NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ACCESSING A FUNCTIONAL DATABASE
VIA
CODASYL-DML TRANSACTIONS

by

Harry Coker, Jr.

June 1987

Thesis Advisor: D. K. Hsiao

unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School; 6b OFFICE SYMBOL (If applicable): 52	7a NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code): Monterey, California 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): ACCESSING A FUNCTIONAL DATABASE VIA CODASYL-DML TRANSACTIONS

12 PERSONAL AUTHOR(S): Coker, Harry Jr.

13a TYPE OF REPORT	13b TIME COVERED FROM ___ TO ___	14 DATE OF REPORT (Year Month Day)	15 PAGE COUNT
Master's Thesis		1987 June	74

16 SUPPLEMENTARY NOTATION

17 COSATI CODES: FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number): MLDS, Multi-lingual Database System; MBDS, Multi-Backend Database System; Functional Data Model; Network Data Model; CODASYL-DML; Daplex

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

Conventional approaches to the design and implementation of database systems have been based upon the premise of a single data model with its model-based data language, thus restricting the database system to transactions based solely on a specific model and written in a specific data language. This traditional approach has drastically hindered the widespread interaction of database systems based on various data models and languages. As an alternative to this traditional and less effective approach to database systems, the multi-lingual database system (MLDS) has evolved. MLDS has allowed the user to access and interact with numerous databases in various data models via their corresponding data languages.

This thesis implements a methodology for accessing and manipulating databases stored in a particular data model via transactions of a separate data model; specifically, a functional database is accessed via CODASYL-DML transactions. This interface is the initial move toward extending MLDS to a thoroughly Multi-Model Database System (MMDS).

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL: Prof. D. K. Hsiao	22b TELEPHONE (Include Area Code): (408) 646-2253	22c OFFICE SYMBOL: Code 52Hq

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

SECURITY CLASSIFICATION OF THIS PAGE
unclassified

Approved for public release; distribution is unlimited.

Accessing a Functional Database Via CODASYL-DML Transactions

by

Harry Coker, Jr.
Lieutenant , United States Navy
B.S., United States Naval Academy, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1987

ABSTRACT

Conventional approaches to the design and implementation of database systems have been based upon the premise of a single data model with its model-based data language, thus restricting the database system to transactions based solely on a specific model and written in a specific data language. This traditional approach has drastically hindered the widespread interaction of database systems based on various data models and languages. As an alternative to this traditional and less effective approach to database systems, the multi-lingual database system (MLDS) has evolved. MLDS has allowed the user to access and interact with numerous databases in various data models via their corresponding data languages.

This thesis implements a methodology for accessing and manipulating databases stored in a particular data model via transactions of a separate data model; specifically, a functional database is accessed via CODASYL-DML transactions. This interface is the initial move toward extending MLDS to a thoroughly Multi-Model Database System (MMDS).

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

I. INTRODUCTION 10
A. MOTIVATION 10
B. SYSTEM ORGANIZATION 11
1. The Multi-Lingual Database System 11
2. The Multi-Backend Database System 12
C. THESIS OVERVIEW 14

II. THE DATA MODELS 15
A. THE FUNCTIONAL DATA MODEL AND DAPLEX 15
1. The Data Model 15
2. The Data Language 15
B. THE NETWORK DATA MODEL AND CODASYL-DML 16
1. The Data Model 16
2. The Data Language 19
C. THE ATTRIBUTE-BASED DATA MODEL AND ABDL 20
1. The Data Model 20
2. The Data Language 21

III. DATABASE MAPPINGS 22
A. BACKGROUND MATERIAL 23
B. MAPPING THE FUNCTIONAL(DAPLEX) MODEL(LANGUAGE) TO THE NETWORK(CODASYL-DML) MODEL(LANGUAGE) 23
1. Available Strategies 23
2. The Selected Mapping Strategy 24
C. DATA-MODEL TRANSFORMATIONS REFERENCED IN THIS THESIS 24
1. The Functional to ABDM Mapping 25
2. Functional to Network Mapping 26

IV. THE DATA STRUCTURES 28
A. DATA SHARED BY ALL USERS 28
1. Data Shared by All Users of a Network Database 29
2. Data Shared by All Users of a Functional Database 32
B. DATA SPECIFIC TO EACH USER 38

V. FUNCTIONAL TO NETWORK TRANSFORMATION ALGORITHMS 41
A. ENTITY TYPES 42
B. ENTITY SUB-TYPES 47
C. NON-ENTITY TYPES 49
D. UNIQUENESS CONSTRAINTS 51
E. OVERLAPPING CONSTRAINTS 51
F. SET TYPES 52

VI. TRANSLATION OF CODASYL-DML STATEMENTS TO ABDL REQUESTS 54
A. OVERVIEW OF THE DESIGN 54
B. MAPPING CODASYL-DML FIND STATEMENTS 55
1. The FIND ANY Statement 55
2. The FIND CURRENT Statement 56
3. The FIND DUPLICATE WITHIN Statement 56
4. The Find FIRST/LAST/NEXT/PRIOR Statements 57
5. The FIND OWNER Statement 59
6. The FIND WITHIN CURRENT Statement 59
C. MAPPING CODASYL-DML GET STATEMENTS 60
1. The GET Statement 60
2. The GET record_type Statement 60
3. The GET item_1, ..., item_n Statement 60
D. MAPPING CODASYL-DML CONNECT STATEMENTS 60
1. Sets Representing an ISA Relationship 61
2. Sets Representing Daplex Functions 61
E. MAPPING CODASYL-DML DISCONNECT STATEMENTS 63
F. MAPPING CODASYL-DML MODIFY STATEMENT 65

G. MAPPING CODASYL-DML STORE STATEMENTS 65
H. MAPPING CODASYL-DML ERASE STATEMENTS 66
1. The ERASE Option 67
2. The ERASE ALL Option 68

VII. CONCLUSIONS 69
A. A REVIEW OF OUR WORK 69
B. FUTURE RESEARCH 70

LIST OF REFERENCES 71

INITIAL DISTRIBUTION LIST 73

LIST OF FIGURES

1.1 The Multi-Lingual Database System (MLDS) 12
1.2 Multiple Language Interfaces for KDS 13
1.3 MBDS Architecture 14
2.1 The University Database Schema 17
2.2 Graphical Representation of Univ Schema 19
2.3 Attribute-Based Data Model Record 21
3.1 MLDS Mapping of the Network and Functional Data Models 22
3.2 Direct Language Interface Approach 25
3.3 The AB(functional) University Database Schema 27
4.1 The dbid_node Data Structure 28
4.2 The net_dbid_node Data Structure 29
4.3 The nset_node Data Structure 30
4.4 The set_select_node Data Structure 30
4.5 The nrec_node Data Structure 31
4.6 The nattr_node Data Structure 31
4.7 The fun_dbid_node Data Structure 32
4.8 The ent_node Data Structure 33
4.9 The gsn_sub_node Data Structure 34
4.10 The ent_non_node Data Structure 35
4.11 The sub_non_node Data Structure 35
4.12 The der_non_node Data Structure 36
4.13 The overlap_node Data Structure 36
4.14 The function_node Data Structure 37
4.15 The ent_node_list Data Structure 37
4.16 The sub_node_list Data Structure 37
4.17 The ent_value Data Structure 38
4.18 The user_info Data Structure 38
4.19 The li_info Data Structure 39

4.20 The dml_info Data Structure 39
4.21 The dap_info Data Structure 40
5.1 The Functional Schema of the University Database Transformed to a Network Schema 43
5.2 Entity Type Declaration 46
5.3 A functional entity type and its network representation 48
5.4 Entity Subtype Declaration 49
5.5 A functional entity subtype and its network representation 50

I. INTRODUCTION

A. MOTIVATION

Traditionally database systems have been limited to a single data model along with its respective model-based data language. This conventional approach to Database Management System (DBMS) development has resulted in the evolution of a DBMS that has restricted the user to transactions on a single data model and its corresponding data language.

Ideally, an effective and practical DBMS should be able to access and interact with numerous databases based on various data models via their respective data languages. Thus, the motivation behind *Multi-Lingual Database System* (MLDS) is to have one DBMS that is able to support numerous databases that may be structured in various data models by executing transactions written in their model-based data languages [Ref. 1]. MLDS is a modern approach to DBMS development that is attacking the problems of the older, conventional, ***homogeneous*** database system designs that are currently in abundance. More precisely, MLDS allows the user to access a DBMS that is comprised of a hierarchical/DL/I interface, a relational/SQL interface, a network/CODASYL-DML interface, a functional/DAPLEX interface, and an attribute-based/ABDL interface; the system functions as if it were a ***heterogeneous*** collection of database systems.

The primary advantages to be gained from MLDS are (1) ***reusability*** of database transactions developed on existing systems, (2) more economical and efficient ***hardware upgrades*** by spreading the upgrade benefit to each of the data models rather than a single model, and (3) an ability to ***support a variety of databases*** built around any of the major data models.

Up to this point MLDS has permitted the user to access and interact with several databases in the five major data models via their corresponding data languages. This thesis implements a design methodology, [Ref. 2], for accessing and manipulating databases stored in a particular data model via transactions of a separate data model; specifically a *functional* database is accessed via *CODASYL-DML* transactions. This interface is the initial move toward extending the MLDS to a thoroughly *Multi-Model Database System* (MMDS).

B. SYSTEM ORGANIZATION

In order to meet the aforementioned capabilities, MLDS is supported by an underlying database system that is fast, efficient, and effective, therefore necessitating a powerful kernel data model and kernel data language, as well as a high-performance, high-capacity database system [Ref. 3: page 12].

The *kernel data model* and the *kernel data language* are the underlying model and language for MLDS. The attribute-based data model and the attribute-based data language were chosen as the kernel data model and the kernel data language for reasons that will be explicitly cited and analyzed in the following chapter. The *software multiple-backend approach* is used to provide the required high-performance and high-capacity underlying database system that MLDS requires. This system, known as the Multi-Backend Database System (MBDS), will be examined later in this chapter.

1. The Multi-Lingual Database System

The system structure of MLDS is depicted in Figure 1.1. The *language interface layer* (LIL) supports user interaction with the system via a *user-selected data model* (UDM) with transactions written in a corresponding *user data language* (UDL). The user's transaction is routed to the *kernel mapping subsystem* (KMS) by LIL, where KMS performs one of two possible tasks. It either transforms the UDM-database definition into an equivalent *kernel data model* (KDM) database definition; or it translates a UDL transaction into an equivalent *kernel-data-language* (KDL) transaction.

The first of the two possible tasks of KMS occurs if the user indicates that a new database is to be created. KMS forwards the KDM-database definition to the *kernel controller subsystem* (KCS), where the KDM-database definition is then sent to the *kernel database system* (KDS). Upon completion, the user is notified by LIL, via KDS and KCS, that the database definition has been processed and that the loading of the database may continue.

The second of the possible tasks of KMS occurs if the user chooses to process an existing database. KMS sends the KDL transaction to KCS, which in turn forwards the KDL transaction to KDS for execution. When KDS has finished executing the transactions, the results, in KDM format, are sent back to KCS, where they are routed to the *kernel formatting subsystem* (KFS). KFS reformats the results into UDM format and displays them , via LIL, to the user.

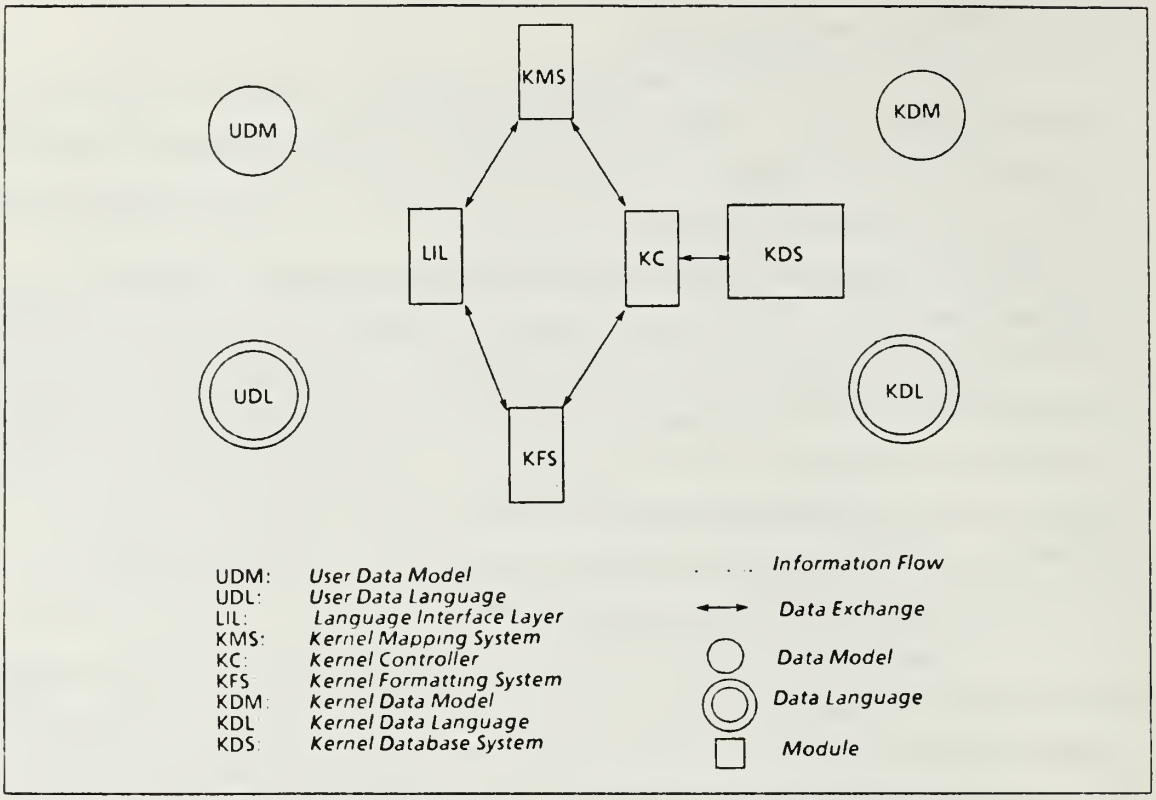

Figure 1.1 The Multi-Lingual Database System (MLDS).

LIL, KMS, KCS, and KFS make up a ***language interface*** of MLDS. Four language interfaces exist, one for each of the respective UDM/UDL combinations. This thesis modifies the network/CODASYL-DML language interface in order to allow the accessing and manipulation of a functional database via CODASYL-DML transactions. KDS, on the other hand, is a single and major component that is accessed by all of the languages interfaces, as shown in Figure 1.2.

2. The Multi-Backend Database System

The traditional approach to a DBMS is to have the database-system software running as an application program on a mainframe computer system. This requires the DBMS to share the use and control of the resources with the other applications of the mainframe system. It is obvious that, with the traditional approach, as the workload of the DBMS increases, the performance of the DBMS degrades. [Ref. 4: page 14]

The ***software single-backend approach***, developed by Bell Laboratories [Ref. 5], offloaded the database-system software from the mainframe computer to a separate dedicated computer and partially solved the problems of performance degradation and resource and control sharing.

