DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-6002

# NAVAL POSTGRADUATE SCHOOL
Monterey, California

# THESIS

ACCESSING AND UPDATING FUNCTIONAL
DATABASES USING CODASYL-DML

by

Brian D. Rodeck

June 1986

Thesis Advisor: David K. Hsiao

Approved for Public Release; Distribution is Unlimited

T232479

NCLASSIFIED
RITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

| EPORT SECURITY CLASSIFICATION<br>NCLASSIFIED | 1b. RESTRICTIVE MARKINGS | |
|---|---|---|
| ECURITY CLASSIFICATION AUTHORITY | 3 DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for Public Release;<br>Distribution is Unlimited | |
| ECLASSIFICATION/DOWNGRADING SCHEDULE | | |
| RFORMING ORGANIZATION REPORT NUMBER(S) | 5. MONITORING ORGANIZATION REPORT NUMBER(S) | |
| AME OF PERFORMING ORGANIZATION<br>al Postgraduate School | 6b. OFFICE SYMBOL *(If applicable)*<br>Code 52 | 7a. NAME OF MONITORING ORGANIZATION<br>Naval Postgraduate School |
| DDRESS *(City, State, and ZIP Code)*<br>terey, California 93943-5000 | | 7b. ADDRESS *(City, State, and ZIP Code)*<br>Monterey, California 93943-5000 |
| AME OF FUNDING/SPONSORING RGANIZATION | 8b. OFFICE SYMBOL *(If applicable)* | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| DDRESS *(City, State, and ZIP Code)* | | 10. SOURCE OF FUNDING NUMBERS |

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO | WORK UNIT ACCESSION NO |
|---|---|---|---|
| | | | |

TLE *(Include Security Classification)*
CESSING AND UPDATING FUNCTIONAL DATABASES USING CODASYL-DML

RSONAL AUTHOR(S)
deck, Brian D.

| YPE OF REPORT | 13b TIME COVERED | 14. DATE OF REPORT *(Year, Month, Day)* | 15 PAGE COUNT |
|---|---|---|---|
| ter's Thesis | FROM ______ TO ______ | 1986, June | 127 |

PPLEMENTARY NOTATION

| COSATI CODES | | | 18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)* |
|---|---|---|---|
| ELD | GROUP | SUB-GROUP | Data Model Transformation; DML Mapping; Schema Transformation; Multi-model Database System; Muti-lingual Database System; |
| | | | |
| | | | |

STRACT *(Continue on reverse if necessary and identify by block number)*


Traditional approaches to database system design and implementation olve single-model, single-language database systems with their inherent k of flexibility and extensibility. An alternative to the traditional roach to database system design and implementation is the multi-lingual abase system (MLDS). This approach allows the user to access and ate one or many databases in different data models using corresponding a languages, thus countering the aforementioned flexibility and ensibility restrictions.

In this thesis, we present a methodology for accessing and updating abases stored in one model with the data manipulation facilities of a ferent data model. Specifically, we design an interface for allowing


| STRIBUTION/AVAILABILITY OF ABSTRACT<br>NCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified | |
|---|---|---|
| AME OF RESPONSIBLE INDIVIDUAL<br>f. David K. Hsiao | 22b. TELEPHONE *(Include Area Code)*<br>(408) 646-2253 | 22c. OFFICE SYMBOL<br>Code 52Hq |

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

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

#18 - KEY WORDS - (CONTINUED)

Multi-backend Database System; CODASYL-DML

#19 - ABSTRACT - (CONTINUED)

the network/CODASYL-DML user to access and update a functional database as supported by MLDS. This is the first step in the process of extending the multi-lingual database system to a true multi-model database system (MMDS).

Approved for public release; distribution is unlimited

# Accessing and Updating Functional Databases Using CODASYL-DML

by

Brian D. Rodeck
Captain, United States Marine Corps
B.S., Iowa State University, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1986

## ABSTRACT

Traditional approaches to database system design and implementation involve single-model, single-language database systems with their inherent lack of flexibility and extensibility. An alternative to the traditional approach to database system design and implementation is the multi-lingual database system (MLDS). This approach allows the user to access and update one or many databases in different data models using corresponding data languages, thus countering the aforementioned flexibility and extensibility restrictions.

In this thesis, we present a methodology for accessing and updating databases stored in one model with the data manipulation facilities of a different data model. Specifically, we design an interface for allowing the network/CODASYL-DML user to access and update a functional database as supported by MLDS. This is the first step in the process of extending the multi-lingual database system to a true multi-model database system (MMDS).

