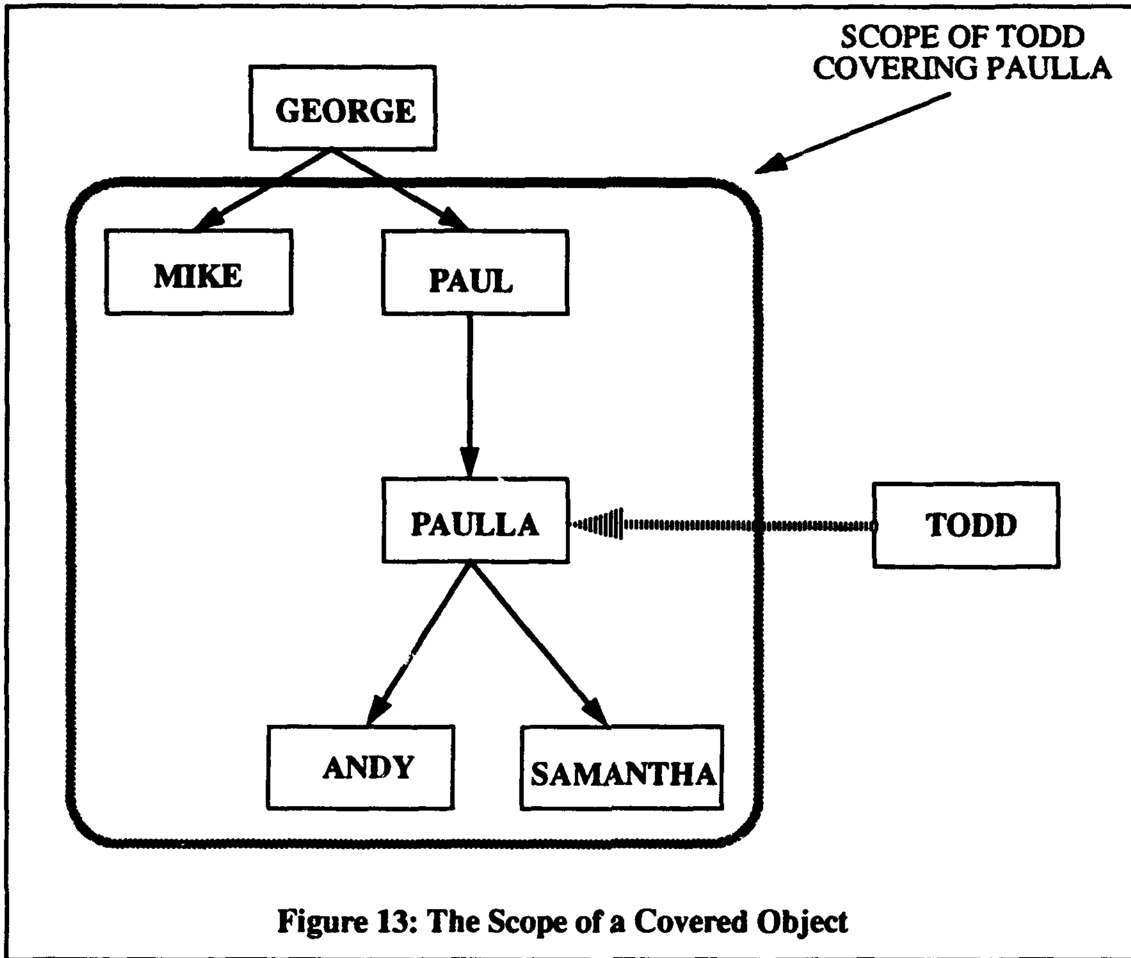


Figure 13: The Scope of a Covered Object

cover_cls pointer is now the starting point for collecting the objects to be covered in the scope.

3. Writing the data to the data file.

The first three items written into the covering data file are the name of the covering relation, **to_obj**, and **from_obj**. The relation name is taken from the entry by the user when the relation was named. The values for **from_obj** and **to_obj** are determined by accessing the attribute name of the objects pointed to by **from_rel_ptr** and **to_rel_ptr**, respectively. Once these three items are written to the data file, the covered objects can then be added to the list.

Beginning at the node pointed by **cover_cls**, which is still pointing to the highest object in the scope, the tree is traversed downward. The downward traversal continues until the lower limit of the scope is reached. Each node is first checked for the existence of subclasses that fall within the scope. Once the subclasses have been checked, the search moves to the next node within the same level.

As each node that satisfies the scope conditions is visited, the variable **ocn_marked** is set to true. This setting tags the object for membership in the covered relation and the data are then sent to the data file. It also prevents double-counting when the tree is traversed upward in order to move to a subclass of the next object class. For instance, the node belonging to **to_obj** is skipped because its **ocn_marked** variable is already set to true. Once the lower limit of the scope is reached, all covered objects meeting the scope criteria have then been written to the data file. Following the writing of data to the data file, the hierarchy is traversed again to reset all of the **ocn_marked** variables to false. Figure 14 shows an example data file. Compare this example to the scope of the covering relationship shown in Figure 12.

FILE: FAMILY.cover

IN-LAW TODD PAULLA MIKE PAUL ANDY SAMANTHA

@

Figure 14: Example of a Covering Data File

D. ACCESSING THE DATA VIA QUERIES

Queries to the database using the covering property illustrate the power and flexibility of the covering relationship. Since queries on the covering property of one hierarchy to the other involve these hierarchies, it is essential that the query format be compatible with

object-oriented queries already in place in the model for the inheritance property. With this in mind, queries to the database using the covering property are constructed almost identical to those without the covering property.

The Object-Oriented model on MDBS as implemented by Karlider and Moore [BENV 91] uses the following format to make queries on the data:

<object name>. retrieve <attribute 1>, <attribute 2>,..., <attribute n>

The object name and the attributes in question are specified by the user. The query is either read from the screen in this format or stored with other queries in a *request file* from which the user can choose queries from a list.

The following modification was made to the query format to handle queries using the covering property:

<(from object.defining relation)> <object>.retrieve <attribute 1>,..., <attribute n>

Queries constructed using this format are converted to the original format before processed in the ABDL transaction. Each query is intercepted in the KMS module where it is transformed before being sent to the kernel database system, i.e., KDS. Inside the KMS module, each query is first examined for format. Queries involving covering are then parsed and checked for compliance with the covering relationships found in the data file. This procedure acts like a filter, transforming the covering query into a straight inheritance query. Compliance with the procedure satisfies a boolean variable, which enables the query to pass to the KDS as if it was a normal inheritance query. The major difference is that one object outside the normal inheritance structure is now accessing objects in another hierarchy.

The transformation process operates in the following manner. Each query is initially parsed to check for parentheses. The presence of parentheses signals a covering relationship query. First, the defining relation found inside the parentheses is compared to the covering names, each of which is the first item in each list found in the covering data file. If it does not match any of the covering names, then the query is rejected because no covering relationship exists to satisfy the query.

If the defining covering matches a covering name, then **from_object** in the query is compared to **from_obj**, the second item in each list in the data file. Again, failure to match results in rejection of the query. However, if the names do match, the balance of the list is scanned to check for a match with the query object. A successful match here enables the query to proceed.

The query sent to the ABDL translator is now in the original format, equivalent to the covering query minus the parenthetical portion. From there, the query is handled by the ABDL translator in the normal manner. The linked list of each object is traversed searching for a match. If found, the attribute linked list of each object is then scanned, matching the query attributes with those found in the object. These objects and their corresponding attributes are collected and compiled in the ABDL form. After the translated transaction has been executed by MDBS, the file of data satisfying the query is then sent back to KMS for conversion back to the object-oriented form.

V. ADDITIONAL MODIFICATIONS AND LIMITATIONS OF THE OBJECT-ORIENTED INTERFACE

A. ADDITIONAL MODIFICATIONS TO THE OBJECT-ORIENTED INTERFACE

In addition to implementing the covering property, we incorporated the ability for the object-oriented interface to generate object-ids. The object-id is set up as an attribute for each object, and each record within that object requires a unique object-id. Prior to modification, the user was required to manually assign an object-id to each record. In addition, on-line insertions required that the user know in advance which object-ids had already been used in order to avoid duplication. In a large database, this can become a problem if careful attention is not paid to keeping track of object-ids.

Our modification uses the system clock to generate the object-id at the time of record loading. The basic strategy used is to retrieve the Unix system time in integer format and convert it to a string. Conversion to a string is required because the Lil module of the object-oriented interface reads all attribute values as strings from a text file before passing it on to the kernel.

The advantage of using the system clock is twofold. First, each record loaded is essentially time stamped, which gives each record a unique time. Therefore, by definition, the object-id is also unique. Second, the use of system time allows records to be clustered on the data disk according to their load sequence rather than the value of an attribute other than the object-id. This is a desired characteristic for secure database systems because intrusions into the database do not reveal the contents of a cluster data base on its grouping.

The procedure used is called **get_objectid()** and returns a character string. It is called by the **mss_load()** procedure in the **Lil** directory of the object-oriented interface. The mass loading procedure is responsible for loading records into the database. As the mass load procedure scans each line of the record file, it parses the first attribute of each record, verifying it as the object-id. If the first attribute is the object-id, **get_objectid()** is called and

returns the new object-id for the record being loaded. This in turn is passed to the kernel and stored in the database. The object-id attribute is still maintained in the schema because it is an attribute of the record. Within the record file, the object-id is still used for insertion as a place holder to avoid confusing the system when parsing the insert request. However, all the object-id values can be the same number because they will be reassigned at load time. The source code for this feature can be found in Appendix B.

Several implementation details arose in the writing of this procedure which are hardware and software related. First, the Unix system time is tracked in elapsed seconds since January 1, 1970. This time can be accessed down to the microsecond level. In order to get a fine enough time slice for the object-id assignment, we needed to access the system time at the microsecond level. This was necessary in order to prevent duplicate object-id assignments when loading records. Storing a number this size would exceed the 32-bit registers on the current platform. Given a machine with 64-bit registers, this problem would be eliminated.

In order to work with 32-bit registers, we modified the output of the system time into an eight digit number. The number assigned to the object-id is in the format of <**hhmmss*uu***>, where **hh** is for hour of the day (24 hour clock), **mm** is for minutes in the hour, ss is seconds in the minute, and ***uu*** is thousands of microseconds.

Unfortunately, due to the limitations of the time slice, testing revealed several cases of duplicate object-ids. To correct this problem, a delay loop was added to the procedure to delay assignment of the object-id. This resulted in zero duplication of object-ids and the time delay is not noticed by the user and does not affect system performance. The biggest drawback to this solution is that non-duplication of object-ids can only be guaranteed for a session which lasts less than 24 hours. However, since we are not producing a commercial database, and data is not maintained in the database beyond each session, this did become a problem during operation.

The second implementation issue is software related. The current C compiler is an older version dated prior to the ANSI C standard. This version contains no library functions

which convert an integer into a character string. To overcome this problem, each digit of the time slice is fed into an integer array. After loading the array, each field of the array is read and a character version of the digit read is copied into a corresponding character array. The resulting character array is returned to the **mss_load** procedure and assigned to the object-id. The incurred overhead is not noticeable to the user and can be eliminated when the software is ported to a system with a compiler having the required libraries.

To recapitulate, the automatic assignment of object-ids is a vast improvement over the method of manual assignment. When the software for the object-oriented interface is ported to a newer, more powerful system, the implementation drawbacks discussed above can be eliminated.

B. LIMITATIONS

In addition to the limitations discussed in the section above, there are some significant limitations to the object-oriented in general and covering in particular. The object-oriented interface does not have the capability to conduct on-line deletion and modification of records. Deletion and modification to records must be done off-line by editing the record and schema text files.

Additionally, methods (or actions), as described in Chapter I, are not implemented. Inheritance of actions as well as attributes is a powerful feature of the object-oriented model. Allowing user-defined actions to exist within an object class is essential if the full benefits of the object-oriented data model (OODM) are to be utilized.

The object-oriented interface cannot retrieve a record without specifying the unique object class associated with it. This is a major limitation of the current system because it requires the user to know in advance which object class contains the desired object (or record). Ideally, the system should be able to process the following query:

retrieve firstn, lastn, salary if ssn= 283764958

This query asks for the first name, last name, and salary for the individual record with the social security number 283764958. Under the current configuration, this query requires the user to specify a particular object as shown below:

Joe.retrieve firstn, lastn, salary if ssn= 28376495

Because of this restriction, the current implementation requires the use of object classes with single instances only. The end result is that each instance is actually an object. Unfortunately, this prevents the covering property from being fully utilized. If multiple-instance objects were possible, then covering could map from one instance in an object class to many instances of one or more object classes.

Consider an object class called STUDENT and an object class called COURSE. Each student is enrolled in multiple courses, while each course contains multiple students. A query to retrieve the course load of student Jones requires a covering relationship between one instance of STUDENT(Jones) to many instances of COURSE. If a multiple-instance covering relationship is called ENROLL, then the query would appear as shown:

(ENROLL.Jones. COURSE) retrieve coursename

This query will retrieve all courses enrolled by Jones. Unfortunately, under the present system configuration, this query cannot be performed. The best that can be achieved is to make each instance of STUDENT and COURSE individual objects. Then a covering relationship called ENROLL must be defined linking the object class JONES with the group of covered object classes.

Another limitation of the object-oriented interface is the inability to perform a multi-class join with more than two objects. The original design of the object-oriented interface allows for an attribute of one object in an object hierarchy to point to an object class of another object hierarchy. In a sense, this is covering, but it is covering which is hard coded at the schema design level.

In order to perform a join under the original design, a data structure called **oki_tgt_cls1** and **oki_tgt_cls2** [Karl93] is used to retain two values for object names. The purpose of these variables is to store the names of the objects used in a join for access

during retrieval. The two-object limitation discussed earlier is a result of only these two variables being present. A possible solution to this is to change this data structure into a dynamic one which can grow with the number of objects in a join. However, the need for hard coding the relation between two hierarchies diminishes because covering can now be done on the fly. The idea of a dynamic data structure still holds merit and will be discussed further when we review the limitations of the covering property.

The covering implementation in the object-oriented interface cannot be applied to objects across different databases. MDBS allows for loading multiple databases in a single session. However, only one database can be accessed at a time. If the need exists for accessing data across different databases, the object-oriented database must be able to provide simultaneous processing of databases. The covering construct would then need the name of the desired database in addition to the current information normally needed to access objects of different hierarchies.

Another limitation of the covering implementation is that the covering construct can only retrieve data from the object being covered. It cannot retrieve data both from the covering object and the covered object. The absence of this limitation would allow the system to perform joins by taking advantage of the current data structure. However, as discussed above, this join would be limited to only two objects. Therefore, since one object can have multiple covering relationships, the implementation of the proposed dynamic data structure would be of great benefit.

The limitations discussed above are provided to inform the reader as to what is needed to make the object-oriented interface a production-ready database interface. However, our work here is to prove the feasibility of the object-oriented data model and in particular the implementation of the covering property. These limitations did not interfere with our ability to prove these points.