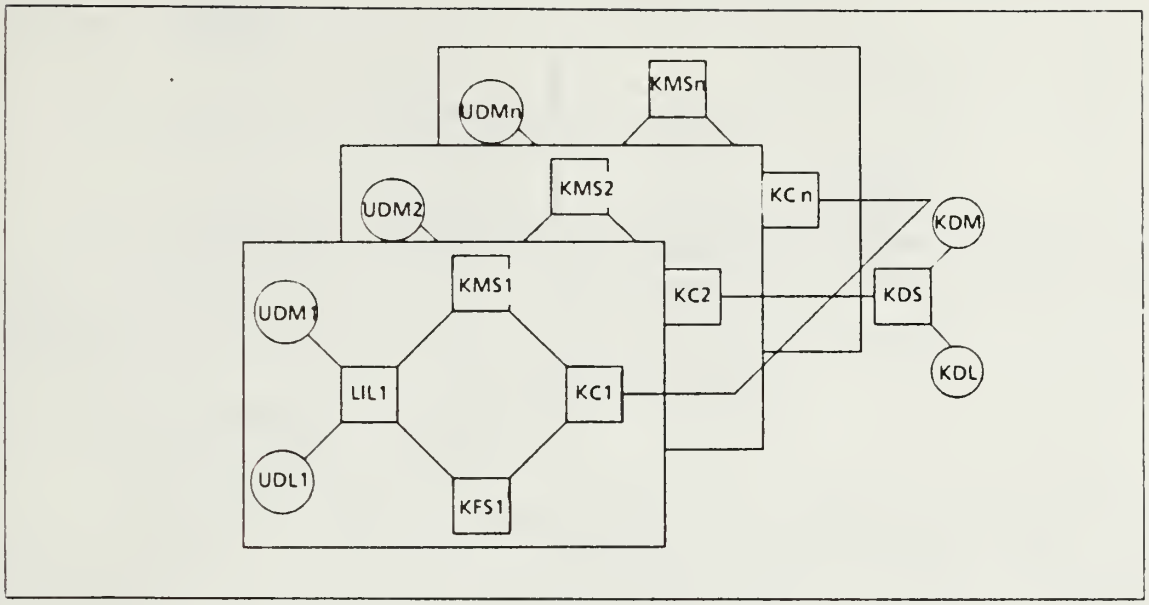

Figure 1.2 Multiple Language Interfaces for KDS.

The Multi-Backend Database System (MBDS) uses a ***software multiple-backend approach*** to overcome the performance problems that remained in the single-software backend approach by utilizing multiple backends connected in parallel. The backends have identical software and their own disks. There is a backend controller, the *master*, which supervises the execution of the database transactions and the interfacing of hosts and users. The backend controller is connected to the individual backends by a communication bus. The backends, or *slaves*, perform the database operations with the database stored on the dedicated disk system of each backend. Users access MBDS through either the host or directly through the backend controller. Figure 1.3 shows the architectural configuration of MBDS.

MBDS realized performance gains over the single-software backend system in two significant areas. First, by increasing the number of backends, while maintaining the size of the database and the size of the responses to the transactions at a constant level, MBDS yields a nearly reciprocal decrease in the response times of the user transactions. The number of backends corresponds directly to ***performance gains*** in terms of ***reduction in response-time.*** Secondly, by increasing the number of backends proportionally with an increase in the size of the database and in the size of responses to user transactions, MBDS produces invariant response-times for the user transactions. This relates the number of backends to the ***capacity growth*** of MBDS in terms of ***response-time invariance.*** [Ref. 6: page 11]

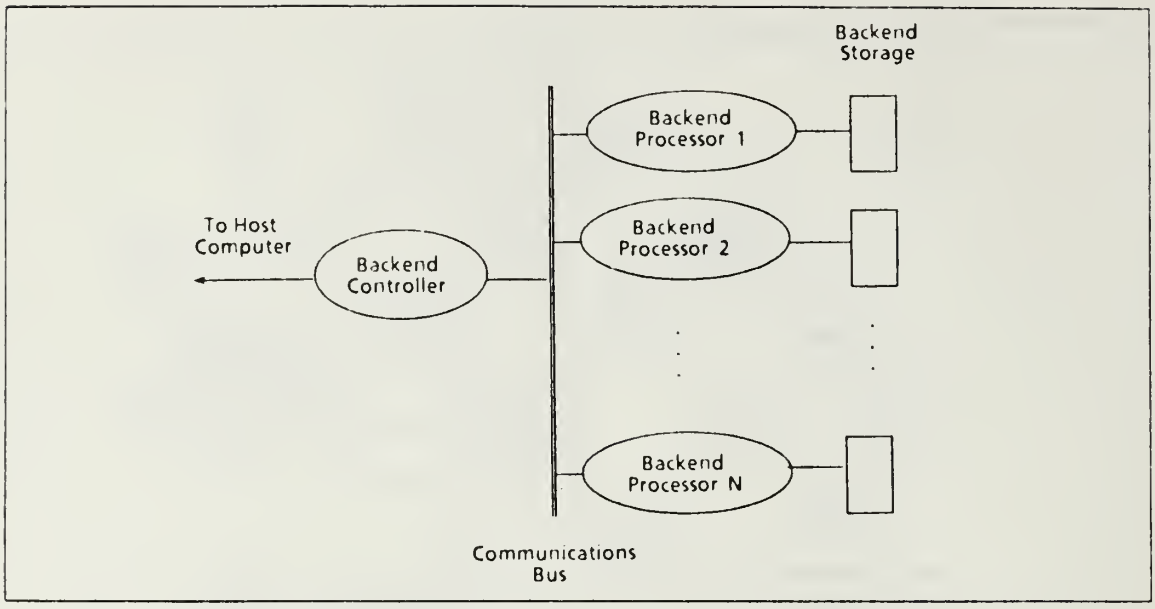

Figure 1.3 MBDS Architecture.

C. THESIS OVERVIEW

This thesis implements the initial step, as described by Rodeck [Ref. 2], in a move toward the Multi-Model Database System (MMDS). Fundamental to this work are the Multi-Lingual Database System and the Multi-Backend Database System as described earlier in this chapter. Additional background material is provided in Chapter II, where the functional, network, and attribute-based data models are discussed along with their respective model-based data languages. Chapter III presents the possible mapping strategies for transforming a functional database into a network database and the generalized translation of CODASYL-DML statements into attribute-based data language requests. Of the three approaches discussed in the chapter, the best solution is chosen and described in greater detail.

Contained in Chapter IV are the various data structures required for this implementation. Each of the data structures is depicted and described along with it's use in the system.

The actual mapping methodology is given in Chapters V and VI. Chapter V discusses the transformation of functional structures into network structures; each structure is described in detail. The translation of CODASYL-DML statements into attribute-based data language requests is specified in Chapter VI. Finally, the conclusions are presented in Chapter VII.

II. THE DATA MODELS

This chapter provides material that will enable the user to become familiar with each of the three data models whose terminologies are needed in this thesis, the *functional* model, the *network* model, and the *attribute-based* data model.

A. THE FUNCTIONAL DATA MODEL AND DAPLEX

1. The Data Model

Sibley and Kershberg [Ref. 7] first introduced the notion of a functional data model while Shipman [Ref. 8] completed the final design of the data model. The functional data model is primarily a logical database model that provides a somewhat natural view of the real world based on *entities* and *relationships*, [Ref. 9: page 9]. The model is based on sets and relationships and maintains a high degree of data independence.

An entity can be considered a distinctly identifiable "thing", while a relationship, or *function*, is an association among these things. Entities of similar structure are collected into *entity sets*. A set of functions will be affiliated with each entity, while the role of an entity in a relationship is the function that it performs in the relationship [Ref. 9: page 11]. A *property* is a piece of information that describes an entity, while an *association* is a many-to-many relationship among entities, [Ref. 10]. A weak entity, or *subtype* is an entity whose existence is dependent on another entity, it's *supertype* or ancestor, in a way that the subtype cannot exist if it's supertype does not also exist. A subtype exists such that entity type A is a subtype of entity type B if and only if every type A is necessarily of type B. Subtyping establishes an ISA relationship among entities and implies value inheritance. Subtypes also have a set of functions associated with them.

Functions can be either *single-valued* or *multi-valued* and those that are defined over entities (types or subtypes) can return scalar values, entities, or set of entities. Scalar values are atomic values which have a literal representation.

2. The Data Language

Whereas a data definition language (DDL) provides for the definition or description of databases, a data manipulation language (DML) supports the accessing or processing of the databases. Daplex is the DDL and the DML for the functional

data model. Most of the concepts on which Daplex is based come from previous work in database management; however, Daplex managed to integrate them into a single framework, the functional data model, and provided a straightforward and almost natural syntax.

It was intended for Daplex to model real-world situations in a manner that is very similar to the conceptual constructs that a person might use when focusing on those same situations; it's goal is "to provide a 'conceptually natural' database interface language" [Ref. 8] and a database system interface which permits the user to more directly model the way he/she attacks the database manipulations. This conceptual naturalness simplifies the use of Daplex since the translation between the user's logical model and model's physical representation in the syntax of Daplex is fairly direct.

The fundamental data definition constructs of Daplex are the entity and the function, with the function mapping a given entity into a set of target entities. The University database schema defined by Shipman and referenced throughout this thesis, is presented in Figure 2.1 and a graphical representation of the database is shown in Figure 2.2.

B. THE NETWORK DATA MODEL AND CODASYL-DML

The network data model is one of the oldest of the data models and may be thought of as an extended form of the hierarchical data model, [Ref. 10: page 542]. It was developed in the late 1960's by the Conference on Data System Languages, Database Task Group, (CODASYL DBTG), which yielded quite a comprehensive specification, [Ref. 11].

1. The Data Model

A network **schema** is about a collection of **records** and **sets.** The schema is a logical view of the database that defines every record field and relationship of the database. The schema contains only the data description; physical constructs are avoided, thus the number of pathological connections to the database architecture are reduced [Ref. 12: page 336].

A ***data-item*** is simply a field or an attribute, whereas a record type is a collection of these data-items. A set is a one-to-many relationship between record types and each set type involves an ***owner*** record type and a ***member*** record type. The owner record types are the "parents" of the member record types, which can be considered the "children" in a one-to-many relationship. A set is defined by specifying its name and identifying the owner record type and the member record type(s). A set

```
DATABASE university IS
   TYPE person;
   SUBTYPE employee;
   SUBTYPE support staff;
   SUBTYPE faculty;
   SUBTYPE student;
   SUBTYPE graduate;
   SUBTYPE undergraduate;
   TYPE course;
   TYPE department;
   TYPE enrollment;
   TYPE rank name IS (assistant, associate, full);
   TYPE semester name IS (fall, spring, summer);
   TYPE grade point IS FLOAT RANGE 0.0 .. 4.0;

   TYPE person IS
      ENTITY
         name : STRING (1 .. 25);
         ssn  : STRING (1 .. 9) := "000000000";
      END ENTITY;

   SUBTYPE employee IS person
      ENTITY
         home address : STRING (1 .. 50);
         office       : STRING (1 .. 8);
         phones       : SET OF STRING (1 .. 7);
         salary       : FLOAT;
         dependents   : INTEGER RANGE 0 .. 10;
      END ENTITY;

   SUBTYPE support staff IS employee
      ENTITY
         supervisor : employee WITHNULL;
         full time  : BOOLEAN;
      END ENTITY;

   SUBTYPE faculty IS employee
      ENTITY
         rank     : rank name;
         teaching : SET OF course;
         tenure   : BOOLEAN := FALSE;
         dept     : department;
      END ENTITY;

   SUBTYPE student IS person
      ENTITY
         advisor     : faculty WITHNULL;
         major       : department;
         enrollments : SET OF enrollment;
      END ENTITY;
   SUBTYPE graduate IS student
      ENTITY
         advisory committee : SET OF faculty;
      END ENTITY;

   SUBTYPE undergraduate IS student
      ENTITY
         gpa  : grade point := 0.0;
         year : INTEGER RANGE 1 .. 4 := 1;
      END ENTITY;
```

Figure 2.1 The University Database Schema.

```
    TYPE course IS
       ENTITY
          title     :  STRING (1 .. 10);
          deptmt    :  department;
          semester  :  semester name;
          credits   :  INTEGER;
          taught by :  SET OF faculty;
       END ENTITY;

    TYPE department IS
       ENTITY
          name :  STRING (1 .. 20);
          head :  faculty WITHNULL;
       END ENTITY;

    TYPE enrollment IS
       ENTITY
          class : course;
          grade :  grade point;
       END ENTITY;

    UNIQUE ssn WITHIN person;
    UNIQUE name WITHIN department;
    UNIQUE title, semester WITHIN course;

    OVERLAP graduate WITH faculty;
END university;
```

Figure 2.1 . (cont'd.)

can have one and only one record type as owner, however, more than one record type may be members. Additionally, a member record can belong to only one instance of a set. The set characteristics are summarized as follows:

- A set is a collection of records.
- There are an arbitrary number of sets in the database.
- Each set has one owner record type and one or more member record types.
- Each owner record occurrence defines a set occurrence.
- There are an arbitrary number of member record occurrences in one set occurrence.
- A record may be a member of more than one set.
- A record may not be a member of two occurrences of the same set.
- A record may be a member and an owner of the same set.

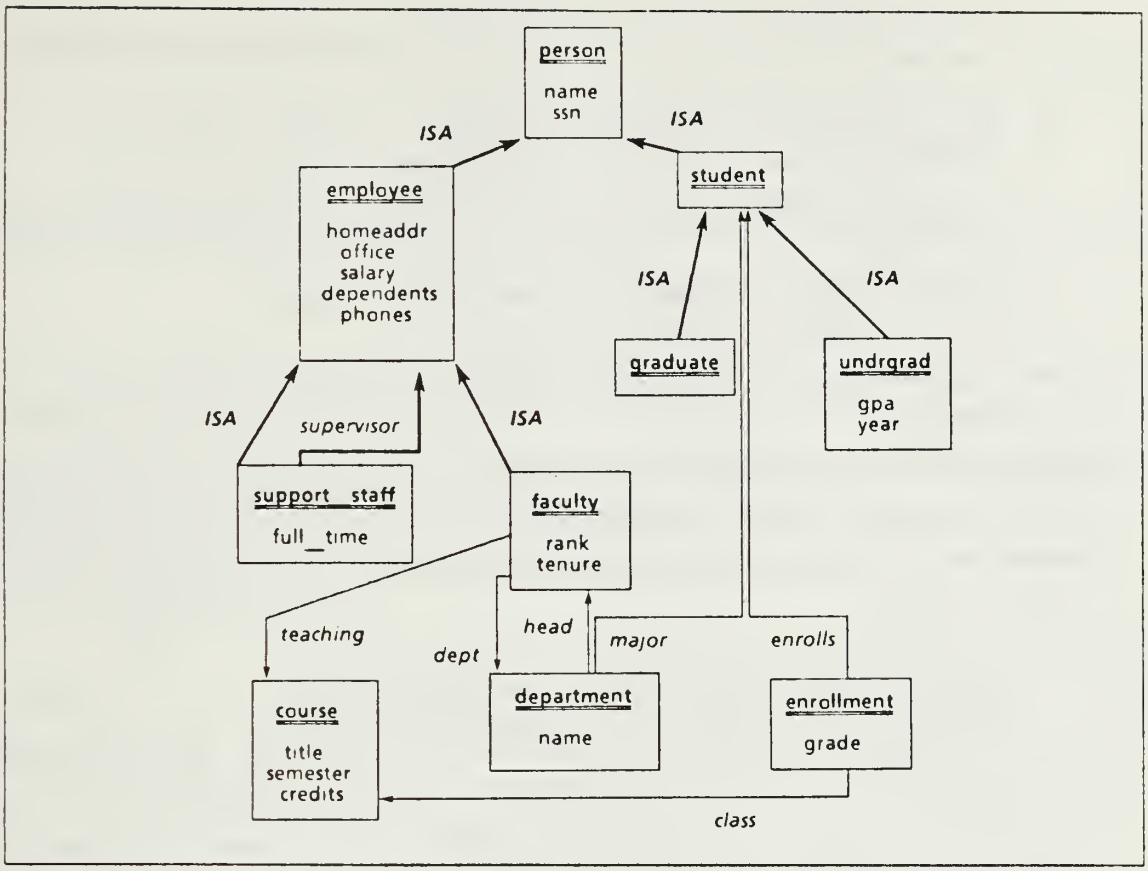

Figure 2.2 Graphical Representation of Univ Schema.