TABLE OF CONTENTS

I. INTRODUCTION ---------------------------------- 10

A. THE MOTIVATION ---------------------------- 10

B. SOME BACKGROUND --------------------------- 12

1. The Multi-Lingual Database System --- 12

2. The Multi-Backend Database System --- 16

C. THE ORGANIZATION OF THE THESIS ----------- 17

II. THE DATA MODELS ------------------------------ 19

A. THE FUNCTIONAL MODEL ---------------------- 19

1. A Conceptual View of the Model ------ 20

2. The Daplex-DDL University Database Schema ------------------------------ 22

B. THE CODASYL DATA MODEL ------------------ 22

1. A Conceptual View of the Model ------ 25

2. The CODASYL Data Manipulation Language ---------------------------- 27

C. THE ATTRIBUTE-BASED DATA MODEL ---------- 31

1. A Conceptual View of the Model ------ 31

2. The Attribute-Based Data Language (ABDL) ------------------------------ 33

3. The Functional-AB(functional) Schema Mapping Process -------------- 35

III. THREE APPROACHES TO THE MAPPING FROM THE FUNCTIONAL MODEL TO THE CODASYL MODEL ------- 39

A. THE DIRECT LANGUAGE INTERFACE APPROACH -- 42

B. THE ATTRIBUTE-BASED POSTPROCESSING APPROACH ---------------------------------- 45

C. THE HIGH-LEVEL PREPROCESSING APPROACH --- 48

D. CHOOSING THE BEST APPROACH ------------- 49

IV. TRANSFORMING A FUNCTIONAL SCHEMA TO A CODASYL SCHEMA ------------------------------- 52

A. ENTITY TYPES ---------------------------- 52

B. ENTITY SUBTYPES ------------------------- 57

C. NON-ENTITY TYPES ------------------------ 58

D. UNIQUENESS CONSTRAINTS ------------------ 60

E. OVERLAP CONSTRAINTS --------------------- 61

F. SET TYPES ------------------------------- 63

G. IMPLICATIONS OF THE METHOD CHOSEN FOR SET-TYPE DECLARATIONS ------------------ 65

H. A COMPLETE MAPPING EXAMPLE ------------- 68

V. MAPPING CODASYL-DML STATEMENTS TO ABDL REQUESTS ------------------------------------ 73

A. THE NOTION OF CURRENCY ------------------ 74

B. DATA STRUCTURES NECESSARY FOR ACCURATE TRANSLATION -------------------- 75

1. The Currency Indicator Table (CIT) -- 75

2. The Request Buffer (RB) ------------- 75

C. MAPPING FIND STATEMENTS ----------------- 77

1. The FIND ANY Statement -------------- 77

2. The FIND CURRENT Statement ---------- 79

3. The FIND DUPLICATE WITHIN Statement - 80

4. The FIND FIRST Statement ------------ 81

5. The FIND OWNER Statement ------------ 87

6. The FIND WITHIN CURRENT Statement --- 88

D. MAPPING GET STATEMENTS ----------------- 88

1. The GET and GET record_type Statements ---------------------------- 89

2. The GET item1, ... ,itemn Statement - 90

E. MAPPING DATA-UPDATING STATEMENTS -------- 90

1. The CONNECT Statement --------------- 91

2. The DISCONNECT Statement ------------ 102

3. The MODIFY Statement ---------------- 106

4. The STORE Statement ----------------- 107

5. The ERASE Statement ----------------- 110

VI. CONCLUSIONS --------------------------------- 118

A. THE CONTRIBUTION OF THE RESEARCH -------- 118

B. SOME OBSERVATIONS AND INSIGHTS ---------- 120

LIST OF REFERENCES ---------------------------------- 123

INITIAL DISTRIBUTION LIST -------------------------- 126

LIST OF FIGURES


1. The Multi-Lingual Database System (MLDS) ------- 13

2. Multiple Language Interfaces for the Same KDS -- 15

3. The University Database Schema ----------------- 23

4. A CODASYL Set Occurrence ----------------------- 26

5. An Attribute-Based Record ---------------------- 32

6. The Logical AB(functional) University Database Schema ---------------------------------- 37

7. The "Real" AB(functional) University Database Schema ---------------------------------- 38

8. Block Diagram Summary of MLDS ------------------ 41

9. Direct Language Interface Approach ------------- 43

10. The AB-AB Postprocessing Approach ------------- 45

11. The High-Level Preprocessing Approach ---------- 48

12. The Representation of an Entity Type in CODASYL ---------------------------------------- 53

13. The University Database PERSON Hierarchy ------- 61

14. CODASYL Set Declaration Format ----------------- 64

15. PERSON_STUDENT Set Type Declaration ------------ 66

16. CODASYL University Database Schema Conversion -- 69

17. Information Contained in CIT ------------------- 76

18. Contents of RB After RETRIEVE ------------------ 79

19. PERSON and STUDENT Records for 'Allan Jones' --- 82

20. A Functional/AB(functional)/CODASYL Mapping Example ------------------------------- 92

21. AB(functional) Occurrence for NULL "Enrolled" Function ----------------------------- 97

22. Singleton AB(functional) "Enrolled"
Function Set ------------------------------------- 104

23. University Database Schema Fragment ------------- 112

# I. INTRODUCTION

## A. THE MOTIVATION

During the past two decades, the method by which database systems were designed and implemented was fairly standardized. The typical approach taken has been to choose a data model, specify a model-based data language, and ultimately develop a system for managing and executing the transactions written in the data language. This approach to database system development has resulted in homogeneous database systems, each of which restricts the user to a single data model and a specific model-based data language. Some examples of systems developed using this approach include IBM's Information Management System (IMS) supporting the hierarchical data model and Data Language I (DL/I), Sperry Univac's DMS-1100 which supports the network data model and the CODASYL data language, IBM's SQL/Data System, dedicated to the relational data model and the Structured English Query Language (SQL), and Computer Corporation of America's Local Data Manager (LDM), which uses the functional data model and the Daplex data language.

An unconventional approach to the problem of database management system development, referred to as the <u>multi-lingual database system</u> (MLDS), eliminates the restrictions

outlined above [Ref. 1]. MLDS is designed to give the user the ability to access and manage a large collection of databases, using several data models and their corresponding data languages. The major design goal of MLDS has been the development of a system that is accessible via a hierarchical/DL/I interface, a relational/SQL interface, a network/CODASYL interface, and an functional/Daplex interface. Thus, MLDS functions as though it were a heterogeneous collection of database systems instead of a single data model, single data language system.

Some of the advantages of MLDS are the reusability of database transactions developed on a conventional system, economy and effectiveness of hardware upgrades (since just one system is upgraded instead of a number of different systems), and its ability to support a variety of databases built around any of the well-known data models.

There is one further step that can be taken towards a more complete utilization of databases currently available with MLDS. The current version of MLDS has certain restrictions: network databases are accessable only through CODASYL-DML, hierarchical databases are accessable only through DL/I, relational databases are accessible only through SQL, and functional databases are accessible only through Daplex. A system which would remove these restrictions would have profound implications. By allowing

the databases based on different models to be accessed by data languages based on different data models, we extend our multi-lingual database system to a multi-model database system (MMDS). In this environment, a user of one data model could access and manipulate information stored in another data model. The obvious benefit of extending the multi-lingual database system to a multi-model database system is to provide true sharing of databases--an ample motivation for the effort. As a first step, in this thesis we investigate the methods which allow a CODASYL-DML user to access a functional database.

## B. SOME BACKGROUND MATERIAL

In this section, some background material for the thesis is provided. First, we give the reader an overview of the system structure and system functions of the multi-lingual database system (MLDS). Then, we introduce the reader to the architecture of the multi-backend database system (MBDS). MBDS is the underlying database system utilized by MLDS to support database transaction processing.

### 1. The Multi-Lingual Database System

The system structure of the multi-lingual database system (MLDS) is shown in Figure 1. Users issue transactions through the language interface layer (LIL) using a user-chosen data model (UDM) and written in a corresponding model-based data language (UDL). LIL then

routes the user transactions to the kernel mapping system (KMS). KMS has two tasks. First, if the user specifies that a new database is to be created, KMS transforms the UDM-database definition to an equivalent kernel-data-model-(KDM)-database definition. The KDM-database definition is then sent to the kernel controller (KC). KC sends the KDM-database definition to the kernel database system (KDS). Upon completeion, KDS notifies KC, which in turn notifies the user that the database definition has been processed and that the loading of the database may commence.

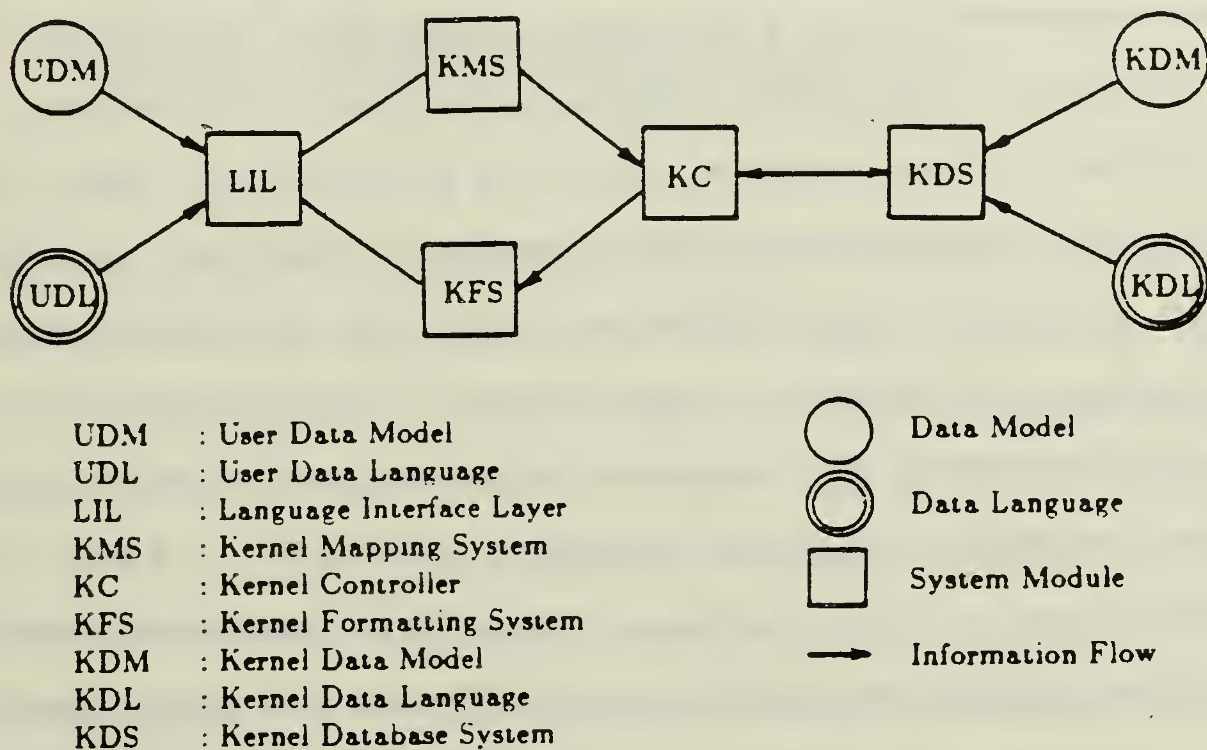



Figure 1. The Multi-Lingual Database System (MLDS)

The second task of KMS is to handle UDL transactions. In this situation, KMS translates the UDL transaction to the equivalent KDL transaction and sends it to KC, which in turn sends the KDL transaction to KDS for execution. Upon completion, KDS sends the results in KDM form back to KC. KC forwards these results to the kernel formatting system (KFS) for transforming them from the KDM form to the UDM form. After the data is transformed, KFS returns the results, i.e., the response set, to the user via LIL.

One last point must be made concerning the general system structure. Four of the five components of the multi-lingual database system, namely LIL, KMS, KC and KFS, are referred to as a <u>language interface</u>. For MLDS, a new language interface is required for each chosen data language. For example, there is a set of LIL, KMS, KC and KFS for the relational/SQL language interface, a separate set of these four components for the hierarchical/DL/I language interface, a third set of components for the network/CODASYL-DML language interface and a fourth set for the functional/Daplex language interface. KDS, on the other hand, is a single and major component that is accessed and shared by all of the various language interfaces, as depicted in Figure 2. The concept of language interfaces plays a central role in this thesis.