VI. FUTURE WORK

There are several issues for future work which concern both the object-oriented interface on MDBS and the covering property. While the object-oriented interface is sufficient for research purposes, it is not ready for real time use. Much of the future work is related to the limitations listed in the Chapter V and will be touched upon again in this chapter.

As discussed in Chapter V, the object-oriented interface on MDBS does not have the capability to conduct on-line updates and deletes. Incorporating these operations into the object-oriented interface would complete the model. In addition, this capability would be a significant step towards a real-time user interface.

Incorporation of methods (or actions) into the object-oriented interface is still needed. The current model on MDBS allows only the inheritance of attributes. The addition of methods to the interface enables the user to take full advantage of the power of object-oriented data model (OODM). This would allow CAD/CAM databases to be implemented as well as other simulation based applications.

While we eliminated the need for the user to supply and keep track of object-ids, the user must still name a specific object in the query. This precludes the covering of one instance of an object class to many instances of another object class. This limitation is discussed in depth in Chapter V. Removing this limitation is required before the full benefits of covering can be achieved.

Another area for future research, also discussed in Chapter V, is to remove the limitations on two class joins. This would significantly expand and enhance the capability of the model. Furthermore, the covering construct should be enhanced to allow for single command queries using multiple covering relationships. In conjunction with this, the ability to retrieve data via covering from objects in other databases would also make the system more robust. This requires simultaneous processing of multiple databases as well as safeguards to ensure the security and integrity accessed databases are maintained.

The current configuration of the object-oriented interface allows the user to set certain conditions for a search. However, this is limited to the "and" operation and does not provide for an "or" operation. The disjunctive "or" is convenient when searching for multiple instances within the database. For example, the following request is presently allowed by the object-oriented interface:

george.retrieve firstn, lastn, salary if salary < 100000 and lastn = "Smith"

This query requests the first name, last name and salary of all objects whose salary is less than 100000 and whose last name is "Smith". If we wished to retrieve all records where salary is greater than 1000000 *or* whose last name is "Smith", we would have to retrieve the data with two requests instead of one. The proposed request would be in the following format:

george.retrieve firstn, lastn, salary if salary < 100000 or lastn = "Smith"

The current configuration of the object-oriented interface also lacks a text retrieval capability. This feature is useful in many search applications., such as finding particular paragraphs in classified documents. With this capability, the object-oriented interface would be an exceptional platform for maintaining a multi-level security database.

Porting the source code for the object-oriented interface to a programing language such as C++ or Ada would reap immense benefits. Encapsulation of code using one of these languages improves the modularity and maitainability of the program. By taking advantage of the sophisticated libraries in these languages, code becomes more streamlined and easier to maintain. Using an object-oriented programming language, such as C++, provides encapsulation and modularity to the program throughout the entire design process.

There is also a need for more sophisticated exception handling capabilities in the object-oriented interface. Greater depth of error control pays dividends in keeping the system on line. Often times once a mistake is made, it is impossible to correct without stopping MDBS, zeroing the processes, and restarting the system. One advantage to Ada is that it provides this simplified exception handling ability within its standard libraries.

The transition from research product to commercial product normally focuses on improvements that make the product more user-friendly. The conceptual design is proven, but the interface is weak. In view of this, the issues discussed above are primarily provided as suggestions for making the object-oriented interface ready for real-time use.

VII. CONCLUSION

This thesis focused on the implementation of the covering property of OODM on MDBS. First, a case was made for the value of the object oriented model in general. Unlike the traditional models of database design, the object-oriented model avoids arbitrary modeling while allowing greater abstraction. At the same time, OODM provides powerful constructs, such as inheritance and covering, which yield flexibility and modularity not available in other models. These factors make OODM an ideal data model for database design.

Second, the case was made for the use of the federated database structure within the context of MDBS. Federated databases allow greater local autonomy and provide the user with transparent access to diverse databases. The multimodel/multilingual system software on MDBS is one means to achieve rapid access to databases using a variety of models and languages. Using MDBS as the development platform allowed design and implementation to concentrate on the object-oriented interface without having to develop a new DBMS from scratch. Finally, with this background, the actual implementation of covering was described in detail.

Covering required the modification to the data structure used in MDBS as well as the creation of a user-defined relation between the hierarchies. Depending on the existing data structure, changes to the data structure may or may not be necessary. In this case, the modification was necessary, and consisted of adding an additional pointer and a boolean attribute to the object class node.

Before two or more hierarchies can be linked together, a covering relation must be defined. Without this relation, there is no meaning to the covering property. Covering simply becomes an arbitrarily defined construct that has no relevance to the database. Furthermore, it cannot be reproduced systematically with meaning from one user to the next. By naming each covering relation, multiple coverings can be created on the same database.

Finally, the covering relationship must be *user-defined.* This accomplishes two objectives. First, it removes the notion of arbitrariness from the database and imposes clarity and meaning. A covering property defined at the time of database creation, is a static structure, defined by a programmer who may not understand the needs or purposes of the database. The user, in contrast, has a specific need for the covering relationship and has specific uses for the information. Therefore the user is much better suited to create and modify the covering property. Covering now becomes a dynamic construct. Second, the scope of the covering must also be defined by the user. Incorporating these objectives allows the user to tailor the database to his particular needs and provides flexibility and clarity when forming queries.

This thesis demonstrated that it is possible to implement a covering relationship across two hierarchies within an existing object oriented database structure. The actual product is a working database which illustrates the covering property. The implementation of covering onto MDBS was accomplished as proof of viability of a specific concept. Like the multimodel/multilingual software on MDBS, however, it is currently not a working program suitable for commercial use. With this in mind, it is worth exploring alternative applications of the covering property beyond the family tree analogy.

A. COVERING APPLICATION ONE: A NAVAL TASK FORCE

A realistic application of the covering property within a military context is the creation of a task force. The task force structure, in widespread use within the U.S. Navy, has applications to the Navy as well as other services. Furthermore, the inherent hierarchical structure of particular classes of ships makes it not only an ideal example, but also an appropriate and applicable use for the covering property.

Carrier battle groups are normally composed of twelve ships, with the aircraft carrier serving as the flagship and centerpiece of the task force. Each ship within the task group serves a specific purpose and each brings their unique strengths and abilities to the battle

group. Yet, when the task force is constructed, each particular ship is drawn from their own respective classes and not from a task group hierarchy. In other words, the task group is created from existing ship hierarchies.

For example, the carrier is drawn from the object class *CARRIER*, which is further divided into the object classes *CONVENTIONAL* and *NUCLEAR*. Within each of these object classes are specific classes of carriers: the *NIMITZ*, *KENNEDY*, or *ENTERPRISE* nuclear carriers and the *KITTY HAWK*, *FORRESTAL*, or *RANGER* conventional carriers. Each of these object classes contains the specific ship objects for their respective class. Similarly, each cruiser, destroyer, and frigate chosen for the task force comes from their own respective class hierarchies. An example of the *CARRIER* object class is illustrated in Figure 15.

The task force is essentially a collection of ships drawn from unrelated hierarchies. Thus, the creation of a task force is actually the creation of a specific covering construct. For example, Battle Group Bravo, as defined by higher authority, might consist of the following ships:

CARRIER: **CVN - 72**

CRUISERS: **CG - 52**

DESTROYERS: **DD - 963, DD - 982, DD - 983**

FRIGATES: **FFG - 7**

Battle Group Bravo, as defined by the Battle Group Commander, is the covering property that links the various hierarchies together. The scope of the covering is defined when the admiral chooses the ships that will make up the task force. The creation of Battle Group Bravo from the various ship hierarchies is illustrated in Figure 16.

Prior to the creation of the task force, each object in its respective class hierarchy is able to access data on any object in the same hierarchy through the use of the inheritance property. For example, the phalanx anti-missile gun mount is found on every aircraft

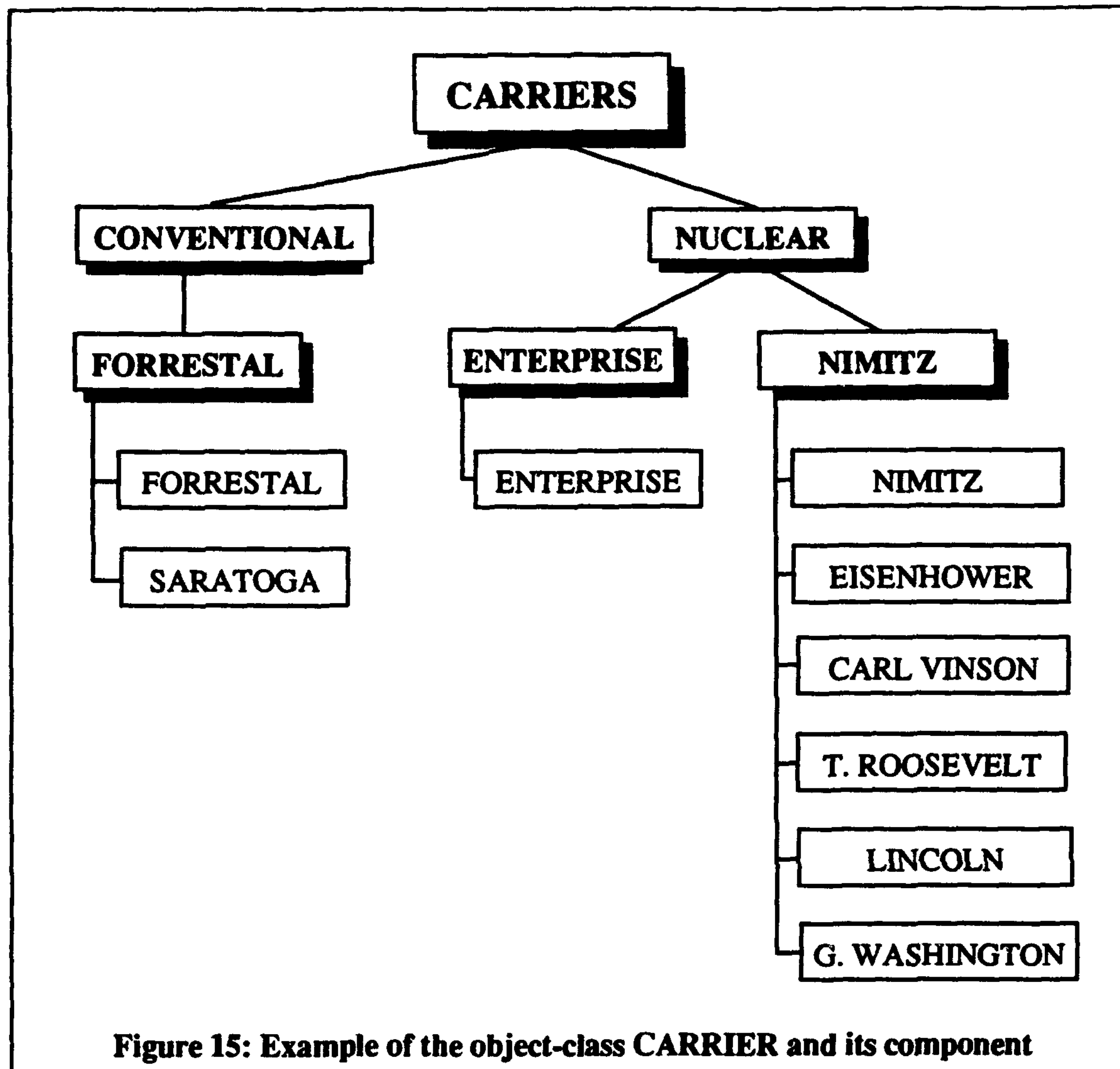


Figure 15: Example of the object-class CARRIER and its component

carrier. If parts or ammunition for the phalanx are required, the carrier hierarchy could be searched using inheritance to find spare parts among the other carriers. However, the phalanx is found on other ships as well. Without covering, the parts inventories of the other ships could not be accessed from the carrier. This is because inheritance does not cross hierarchies. Since the parts inventory can be accessed through the inheritance hierarchy of each object class, accessing the hierarchy via covering allows access to the respective inventories.

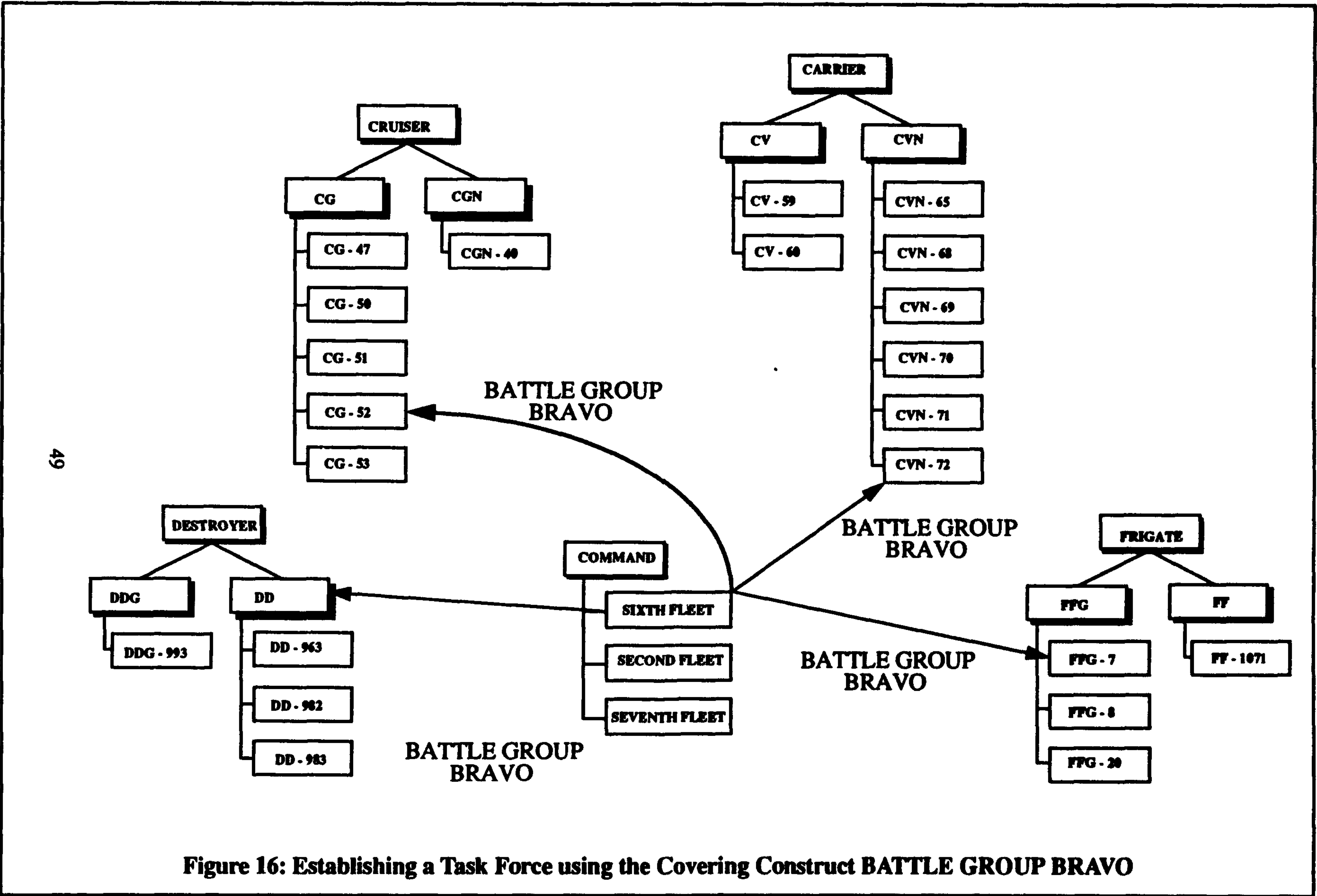


Figure 16: Establishing a Task Force using the Covering Construct BATTLE GROUP BRAVO

B. COVERING APPLICATION TWO: MULTI-LEVEL SECURITY

A second application involves the access requirements of multi-level security arrangements normally found in the military environment. OODM is ideal for supporting the multi-level security policy and its associated access control requirements [Hsia91b]. This is because OODM properties of inheritance and covering provide the necessary mechanisms to preserve security constraints while still allowing access to the data.