2. The Data Language

CODASYL-DML is a procedural language based upon the concept of currency. A *currency indicator* defines the current position with in a file by maintaining a value of either (1) null, which means that it currently does not identify a record or (2) the address of a record in the database [Ref. 10: page 553]. A ***run-unit*** is essential to this notion of currency and is defined as the execution of a program on behalf of a user. The currency indicator, then, serves as a ***database pointer*** by identifying:

- the current record of the run unit.
- the current record of each record type.
- the current record of each set type.

This thesis will limit itself to a subset of the CODASYL-DML operations, which were implemented as part of the CODASYL-DML language interface in MLDS [Refs. 3,13]. These major operations are listed below:

- FIND identifies a record to be manipulated and marks it as the current of the run unit.
- GET retrieves the current of the run unit.
- MODIFY updates the current of the run-unit.
- CONNECT attaches the current of the run-unit to the current occurrence of the stated set.
- DISCONNECT detaches the current of the run-unit.
- ERASE deletes the current of the run-unit.
- STORE creates a new record occurrence and marks it as the current of the run-unit.

CODASYL-DML tasks are generally executed in two phases. First a FIND command identifies a record to be manipulated and then a second DML command is issued to perform an operation. Most importantly, it is the FIND commands that updated the currency indicators.

C. THE ATTRIBUTE-BASED DATA MODEL AND ABDL

The attribute-based data model (ABDM) was originally proposed by Hsiao [Ref. 14], extended by Wong [Ref. 15], and examined by Rothnie [Ref. 16]. It was chosen as the native model of the MLDS because of it's excellent combination of simplicity and power. The fundamentals of the ABDM are basic, yet the model is capable of representing diverse data models without loss of information.

1. The Data Model

ABDM is based on the ***attribute-value pair*** or ***keyword.*** These attribute-value pairs are formed from a cartesion product of the attribute names and the domains of the values for the attributes. This allows for the representation of ***any*** and ***all*** logical concepts. In order to more fully understand the attribute-value pair we must first define several other terms.

A ***file*** of the database contains groups of ***records***, each of which represents a logical concept. A record is comprised of at most one keyword for each attribute defined in the database and a textual portion, allowing for a verbal description of the record or concept. Figure 2.3 shows the general format of an ABDM record.

Keyword predicates are employed by ABDM to access the database and identify the specific records. A ***keyword predicate*** is a 3-tuple of the form (directory attribute, relational operator, attribute-value). A ***query*** of the database is then the combination, in disjunctive normal form, of keyword predicates.

```
(<attribute_1, value_1>, <attribute_2, value_2>, <attribute_3, value_3>,
 <attribute_n, value_n>, {text})
```

Figure 2.3 Attribute-Based Data Model Record.

A keyword predicate is satisfied only when the attribute of a particular record's keyword is identical to the attribute of the keyword predicate and the relation specified by the relational operator of the keyword predicate holds between the value of the predicate and the value of the keyword predicate. Hence, a record satisfies a query only when all predicates of the query are satisfied by certain keywords of the record.

2. **The Data Language**

ABDL, as defined by [Ref. 17], is the kernel data language of MLDS. It allows five basic database operations that are capable of making numerous in-depth transactions on the database. The database operations provided by ABDL are, INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON, however, this implementation will not concern itself with the latter operation.

ABDL allows the user to issue either a request or a transaction. A ***request*** is a basic operation with an attached qualification. The ***qualification*** specifies the portion of the database that is to be manipulated, while a ***transaction*** is defined as the grouping together of two or more sequentially executed requests. The four operations used in this work are explained below, [Ref. 6: page 10].

- INSERT places a new record into the database and is qualified by a list of keywords.
- DELETE removes one or more records from the database and qualified by a query.
- UPDATE modifies records of the database and is qualified by a query and a modifier. The query identifies one or more records to be updated, while the modifier specifies how the target record(s) are to be modified.
- RETRIEVE accesses and returns records of the database and is qualified by a query, a target-list, and a by-clause. The query identifies the record(s) to be retrieved. The target-list contains a list of output attributes, and the by-clause may be used to group records when an aggregate operation is specified.

Together, these five ABDL operations provide all of the required processing to support data-language translation.

III. DATABASE MAPPINGS

For the purpose of this thesis, ***data-model transformation*** is the mapping process from a given data model to the kernel data model (ABDM), and ***data-language translation*** is the mapping process from a given data language to the kernel data language (ABDL). MLDS has already implemented four data-model transformations (hierarchical, relational, network, and functional to ABDM) and four data-language translations (SQL, DL/I, CODASYL-DML, and Daplex to ABDL). This thesis makes use of two of the aforementioned data-model transformations (network to ABDM and functional to ABDM) of Lim and Emdi [Refs. 18,19], and one of the data-language translations (CODASYL-DML to ABDL) [Ref. 19]. Figure 3.1 depicts the high-level transformations and translations of the network and functional data models. It should be noted that the databases that are transformed from the network schema and the functional schema to an attribute-based schema are represented throughout this thesis as *AB(network)* and *AB(functional)*, respectively.

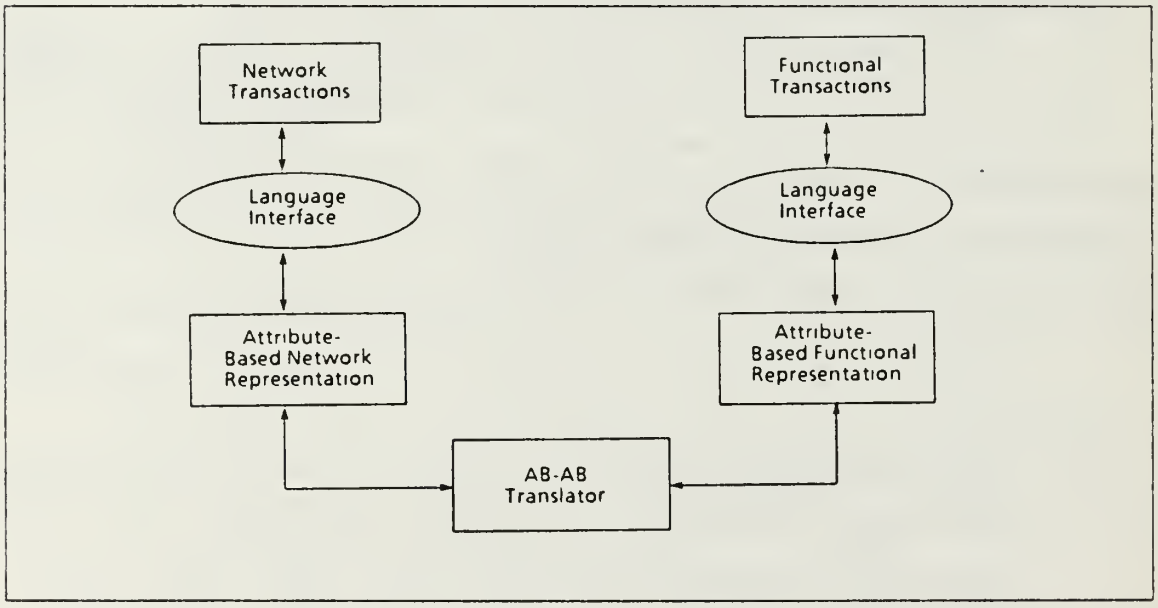

Figure 3.1 MLDS Mapping of the Network and Functional Data Models.

The thrust of this work is (1) transforming a functional database into a network database and (2) modifying the CODASYL-DML to ABDL translation in order to

allow CODASYL-DML transactions on an AB(network) database that has been previously transformed from the functional data-model to the network data-model.

A. BACKGROUND MATERIAL

The MLDS mappings of network(CODASYL-DML) to ABDM(ABDL) is a modification of the procedure developed by Banerjee [Ref. 17], explicitly defined by Wortherly [Ref. 3: pages 31-37], and will therefore only be generalized in the following paragraph.

The key point in the mapping process is the retention of the network records and sets; the mapping algorithm does, in fact, retain those notions through the use of attribute-based constructs. The translation of CODASYL-DML to ABDL requests was implemented by Emdi [Ref. 19], and as previously discussed, only a subset of the CODASYL-DML statements were considered: FIND, GET, STORE, CONNECT, DISCONNECT, ERASE, and MODIFY. The translation maintains the all important notion of currency by using a ***Currency Indicator Table*** (CIT). The actual structure and implementation of the CIT are defined in detail in a later chapter. Another translation consideration is the one-to-many correspondences between the CODASYL-DML statements and the ABDL requests; this necessitated a storage facility to maintain the intermediate information for the ABDL requests. The ***request buffer*** (RB) is used to store the information returned by the auxiliary retrieve requests (ARR), of which several may be generated by the translation of a single CODASYL-DML statement. With the exception of several flags and special conditions, the translation process of this thesis is similar to that of Emdi [Ref. 19].

B. MAPPING THE FUNCTIONAL(DAPLEX) MODEL(LANGUAGE) TO THE NETWORK(CODASYL-DML) MODEL(LANGUAGE)

1. Available Strategies

The goal of this thesis is to provide the network/CODASYL-DML user with the means of accessing a functional database without the user having to be familiar with the functional data-model and Daplex. As one might imagine, this task requires a sound mapping strategy that maintains the constructs and characteristics of the target (functional) database while allowing the CODASYL-DML statements to access this target database. Rodeck, [Ref. 2], proposed the following mapping strategies:

- DIRECT LANGUAGE INTERFACE: modify MLDS's existing LIL to allow the transformation of a functional schema to a network schema along with a new language interface between the network model and AB(network).

- AB-AB POSTPROCESSING: create a language interface between the AB(functional) and the AB(network) databases along with a CODASYL-DML translator.
- HIGH-LEVEL PREPROCESSING: create a functional schema to network schema transformer along with a CODASYL-DML to Daplex translator.

The Direct Language Interface approach proved to be best suited for this implementation and the reasons for its selection are discussed in the following section.

2. **The Selected Mapping Strategy**

Each of the three mapping strategies was analyzed and compared with the other two strategies by Rodeck [Ref. 2]. The evaluation process looked at their respective advantages and disadvantages before finally selecting the direct language interface approach primarily because of the following implementation considerations:

- a one-step schema transformation.
- a faster schema transformation.
- highest compatibility with existing components of MLDS.

The direct language interface strategy transforms the functional database into a network database and allows the user to access the transformed database with a subset of CODASYL-DML statements. These statements are translated into one or more ABDL requests and executed on the AB(network) database. Figure 3.2 depicts the direct language approach. By comparing Figure 3.2 with Figure 3.1, one can see that the primary difference is the addition of the schema transformer and the modified language interface. It is the schema transformer that represents the process of transforming the functional schema into the network schema. With the exception of the schema transformer, this approach is similar to the approach with the network to AB(network) and the functional to AB(functional) transformation. The goal of the Multi-Model and Multi-Lingual Database System can be conceptualized by placing schema transformers between all model/language pairs, thereby arriving at a fully-database-sharing environment.

C. DATA-MODEL TRANSFORMATIONS REFERENCED IN THIS THESIS

This section provides a ***high-level*** view of the data-model transformations that are referenced in this thesis. In the first subsection, the functional to ABDM mapping is presented. The functional to network mapping is introduced in the second subsection.

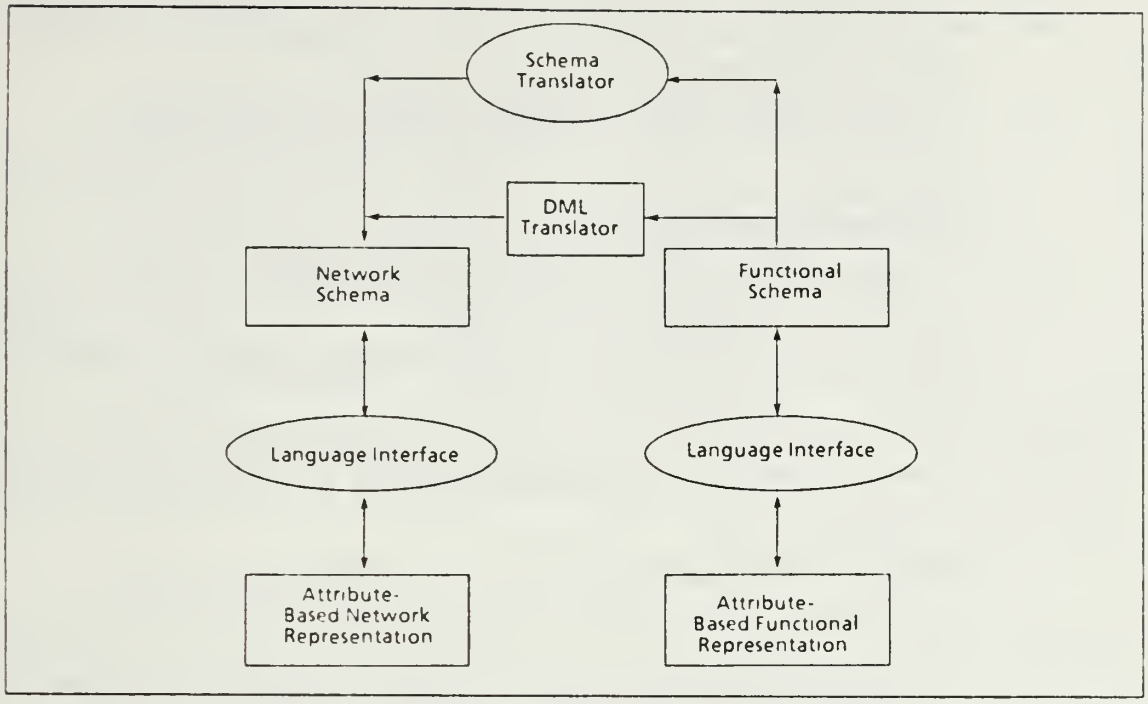

Figure 3.2 Direct Language Interface Approach.

1. **The Functional to ABDM Mapping**

The primary task of this mapping is to transform the constructs of the functional data-model into ABDM constructs. This approach shows that, given the attribute-value pairs in a record in ABDM, the functions of the functional data-model map into the attributes of the corresponding attribute-value pairs. An algorithm to map the entity types and subtypes into ABDM constructs was designed by Goisman [Ref. 20], and implemented by Anthony and Billings [Ref. 21].

