1993-03

The relational-to-object-oriented cross-model accessing capability in a multi-model and multi-lingual database system

Johnston, Richard Karl.

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/39867

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE RELATIONAL-TO-OBJECT-ORIENTED CROSS-MODEL ACCESSING CAPABILITY IN A MULTI-MODEL AND MULTI-LINGUAL DATABASE SYSTEM

by

Richard Karl Johnston

March 1993

Thesis Advisor: David K. Hsiao

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. · PROJECT NO · TASK NO · WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
THE RELATIONAL-TO-OBJECT-ORIENTED CROSS-MODEL ACCESSING CAPABILITY IN A MULTI-MODEL (Continued)

12. PERSONAL AUTHOR(S)
Johnston, Richard K.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM TO	March 1993	89

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

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Cross-model accessing capability; Database design; Database implementation; Database management systems; (Continued)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Conventional *database management systems* (DBMS) are stand-alone, each supporting a single *data model and corresponding data language* (ML). One organization might operate several stand-alone DBMS independently, each of which requires the knowledge of a different ML to operate. The *multi-model and multi-lingual database system* offers a different approach. This system supports multiple MLs in a single database system. Thus, a relational database user of the multi-model and multi-lingual database system can create and manipulate a database according to the relational model and the SQL data language. On the same system, a hierarchical user can create and manipulate a database according to the hierarchical model and DL/I data language, and so on.

Besides supporting many different models and languages on a single system, the multi-model and multi-lingual database system also allows the user to access a database created according to one ML as if it were created according to another. Thus, a relational user could manipulate a hierarchical database as if it is relational, i.e., the user would use a relational schema and SQL commands to manipulate a hierarchical database. The (Continued)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL David K. Hsiao	22b. TELEPHONE (Include Area Code) (408) 646-2253 · 22c. OFFICE SYMBOL CS/37

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

11. AND MULTI-LINGUAL DATABASE SYSTEM.

18. Object-Oriented Databases; Relational databases; Schema transformations; Multi-model and multi-lingual database system.

19. base-model and base-language (i.e., hierarchical and DL/I) are invisible to the user. This additional capability is termed the *cross-model accessing capability*

At this time the multi-model and multi-lingual database system supports the following MLs: relational and SQL, hierarchical and DL/I, network and CODASYL-DML, and object-oriented and the object-oriented data language. The system also supports a relational-to-hierarchical cross-model accessing capability. The work of this thesis adds to the system a relational-to-object-oriented cross-model accessing capability. A relational user can now access an object-oriented database using SQL commands and viewing the object-oriented database via a relational schema. The thesis analyzes the semantic equivalencies of the relational and object-oriented data models. The analysis is necessary in order to establish the rules for transforming the object-oriented schema into an equivalent relational schema. The work also describes the software design and integration with the existing system and outlines future development steps for new cross-model accessing capabilities.

THE RELATIONAL-TO-OBJECT-ORIENTED CROSS-MODEL ACCESSING CAPABILITY IN A MULTI-MODEL AND MULTI-LINGUAL DATABASE SYSTEM

by
Richard Karl Johnston
Lieutenant Commander, United States Navy
A.B., University of Georgia, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1993

Author: [redacted]
Richard Karl Johnston

Approved By: [redacted]
David K. Hsiao, Thesis Advisor

[redacted]
Thomas Wu, Second Reader

[redacted]
CDR G. Hughes, Acting Chairman,
Department of Computer Science

ABSTRACT

Conventional *database management systems* (DBMS) are stand-alone, each supporting a single *data model and corresponding data language* (ML). One organization might operate several stand-alone DBMS independently, each of which requires the knowledge of a different ML to operate. The *multi-model and multi-lingual database system* offers a different approach. This system supports multiple MLs in a single database system. Thus, a relational database user of the multi-model and multi-lingual database system can create and manipulate a database according to the relational model and the SQL data language. On the same system, a hierarchical user can create and manipulate a database according to the hierarchical model and DL/I data language, and so on.

Besides supporting many different models and languages on a single system, the multi-model and multi-lingual database system also allows the user to access a database created according to one ML as if it were created according to another. Thus, a relational user could manipulate a hierarchical database as if it is relational, i.e., the user would use a relational schema and SQL commands to manipulate a hierarchical database. The base-model and base-language (i.e., hierarchical and DL/I) are invisible to the user. This additional capability is termed the *cross-model accessing capability*

At this time the multi-model and multi-lingual database system supports the following MLs: relational and SQL, hierarchical and DL/I, network and CODASYL-DML, and object-oriented and the object-oriented data language. The system also supports a relational-to-hierarchical cross-model accessing capability. The work of this thesis adds to the system a-relational-to-object-oriented cross-model accessing capability. A relational user can now access an object-oriented database using SQL commands and viewing the object-oriented database via a relational schema. The thesis analyzes the semantic equivalencies of the relational and object-oriented data models. The analysis is necessary in order to establish the rules for transforming the object-oriented schema into an equivalent relational schema. The work also describes the software design and integration with the existing system and outlines future development steps for new cross-model accessing capabilities.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMNS

ABDL - Attribute-Based Data Language

ABDM - Attribute-Based Data Model

KCS - Kernel Controller System

KDL - Kernel Data Language

KDM - Kernel Data Model

KDS - Kernel Database System

KFS - Kernel Fomatting System

KMS - Kernel Mapping System

MDBS - Multi-Backend Database Super-Computer

ML - Model and Language

LI - Language Interface

LIL - Language Interface Layer

UDL - User Data Language

UDM - User Data Model

I. INTRODUCTION

Database management systems (DBMS) have proliferated in response to a demand for data models and languages which are capable of expressing and accessing a wide variety of data relationships for specific applications. Unfortunately, these DBMS are stand-alone systems. The user of one system is unable to access the data maintained by a different one. These databases and DBMS are termed *heterogeneous databases and systems*.

Since DBMS have evolved over an extended period, by now many organizations have two or more different systems, purchased at different times and run independently, among which data sharing and resource consolidation are impossible. Moreover, the introduction of new stand-alone DBMS into the workplace continues, complicating the work environment of organizations even more. The *multi-model and multi-lingual database system* at the Naval Postgraduate School is a research platform that offers a solution to these problems.

A. SOLUTIONS TO THE PROLIFERATION OF DBMS

A comprehensive solution to the problem of heterogeneous databases and systems will address the issues of both data sharing and resource consolidation. Obviously, if it were possible to establish a single *model and language* (ML) as a universal database standard, this problem would cease to exist. A single, universally-accepted ML, however, is nowhere in sight. Despite the popularity of the relational model, for example, the older hierarchical and network models continue to survive. Although their persistence owes much, no doubt, to the cost associated with converting to a new system, it is also in great part due to the preference of database users for these other models as more accurate representations of their applications. The interest in development of the object-oriented data model, moreover, portends that new data models will continue to appear as the result of new applications.

Effectively, this multitude of stand-alone systems in the workplace restricts data accesses to those workers trained to use a particular ML on a particular set of database system and computer hardware. Any database system that resolves the problems of *database interoperability*, therefore, must accomplish the seemingly contradictory objectives of (1) providing the user with access to databases created according to many different MLs and (2) at the same time, not requiring the user to learn those new MLs. The user of such a system would access a database using whatever ML he desires. Such a system is "federated", i.e., each database retains its autonomy. The key to achieving autonomy and resolving the mutually contradictory objectives is to separate the data itself from the user's view of the data. For though a database is stored and accessed in an unique way for a specific ML, the storage structure and access method are of no concern to the user. In this sense the data model is a special pair of "glasses" through which the user views the data stored on the disk. The glasses transform the data into whichever model the user prefers.

Such transformations could occur in several different ways. A standard terminology appears below [Hsiao1992]

- Single-Model-and-Language-to-Multiple-Models-and-Languages
- Single-Model-and-Language-to-Single-Model-and-Language
- Multiple-Models-and-Languages-to-Multiple-Models-and-Languages
- Multiple-Models-and-Languages-to-Single-Model-and-Language

Briefly, the first of the above four technical approaches would restrict the user to a single, universal ML which is unlikely either to satisfy everyone or to prove easy to master. The second also presupposes a single acceptable ML, but with the additional, significant burden of converting all of the existing heterogeneous databases to this ML. The third possibility, the Multiple-Models-and-Languages-to-Multiple-Models-and-Languages, appeals to the Department of Defense and similar large, multi-department organizations, which often require data sharing via networks over great distances. This approach, however, is intractable from a technical standpoint. Furthermore, it would solve the data sharing problems of existing systems without replacing them. Even if this approach were practical, it fails to address the problem of resource consolidation and, in general, would further complicate an already complex database installation.

The last approach, the Multiple-Models-and-Languages-to-Single-Model-and-Language, is used in the Naval Postgraduate School Laboratory for Database Systems Research [Hsiao1989]. This approach takes advantage of the separation of base data from viewed data. All of the base data are stored according to a single, *kernel* model. Similarly, there is a separation of transactions written from transactions executed. All written transactions are translated into a single, kernel language for execution. An immediate benefit gained from having a single data ML is a consolidation of resources, since now only one database system is needed to support the base data and to execute the transactions. This system will support any number of data models and languages via software interfaces that transform schemas and translate languages into those of the kernel. This feature of the system is referred to as the *multi-model and multi-lingual capability*. Software interfaces also allow a user, using a familiar ML, to access a database created according to an unfamiliar ML by further transformations and translations between the models. This is referred to as the *cross-model accessing capability*. To the user, the transformations and translations are transparent.

By adding the interface software, the multi-model and multi-lingual database system with cross-model accessing capabilities achieves two objectives:

- It supports any number and type of MLs.
- It allows a database user to access a heterogeneous database as if it is homogeneous to the user. --

Figure 1 depicts graphically the multi-model, multi-lingual, and cross-model accessing capabilities.

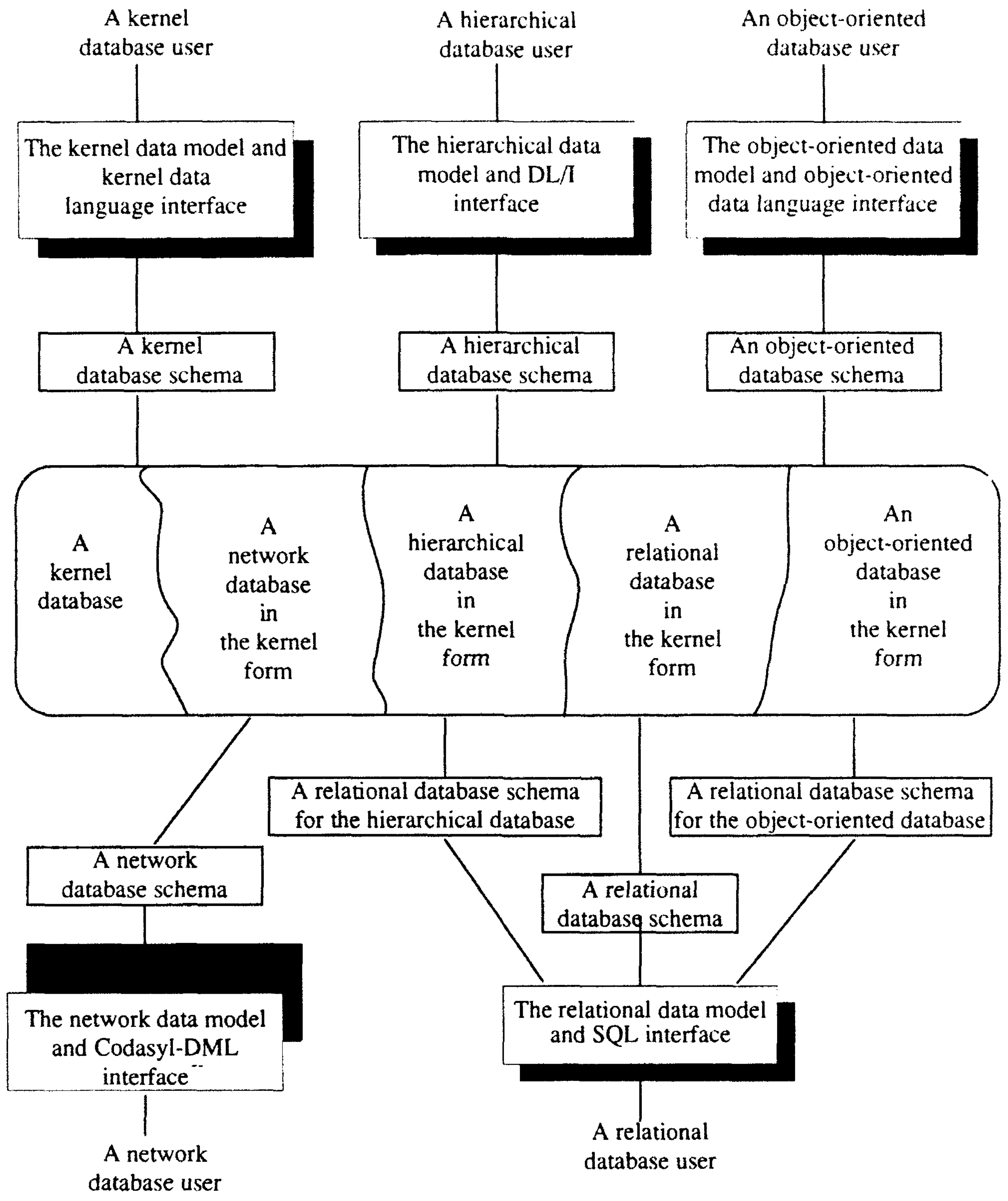

A kernel database user
A hierarchical database user
An object-oriented database user
The kernel data model and kernel data language interface
The hierarchical data model and DL/I interface
The object-oriented data model and object-oriented data language interface
A kernel database schema
A hierarchical database schema
An object-oriented database schema
A kernel database
A network database in the kernel form
A hierarchical database in the kernel form
A relational database in the kernel form
An object-oriented database in the kernel form
A relational database schema for the hierarchical database
A relational database schema for the object-oriented database
A network database schema
A relational database schema
The network data model and Codasyl-DML interface
The relational data model and SQL interface
A network database user
A relational database user

Figure 1. The Multi-model, Multi-lingual, and Cross-Model Accessing Capability

B. SCOPE OF THE THESIS

The design and implementation of software interfaces for databases and languages is contained in works on relational and SQL [Rollins1984], network and CODASYL-DML [Emdi1985], hierarchical and DL/I [Benson1985], functional and DAPLEX [Mak1992], and object-oriented and the object-oriented language [Moore1993]. Each of these data models is translated into the *kernel data language* (KDL). Past contributions also include the design and implementation of a relational-to-hierarchical cross-model accessing capability, which enables a relational database user to access a hierarchical database as if it is relational, i.e., using SQL commands to access the data according to a relational schema of a hierarchical database.

This thesis adds to the existing system a relational-to-object-oriented cross-model accessing capability. A relational database user now can access an object-oriented database using SQL commands and viewing the object-oriented database via a relational schema. Though similar in concept to the relational-to-hierarchical, the relational-to-object-oriented cross-model accessing capability requires a schema transformer and language translator that differ significantly. The thesis compares semantic equivalencies between the object-oriented and relational models and identifies methods and deficiencies in the transformation. The thesis also elaborates the design and implementation of the cross-model accessing capability in order to facilitate the integration into the system of future, similar capabilities.

II. THE MULTI-MODEL AND MULTI-LINGUAL DATABASE SYSTEM

The *multi-backend database super-computer* (MDBS) at the Laboratory for Database Systems Research supports research into the problems of heterogeneous databases. The MDBS hardware and the software (i.e., the multi-model and multi-lingual database system it supports) demonstrate the feasibility of a federated database design based on the Multiple- Models - and - Languages - to - Single - Model - and - Language transformations articulated in Chapter I.

A. THE MULTI-BACKEND DATABASE SUPER-COMPUTER

Requirements for the MDBS design included optimal performance as well as resource consolidation and a capability for data sharing. The following five features, identified as requisite characteristics of federated databases in [Hsiao1992], have guided the system design and development of the MDBS software and hardware:

- Transparent access to heterogeneous databases
- Local autonomy of each database
- Multiple-model and multiple-lingual capabilities
- Multiple-backends
- Effective and efficient access and concurrency control

Figure 2, taken from [Hsiao1991], depicts the configuration of the MDBS. Each backend contains portions of the base data as well as a complete copy of the meta data, as indicated. The base data is divided evenly into clusters across the backends. A single controller issues commands to the backends simultaneously over a network. On receiving a command from the controller, each backend performs manipulations of data coming from the secondary storage disk, using the "processing-on-the-fly" technique first proposed for CASSM [Su79] and the "logic-per-head" design first proposed in DBC [Bannerjee1979].