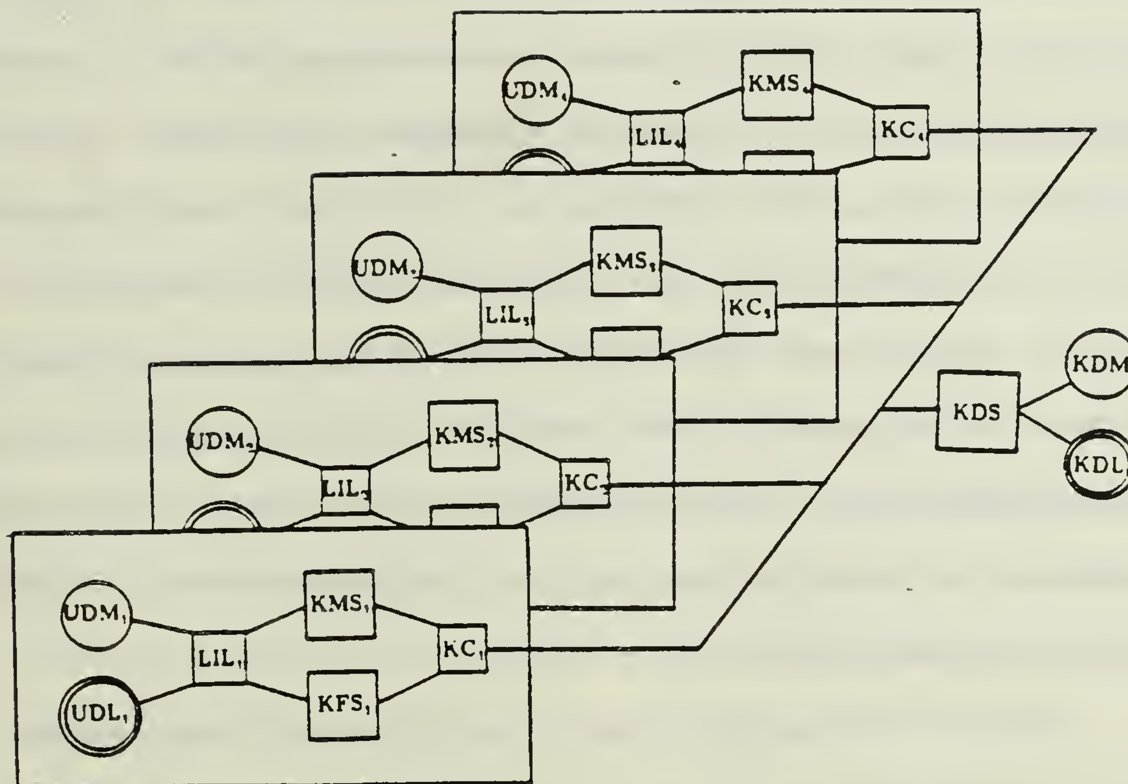



Figure 2. Multiple Language Interfaces for the Same KDS

In the preceeding discussion of MLDS, we have discussed both KDM and KDL in generic terms. In fact, the attribute-based data model and attribute-based data language (ABDL) have been chosen as KDM and KDL, respectively, for MLDS. A series of papers [Refs. 2,3,4] has shown how the relational, hierarchical, network and functional data can be transformed to attribute-based data, while at the same time presenting preliminary work on the corresponding data-language translations. In more recent work, the complete set of algorithms for the data-language translations from SQL to ABDL [Refs. 5,6], from DL/I to

ABDL [Ref. 7], from CODASYL-DML to ABDL [Ref. 8], and from Daplex to ABDL [Ref. 9] have been specified. Software development efforts for the language interfaces (i.e., one set of LIL, KMS, KFS and KC for the relational interface [Ref. 10], another set for the hierarchical interface [Ref. 11] and a third set for the network interface [Ref. 12]) have been completed. The fourth set for the functional language interface has not been completed at the time of this thesis, although the initial implementation effort has been documented [Ref. 13].

Beyond the simple fact that the implementation work has completed using the attribute-based data model, an equally important reason for choosing it and ABDL as KDM and KDL, respectively, lies in the availability of a research database system in current use at the Naval Postgraduate School Laboratory for Database Systems Research. This database system, the multi-backend database system (MBDS), uses respectively the attribute-based data model and ABDL as the native data model and data language of the system. MBDS is discussed in the following section.

2. The Multi-Backend Database System

The multi-backend database system (MBDS) has been designed to overcome the performance and upgrade problems associated with the traditional approach to database system design. This goal has been realized through the utilization of multiple backends configured in a parallel

fashion. The backends have identical hardware, replicated software, and their own disk systems. In the multi-backend configuration, there is a backend controller, which is responsible for supervising the execution of database transactions and for interfacing with the hosts and users. The backends perform the database operations with the database stored on the disk system of the backends. The controller and backends are connected by a communications bus. Users access the system through either the hosts or the controller directly.

Performance gains are realized by increasing the number of backends. If the size of the database and the size of the responses to the transactions remain constant, then MBDS produces a nearly reciprocal decrease in the response times for the user transactions when the number of backends is increased. On the other hand, if the number of backends increases proportionally with the increase in the database size and transaction responses, then MBDS produces nearly invariant response times for the same transactions. For a more detailed discussion of MBDS the reader is referred to [Refs. 14 and 15].

## C. THE ORGANIZATION OF THE THESIS

In this thesis, the first developmental work on the multi-model database system (MMDS) is described. We investigate several methods for accessing and updating a

functional database, using the network/CODASYL-DML model and language, within the context of MLDS and MBDS. Having chosen a particular method, we discuss issues which concern the schema mapping as well as the DML translation. We also present a specification for the kernel mapping system (KMS) that is to be used in the network-functional interface.

In Chapter II, we provide a description of the functional, network (i.e., CODASYL) and attribute-based (AB) data models, as well as their associated data languages. In Chapter III, we examine three approaches to mapping between functional and CODASYL databases. We then choose an approach and provide justification for the chosen approach. In Chapter IV, a methodology for mapping an fuctional schema to a CODASYL schema is presented, along with a complete mapping example. In Chapter V, we discuss the necessary data manipulation language translations between CODASYL-DML and ABDL. Finally, in Chapter VI, we make our conclusions about the proposed design.

# II. THE DATA MODELS

In this chapter, we provide a summary description of the functional, network and attribute-based data models. Only the data definition portion (DDL) of the functional model is discussed, since the approach chosen does not involve the data manipulation portion of Daplex. The other data models are summarized in their entirety.

## A. THE FUNCTIONAL DATA MODEL

The functional data model was developed by Shipman while working at MIT and CCA [Ref. 16]. The model is based on the artificial intelligence idea of the _semantic net_, a structure used to represent relationships between _objects_ or _entities_. Each entity has a corresponding set of _functions_ associated with it. Functions may provide one or more values of varying types, or may provide a connection or "arc" to other entities. These entities may be connected via functions to still other entities, and so on. Thus, an entity can be thought of as a dimensionless token whose properties are determined by functions of data values or other associated entities. One can see that information is obtained either directly through a function of the associated data value or via a composition of functions. The idea of functional composition allows one to explore

the associations within the network. This concept is crucial to the functional model.

The data definition language (DDL) for the functional model is used to define the database schema being used. We describe its constructs previously mentioned in greater detail, along with other capabilities provided in Daplex. The format of the Daplex DDL is then demonstrated by using it to define the University database, taken from Shipman's paper and the Daplex User's Manual [Ref. 17]. This schema is used extensively throughout the rest of the thesis.

### 1. A Conceptual View of the Model

The functional data model is mainly concerned with two classes of items--scalar values and entities. Scalar values are simply atomic values which have a literal representation, like "2", or "Hsiao". On the other hand, an entity has no literal value associated with it, nor is it in any way atomic. One can print only scalar-valued functions of it. For example, an entity named PERSON has no value associated with it. However, the function "name(PERSON)" denotes a specific value which may be printed out. On the other hand, the function employees_of(PERSON) may associate it with certain entities named EMPLOYEE, with no immediately printable value available.

Functions defined over entities can therefore return scalar values, entities, or sets of entities. A

distinction is made between single-valued and multi-valued functions, and again concerning whether the function returns scalar values or entities. A multi-valued function is very similar to the set type described in the network model.