In order to represent the relationships of the functional data-model that must exist between individual records of ABDM, the related attributes for each related record must be repeated [Ref. 20: page 35]. This is accomplished by using an artificial attribute and its associated value to allow for unique mappings. The artificial attribute is in fact a unique key for each entity type or subtype in the functional data-model, thereby allowing for the relationships amongst entities to exist in accordance with the unique key. The remainder of the transformation algorithm is given below:

(1) An ABDM file is created for each entity type and subtype. The first attribute-value pair has as its attribute "File" and its value is the entity type or subtype name.

(2) The second attribute attribute-value pair for each ABDM file representing an entity has as its attribute the name of the corresponding entity type. The value of this attribute-value pair is the unique key.

(3) For each ABDM file transformed from an entity subtype, the second attribute-value pair of each record has as its attribute the name of the corresponding entity subtype and its value is the record consisting of its entity supertype and its unique key.

(4) For each function applied to an entity type or subtype, an attribute-value pair is inserted into the corresponding ABDM file. The attribute of the attribute-value pair is the functions name and the value is the value returned by the particular function.

Using this algorithm to transform the University database schema of Figure 2.1 results in the AB(functional) database as depicted in Figure 3.3. The asterisks represent relationship-dependent values.

2. **Functional to Network Mapping**

This subsection provides the reader with a ***high-level*** view of the mapping algorithm described by Rodeck [Ref. rRod]. The specific implementation issues of the algorithm are discussed in Chapter V of this thesis. As is the case in all data-model transformations, the goal is to provide the user with a familiar and accurate representation of the source database schema. In mapping the functional data-model to the network data-model we are primarily concerned with the basic functional constructs: the entity type, the entity subtype, and the non-entity types.

(1) Entity types are mapped into network records with the record name being the name of the corresponding entity type. Additionally, each entity type is a member of a set type which is owned by SYSTEM.

(2) For each entity subtype, a record type must be declared with the record name being the name of the subtype. A set type is also declared with the owner being the subtype's entity supertype.

(3) Non-entity types map fairly directly to network constructs:

  (a) Integers map to integers.

  (b) Strings map into characters.

  (c) Floating-points map into floating-points.

  (d) Enumeration types map into characters.

(4) The functions that are applied to entity types and subtypes can be scalar, scalar multi-valued, single-valued, or multi-valued:

  (a) Scalar and scalar multi-valued functions map into attributes of the corresponding record type of the entity type or subtype.

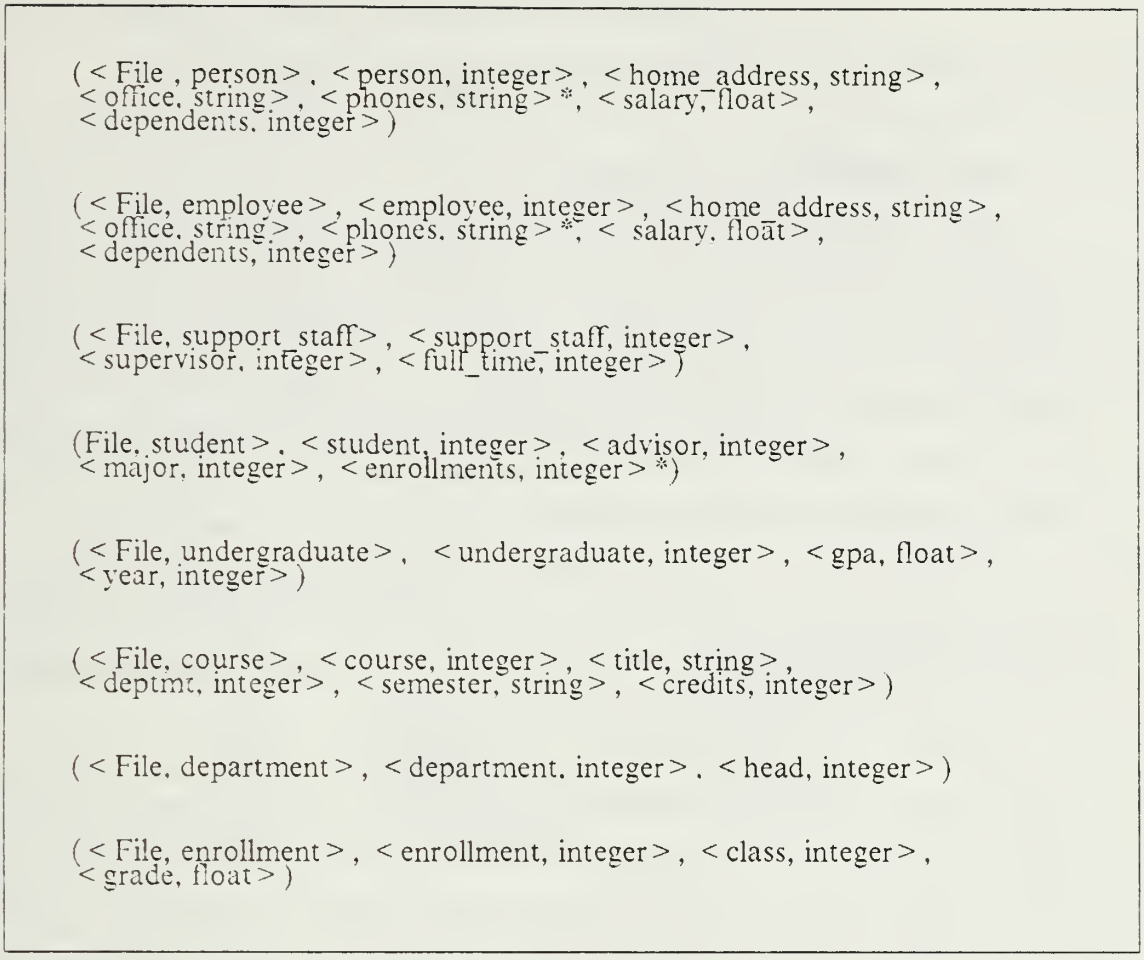

Figure 3.3 The AB(functional) University Database Schema.

(b) Single-valued functions map into sets with the name of the particular function, owned by the corresponding record type of the entity type or subtype.

(c) The mapping of multi-valued functions is performed depending upon whether the multi-valued function is a one-to-many or a many-to-many relationship.

Chapter V provides detailed explanations of the mapping algorithm as well as a complete database transformation. In a later chapter, we demonstrate this transformation process for the Daplex university schema given.

IV. THE DATA STRUCTURES

A. DATA SHARED BY ALL USERS

Both the CODASYL-DML and the Daplex language interfaces have been developed as single-user systems that will eventually will be modified to multi-user systems. Appropriately, two separate concepts of data are used the in the language interface: (1) data structures that are shared by all users, and (2) data specific to each user. The requirements of this thesis work have necessitated the slight modification of several existing data structures from previous implementations on MLDS; however, the generic data structures are for this implementation are not drastically altered.

The data structures that are shared by all users are the database schemas that have been loaded (defined) by the users. The schemas that are of interest to this thesis are the functional schemas, consisting of entities and the functions of the entities, and the network schemas, comprised of sets and attributes.

The first data structure, Figure 4.1, is represented as a union and supports each of the previous MLDS implementations (i.e., SQL, DL/I, CODASYL-DML, or Daplex) as well. At this point, our interest lies with the functional and network models. In this regard, either the third or fourth fields will be activated. Should the selected database be based on the functional model, the fourth field of the union would point to the structure represented in Figure 4.7, fun_dbid_node. Likewise, if a network schema were being manipulated, the third field of the dbid_node would be activated and point to a structure of type net_dbid_node, Figure 4.2.

```
union dbid_node
{
   struct      rel_dbid_node      *rel;
   struct      hie_dbid_node      *hie;
   struct      net_dbid_node      *net;
   struct      fun_dbid_node      *ent;
}
```

Figure 4.1 The dbid_node Data Structure.

1. **Data Shared by All Users of a Network Database**

The first field of the net_dbid_node is a character array holding the name of the respective network schema. The second and third fields are integer values representing the number sets and records in the schema. An integer value representing a database key is maintained in the fourth field, while the fifth, sixth, and seventh fields are pointers to structures containing information about each set and record of the schema. Specifically, the fifth field and seventh fields point to the first set and record, respectively, of the schema, and the sixth and eighth fields point to the current set and record, respectively, of the schema. The final field of the net_dbid_node is a pointer to a structure representing the next network schema in the MLDS.

```
struct net_dbid_node
{
   char                                ndn_name[DBNLength + 1];
   int                                 ndn_num_set;
   int                                 ndn_num_rec;
   int                                 ndn_dbkey;
   struct      nset_node               *first_set;
   struct      nset_node               *curr_set;
   struct      nrec_node               *first_rec;
   struct      nrec_node               *curr_rec;
   struct      net_dbid_node           *next_db;
}
```

Figure 4.2 The net_dbid_node Data Structure.

The nset_node data structure, Figure 4.3 represents information each set in the schema. The first field, nsn_name, is a character array holding the name of the particular set, while the second and third fields are also character arrays containing the names of the owner and member of the set. The fourth and fifth fields are characters representing the insertion and retention modes of the set. The insertion mode can be either automatic, 'a', or manual, 'm', and the retention mode can be fixed, 'f', manual, 'm', or optional, 'o'. The select_mode field is a pointer to a set_select_node. The seventh field is a pointer to the owner record type of the respective set type and the eighth field is a pointer to the member record type of the respective set type.

Figure 4.4 shows the set_select_node data structure. This structure maintains the set selection mode information for each set. The first field is a character representation of the set selection mode, either by VALUE, 'v', by STRUCTURE, 's',

```
struct nset_node
{
   char                                  name[SNLength + 1];
   char                                  owner_name[ONLength + 1];
   char                                  member_name[MNLength + 1];
   char                                  ancestor[ANLength + 1];
   char                                  insert_mode[INLength + 1];
   char                                  retent_mode[RLength + 1];
   struct      set_select_node           *select_mode;
   struct      nrec_node                 *owner;
   struct      nrec_node                 *member;
   struct      nset_node                 *next_set;
}
```

Figure 4.3 The nset_node Data Structure.

by APPLICATION, 'a', or not specified, 'o'. If the set selection mode of the set is by value or by structural, the second field, a character array, will hold the item name of the specified record and the third field will hold the name of the record. The fourth field will contain the name of a second record only if the set selection mode is by structural.

```
struct set_select_node
{
   char        select_mode[SLength + 1];
   char        item_name[ANLength + 1];
   char        record1_name[RNLength + 1];
   char        record2_name[RNLength + 1];
}
```

Figure 4.4 The set_select_node Data Structure.

The nrec_node, Figure 4.5, contains information concerning each record in the schema. The first field is a character array holding the name of the record and the second field is an integer representation of the number of attributes of the record. The third and fourth fields are pointers to structures containing information about the first and current attributes of the particular record. The final field of nrec_node is a pointer to the next record type representation in the schema.

```
struct  nrec_node
{
   char                                 nrn_name[RNLength + 1];
   int                                  nrn_num_attr;
   char                                 nrn_ancestor[ANLength + 1];
   struct       nattr_node              *first_attr;
   struct       nattr_node              *curr_attr;
   struct       nrec_node               *next_rec;
}
```

Figure 4.5 The nrec_node Data Structure.

The nattr_node is depicted in Figure 4.6. Information about the attributes of each CODASYL record type is maintained in this data structure. The first field is a character array containing the name of the attribute while the second and third fields represent the level number and type of the attribute. The attribute can be either an integer, 'i', a floating point number, 'f', or a string, 's'. The fourth field determines the maximum length that a value of this attribute may possibly have and the fifth field indicates the maximum length of the decimal portion of a value if this attribute type is a floating point number. The sixth field is an integer valued flag indicating whether or not the attribute can have duplicates. It is initialized to '0', allowing for duplicates. The seventh, eighth, and ninth fields are pointers to structures representing the next attribute, the child of the attribute, and the parent of the attribute, respectively.

```
struct  nattr_node
{
   char                                 nan_name[ANLength + 1];
   char                                 nan_level_num[ALLength + 1];
   char         nan_type;
   int                                  length1;
   int                                  length2;
   int                                  dup_flag;
   struct       nattr_node              *next_attr;
   struct       nattr_node              *child;
   struct       nattr_node              *parent;
}
```

Figure 4.6 The nattr_node Data Structure.

2. **Data Shared by All Users of a Functional Database**

If the database accessed by the user is based on the functional data-model, then the fourth field of the dbid_node data structure, Figure 4.1, will be activated. The pointer will be directed to a structure of type fun_dbid_node, Figure 4.7.

The fun_dbid_node contains information about a functional database. The first field is a character array which represents the name of the database. The second field is a pointer to the base-type nonentity node, and fdn_num_nonent is and integer value of the number of the base-type nonentity nodes in the database. The following field, *fdn_entity, points to the entity node and while the fifth field is an integer value of the number of these nodes. The sixth field is a pointer to the generalized entity subtype node and as before the field that immediately follows contains an integer value representing the number of such nodes. The fdn_nonsubptr is the nonentity subtypes and the number of these nodes is maintained in the ninth field, fdn_num_nonsub. The next field, *fdn_nonderptr, is a pointer to the nonentity derived types respectively with the eleventh field containing the integer value for the number of such nodes. The fdn_ovrptr is a pointer to a structure containing the overlap constraints of the database and the thirteenth field, fdn_num_ovr keeps track of the number of overlap constraints. The final field of the fdn_dbid_node structure is a pointer to the next functional schema in the MLDS.

```
struct  fun_dbid_node
{
    char        fdn_name[DBNLength + 1];
    struct      ent_non_node        *fdn_nonentity;
    int                             fdn_num_nonent;
    struct      ent_node            *fdn_entity;
    int                             fdn_num_ent;
    struct      gen_sub_node        *fdn_subptr;
    int                             fdn_num_gen;
    struct      sub_non_node        *fdn_nonsubptr;
    int                             fdn_num_nonsub;
    struct      der_non_node        *fdn_nonderptr;
    int                             fdn_num_der;
    struct      overlap_node        *fdn_ovrptr;
    int                             fdn_num_ovr;
    struct      fun_dbid_node       *fdn_next_db;
}
```

Figure 4.7 The fun_dbid_node Data Structure.

The ent_node data structure is shown in Figure 4.8. The first field of this structure is a character array containing the name of the entity. The en_last_ent_id field is an integer value representing the last unique number assigned to the particular entity node. The third field is an integer representation of the number of functions associated with the particular entity type, while the fourth field, en_terminal, is an integer representation of a boolean flag that indicates whether or not the entity is a terminal type. An entity type is a terminal type only when it is not a supertype to any entity subtype. The *en_ftnptr field is a pointer to the function nodes associated with the particular entity node. The final field of the ent_node data structure is a pointer to the next entity (ent_node) in the schema.

```
struct  ent_node
{
   char                                 en_name[ENLength + 1 ];
   int                                  en_last_ent_id;
   int                                  en_num_funct;
   int                                  en_terminal;
   struct        function_node          *en_ftnptr;
   struct        ent_node               *en_next_ent;
}
```

Figure 4.8 The ent_node Data Structure.