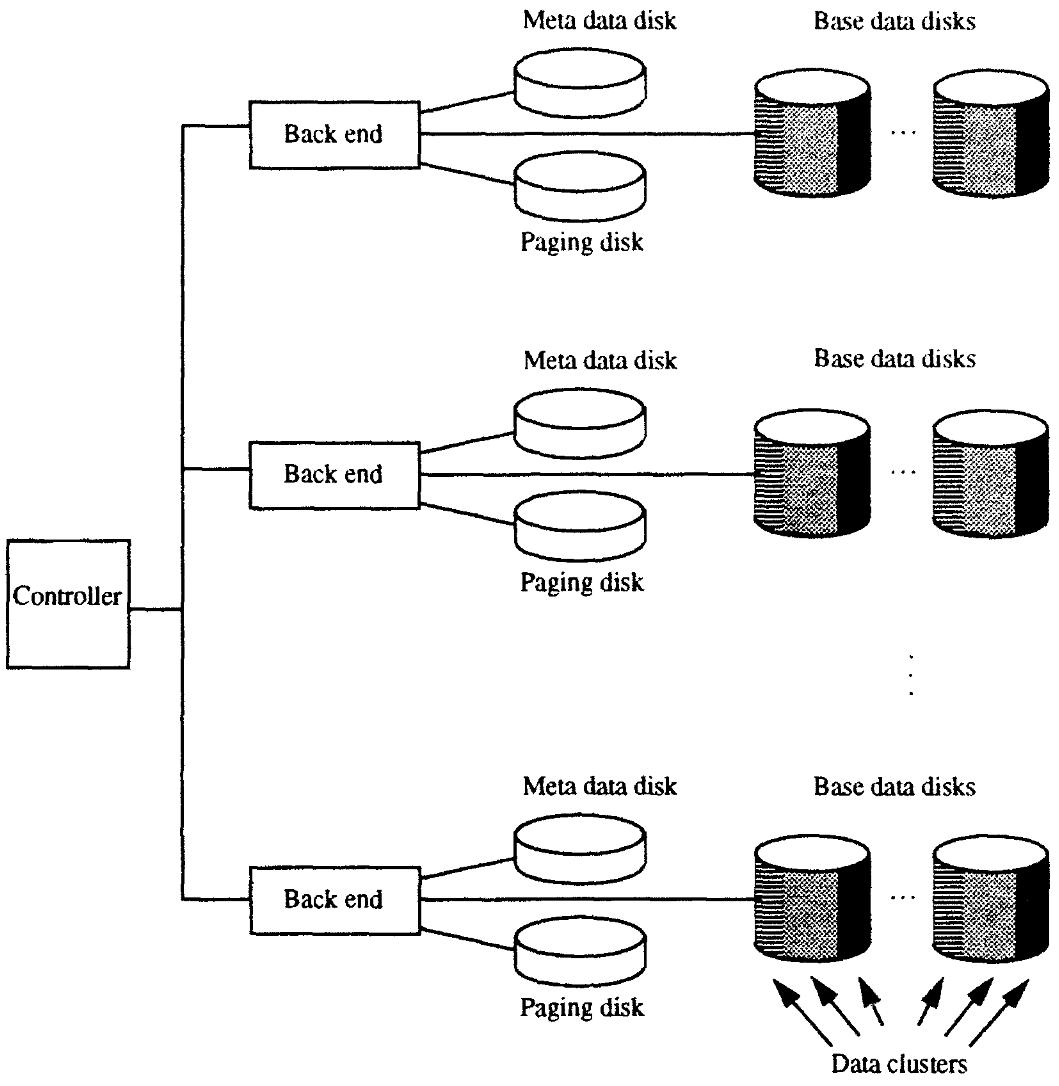


Figure 2. The Multi-Backend Database Super-Computer

The use of multiple backends combined with the "logic-per-head" processing of the disk data allows the MDBS to achieve a high degree of parallelism, concurrency, and pipelining. An intelligent clustering algorithm divides the base data across the backends in such a way as to best facilitate the execution of database transactions in parallel. An index, which is searched and stored with a dedicated set of meta-data tracks on discs, indicates the tracks on which requested data are stored. All of the backends process a data transaction simultaneously.

The use of multiple backends offers the advantages of response-time reduction and response-time invariance. The addition of parallel backends results in a corresponding reduction in response time that is directly proportional to the number of backends added. To halve a response time, one doubles the number of backends. Similarly, if the size of a database is increased, the response-time is invariant if a proportional number of backends is added to the MDBS.

B. THE KERNEL

All data in the MDBS is stored in the *kernel database system* (KDS) according to the KDM and KDL. While any one of many MLs (the relational for example) would suffice for the kernel, the attribute-based data model (ABDM) and attribute-based data language (ABDL) best support the MDBS architecture, especially its reliance on parallelism and data clustering [Hsiao1991].

1. The Data Model

In the ABDM each piece of data occurs as an attribute-value pair, notated by enclosing angular brackets:

<ATTRIBUTE, Value>

The basic structure of the ABDM is the record. A group of attribute-value pairs forms a record, and the record is notated by enclosing the list of attribute-values in parenthesis:

(<TEMP, Value>, <ATTRIBUTE, Value>, <ATTRIBUTE, Value>)

The first attribute-value pair is always the attribute "TEMP" and the name of the record type (or file) is its value. Formally, in a record no two attribute-value pairs may have the same attribute. Thus, the database consists of thousands or even millions of these records. A sample "Vehicle" database the form of the ABDM appears in Figure 3. In the sample data model, it is possible to identify various common values which could serve as *keys*. For

(<TEMP, Vehicle>, <ID, int>, <MODEL, string>, <MANUFACTURER, string>)

(<TEMP, Commercial>, <VEHID, int>, <COMPANY, string>, <REVENUE, int>)

(<TEMP, Automobile>, <AUTOID, int>, <PASSENGERS, int>)

(<TEMP, Fornauto>, <FORNAUTOID, int>, <CATEGORY, string>)

(<TEMP, Truck>, <TRUCKID, int>, <TONNAGE, int>)

(<TEMP, Company>, <CONAME, string>, <LOCATION, string>)

(<TEMP, Fornco>, <FORNCONAME, string>, <COUNTRY, string>)

Figure 3. The Vehicle Database Schema

example, one could relate Company, Fornco, and Vehicle (via the MANUFACTURER attribute) using the common value of a company's name as a key.

2. The Data Language

Though simple, the ABDL is a rich, complete data language which also supports a parallel search algorithm used in the MDBS. The ABDL includes the transactions RETRIEVE, DELETE, UPDATE, INSERT, and RETRIEVE COMMON.

The syntax for the INSERT operation appears below:

[INSERT (Record)]

The MDBS uses a series of INSERT transactions to create a database. The series of INSERTS in Figure 4 would create a "Vehicle" database according to the schema developed in the previous section.

```
[ INSERT(<TEMP, Vehicle>, <ID, 01>, <MODEL, Mustang>, <MANUFACTURER, Ford) ]

[ INSERT(<TEMP, Vehicle>, <ID, 02>,<MODEL F100>, <MANUFACTURER, Ford>) ]

[ INSERT(<TEMP, Veh,icle>, <ID, 03><MODEL, Accord>, <MANUFACTURER, Honda>) ]

[ INSERT(<TEMP, Commercial>, <VEHID, 01>,< CUSTOMER, National>,< REVENUE, 290>) ]

[ INSERT(<TEMP, Commercial>, <VEHID, 02>,< CUSTOMER, National>,< REVENUE, 290>) ]

[ INSERT(<TEMP, Commercial>, <VEHID, 03>,< CUSTOMER, National>,< REVENUE, 290>) ]

[ INSERT(<TEMP, Automobile>, <AUTOID, 01>,<PASSENGERS, 6>) ]

[ INSERT(<TEMP, Automobile>, <AUTOID, 03>,<PASSENGERS, 6>) ]

[ INSERT(<TEMP, Fornauto>, <FORNAUTOID, 03>,<CATEGORY, Compact>) ]

[ INSERT(<TEMP, Truck>, <TRUCKID, 02>,<TONNAGE, 3>) ]

[ INSERT(<TEMP, Company>, <CONAME, National>, <LOCATION, Newyork>) ]

[ INSERT(<TEMP, Company>, <CONAME, Ford>, <LOCATION, Newark>) ]

[ INSERT(<TEMP, Company>, <CONAME, Honda>, <LOCATION, Tokyo>) ]

[ INSERT(<TEMP, Fornco>, <FORNCONAME, Honda>, <COUNTRY, Japan>) ]
```

Figure 4. The Vehicle Database

The ABDL syntax for the remainder of the transactions includes a *query* composed of booleans and logical AND/OR connectors. The syntax for the DELETE request is indicated below:

[DELETE (Query)]

The following series of transactions would delete from the Vehicle database the Vehicle with ID number 01. This is a Mustang and it refers to the Automobile record with AUTOID = 01 as well as to the Vehicle record with ID = 01. The user must delete both records in order to maintain the integrity of the database:

[DELETE((TEMP = Vehicle) and (ID = 01))]

[DELETE((TEMP = Automobile) and (AUTOID = 01))]

For the UPDATE transaction, one adds a modifier:

[UPDATE ((Query) (Modifier))]

The following transaction changes the location of the Ford company from Newark to Detroit:

[UPDATE((TEMP = Company) and (NAME = 'Ford') (LOCATION = 'Detroit'))]

For the RETRIEVE transaction, additional fields indicate desired attributes and grouping characteristics. The retrieved attributes are designated the *target list*. A "group-by clause" indicates the grouping of the records:

[RETRIEVE ((Query) (Target list) BY group-by clause)]

The transaction below would retrieve the ID numbers of all of the Fords ordered by ID number:

[RETRIEVE(((TEMP = Vehicle) and (MANUFACTURER = Ford)) (ID) BY ID)]

The final transaction, RETRIEVE COMMON, is similar to the relational EQUI-JOIN. The transaction request includes a field for attributes common to two records. The generic syntax follows:

[RETRIEVE ((Query1) (Target list1))

COMMON (Attribute1, Attribute2)

RETRIEVE ((Query2) (Target list2))]

The transaction below would retrieve the models and ID numbers of all of the vehicles manufactured in Newark, ordered by ID number:

[RETRIEVE((TEMP = Vehicle) (MODEL, ID) BY ID)

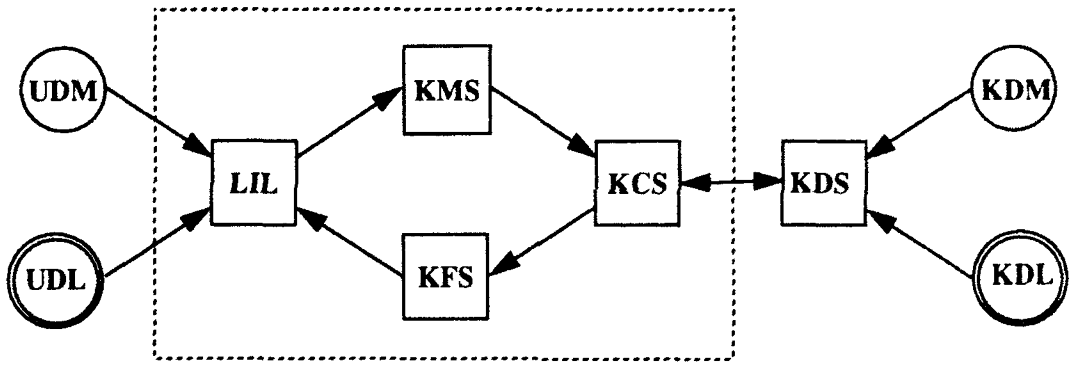
COMMON (Manufacturer, Name)

RETRIEVE((TEMP = Company) and (LOCATION = Newark))]

C. THE MULTI-MODEL AND MULTI-LINGUAL CAPABILITY

The multi-model and multi-lingual database system is based on the Multiple-Models-and-Languages-to-Single-Model-and-Language transformations discussed earlier. The single ML consists of the KDM and KDL, i.e., the ABDM and ABDL. The Language Interface (LI) software, therefore, must transform each of the multiple MLs into the attribute-based ML. This capability, which allows users to create, maintain, and manipulate databases of many different MLs on a single system, is designated the multi-model and multi-lingual capability.

Figure 5, taken from [Hsiao1992], depicts the modular software design of the LI. The four modules inside the dashed box (i.e., LIL, KMS, KFS, and KCS) perform the



▢	System Module
○	Data Model
◎	Data Language

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

Figure 5. The Software Design of a Language Interface

transformations and translations from the *user data model* and *user data language* (UDM/UDL) to the KDM/KDL and back again. Extensive literature documents the functions of these software modules, including that of [Demurjian1987] and [Benson1985]. Briefly, the function of each follows:

- LIL - Performs the functions necessary for hand-shaking with the user interface and routing results to the user.
- KMS - Builds the schema according to the UDM and parses the language requests. The KMS maps each transaction into the kernel ML (i.e. the attribute-based ML).
- KCS - Passes ABDL commands to the KDS for execution and receives results in the kernel form.
- KFS - Performs the functions of the KMS in reverse. When executing an ABDL transaction, the KDS returns the result via the KFS, which re-formats the result into the equivalent UDM and UDL.

Figure 6 depicts the modular software design of the multi-model and multi-lingual database system. The MDBS supports interfaces for relational, network, hierarchical, and object-oriented databases. An interface design for the Functional database also is complete, but lacks a working implementation at this time. For each of these different user models and its respective language, the four modules described briefly above are necessary. Since the basic functionality of the interface is the same regardless of the requirements of a specific user model and language, much of the software of these four modules is similar.

D. THE CROSS-MODEL ACCESSING CAPABILITY

The multi-model and multi-lingual capability does not solve entirely the problems of heterogeneous databases. For though the user does have the capability now to create and manipulate a database using any of the four models and languages discussed in the preceding section, the multi-model and multi-lingual capability alone does not enable another user of a ML to access this database which was originally created on a different data