Subtyping is also used in the functional model. An entity may be a subtype of another entity. Thus, the EMPLOYEE entity is a subtype of the PERSON entity, since all EMPLOYEEs are PERSONs. Subtyping establishes a relationship among entities, often referred to as an ISA relationship. As with the semantic net in artificial intelligence, implicit value inheritance goes along with subtyping of entities. For example, if a PERSON entity has two functions, name and ssn (social security number), an EMPLOYEE subtype of PERSON inherits both of these functions.

Non-entity types are also allowed in Daplex DDL. Daplex has string, integer, floating point and enumeration data types. Using these as building blocks, Daplex allows one to declare ranges of values, base types, subtypes of the base types, and derived types which inherit characteristics of a named type or subtype.

One important concept dealt with in Daplex DDL is that of overlap constraints. Referring to the discussion on subtypes and putting their definition in a slightly different light, one can see that the set of all EMPLOYEEs

is in a one-to-one correspondence with a subset of the set of all PERSONs. In a like manner, a STUDENT entity may also be defined as a subtype of PERSON. The two subtype sets (STUDENT and EMPLOYEE) may overlap or be disjoint, depending on whether a STUDENT can also be an EMPLOYEE. In Daplex, the overlap of any entity types or subtypes must be explicitly defined. All sets not specifically overlapped are assumed to be disjoint.

2. <u>The Daplex-DDL University Database Schema</u>

We can now present the University database schema as defined by Shipman. For each entity type and subtype declared, the names of the functions on them are defined. If the functions are scalar-valued, the usual data type declaration is given. If they are single-valued functions associated with another entity type or subtype, the entity name is given as the type of the function. If they are multi-valued functions mapping to another entity type or subtype, the type is declared to be "set of <entity name>". Finally, key functions are declared by "unique" clauses, and subtypes by "overlap" clauses. The University database is shown in Figure 3.

## B. THE CODASYL DATA MODEL

In general, the network (CODASYL) data model is based on the concept of directed graphs. The nodes of the graphs usually represent entity types which are records, while the arcs of the graphs correspond to relationship types that

```
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;
```

Figure 3. The University Database Schema

```
    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;

    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 3. (Continued)

are represented as connections between records. The CODASYL (Conference on Data System Languages) data model is referred to by Tsichritzis and Lochovsky [Ref. 18:pp. 119-132] as the most comprehensive specification of a network model that exists. It is important to note that MLDS uses only a subset of the entire data model as specified by CODASYL. The specific constructs and clauses used in connection with MLDS are described clearly and succinctly by Wortherly [Ref. 8]. In an effort to facilitate understandability, our description of the network data model follows his work.

1. A Conceptual View of the Model

CODASYL databases are networks of record types and set types, where records and sets are the entities which describe the databases. A record type in a CODASYL database is a collection of hierarchically related data items or field names [Ref. 19]. A record is any occurrence of a record type and has specific values assigned to the data items named in the schema declarations. This implies that a record type is simply a generic name for all of the records that are described by the same template.

Set types in a CODASYL database indicate relationships between record types. They consist of a single record type called the owner record type, and one or more record types called the member record types. Thus, a set type expresses explicit associations between different

record types in the database. This characteristic makes it possible for a designer to model a large variety of real-world database management problems involving diverse record types. It is important to note the fact that the owner record type of a set type is not allowed to be a member of the same set type.

Set types have occurrences just as record types do. Each occurrence of a set type has one occurrence of the owner record type and zero or more of each of its member record types. Again it must be noted that a record occurrence cannot be present in two different occurrences of the same set type. This qualification emphasizes the pairwise disjointness of set occurrences of a given set type. Figure 4 gives an example of a set occurrence involving an owner record occurrence and two member record occurrences.

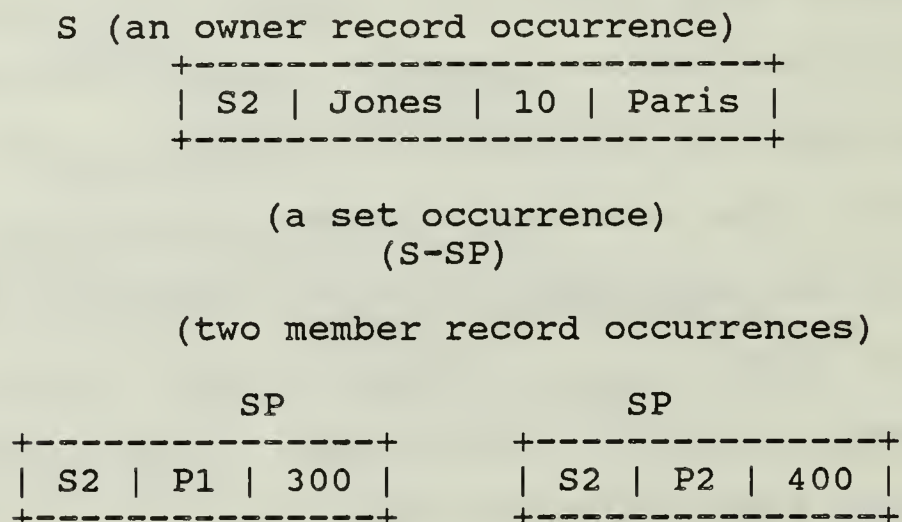



Figure 4. A CODASYL Set Occurrence

2. The CODASYL Data Manipulation Language

CODASYL-DML is a procedural language. The user of a CODASYL database writes his programs in a general-purpose language that hosts the CODASYL-DML. In general, most operations necessary in a CODASYL database are carried out by "navigating" through set occurrences. The starting point for this navigation is usually the current record of the run unit. The run unit is the application program (transaction) being executed. A full explanation of currency will be provided later in the thesis. Other DML operations can be based on the current record occurrence of a set type or record type.

CODASYL-DML has several primary operations which support the primary database operations of retrieval, insertion, deletion and modification (updating existing records). Different implementations provide varying collections of these operations, but we concentrate our discussion on the basic ones.

The most important primary operation of the CODASYL-DML is the FIND statement. This statement is used to establish the currency of the run unit, and optionally to establish the currency of the set type and the record type. The general format of the FIND statement is

FIND record-selection-expression [ ],

where the square brackets contain optional expressions for the suppression of updates to the currency indicators. In other words, we may suppress the updating of the currency for a record type, a set type, or both. The record-selection-expression has several different forms, each designed to access a particular record in three different ways: without reference to a previously accessed record; relative to a previously accessed record; or by repetition.

The GET statement in CODASYL-DML complements the FIND statement. Once a record is found, the GET statement places the record in the transaction's working area for access by the transaction. There are two basic formats for the GET statement. They include GET record_type, which gives the transaction access to the entire record, and GET items IN record_type, which gives access only to requested data items in the record type.

The STORE statement is used to place a new record occurrence into the database. The programmer must build up an image of the record prior to the store request using assignment statements which are a part of the host language in which CODASYL-DML is embedded. Once the record image has been created, the proper set occurrence for the record must be selected by the database management system.

The set occurrence in which the new record is stored is determined by the SET SELECTION clause specified in the schema definition for the object database. The

three options available are: BY APPLICATION, which means that the application program (transaction) is responsible for selecting the correct occurrence; BY VALUE, which means the system selects the proper occurrence based on data item values specific to the owner of the set occurrence desired; and BY STRUCTURAL, which means that the system selects an occurrence by locating the owner record with a specific item value equal to the value of that same item in the record being stored. The restriction on the last two options is that the data items being used must have been specified with DUPLICATES NOT ALLOWED in the schema definition.

If the user transaction desires to manually insert records into the database, two requirements exist. First, the schema definition must include the INSERTION IS MANUAL clause in the set description for this particular member record. Then the CONNECT statement is used, instead of the STORE statement, for insertion of the record into the database. The record to be inserted is the current record of the run unit. The set occurrence into which the record is to be inserted is determined in the same way as for the STORE statement.

There is also a statement in the CODASYL-DML which performs the opposite operation, namely, the manual removal of a record occurrence from a set. The DISCONNECT statement performs this operation. It disconnects the

current record of the run unit from the occurrence of the specified set that contains the record. The record occurrence still resides within the database, but is no longer a member of the specified set. There is a qualification involved with this statement, however. The record to be disconnected must have a RETENTION IS OPTIONAL clause in the member description for the set type in the schema.

In order to delete records from a CODASYL database, the ERASE statement is used. There are four basic options to this statement, but two are not used in connection with MLDS. The simplest of the two used in MLDS is the ERASE without the ALL option. This statement causes the current record of the run unit to be deleted from the database if, and only if, it is <u>not</u> the owner of a non-empty set. If it is the owner of a non-empty set, the ERASE fails.

The ERASE ALL statement causes the current record of the run unit to be deleted whether or not it is the owner of a non-empty set. Additionally, this option causes each member record of the set to be deleted, and if they are owners of non-empty sets, their members are deleted as well. This action continues all the way down the database hierarchy. As one can see, an entire database could be destroyed if the user is not careful when using this option.

The final statement included in the MLDS subset of CODASYL-DML is the MODIFY statement. It is used to modify values of data items in a record occurrence. This includes modifying all data items or any subset of the data items in the record type. It may also be used to change the membership of a record occurrence from one set occurrence to another, as long as they are of the same set type.

## C. THE ATTRIBUTE-BASED DATA MODEL

The attribute-based model was originally described by Hsiao [Ref. 20]. It is a very simple but powerful data model capable of representing many other data models without loss of information. It is this simplicity and universality that makes the attribute-based model the ideal choice as the kernel data model for the MLDS, and the attribute-based data language (ABDL) as the kernel language for the system.

### 1. A Conceptual View of the Model

The attribute-based data model is based on the notion of attributes and values for the attributes. An attribute and its associated value is therefore referred to as an attribute-value pair or keyword. These attribute-value pairs are formed from a Cartesian product of the attribute names and the domains of the values for the attributes. Using this approach, any logical concept can be represented by the attribute-based model.

A record in the attribute-based model represents a logical concept. In order to specify the concept thoroughly, keywords must be formed. A record is simply a concatenation of the resultant keywords, such that no two keywords in the record have the same attribute. Additionally, the model allows for the inclusion of textual information, called the record body, in the form of a (possibly empty) string of characters describing the record or concept. Figure 5 gives the format of an attribute-based record.