Figure 4.9 depicts the gen_sub_node. This data structure contains information about the entity subtypes of the accessed database. The first field is a character array holding the name of the generalized entity subtype. The gsn_num_funct field is an integer value representing the number of functions associated with the entity subtype, while the third field, gsn_terminal is an integer representation of a boolean flag indicating whether or not the entity subtype is a subtype of an entity type and not a supertype to any entity subtypes. The fourth field is a pointer to the entity supertype of the particular entity subtype represented by the gen_sub_node. The gsn_num_ent field is an integer value indicating the number of entity supertypes of the subtype. The next field, *gsn_ftnptr, is a pointer to the functions associated with the entity subtype, and the *gsn_subptr field is a pointer to the subtype supertype. The eighth field holds the number of these subtype supertypes. The last field of the gen_sub_node data structure is simply a pointer to the next generalized entity subtype in the schema.

```
struct  gsn_sub_node
{
   char                                gsn_name[ENLength + 1];
   int                                 gsn_num_funct;
   int                                 gsn_terminal;
   struct      ent_node_list           *gsn_entptr;
   int                                 gsn_num_ent;
   struct      function_node           *gsn_ftnptr;
   struct      sub_node_list           *gsn_subptr;
   int                                 gsn_num_sub;
   struct      gen_sub_node            *gsn_next_genptr;
}
```

Figure 4.9 The gsn_sub_node Data Structure.

Information concerning each nonentity base-type is maintained in a data structure of type ent_non_node, Figure 4.10. The first field of this data structure, similar to previous data structures, is a character array containing the name of the nonentity base-type. The enn_type field is a character flag indicating the type of nonentity node, either, integer, 'i'; enumeration, 'e'; floating point number, 'f'; character string, 's'; or boolean, 'b'. The third field is an integer value which represents the maximum length of the nonentity base-type value. The enn_range field contains an integer representation of a boolean flag that indicates whether or not there is a range of values associated with the nonentity base-type. The next field, enn_num_values, represents the number of different values that the nonentity can assume. The sixth field is a pointer to the actual value of the nonentity base-type, while the following field, enn_constant, is an integer representation of a boolean flag indicating whether or not the base-type is a constant. The last field in the ent_non_node data structure is a pointer to the next nonentity base-type in the schema.

The sub_non_node data structure, Figure 4.11, contains information about the nonentity subtype base-types in the functional schema. The first field of the data structure is a character array containing the name of the nonentity subtype and the enn_type field holds a character that indicates the type of nonentity subtype, either integer, 'i'; enumeration, 'e'; floating point number, 'f'; character string, 's'; or boolean, 'b'. The next field, snn_total_length, contains an integer that indicates the maximum length of the nonentity subtype value. The snn_range field is an integer representation of a boolean flag which indicates whether the nonentity subtype has a range of values associated with it. The fifth field contains an integer that represents the number of

```
struct  ent_non_node
{
   char                                 enn_name[ENLength + 1];
   char                                 enn_type;
   int                                  enn_total_length;
   int                                  enn_range;
   int                                  enn_num_values;
   struct       ent_value               *enn_value;
   int                                  enn_constant;
   struct       ent_non_node            *enn_next_node;
}
```

Figure 4.10 The ent_non_node Data Structure.

different values that the nonentity subtype can assume. The next field is a pointer to the actual value of the node, while the final field of the data structure, *snn_next_node, is a pointer to the next nonentity subtype in the schema.

```
struct  sub_non_node
{
   char                                 snn_name[ENLength + 1];
   char                                 snn_type;
   int                                  snn_total_length;
   int                                  snn_range;
   int                                  snn_num_values;
   struct       ent_value               *snn_value;
   struct       sub_non_node            *snn_next_node;
}
```

Figure 4.11 The sub_non_node Data Structure.

The der_non_node data structure, Figure 4.12, pertains to the derived nonentity types of the functional schema; it is identical in structure to the sub_non_node, Figure 4.11.

Figure 4.13 depicts the organization of the overlap_node data structure. The initial field of the structure contains the name of the base type for the overlapping entities. The *snlptr field is a pointer to the list of terminal subtypes, sub_node_list, that are overlapped. The next field, num_sub_node, indicates the number of overlapped subtypes in sub_node_list. The final field in Figure 4.13 is a pointer to the next overlap_node in the schema.

```
struct  der_non_node
{
    char                                dnn_name[ENLength + 1];
    char                                dnn_type;
    int                                 dnn_total_length;
    int                                 dnn_range;
    int                                 dnn_num_values;
    struct      ent_value               *dnn_value;
    struct      der_non_node            *dnn_next_node;
}
```

Figure 4.12 The der_non_node Data Structure.

```
struct          overlap_node
{
    char                                base_type_name[ENLength + 1];
    struct      sub_node_list           *snlptr;
    int                                 num_sub_node;
    struct      overlap_node            *next;
}
```

Figure 4.13 The overlap_node Data Structure.

Each function declared in the functional schema is represented by a data structure of the type function_node, Figure 4.14. The name of the function is contained in the first field of the structure, while the second field is a character which represents the type of the function, either floating point number, 'f'; integer, 'i'; character string, 's'; boolean, 'b'; or entity, 'e'. The next field, fn_set, is an integer value representing a boolean flag that is used to indicate whether the function is a set-valued function. The fn_range field indicates whether or not there is a range of values associated with the function. The next field indicates the maximum length of the values and the fn_num_value field indicates number of values. The following field is a pointer to the actual value, which the next five fields hold pointers to the type to which the particular function belongs. A function may belong to only one type, either an entity, an entity subtype, an nonentity, a nonentity subtype, or a nonentity derived type. The thirteenth field indicates whether or not there is an entity value associated with the function. The fn_unique field is used to indicate whether or not the function is unique, while the final field is a pointer to the next function in the schema.

```
struct  function_node
{
   char                                fn_name[ENLength + 1];
   char                                fn_type;
   int                                 fn_set;
   int                                 fn_range;
   int                                 fn_total_length;
   int                                 fn_num_value;
   struct        ent_value             *fn_value;
   struct        ent_node              *fn_entptr;
   struct        gen_sub_node          *fn_subptr;
   struct        ent_non_node          *fn_nonentptr;
   struct        sub_non_node          *fn_nonsubptr;
   struct        der_non_node          *fn_nonderptr;
   int                                 fn_entnull;
   int                                 fn_unique;
   struct        function_node         *next;
}
```

Figure 4.14 The function_node Data Structure.

The ent_node_list, Figure 4.15, and the sub_node_list, Figure 4.16, data structures are used to maintained linked lists of entity types and generalized entity subtypes, respectively.

```
struct  ent_node_list
{
   struct        ent_node              *entptr;
   struct        ent_node_list         *next;
}
```

Figure 4.15 The ent_node_list Data Structure.

```
struct  sub_node_list
{
   struct        gen_sub_node          *subptr;
   struct        sub_node_list         *next;
}
```

Figure 4.16 The sub_node_list Data Structure.

The final data structure that is shared by users accessing a functional schema, ent_value, is shown in Figure 4.17. The structure's function is to maintain a linked list of entity values.

```
struct  ent_value
{
   char                                  *ev_value;
   struct        ent_value               *next;
}
```

Figure 4.17 The ent_value Data Structure.

B. DATA SPECIFIC TO EACH USER

The data structures that are discussed in this section are necessary in order to support each user's particular interface requirements. The key structure is depicted in Figure 4.18, user_info. This structure holds information on each user currently using a particular language interface of MLDS. The first field of user_info is a character array containing the user's ID. The next field is a union that describes a particular interface and the last field is simply a pointer to the next user of MLDS.

```
struct  user_info
{
   char                                  uid[UIDLength + 1];
   union         li_info                 li_type;
   struct        user_info               *next_user;
}
```

Figure 4.18 The user_info Data Structure.

The union, li_info, depicted in Figure 4.19, can hold the data for a user accessing any type language interface of database schemas supported by MLDS, (SQL, DL/1, CODASYL-DML, or Daplex). For the purpose of this thesis the data structures peculiar to the CODASYL-DML language interface and the Daplex language interface will discussed.

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

Figure 4.19 The li_info Data Structure.

Should the user access a network database, the third field of Figure 4.19, li_dml, will be activated. This action will call upon Figure 4.20, dml_info.

```
struct  dml_info
{
   struct        cur_db_info       curr_db;
   struct        file_info         file;
   struct        tran_info         dml_tran;
   struct        ddl_info          *ddl_files;
   int                             operation;
   int                             answer;
   int                             error;
   union         kms_info          kms_data;
   union         kfs_info          kfs_data;
   union         kc_info           kc_data;
   struct        cur_table         *cur_table;
   int                             buff_count;
}
```

Figure 4.20 The dml_info Data Structure.

The dml_info data structure, Figure 4.20, contains user information mation concerning the CODASYL-DML language interface. The curr_db_info field is also a structure; it contains information about the network database being accessed by the user. The file field is a data structure which contains the file descriptor and identifier of a file of CODASYL-DML transactions. The third field, dml_tran, is a data structure that maintains information describing the CODASYL-DML transactions that are awaiting processing. The next field is a pointer to the ddl_info data structure, which describes the descriptor and template files. The operation field is an integer representation of a flag used to indicate the operation to be performed on the network

database, either loading a new network database, or executing a request on an existing network database. The answer is an integer value that is used by the Language Interface Layer (LIL) to record the answer it receives from interfacing with the user. The eleventh field is a pointer to the Currency Indicator Table (CIT), as discussed by Meyer and MacDougal, buff_count, is a count of the result buffers of the Kernel Controller (KC).

If the user accesses a functional database, then the fourth field of the li_info data structure, li_dap, is activated, referencing the dap_info data structure, Figure 4.21. This structure contains information about the Daplex language interface and is similar to the dml_info data structure, Figure 4.20, with the exception that it applies to Daplex rather than CODASYL-DML.

```
struct  dap_info
{
   struct        curr_db_info        dpi_curr_db;
   struct        file_info           dpi_file;
   struct        tran_info           dpi_dml_tran;
   int                               dap_operation;
   struct        ddl_info            *dpi_ddl_files;
   union         kms_info            dpi_kms_data;
   union         kfs_info            dpi_kfs_data;
   union         kc_info             dpi_kc_data;
   int                               dap_error;
   int                               dap_answer;
   int                               dap_buff_count;
}
```

Figure 4.21 The dap_info Data Structure.

V. FUNCTIONAL TO NETWORK TRANSFORMATION ALGORITHMS

The Language Interface Layer (LIL) is the first module in the mapping process of MLDS. Its function is to control the order in which the other modules are called and to allow the user to either load a new database or process an existing database. The implementation of this thesis, in addition to permitting the user to load a new network database, allows the user to apply transactions to either a network or a functional database. When an existing database (network or functional) is to be processed, the user is queried for the name of the database. LIL then uses the user-supplied name and first searches the existing network schemas; if the desired database is in fact a network database, then LIL primarily functions as implemented in Reference 19. If the desired database is not found to be in the list of existing network schemas, the list of functional schemas is then searched. If the desired database is found to be an existing functional database, a mapping process is initiated in order to transform the functional schema into a network schema. This transformed database is actually a ***network representation*** of the functional database which maintains the characteristics of the functional database while preserving its constraints [Ref. 2: page 52].

In order to preserve the constraints of the ***source*** database (functional), there are six essential constructs of the functional schema that must be accurately transformed to equivalent constructs of the ***target*** database (network). The constructs of the functional schema are:

- the entity type
- the entity subtype
- the non-entity types
- the uniqueness constraints
- the overlap constraints
- the set type

The methodology for the transformation was primarily implemented as designed in Reference 2 and is described in detail in the following subsections. In order to provide the reader with realistic examples of the mapping of a functional schema to a network schema, this section depicits the transformation of the functional-based University

database schema of Figure 2.1, to the network University database schema shown in Figure 5.1. Figure 5.1 is referenced throughout this chapter.

A. ENTITY TYPES

In transforming a functional entity, LIL maps not only the entity itself, but also the functions of the entity, as the functions are applied to the respective entity type. The function-type may be string, scalar (integer, floating-point, enumeration), entity, non-entity or a set of any of the above. The form of an entity-type declaration is shown in Figure 5.2, where ***entityXX*** is the unique name of the entity being declared and ***functionXX1***, ***functionXX2***, ..., ***functionXXn*** are the names of the functions that can be applied to ***entityXX***. The ***function_types*** determine what type of value will be returned by the respective functions.

In the transformation process, an entity type is mapped into a network record type. Each entity is also made a member of a set type which is owned by SYSTEM. When mapping the function types associated with a particular entity, LIL must determine whether or not the function type is a scalar function, scalar multi-valued function, single-valued function, or multi-valued function. It accomplishes this task by checking several fields of the ***function_node*** data structure of Figure 4.14.

A particular function of an entity is a scalar function if the ***fn_subptr*** and ***fn_entptr*** fields are NULL and the ***fn_set*** field is not set (i.e., has a value of zero), indicating that the function does not belong to a specific entity type or subtype, nor is it set-valued. Scalar functions are mapped into attributes of the record type that has been transformed from the function's entity.

A function is determined to be a scalar multi-valued function if it meets two of the three criteria discussed in the preceding paragraph; it's ***fn_entptr*** and ***fn_subptr*** fields are NULL, however, the ***fn_set*** field is set to a value of "1", indicating that it is a set-valued function. The scalar multi-valued function is declared as an attribute in the corresponding record type. It must be noted, however, that only one occurrence of the single multi-valued function may be stored in the record, therefore the ***nan_dup_flag*** field of the nattr_node, Figure 4.7, is not set, indicating that the attribute cannot have duplicates.