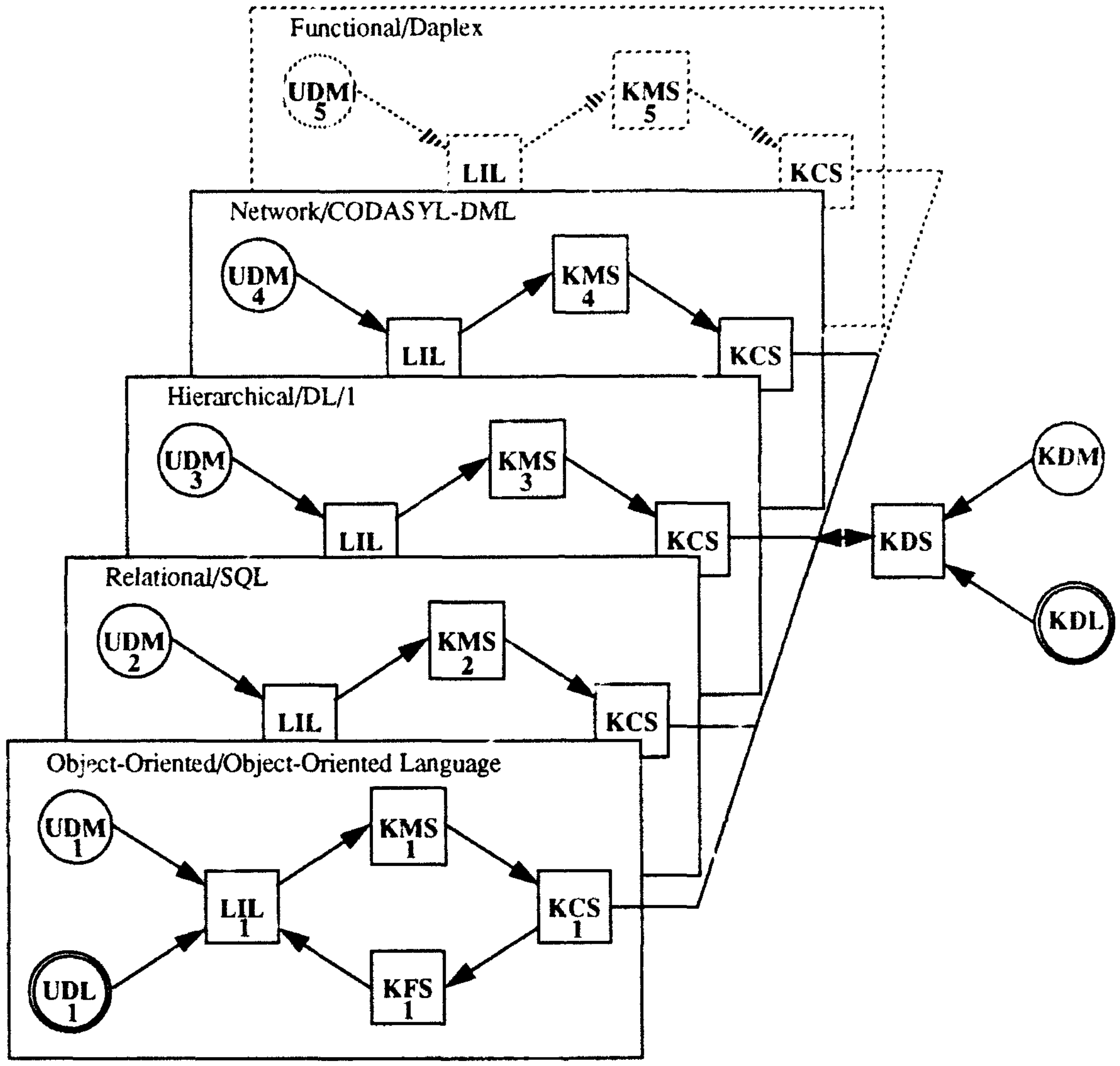
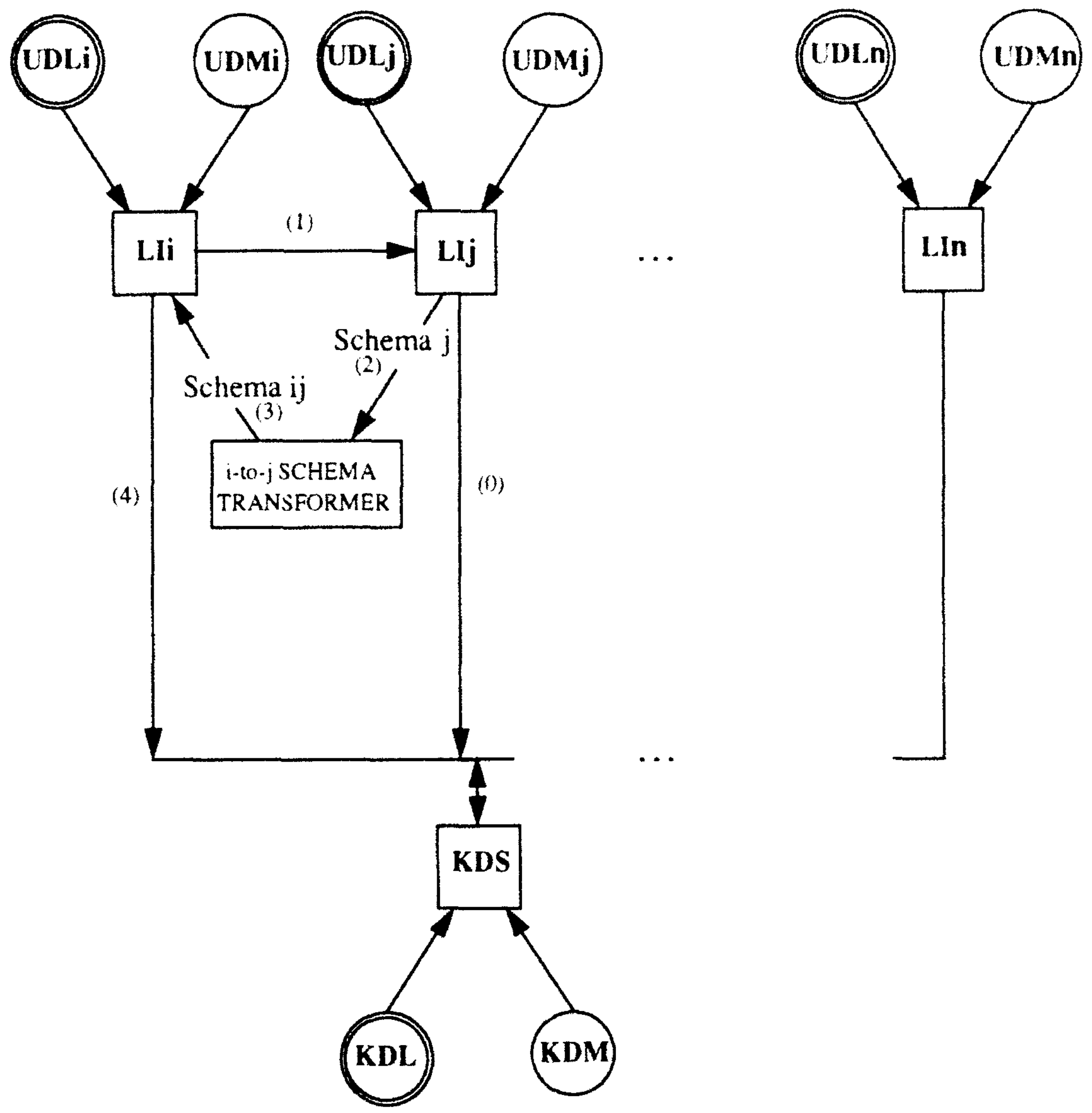


Figure 6. The Modular, Conceptual Software Design for the Multi-Model and Multi-Lingual Interfaces

model and for transactions in a different data language. For this, one requires additional capability, designated the cross-model accessing capability [Hsiao1992].

The subject of this thesis is such a cross-model accessing capability that will enable a user to access an object-oriented database as if it is relational. For example, an object-oriented database user working on the multi-model and multi-lingual database system can create an object-oriented database that he can access using an object-oriented language. Now, the cross-model accessing capability will enable a second, relational database user to log onto the same system and access the object-oriented database as if it is relational. The system will display to this new user a relational schema and will respond to SQL commands. Although, theoretically, such cross-model accessing is possible among all of the four databases in any combination, prior to this thesis the cross-model capability existed only for accessing a hierarchical database using the relational model and SQL language [Zawis1987].

Like the individual LIs, the cross-model accessing capability becomes a reality by developing additional software. Moreover, the additional software is much less complicated than the conceptual and modular LIs depicted in Figure 6. Figure 7 illustrates the cross-model design. A user of database i wishes to access database j using i's model and language. To accomplish this, it is not necessary to implement another LI. As depicted in Figure 7, LIi obtains database j's schema, which is transformed into that of i. Using this transformed schema, now the user of database i can manipulate database j using i's language. The new software required for the cross-model accessing capability includes the schema transformer and a number several other procedures, which are added to i's LI. The cross-model access is entirely transparent to the user of database i.



(0) A user of database j has been using the database j with the the schema j.

(1) A user of i requests the use of a database, j.

(2) The schema of database j, i.e., schema j, is fed into the i-to-j schema transformer.

(3) The output from the transformer is an equivalent schema, schema ij, for the same database, j.

(4) The user of i can now access database j with the new schema, ij

Figure 7. The Cross-Model Accessing Capability

III. THE OBJECT-ORIENTED DATABASE

The object-oriented data model is the most expressive yet developed. Unlike the relational and other established DBMS, however, the object-oriented lacks a standard specification of its model and language, or any standardized set of capabilities beyond the general features which distinguish the object-oriented paradigm.Thus, like all object-oriented databases, the object-oriented LI in the multi-model and multi-lingual database system is unique [Moore1993].

A. THE SCHEMA

The object-oriented schema is based on the abstract concepts of *specialization* and *generalization* expressed through the fundamental object-oriented data structure, the *class*. In an object-oriented database a class consists of a named set of attributes and transactions. A class is a specialization of another class if it contains all of the attributes and transactions of the other class plus some additional attributes and/or transactions the other class does not contain. A specialization of a class is called a *subclass* of that class. A generalization of a class, on the other hand, contains some, but not all, of the attributes and transactions of another class. A generalization of a class is called a *superclass* of that class. The term *inheritance* also describes this relationship. A subclass, as a specialization of a superclass, acquires (i.e., inherits) all of the attributes and transactions of its superclass. Conversely, the superclass, as a generalization of its subclasses, does not acquire any of the attributes and/or transactions of its subclasses. A subclass may inherit attributes and transactions from more than one superclass, a relationship termed *multiple inheritance*. [Elmasri1986].

The schema defined thus far forms a directed lattice of an unlimited number of levels. The additional abstract concept of *aggregation* completes the object-oriented schema structure. Aggregation describes a class which includes as attributes one or more other classes termed *component classes* [Elmasri1986]. A class that includes one or more component class attributes is called a *composite class*. A component class may itself include component class attributes which are in turn composite classes in themselves. A component class may have its own inheritance lattice as well. The schema's lattice can extend on indefinitely in this way.

Figure 8 depicts an object-oriented version of the attribute-based Vehicle database referred to earlier in Figure 3. Key values of subclasses have been dropped. The solid lines indicate the inheritance relationship, and the dashed lines indicate components. The phrases "is-a-kind-of" and "is-a-part-of" further describe the inheritance and the component relationships respectively. Since the schema is a directed lattice, these descriptors work one way only. Thus, an Automobile "is-a-kind-of" Vehicle, but the converse is not true, i.e., the Vehicle is not "a-kind-of" Automobile. Similarly, the Company class "is-a-part-of" the Vehicle class, but the opposite is not true.

Since any marketable DBMS allows the user to modify the schema after instantiating the database, operational systems such as ORION, IRIS, and GEMSTONE include this capability [ACM1989]. Schema modification, however, is not necessary to demonstrate the Multiple-Models-and-Languages-to-Single-Model-and-Language concept in the multi-model and multi-lingual database system. The system, therefore, lacks this capability, though it could be added in the future if desired.

B. THE DATABASE

An instance of a class is an *object*. As the class defines the structure of the object-oriented schema, the object defines the instantiation of the classes. The Vehicle database could include any number of objects of the classes Vehicle, Commercial, Automobile, Fornauto, Truck, Company, and Fornco, each of which would correspond to an actual thing or concept in-the real world. It is important to note that an object of the class Fornauto, in addition to its own attributes, includes all of the attributes and transactions from its

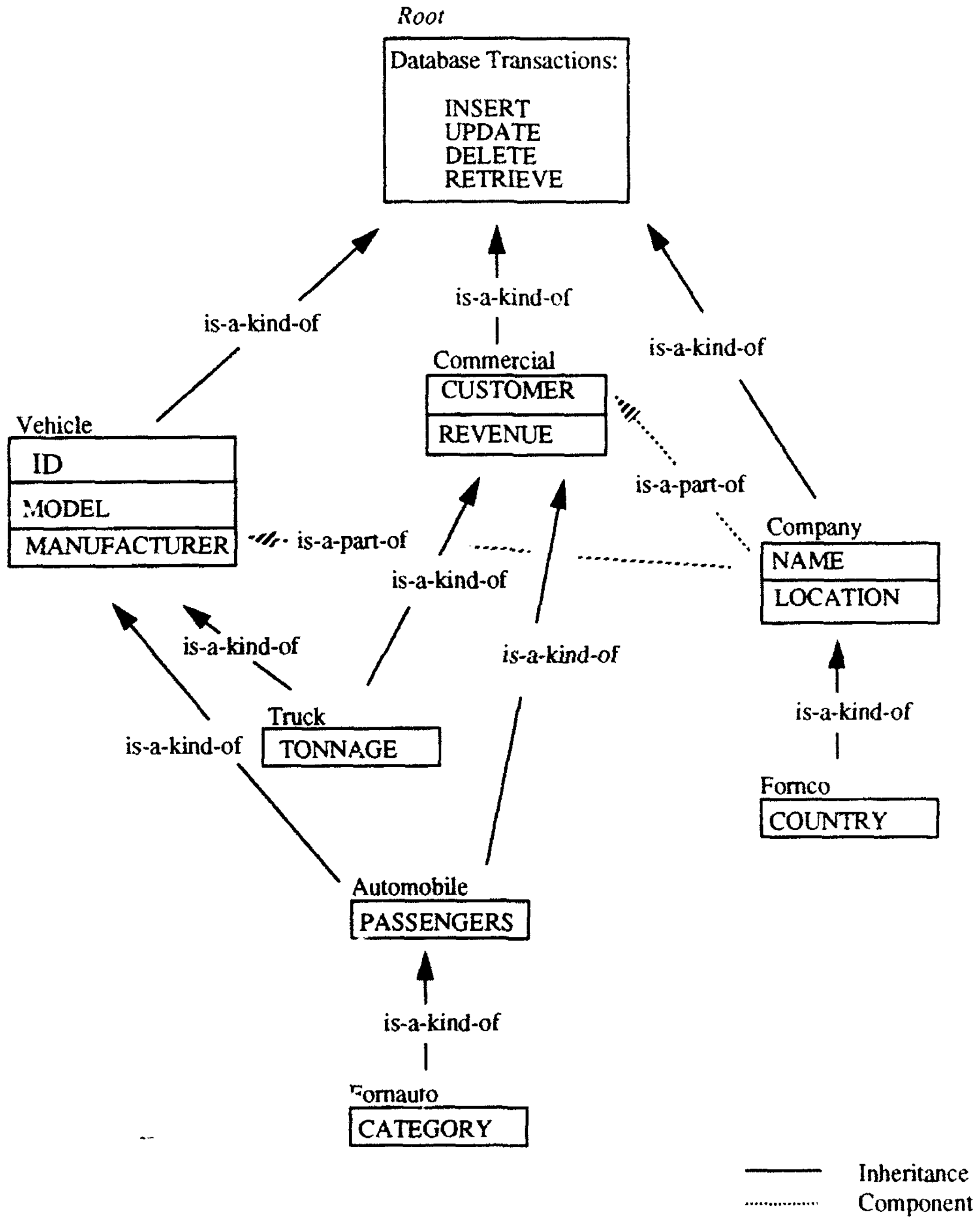


Figure 8. The Object-Oriented Schema of the Vehicle Database

superclasses Automobile, Vehicle, Commercial, and *Root* via class inheritance. An object which is an instance of the class Vehicle, however, will include, in addition to its own, only those attributes and transactions in the *Root* class. To illustrate, Figure 9 depicts a Vehicle object with a foreign manufacturer.

Instances of component and composite classes, are referred to as *component objects* and *composite objects* respectively. A component object is an independent thing in itself, separate from the composite object of which it is a part. Thus, a single object may occur as an attribute value in many composite objects. In an object-oriented database instantiated with the data in Figure 4, the CUSTOMER attribute would have the value "National" for all instances of the composite class Commercial. The object named "National," however, exists only once in the database. The component attribute includes the entire inheritance hierarchy as well. Thus, the CUSTOMER attribute could have been an instance of the subclass Fornco.

In most object-oriented databases the system uniquely defines each object by assigning to it an *object identification value* [KimA1990]. In the multi-model and multi-lingual database system, however, the user assigns this unique identification value in the same way that he would an attribute. The value is a one-up sequence number of type integer and is called "OBJECTID." This value is assigned in the left, highest superclass in the inheritance hierarchy, excluding the *Root* class. Any object which is an instance of a subclass will inherit its OBJECTID from this superclass. In the case of multiple inheritance, a single value is assigned to the OBJECTID in all of the superclasses.

It is important not to confuse the object identification value with the relational primary key. Although in this implementation it resembles an attribute, the OBJECTID field should be invisible to the object-oriented database user.

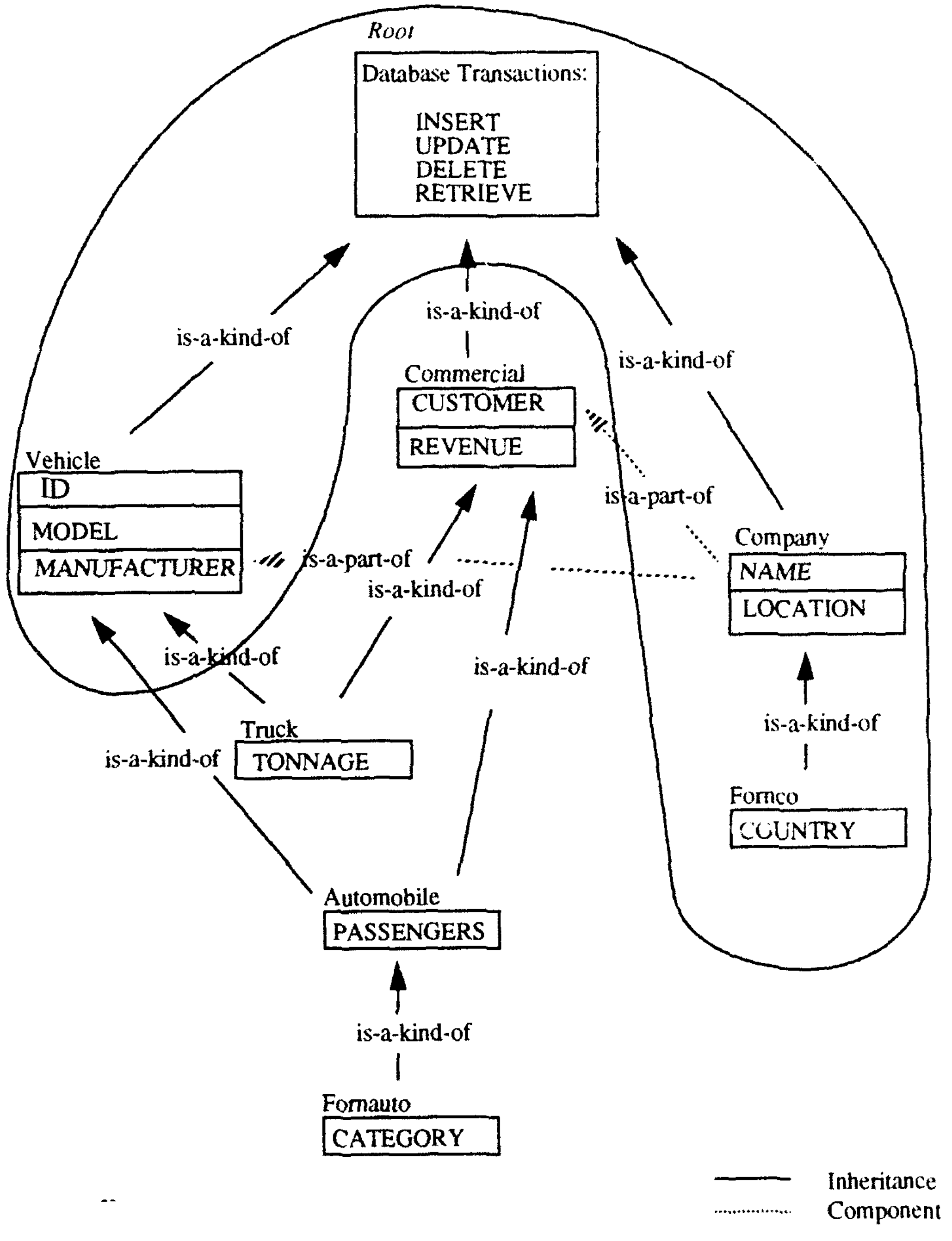


Figure 9. A Vehicle Object

C. THE TRANSACTIONS

The capability which allows the user to specify those operations which manipulate the attributes in a class is called *encapsulation*. Encapsulation is an important feature of object-oriented databases since it allows the user to control data accesses. Plans for future research include implementing this capability in the multi-model and multi-lingual database system. In the meantime, the object-oriented LI is restricted to the four standard database operations of RETRIEVE, UPDATE, INSERT, and DELETE.