All documents within the military are classified according to four primary security classifications. These classifications, listed from lowest to highest classification, are *UNCLASSIFIED*, *CONFIDENTIAL*, *SECRET*, and *TOP SECRET*. As the name implies, there are no restrictions to unclassified documents. However, in order to access information in documents other than unclassified, two conditions must be satisfied.

One, the person must possess a personal clearance level equal to or higher than the clearance classification of the document. For example, a secret clearance satisfies this requirement for secret and confidential documents. Two, the person must have a need-to-know the information. Both conditions must be met before access can be granted. This is known as *compartmentalization*. Therefore, simply possessing a secret clearance does not entitle a person access to any secret or confidential document. Together, the clearance level and the need-to-know policies govern access to classified information.

OODM, using inheritance, naturally supports the need-to-know policy and access control requirements for unclassified data [Hsia91b]. Within the object-class hierarchy, there exists an owner/subclass relationship at each level. Attributes and methods (or actions) are passed down to subclasses via the inheritance property. Restrictions on access to data, namely the need-to-know policy, are manifested in the attributes and actions that are passed to the subclass. This is analogous to the notion of the *view* in the relational model, where the system filters the data to the user according to the parameters defined by the owner. The advantage of OODM is that it avoids the "triggering mechanism" used to maintain data integrity in other models.

While inheritance will suffice for meeting the access control requirements for unclassified data, it is insufficient for classified data. This is due to the "read down/write up" operations required in the multi-level security policy:

> The *read-down operation* allows a user of the classified database to read all the data whose classifications are either below or identical to the clearance of the user. The *write-up operation* allows a user of classified data to write into the database a piece of classified data whose classification is either above or identical to the clearance of the user [Hsia91c].

The read down operation may be thought of as mapping a cleared user to classified data. Consider an object, called USER, with an attribute CLEARANCE LEVEL. A user with a top-secret clearance level maps to all documents classified top-secret, secret and confidential and unclassified; a secret user maps to all documents classified secret, confidential and unclassified; a confidential user maps only to those documents confidential or unclassified. Figure 17 illustrates this concept [Hsia91c].

Similarly, a write-up covering maps cleared users for writing data into classified documents. However, because of the nature of the write-up policy, the higher the clearance level of the user, the narrower the mapping of the write-up operation. This peculiarity is due to the intent behind the write-up operation. The write-up operation is constructed so that high clearance users are not wasting time inserting data into databases at a lower classification than their own. It is important to note that the write operation allows the user to insert data into the data base, not write-over existing data. The write-up covering is illustrated in Figure 18 [Hsia91c].

Once the covering operation is defined (either read-down or write-up), the scope of the covering determines the structure of the *read-down hierarchy* or *write-up hierarchy*. The method that will perform either the read-down or write-up operations will be contained in the object CLASSIFIED DATA. However, the read-down method can only be performed in the read-down hierarchy, while the write-up method can only be performed in the write-up hierarchy. The methods operating within the defined covering hierarchies establish the access control requirements for the multi-level security policy.

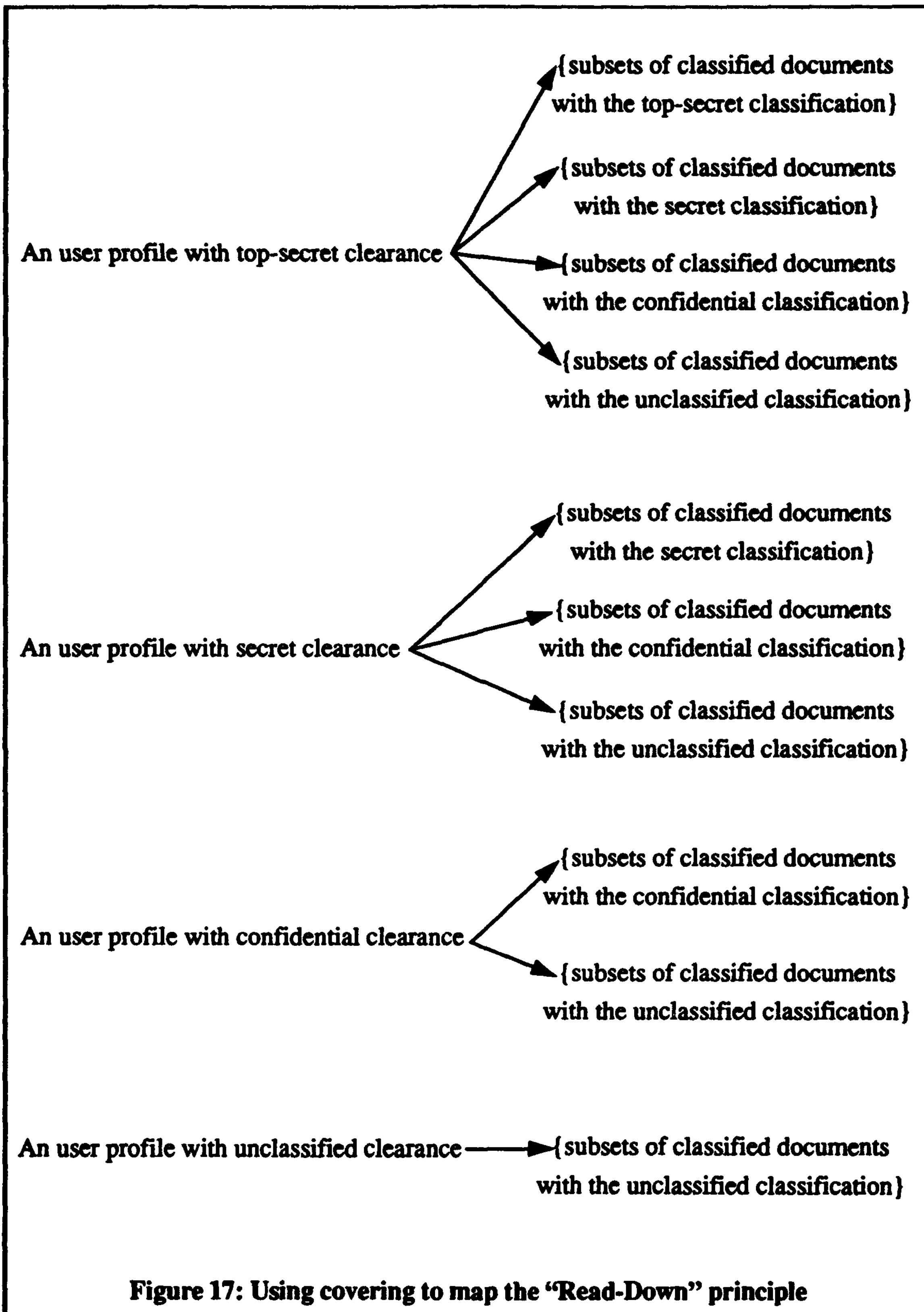


Figure 17: Using covering to map the "Read-Down" principle

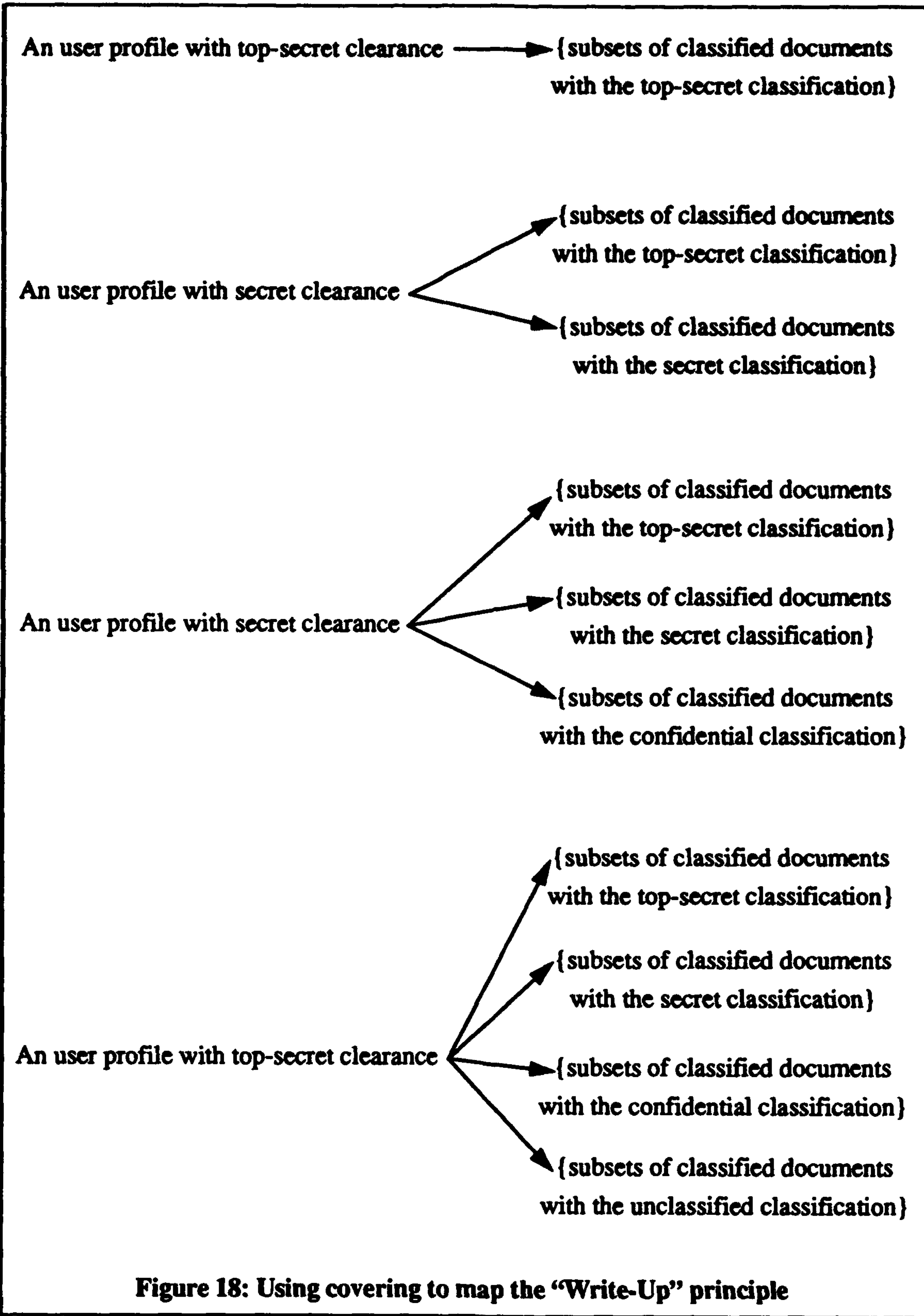


Figure 18: Using covering to map the "Write-Up" principle

C. SUMMARY

In conclusion, we have shown the utility and benefits of using covering as a means of enhancing OODM. OODM provides unique modeling capabilities which can be directly transferred into implementation. It has been shown that inheritance is a necessary but not a sufficient property for the OODM. Covering allows access between inheritance hierarchies within an object-oriented database. Without covering, access to objects is limited within the hierarchy of that object. Together, inheritance and covering meet the necessary and sufficient conditions to make the OODM a complete data model.

APPENDIX A. DATA STRUCTURES OF THE OODM

This appendix presents the data structures used in the object-oriented database and their relationship with one another.