```
(<attribute1,value1>,...,
 <attributen,valuen>,
 { text })
```

Figure 5. An Attribute-based Record

The angled brackets, <,>, are used to enclose a keyword where the attribute is first followed by a comma and then the value of the attribute. The record body is then set apart by curly brackets, {,}. The record itself is identified by enclosure within parentheses. As can be seen from the above, this is quite a simple way of representing information.

In order to access the database, the attribute-based model employs an entity called predicates. A keyword predicate, or simply predicate, is a triple of the form (attribute, relational operator, value). These predicates

are then combined in disjunctive normal form to produce a query of the database. In order to satisfy a predicate, the attribute of a keyword in a record must be identical to the attribute of the predicate. Also, the relationship specified by the relational operator of the predicate must hold between the value of the predicate and the value of the keyword. A record satisfies a query if all predicates of the query are satisfied by certain keywords of the record. A query of two predicates

(FILE = PERSON) and (SSN = 123456789)

would be satisfied by any record of file PERSON whose SSN is 123456789, and it would have the form,

(<attribute1,value1>, ... ,<FILE,PERSON>, ... ,
<SSN,123456789>, ... ,<attributen,valuen>,{text}).

2. The Attribute-Based Data Language (ABDL)

The ABDL as defined by Bannerjee, Hsiao and Kerr [Ref.21] was originally developed for use with the Database Computer (DBC). This language is the kernel language used in the MLDS. The ABDL supports the five primary database operations, INSERT, DELETE, UPDATE, RETRIEVE and RETRIEVE-COMMON. Those of use to us are INSERT, DELETE, UPDATE and RETRIEVE. A user of this language issues either a request or a transaction. A request in the ABDL consists of a

primary operation with a qualification. The qualification specifies the portion of the database that is to be operated on. When two or more requests are grouped together and executed sequentially, we define this to be a transaction in the ABDL. There are four types of requests, corresponding to the four primary database operations listed above. They are referred to by the same names.

Records are inserted into the database with an INSERT request. The qualification for this request is a list of keywords and a record body. Records are removed from the database by a DELETE request. The qualification for this request is a query.

When records in the database are to be modified, the UPDATE request is utilized. There are two parts to the qualification for this request, namely, the query and the modifier. The query specifies the records to be modified, while the modifier specifies how the records are to be changed.

The final request mentioned here is the RETRIEVE request. As its name implies, it retrieves records from the database. The qualification for this request consists of a query, a target-list and an optional "by" clause. The query specifies the records to be retrieved. The target-list contains the output attributes whose values are required by the request, or it may contain an aggregate operation, i.e., AVG, COUNT, SUM, MIN or MAX on one or more

output attribute values. The by-clause is optional and is used to group records when an aggregate operation is specified.

As indicated, ABDL consists of some very simple database operations. These operations, nevertheless, are capable of supporting complex and comprehensive transactions. Thus, ABDL meets the requirement of capturing all of the primary operations of a database system, and is quite capable of filling the role of kernel data language for MLDS.

### 3. The Functional-AB(functional) Schema Mapping Process

Of great importance to our thesis is the mapping of a functional schema to an attribute-based (AB) schema. This mapping should maintain the characteristics and constraints of the functional schema within the AB schema. For the purposes of this thesis, we call such a schema an AB(functional) schema, since it is the representation of a functional schema in AB. The algorithm for the functional-AB(functional) mapping is described and implemented in previous theses [Refs. 9,13], and therefore not described in detail here. We would, however, like to provide a quick overview of the mapping process. Basically, the general structure of the mapping has an AB record type being created for each entity type or entity subtype in the functional database. Within a particular AB record type,

the attribute-value pairs are specified for the entity type or subtype name, for each function in the type or subtype, and for each type or subtype that the current type inherits.

We now demonstrate and discuss this process with an example. Of special interest to us is the schema mapping from the functional University database schema depicted in Figure 3 to its equivalent AB(functional) schema. Figure 6 shows what may be considered a logical view of the AB(functional) schema. Herein, we can determine what the relationships are between records by the recursive database key declarations. A good example of this is found in the third file declaration, for "support_staff". The database key for "support_staff" is declared as

<suuport_staff, <employee, <person,**>>>

We can therefore see that relationships exist between the "support_staff", "employee" and "person" files.

The actual schema maintained is shown in Figure 7. Here, we see that the information maintained is far less detailed. The corresponding declaration for the database key in "support_staff" is

<support_staff, integer>

Any further information required is obtained from the functional schema which is also maintained.

```
(<File, person>, <person, **>, <name, string>,
 <ssn, string = 000000000>)

(<File, employee>, <employee, <person, **>>,
 <home_address, string>, <office, string>,
 <phones, set of string>, <salary, float>,
 <dependents, integer range>)

(<File, support_staff>,
 <support_staff, <employee, <person, **>>>,
 <supervisor, <employee, <person, **>>>,
 <full_time, boolean>)

(<File, faculty>,
 <faculty, <employee, <person, **>>>,
 <rank, rank_name>, <teaching, <course, ***>>,
 <tenure, boolean = FALSE>,
 <dept, <department, ****>>)

(<File, student>,
 <student, <person, **>>,
 <advisor,<faculty,<employee,<person,**>>>>,
 <major, <department, ****>>,
 <enrollments, <set of enrollment, *****>>)

(<File, graduate>,
 <graduate, <student, <person, **>>>,
 <advisory_committee,<faculty, <employee,
     <person,**>>>>>)

(<File, undergraduate>,
 <undergraduate, <student, <person, **>>>,
 <gpa, grade_point>, <year, integer
     range 1 .. 4 :=1>)

(<File, course>,
 <course, ***>, <title, string>
 <dept, <department, ****>, <semester, semester_name>,
 <credits, integer>)

(<File, department>, <department, ****>,
 <head,<faculty,<employee,<person,**>>>>,

(<File, enrollment>, <enrollment, *****>,
 <class, <course, ***>>,
 <grade, grade_point>)
```

Figure 6. The Logical AB(functional) University Database Schema

```
(<File, person>, <person, integer>, <name, string>,
 <ssn, string = 000000000>)

(<File, employee>, <employee, integer>,
 <home_address, string>, <office, string>,
 <phones, string>, <salary, float>,  ** MULT REC'DS,
                                        PHONE SET **
 <dependents, integer>)

(<File, support_staff>,
 <support_staff, integer>,
 <supervisor, integer>,
 <full_time, integer>)