The *Root* class, a standard feature of most object-oriented DBMSs, contains all of those system features that it is desirable or necessary for all of the classes in the database to inherit. In the multi-model and multi-lingual database system, all of the classes in the data model inherit the transactions in the *Root* class and have no transactions of their own. Further, of the four transactions listed in Figure 8, only RETRIEVE has been implemented.

The RETRIEVE transaction is limited to two levels of the inheritance hierarchy. The remaining three transactions and the capability to query across more than two levels of inheritance will be added in future work on the object-oriented LI. This thesis aims only to demonstrate relational-to-object-oriented cross-model accessing via schema transformations. Completing the object-oriented transactions, therefore, is not an objective of this work.

1. Creating the Database

To create an object-oriented database all of the attributes of each object are listed in the order of a descending traversal of the inheritance hierarchy. The traversal begins with the left, highest superclass and ends with the last attribute of the object's class. The object-oriented KMS maps the schema to an equivalent ABDM in which each class corresponds to a single attribute-based record. The KMS adds the OBJECTID field to each of the attribute-based records in order to link all of the classes of an object. The KMS then translates the list of attributes into a series of ABDL INSERTs, and the KCS enters them into the system. Figure 10 depicts the INSERTs which would create an object-oriented version of the Vehicle database.

```
[ INSERT(<TEMP, Vehicle>, OBJECTID, 1>, <Id, 01>, <MODEL, Mustang>, <MANUFACTURER, 5) ]
[ INSERT(<TEMP, Vehicle>, <OBJECTID,2>, <Id, 02>,<MODEL, F100>, <MANUFACTURER, 5>) ]
[ INSERT(<TEMP, Vehicle>, <OBJECTID, 3>, <Id, 03>,<MODEL, Accord>, <MANUFACTURER, 6>) ]
[ INSERT(<TEMP, Commercial>, <OBJECTID, 1>,< CUSTOMER, 4>,< REVENUE, 290>) ]
[ INSERT(<TEMP, Commercial>, <OBJECTID, 2>,< CUSTOMER, 4>,< REVENUE, 290>) ]
[ INSERT(<TEMP, Commercial>, <OBJECTID, 3>,< CUSTOMER, 4>,< REVENUE, 290>) ]
[ INSERT(<TEMP, Automobile>, <OBJECTID, 1>,<PASSENGERS, 6>) ]
[ INSERT(<TEMP, Automobile>, <OBJECTID, 3>,<PASSENGERS, 6>) ]
[ INSERT(<TEMP, Fornauto>, <OBJECTID, 3>,<CATEGORY, Compact>) ]
[ INSERT(<TEMP, Truck>, <OBJECTID, 2>,<TONNAGE, 3>) ]
[ INSERT(<TEMP, Company>, <OBJECTID, 4>, <NAME, National>, <LOCATION, Newyork>) ]
[ INSERT(<TEMP, Company>, <OBJECTID, 5>, <NAME, Ford>, <LOCATION, Newark>) ]
[ INSERT(<TEMP, Company>, <OBJECTID, 6>, <NAME, Honda>, <LOCATION, Tokyo>) ]
[ INSERT(<TEMP, Fornco>, <OBJECTID, 6>, <COUNTRY, Japan>) ]
```

Figure 10. Creating the Object-Oriented Database

In the case of composite objects, the user assigns an OBJECTID value to the component class attribute rather than a key as in the attribute-based database. This artificiality would normally be hidden from the user, who would make a selection from a list of existing database objects.

2. Accessing the Database

Because only the RETRIEVE query is functioning at this time, the comments on object-oriented transactions which follow are general rather than specific to the implementation. They describe the constraints inherent in the design of the object-oriented model.

An object-oriented transaction against a class affects all instances of that class's subclasses. Thus, a RETRIEVE or DELETE against the Vehicle class in the object-oriented database of Figure 8 will act not only upon all Vehicle objects, but upon all Truck, Automobile, and Fornauto objects as well. On the other hand, a transaction does not affect instances of a subclass object's superclasses. A retrieval against the Truck class, for example, will act only upon Truck objects.

An object-oriented transaction against a component class, either directly or through the composite object of which it is a member, raises concerns with its referential integrity. Deleting the Company class object named "National" from the Vehicle database affects all of the composite objects which inherit the class Commercial. The same holds true for UPDATEs of component objects. Inserting a component object, however, can affect no existing objects, and the only integrity constraint associated with the INSERT transaction is maintaining the uniqueness of each of the OBJECTIDs in the database. RETRIEVE transactions do not alter the state of the database and are, therefore, of no concern with respect to database integrity.

In the multi-model and multi-lingual database system the user is responsible for enforcing the overall integrity of the database through the assignment of unique OBJECTID values. The user must also enforce the referential integrity in the component and composite classes.

IV. EQUIVALENT RELATIONAL SEMANTICS

The object-oriented data model owes its richness primarily to three features: inheritance, composite classes, and encapsulation. The relational model achieves an equivalent semantics of inheritance and composite classes through the use of keys. Similarly, by the use of security views and integrity constraints the relational model can approximate encapsulated transactions.

A. GENERALIZATION, SPECIALIZATION, AND AGGREGATION

At the highest level it is the abstract concepts of generalization, specialization, and aggregation which an equivalent relational schema must express. While these abstractions are inherent in the object-oriented class structure, the equivalent relational schema must adhere to some system of transformational rules for assigning keys and constructing relations in order to express them.

1. The Object Identification Value as the Key

The obvious structural differences between the relational and object-oriented models reflect deeper, underlying differences in approaches to data modeling. These differences become apparent upon examination of the object identification value.

In the relational model each tuple of a relation must have a unique attribute value, or group of attribute values, which is the *primary key*. To the user every instance of a relation appears different from every other instance in some way. In the object-oriented model this constraint does not apply. An object-oriented database may include any number of identical objects as database items. The system identifies each object by assigning it a unique object identification value, but to the user all of the objects would look exactly the same. Furthermore, unlike a relational key, the object identification value is assigned to an object only once, and if the object is deleted, the value is never again used. Thus, the object-oriented identification value is integral to an object's existence in the database system rather than only a descriptive attribute. The system implementation must have this value in order to manipulate the different objects in the database, but it is of no use to the object-oriented user, nor is it reflected in the data model. In the absence of any object-oriented key values, however, the transformation to a relational schema is only possible by providing the user with this object identification value as a key.

On the object-oriented side, such a use of the object identification value is a definite, if subtle, corruption of the data model. Now, the user sees a code which appears to be another field describing the object. It is not integral to the object and, depending on the method of transformation, it may only refer indirectly to all of the different parts of a fragmented object via *foreign keys*.

On the relational side, this means that any attributes of a relation will depend only upon the object identification value as primary key, and the relations will, therefore, contain no functional dependencies [KimB1990]. The object identification value can serve as a foreign key also, in which case a relation would have an inclusion dependency.

2. The Class

As the relation is the structural basis of the relational model, so the class is the structural basis of the object-oriented model. The problem of transforming a structure of classes to a structure of relations, however, lies in transforming the relationships among the classes, rather than only the class itself which is a trivial operation. An equivalent relational schema requires a transformation of the relationships among object-oriented classes into equivalent relational structures. Two features of the object-oriented data model relate one class to another: composite classes and inheritance.

The composite class expresses the concept of aggregation, an abstraction which comes naturally to the relational model. The composite class translates directly to a relation by the use of foreign keys. As one relation might include another via its foreign key attribute, so a composite class might include another via its object identification value.The object identification value of a component class becomes a primary key attribute of a corresponding relation. The relation corresponding to the composite class then includes this primary key value as a foreign key attribute. Further, a component class attribute (i.e., component object) of a composite object must refer to an existing object of that component class, just as in a relation a foreign key attribute must refer to an existing tuple of some other relation. Formal relational terminology describes this relationship as an inclusion dependency. For tuples r and s of relations R and S with attributes X and Y respectively.

$$\pi\langle X\rangle(r) \subseteq \pi\langle Y\rangle(s)$$

Substituting object-oriented terms—i.e., for objects r and s of component class R with attributes X, and composite class S with attributes Y, respectively—the above relation also applies. Save for the reservations concerning the equivalencies of key values addressed at the end of the previous section, the composite class loses nothing in translation.

Unlike the composite class, inheritance corresponds directly to no relational structure. On the one hand there is the system of superclasses and subclasses which make up an object, on the other there is the object itself. A separate relation could correspond to each class in an object-oriented schema, or to each possible occurrence of an object in the schema. Preserving the class distinction in the first fragments the object; preserving the object's coherence in the second loses the properties of generalization and specialization. Other methods of transformation would convey the object-oriented model, but pose a different set of problems for the resulting relational schema.

B. TRANSFORMING THE SCHEMA

A successful schema transformation must remain as faithful as possible to the nuances of structure and relationships in the original object-oriented data model. In other words, the relational schema produced by the schema transformer must reflect the various structures of the original object-oriented model, i.e., superclasses and subclasses, even though the relational model lacks comparable structures. At the same time the resulting relational schema must follow, as much as possible, accepted relational standards of normalization and design.

None of the three methods discussed below results in a perfect transformation. All produce relational schemas which reflect the functional dependencies inherent in the original object-oriented database. Of the three, the class-based approach best reflects the original object-oriented database without corrupting the relational. Accordingly, the multi-model and multi-lingual database system uses this approach for the relational-to-object-oriented cross-model accessing capability.

1. The Class-based Approach

Figure 11 illustrates the relational schema resulting from the application of the class-based approach to the Vehicle schema in figure 8. This is the simplest way to transform the schema. Each class becomes a separate relation containing all of the attributes of the class. If a class is not a subclass, its object identification value becomes the primary key attribute of its corresponding relation. If a class is a subclass, the object identification

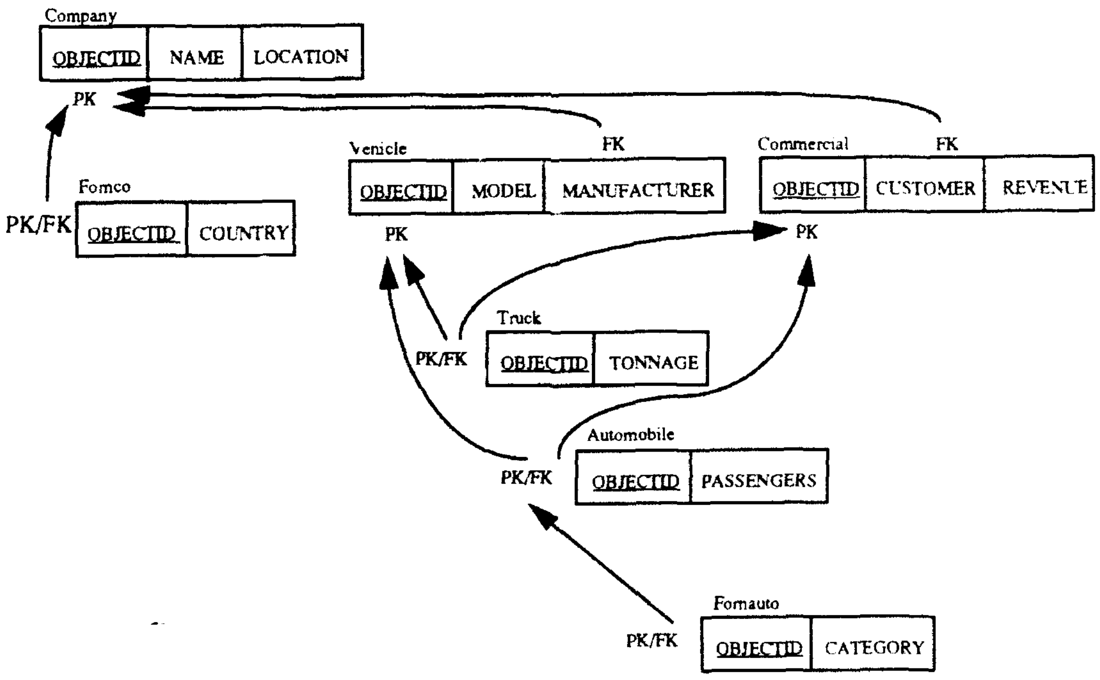


Figure 11. The Class-Based Transformation of the Vehicle Database

value becomes the primary key attribute of its corresponding relation as well as a foreign key which points to relations corresponding to its superclasses. In cases of multiple inheritance, all of the relations corresponding to superclasses of a subclass will have the same primary key value.

This method preserves all of the superclass and subclass distinctions, but results in a fragmented object. On the relational side, this transformation works well without producing nulls in the relations. An inclusion dependency holds here for the subclasses of a superclass. Formally, for tuples r and s of relations corresponding to subclass R and superclass S with attributes X and Y respectively:

$$\pi\langle X\rangle(r) \subseteq \pi\langle Y\rangle(s)$$

2. The Object-based Approach

Figure 12 illustrates the object-based approach. In this method all of an object's subclasses and superclasses are combined into a single relation. The object identification value becomes the relation's primary key. Unlike the class-based method, this method preserves very well the correspondence to an object upon which object oriented design is based. It also works well from a relational standpoint, without producing null values. On the object-oriented side, however, it loses the concept of the superclass-subclass relationship entirely. This transformation is not as satisfactory as the class-based, since it does not convey the *generalization and specialization abstractions.*

3. The Combined Approach

A third method combines the first two approaches. In this method all of tne classes in an inheritance hierarchy would become a single relation containing all of the attributes of the classes. Again, the object identification value becomes the relation's primary key. An attribute, "TYPE", would indicate of which class the object was an instance.