If either the ***fn_entptr*** or the ***fn_subptr*** field of Figure 4.14 is not NULL, then the function in question is either a single or a multi-valued function. Again, the determining factor is the ***fn_set*** field; if it is set to a value of "1", then the function is a multi-valued function. In the case of a single-valued function, a network set type is

```
SCHEMA NAME IS university;


RECORD NAME IS person;
DUPLICATES ARE NOT ALLOWED FOR ssn;
   name:  CHARACTER 25;
   ssn:   CHARACTER 9;
RECORD NAME IS employee;
DUPLICATES ARE NOT ALLOWED FOR phones;
   home_address:  CHARACTER 50;
   office:        CHARACTER  8;
   phones:        CHARACTER  7;
   salary:        FLOAT;
   dependents:    FIXED 10;
RECORD NAME IS support_staff;
   full_time:  CHARACTER 1;
RECORD NAME IS faculty;
   rank:    CHARACTER  9.
   tenure:  CHARACTER  1.
RECORD NAME IS link1;
RECORD NAME IS student;
RECORD NAME IS graduate;
RECORD NAME IS undergraduate;
   gpa:   FLOAT.
   year:  FIXED 1.
RECORD NAME IS course;
DUPLICATES ARE NOT ALLOWED FOR title, semester;
   title:     CHARACTER 10.
   semester:  CHARACTER  6.
   credits:   FIXED 1;
RECORD NAME IS department;
DUPLICATES ARE NOT ALLOWED FOR name;
   name:  CHARACTER 20;
RECORD NAME IS enrollment;
   grade:  FLOAT;


SET NAME IS system_person;
OWNER IS system;
MEMBER IS person;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;


SET NAME IS person_employee;
OWNER IS person;
MEMBER IS employee;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;
```

Figure 5.1 The Functional Schema of the University Database Transformed to a Network Schema

```
SET NAME IS supervisor;
OWNER IS employee;
MEMBER IS support_staff;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;


SET NAME IS employee_support_staff;
OWNER IS employee;
MEMBER IS support_staff;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;

SET NAME IS teaching;
OWNER IS faculty;
MEMBER IS link1;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS taught_by;
OWNER IS course;
MEMBER IS link1;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS taught_by;
OWNER IS course;
MEMBER IS link1;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS dept;
OWNER IS department;
MEMBER IS faculty;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS employee_faculty;
OWNER IS employee;
MEMBER IS faculty;
INSERTION IS AUTOMATIC
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;


SET NAME IS advisor;
OWNER IS faculty;
MEMBER IS student;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;
```

Figure 5.1 . (cont'd.)

```
SET NAME IS major;
OWNER IS department;
MEMBER IS student;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS enrollments;
OWNER IS student;
MEMBER IS enrollment;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS person_student;
OWNER IS person;
MEMBER IS student;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;

SET NAME IS advisory_committee;
OWNER IS graduate;
MEMBER IS faculty;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS student_graduate;
OWNER IS student;
MEMBER IS graduate;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;

SET NAME IS student_undergraduate;
OWNER IS student;
MEMBER IS undergraduate;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;

SET NAME IS deptmt;
OWNER IS department;
MEMBER IS course;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS system_course;
OWNER IS system;
MEMBER IS course;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;
```

Figure 5.1 . (cont'd.)

```
SET NAME IS head;
OWNER IS faculty;
MEMBER IS department;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS system_department;
OWNER IS system;
MEMBER IS department;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;

SET NAME IS class;
OWNER IS course;
MEMBER IS enrollment;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS system_enrollment;
OWNER IS system;
MEMBER IS enrollment;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;
```

Figure 5.1 . (cont'd.)

```
TYPE entityXX IS
   ENTITY
      functionXX1:  function_type;
      functionXX2:  function_type;
      .
      .
      .
      functionXXn:  function_type
   END ENTITY
```

Figure 5.2 Entity Type Declaration.

created whose name is the single-value function name. The owner and the ancestor of the set type is the record type declared for the range entity type, and the set member is the record type declared for the domain entity type.

Multi-valued functions are defined over entities and return sets of entities. When applied to an entity or an entity subtype, a multi-valued function returns zero or more data values, where each of these values is of the same data type as the functions range type [Ref. 8]. A multi-valued function represents either a one-to-many relationship or a many-to-many relationship as defined below.

A many-to-many relationship of a multi-valued function exists in the case where entity A has a multi-valued function with entity B declared as the range entity type. Additionally, entity B must also have a multi-valued function with entity A as the range entity type. In order to determine whether or not this situation exists, for each multi-valued function of an entity LIL traverses the network database's list of entities and searches for a separate entity that has been declared the range entity type of the multi-valued function of the first entity type; should a match be found, the matched entity is checked to determine if it has any multi-valued functions (fn_set != 0) associated with it and whether or not its multi-valued function declare the first entity type as the range entity type. If the above conditions are satisfied, indicating a many-to-many relationship for the multi-valued function, a new record type is defined with its name being LINK_X, where X is an integer representing the numerical standing of this particular many-to-many relationship. Additionally, two set types are declared -- one each with the record type for the two respective entity types as the set owner and the LINK_X record as the set member.

A one-to-many relationship exists when a multi-valued function is determined not to have a many-to-many relationship. In this case a set type is defined with the record type of the domain entity as the set owner, and its range entity record type as the set member.

In order to properly illustrate the transformation process of a functional entity and its associated properties Figure &entexamp is presented. This figure shows a functional entity taken from the University database schema of Figure 2.1, and in its network representation following the application of the transformation.

B. ENTITY SUB-TYPES

The entity subtypes of the functional database are pointed to by the ***edn_subptr*** field of the gsn_sub_node structure which is depicted in Figure 4.10. As long as this field is active (not equal to NULL), there are entity subtypes that must be transformed into network structures. As is the case in the entity type transformation, LIL must also concern itself with the functions associated with the entity subtype. Figure 5.4

Functional

```
TYPE course IS
   ENTITY
      title:          STRING (1..10);
      dept:           department;
      semester:       semester_name;
      credits:        INTEGER;
      taught_by:      SET OF faculty;
   END ENTITY;
```

Network

```
RECORD NAME IS course;
DUPLICATES ARE NOT ALLOWED FOR title, semester;
   title                ; CHARACTER 10.
   semester             ; CHARACTER 6.
   credits              ; FIXED 1.

SET NAME IS system_course;
OWNER IS system;
MEMBER IS course;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;

SET NAME IS deptmt;
OWNER IS department;
MEMBER IS course;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;

SET NAME IS taught_by;
OWNER IS course;
MEMBER IS link1;
INSERTION IS MANUAL;
RETENTION IS OPTIONAL;
SET SELECTION IS BY APPLICATION;
```

Figure 5.3 A functional entity type and its network representation.

shows the form of an entity subtype declaration, where ***subtype YY*** is the unique name of the subtype and ***supertypeAA*** is a list of one or more entity types and subtypes that are supertypes or ***ancestors*** of subtypeYY.

Each entity subtype is declared as a record type with the record name being identical to that of the entity subtype. A set type is also declared with its name being the concatenation of the subtypes entity supertype, an underscore (_), and the subtypes

```
SUBTYPE subtypeYY IS supertypeAA
   ENTITY
      functionYY1:  function_type
      functionYY2:  function_type
      .
      .
      .
      functionYYn:  function_type
   END ENTITY
```

Figure 5.4 Entity Subtype Declaration.

name. The subtypes entity supertype is pointed to by the ***gsn_entptr*** field of the gsn_sub_node structure. The set member is the particular entity subtype (gsn_nsn_name), and the set owner is the subtypes entity supertype. The functions associated with an entity subtype are transformed as previously described for the functions defined on the entity types. An example of the transformation of a functional entity subtype to the equivalent network structures is shown in Figure 5.5.

C. NON-ENTITY TYPES

Non-entity types are represented by those functional schema statements that declare data types other than entities and functions. The non-entity types of Daplex are:

(1) strings

(2) scalars

   (a) integers

   (b) floating-points

   (c) enumeration (including Boolean)

(3) numeric constants

These non-entity types form a rich set of tools that allow the user to provide semantically meaningful names to data types and to limit the range of values that may be assumed by a particular data type [Ref. 8: page 34]. Non-entity types have corresponding counter parts in programming languages such as Pascal and Ada.

The transformation of the Daplex non-entity types impacts upon the attributes of network records, where these records have been transformed from functional entity types or subtypes. The task is to maintain the integrity constraints of the non-entity

```
SUBTYPE employee IS person
   ENTITY
      home_address:        STRING ( 1..50);
      office:              STRING (1..8);
      phones:              SET OF STRING (1..7);
      salary:              FLOAT;
      dependents:          INTEGER RANGE 0..10;
   END ENTITY;


RECORD IS employee
DUPLICATES ARE NOT ALLOWED FOR phones;
   home_address          : CHARACTER 50;
   office                : CHARACTER 8;
   phones                : CHARACTER 7;
   salary                : FLOAT.
   dependents            : FIXED 10.


SET NAME IS person_employee;
OWNER IS system;
MEMBER IS employee;
INSERTION IS AUTOMATIC;
RETENTION IS FIXED;
SET SELECTION IS BY APPLICATION;
```

Figure 5.5 A functional entity subtype and its network representation.

types as they are mapped into the network data types. These data types are characters, integers, and floating-points. The mapping of the non-entity types is based on determining the Daplex data type by implementing the "switch" facility of *C*. The source of the switch is the ***fn_type*** field of the function_node shown in Figure 4.14. The targets of the switch are the ***nan_type*** and ***nan_length*** fields of the nan_attr_node structure depicted in Figure 4.7. The mapping is conducted as shown below:

(1) The Daplex string data type (fn_type = 's') maps directly into network characters (nan_type = 'c'). The length of the type is set by making ***nan_length*** equal to the value of ***fn_total_length.***

(2) The Daplex floating-point (fn_type = 'f') maps directly to network floating (nan_type = 'f').

(3) The Daplex integer is mapped directly into a network integer.

(4) Daplex enumeration types are mapped into network characters with the length of the character string (nan_length) set equal to the length of the longest of the enumeration types.

The goal of the non-entity mappings is achieved by the aforementioned algorithm, thus preventing the network user from destroying the integrity of the functional schema.

D. UNIQUENESS CONSTRAINTS

Daplex utilizes uniqueness constraints in order to identify a collection of functions whose values are unique across all database entities belonging to a particular entity type or subtype [Ref. 23: page 72]. Uniqueness constraints conform to the following representation in a functional schema declaration:

UNIQUE A,B,C WITHIN D

A,B,C represents a list of one or more functions declared for the entity type ***D.*** The values of the list of functions, when combined, uniquely identify the specified entity type or subtype. MLDS identifies a uniqueness constraint by setting the value of the ***fn_unique*** field of the function_node, which is shown in Figure 4.14. A uniqueness constraint is mapped directly into the network schema by adhering to the following algorithm:

(1) locate the record type that has been transformed from the specified entity type or subtype by traversing the ***ent_node*** or ***sub_node*** fields and comparing names.

(2) locate the attribute type, ***nattr_node,*** of the record type located in step (1).

(3) set the ***nan_dup_flag*** of the attribute located in step (2), indicating that DUPLICATES ARE NOT ALLOWED.

The algorithm is implemented as a loop following the declaration and subsequent transformation of the entity types, subtypes, and non-entity types.

An example of a functional uniqueness constraint mapped into its network equivalent can be seen in Figure 5.3. One should note the declared uniqueness of ***title*** and ***semester.*** This constraint is transformed into the CODASYL-DML statement "DUPLICATES ARE NOT ALLOWED FOR title, semester".

E. OVERLAPPING CONSTRAINTS

Functional subtypes are assumed to be disjoint unless an overlapping constraint has been declared, specifying otherwise. Basically, the notion of overlapping constraints is used to indicate whether or not an entity can belong to more than one terminal entity subtype within a hierarchy. Overlapping constraints are represented in the functional schema in the following manner:

OVERLAP E,F WITH G,H;

E,F and *G,H* are lists of one or more entity subtypes. The overlap constraint specifies that data items of an entity subtype of the class E or F may also belong to an entity subtype of the class G or H. The implementation of the overlapping constraint is through the use of an overlap table which verifies the existence of such a constraint prior to allowing the addition of a record to the database. The specifics of the overlap table are given in the following chapter.

F. SET TYPES

Network set types were described in Chapter II of this thesis. The functional data model does not have a structural equivalent for the set type, however, the network set type plays a vital role in the database transformation scheme. Earlier in this chapter the role of the set was discussed in the mapping of entity types and subtypes. The details of the set implementation include the ***insertion***, ***retention***, and ***selection*** rules.

The set is represented in the network language interface of MLDS by the ***nset_node*** structure of Figure 4.4 and the specifics of fully defining a set are described below:

(1) With the exception of sets declared from the transformation of single or multi-valued functions, the set name is defined as the owner record type name (***nrn_name*** field of the nrec_node, Figure 4.6), followed by an underscore (_), followed by the member record type name. For example if ***employee*** is the owner record type and ***faculty*** is the member record type, then the set name is of the form:

SET NAME IS employee_faculty

(2) The set owner and set member name, ***nsn_owner_name*** and ***nsn_member_name*** respectively, are declared as the corresponding record type name. Continuing with the example from (1) above, ***employee*** is the owner record type and will be declared the set owner while ***faculty*** is the member record type and is declared the set member as shown below:

OWNER IS employee
MEMBER IS faculty

(3) When a set is defined in the schema it is given an ***insertion status***. Each record type that has been transformed from an entity type or subtype is required to belong to a particular set and therefore the insertion mode of the set is always ***automatic***, indicating that whenever a member record is created, it is automatically inserted into the corresponding set. The assignment of the automatic insertion mode is shown below:

nset_node-> nsn_insert_mode = InAutMode;

(4) Set types declared from the transformation of functions applied to entity types or subtypes, however, are not required to be inserted and the insert mode is therefore ***optional***, with the assignment as shown:

nset_node-> nsn_insert_mode = InOptMode;

(5) There are three separate rules governing the ***retention*** mode of sets depending upon the basis of the set declaration:

(a) A set type that is owned by SYSTEM can never allow its member record types to change owners, therefore its retention is always ***fixed***, ensuring that records connected to the set occurrence, remain in the set occurrence.

```
nset_node-> retent_mode = RetFixMode;
```

(b) A member record type transformed from an entity subtype always belongs to the same owner record type and its retention mode is also fixed.

(c) The set types resulting from the mapping of single- or multi-valued functions must allow their member record types to be deleted, modified, or reattached and thus their retention mode is ***optional***, allowing the member records to be disconnected, connected or reconnected.

```
nset_node-> nsn_retent_mode = RetOptMode.
```

(6) When a record is to be inserted into a set type , the set must be the current of the set type. Therefore, set selection is always ***by application***:

```
nset_node-> select_mode = SelAppMode.
```

The above algorithm for mapping into network set types supports both set type declarations used in Daplex: set types reflecting an ISA relationship between two entity types or subtypes, and the set types representation of a Daplex function.

VI. TRANSLATION OF CODASYL-DML STATEMENTS TO ABDL REQUESTS

Having presented an algorithm for the transformation of a functional schema into a network schema, we are now ready to discuss the mapping of CODASYL-DML statements into ABDL requests that will be able to accurately carry out the equivalent operations on an AB(functional) database.

The DML translation takes place in the Kernel Mapping System (KMS), the second module in MLDS. KMS is called from the language interface layer (LIL) when LIL receives CODASYL-DML requests from the user. The two functions of KMS are: (1) parse the user's CODASYL-DML request to validate the syntax, and (2) map the request to an equivalent ABDL request. As previously stated, in the MLDS network interface we restrict ourselves to the following subset of CODASYL-DML statements: FIND, GET, STORE, CONNECT, DISCONNECT, ERASE, MODIFY.

This chapter discusses each of the above statements and the required mapping process. Generally speaking, the mapping process is to be somewhat similar to the mapping that was presented by Wortherly [Ref. 3] with the modifications described by Rodeck [Ref. 2], and with further modifications as implemented in this work. Additionally, we give our rationale for building onto KMS of the original MLDS network interface as implemented by Emdi [Ref. 19] rather than developing an entirely new module.

A. OVERVIEW OF THE DESIGN

The second component of a database model is the ***data manipulation language*** (DML). DML is a vocabulary for describing the processing of the database. A ***procedural*** DML is a language for describing action to be performed on the database. It obtains a desired result by specifying operations to be performed. CODASYL-DML statements are procedural, [Ref. 12: pages 191-192]. As one may surmise, a data-model transformation is virtually useless without an accurate and efficient DML translation that allows the user to perform the desired operations on the target database. It is with this thought that the DML translation proceeded.

Most CODASYL-DML operations are executed in two phases: first, a FIND command is issued to identify a record, and then a second CODASYL-DML command

is issued to perform an operation. This section will briefly describe the format and intent of each of the pertinent CODASYL-DML statements, as well as give the translation algorithm for these statements.