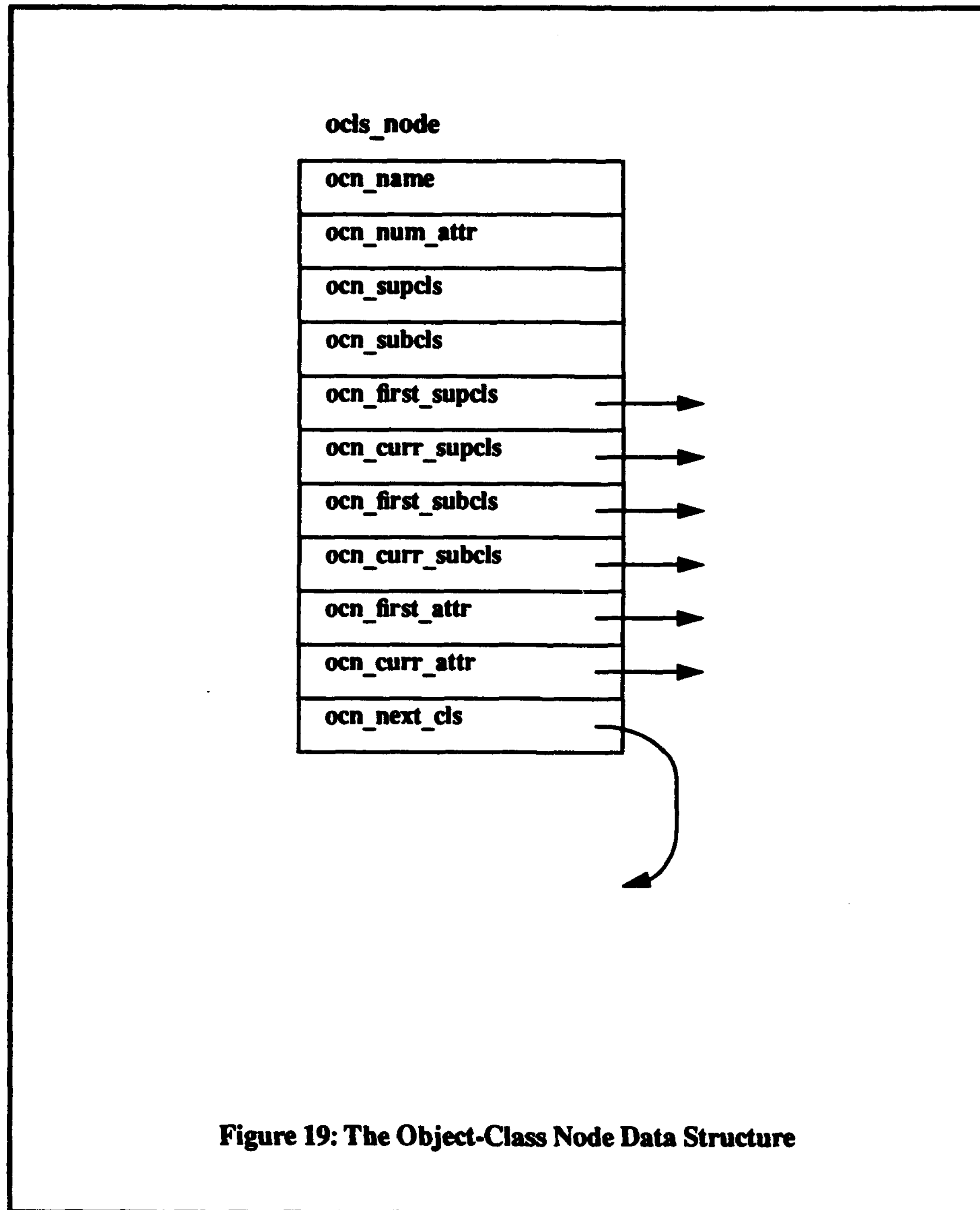


Figure 19: The Object-Class Node Data Structure

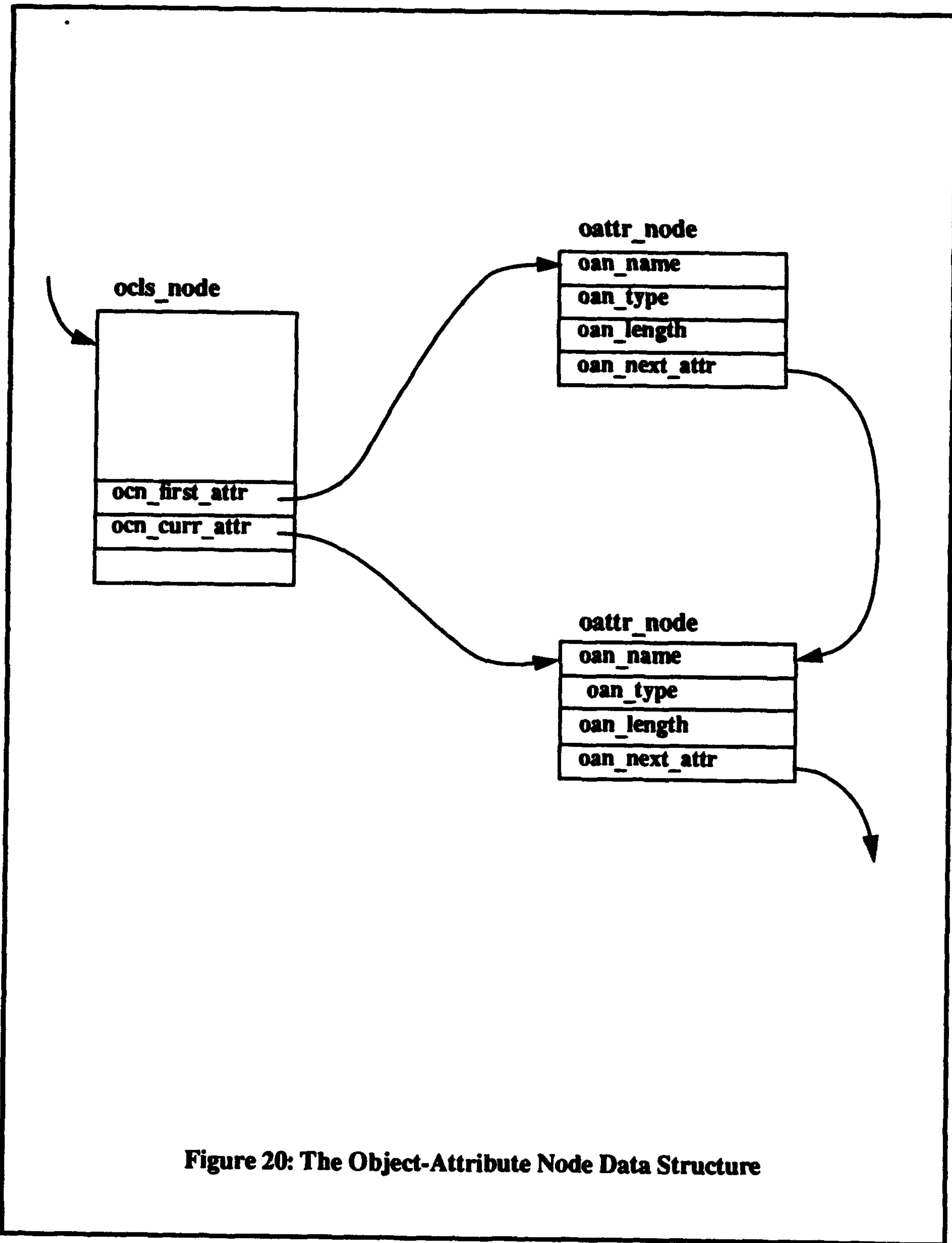


Figure 20: The Object-Attribute Node Data Structure

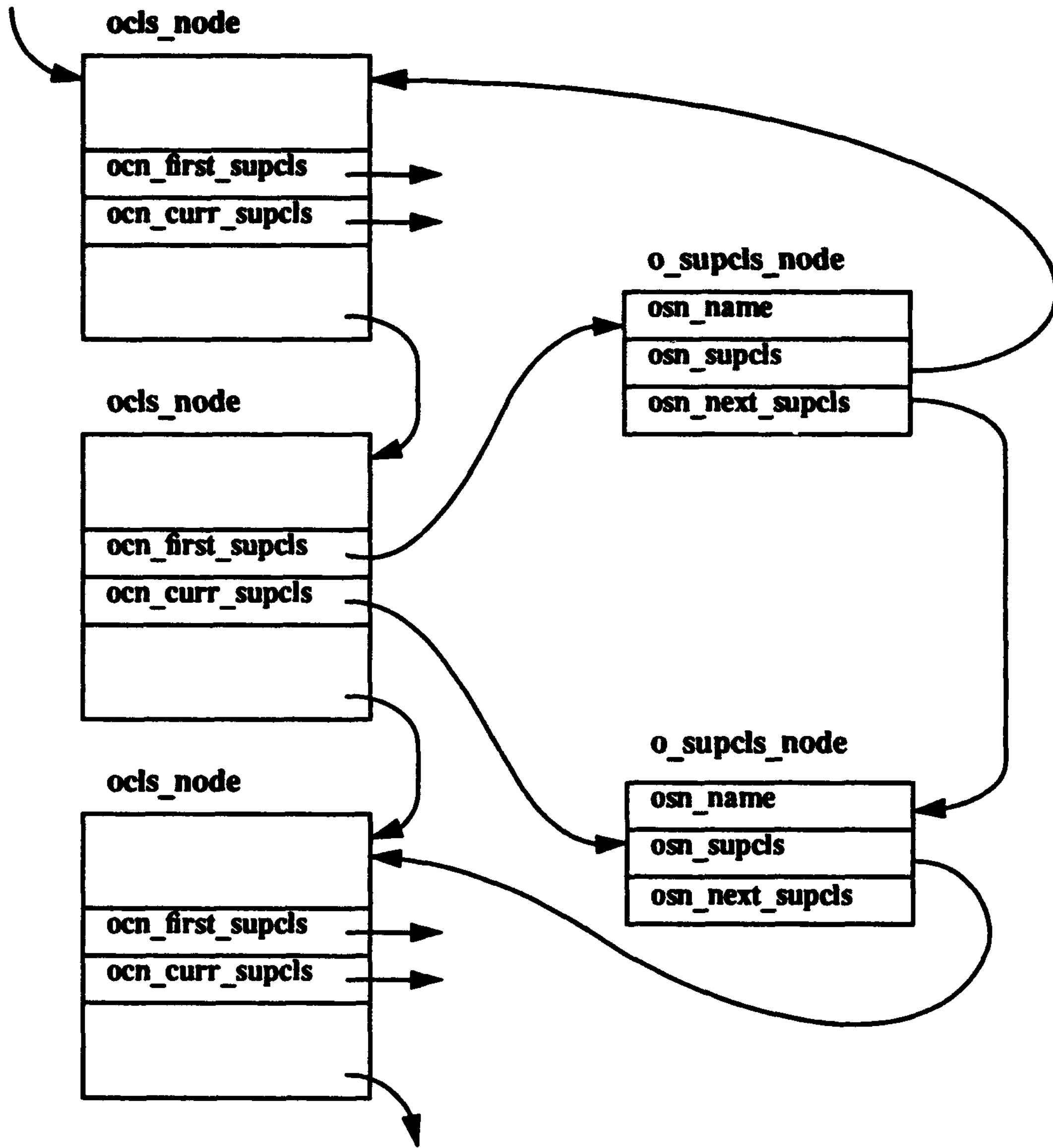


Figure 21: The Object-Superclass Node Data Structure

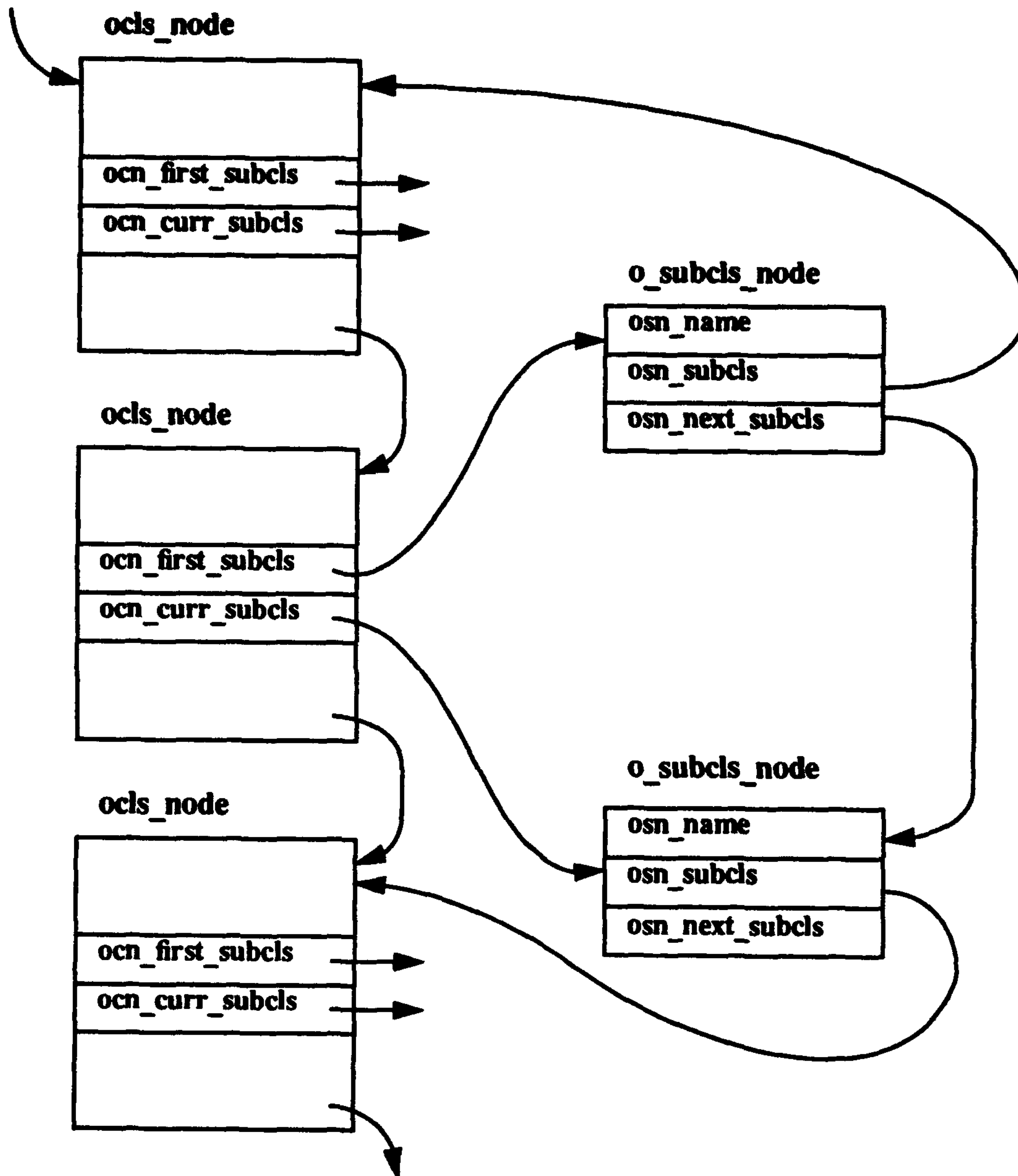


Figure 22: The Object-Subclass Node Data Structure

APPENDIX B. SOURCE CODE

A. SOURCE CODE FOR THE CREATION OF A COVERING RELATION

```
/*
 * $Header: o_cover.c,v 0.0 93/08/31 15:10 mdbs Exp $
 * $Source: /u/mdbs/rich/CNTRL/TI/LangIF/src/Obj/Lil/o_cover.c,v $
 * $Log:      o_cover.c,v $
 * Revision 0.0 93/08/31 15:10 mdbs
 * creation
 * Revision 1.0 93/08/31 20:55 mdbs
 * modified for INSERT statement accomplishment
 *
 * This file contains the source code for the implementation of the covering
 * property. Its main purpose is to allow the user to define a covering relation
 * by naming it and defining it in terms of end points of the relation and the
 * scope of covering. This file also includes procedures which display the
 * covering choice menu, allow the displaying of existing covering constructs,
 * and displaying objects available for the construction of a covering construct.
 * Source code for using the covering constructs in the retieval of data can be
 * found in the file obj_parser.c located in the Kms directory.
 */

#include <stdio.h>
#include <licommdata.h>
#include <ool.h>
#include <ool_lildcl.h>
#include "flags.def"
```

```
/* This procedure displays the covering menu which allows for the display or
 * construction of covering relations and display the objects available for
 * covering.
 */

o_cover_menu()
  {

  char  choice;
  int   num;
  int   stop;    /*Boolean*/

#ifdef EnExFlag
 printf("Enter o_cover_menu\n");
 fflush(stdout);
#endif

  system("clear");  /*Clear screen*/

  ool_info_ptr = &(cuser_obj_ptr->ui_li_type.li_ool);
  stop = FALSE;
  while (stop == FALSE)
    {
    printf("\nEnter type of operation desired\n");
    printf("\t(d) - Display Existing Covering Constructs\n");
    printf("\t(c) - Display Available Classes\n");
    printf("\t(e) - Establish a Covering Construct Between Two Objects\n");
    printf("\t(x) - Return to previous menu\n");
    ool_info_ptr->oi_answer = get_ans(&num);

    switch (ool_info_ptr->oi_answer)
      {
      case 'd' : /*View existing covering construct*/
              o_read_cover_file();
              break;
      case 'c' : /*View existing objects in current database*/
              o_display_classes();
              break;
      case 'e' : /*Establish covering relation betwen two objects*/
              o_get_objects();
              break;
      case 'x' : /*Exit menu*/
              stop = TRUE;
```

```
            break;
      default : /*Incorrect choice*/
            printf("\nError - invalid operation selected\n");
            printf("Please select a valid operation.\n");
            break;
      } /*End switch*/

    }   /*End while(stop==FALSE)*/

  system("clear"); /*Clear screen*/
      /*Return to previous menu*/

#ifdef EnExFlag
 printf("Exit o_cover_menu\n");
 fflush(stdout);
#endif

   }    /*End o_cover_menu()   */
```

```
/*This procedure checks for existing covering constructs. Opens and reads
"<dbname>.cover"*/
o_read_cover_file()
 {
 char              cover_file[FNLength + 3];
 struct obj_dbid_node   *db_ptr;
 FILE              *fid;

 db_ptr = cuser_obj_ptr->ui_li_type.li_ool.oi_curr_db.cdi_db.dn_obj;
 strcpy(cover_file, db_ptr->odn_name);
 strcat(cover_file, ".cover");
 fid = fopen(cover_file, "r");
 if (fid == NULL)         /*No covering files exist for current database*/
   {
   printf("\nNo covering constructs exists for this database\n");
   }
 else                  /*Covering file found for current database*/
   {
   o_read_cover(fid);      /*Read covering file*/
   fclose(fid);
   }
 }   /*End o_read_cover_file()*/
```

```
/* This procedure reads the openned covering file and displays its contents to
 * the user.
 */
o_read_cover(fid)

 FILE   *fid;

 {
 char   name[21];
 int    cont = TRUE;  /*BOOLEAN*/
 int    in_char;

 while ((in_char = getc(fid)) != EOF)  /*While not End of file*/
   {
   if (in_char != '@@')  /* "@@" seperates records. Looking for end of record.*/
     {
     printf("\n");
     putchar(in_char);
     fscanf(fid, "%s", name);       /*Gets name of covering construct*/
     printf("%-6s : ", name);
     fscanf(fid, "%s", name);       /*Gets covering object*/
     printf("%-4s covers ", name);
     fscanf(fid, "%s", name);       /*Gets covered objects*/
     printf("%-4s and includes the object(s): ", name);
     }
     while(cont)     /*Get rest of objects in scope of covering relationship*/
       {
       fscanf(fid, "%s", name);
       if (strcmp(name, "@@"))
         {
         printf("%s  ", name);
         }
       else
         {
         cont = FALSE;
         fscanf(fid, "\n");
         }
       }  /*End while(cont)*/
     cont = TRUE;

   }  /*End while*/
 printf("\n");
 }       /*End o_read_cover()*/
```

```
/* Provides the user with a list of objects available for creating a covering
 * construct. This is accomplished by pointing to the first object of the
 * database and writing the name of the object to the file "cover" and then
 * moving to the next object in the list. Once the list has been traveresed, the
 * file is closed and then displayed to the screen. The file is deleted upon
 * exiting the procedure.
 */

o_display_classes()
 {
 struct obj_dbid_node  *db_ptr;
 struct ocls_node      *cls_ptr;
 struct o_supcls_node  *supcls_ptr;
 struct o_subcls_node  *subcls_ptr;
 struct oattr_node     *attr_ptr;
 FILE                  *fid;

 /*Point to first object of current database.*/
 db_ptr = cuser_obj_ptr->ui_li_type.li_ool.oi_curr_db.cdi_db.dn_obj;
 fid = fopen("cover","w");
 fprintf(fid,"Database Name : %s\n",db_ptr->odn_name);/*Print name of database*/

 /*Traverse list and write object names to the recieving file.*/
 cls_ptr = db_ptr->odn_first_cls;
 while (cls_ptr)
  {
  fprintf(fid,"CLASS %s\n", cls_ptr->ocn_name);
  cls_ptr = cls_ptr->ocn_next_cls;
  }

 fclose(fid);
 system("more cover");   /*Display list of objects.*/
 system("rm cover");     /*Delete file*/
 }  /*End o_display_classes()*/
```

```
/* This procedure allows the user to define a covering relationship. It recieves
 * as input the name of the covering relation desired, the name of the covering
 * object and the name of the covered object. A search is conducted to verify that
 * the objects exist in the current database. If both objects exist, then a check
 * is made to verify that both objects are not in the same inheritance hierarchy.
 * If they are, the request is rejected. If the two objects are in seperate
 * hierarchies, then the procedure continues and the user is prompted for the
 * scope of the covering relation. Once the scope is defined, a procedure is
 * called to find the objects with in the scope of the covering and writes them
 * to the covering file for that database.
 */
o_get_objects()
  {
  struct obj_dbid_node   *db_ptr;
  struct ocls_node       *cls_ptr;
  struct o_supcls_node   *supcls_ptr;
  struct ocls_node       *from_ptr,
                 *to_ptr,
                 *from_rel_ptr,
                 *to_rel_ptr,
                 *temp1,
                 *cover_ptr,
                 *temp2;
  char           choice;
  char           from_obj[11];
  char           to_obj[11];
  char           cover_name[21];  /*Name of covering construct*/
  int            found = FALSE;   /*Boolean*/
  int            counter = 0;
  int            u_limit = 0;     /*Upper limit for covering*/
  int            l_limit = 0;     /*Lower limit for covering*/
  int            num;

#ifdef EnExFlag
 printf("Enter o_get_objects\n");
 fflush(stdout);
#endif


  /*point to current database*/