Although this method seems to maintain the superclass-subclass relationship, it actually would only directly indicate the superclass, i.e., Vehicle, and subclass to which an

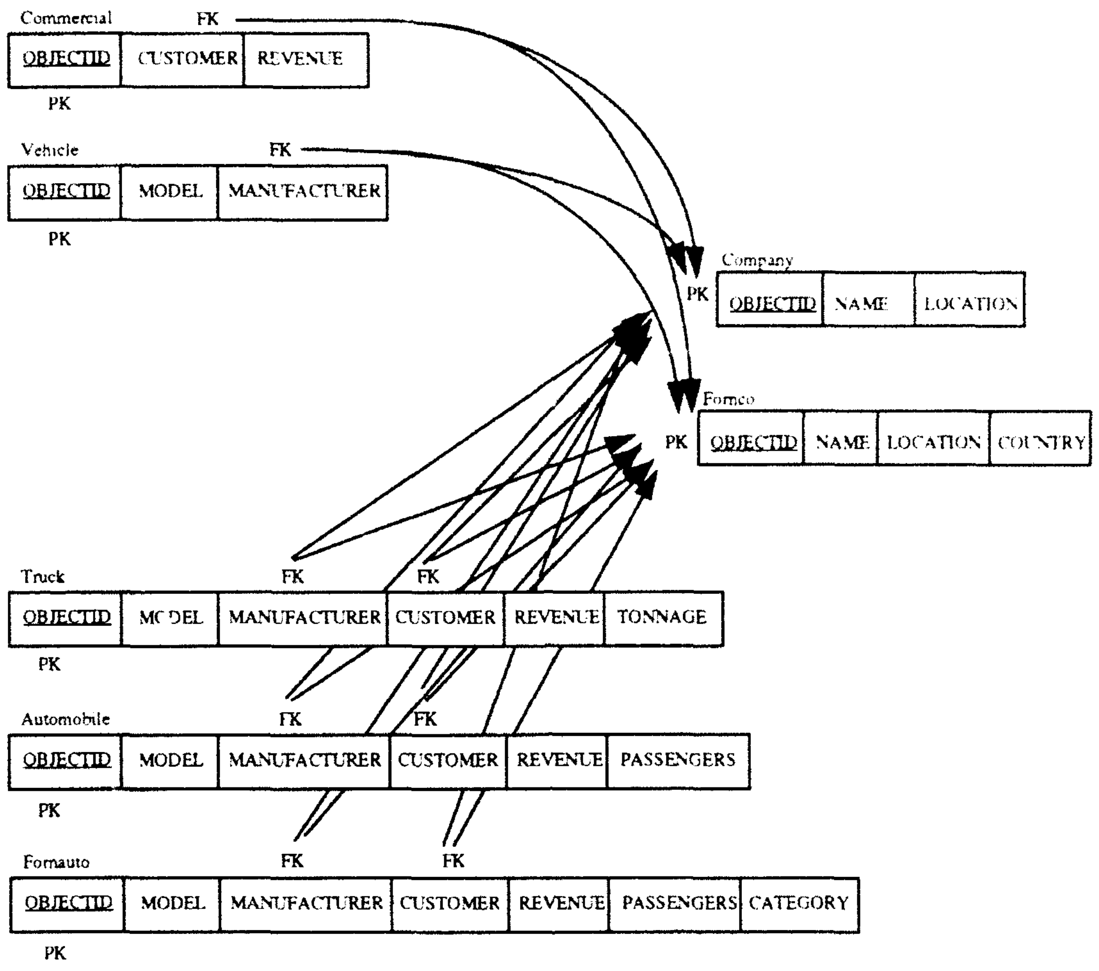


Figure 12. The Object-Based Transformation of the Vehicle Database

object corresponds. Intermediate classes in the inheritance hierarchy do not convey. There is also a pr blem with multiple inheritance. An instance of a superclass other than the one naming the relation would appear to be a subclass.

Figure 13 illustrates the combined method. The problem with multiple inheritance is resolved here by making a separate relation for each additional superclass after the first. Thus, Commercial is a relation by itself, but is also included in the Vehicle relation. An alternative solution would include all superclasses in the single relation's name, e.g., Vehicle-Commercial, and thus do away with the separate relation corresponding only to the

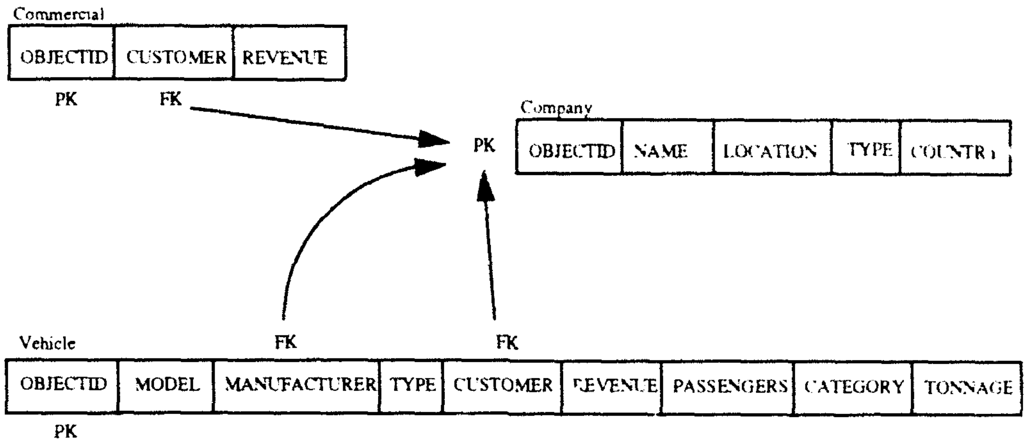


Figure 13. The Combined Class- and Object-Based Transformation of the Vehicle Database

superclass. Neither of these alternatives, however, produces a clean transformat and the problem of directly expressing the intermediate classes remains.

Another solution would add to the relations in Figure 13 boolean fields for each class in an inheritance hierarchy. The relation would reflect via the booleans all of the participating classes in the inheritance hierarchy and solve both the multiple inheritance problem and the expression of intermediate classes. This method, however, would still have the problem of assigning to the relation an appropriate name. Also, from a relational standpoint, this method would result in an unsatisfactory number of null fields.

Although, graphically, Figure 13 appears the simplest of the three methods, the problems from both the relational and object-oriented sides of the transformation make it less desirable than the other two.

C. THE TRANSACTIONS

Since the object-oriented LI lacks encapsulation, the cross-model accessing capability does not include an equivalent capability using security views and integrity constraints. The much simpler problem consisted of translating SQL commands into an equivalent series of ABDL commands constrained to maintaining the integrity of the object-oriented database.

The following comments address the affects of the four basic SQL commands upon an object-oriented database which was transformed using the class-based approach. Another method of transforming the schema would affect the database differently. Since all of the transactions affect composite classes exactly as they would relations with foreign keys, the discussion is limited to the affects on the inheritance hierarchy.

1. SELECT

The requirement to execute EQUI-JOINs in order to retrieve subclass objects characterizes the SQL SELECT. A SELECT against a Fornauto relation, for example, yields only the single attribute, CATEGORY, of which it is composed. Retrieving all of the attributes of the Fornauto object would require an EQUI-JOIN on the OBJECTID primary and foreign keys of all of the relations corresponding to the object's superclasses.

A SELECT against a relation corresponding to a class that is not a subclass, on the other hand, will act on all objects of that class as well as on all objects that are subclass objects of that relation's class. A SELECT against the Vehicle relation, for example, will yield requested attributes of all Vehicle objects in the database. It will also yield these attributes of all Fornauto, Automobile, and Truck objects in the database. Such a SELECT projects only those attributes in the Vehicle relation, however, and none from the relations corresponding to the subclasses.

The required EQUI-JOIN transactions are unnatural to the object-oriented paradigm. Unlike the SQL SELECT, an object-oriented retrieval against Fornauto would yield all of the attributes of the object, including those of classes Automobile, Vehicle, and Commercial. The inability of the relational transformation to match this capability without the use of the EQUI-JOIN is a shortcoming of the class-based approach to schema transformation.

2. UPDATE

UPDATE of a tuple of any relation affects only the object corresponding to the OBJECTID field of the relation. Since the relational model produced by the schema transformer makes no attempt to avoid duplicated values, UPDATE anomalies are not an issue.

3. INSERT

An SQL INSERT of a tuple corresponding to an object whose class is not a subclass requires no translation. Only the requirement to assign a unique OBJECTID value constrains such an INSERT.

INSERT of a tuple corresponding to a subclass object, however, acts upon all of the relations which correspond to its superclasses as well. Moreover, the INSERT transaction alters the object-oriented database. Since the inheritance hierarchy is not apparent in the relational model, the system must prompt the user to enter values for the attributes of all of the relations which make up the object or else reject the INSERT.

4. DELETE

Like the INSERT, DELETE also alters the database. An unqualified DELETE against a relation corresponding to a non-subclass, must automatically delete all tuples of the relation as well as all tuples of relations corresponding to subclasses. Similarly, a DELETE of a subclass also must delete all tuples of the subclass relation and all tuples of relations corresponding to subclasses of the subclass. Additionally, the subclass DELETE must delete those tuples corresponding to the object's superclasses which have the same OBJECTID foreign key values as the subclass instances.

In Figure 11, therefore, a DELETE against the Vehicle relation will delete from the database all Vehicle, Fornauto, Automobile, and Truck tuples. A DELETE against Automobile, however, deletes only Automobile and Fornauto tuples and those tuples of Vehicle and Commercial which are related to Automobile by foreign key OBJECTID values.

V. THE CROSS-MODEL ACCESSING SOFTWARE

The software for the cross-model accessing capability is neither a new, separate LI nor is it an interface between the relational and object-oriented LIs. Rather, the software extends the capability of the relational LI and is integrated in the relational software.

A. THE DESIGN

The software design for the schema transformer follows closely that of the previous work on the relational-to-hierarchical cross-model accessing in [Zawis1986]. For the transactions, however, the design departs from the method of the previous work, which uses the hierarchical LI procedures to process SQL transactions.

Using the object-oriented I I would require two sets of data structures, one for the relational and one for the object-oriented. The software's independence from the object-oriented LI eliminates the need for these additional data structures, with the except ion of those holding the object-oriented schema. Additionally, this independence means that SQL commands translate directly into the kernel language rather than into intermediary object-oriented transactions, greatly simplifying the language translation. Incidentally, this also allows the cross-model accessing software to avoid the temporary limitations of the object-oriented LI's transactions. Unlike the object-oriented LI, the cross-model accessing capability includes the full range of standard SELECT, INSERT, UPDATE, and DELETE transactions.

Rather than divide the cross-model accessing procedures among several modules of the relational LI, the design favors completing necessary actions in a single module, which then calls other relational modules using control statements. Thus, rather than scatter numerous procedures throughout the KMS, KC, and LIL modules, the software design favors performing all of the actions in the single module most involved in the execution of a transaction. This practice minimizes the quantity of code required and makes the procedures much easier to understand.

The reasoning of the previous paragraphs resulted in the following set of design principles for the cross-model accessing software:

- Restrict code to the relational LI and its data structures, except for the object-oriented schema.
- Change the relational LI and the object-oriented and relational data structures as little as possible.
- Cluster new procedures in as few of the relational modules as possible, rather than scatter pieces throughout the modules.
- Maintain as much as possible the modular framework of the multi-model and multi-lingual database system.

The work of [Bourgeois1993] provides a systematic plan for the integration of new capabilities in the multi-model and multi-lingual database system.

B. THE DATA STRUCTURES

The header file "llicommdata.h" contains most of the C programming language data structures used in the multi-model and multi-lingual database system. The data structures of each of the LIs, including the cross-model accessing procedures, are globals and differ only in so far as necessary to implement their respective databases. The use of common data structures and standardized naming conventions greatly reduces the quantity of new code required to add new software procedures. Code patched from one LI to a new LI often requires only minor coding changes and changing the names of the data structures in order to make it work in the new LI.

The software for the cross-model accessing capability adds no new data structures to those of the relational and object-oriented LIs. The union data structure in Figure 14 provides access to the schema of any of the MLs of the system via pointers contained in individual header files of the respective LIs. On the object-oriented side, the only data structures used for cross-model accessing are those which hold the schema. The dbid_node union points to a linked list of obj_dbid_node structs, also depicted in Figure 14, which hold information

```
union     dbid_node
          {
          struct  rel_dbid_node        *dn_rel;
          struct  hie_dbid_node        *dn_hie;
          struct  net_dbid_node        *dn_net;
          struct  ent_dbid_node        *dn_fun;
          struct  obj_dbid_node        *dn_obj;
          }

struct    obj_dbid_node
          {
          char                     odn_name[DBNLength + 1];
          int                      odn_num_cls;
          struct ocls_node         *odn_first_cls;
          struct ocls_node         *odn_curr_cls;
          struct obj_dbid_node     *odn_next_db;
          };
```

Figure 14. Object-Oriented Database Node Structures

about each object-oriented database in the system. Each of these structures in turn points to a schema, which consists of linked lists of classes, each class in turn pointing to a linked list of attributes. The class nodes also point to lists of superclasses and subclasses. The superclass and subclass nodes include pointers back to the class node that represents the superclass or subclass itself. Figure 15 depicts the object-oriented schema data structures.

The relational data structures differ in no significant way from those of the object-oriented. The relational schema, like the object-oriented, consists of linked lists of relations and attributes, without the lists of superclasses and subclasses.

It is important to note that the schema data structure represents only the data model, not the actual storage of the data. The schema provides all of the information necessary to

```
struct   ocls_node                    /*class nodes*/
        {
        char                          ocn_name[RNLength + 1];
        int                           ocn_num_attr;
        int                           ocn_supcls;
        int                           ocn_subcls;
        int                           ocn_visited;
        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
        };

struct   oattr_node              /*attribute nodes*/
        class node */
        {
        char                          oan_name[ANLength + 1];
        char                          oan_type[RNLength + 1];
        int                           oan_length;
        int                           oan_key_flag;
        struct   oattr_node           *oan_next_attr;
        };

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

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

Figure 15. The Object-Oriented Schema Data Structures

build the ABDL commands, which the KCS then passes into the KDS via system functions. On receiving an ABDL command, the KDS performs all of the necessary operations to create and manipulate the database. The LI and cross-model accessing software, therefore, need only build the appropriate ABDL commands in order to create and manipulate a database. This is a fundamental property and definite advantage of the Multiple-Models-and-Languages-to-Single-Model-and-Language technical approach.

The software design of the multi-model and multi-lingual database system includes a multi-user capability, although the system functions as a single-user system at this time. Figure 16 depicts the struct, called user_info, that holds all of the information associated with each user. This data structure contains the li_info union, also in Figure 16, which

```
struct  user_info

        {
        char                    ui_uid[UIDLength + 1];
        union   li_info         ui_li_type;
        struct  user_info       ui_next_user;
        };

union   li_info
        {
        struct  sql_info        li_sql;
        struct  dli_info        li_dli;
        struct  dml_info        li_dml;
        struct  dap_info        li_dap;
        struct  ool_info        li_ool;
        };
```

Figure 16. The User Data Structures

contains the data structures for the specific LI in use. The sql_info struct, called li_sql in Figure 16, provides access to information necessary to process SQL requests.

Figure 17 shows the sql_info struct. Typically, procedures access this struct in the first few lines of code to access all of the required information. This data structure holds information on the current database, files, and relational catalog, as well as all of the other data structures used by the LIL, KMS, KCS, and KFS modules. Since the cross-model accessing software does not use the object-oriented LI, it requires only the relational user_info data structures. The object-oriented schema is acquired through the use of a global pointer and is not attached to any user structure.

```
struct sql_info

        {
        struct   curr_db_info        si_curr_db;
        struct   file_info           si_file;
        struct   tran_info           si_sql_tran;
        int                          si_operation;
        struct   ddl_info            *si_ddl_files;
        struct   tran_info           *si_abdl_tran;
        int                          si_answer;
        union     kms_info           si_kms_data;
        union     kfs_info           si_kfs_data;
        union     kc_info            si_kc_data;
        int                          si_error;
        int                          si_subreq_stat;
        };
```

Figure 17. The sql_info Data Structure

The cross-model accessing software uses only those parts of the user data structures which are necessary to execute the transactions. Effective use of these data structures, however, requires an understanding of the overall framework of their organization. Appendices A and B contain schematics of the most important data structures.

C. THE IMPLEMENTATION

The cross-model accessing implementation depends heavily on the method that the object-oriented LI uses to map the database into an equivalent, object-oriented ABDM (i.e., ABDM(object-oriented)) in the KDS. Because the object-oriented LI maps each superclass and subclass of an object— rather than the entire object— to a separate attribute-based record, the SQL transactions map to attribute-based transactions in a specific way [Hughes1991].

Eight procedures comprise the whole of the cross-model accessing software. Appendices C, D, and E contain all eight of the cross-model procedures as well as those relational ones which are germane.

1. The Schema Transformer

The schema transformer consists of two procedures in the relational LIL: *traverse_ool_schema()* and *translate_obj_to_rel()*. After a user identifies himself as a relational user, he enters a database name at the terminal. The *r_process_old()* procedure searches first among all of the relational databases in the system for the requested database. If the search fails, the procedure calls *check_alternate_models()* which searches among the hierarchical and, finally, the object-oriented databases. If the database is object-oriented, the search procedure, *traverse_ool_schema()*, returns a pointer to the schema. Procedure *check_alternate_models()* then passes the object-oriented schema to *translate_obj_to_rel()*, which allocates a relational schema according to the transformation rules articulated in Chapter IV. After the transformation, procedure *check_alternate_models()* assigns to the sql_info user structures those values necessary to execute transactions against the object-oriented database.

Because the object-oriented LI maps the database to the kernel by class, the cross-model accessing software uses only the object-oriented schema to translate SQL transactions to the ABDL(object-oriented). The relational schema serves only to provide the user with a relational interpretation of the object-oriented database. If the object-oriented mapping were by objects, however, the software would have referred to the relational schema as well.

2. The Transactions

As noted earlier, the cross-model accessing software translates SQL commands to equivalent ABDL commands rather than to the object-oriented language. The relational LIL and KCS modules contain all of the new procedures required to perform these translations.

a. SELECT and UPDATE

For these transactions, the mapping by class makes translation unnecessary. There is a one-to-one-to-one correspondence between relational relations, object-oriented classes, and attribute-based records. An SQL SELECT or UPDATE command will affect only that record corresponding to the relation indicated. Since each relation, in turn, corresponds to a class, these comi.iands are transparent.

b. INSERT

For INSERT commands the software must maintain the object-oriented database structure of superclasses and subclasses. This means that the system must not allow the user to insert a tuple of a relation corresponding to a subclass without inserting all of the tuples of relations corresponding to the appropriate superclasses as well. Since the INSERT requires that the user enter data, the problem becomes one of user interface.

Procedure *queries_to_KMS()* initiates the cross-model accessing INSERT. A conditional checks to see if the transaction is an object-oriented INSERT request and, if so, calls procedure *insert_to_object_oriented()*. Procedure *insert_to_object_ oriented()* calls the recursive procedure *get_obj_inserts()*. If the INSERT relation corresponds to a subclass, the following message appears on the screen:

```
In order to maintain the integrity of the object-oriented base model,
INSERTs of all relations which correspond to superclasses in the
object-oriented hierarchy of the original INSERT are necessary.
INSERT a tuple for each relation when prompted. If entering from a
file, selections must offer appropriate INSERTs.If any INSERT is in
error or any superclass INSERTs are lacking, all preceding INSERTs
will be cancelled. <CR> to continue.
```

Procedure get_obj_inserts() traverses the inheritance hierarchy, at each node prompting the user to select an INSERT corresponding to the appropriate superclass before proceeding to the next. The following message appears on the screen at each node of the hierarchy:

```
INSERT a tuple of relation (relation name) now, having the same
OBJECTID attribute as the INSERT just attempted. If such a (relation
name) tuple already exists (i.e. has the same OBJECTID) you must start
INSERTs over, using a unique OBJECTID. <CR> to quit or continue.
```

For each superclass, *get_obj_inserts()* stores a separate ABDL INSERT in an array of character pointers. If the user enters an incorrect superclass or an OBJECTID value that is not unique, all of the INSERTs are cancelled. Otherwise, on completion of the traversal, control returns to *insert_to_object_oriented()*, and the procedure passes the INSERTs one by one to the KCS for entry into the kernel.

c. *DELETE*

Like the INSERT, the SQL DELETE transaction must maintain the integrity of the object-oriented database. All tuples of relations corresponding to superclasses and subclasses of tuples that satisfy the conditions of a transaction must be deleted. The OBJECTID values identify affected tuples.

The KCS module contains all of the software for execution of the cross-model accessing DELETE, since it consists of building and sending to the kernel an appropriate series of ABDL DELETE transactions and does not require user interaction. On receiving an object-oriented DELETE, the procedure *r_Kernel_Controller()* calls *object_oriented_delete()*. This procedure must first identify the OBJECTID values of all of the specified tuples by calling procedure *get_objectids()*. Procedure *get_objectids()* transforms the DELETE into an ABDL RETRIEVE command, calls the relational procedure *get_response()*, and stores the returned OBJECTID values in an array. Procedure *search_for_sub_and_superclasses()* then traverses the object-oriented schema, building at each node a series of ABDL DELETE requests—one for each OBJECTID value—and then passing them one-by-one to the KCS for execution.

VI. CONCLUSIONS

The relational data model fails to convey the richness of expression inherent in the object-oriented. The relational model is capable, however, of approximating the object-oriented data model sufficiently to enable a relational user to manipulate the database via a transformed schema. Although the resulting relational data model is not as rich, the benefits gained by increasing access without training are often greater than those achieved by learning to use the more expressive data ML.

A. SHORTFALLS

The object-oriented concepts of objects and inheritance do not transfer to the relational. This deficiency is impossible to remedy without fundamentally changing the relational data model. The transformation by class fragments the object; the transformation by object loses the subclass and superclass inheritance relationships; combined methods of transformation duplicate classes and relations and are ambiguous. These shortfalls are the consequence of fundamental differences in the two data models.

With respect to the cross-model accessing capability in the multi-model and multi-lingual database system, the capability is fully developed within the limitations of the system hardware and software. All four of the standard transactions work. The system handles all of the features of the object-oriented database, including composite classes and single and multiple inheritance.

B. FUTURE RESEARCH

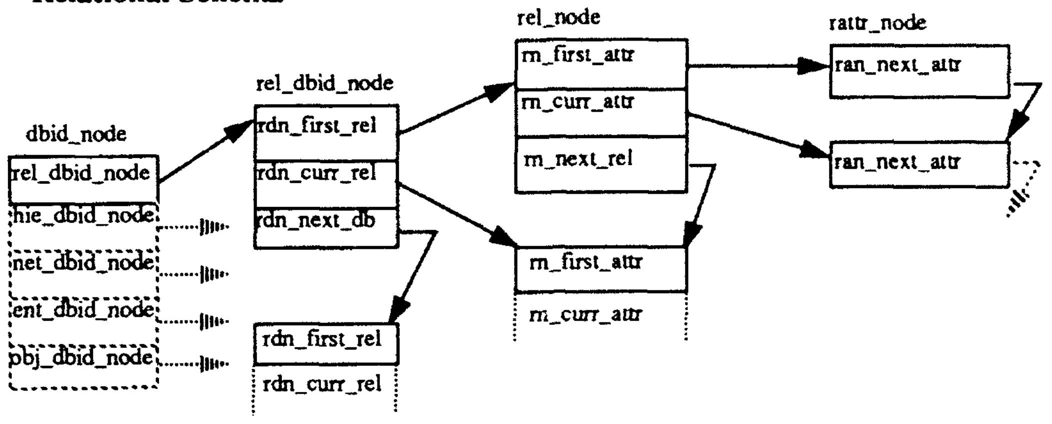
The object-oriented LI as well as the cross-model accessing capability were implemented in the C programming language, since the MDBS hardware lacked the system software to handle the object-oriented C++ language. If the object-oriented LI is converted to C++ in the future, the cross-model accessing capability must accommodate the changes.

Specifically, implementing encapsulation is greatly simplified working in an object-oriented language. When encapsulation is added to the object-oriented LI, whether or not as a result of a conversion to C++, a corresponding transformation to the relational model via security views and integrity constraints must accompany it. This complex problem would provide the subject matter of another thesis.

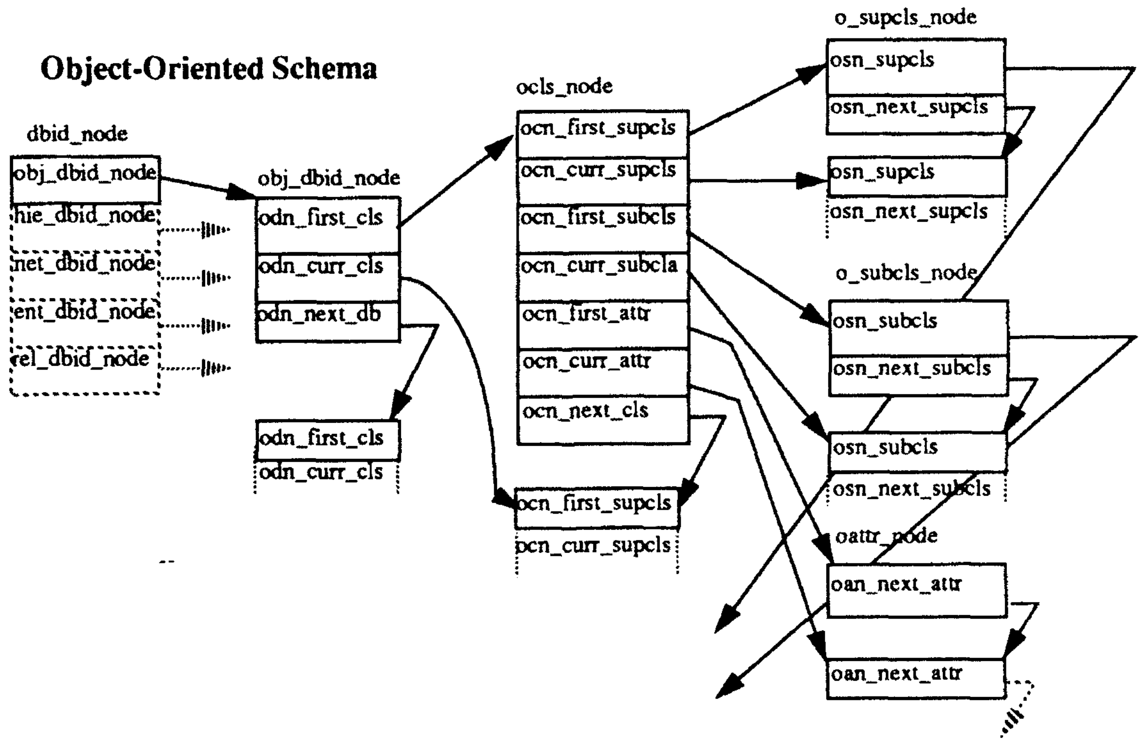
The object-oriented method of mapping to the KDS by class closely resembles the mapping of a relational database and facilitates the implementation of the cross-model accessing capability. Due to the inability to retrieve across more than two attribute-based records, however, an object-oriented retrieval of attributes from objects with more than two levels of inheritance is greatly complicated and would require buffering intermediate results and performing a series of queries in order to execute. A mapping by object, rather than by class, may resolve this problem in a simpler way. Adopting a different method of mapping the object-oriented database to the kernel will require re-working the cross-model accessing queries accordingly.

APPENDIX A - THE SCHEMA DATA STRUCTURES

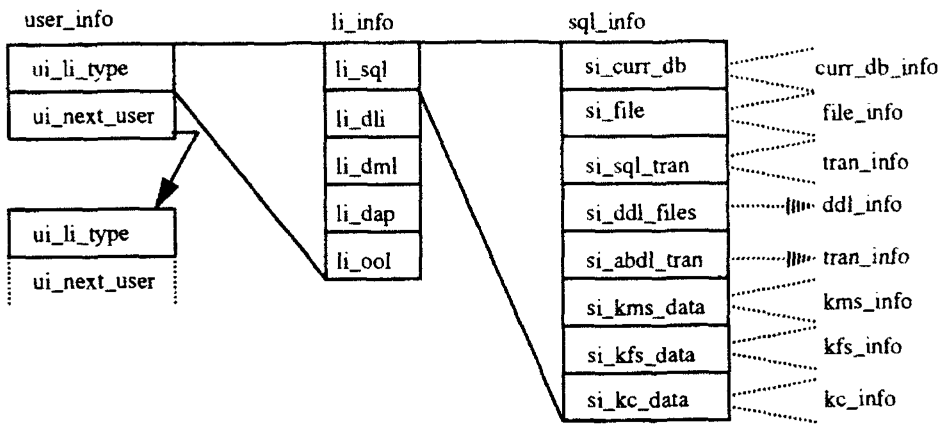
Relational Schema



Object-Oriented Schema



APPENDIX B - THᴱ USER DATA STRUCTURES



APPENDIX C - THE SCHEMA TRANSFORMER

```
R_PROCESS_OLD()
/* This proc accomplishes the following: */
/* (1) determines if the database name already exists,  */
/*    as a Relational model. If not, other models are  */
/*    checked, and if found, the schema is converted  */
/*    to a relational schema.                 */
/* (2) determines the user input mode (file/terminal),  */
/* (3) reads the user input and forwards it to the parser */

{
  char  cat_name[DBNLength];
  int   found, more_input;        /* boolean flags */
  int   num;
  int   i;
  struct rel_dbid_node    *rdb_list_ptr,       /* ptrs to the current     */
                          *temp_rdb_list_ptr;  /* database catalog        */
  struct rel_db_list_node *rdln_ptr;       /* pointer to current database list node */
 struct ddl_info        *ddl_info_alloc();  /* template and descriptor */
                                          /* file structure       */              /
 struct ocls_node *cls_ptr;

  /* create the template and descriptor structure if it doesn't already exist*/
  if (sql_info_ptr -> si_ddl_files == NULL)
    sql_info_ptr -> si_ddl_files = ddl_info_alloc();

 /* prompt user for name of existing database */
 printf ("[7;7m\nEnter name of database ---->[0;0m ");
 readstr (stdin, sql_info_ptr->si_curr_db.cdi_dbname);
 to_caps (sql_info_ptr->si_curr_db.cdi_dbname);
 found = FALSE;
 rdb_list_ptr = dbs_rel_head_ptr.dn_rel;
 temp_rdb_list_ptr = rdb_list_ptr;
 while (found == FALSE)
  {
   /* determine if database name does exist   */
   /* by traversing list of relational schemas */
   if ((dbs_rel_head_ptr.dn_rel) && (strcmp(sql_info_ptr->
       si_curr_db.cdi_dbname,rdb_list_ptr->rdn_name)== 0))
     {
      found = TRUE;
      sql_info_ptr->si_curr_db.cdi_db.dn_rel = rdb_list_ptr;
```

```
        sql_info_ptr->si_curr_db.cdi_dbtype = REL;
        strcpy(sql_info_ptr->si_ddl_files->ddli_temp.fi_fname,R-
TEMPFname);            strcpy(sql_info_ptr->si_ddl_files->ddli_desc.fi_fname,R-
DESCFname);           } /* end if */
     else /* found == false */
       {
        if (rdb_list_ptr)
          {
          temp_rdb_list_ptr = rdb_list_ptr;          /* save temp ptr to curr db */
          rdb_list_ptr = rdb_list_ptr->rdn_next_db; /* get next rel db */
          }
        /* db name is not a rel db so end of list('NULL') is reached */
        if (rdb_list_ptr == NULL)
          {
           /*check if db name is defined in the db list */
           rdln_ptr = db_list_head_ptr;
           while ((rdln_ptr != NULL)&&(strcmp(rdln_ptr->rdln_name,
                 sql_info_ptr->si_curr_db.cdi_dbname)))
             {

             rdln_ptr = rdln_ptr->rdln_next_db;
             }

           if (rdln_ptr)
             {
             r_load_catalog(rdln_ptr->rdln_name);
             found = TRUE;
             sql_info_ptr->si_curr_db.cdi_db.dn_rel =
                dbs_rel_head_ptr.dn_rel;
             sql_info_ptr->si_curr_db.cdi_dbtype = REL;
             strcpy(sql_info_ptr->si_ddl_files->ddli_temp.fi_fname,
                  RTEMPFname);
             strcpy(sql_info_ptr->si_ddl_files->ddli_desc.fi_fname,
                  RDESCFname);
             }
           else
             {

             /* check if db name is defined in another model */
              rdb_list_ptr = temp_rdb_list_ptr; /* reset to last db */
             check_alternate_models (&found, rdb_list_ptr);
             /* if not, an error has been made, so re-enter name */
             if (found == FALSE)
```

```
                {
                printf ("\nError - db name does not exist\n");
                printf ("[7;7mPlease reenter valid db name ---->[0;0m ");
                readstr (stdin, sql_info_ptr->si_curr_db.cdi_dbname);
          to_caps (sql_info_ptr->si_curr_db.cdi_dbname);
                rdb_list_ptr = dbs_rel_head_ptr.dn_rel;
                } /* end if found == false  */
               }
             } /* end if rdb_list_ptr == null */

         } /* end else */
      } /* end while */
   .
   .
   .
} /* end r_process_old */
```

```
CHECK_ALTERNATE_MODELS(found, rdb_list_ptr)
/* this routine calls other subroutines that check the Object-oriented,
  Network, Hierarchical, and Functional schemas for the desired
  database name. If found, the schema is translated to a corresponding
  Relational schema and prepared for processing. */

 int *found;
 struct rel_dbid_node *rdb_list_ptr; /* ptr to the current rel database */

 {
 struct hie_dbid_node *hdb_list_ptr; /* ptr to the current hie database */
 struct hie_dbid_node *traverse_dli_schema();

 struct obj_dbid_node *odb_list_ptr; /* ptr to the current obj database */
 struct obj_dbid_node *traverse_ool_schema();


 /*first check the hierarchical databases*/
 hdb_list_ptr = NULL;
 hdb_list_ptr = traverse_dli_schema();
 if (hdb_list_ptr != NULL)
   {
   .
   .
   .
   }

/*if db not found in hierarchical, check object-oriented*/
if (found == FALSE)
 odb_list_ptr = NULL;
 odb_list_ptr = traverse_ool_schema();
 if (odb_list_ptr != NULL)
   {
   *found = TRUE;
   sql_info_ptr->si_curr_db.cdi_dbtype = OBJ;
   translate_obj_to_rel(rdb_list_ptr, odb_list_ptr);
   strcpy(ZTEMPFname, add_path(sql_info_ptr->si_curr_db.cdi_dbname));
   strcat(ZTEMPFname, ".t");
   strcpy(sql_info_ptr->si_ddl_files->ddli_temp.fi_fname, ZTEMPFname);
   strcpy(ZDESCFname, add_path(sql_info_ptr->si_curr_db.cdi_dbname));
   strcat(ZDESCFname, ".d");
   strcpy(sql_info_ptr->si_ddl_files->ddli_desc.fi_fname, ZDESCFname);
  /* initialized the data base. */
  sql_info_ptr->si_operation = CreateDB;

  r_Kernel_Controller();

  strcpy(cuser_obj_ptr->ui_li_type.li_ool.oi_curr_db.cdi_dbname,
       sql_info_ptr->si_curr_db.cdi_dbname);
  cuser_obj_ptr->ui_li_type.li_ool.oi_operation = ExecRetReq;
  }
if (found == FALSE)
  {
  /* stub for future implementation of network model */
  }
if (found == FALSE)
  {
  /* stub for future implementation of functional model */
  }
} /* end check_alternate_models */
```