(<File, faculty>,
 <faculty, integer>,
 <rank, string>, <teaching, integer>,
 <tenure, boolean = FALSE>,
 <dept, integer>)

(<File, student>,
 <student, integer>,
 <advisor, integer>,
 <major, integer>,
 <enrollments, integer>)   ** MULT REC'DS FOR
                              MULTI-VALUED SET **

(<File, graduate>,
 <graduate, integer>,
 <advisory_committee, integer>)

(<File, undergraduate>,
 <undergraduate, integer>,
 <gpa, float>, <year, integer>)

(<File, course>,
 <course, integer>, <title, string>
 <deptmt, integer>,
 <semester, string>, <credits, integer>)

(<File, department>, <department, integer>,
 <head, integer>)

(<File, enrollment>, <enrollment, integer>,
 <class, integer>,
 <grade, float>)
```

Figure 7. The "Real" AB(functional) University Database Schme

# III. THREE APPROACHES TO THE MAPPING FROM THE FUNCTIONAL DATA MODEL TO THE CODASYL DATA MODEL

As previously mentioned, this thesis is directed toward providing the capability for the user of the network/CODASYL-DML language interface to access and/or update a database that has been defined using the functional/Daplex language interface. A typical example would be the user of the network language interface having some updates to be done for a friend, who operates on the functional language interface. This user would need to see what the other database looked like in order to operate on the database. Unfortunately, the user only understands a network schema and CODASYL-DML, so these are the only tools he can work with.

In this scenario, we must provide the network/CODASYL-DML user with a method to access the functional database. The sequence of events in this scenario would be as follows:

1. The user requests to see a specific, named database (in our case, a functional database).

2. The screen displays the database in a network schema representation.

3. Having viewed this network schema, the user composes the CODASYL-DML transaction(s) necessary to accomplish the task.

4. The DML statements are applied to the attribute-yielding the desired result.

In previous theses, we have seen the language interfaces discussed and specified of the network model and the AB(network) model, and of the functional model and the AB(functional) model [Refs. 8,9]. The basic idea is that the database itself is stored in the AB model, rather than the network or functional models. This entire process is transparent to the user. That is, the user works with the network (or functional) schema and composes DML (or Daplex) transactions to access and/or update the database. The user may not know that the DML (or Daplex) transactions are actually translated into ABDL requests. It is the language interface of the network model and AB(network) database, and of the functional model and AB(functional) database which accomplishes these tasks. Figure 8 depicts these relationships, and is used as the basis for further illustrations in this chapter.

With the context of the existing MLDS language interfaces in mind, we can return to the previous scenario. We need to provide the user with a network schema representing the target (functional) database, and we need to accept his CODASYL-DML statements and correctly apply them to the AB(functional) database. In considering the best way to accomplish the aforementioned tasks, three main strategies or approaches come to mind. The first approach, the <u>direct language interface</u> approach, involves creating a new language interface between the network model and the

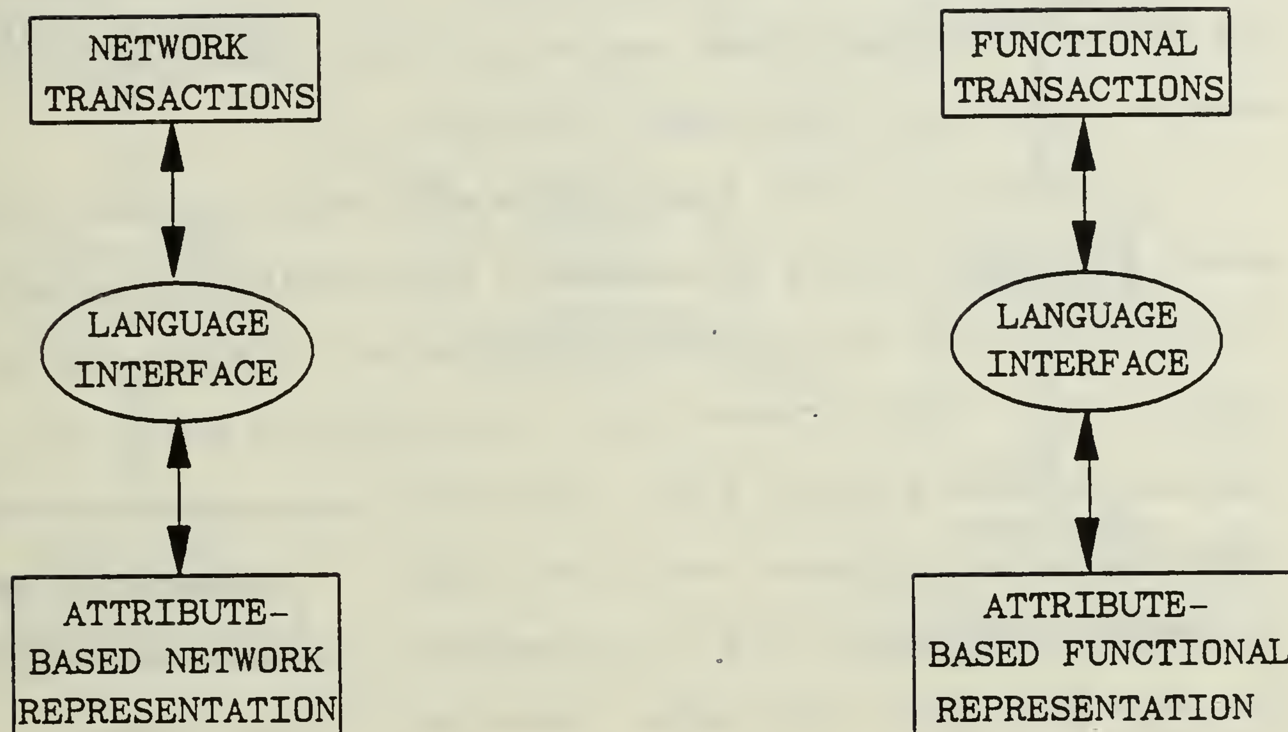



Figure 8. Block Diagram Summary of MLDS

AB(functional) database, along with a facility to translate from the functional schema to the network schema. The second approach, the AB-AB postprocessing approach, seeks to create a language interface of the AB(functional) and AB(network) databases along with a translator of the AB(functional) to network schemata. Finally, the third approach, the high-level preprocessing approach, uses the

schema translator in the first approach, and translates CODASYL-DML statements into Daplex DML statements.

In the rest of this chapter, we discuss each of the three approaches in turn. For each approach, we give advantages as well as disadvantages . Finally, we compare the approaches described and select the best one for the network-functional data model mapping.

It should be noted that there are many possible data model mappings, since relational, hierarchical, network, functional and attribute-based data models are all supported in MLDS. There are other mappings which may be implemented at a later time, for which perhaps one of the approaches not chosen would be ideal. Therefore, the approaches not chosen are also presented in some detail, in the hope that they will offer ideas for future theses.