  db_ptr = cuser_obj_ptr->ui_li_type.li_ool.oi_curr_db.cdi_db.dn_obj;
  cls_ptr = db_ptr->odn_first_cls;  /*point to first in database.*/
```

```
/*Get desired object names and name covering construct*/
printf("\nEnter the name of the covering construct you wish to create\n");
printf("Use all CAPITAL letters when entering name.\n");
scanf("%s", cover_name);
printf("\nEnter the name of the object you wish to cover FROM.\n");
printf("Use all CAPITAL letters when entering name.\n");
scanf("%s", from_obj);
printf("\nEnter the name of the object you wish to cover TO.\n");
printf("Use all CAPITAL letters when entering name.\n");
scanf("%s", to_obj);

/*Search for objects*/
while (!found && cls_ptr)
 {
 if ((strcmp(cls_ptr->ocn_name, from_obj)) == 0)
  {
  found = TRUE;           /*Object found*/
  from_ptr = cls_ptr;
  }
 else
  {
  cls_ptr = cls_ptr->ocn_next_cls;
  }  /*End if(strcmp()*/
 }  /*End while(not_found)*/

if (!found)
 {
 printf("\n%s", from_obj);
 printf(" does not exist.\n");
 }
else
 {
 found = FALSE;
 cls_ptr = db_ptr->odn_first_cls;
 while (!found && cls_ptr)
  {
  if ((strcmp(cls_ptr->ocn_name, to_obj)) == 0)
   {
   to_ptr = cls_ptr;
   cover_ptr = to_ptr;
   found = TRUE;          /*Second object found*/
   }
  else
```

```
    {
    cls_ptr = cls_ptr->ocn_next_cls;
    }    /*End if(strcmp())*/
   }  /*End while(not_found)*/

if (!found)
  {
  printf("\n%s", to_obj);
  printf(" does not exist.\n");
  }
else
  {

  from_rel_ptr = from_ptr; /*Marks "from object" for relationship reference*/
  to_rel_ptr = to_ptr;    /*Marks "to object" for relationship reference*/

  /*Check to see if objects are in same hierarchy*/
   while(to_ptr)
    {
    temp1 = to_ptr;
    to_ptr = to_ptr->ocn_first_supcls->osn_supcls;
    }    /*End while(to_obj)*/
   while(from_ptr)
    {
    temp2 = from_ptr;
    from_ptr = from_ptr->ocn_first_supcls->osn_supcls;
    }

   if(temp1 != temp2)
    {
    printf("\nA covering relation is being created from ");
    printf("%s", from_obj);
    printf(" to ");
    printf("%s", to_obj);
    printf("\nIs this correct (y or n)? ");
    scanf(" %c", &choice);

    switch (choice)
     {
     case 'y' :  /*User accepts covering construct*/
      {

      /*Determine number of levels above and below to object in the
```

```
 hierarchy which will be included in the covering construct*/
printf("\nHow many levels above %s should the covering construct include? ",
       to_rel_ptr->ocn_name);
scanf(" %d", &u_limit);
printf("\nHow many levels below %s should the covering construct include? ",
       to_rel_ptr->ocn_name);
scanf(" %d", &l_limit);
counter = l_limit + u_limit;

/*Check and adjust counter for lower limit of hierarchy incase the
 *lower limit exceeds the depth of the hierarchy.
 */
to_ptr = to_rel_ptr;
while ((l_limit > 0) && to_ptr->ocn_first_subcls->osn_subcls)
  {
  to_ptr = to_ptr->ocn_first_subcls->osn_subcls;
  l_limit--;
  }
counter = counter - l_limit;

/*Mark top object in covered part of hierarchy.*/
while((u_limit > 0) && cover_ptr->ocn_first_supcls->osn_supcls)
  {
  cover_ptr = cover_ptr->ocn_first_supcls->osn_supcls;
  u_limit--;
  }     /*End while(u_limit);
counter = counter - u_limit; /*Adjusted counter for hierarchy size*/
if (cover_ptr)
  {
  from_rel_ptr->cover_cls = cover_ptr;
  }
else
  {
  from_rel_ptr->cover_cls = temp1;
  cover_ptr = temp1;
  }     /*End if(cover)*/

/*Places objects of covering construct into file*/
o_cover_file(cover_ptr, to_rel_ptr, from_rel_ptr, counter, cover_name);
printf("\nThe covering construct has been completed.\n");
break;
}
```

```
        case 'n' :  /*User does not accept the covering construct*/
         {
         printf("\nThe covering construct has been deleted.\n");
         break;
         }
        default :
         {
         printf("\nInvalid selection.  Start again.\n");
         }
        }              /*End switch(choice)*/
       }
      else
       {
       printf("\n%s", from_obj);
       printf(" and ");
       printf("%s", to_obj);
       printf(" belong to the same inheritance hierarchy.\n");
       printf("A covering construct is not needed for these two objects.\n");
       }   /*End if(temp1 != temp2)*/
     }   /*End if (!found)*/
    } /*End if(not_found)*/

#ifdef EnExFlag
 printf("Exit o_get_objects\n");
 fflush(stdout);
 fflush(stdin);
#endif

   }    /*End o_get_objects()*/
```

```
/* This procedure traverses the hierarchy of the covered object and writes
 * to the covering file the objects which fall within the scope of the
 * covering construct.
 */
o_cover_file(cover_ptr, to_rel_ptr, from_rel_ptr, counter, cover_name)
 struct ocls_node  *to_rel_ptr,
                   *from_rel_ptr,
                   *cover_ptr;
 int                counter;
 char               cover_name[21];

 {
 char               cover_file[FNLength + 3];
 FILE               *fid;
 struct obj_dbid_node *dbase_ptr;
 struct ocls_node     *temp_ptr;
 struct o_subcls_node *follow;
 int                max = 0;
 int                leave = TRUE;
 int                start_up = TRUE;  /*Allows counter to get into initial
                                        loop at a max value.*/


 temp_ptr = cover_ptr;
 dbase_ptr = cuser_obj_ptr->ui_li_type.li_ool.oi_curr_db.cdi_db.dn_obj;
 strcpy(cover_file, dbase_ptr->odn_name);  /*Gets name of database*/
 strcat(cover_file, ".cover");            /*Creats name of covering file*/
 fid = fopen(cover_file, "a");
 fprintf(fid, "%s  ", cover_name);        /*Name of covering construct inserted*/
 fprintf(fid, "%s  ", from_rel_ptr->ocn_name); /*From object inserted*/
 fprintf(fid, "%s  ", to_rel_ptr->ocn_name); /*To object inserted next in line*/
 from_rel_ptr->ocn_marked = TRUE;
 to_rel_ptr->ocn_marked = TRUE;
 max = counter;

/*The following code traverses the covered hierarchy and retrieves the names
 *of the include objects. As each node is traversed, the "ocn_marked" bit is
 *set to signify that the node has already been searched and prevents another
 *retrieval.  The counter keeps track of how many levels have been traversed
 *so as not to exceed the setting for the covering construct.
 */
 if (!temp_ptr->ocn_marked)
   {
```

```
  fprintf(fid, "%s  ", temp_ptr->ocn_name);
  }
follow = temp_ptr->ocn_first_subcls;
temp_ptr->ocn_marked = TRUE;   /*Marks object as visited*/
temp_ptr = temp_ptr->ocn_first_subcls->osn_subcls;

 while (((counter < max) || start_up) && (max != 0))
  {

  start_up = FALSE;
  while(counter !=0 && temp_ptr && leave)
    {
    if (!temp_ptr->ocn_marked)
      {
      fprintf(fid, "%s  ", temp_ptr->ocn_name);
      temp_ptr->ocn_marked = TRUE;
      temp_ptr = temp_ptr->ocn_first_subcls->osn_subcls;
      if (temp_ptr)
        {
        follow = temp_ptr->ocn_first_supcls->osn_supcls->ocn_first_subcls;
        }
      counter--;
      }
    else if (temp_ptr->ocn_first_subcls->osn_subcls)
      {
      temp_ptr = temp_ptr->ocn_first_subcls->osn_subcls;
      if (temp_ptr)
        {
        follow = temp_ptr->ocn_first_supcls->osn_supcls->ocn_first_subcls;
        }
      counter--;
      }
    else
      {
      leave = FALSE;
      }  /*End if(!temp_ptr->ocn_marked)*/
    }  /*End while()*/

  if (follow->osn_next_subcls)
    {
    follow = follow->osn_next_subcls;
    temp_ptr = follow->osn_subcls;
    counter++;
```

```
    start_up = TRUE;
    }
  else
    {
    if(follow->osn_subcls)
      {
      follow = follow->osn_subcls->ocn_first_supcls->osn_supcls->
             ocn_first_supcls->osn_supcls->ocn_first_subcls;
      temp_ptr = follow->osn_subcls;
      }
    counter = counter + 2;
    } /*End if(follow)*/
  leave = TRUE;
  }  /*End while(counter <= max)*/


fprintf(fid, "\n@@\n");
fclose(fid);

/*Clear marked bits in covered hierarchy*/
temp_ptr = dbase_ptr->odn_first_cls;
while (temp_ptr)
  {
  temp_ptr->ocn_marked = FALSE;
  temp_ptr = temp_ptr->ocn_next_cls;
  }

 }  /*End o_cover_file()*/
@
```