```
static struct obj_dbid_node *TRAVERSE_OOL_SCHEMA()
  /* This proc accomplishes the following:              */
  /* (1) determines if the database name already exists, as */
  /*     an object-oriented schema.                    */


  {
  int       obj_found,
            end_of_list;                   /* boolean flags */
  struct  obj_dbid_node *temp_odb_list_ptr; /* ptr to the current
                                                object-oriented database */
  FILE   *obj_dblist_fd;
  char    obj_name[DBNLength + 1];


  temp_odb_list_ptr = dbs_obj_head_ptr.dn_obj;
  obj_found = FALSE;
  end_of_list = FALSE;
  if (temp_odb_list_ptr == NULL)
    end_of_list = TRUE;
  while (obj_found == FALSE && end_of_list == FALSE)
    {
    /* determine if database name does exist     */
    /* by traversing list of object-oriented schemas */
    if (!strcmp(sql_info_ptr->si_curr_db.cdi_dbname,
            temp_odb_list_ptr->odn_name))
      obj_found = TRUE;
    else
      {
      temp_odb_list_ptr = temp_odb_list_ptr->odn_next_db;
      if (temp_odb_list_ptr == NULL)
        {
        end_of_list = TRUE;
        } /* end if */
      } /* end else */
    } /* end while */

        --
if (!obj_found)
 {
 strcpy(ODBCat,".");
 strcat(ODBCat, sql_info_ptr->si_curr_db.cdi_dbname);
 strcat(ODBCat, ".cat");


 if (obj_dblist_fd = fopen(ODBCat, "r"))
  {
  obj_found = TRUE;
  fclose(obj_dblist_fd);
  o_load_catalog(sql_info_ptr->si_curr_db.cdi_dbname);
  temp_odb_list_ptr = dbs_obj_head_ptr.dn_obj;
  }
 }
return (temp_odb_list_ptr);
} /* end traverse_ool_schema */
```

```
TRANSLATE_OBJ_TO_REL(rdb_list_ptr, odb_list_ptr)
/* this routine converts the object-oriented schema to a relational schema*/
 struct rel_dbid_node *rdb_list_ptr; /* ptr to the current rel database */
 struct obj_dbid_node *odb_list_ptr; /* ptr to the current obj database */

 {
 struct  rel_dbid_node   *new_rdb_ptr;              /* ptrs to database nodes */
 struct  rel_node        *new_rel_ptr, *rel_ptr;  /* ptrs to relation nodes */
 struct  rattr_node      *new_rattr_ptr, *rat_ptr; /* ptrs to attribute nodes */
 struct  rel_dbid_node   *mk_rel_dbid_node();
 struct  rel_node        *mk_rel_node();
 struct  rattr_node      *mk_rattr_node();
 struct obj_dbid_node    *odb_ptr;
 struct ocls_node        *cls_ptr, *supcls_ptr;
 struct o_supcls_node    *first_supcls_ptr;
 struct oattr_node       *oattr_ptr;


 /* head of current ool db schema */
 /* the new rel database node is allocated and filled here with
    information from the ool database node*/
 odb_ptr = odb_list_ptr;
 new_rdb_ptr = mk_rel_dbid_node();
 strcpy(new_rdb_ptr->rdn_name,odb_ptr->odn_name);
 new_rdb_ptr->rdn_num_rel  = odb_ptr->odn_num_cls;
 new_rdb_ptr->rdn_num_view  = 0;
 new_rdb_ptr->rdn_first_rel = NULL;
 new_rdb_ptr->rdn_curr_rel  = NULL;
 new_rdb_ptr->rdn_next_db   = NULL;
 new_rdb_ptr->rdn_dbtype    = OBJ;  /* identify db as object-oriented */
 if (dbs_rel_head_ptr.dn_rel)
  rdb_list_ptr->rdn_next_db = new_rdb_ptr; /* connect to rel db list */
 else
  dbs_rel_head_ptr.dn_rel = new_rdb_ptr;
 sql_info_ptr->si_curr_db.cdi_db.dn_rel = new_rdb_ptr;

 cls_ptr = odb_ptr->odn_first_cls;
 while (cls_ptr)
   {
   /* the relation nodes are allocated and filled here */
   new_rel_ptr = mk_rel_node();
   strcpy(new_rel_ptr->rn_name,cls_ptr->ocn_name);
    new_rel_ptr->rn_num_attr  = cls_ptr->ocn_num_attr;
```

```
new_rel_ptr->rn_first_attr = NULL;
new_rel_ptr->rn_curr_attr  = NULL;
new_rel_ptr->rn_next_rel   = NULL;
new_rel_ptr->rn_type       = 'T';
if (cls_ptr == odb_ptr->odn_first_cls)
  {
  /* special case of first relation */
   new_rdb_ptr->rdn_first_rel = new_rel_ptr;
   rel_ptr = new_rel_ptr;
   }
else
  {
  rel_ptr->rn_next_rel = new_rel_ptr;
  rel_ptr = new_rel_ptr;
  }

oattr_ptr = cls_ptr->ocn_first_attr;
while (oattr_ptr)
  {
  /* the attribute nodes are allocated and filled here */
  new_rattr_ptr = mk_rattr_node();
  strcpy(new_rattr_ptr->ran_name,oattr_ptr->oan_name);

  /*if it is a component class attribute, type is integer;
    else it is the same as for object-oriented*/
  if (!strcmp(oattr_ptr->oan_type, "CHAR"))
    {
    new_rattr_ptr->ran_type   = 's';
    new_rattr_ptr->ran_length = RNLength;
    }
  else if (!strcmp(oattr_ptr->oan_type, "FLOAT"))
    {
    new_rattr_ptr->ran_type   = 'f';
    new_rattr_ptr->ran_length = 3;
    }
 else
    {
    new_rattr_ptr->ran_type   = 'i';
    new_rattr_ptr->ran_length = 4;
    }
 new_rattr_ptr->ran_key_flag  = FALSE;
 new_rattr_ptr->ran_next_attr = NULL;
 if (oattr_ptr == cls_ptr->ocn_first_attr)
```

```
    {
    /* special case of first attribute */
    rel_ptr->rn_first_attr = new_rattr_ptr;
    rat_ptr = new_rattr_ptr;
    }
  else
    {
    rat_ptr->ran_next_attr = new_rattr_ptr;
    rat_ptr = new_rattr_ptr;
    }

  oattr_ptr = oattr_ptr->oan_next_attr;
  } /* end attr loop */
 rel_ptr->rn_curr_attr = rat_ptr;
 if (cls_ptr->ocn_supcls)
  {
  rel_ptr->rn_num_attr = rel_ptr->rn_num_attr + 1;
  supcls_ptr      = cls_ptr->ocn_first_supcls->osn_supcls;
 /*loop until reaching the left root of class; assign
    object id of this root supclass to relation*/
   while (supcls_ptr)
      {
      /*if this class doesn't have a superclass, it is the root*/
       if (!supcls_ptr->ocn_supcls)
         {
         new_rattr_ptr = mk_rattr_node();
         strcpy(new_rattr_ptr->ran_name, "OBJECTID");
         new_rattr_ptr->ran_type   = 'i';
         new_rattr_ptr->ran_length =  3;
         new_rattr_ptr->ran_key_flag  = FALSE;
         }
       supcls_ptr = supcls_ptr->ocn_first_supcls->osn_supcls;
       }/* end while supcls_ptr */
     new_rattr_ptr->ran_next_attr = rel_ptr->rn_first_attr;
     rel_ptr->rn_first_attr       = new_rattr_ptr;
     }/* end if supcls */
   cls_ptr              = cls_ptr->ocn_next_cls;
   new_rdb_ptr->rdn_curr_rel = rel_ptr;
   } /* end while cls_ptr   */
 } /* end translate_obj_to_rel() */
```

APPENDIX D - INSERT TRANSACTIONS

```
QUERIES_TO KMS()
/* This ro[illegible] ses the queries to be listed on the screen. */
/* The selection menu is then displayed allowing any of the    */
/* queries to be executeed*/

 {

 int proceed;   /* boolean flag */
 int num;


 num = 0;
 list_queries();
 proceed        = TRUE;
 while (proceed == TRUE)
  {
   printf ("\nPick the number or letter of the action desired\n");
   printf ("\t(num) - execute one of the preceding queries\n");
   printf ("\t(d)   - redisplay the file of queries\n");
   printf ("\t(x)   - return to the previous menu\n");
   sql_info_ptr->si_answer = get_ans(&num);

   swi[illegible] _ptr->si_answer)
   {
   case 'n' : /* execute one of the queries */
       if (num > 0 && num <= r_tran_info_ptr->ti_no_req)

           find_query (num);
              is is the default value for si_operation */
            if not a retrieve request, this value is reset */
           /* in sql_kernel_mapping_system */
           sql_info_ptr->si_operation = ExecRetReq;
                ernel_mapping_system();
                ._info_ptr->si_operation != ExecNoReq)

             if (sql_info_ptr->si_error == NOErr)
               if ((sql_ptr->si_curr_db.cdi_dbtype == OBJ) &&
                  (sql_info_ptr->si_operation == ExecInsReq))
                 insert_to_object_oriented();
                else
                   r_Kernel_Controller();

             else
                sql_info_ptr->si_error = NOErr;
           } /* end if */
      else
        {
        printf ("\nError - the query for the number you ");
        printf ("selected does not exist\n");
        printf ("Please pick again\n");
        }/* end else */

             break;
    case 'd' : /* redisplay queries */
      list_queries();
      break;
    case 'x' : /* exit to mode menu */
      proceed = FALSE;
             r_tran_info_ptr->ti_no_req = 0;
      break;
    default : /* user did not select a valid choice from the menu */
      printf ("\nError - invalid option selected\n");
      printf ("Please pick again\n");
      break;
     } /* end switch */
    } /* end while */
  } /* end queries_to_KMS */
```

```
INSERT_TO_OBJECT_ORIENTED()
/*called by queries_to_KMS for object-oriented cross-model inserts.
This procedure sends in an array of char pointers to get_obj_inserts,
which traverses the schema and prompts the user to enter appropriate
superclasses in order to maintain the object-oriented base model.
Once all of the INSERTs are built, they are sent to the KC one at a
time in a loop.*/

 {
 static char *insert_ptrs[NUMCLasses]; /*pointers to ABDL INSERTs*/
 int         i,
             begin_inserts;      /*used to flag the original insert*/


 begin_inserts = TRUE;
 for (i = 0; i < NUMCLasses; i++)
  insert_ptrs[i] = NULL;
 get_obj_inserts(begin_inserts, insert_ptrs);

 /*If user has entered all of the superclasses correctly, the INSERTs
 are sent to the KC below, one-by-one.*/
 if (sql_info_ptr->si_error == NOErr)
  {
  for (i = 0; insert_ptrs[i]; i++)
   {
   sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req = insert_ptrs[i];
   r_Kernel_Controller();
   }
  }
 else
  sql_info_ptr->si_error = NOErr;
}/*end insert_to_object_oriented*/
```

```
GET_OBJ_INSERTS(start, insert_ptrs)
/*this procedure checks to see if the class to which the INSERT relation
corresponds is a subclass and, if so, prompts the user to enter the relations
corresponding to its superclasses. The relations(classes) must
be entered one at a time from the bottom subclass up the hierarchy and
left to right at the screen prompts. They can be entered from the file or
from the terminal. As each is entered, the procedure recurses until a
mistake is made or all of the superclass relations have been correctly
entered. Calls obj_inserts_to_KMS for user inerface.*/


 int                   start;
 char                  *insert_ptrs[NUMCLasses]; /*ABDL INSERTs*/

 {
 int                   i,
                       char_position,
                       req_length,
                       original_insert,
                       object_exists,
                       check_insert_objids();
 struct ocls_node      *cls_ptr,
                       *kms_cls_ptr,
                       *find_class_in_obj_trans();
 struct o_supcls_node  *supcls_ptr;
 char                  *var_str_alloc();


 if (sql_info_ptr->si_operation != ExecInsReq)
  sql_info_ptr->si_error = ExecNoReq;


 else
  {
  /*find the next empty cell and allocate memory to accept the current
  ABDL INSERT*/
  for (i = 0; insert_ptrs[i]; i++)
   ;
  req_length =
      strlen(sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req);
  insert_ptrs[i] = var_str_alloc(req_length);
  strcpy(insert_ptrs[i],
      sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req);
```

```
char_position   = 17;  /*position of the first letter of the
                         relation name in the ABDL INSERT string*/
original_insert  = start;/*boolean*/

/*retrieve class name of insert*/
cls_ptr = find_class_in_obj_trans(char_position);

if (!cls_ptr)
  sql_info_ptr->si_error = ExecNoReq;

/*the statements below increment the char_postion variable to the
position of the value of the objectid. If a value is returned by
the function check_insert_objids, an object exists already and the
INSERT is cancelled.*/
char_position += strlen(cls_ptr->ocn_name);
char_position += 14;
if (object_exists = check_insert_objids(cls_ptr, char_position))
 {
 printf("\nERROR - An instance of this class already exists.");
 sql_info_ptr->si_error = ExecNoReq;
 }

/*If the class has superclasses, this procedure calls itself until
all have been visited.*/
if ((sql_info_ptr->si_error == NOErr) && cls_ptr->ocn_supcls)
 {
 if (original_insert)
  {
  start           = FALSE;
  cls_ptr->ocn_visited = TRUE;
  system("clear");
printf("\n\n\n\n\nIn order to maintain the integrity of the object-oriented base");
printf("\nmodel, INSERTs of all relations which correspond to superclasses");
printf("\nin the object-oriented hierarchy of the original INSERT are");
printf("\nnecessary. INSERT each superclass relation when prompted. If");
printf("\nentering from a file, selections must offer appropriate INSERTs.");
printf("\nIf any INSERT is in error or any superclass INSERTs are lacking,");
printf("\nall preceding INSERTs will be cancelled. <CR> to continue.");
  getchar();
  system("clear");
}
```

```
    supcls_ptr = cls_ptr->ocn_first_supcls;
    while(supcls_ptr && (sql_info_ptr->si_error == NOErr))
     {
     if (!supcls_ptr->osn_supcls->ocn_visited)
       {
printf("\n\n\n\n\nINSERT a tuple of relation %s now, having the same OBJECTID
\n",                                                    supcls_ptr->osn_name);
printf("attribute as the INSERT just attempted. If such a %s tuple \n",
                                                      supcls_ptr->osn_name);
printf("already exists (i.e. has the same OBJECTID) you must start \n");
printf("INSERTs over, using a unique OBJECTID. <CR> to quit or continue.\n");
       getchar();
       system("clear");
       supcls_ptr->osn_supcls->ocn_visited = TRUE;
       obj_insert_to_KMS();
       if (sql_info_ptr->si_error == NOErr)
         {
         char_position = 17;
         kms_cls_ptr = find_class_in_obj_trans(char_position);
         if (!strcmp(kms_cls_ptr, supcls_ptr->osn_name))
           get_obj_inserts(start, insert_ptrs);
         else
           {
           printf("\nERROR - %s is not a valid superclass.",
           kms_cls_ptr->ocn_name);
           sql_info_ptr->si_error = ExecNoReq;
           }
       }/*end if NOErr*/
     }/*end if not visited*/
     supcls_ptr = supcls_ptr->osn_next_supcls;

   } /*end while supcls*/
  } /*end if cls_ptr->ocn_supcls*/
 }/*end if ExInsReq*/
 if (sql_info_ptr->si_error != NOErr)
   printf("\n%s INSERT cancelled\n",cls_ptr->ocn_name);

  /*sets all of the visited flags in the schema data structure back
  to FALSE.*/
  if (original_insert)
    reset_obj_visited(cls_ptr);
}/*end get_object_inserts*/
```

```
int CHECK_INSERT_OBJIDS(cls_ptr, posit)
/*this procedure gets a pointer to a class in the object-oriented schema
and the position of the value of the objectid in the INSERT request string.
It builds an ABDL retrieval request and sends it in. If a response is
returned, an object already exists and the INSERT is cancelled.*/

 int              posit;  /*objectid position*/
 struct ocls_node *cls_ptr; /*insert relation/class*/
 {
 int              obj_id_posit;
                  *empty;
 char             *temp_string_ptr,
                  retrieve_ptr[InputCols];

 empty = FALSE;
 temp_string_ptr = sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req;