## A. THE DIRECT LANGUAGE INTERFACE APPROACH

The direct language interface approach can be depicted as in Figure 9. Herein, we see the basic figure depicted in Figure 8 with two added modules, namely, the Schema Translator and the Direct Language Interface. The schema translator represents a process which produces a network schema representing the characteristics and constraints of the functional schema. This process is initiated at the point at which the user specifies that he is a network user and wishes to access and/or update a functional database. The direct language interface accepts

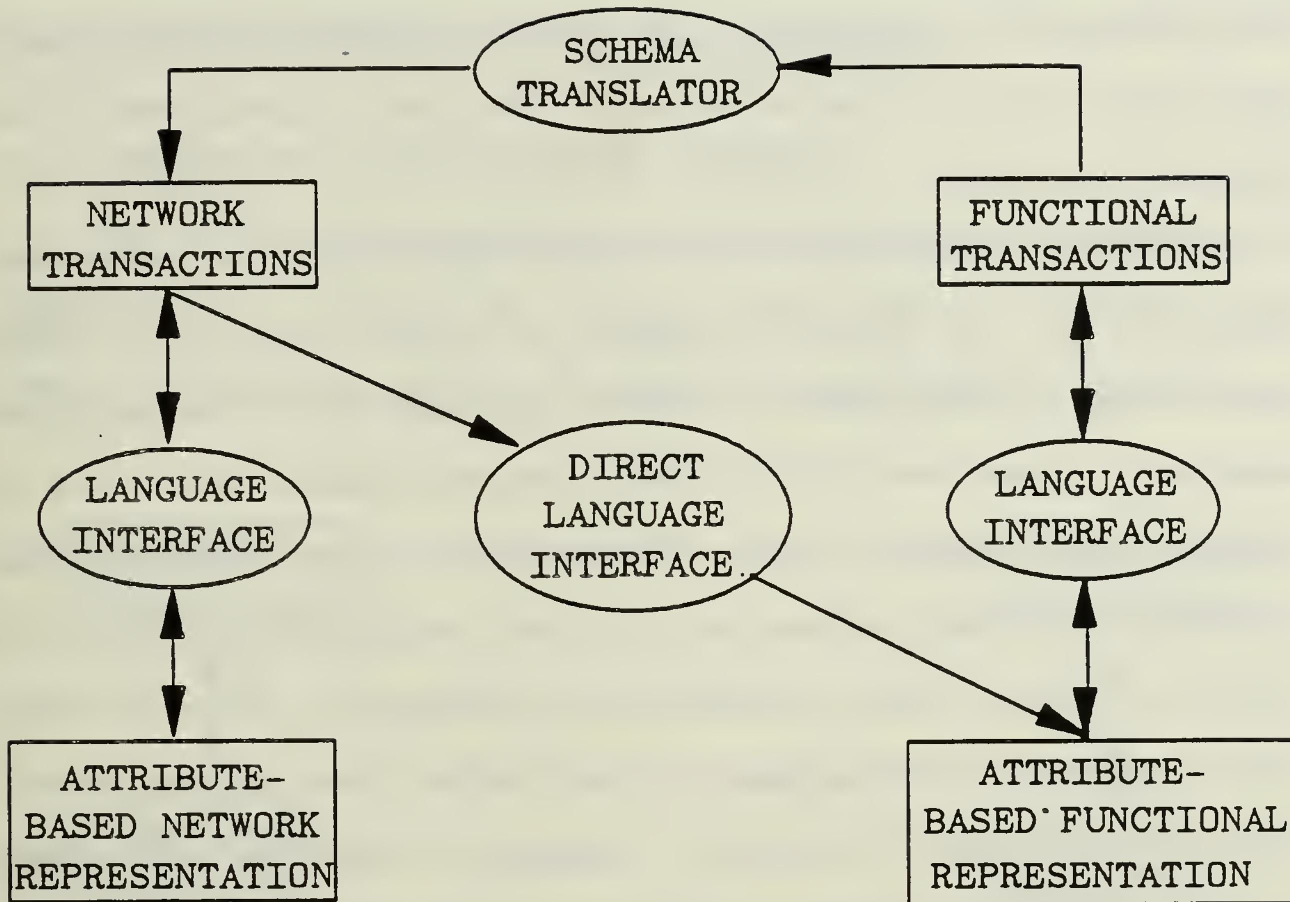



Figure 9. Direct Language Interface Approach

CODASYL-DML statements as input and maps them to attribute-based data language (ABDL) requests. One can see that, except for the schema translator, this approach is basically the same as has been taken with the network to AB(network) and functional to AB(functional) language interfaces. One can further imagine, for example, language interfaces from network to AB(network), AB(functional), AB(relational) and AB(hierarchical) databases in a fully connected, network language interface environment. Eventually, the other major data models could be connected

as well, so that one could work in any preferred data model and access any database. Of course, schema translators from each of the data models to each other would be required as well.

A major advantage to this approach is that the interface is not executed serially either before or after the other interfaces in order to achieve the desired result. As we see in the next two sections, what are in essence preprocessing and postprocessing steps are required of the other approaches before the language interface is entered. Seen from a conceptual standpoint, going through one interface would appear to be faster than going through two. Furthermore, the direct language interface would seem to take the same amount of time to execute DML commands against a functional database as is needed to execute DML commands against a CODASYL database. Finally, the schema translation would be a one-time, on-demand operation, maintained for the duration of the user session.

One disadvantage of this approach is that the AB(functional) database is not designed to accomodate the operations required in CODASYL-DML. As was discussed in a previous thesis, special attributes are included in AB records representing a network database which allow them to be easily manipulated and accessed in the network model [Ref. 8]. These attributes are non-existent in AB records representing a functional database. Consequently, AB

records of a functional database are not capable of being manipulated by means of DML transactions.

## B. THE AB-AB POSTPROCESSING APPROACH

The AB-AB postprocessing approach can be depicted as in Figure 10. In it, we see one extra module added to the basic diagram of Figure 8, namely the AB-AB Translator.

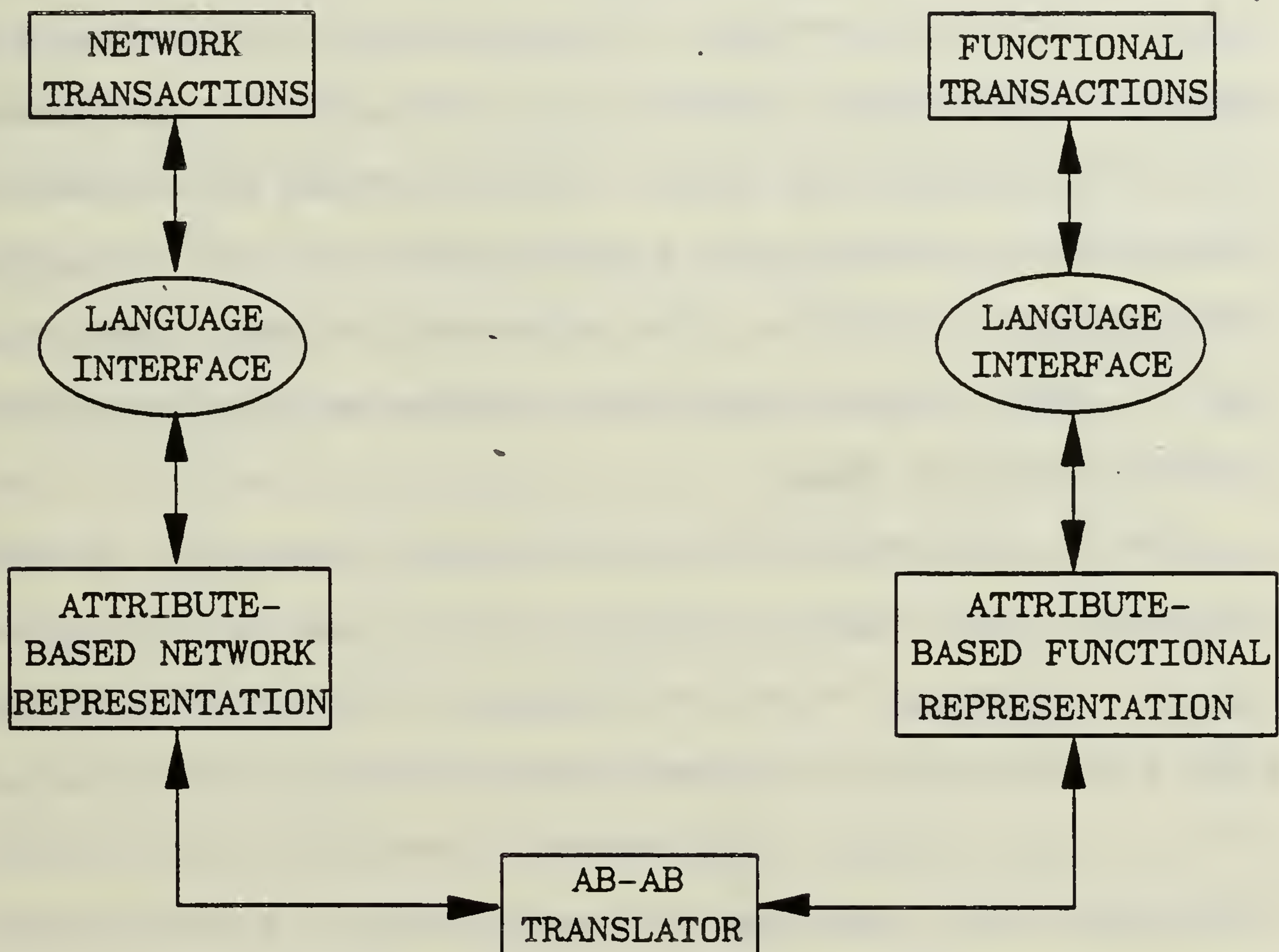



Figure 10. The AB-AB Postprocessing Approach

Given a collection of real-world data, we could define a database for it in any of the data models supported by

MLDS - hierarchical, relational, network and functional. Using algorithms and methodologies already described in previous theses, the defined databases can be converted to AB(hierarchical), AB(relational), AB(network) and AB(functional) databases. It seems reasonable to suppose that the differences between AB databases (in this case, the AB(functional) and AB(network) databases) could be determined to the extent that one could "translate" from one to another as needed. So, if a network user wished to access a functional database, he could see it by viewing the AB-AB translation of it. It also seems possible that transactions could be mapped back to the original AB(functional) database. The advantage of this approach is that it takes maximal advantage of the existing interfaces without changing them.

There are several disadvantages inherent in this approach, which tend to disqualify it from consideration for our study. One disadvantage is the fact that the module would have to be executed twice for each procedure: once to map DML statements from AB(network) to AB(functional), and once to map the results back. Thus, the approach appears to take twice as long from a conceptual standpoint.

The disadvantage in this case resulted from the fact that the AB(functional) schema cannot be mapped directly to a CODASYL schema without having ongoing access to the