B. SOURCE CODE FOR QUERYING VIA THE COVERING CONSTRUCT

```
/*
 * $Header: obj_parser.c,v 0.0 92/11/29 13:53 ksh Exp $
 * $Source: /u/mdbs/rich/CNTRL/TI/LangIF/src/Obj/Kms/obj_parser.c,v $
 * $Log:     obj_parser.c,v $
 * Revision 0.0 92/11/29 13:53 ksh
 * creation
 *
 */

#include <stdio.h>
#include <string.h>
#include <licommdata.h>
#include <ool.h>
#include <ool_lildcl.h>
#include "flags.def"



parse_ool_request()
  {
  char                    *ool_req,
                          *new_ool_req, /*Added to implement covering.
                                         Replaced *ool_req after assignment*/
                           temp_str[InputCols],
                           cover_temp[InputCols],
                           condition_str[InputCols],
                           *check_cover();
  int                      i,
                           in_char,
                          covered,
                          qualified_attr = FALSE;
  struct obj_kms_info     *kms_ptr,
                          *obj_kms_info_alloc();
  struct ocls_node        *requested_class,
                          *find_class();
  FILE                    *temp_file;


#ifdef EnExFlag
  printf("Enter parse_ool_request\n");
#endif
```

```
/* allocate and initialize first obj_kms_info struct */
kms_ptr = obj_kms_info_alloc();
ool_info_ptr->oi_kms_data.ki_o_kms = kms_ptr;
kms_ptr->oki_next = NULL;
strcpy(kms_ptr->oki_tgt_cls1.tci_name, BLANK);
strcpy(kms_ptr->oki_tgt_cls1.common_attr, "OBJECTID"); /* default common attr */
kms_ptr->oki_tgt_cls1.tci_tgt_attrs = NULL;
kms_ptr->oki_tgt_cls1.tci_conditions = NULL;
strcpy(kms_ptr->oki_tgt_cls2.tci_name, BLANK);
strcpy(kms_ptr->oki_tgt_cls2.common_attr, "OBJECTID"); /* default common attr */
kms_ptr->oki_tgt_cls2.tci_tgt_attrs = NULL;
kms_ptr->oki_tgt_cls2.tci_conditions = NULL;

/* pointer to process actual ool request */
ool_req = o_curr_req_ptr->ri_obj_req->ori_req;

in_char = strncmp(ool_req, "(", 1);  /* Checks to see if the request
                              involves a covering cons  uct*/
strcpy(cover_temp, ool_req);
new_ool_req = cover_temp;
if (in_char == 0)
  {

  /* Sends request to check_cover() to verify if it is a valid
   * covering request.*/
  new_ool_req = check_cover(new_ool_req);
  }
else
 {
 new_ool_req = ool_req;
 }  /*End if(char == 0)*/


strcpy(temp_str, BLANK);       /* temp string for tokens */

  i = 0;
  while (*new_ool_req != ' ')
   {
    if (*new_ool_req == '.')
      {
      temp_str[i] = '\0';
      strcpy(kms_ptr->oki_tgt_cls1.tci_name, temp_str);
      requested_class = find_class(ool_info_ptr->oi_curr_db.
```

```
                    cdi_db.dn_obj->odn_first_cls, temp_str);

        strcpy(temp_str, BLANK);
        i = 0;
        }
      else
        {
        temp_str[i] = *new_ool_req;
        ++i;
        }
      ++new_ool_req;
    }

  temp_str[i] ='\0';
  i = 0;
  if (!strcmp(temp_str, "RETRIEVE"))
    ool_info_ptr->oi_operation = ExecRetReq;
  else if (!strcmp(temp_str, "INSERT"))
    ool_info_ptr->oi_operation = ExecInsReq;
  else
    /* other operations are not implemented yet.
    if (!strcmp(temp_str, "DELETE"))
      ool_info_ptr->oi_operation = ExecDelReq;
    else if (!strcmp(temp_str, "UPDATE"))
      ool_info_ptr->oi_operation = ExecUpdReq; */
   {
    ool_info_ptr->oi_operation = ExecNoReq;
    return ;
   }
  strcpy(temp_str, BLANK);

  for ( ; (*new_ool_req == ' '); ++new_ool_req)
     ;                          /* eat the blanks in between */

  switch (ool_info_ptr->oi_operation)
  {
   case ExecRetReq:
       while (*new_ool_req != ' ' && *new_ool_req != ',')
        {
         if (*new_ool_req == '.')
           {
            temp_str[i] = '\0';
            strcpy(kms_ptr->oki_tgt_cls1.common_attr, temp_str);
```

```
      find_component_class(kms_ptr, temp_str, requested_class);
      strcpy(temp_str, BLANK);
      qualified_attr = TRUE;
      i = 0;
     }
    else
     {
      temp_str[i] = *new_ool_req;
      ++i;
     }
    ++new_ool_req;
  }
temp_str[i] = '\0';

while (strcmp(temp_str, "IF"))
 {
  insert_target_attr(kms_ptr, temp_str, requested_class,
                                    qualified_attr);
  strcpy(temp_str, BLANK);
  qualified_attr = FALSE;
  for ( ; (*new_ool_req == ' ' || *new_ool_req == ','); ++new_ool_req)
     ;                   /* eat blanks or commas */

  if (*new_ool_req == '\0')  /* end of request is reached */
  break;

  i = 0;
  while (*new_ool_req != ' ' && *new_ool_req != ',')
   {
    if (*new_ool_req == '.')
     {
      temp_str[i] = '\0';
      strcpy(kms_ptr->oki_tgt_cls1.common_attr, temp_str);
      find_component_class(kms_ptr, temp_str, requested_class);
      strcpy(temp_str, BLANK);
      qualified_attr = TRUE;
      i = 0;
     }
    else
     {
      temp_str[i] = *new_ool_req;
      ++i;
     }
```

```
      ++new_ool_req;
     }

    temp_str[i] = '\0';
   } /* end while "IF" && BLANK */


 if (!strcmp(temp_str, "IF"))  /* condition part begins */
  {
   strcpy(temp_str, BLANK);
   for ( ; (*new_ool_req == ' '); ++new_ool_req)
      ;                    /* eat blanks */
   i = 0;
   while (*new_ool_req != '\0')  /* while not end of request */
    {
     if (*new_ool_req == '.')
      {
       temp_str[i] = '\0';
       strcpy(kms_ptr->oki_tgt_cls1.common_attr, temp_str);
       find_component_class(kms_ptr, temp_str, requested_class);
       strcpy(temp_str, BLANK);
       qualified_attr = TRUE;
       i = 0;
      }
     else if (*new_ool_req == ' ' && conjunct_exist(new_ool_req))
      {
       temp_str[i] = '\0';
       insert_condition(kms_ptr, temp_str,
                  requested_class, qualified_attr);
       ++new_ool_req; ++new_ool_req; ++new_ool_req;
       i =0;
       strcpy(temp_str, BLANK);
       qualified_attr = FALSE;
      }
     else
      {
       temp_str[i] = *new_ool_req;
       ++i;
      }
     ++new_ool_req;
    } /* end while */

   /* one more insert for the last condition */
```

```
      temp_str[i] = '\0';
      insert_condition(kms_ptr, temp_str,
                        requested_class, qualified_attr);
    } /* end if (!strcmp(temp_str, "IF")) */

  break;

case ExecInsReq:
  /* create .insert_file by parsing INSERT request */
  temp_file = fopen(".insert_file", "w");
  fprintf(temp_file, "%s\n@@\n%s\n",
              ool_info_ptr->oi_curr_db.cdi_dbname,
              kms_ptr->oki_tgt_cls1.tci_name);

  while (*new_ool_req != '\0')
   {
     if (*new_ool_req != ' ')
      if (*new_ool_req == ',')
       {
         temp_str[i] = '\0';
         fprintf(temp_file, "%s\t", temp_str);
         i = 0;
         strcpy(temp_str, BLANK);
       }
      else
       {
         temp_str[i] = *new_ool_req;
         ++i;
       }

     ++new_ool_req;

   } /* end while */

  temp_str[i] = '\0';
  fprintf(temp_file, "%s\n$\n", temp_str);
  fclose(temp_file);

  break;

case ExecDelReq:
  break;
```

```
        case ExecUpdReq:
            break;

        default:
            break;
    } /* end switch */

#ifdef EnExFlag
    printf("Exit parse_ool_request\n");
/*  test_parser(kms_ptr); */
#endif

} /* end parse_ool_request */
```

```
/* This procedure processes a request which involves a covering construct.
 * It first checks to see if a covering file exists for this database. If
 * no file exists, the procedure is terminated. If a file exists, a check
 * is made to see if there is a covering construct which has the same name
 * as the request. It then verifies that the covering object is the same
 * and if the requested object falls within the scope of the covering
 * construct. The source code for the creation of covering constructs is
 * located in the file o_cover.c, located in the Lil directory.
 */
char *check_cover(req_ptr)

  char *req_ptr;

  {
  char             temp[InputCols],
                  cover_name[InputCols],
                  from_obj[InputCols],
                  to_obj[InputCols],
                  cover_file[FNLength + 3];
  char             *str_chk,
                  *string1,
                  *string2,
                  *string3,
                  *string4,
                  *string5 = NULL;
  struct obj_dbid_node  *db_ptr;
  FILE               *fid;


    db_ptr = cuser_obj_ptr->ui_li_type.li_ool.oi_curr_db.cdi_db.dn_obj;
    strcpy(cover_file, db_ptr->odn_name);
    strcat(cover_file, ".cover");  /*Gets name of covering file for
                            current database*/
    str_chk = req_ptr;
    fid = fopen (cover_file, "r");
    if (fid == NULL)         /*No file exists for current database*/
      {
      printf("\nNo covering file exists for this database.\n");
      printf("Query refused.\n\n");
     /* fclose(fid);*/
      return str_chk;
      }
    else
```

```
{

/*Get covering construct name and name of covering object*/
string1 = strtok(str_chk, "(.)"); /*Get covering object name*/
string2 = strtok(NULL, "(.)");    /*Get covering construct name*/
string3 = strtok(NULL, "()");     /*Get rest of querry*/
++string3;
strcpy(temp, string3);
string4 = temp;
string5 = strtok(string4, ".");   /*Get requested object name*/

while (fscanf(fid, "%s", cover_name) != EOF)
  {

  /*If covering construct name is found*/
  if (!strcmp(cover_name, string2))
    {
    fscanf(fid, "%s", from_obj);

    /*If covering object is found*/
    if (!strcmp(from_obj, string1))
      {
      fscanf(fid, "%s", to_obj);

      /*Scan file until the end of covering construct is reached*/
      while (strcmp(to_obj, "@@"))
        {

        /*If requested object is within the scope of
          the covering construct*/
        if (!strcmp(to_obj, string5))
          {
          fclose(fid);
          return string3; /*Return request to be retrieved*/
          }
        else
          {
          fscanf(fid, "%s", to_obj);
          } /*End strcmp(string5)*/
        }  /*End while(fscanf "@@")*/

      } /*End if(strcmp string1)*/
    } /*End if(strcmp string2)*/
```

```
      fscanf(fid, "\n");
      } /*End while(fscanf != EOF)*/
    } /*End if(fid == NULL*/
  fclose(fid);

  /*No covering construct was found matching the query.*/
  printf("\nNo covering construct exists for this query.\n");
  printf("\nQuery refused.\n\n");
  return str_chk;
 } /*End check_cover()*/
@
```

C. SOURCE CODE FOR THE OBJECTID GENERATOR

```
/
********************************************************************************
****/
/* This function gets the time of day (GMT) from the system, and parses it */
/* into an 8 digit number, which is then used as an objectID. This function*/
/* is called by mss_load() and returns a character string.          */
/
********************************************************************************
****/



char   *get_objectid()
  {
  time_t mytime = 0;
  long int years;
  long int remainder;
  long int yearseconds;
  long int julianday;
  long int dayremainder;
  long int dayseconds;
  long int hourseconds;
  long int minuteseconds;
  long int objectid;
  char today;
  int microseconds;
  int hours;
  int minuteremainder;
  int minutes;
  int seconds;
  int i = 0;
  int j = 0;
  long int temp;
  int  int_array[9];
  char temp_array[9];

  struct timeval  tp;
  struct timeval tzp;

  mytime = gettimeofday(&tp, &tzp);

  /*Total secs mod # secs in one year = total years since 1970*/
```

```
years = tp.tv_sec / 31536000;

/*computes the elapsed years*/
yearseconds  = years * 31536000;         /* 23 years * # sec/year */
remainder = tp.tv_sec - yearseconds;
julianday = remainder / 86400;

/*computes the elapsed days (julian day) */
dayseconds = julianday * 86400;  /* julianday * 86400 sec/year*/
dayremainder = tp.tv_sec - (yearseconds + dayseconds);
hours = dayremainder / 3600;             /* leftover day in seconds */

/*computes the elapsed hours*/
hourseconds = hours * 3600;
minuteremainder = tp.tv_sec - (yearseconds + dayseconds + hourseconds);
minutes= minuteremainder / 60;
minuteseconds = minutes * 60;

/*computes the elapsed seconds*/
seconds = tp.tv_sec - (yearseconds + dayseconds +
                hourseconds + minuteseconds);


/*computes the elapsed milliseconds*/
microseconds = tp.tv_usec / 10000;

/*Set variable temp to format needed for objectid*/
temp = hours * 1000000;
temp = temp + (minutes * 10000);
temp = temp + (seconds * 100);
temp = temp + microseconds;


/*Read objectid into integer array*/
int_array[0] = hours / 10;
int_array[1] = hours % 10;
int_array[2] = minutes / 10;
int_array[3] = minutes % 10;
int_array[4] = seconds / 10;
int_array[5] = seconds % 10;
int_array[6] = microseconds / 10;
int_array[7] = microseconds % 10;
```

```
i = 0;

/*Convert integer string into character string*/
while (i < 8)
  {
  switch(int_array[i])
    {
    case (1) : { temp_array[i] = '1'; break;}
    case (2) : { temp_array[i] = '2'; break;}
    case (3) : { temp_array[i] = '3'; break;}
    case (4) : { temp_array[i] = '4'; break;}
    case (5) : { temp_array[i] = '5'; break;}
    case (6) : { temp_array[i] = '6'; break;}
    case (7) : { temp_array[i] = '7'; break;}
    case (8) : { temp_array[i] = '8'; break;}
    case (9) : { temp_array[i] = '9'; break;}
    default  : { temp_array[i] = '0'; break;}
    }
  i++;
  }
for (j = 0; j < 10000; j++); /*Delays program to prevent duplicate object ids*/

return temp_array;
} /* end get_objectid function */
```

APPENDIX C. TUTORIAL FOR THE OBJECT-ORIENTED INTERFACE ON MDBS

The purpose of this appendix is to provide the user with a brief tutorial of the Object-Oriented Interface on MDBS. Snapshots of the system output, along with a brief description of what is occurring will be provided to the user. Entries made by the user are depicted in bold print.