 /*build the ABDL retrieve request*/
 strcpy(retrieve_ptr, "[ RETRIEVE  ((TEMP = ");
 strcat(retrieve_ptr, cls_ptr->ocn_name);
 strcat(retrieve_ptr, ") and (OBJECTID = ");
 obj_id_posit = strlen(retrieve_ptr);
 do
   retrieve_ptr[obj_id_posit++] = temp_string_ptr[posit];
 while (temp_string_ptr[++posit] != '>');
 retrieve_ptr[obj_id_posit] ='\0';
 strcat(retrieve_ptr, "))(OBJECTID) ]");
 sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req = retrieve_ptr;
 fix_up_objectid_ABDL_req();

 /*send it in*/
 TI_S$TrafUnit(sql_ptr->si_curr_db.cdi_dbname,
          sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req);
 get_response(&empty);
 /*re-attach the original INSERT request*/
 sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req = temp_string_ptr;

 /*'?' means that there were no instances of this object in the database*/
 if (sql_ptr->si_kfs_data.kfsi_rel.kri_response[1] != '?')
   return 1;
 else
   return 0;
}/*end check_insert_objectids*/
```

```
OBJ_INSERT_TO_KMS()
/*called by the recursive procedure, insert_to_object_oriented. User*/
/*is provided the same list of queries entered into queries_to_KMS*/
/* This routine causes the queries to be listed on the screen. */
/* The selection menu is then displayed allowing any of the    */
/* queries to be executeed*/

 {
 int proceed;   /* boolean flag */
 int num;

 num = 0;
 list_queries();
 proceed       = TRUE;
 while (proceed == TRUE)
  {
   printf ("\nPick the letter or number of the action desired\n");
   printf ("\t(num) - execute one of the preceding INSERT queries\n");
   printf ("\t(d)   - redisplay the file of queries\n");
   printf ("\t(x)   - cancel INSERT.\n");
   sql_info_ptr->si_answer = get_ans(&num);

   switch (sql_info_ptr->si_answer)
     {
     case 'n' : /* execute one of the queries */
      if (num > 0 && num <= r_tran_info_ptr->ti_no_req)
          {
          find_query (num);
          /* This is the default value for si_operation */
          /* If not a retrieve request, this value is reset */
          /* in sql_kernel_mapping_system */
          sql_info_ptr->si_operation = ExecRetReq;
          sql_kernel_mapping_system();
          proceed = FALSE;
          } /* end if */
       else
          {
          printf ("\nError - the query for the number you ");
          printf ("selected does not exist\n");
          printf ("Please pick again\n");
          } /* end else */
       break;
```

```
    case 'd' : /* redisplay queries */
      list_queries();
      break;
    case 'x' : /* exit INSERTS */
      proceed = FALSE;
      sql_info_ptr->si_error = ExecNoReq;
      break;
    default : /* user did not select a valid choice from the menu */
      printf ("\nError - invalid option selected\n");
      printf ("Please pick again\n");
      break;
      } /* end switch */
    } /* end while */
} /* end object_insert_to_KMS */
```

```
RESET_OBJ_VISITED(cls_ptr)
/*This is a utility procedure that resets all of the visited flags
in the object-oriented ocls_node data structure to FALSE.*/

 struct ocls_node        *cls_ptr;

 {
 struct o_supcls_node *supcls_ptr;

 cls_ptr->ocn_visited = FALSE;
 if (cls_ptr->ocn_supcls)
   {
   supcls_ptr = cls_ptr->ocn_first_supcls;
   while (supcls_ptr)
     {
     reset_obj_visited(supcls_ptr->osn_supcls);
     supcls_ptr = supcls_ptr->osn_next_supcls;
     }
   }
 }/*end reset_obj_visited*/
```

```
struct ocls_node *FIND_CLASS_IN_OBJ_TRANS(start_posit)
/* This procedure takes an int, which indicates the position of the first
letter in the relation name of a query, extracts the relation name, and
returns a pointer to the class node to which it corresponds in the
object-oriented schema.*/

 int                start_posit;

 {
 int                i,
                    not_found;
 char               cls_name[RNLength + 1],
                    *temp_string_ptr;
 struct ocls_node  *cls_ptr;

 for (i = 0; i < (RNLength + 1); i++)
   cls_name[i] = '\0';

 /*copy the relation name into the temp_string_ptr for comparison*/
 temp_string_ptr = sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req;
 cls_ptr        = dbs_obj_head_ptr.dn_obj->odn_first_cls;
 while ((temp_string_ptr[i] != ')') && (temp_string_ptr[i] != '>'))
  {
  cls_name[i - start_posit] = temp_string_ptr[i];
  i++;
  }

 /*search for the class in the object-oriented schema*/
 while (cls_ptr && not_found)
  {
  if (!strcmp(cls_name, cls_ptr->ocn_name))
    not_found = FALSE;
  else
    cls_ptr = cls_ptr->ocn_next_cls;
  }
if (!cls_ptr)
 printf("ERROR - Relation %s is not in the schema", cls_name);

 return cls_ptr;
 }/*end find_class_in_obj_trans*/
```

APPENDIX E- DELETE TRANSACTIONS

```
#include <stdio.h>
#include <licommdata.h>
#include <ool_sql.h>
#include <sql.h>
#include <sql_kcdcl.h>
#include <flags.def>
#include <ctype.h>
#include <sql_kc.h>
#include <dli.h>

R_KERNEL_CONTROLLER()

 /*  This procedure accomplishes the following:                    */
 /*  (1) Checks si_operation to determine whether we are creating a      */
 /*      database or querying the database or if there are errors.     */
 /*  (2) Depending on the si_operation the corresponding              */
 /*      procedure is called.                         */


{
int     groupby = FALSE;

 sql_ptr = &(cuser_rel_ptr->ui_li_type.li_sql);/* Initialize pointer */
 kc_ptr  = &(sql_ptr->si_kc_data.kci_r_kc);   /* Initialize pointer */
 sql_ptr->si_subreq_stat = LASTSUBREQ;

  /*  look at si_operation to determine what action to take    */

   switch (sql_ptr->si_operation)
    {
     case CreateDB:   /*case where we are creating a database*/
        r_load_tables();
        break;

     case ExecRetReq: /*case where we are executing a regular or */
                 /*nested select */
        select_requests_handler(groupby);
        break;

     case ExecRetCReq: /* any other type of select */
        sql_ptr->si_abdl_tran->ti_no_req--; /* decrement */
```

```
            rest_requests_handler(groupby);
            break;

        case ExecDelReq: /* a delete request */
            sql_ptr->si_abdl_tran->ti_no_req--; /* decrement */
            if (sql_ptr->si_curr_db.cdi_dbtype == OBJ)
              object_oriented_delete(groupby);
            else
              rest_requests_handler(groupby);
            break;

        case ExecInsReq: /* an insert request */
            sql_ptr->si_abdl_tran->ti_no_req--; /* decrement */
            insert_request_handler(groupby);
            break;

        case ExecUpdReq: /* an update request */
            sql_ptr->si_abdl_tran->ti_no_req--;
            rest_requests_handler(groupby);
            break;

        case ExecGrpReq: /* an update request */
            groupby = TRUE;
            group_requests_handler(groupby);
            break;

        default:
            break;
      }  /* end switch */
} /* end procedure r_Kernel_Controller */
```

```
OBJECT_ORIENTED_DELETE(groupby)
/*Called by r_kernel_controller. This procedure calls two others: one to
retrieve all of the objectids in the delete request, the other to traverse the
object-oriented lattice and execute deletes for each instance in all of the
super and subclasses .*/

 int groupby;

 {
 int                delete_and = 20;
 int                delete_all = 19;
 int                type_of_delete;
 struct ocls_node  *cls_ptr,
                    *find_class_in_obj_trans();
 char               *temp_string_ptr,
                    *objectids,
                    *get_objectids(); /*ids of instances to be deleted*/

 temp_string_ptr =  sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req;

 /*The type of delete request is determined by the positions of the
   parenthesis in the string. This is needed to retrieve the class name.*/
 if (temp_string_ptr[12] == '(')
  type_of_delete = delete_and;
 else
  type_of_delete = delete_all;

 cls_ptr = find_class_in_obj_trans(type_of_delete);

 if (cls_ptr)
  {
  cls_ptr->ocn_visited = TRUE;
  objectids = get_objectids();
  search_for_sub_and_superclasses_and_execute_deletes
                              (cls_ptr, objectids, groupby);
  }
 reset_obj_visited(cls_ptr); /*reset to FALSE the visited flag in the
                                 object-oriented schema data structure.*/
}/*end object_oriented_delete*/
```

```
char *GET_OBJECTIDS()
/*This procedure is called by object_oriented_deletes. It retrieves all of
the objectids of instances which are to be deleted. These are then used to
delete the proper instances of super and subclasses in another procedure*/

 {
 int    i,
        r,
        *empty,
        last_paren;
 char   retrieve_ptr[InputCols],
        *temp_string_ptr,
        *response_ptr,
        *object_ids,
        *var_str_alloc();


 empty = FALSE;
 last_paren  = FALSE;

 /*copy the string into the array and build and execute a retrieve
   request to obtain all of the objectids in the delete request.*/
 temp_string_ptr = sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req;
 strcpy(retrieve_ptr, "[ RETRIEVE  (");
 r = 13;
 i = 12;
 while (!last_paren)
   {
   retrieve_ptr[r] = temp_string_ptr[i];
   if (temp_string_ptr[i] == ')')
     {
     if (temp_string_ptr[i + 1] == ')')
       {
       retrieve_ptr[r + 1] = ')';
       retrieve_ptr[r + 2] = '\0';
       last_paren        = TRUE;
       }
     else if ((temp_string_ptr[i + 1] == ' ') &&
          (temp_string_ptr[i + 2] == ']'))
       {
       retrieve_ptr[r + 1] = '\0';
       last_paren        = TRUE;
       }
 }
 i++;
 r++;
}
strcat(retrieve_ptr, "(OBJECTID) ]");

sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req = retrieve_ptr;
fix_up_objectid_ABDL_req();

TI_S$TrafUnit(sql_ptr->si_curr_db.cdi_dbname,
          sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req);
get_response(&empty);

/*the objectids of all of the instances to be deleted were retrieved above
  and attached as a char* in the global user structure. This string is
  copied to the char array, objectids.*/

response_ptr = sql_ptr->si_kfs_data.kfsi_rel.kri_response;
object_ids   = var_str_alloc(sql_ptr->si_kfs_data.kfsi_rel.kri_res_len);
i = 0;
do
  object_ids[i] = response_ptr[i];
while (response_ptr[i++] != '?');

return object_ids;

}/*end get_objectids*/
```

```
SEARCH_FOR_SUB_AND_SUPERCLASSES_AND_EXECUTE_DELETES
                                        (class_ptr, obj_ids, groupby)
/*called by object_oriented_delete. Recursively traverses the object-oriented
schema and calls execute_deletes() for each of the super and subclasses.*/

 struct ocls_node      *class_ptr;
 int                    groupby;
 char                   *obj_ids;

 {
 int                    no_cycle;
 struct o_supcls_node *supcls_ptr;
 struct o_subcls_node *subcls_ptr;

 /*get all of the superclasses*/
 supcls_ptr = class_ptr->ocn_first_supcls;
 while(supcls_ptr)
  {
  if (!supcls_ptr->osn_supcls->ocn_visited)
   {
   supcls_ptr->osn_supcls->ocn_visited = TRUE;
   search_for_sub_and_superclasses_and_execute_deletes
                                (supcls_ptr->osn_supcls, obj_ids, groupby);
   }
  supcls_ptr = supcls_ptr->osn_next_supcls;
  }

 /*get all of the subclasses*/
 subcls_ptr = class_ptr->ocn_first_subcls;
 while(subcls_ptr)
  {
  if (!subcls_ptr->osn_subcls->ocn_visited)
   {
   subcls_ptr->osn_subcls->ocn_visited = TRUE;
   search_for_sub_and_superclasses_and_execute_deletes
                                (subcls_ptr->osn_subcls, obj_ids, groupby);
   }
  subcls_ptr = subcls_ptr->osn_next_subcls;
  }
 class_ptr->ocn_visited = FALSE;

 execute_object_oriented_deletes(obj_ids, class_ptr->ocn_name, groupby);
 }/*end search_for_sub_and_superclasses_and_execute_deletes*/
```

```
EXECUTE_OBJECT_ORIENTED_DELETES(ob_ids, class_name, groupby)
/*Called by search_for_sub_and_superclasses_and_execute_deletes(). Sends in
  the ABDL requests for execution. Makes a separate request for each object to be
 deleted. The objectids are stored as "attribute'\0'value'\0'attribute..."*/


 char  *ob_ids,
       class_name[RNLength + 1];
 int    groupby;

{
int   i,
      obj_id_place,
      t;
char  temp_string_ptr[InputCols];

i = 0;
while (ob_ids[i] != '?')
 {
 if (ob_ids[i] == '\0')
  {
  for (t = 0; t < InputCols; t++)
   temp_string_ptr[t] = '\0';/*initialize the array to nulls*/

  strcpy(temp_string_ptr, "[ DELETE  ( (TEMP = ");
  strcat(temp_string_ptr, class_name);
  strcat(temp_string_ptr, ") and (OBJECTID = ");
  obj_id_place = strlen(temp_string_ptr);

  /*copy in the object OBJECTID value*/
  do
    temp_string_ptr[obj_id_place++] = ob_ids[++i];
  while(ob_ids[i] != '\0');

  strcat(temp_string_ptr, ")) ]");
  sql_ptr->si_abdl_tran->ti_curr_req.ri_ab_req->ari_req = temp_string_ptr;

  /*send the DELETEs to the system for execution*/
  rest_requests_handler(groupby);
  }
 i++;
 }
}/*end execute_object_oriented_deletes*/
```

LIST OF REFERENCES

[ACM1989] *Object-Oriented Concepts, Databases, and Applications*, pp. 219-337, ACM Press, 1989.

[Banerjee1979] Banerjee, J. and Hsiao, D.K., "A Database Computer for Very Large Databases," *IEEE Transactions on Computers*, v. c-28, no. 6, pp. 414-429, June 1979.

[Benson1985] Benson, T. P. and Wentz, G. L., *The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[Bourgeois1993] Bourgeois, P., *The Implementation of the Multi-Model and Multi-Lingual User Interface*, Master's Thesis, Naval Postgraduate School, Monterey, California March 1993.

[Demurjian1987] DeDmurjian, S. A., *The Multi-Lingual Database System - a Paradigm and Test-Bed for the Investigation of Data-Model Transformations, Data-Language Translations and Data-Model Semantics*, U-M-I Dissertation Information Service, 1987.

[Elmasri1989] Elmasri, R. and Navathe S.B., *Fundamentals of Database Systems*, pp. 409-452, The Benjamin/Cummings Publishing Company, Inc.,1989.

[Emdi1985] Emdi, B., *The Implementation of a CODASYL-DML Interface for a Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[Hsiao1989] Hsiao, D. K. and Kamel, M. N., "Heterogeneous Databases: Proliferations, Issues, and Solutions," *IEEE Transactions on Knowledge and Data Engineering*, v. 1, no. 1, pp. 45-62, March 1989.

[Hsiao1991] Hsiao, D. K., "A Parallel, Scalable, Microprocessor-based Database Computer for Performance Gains and Capacity Growth," *IEEE MICRO*, pp. 44-60, December 1991.

[Hsiao1992] Hsiao, D. K., "Federated Databases and Systems: Part 1 - A Tutorial on Their Data Sharing," *VLDB Journal*, v. 1, pp. 127-179, 1992.

[Hughes1991] Hughes, J. G., *Object-Oriented Databases*, pp. 212-224, Prentice Hall International (UK) Ltd., 1991.

[KimA1990] Kim, W., "Defining Object Databases Anew," *Datamation*, pp. 33-36, 1 February 1990.

[KimB1990] Kim, W., "Object-Oriented Databases: Definition and Research Directions," IEEE Transactions on Knowledge and Data Engineering, v. 2, no. 3, pp. 327-341, September 1990.

[Mak1992] Mak, S. B., *The Design and Implementation of a Functional Interface for the Attribute-Based Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, March 1992.

[Moore1993] Moore, J. W. and Karlidere, T., *The Design and Implementation of an Object-Oriented Interface for the Multi-Model and Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1993.

[Rollins1984 Rolliins, R. E., *Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System*, Master's Thesis, Naval Pcstgraduate School, Monterey, California, June 1984.

[Su1979] Su, S. Y. W, and others, "Architectural Features and Implementation Techniques of the Multicell CASSM," *IEEE Transactions on Computers*, v. c-28, no. 6, pp. 430-445, June 1979.

[Zawis1987] Zawis, J. A., *Accessing Hierarchical Databases Via SQL Transactions in a Multi-Model Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1987.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
Ms. Doris Mlezco Code 9033 Naval Pacific Missile Test Center Point Mugu, CA 93042	1
LCDR Richard K. Johnston Naval Security Group Activity Naval Base Charleston, SC 29408-0008	1
Dr. David K. Hsiao Code CS/Hs Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1