B. MAPPING CODASYL-DML FIND STATEMENTS

The FIND statement is logically required before each of the major CODASYL-DML statements, except for the STORE statement. When a user issues a FIND command, a record is found, and it is placed in the currency indicator table (CIT). The format of the FIND statement is:

FIND record_selection_expression [ ],

while the general format of the ABDL RETRIEVE statement is:

RETRIEVE Query Target-list [by attributes]

Each of the preceding formats is presented using the following conventions: upper-case notation represents literals, lower-case represents user-supplied variable names, and square brackets contain optional clauses. As discussed in Chapter II, the FIND statement has several variants, and we will, in turn, present each of these.

1. The FIND ANY Statement

The FIND ANY statement locates a specified record of type whose value for the specified data items are equal to those in that record's template in the user work area (UWA). The syntax of the statement is:

FIND ANY record_type_x USING item_1, ..., item_n IN record_type_x

KMS, in mapping the FIND ANY statement, must use the ABDL RETRIEVE statement and form a query whose first predicate is (FILE = record_type_x). KMS then forms the additional predicates by locating the values of the relevant data items in the record-template. The request is then executed with the results being placed in the result buffer (RB). Following the request execution, KMS creates the target list consisting of the requested records attributes. Thus, the ABDL translation of the the CODASYL-DML statement is:

```
RETRIEVE ((FILE:  record_type_x) AND
   (item_1 =  value_1) AND
                  .
                  .
                  .
   (item_n =  value_n))
   (all attributes) [by record_type_x]
```

The translated request is then forwarded to KC for execution.

The following example taken from the University database illustrates the mapping of the FIND ANY statement. The requirement is to find any ***course*** record whose ***title*** is 'Advanced Database'. The CODASYL-DML procedure is:

MOVE 'Advanced Database' TO ***title*** IN ***course***

FIND ANY ***course*** USING ***title*** IN ***course***

It should be noted that the MOVE statement is an assignment statement found in the host COBOL language and in the above transaction it serves to initialize the UWA field ***title*** in ***course.*** KMS would make the following translation and actions:

(1) 'Advanced Database' is placed in the ***course*** template of the UWA for the attribute ***title.***

(2) A RETRIEVE request is formed:

```
RETRIEVE ((FILE = course) AND
     (title = 'Advanced Database))
     title, dept, semester, credits)
     BY course
```

(3) Pass the request to KC for execution.

The result is that the ***course*** record satisfying the search criteria are placed in RB.

2. The FIND CURRENT Statement

The FIND CURRENT statement causes an update of CIT by changing the current of the run-unit from its present value to the value of the database key of the current record of a specified set type. The statement is of use when we want to begin a search at the current of a particular set, which requires that the current of the run-unit be updated to agree with it. The syntax of the FIND CURRENT statement is :

FIND CURRENT record_type_x WITHIN set_type_y

The ***only*** function of this statement is to update CIT, and therefore it is a relatively simple task for KMS to handle as there is no direct mapping to an ABDL statement. An example taken from the University database illustrates the use of the FIND CURRENT statement:

FIND CURRENT ***student*** WITHIN ***person_student***

KMS would pass the CIT update information to KC for execution, and where CIT is actually updated. The current of run-unit becomes the current ***student*** record occurrence of the current ***person_student*** set occurrence.

3. The FIND DUPLICATE WITHIN Statement

The FIND DUPLICATE WITHIN statement is used to sequentially access records within a particular set occurrence. A basic assumption is that the requested

records have previously been located by another FIND and are therefore already resident in RB. The statement then locates the first record with the current set occurrence whose values for the listed items match those of the current record of the set. The syntax of the FIND DUPLICATE WITHIN is:

FIND DUPLICATE WITHIN set_type_x USING

item_1, ... item_n IN record_type_y

The translation actions are as listed below:

(1) KMS forwards set_type_x, record_type_y, and item_1,..., item_n to KC.

(2) KC locates the relevant RB using the information from (1) above.

(3) Each record with RB is searched until the first duplicate record with the set is found.

(4) The record is made available to the user.

Additionally, KC will update CIT following the accessing of each record presented to the user.

4. The Find FIRST/LAST/NEXT/PRIOR Statements

This subsection presents several related variants of the FIND statement; they identify a record by its position in a set. For instance, the FIND FIRST statement locates the ***first*** record of a set occurrence, the ***FIND LAST*** statement locates the ***last*** record of a set occurrence, and so on. Each of these statements is mapped in the same manner, and therefore we will focus the translation explanation on the FIND FIRST statement. The syntax for the FIND FIRST statement is:

FIND FIRST record_type_x WITHIN set_type_y

First of all, KMS ensures that the specified record type is a member of the specified set occurrence. This is accomplished by checking the ***nsn_set_member_name*** field of the ***nset_node*** data structure of Figure 4.4. Once the set membership is verified, KMS forms a RETRIEVE request that places every member record of the set occurrence into its RB. The request is satisfied by returning the first record.

In the case of FIND NEXT and FIND PRIOR, the set occurrence must have previously been retrieved and placed into RB. KMS must simply check CIT and determine the current of the set and return either the ***next*** or the ***prior*** record. Recalling the two types of sets in the functional data model, ISA relationships and Daplex functions, we have devised two methods for accessing all members of a particular set occurrence.

The first method is for retrieving members of a set type reflecting an ISA relationship where the set name consists of the owner name, followed by "_", followed by the member record name. KMS generates the following ABDL request:

```
RETRIEVE ((FILE = record_type_x) AND
   (MEMBER, set_type_y = set_type_x.owner.dbkey))
   (all attributes)
```

As an example, suppose we query the University database in order locate students majoring in 'Computer Science'. The CODASYL-DML transaction reads:

```
MOVE 'Computer Science' TO major IN student
FIND ANY student USING major IN student
MOVE 'NO' TO EOF
FIND FIRST person WITHIN person_student
PERFORM UNTIL EOF = 'YES'
   GET student
   FIND NEXT student WITHIN person_student
END PERFORM
```

In response to the above CODASYL-DML sequence KMS would issue the following ABDL request:

```
RETRIEVE ((FILE = person) AND
   (MEMBER, person_student = dbkey of 'CS'))
   (all attributes) [by major]
```

In the case of a set representing a Daplex function, there are two possibilities: either the function belongs to the owner record type or the function belongs to the member record type. In order to determine which record type a particular function belongs to KMS must traverse the functional schema to check the required function. If the Daplex function belongs to a owner record type the translation is as described in the previous paragraph. However, if the Daplex function belongs to a member record the translation is altered as follows:

```
RETRIEVE ((FILE = record_type_x) AND
   (set_type_y = CIT.set_type_y.owner.dbkey))
   (all attributes)
```

By definition, the set type representing a Daplex function belonging to a member record type has only one member--the member record occurrence that we are seeking.

5. **The FIND OWNER Statement**

The FIND OWNER statement identifies records by ownership and causes the owner of the current of set type to be returned. The syntax of the FIND OWNER statement is:

FIND OWNER WITHIN set_type_x. Since all of the necessary information is already present in CIT, the mapping is simple. KMS extracts the set owner and database key for the specified set and issues a RETRIEVE of the form:

```
RETRIEVE ((FILE = CIT.set_type_x.owner) AND
   (CIT.set_type_x.owner = CIT.set_type.dbkey))
   (all attributes)
```

KC then executes the RETRIEVE request and returns the owner record-type.

6. **The FIND WITHIN CURRENT Statement**

The FIND WITHIN CURRENT statement causes a record which is the current of the specified set type whose values match the specified values of UWA for the specified record type. The syntax of the statement is:

```
FIND record_type_x WITHIN set_type_y CURRENT
   USING item_1, ..., item_n IN record_type_x
```

The FIND WITHIN CURRENT is very similar to the FIND DUPLICATE statement, the difference being that FIND WITHIN CURRENT uses the values resident in UWA while FIND DUPLICATE uses the value of the current set type. Once it is determined that the specified record is a member of the set KMS generates a RETRIEVE request of the form:

```
RETRIEVE ((FILE = record_type_x) AND
   (record_type_x = CIT.set_type_y.owner.dbkey) AND
   (item_1 = user_value_1)AND
                .
                .
                .
   (item_n = user_value_n)
   (all attributes)
```

KMS then passes the request to KC for execution and the records satisfying the retrieval are placed in RB with the first record being returned to the user.

C. MAPPING CODASYL-DML GET STATEMENTS

CODASYL-DML GET statements are data retrieval statements, but they can only access records that have been previously located by FIND statements. It is the GET statement that actually allows the user to access a record for the purpose of displaying it. As was done in the network interface, the GET statements are handled through KC rather than mapping them directly into ABDL RETRIEVES. There are three options with the GET statement and they will be discussed in the following subsections.

1. The GET Statement

The GET option places the entire current record of the run-unit into UWA for user access. When KMS receives the GET statement it informs KC that the record in RB containing records of the type ***CIT.run_unit.type*** is to be passed to the user via UWA.

2. The GET record_type Statement

The GET record_type statement is similar to the GET option in that it retrieves the current record for the user, however, this option allows the user to specify a particular record type. In this instance, KMS checks to ensure that the record type being accessed is in the current of the run-unit RB, and if so, ***all*** data items are returned to the user.

3. The GET item_1, ..., item_n Statement

This statement differs from the previous GET options in that the user specifies the data items which are to be returned for a particular record. The syntax for this option is:

GET item_1, ..., item_n IN record_type_n

Again, KMS checks to ensure that the specified record type is resident in the RB containing the current of the run-unit, then the specified data items are used as search criteria to locate a matching record. If KMS is successful in locating a record, KMS informs KC and KC places the ***desired*** data items in UWA.

D. MAPPING CODASYL-DML CONNECT STATEMENTS

The CONNECT Statement manually inserts the current record of the fun-unit into the current occurrence of the specified set(s). The use of this statement requires the record to be a member of the specified set(s) and that the set(s) have an insertion clause of ***manual.*** The syntax of the CONNECT statement is:

CONNECT record_type_x TO set_type_1, ..., set_type_n

There are several ways that the CONNECT statement operates on an AB(functional) record and these could result in varying results as follows: adding information to an existing AB(functional) record, creating a new AB(functional) record, or creating a new set of AB(functional) records. The particular operation depends on the manner in which the network set types were declared in the transformation from the functional schema. Recalling the transformation algorithm of Chapter V, we know that set types represent either an ISA relationship or a Daplex function. The insertion of information into set types representing a Daplex function is further complicated depending on whether the information is to be inserted into an owner record of the set or a member record of the set.

1. **Sets Representing an ISA Relationship**

As described in Section F of Chapter V, each network record type that has been transformed from an entity type or subtype represents a functional ISA relationship. These record types are required to belong to a particular set and therefore the insertion mode of the set is always ***automatic.*** This indicates that whenever a member record is created during the transformation, it is automatically inserted into the corresponding set. Therefore, sets with an insertion clause of automatic cannot be used in CONNECT statements.

2. **Sets Representing Daplex Functions**

The destination of the information that is to be inserted will be in either an owner record or a member record type of the set occurrence. This location determines the method of translating the CONNECT statement. Each of these methods is discussed in the ensuing sections.

a. Information Resides in Owner Record

When the specified record type is the owner of the set type, the set can be null or it can contain one or more members. If the set type is null, then there are no member records associated with it. If the set type is representing a scalar multi-valued function, then there may be more than one member record associated with the set. We can see that there are four cases that must be considered when applying the CONNECT statement when the information resides in the set type owner. The situation depends on whether or not the set representing a Daplex function is null or not, and also on whether or not there are scalar multi-valued functions associated with the ***original*** functional entity type or subtype.

(1) Null Set and No Scalar Multi-Valued Function--The AB(functional)record is the only record to be updated. The null value of the attribute-value pair

representing the attribute of the set type is replaced with the database key of the current of the run-unit as shown below:

```
UPDATE ((FILE = CIT.set_type_1.owner) AND
        (CIT.set_type_1.owner = CIT.set_type_1.owner_dbkey))
        (set_type_1 = CIT.run_unit.dbkey)
```

(2) Null Set and Scalar Multi-Valued Function--The null value in each AB(functional) record created because of the scalar multi-valued function must be updated. Using CIT information KMS duplicates all attribute-value pairs of the attributes that ***do not*** represent scalar multi-valued functions and updates the null value of the attribute-value pairs representing scalar multi-valued functions. The required attribute-valued pairs are retrieved with the following ABDL request:

```
RETRIEVE ((FILE = CIT.set_type_1.owner)AND
        (CIT.set_type_1.owner = CIT.set_type_1.owner.dbkey)
        (all attributes)
```

After the results of the above RETRIEVE are placed in RB, KMS traverses the functional schema and determines which attribute-value pairs represent scalar multi-valued functions. Once these pairs are identified, they are updated as shown below:

```
UPDATE ((FILE = CIT.set_type_1.owner) AND
 (CIT.set_type_1.owner = CIT.set_type_1.owner.dbkey) AND
 (attribute1 = value1)
        .
        .
        .
 (set_type_1 = CIT.run_unit.dbkey)
```

(3) AB(functional) record with identical attribute-value pairs to those of the owner record, with the exception of the attribute-value pair whose attribute name is the same as the set name. This attribute is given the value of the database key of the current of the run-unit. As KMS did in (2) above, the owner record of the set type occurrence is retrieved with the results stored in RB. KMS then maps the following ABDL INSERT request:

```
INSERT ( <FILE, CIT.set_type_x.owner>,
  <CIT.set_type_x.owner, CIT.set_type_x.owner.dbkey>,
  <data item1, value1>,
        .
        .
        .
  <data item_n, value_n>.
  <set_type_x, CIT.run_unit.dbkey> )
```

(4) record representing the scalar multi-valued function that posseses the database key of the set owner. However, the attribute whose name is the same as the set type is assigned the value of the dbkey of the current of the run-unit. This is accomplished by retrieving the AB(functional) record representing the set owner. After the attribute-value pairs representing scalar multi-valued

functions are retrieved, they are used to retrieve the relevant records. Each record in RB will have a new attribute-value pair inserted in it whose values are the same as those in RB, except for the attribute whose name corresponds to the set type member; this value becomes the database key of the current of the run-unit:

```
INSERT (< FILE, CIT.set_type_x.owner >,
  < CIT.set_type_x.owner, CIT.set_type_x.owner.dbkey >,
  < data_item_1, value_1 >,
        .
        .
        .
  < data_item_n, value_n >,
  < set_type_x, CIT.run_unit.dbkey > )
```

b. Information in Member Record

The mapping of the CONNECT statement applied to member record is much less complex then when applied to an owner record. Again KMS must ensure that the record type is a member of the specified set and that the insertion clause of the set is ***manual.*** However, the existence of scalar multi-valued functions is irrelevant because we will update all records whose database key is the same as the database key of the current of the run-unit. This is due to the transformation algorithm specifying the set membership requirements.

The attribute of the attribute-value pair whose attribute name is the same as the set name is updated to equal the value of the database key of the set owner. The ABDL request is:

```
UPDATE ((FILE = record_type_x) AND
  (record_type_x = CIT.run_unit.dbkey))
  (set_type_y = CIT.set_type_y.owner.dbkey)
```

KMS then passes the request to KC where it is executed.

E. MAPPING CODASYL-DML DISCONNECT STATEMENTS

The DISCONNECT statement is the opposite of the CONNECT statement in that it disconnects the current record of the run-unit from the specified set type(s). Once disconnected, the records are simply detached from the set type(s) and they remain in the database. The syntax of the DISCONNECT statement is:

```
DISCONNECT record_type_x FROM set_type_1, ..., set_type_n
```

The requirements for the statement are that the current of the run-unit be a member of the specified set types(s) and that the record be removed from the set types that are current.