A. ACCESSING THE OBJECT-ORIENTED INTERFACE

To access the Object-Oriented Interface, the user must first log on to DB3, using the CS4322 account. Once the user has entered the account, he or she must go to the directory: **mdbs/rich/run.** To start mdbs, the user must verify the following: (1) There are no mdbs processes running. This is accomplished by typing "ps -ax" at the command line. If any mdbs processes are running, type the command "stop.cmd" and the processes will be terminated. (2) Verify there is no data on the disk from a previous session. To verify the absence of data, the user should type "pry" at the command prompt. This should display a line of zeros. If text appears, the user should press "CNTRL-c" to exit and then enter "zero" at the command prompt. After the zeroing process is complete, the user is required to enter "run" at the command prompt. This will start the Multilingual/Multimodel system. After the system is loaded, the screen bellow will appear.

```
Select an operation:

    (a) - Execute the attribute-based/ABDL interface
    (r) - Execute the relational/SQL interface
    (h) - Execute the hierarchical/DL/I interface
    (n) - Execute the network/CODASYL interface
    (f) - Execute the functional/DAPLEX interface
    (o) - Execute the object-oriented/OOL interface
    (x) - Exit to the operating system

Select-> o
```

The user should verify that twelve processes (six controllers and six backends)are on-line by running the "ps -ax" command in a separate window. If there are not twelve processes running, select "x" from the menu. When the system has shut down, run the "stop.cmd" again and restart the system. Repeat this procedure until twelve processes are running. Note: The user does not need to zero the database because no data has been loaded to the disk. After the system is successfully running with twelve processes, select "o" from the menu to enter the object-oriented interface.

B. SELECTING A DATABASE

After selecting the Object-Oriented Interface, the user will be presented with the following menu for processing a database.

```
Enter type of operation desired
      (l) - load new database
      (p) - process existing database
      (x) - return to the MLDS/MDBS system menu

Action --- >
```

At this point, the user has two options. Selecting "l" allows the user to load a new database. This is done whenever the interface is first used or when additional databases are required. M^2DBMS allows the user to run multiple databases simultaneously, although access is restricted to one database at a time. Selecting "p" allows the user to process a database which has been previously loaded during the current session. Selecting "x" returns the user to the previous menu. Since this is the beginning of a new session, we will chose "l" and load a new database. This selection produces the menu display below:

```
Enter name of database ---->FAMILY
```

The user is prompted for the name of the database which is to be loaded. If the user is loading the database from disk, the name entered must match the name of the database in the file. This is also necessary for referencing template and descriptor files which will be discussed later in this tutorial.

After the user enters the name of the database, the menu changes as shown below. This menu request the mode of input for the schema. The user may select input from a file or may choose to enter the schema manually from the screen. In this tutorial we will select "f" and read the input from a text file.

```
Enter mode of input desired
    (f) - read in a group of creates from a file
    (t) - read in creates from the terminal
    (x) - return to the main menu

Action --- >
```

Selecting "f" will cause the system to prompt the user for the name of the file. For loading the schema, the convention used is <filename>ooldb. For this tutorial we will use the file FAMILYooldb. Below is an example of the screen display for this procedure. Figure 23 shows the text file, FAMILYooldb, for the FAMILY database schema.

```
What is the name of the CREATE/QUERY file ---->FAMILYooldb
```

The user is then asked if he or she desires to use the existing descriptor file. The screen display is shown below. The contents of the descriptor file for the FAMILY database is shown in Figure 24. The earlier reference to the importance of naming the database

```
CLASS GEORGE
  SUBCLASS  MIKE
  SUBCLASS  PAUL
    OBJECTID  INTEGER
    FIRSTN    CHAR 10
    LASTN     CHAR 10
    SALARY    INTEGER
@
CLASS BERTHA
  SUBCLASS  SUE
  SUBCLASS  JOE
    OBJECTID  INTEGER
    FIRSTN    CHAR 10
    LASTN     CHAR 10
    SALARY    INTEGER
@
CLASS MIKE
  SUPCLASS  GEORGE
@
CLASS PAUL
  SUPCLASS  GEORGE
  SUBCLASS  PAULLA
@
CLASS SUE
  SUPCLASS  BERTHA
@
CLASS JOE
  SUPCLASS  BERTHA
  SUBCLASS  TODD
@
CLASS PAULLA
  SUPCLASS  PAUL
  SUBCLASS  ANDY
  SUBCLASS  SAMANTHA
@
CLASS TODD
  SUPCLASS  JOE
@
CLASS ANDY
  SUPCLASS  PAULLA
@
CLASS SAMANTHA
  SUPCLASS  PAULLA
$
```

Figure 23: The Schema Text File for the Family Database: FAMILYooldb

correctly comes into play here. If the user is not consistent in the naming of his or her database, the established descriptor file will not be used.

```
Would you like to use the existing descriptor file, FAMILY.d,
for indexing information?(y or n)

Action --- >y
```

If the user replies to the above query with "n", the database schema will not load and the previous menu will be re-displayed. If the user responds with "y", the schema will be loaded using the current descriptor file. NOTE: If the schema file has been modified, the user must delete the ".d" and the ".t" files associated with that database. These files are found in the UserFiles subdirectory of the MDBS directory. After selecting "y", the menu shown below will be re-displayed, giving the user the option of loading an additional database or processing a previously loaded database. In our example we will process the existing database FAMILY.

```
Enter type of operation desired
    (l) - load new database
    (p) - process existing database
    (x) - return to the MLDS/MDBS system menu

Action --- > p

Enter name of database ---->FAMILY
```

C. LOADING DATA INTO THE DATABASE

After selecting a database to process, the user is ready to proceed to the next menu. The menu shown below provides the user with all the functions needed to manipulate the data.We will review each item in turn.

DESCRIPTOR FILE: FAMILY.d

```
FAMILY
TEMP b s
! George
! Bertha
! Mike
! Paul
! Sue
! Joe
! Paulla
! Todd
! Andy
! Samantha
@
$
```

TEMPLATE FILE: FAMILY.t

```
FAMILY
10
5
George
TEMP s
OBJECTID i
FIRSTN s
LASTN s
SALARY i
5
Bertha
TEMP s
OBJECTID i
FIRSTN s
LASTN s
SALARY i
2
Mike
TEMP s
OBJECTID i
2
Paul
TEMP s
OBJECTID i
2
Sue
TEMP s
OBJECTID i
2
Joe
TEMP s
OBJECTID i
2
Paulla
TEMP s
OBJECTID i
2
Todd
TEMP s
OBJECTID i
2
Andy
TEMP s
OBJECTID i
2
Samantha
TEMP s
OBJECTID i
```

Figure 24: Descriptor and Template Files for the FAMILY Database

```
Enter your choice
    (d) - display schema
    (c) - create covering relation between two objects
    (m) - mass load from a data file
    (f) - read in a group of queries from a file
    (t) - read in queries from the terminal
    (x) - return to previous menu

Action --- >
```

Before we manipulate records, we must first load the raw data into the database. This is done by selecting "m" from the menu above. The system will then ask the user for the name of the record file. As with the schema, the data file maintains the same naming conventions. The name of the data file is <filename.r>. In our case we will use the file FAMILY.r. The screen display for this sequence is shown below. Figure 25 shows the contents of the FAMILY.r record file. Figure 26 shows the screen display as records are loaded into the database. Note that the object-ids in the record file do not match the object-ids in the screen display. This is because the Object-Oriented interface creates its own object-ids which are generated from the system clock. The object-ids in the record file are used as a place holder to ensure the proper matching of attributes in the template file and the record file.

```
Enter name of record file ---->FAMILY.r
```

Once the mass load procedure is complete, the menu below will appear. We have seen this menu before. The options for this menu remain the same. This gives the user the opportunity to switch to another database in order to mass load or manipulate records. We will select option "p" and continue processing the FAMILY database.

```
FAMILY
@
GEORGE
1 George   Jones   50000
@
BERTHA
2 Bertha   Smith   75000
@
MIKE
3 Mike     Jones   32000
@
PAUL
4 Paul     Jones   45000
@
SUE
5 Sue      Smith   30000
@
JOE
6 Joe      Smith   18000
@
PAULLA
7 Paulla   Jones   100000
@
TODD
8 Todd     Smith   200
@
ANDY
10 Andy    Jones   0
@
SAMANTHA
11 Samantha Jones  0
$
```

NOTE: FAMILY indicates the name of the database which is associated with this record file. The "@" indicates an end of record. The "$" indicates end of file. The capitalized names are the names of each object class. The attribute values are listed on the second line of each record. For this database, they are from left to right: object-id, first name, last name, salary.

Figure 25: Record File for the FAMILY Database: FAMILY.r

[INSERT (<TEMP, George>, <OBJECTID, 03065375>, <FIRSTN, George>, <LASTN, Jones>, <SALARY, 50000>)]
[INSERT (<TEMP, Bertha>, <OBJECTID, 03065379>, <FIRSTN, Bertha>, <LASTN, Smith>, <SALARY, 75000>)]
[INSERT (<TEMP, George>, <OBJECTID, 03065380>, <FIRSTN, Mike>, <LASTN, Jones>, <SALARY, 32000>)]
[INSERT (<TEMP, Mike>, <OBJECTID, 03065380>)]
[INSERT (<TEMP, George>, <OBJECTID, 03065382>, <FIRSTN, Paul>, <LASTN, Jones>, <SALARY, 45000>)]
[INSERT (<TEMP, Paul>, <OBJECTID, 03065382>)]
[INSERT (<TEMP, Bertha>, <OBJECTID, 03065385>, <FIRSTN, Sue>, <LASTN, Smith>, <SALARY, 30000>)]
[INSERT (<TEMP, Sue>, <OBJECTID, 03065385>)]
[INSERT (<TEMP, Bertha>, <OBJECTID, 03065387>, <FIRSTN, Joe>, <LASTN, Smith>, <SALARY, 18000>)]
[INSERT (<TEMP, Joe>, <OBJECTID, 03065387>)]
[INSERT (<TEMP, George>, <OBJECTID, 03065390>, <FIRSTN, Paulla>, <LASTN, Jones>, <SALARY, 100000>)]
[INSERT (<TEMP, Paul>, <OBJECTID, 03065390>)]
[INSERT (<TEMP, Paulla>, <OBJECTID, 03065390>)]
[INSERT (<TEMP, Bertha>, <OBJECTID, 03065394>, <FIRSTN, Todd>, <LASTN, Smith>, <SALARY, 200>)]
[INSERT (<TEMP, Joe>, <OBJECTID, 03065394>)]
[INSERT (<TEMP, Todd>, <OBJECTID, 03065394>)]
[INSERT (<TEMP, George>, <OBJECTID, 03065397>, <FIRSTN, Andy>, <LASTN, Jones>, <SALARY, 0>)]
[INSERT (<TEMP, Paul>, <OBJECTID, 03065397>)]
[INSERT (<TEMP, Paulla>, <OBJECTID, 03065397>)]
[INSERT (<TEMP, Andy>, <OBJECTID, 03065397>)]
[INSERT (<TEMP, George>, <OBJECTID, 03065399>, <FIRSTN, Samantha>, <LASTN, Jones>, <SALARY, 0>)]
[INSERT (<TEMP, Paul>, <OBJECTID, 03065399>)]
[INSERT (<TEMP, Paulla>, <OBJECTID, 03065399>)]
[INSERT (<TEMP, Samantha>, <OBJECTID, 03065399>)]

Figure 26: Screen Display of the File FAMILY.r Being Loaded Into the Database

```
Enter type of operation desired
      (l) - load new database
      (p) - process existing database
      (x) - return to the MLDS/MDBS system menu

Action --- > p

Enter name of database ---->FAMILY
```

D. CREATING A COVERING RELATION

The Object-Oriented database now has the ability to access data via a covering relation. In this section we will create covering constructs between object classes, allowing the user access to the data in one object-class through another object class.

Having chosen to continue processing the FAMILY database, the record manipulation menu is redisplayed. As seen below, we will select option "c" in order to create one or more covering relationships.

```
Enter your choice
      (d) - display schema
      (c) - create covering relation between two objects
      (m) - mass load from a data file
      (f) - read in a group of queries from a file
      (t) - read in queries from the terminal
      (x) - return to previous menu

Action --- >
```

Having selected the covering option, the screen will clear and the covering menu will be displayed as shown below. The covering menu gives the user several options, including: displaying existing covering constructs, displaying classes available within the current

database for use in creating a covering construct, and establishing a covering construct between two objects. We will review each option in turn.

```
Enter type of operation desired
        (d) - Display Existing Covering Constructs
        (c) - Display Available Classes
        (e) - Establish a Covering Construct Between Two Objects
        (x) - Return to previous menu

Action --- >
```

The first option in the covering menu allows the user to display any existing covering constructs. When displaying covering constructs, the system reads the covering file associated with each database. If the database has been newly created, no covering file for that database should exist. Covering files are deleted when the users exits from the Multimodel/Multilingual system. In other words, the covering file exists only during the duration of the current session. If we select "d" when no covering file exists, we will see the reply shown below. After we create a covering construct, we will return to this menu selection and show a display of the existing covering constructs.

```
No covering constructs exists for this database

Enter type of operation desired
        (d) - Display Existing Covering Constructs
        (c) - Display Available Classes
        (e) - Establish a Covering Construct Between Two Objects
        (x) - Return to previous menu

Action --- >
```

Before creating a covering relation, it is sometimes useful to see which objects are available in the database. Selecting "c" displays all the objects available in the current database. Since spelling the object's name is important, the user should display the

available object classes prior to creating a covering relationship. The screen display is shown below.

```
Database Name : FAMILY
CLASS GEORGE
CLASS BERTHA
CLASS MIKE
CLASS PAUL
CLASS SUE
CLASS JOE
CLASS PAULLA
CLASS TODD
CLASS ANDY
CLASS SAMANTHA

Enter type of operation desired
      (d) - Display Existing Covering Constructs
      (c) - Display Available Classes
      (e) - Establish a Covering Construct Between Two Objects
      (x) - Return to previous menu

Action --- >
```

The list of available objects is listed in the order they appear in the link list in memory. These objects are not grouped by class, so the possibility exists that two objects from the same class may be chosen. This is not a problem because the procedure that creates the covering construct will not allow two objects in the same object hierarchy to be joined in a covering relationship. If this happens, the system will notify the user that this is the case, and the user can try again.

Having displayed the list of objects available in the database, we can now create a covering relation. To refresh the reader, covering is the mapping of an object in one hierarchy to an object class in another hierarchy. The covered object class can consist of one object, several related objects, or an entire object hierarchy. When creating a covering

relation between two objects in the Object-Oriented Interface, the user can define the scope of the relation by setting the number of objects above and below the covered object within its hierarchy.

To create a covering relation, the user selects "e" from the menu shown above. The user is then prompted to enter the name of the covering relation. This naming convention allows the user to define more than one covering construct for each object or define the same type of covering construct for different objects. After naming the relation, the user is prompted for the name of the covering object and then the name of the object to be covered. Having received this information, the system finds the target objects. If either of the objects is not found, the system informs the user of the situation and returns to the covering menu. If the system finds both objects, it performs a check to see if the two objects are in the same object-class hierarchy. If both objects are in the same hierarchy, the system informs the user that this is the case and returns the user to the covering menu. This is because covering only applies between two objects residing in different hierarchies.

If the above conditions are met, the system will prompt the user for verification. If the user answers "yes", the covering process continues. If "no" is selected, the covering process is terminated and the covering menu returns. Having made a selection, the user is prompted to define the scope of the covering relationship. The user is prompted for the number of levels above and below the covered object. After the system receives this input, it writes the name of the relation, the covering object's name, and the covered object's name to the covering file associated with the current database. Once these names are written to the covering file, the system searches the covering hierarchy for objects within the declared scope of the covering relation. As each covered object is found, the system writes the object's name to the covering file. Upon completion of the search, the covering file is closed and the user is returned to the covering menu. The screen display of the sequence of events mentioned above is shown below.

Enter the name of the covering construct you wish to create
Use all CAPITAL letters when entering name.
IN-LAW

Enter the name of the object you wish to cover FROM.
Use all CAPITAL letters when entering name.
TODD

Enter the name of the object you wish to cover TO.
Use all CAPITAL letters when entering name.
PAULLA

A covering relation is being created from TODD to PAULLA
Is this correct (y or n)? **y**

How many levels above PAULLA should the covering construct include? **1**

How many levels below PAULLA should the covering construct include? **2**

The covering construct has been completed.

Enter type of operation desired
(d) - Display Existing Covering Constructs
(c) - Display Available Classes
(e) - Establish a Covering Construct Between Two Objects
(x) - Return to previous menu

Action --- >

Once a covering construct is created, we can select "d" from the covering menu and display the existing covering constructs. The example below shows a covering file with two covering constructs. Note: the BUSINESS covering construct contains no additional objects. This is because the scope of the covering relation was defined to be zero above and zero below the covered object. Figure 14, in Chapter IV of this thesis, shows the contents of the covering file, FAMILY.cover. The covering file for each database is maintained in the mdbs/rich/run directory during the duration of the sessions.

```
IN-LAW : TODD covers PAULLA and includes the object(s): PAUL ANDY
SAMANTHA
BUSINESS : SAMANTHA covers JOE and includes the object(s):

The covering construct has been completed.
Enter type of operation desired
    (d) - Display Existing Covering Constructs
    (c) - Display Available Classes
    (e) - Establish a Covering Construct Between Two Objects
    (x) - Return to previous menu

Action --- >
```

E. PERFORMING QUERIES ON THE OBJECT-ORIENTED DATABASE

Having completed the creation of a covering construct, we can exit the covering menu by selecting "x" and return to the database operation menu shown below.

```
Enter your choice
    (d) - display schema
    (c) - create covering relation between two objects
    (m) - mass load from a data file
    (f) - read in a group of queries from a file
    (t) - read in queries from the terminal
    (x) - return to previous menu

Action --- >
```

The interface allows the user to perform queries on data in two ways.The user can load a list of queries from a text file by selecting "f", or enter queries manually from the terminal by selecting "t". Both methods use the same format for listing queries. In this tutorial we will load a query file by selecting "f". The user is prompted for the file name. The file name for the queries can be any name. The query file for this example is FAMILYoolreq. The screen display for this sequence of events is shown below. The

contents of FAMILYoolreq is shown in Figure 27. The screen display of the query list is shown in Figure 28.

```
Action --- >f

What is the name of the CREATE/QUERY file ---->FAMILYoolreq
```

Having loaded a list of queries into the system, the user can select a query for retrieval. We will select query number "1". The screen display for this query is shown below. It is important to note that a query on an object not only retrieves the data for that object but all its subclasses in the inheritance hierarchy. If we perform query "11" in Figure 28, only the data for SAMANTHA will be retrieve because this object has no subclasses.

```
Action --- >1

SALARY     |LASTN     |FIRSTN     |OBJECTID     |
----------------------------------------------------------
50000      |Jones     |George     |03065375     |
32000      |Jones     |Mike       |03065380     |
45000      |Jones     |Paul       |03065382     |
100000     |Jones     !Paulla     |03065390     |
0          |Jones     |Andy       |03065397     |
0          |Jones     |Samantha   |03065399     |

Pick the number or letter of the action desired
     (num) - execute one of the preceding queries
     (d)  - redisplay the file of queries
     (x)  - return to the previous menu

Action --- >
```

```
/File Name : FAMILYoolreq

george.retrieve objectid, firstn, lastn, salary
@
bertha.retrieve objectid, firstn, lastn, salary
@
mike.retrieve objectid, firstn, lastn
@
paul.retrieve objectid, firstn, lastn
@
sue.retrieve objectid, firstn, lastn
@
joe.retrieve objectid, firstn, lastn
@
paulla.retrieve objectid, firstn, lastn
@
todd.retrieve objectid, firstn, lastn
@
(paul.in-law) sue.retrieve objectid, firstn, lastn
@
andy.retrieve objectid, firstn, lastn
@
samantha.retrieve objectid, firstn, lastn
@
(todd.in-law) paulla.retrieve firstn, lastn
@
(samantha.business) joe.retrieve objectid, firstn, lastn if firstn = 'Joe'
@
(todd.in-law) andy.retrieve objectid, firstn, lastn
$
```

Figure 27: Request File for the FAMILY Database: FAMILYoolreq

1 george.retrieve objectid, firstn, lastn, salary

2 bertha.retrieve objectid, firstn, lastn, salary

3 mike.retrieve objectid, firstn, lastn

4 paul.retrieve objectid, firstn, lastn

5 sue.retrieve objectid, firstn, lastn

6 joe.retrieve objectid, firstn, lastn

7 paulla.retrieve objectid, firstn, lastn

8 todd.retrieve objectid, firstn, lastn

9 (paul.in-law) sue.retrieve objectid, firstn, lastn

10 andy.retrieve objectid, firstn, lastn

11 samantha.retrieve objectid, firstn, lastn

12 (todd.in-law) paulla.retrieve firstn, lastn

13 (samantha.business) joe.retrieve objectid, firstn, lastn if firstn = 'Joe'

14 (todd.in-law) andy.retrieve objectid, firstn, lastn

Pick the number or letter of the action desired
(num) - execute one of the preceding queries
(d) - redisplay the file of queries
(x) - return to the previous menu

Action --- > 9

Figure 28: Query Menu After Loading Request File FAMILYoolreq

```
Action --- > 11


LASTN       |FIRSTN      |OBJECTID    |
------------------------------------------
Jones       |Samantha    |03065399    |

Pick the number or letter of the action desired
      (num) - execute one of the preceding queries
      (d)   - redisplay the file of queries
      (x)   - return to the previous menu

Action --- >
```

To retrieve only the data for the specified item, the query must contain a condition specifying an attribute value which is unique. In our case, we can select number "13" from the menu which retrieves the object JOE where the attribute "firstn" equals "Joe". This will only retrieve the data for the object JOE as shown below. If this condition was not present, the query would also retrieve the data for the object TODD, since it is a subclass of JOE.

```
Action --- > 13


LASTN       |FIRSTN      |OBJECTID    |
------------------------------------------
Smith       |Joe         |03065387    |

Pick the number or letter of the action desired
      (num) - execute one of the preceding queries
      (d)   - redisplay the file of queries
      (x)   - return to the previous menu

Action --- >
```

We can access data via a covering relation in the same manner. As mentioned earlier, a covering file is maintained for each database. When selecting a query which uses a covering construct, the system verifies that the covering construct used in the query exists

in the covering file. If the construct does exist, the query proceeds, if the construct does not exist, the query is refused. If we select number "9" from the query menu as shown below, the system will refuse the query because no covering construct exist for that query.

```
No covering construct exists for this query.

Query refused.

Pick the number or letter of the action desired
      (num) - execute one of the preceding queries
      (d)   - redisplay the file of queries
      (x)   - return to the previous menu

Action --- >
```

However, if we select query number "14", it will be successful because the covering file contains the covering construct found within the query.

```
Action --- > 14


LASTN        |FIRSTN       |OBJECTID     |
------------------------------------------
Jones        |Andy         |03065397     |

Pick the number or letter of the action desired
      (num) - execute one of the preceding queries
      (d)   - redisplay the file of queries
      (x)   - return to the previous menu

Action --- >
```

There are times when the queries listed in the query file are insufficient to accomplish the job at hand. Instead of having to write a new file in order to process a few queries, the object-oriented interface allows the user to enter queries at the terminal for record

processing. In order to enter queries from the terminal, the user must select "t" from the database processing menu. After selecting this option, the system will prompt the user to enter the queries in the format identical to the format for query file in Figure 27. Each request should be separated by a "@" character and the last request should be followed by a "$" character. The screen display for this procedure is shown below.

```
Enter your choice
      (d) - display schema
      (c) - create covering relation between two objects
      (m) - mass load from a data file
      (f) - read in a group of queries from a file
      (t) - read in queries from the terminal
      (x) - return to previous menu

Action --- > t


Please enter your transactions one at a time.
You may have multiple lines per transaction.
Each transaction must be separated by a line that
      only contains the character '@'.
After the last transaction, the last line must consist only
      of the '$' character to signal end-of-file.

Input the transactions on the following lines :

todd.retrieve firstn, lastn
@
bertha.retrieve firstn, lastn, objectid
$
```

After inputting the request, the system will generate a selection request menu. This menu is visually and functionally identical to the previous request menu, which was used to load requests from a file. Selection of a request by number will perform the query and retrieve the data. The screen display of the query menu and a sample query are shown below.

```
1       todd.retrieve firstn, lastn

2       bertha.retrieve firstn, lastn, objectid

Pick the number or letter of the action desired
      (num) - execute one of the preceding queries
      (d)   - redisplay the file of queries
      (x)   - return to the previous menu

Action --- > 2


OBJECTID    |LASTN      |FIRSTN     |
------------------------------------------
01565702    |Smith      |Bertha     |
01565708    |Smith      |Sue        |
01565710    |Smith      |Joe        |
01565715    |Smith      |Todd       |

Pick the number or letter of the action desired
      (num) - execute one of the preceding queries
      (d)   - redisplay the file of queries
      (x)   - return to the previous menu

Action --- >
```

F. CONCLUSION

Before concluding this tutorial it is important to note that the object-oriented interface, in its current state of development, cannot perform on-line updates or deletes. In order to perform updates or deletes on the database, the user must exit the system and modify the schema and record files. However, the user must be cognizent of the fact that deletions or modifications to the schema may cause unintended changes to the inheritance structure. For instance, when modifying the schema file, the user must avoid deletions or modifications such that objects inheriting from the modified object become corrupted.

Furthermore, if an object containing attributes inherited by its subclasses is deleted, then the same attributes will also be deleted from all of its associated subclasses.

The object-oriented database does have an on-line insert capability. This is performed as a request using the keyword **insert** instead of **retrieve**. The insert procedure can be performed at the terminal or called from a text file in the same manner as the retrieval procedure. An example of an insert request would be:

<object>.insert <attribute 1 value> <attribute 2 value>,..., <attribute n value>

The desired attribute values for the record are entered in the same order that they appear in the schema. For example, assume the record to be inserted into the object GEORGE contained the following attribute values: "1" for objectid, "George" for first name, "Jones" for last name, and "13000" for salary. The resulting insertion request is shown below:

```
george.insert 1, George, Jones, 130000
$
```

It is important to note that this record does not replace the existing record, but rather is *added* to the current object GEORGE. Consequently, if each object in the database represents a single record, as in the case of the FAMILY database, then any query on George will produce two results when only one is desired.

This completes the tutorial for the object-oriented interface on MDBS. To exit the system, select "x" at each menu display. Remember to stop all processes and zero the disk upon completion of the database work. For additional information on using the Object-Oriented Interface on MDBS refer to the original thesis for this interface [Karl93].

LIST OF REFERENCES

[Bour93] Bourgeois, Paul; *The Instrumentation of the Multimodel/Multilingual User Interface*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1993.

[Booc91] Booch, G., *Object Oriented Design*, The Benjamin/Cummings Publishing Company, Inc., 1991.

[Elma89] Elmarsi, R., Navathe, S., *Fundamentals of Database Systems*, The Benjamin/ Cummings Publishing Company, Inc., 1989.

[Hsia91a] Hsiao, David K. and Kamel, M.N., "The Multimodel and Multilingual Approach to Interoperability of Multidatabase Systems," *International Conference on Interoperability of Multidatabase Systems*, Kyoto, Japan, April 1991.

[Hsia91b] Hsiao, David K., "The Relationship of Data Models and Security Requirements: Part One - The Object-Oriented Data Model and the Need-to-Know Policy," submitted to *Fifth IFIP WG 11.3 Working Conference on Database Security*, West Virginia, November 4-7, 1991.

[Hsia91c] Hsiao, David K., "The Relationship of Data Models and Security Requirements: Part Two - The Object-oriented Data Model and the Multilevel Security Policy," November 1991.

[Hsia92a] Hsiao, David K., "Federated Databases and Systems - Part I: A Tutorial on their Data Sharing," *VLDB* Journal, 1, 1992, pp. 127-179.

[Hsia92b] Hsiao, David K., "Federated Databases and Systems: Part II- A Tutorial on their Resource Consolidation," VLDB Journal, 2, 1992, pp. 285-310.

[Hsia92c] Hsiao, David K., "The Object-Oriented Database Management - A Tutorial on its Fundamentals," *Proceedings of the Second Far-East Workshop on Future Database Systems*, Kyoto, Japan, April 1992., pp. 398-416.

[John93] Johnston, Richard, *The Relational-to-Object-Oriented Cross-Model Accessing Capability in a Multimodel and Multilingual Database System*, Master's Thesis, Naval Postgraduate, Monterey, California, March 1993.

[Karl93] Karlider Turgay and Moore, John W., *Design and Implementation of an Object-Oriented Interface for the Multi-Model./Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, March 1993.

[Meek93] Meeks, Andrew P., *The Instrumentation of the Multibackend Database System*, Master's Thesis, Naval Postgraduate School, June 1993, pp. 10-15.

[Roll84] Rollins, R., *Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System*, Master's Thesis, Naval Postgraduate, School, Monterey, California, June 1984.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 52
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Lieutenant Todd G. Estes, USN 1
   66 Bliss Road
   Newport, RI 02840

5. Captain Donald H. Estes, USN 1
   Military Chair of Intelligence
   Naval War College
   Newport, RI 02841

6. Professor David K. Hsiao 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

7. Ms. Doris Mlezko 2
   Code P22305
   NAWCWPNS
   Point Mugu, CA 93042-5001

8. Lieutenant Eric M. Mueller, USN 1
   2823 Kelsey Street
   Berkeley, CA 94705

9. Ronald J. Roland 1
   500 Sloat Avenue
   Monterey, CA 93940