The DISCONNECT statement is similar the CONNECT statement in that in that it has several possible results, dependent on whether the function information is contained in the set owner or set member record. However, the key is whether the function set is a singleton, or whether it has multiple members. The DISCONNECT statement could cause an attribute value to be ***nulled out***, or a single AB(functional) record could be deleted, or a set of AB(functional) records could be deleted. The rationale behind these possibilities is explained in the following paragraphs.

If the information regarding the disconnection concerns a Daplex function represented by a network set owner record, then the function set is either a singleton or it contains multiple members. If the function set is a singleton we want KMS to null out the value of the attribute whose name is identical to the set type name. KMS generates the following ABDL request:

```
UPDATE((FILE = CIT.set_type_y.owner) AND
   CIT.set_type_y.owner = CIT.set_type_y.owner.dbkey)
   set_type_y = NULL)
```

If the above request is applied to a the representation of a scalar multi-valued function, ***all*** of the relevant AB(functional) records will be updated to reflect the null value. Otherwise a single AB(functional) record will have a value nulled out.

If the function set has multiple members KMS deletes all of the AB(functional) records with matching database key and function value. The mapping is as shown below:

```
DELETE ((FILE = CIT.set_type_y.owner) AND
   (CIT.set_type_y.owner = CIT.set_type_y.owner.dbkey) AND
   (set_type_y = CIT.run_unit.db_key))
```

Again, the above would delete ***all*** of the matching AB(functional) records if a scalar multi-valued function is part of the owner record type.

If the AB(functional) record to be deleted is a member record, then, by definition of the schema transformation, we are updating a singleton function set. KMS will null out the value of the applicable attribute as indicated in the following ABDL request:

```
UPDATE ((FILE = record_type_x) AND
   (record_type_x = CIT.run_unit.dbkey) AND
   (set_type_y = CIT.set_type.owner.dbkey))
   (set_type_y = NULL)
```

Prior to mapping the MODIFY statement it should be noted that the CONNECT and DISCONNECT statements are used to modify attribute-values representing functions in the AB(functional) database. In order to perform these modifications the attributes are disconnected from the set type occurrence, modified, and then reconnected to the set type occurrence.

F. MAPPING CODASYL-DML MODIFY STATEMENT

The MODIFY statement either alters the entire current record of the run-unit or it modifies specific data items in a the current record. The syntax of the MODIFY statement updating an entire record is:

MODIFY record_type_x The syntax of the MODIFY statement to alter specific data items of the current record of the run-unit is:

MODIFY item_1, ..., item_n IN record_type_x In each of the aforementioned instances, the data items that are to be modified must be supplied by the user. KMS will then retrieve these data items from the UWA of the specified record and map the following ABDL request:

```
UPDATE ((FILE = record_type_x) AND
   (record_type_x = CIT.run_unit.dbkey))
   (data_item_i = user_value_i)
```

The above UPDATE request is repeated for each field of the record that is to be modified. The only change to the UPDATE would be reflected in the individual data items.

G. MAPPING CODASYL-DML STORE STATEMENTS

The STORE statement creates a new record occurrence and establishes it as the current of the run-unit. Prior to inserting the record, however, it is constructed by having its field values stored in UWA. The syntax of the STORE statement is:

STORE record_type_x The key factors in mapping the STORE statement are:

(1) Set selection status.

(2) Insertion clause.

(3) Duplicate condition.

As defined in the schema transformation algorithm, the set selection status is always BY APPLICATION. Additionally, the STORE statement requires that the insertion clause of the pertinent set types be AUTOMATIC. Furthermore, the interface checks the ***dup_flag*** field of the nattr_node of Figure 4.7 to determine if any of the data items

of the record being inserted has a DUPLICATES NOT ALLOWED clause assigned to it. Should it be determined that one or more fields of the record have the clause associated with it, a RETRIEVE request is formed to see whether or not a duplicate record already exists in the database. Thus, the mapping of the STORE statement consists of an INSERT request to store the request and possibly a RETRIEVE request to determine the status of duplicates.

Once the above requirements are met KMS must ascertain the status of Daplex imposed overlap constraints. As discused in Chapter V, the Overlap Table maintains a list of which set types representing functional subtypes have overlap constraints declared. It is essential that the overlap status be verified in order to maintain the integrity of the database. The mapping of the STORE statement then proceeds with KMS verifying the duplicate status. If data items have been designated DUPLICATES NOT ALLOWED the following ABDL request if formed with the results being placed in UWA:

```
RETRIEVE ((FILE = record_type_x) AND
   (data_item_i = user_value_i))
   (record_type_x)
```

Next KMS forms an INSERT request:

```
INSERT ( < FILE, record_type_x > , < record_type_x, *** > ,
   < data_item_1, user_value_1 > ,
                .
                .
                .
   < set_type_y, CIT.set_type_y.owner.dbkey > )
```

The data items values are user supplied and retrieved via UWA.

H. MAPPING CODASYL-DML ERASE STATEMENTS

The ERASE statement deletes records from the database. When mapping this statement it is imperative that we consider the constraints imposed by the rules of CODASYL-DML as well as those imposed by Daplex. The CODASYL-DML limitation is that the record(s) to be deleted cannot be an owner of a non-null set type occurrence.

In examining the Daplex requirements we must evaluate the Daplex equivalent of the CODASYL-DML ERASE statement, the DESTROY statement. The DESTROY

statement is used to remove entities from the database. If the entity type that is being deleted has any entity subtypes in its hierarchy, then these subtypes are also deleted; the entire hierarchy of the entity type is deleted. However, there is a significant factor that comes into play when processing the DESTROY statement. If the entity being deleted is referenced by a database function, then the DESTROY statement is aborted. The ERASE statement has two options, the ERASE ALL option and the ERASE option. The two options are presented in the following subsections.

1. **The ERASE Option**

The ERASE statement without the ALL option deletes only one record from the database, the current of the run-unit. Its syntax is:

```
ERASE record_type_x
```

Recalling the CODASYL-DML constraint, we realize that KMS must form a RETRIEVE request to determine if there are any sets whose members are connected to the specified record. This is accomplished by checking to see if there are any set type occurrences where the owner database key is the database key of the current of the run-unit. In order to meet both the CODASYL-DML and Daplex imposed constraints, KMS must form two separate RETRIEVE requests for each ERASE statement:

(1) Retrieve all set occurrences where the current of the run-unit is the owner.

(2) Retrieve all set occurrences where the current of the run-unit is a member.

The ABDL translation being:

```
RETRIEVE ((FILE = CIT.set_type_y.member) AND
   (set_type_y = CIT.run_unit.dbkey))
   (set_type_y)
```

If the above request places any set types in RB then the ERASE statement does not satisfy the CODASYL-DML constraints and it is aborted. If RB is empty then KMS forms the next ABDL request:

```
RETRIEVE ((FILE =CIT.set_type_y.owner) AND
   (set_type_y = CIT.run_unit.dbkey))
   (set_type_y)
```

If this request results in an empty RB then the Daplex constraints were satisfied and KMS continues mapping the ERASE statement as follows:

```
DELETE ((FILE = record_type_x) AND
   (record_type_x = CIT.run_unit.dbkey))
```

In mapping the ERASE option KMS always issues the first RETRIEVE request for execution by KC. The results of the first request will determine whether or not the two remaining requests are issued or if the ERASE transaction is aborted.

2. **The ERASE ALL Option**

The second option of the ERASE statement is the ERASE ALL option. It deletes every record in the hierarchy of the current of the run-unit. the syntax of the ERASE ALL statement is:

ERASE ALL record_type_x

In this instance the constraints imposed by CODASYL-DML clash with those imposed by Daplex because of the requirements explained above and therefore the statement is not translated in this implementation. It should be noted that the lack of an ERASE ALL option is not considered to critical because the same effect can be obtained by the repeated use of separate ERASE statement, if the constraints are met.

VII. CONCLUSIONS

As previously mentioned, the conventional approach to the design and implementation database management systems (DBMS) has been based upon the premise of a single data model with its model-based data language. This methodology restricted a DBMS to transactions solely on the specified model and in the specified data language, resulting in the proliferation of single-model, single-language systems with limited flexibility and extensibility. The obvious need for increased efficiency and portability in DBMS has highlighted the requirement for a system that can support databases based on the five major data models using the respective model-based data languages, specifically: functional/Daplex, hierarchical/DL/I, relational/SQL, network/CODASYL-DML, and attribute-based/ABDL. Hence, the Multi-Lingual Database System (MLDS) has evolved, allowing a user to access and interact with numerous databases based on various data models via their corresponding data languages.

While MLDS allows the user to access databases based on the five major data models using their respective data languages, this thesis has presented the partial implementation of a first step toward making MLDS a truly Multi-Model Database System (MMDS). The primary goal of this work is to access a functional database via CODASYL-DML transactions, achieving interaction across the artificial boundaries of data models that the conventional approach to DBMS has yet to cross.

A. A REVIEW OF OUR WORK

We have fully implemented a language interface layer (LIL) that is based on the LIL of the network interface of MLDS as implemented by Emdi [Ref. 19]. The difference, however, is that the LIL of this thesis allows the user to access a database that is based on either the network data model or the functional data model. If the desired database is based on the network data model, then the user inputs his transactions using the data model-based data language, CODASYL-DML. On the other hand, if the desired database is based on the functional data model, LIL transforms the functional schema into a network schema and the user is then allowed to access this transformed database using CODASYL-DML transactions.

The kernel mapping subsystem (KMS) should be modified as described in Chapter VI of this work in order to allow the CODASYL-DML transactions to properly manipulate the AB(functional) database that has become the target database. KMS translates the CODASYL-DML transactions to their equivalent ABDL transactions somewhat differently from the translation designed by Wortherly [Ref. 3] and implemented by Emdi [Ref. 19], due to the fact that the target database is an attribute-based representation of a functional database rather than an attribute-based representation of a network database.

The kernel controller subsystem (KCS) was not implemented as a part of this thesis work. This was due to the uncovering of a problem in January 1987 during the integration of MLDS with the Multi-Backend Database System (MBDS). This problem prevented the connection of KCS to the kernel database system (KDS) and would not have permitted the actual test and evaluation of KCS. Although KCS was not implemented, it was examined and thought to entail only minimal changes to the existing KCS of the network interface of MLDS. The modifications are similar to those described by Rodeck [Ref. 2].

B. FUTURE RESEARCH

Rodeck's design [Ref. 2] and the work completed in this thesis present a bright picture for the emergence of MMDS. It is anticipated that the unfinished work from this thesis will eventually be completed. The remaining work is to implement the translation schema of the CODASYL-DML statements as described in Chapter VI, which entails altering the existing KMS and KC of the network interface of MLDS. Once finished we will have created a complete and full interface allowing the accessing of a functional database via CODASYL-DML transactions.

Along with this interface, the Laboratory for Database Systems Research, Naval Postgraduate School, Monterey, California is continuing to examine other interfaces that should lead to further breakthroughs. Current work includes that of Zawis [Ref. 24], which implements a means for accessing a hierarchical database via SQL transactions. It is expected that the ongoing research and development effort will ultimately result in a comprehensive MMDS.

LIST OF REFERENCES

1. Demurjian, S.A., *The Multi-Lingual Database System*, Doctoral Dissertation, The Ohio State University, December 1986.

2. Rodeck, B.D., *Accessing and Updating Functional Databases Using CODASYL-DML*, Masters Thesis, Naval Postgraduate School, Monterey, California, June 1986.

3. Wortherly, C.R., *The Design and Analysis of a Network Interface for the Multi-Lingual Database System*, Masters Thesis, Naval Postgraduate School, Monterey, California, December 1985.

4. Hsiao, D.K., "New Database Systems," *Computer Science in the Naval Postgraduate School*, pp. 11-14, September 1986.

5. Canaday, R.E., et al., "A Back-end Computer for Data Base Management," *Communications of the ACM*, Vol. 17, No. 10, October 1974.

6. Naval Postgraduate School Report NPS52-86-011, *The Multi-Lingual Database System*, by S.A. Demurjian and D.K. Hsiao, February 1986.

7. Sibley, E.H. and Kershberg, L., "Data abstraction views and updates in RIGEL," *Proc. ACM SIGMOD AFIPS*, Nat. Computer Conference, Dallas, Texas, June 1977.

8. Shipman, D.W., "The Functional Data Model and the Data Language DAPLEX," *ACM Transactions on Database Systems*, Vol. 6, No. 1, March 1981.

9. Chen, Peter Pin-Shan, "The Entity-Relationship Model--Toward a Unified View of Data," *ACM Transactions of Database Systems*, Vol. 1, pp. 9-36, March 1976.

10. Date, C.J., *An Introduction to Database Systems*, Vol. 1, Addison-Wesley Publishing Company, 1986.

11. Tsichritzis, D.C. and Lochovsky, F.H., *Data Models*, pp. 119-147, Prentice-Hall, 1982.

12. Kroenke, David, *Database Processing*, Second Edition, Science Research Associates, Inc., 1983.

13. Emdi, B., *The Implementation of a Network Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

14. Hsiao, D.K. and Haray, F., "A Formal System for Information Retrieval from Files," *Communications of the ACM*, V.13, No. 2, February 1970; *Corrigenda*, Vol. 13, No. 3, March 1970.

15. Wong, E., and Chiang, T.C. "Canonical Structure in Attribute Based File Organization," *Communications of the ACM*, September 1971.

16. Rothnie, J.B. Jr., "Attribute Based File Organization in a Paged Memory Environment," *Communications of the ACM*, September 1971.

17. Banerjee, J. and Hsiao, D.K., The Ohio State University Technical Report No. OSU-CISRC-TR-77-7, *A Methodology for Supporting Existing CODASYL Databases with New Database Machines*, by J. Banerjee and D.K. Hsiao, November 1977.

18. Lim, B.H., *The Implementation of a Functional Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1986.

19. Emdi, B., *The Implementation of a Network Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

20. Goisman, P.L., *The Design and Analysis of a complete Entity-Relationship Interface for the Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

21. Anthony, J.A. and Billings, A.J., *The Implementation of a Complete Entity-Relationship Interface for the Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1986.

22. Meyer, G. and MacDougal, P., *An Attribute-Value Translation of CODASYL's Data Manipulation Language*, Ohio State University, 1982.

23. Computer Corporation of America, Cambridge, Massachusetts, Technical Report CCA-84-01, *Daplex User's Manual*, S. Fox et al., June 1984.

24. Zawis, J.A., *Accessing Hierarchical Databased via SQL Transactions in the Multi-Model Database System*, Masters Thesis, Naval Postgraduate School, Monterey, California, (to be published December, 1987).

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	2
5.	Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
7.	Professor Steven A. Demurjian Computer Science and Engineering Department The University of Connecticut 260 Glenbrook Road Storrs, CT 06268	1
8.	Lieutenant Harry Coker, Jr., USN Code R620 Defense Communications Engineering Center Reston, VA 22090-5500	3
9.	Beng Hok Lim 507, Bedok North Ave 3 #10-347, Singapore 1646 Republic of Singapore	1

Thesis
C53133 Coker
c.1 Accessing a functional
database via CODASYL-DML
transactions.