functional schema. The reason for this can be demonstrated using the depictions of the AB(functional) University database found in Figures 6 and 7. Logically, we know that the schema represents the relationships found in Figure 6, but the actual schema is not stored in this manner. Instead, Figure 7 depicts the way that the schema is actually stored, and demonstrates the fact that actual relationships cannot be deduced from it. Rather, these relationships are found in the functional schema. Therefore, in the context of the functional language interface the AB(functional) schema is sufficient for its purposes, but in the context of AB(functional) and CODASYL schemata, insufficient information is available for a direct translation of them. One may make the argument that the functional-AB(functional) schema translation can somehow be redesigned such that the AB(functional) schema can be translated into a CODASYL schema, preserving the characteristics and constraints of the functional schema. While it is unclear whether this is possible, a cogent argument against this is that the new schema might well be untranslatable to hierarchical or relational schemata. Since, in the broad view, links may be forged between these data models and the functional data model, it seems ill-advised to "custom-modify" the functional-AB(functional) schema translation process.

## C. THE HIGH-LEVEL PREPROCESSING APPROACH

The high-level preprocessing approach can be depicted as in Figure 11. Again, we see that two modules have been

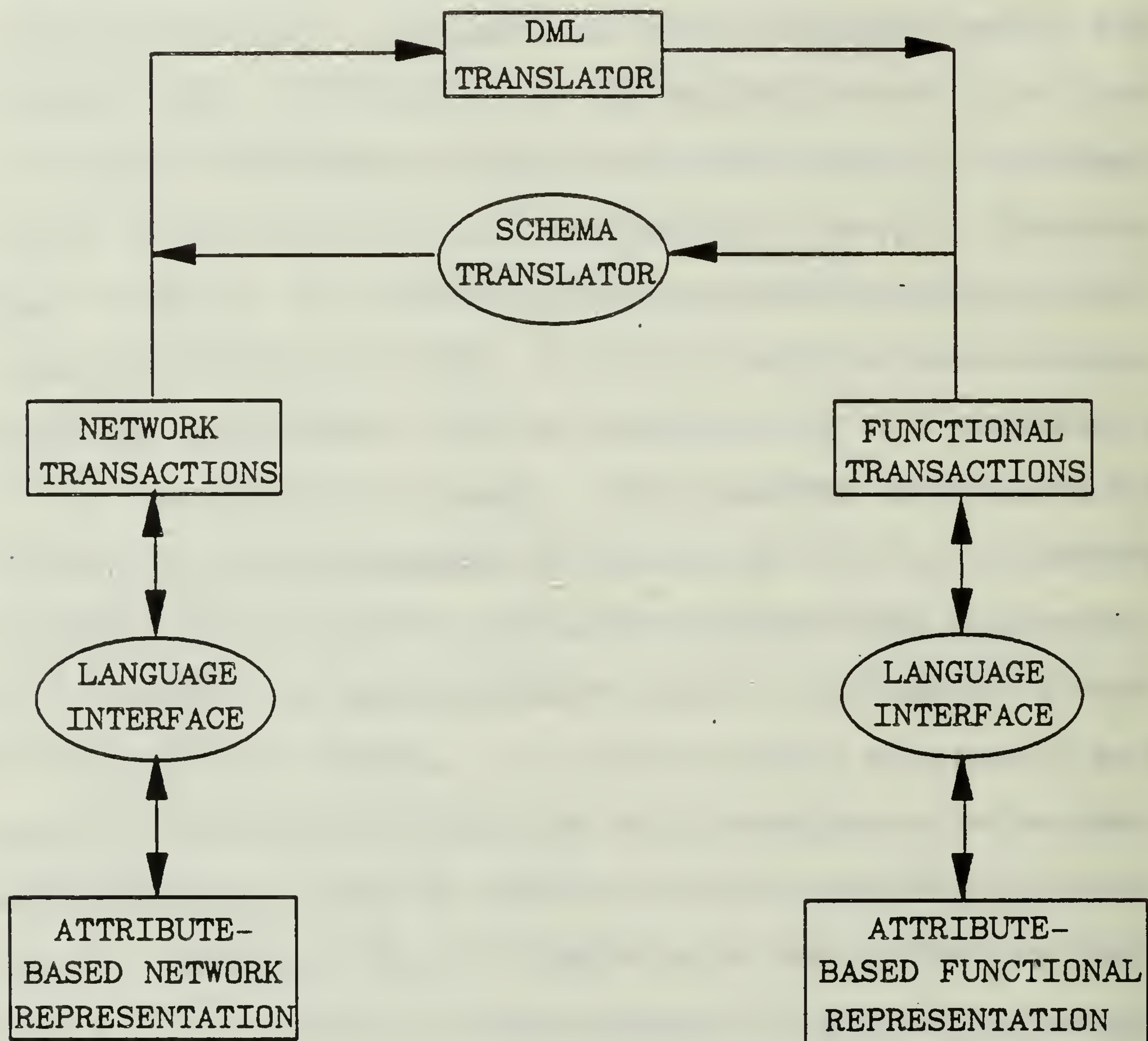



Figure 11. The High-Level Preprocessing Approach

added to the basic diagram of Figure 8, namely, a DML Translator module and a Schema Translator module. In this approach the schema translator discussed in the first approach is used to map the functional schema into an

equivalent network schema. Using this equivalent schema, the user generates CODASYL-DML transactions to access and/or update the functional database. Using the DML translator, the CODASYL-DML statements are translated into equivalent Daplex DML statements. The existing language interface of the functional model and AB(functional) database is then used to translate the Daplex DML statements.

The advantages of this approach are that it takes maximal advantage of existing tools and techniques, and is conceptually easy to envision. It also shares the schema translator needed in the first approach. The disadvantages are that a DML translator between the network and functional data models may be difficult to accomplish, and that serial processing time would have to be added directly to the existing approaches.

D. CHOOSING THE BEST APPROACH

Having disqualified an AB(functional)-CODASYL schema translator from further consideration, one of the three approaches is no longer feasible for the network-functional data model mapping. This leaves the remaining two approaches to satisfy our requirements. The direct language interface approach translates the functional schema to a CODASYL schema and allows CODASYL-DML commands to be applied against it. These DML commands are then translated directly into ABDL commands and executed against

the AB(functional) database. The high-level preprocessing approach translates the schema into CODASYL and accepts CODASYL DML commands as before, but translates them into the Daplex commands one would issue if the schema had been a functional schema. These are then mapped via the existing MLDS language interface into the appropriate ABDL commands for execution against the AB(functional) database.

In choosing between these two approaches, we note that the high-level preprocessing approach may be considered to be the "classical solution". Besides being similar to what Computer Corporation of America has done with their MULTIBASE product [Refs. 16,22], this approach is used in the Sirius-Delta project run by INRIA in France, which deals with a distributed database management system [Ref. 23].

Additionally, from an intuitive standpoint, it also seems that the high-level preprocessing approach is more time-consuming than the direct language interface approach. While both have the same schema translation process, the direct language interface approach involves a direct, one-step DML translation. The high-level preprocessing approach, on the ither hand, translates from CODASYL to Daplex, and then from Daplex to ABDL.

Because the high-level preprocessing approach has already been treated at least in principal, and because from an intuitive standpoint it seems to be slower due to a

two-step DML translation, the direct language interface approach seems preferable. We feel that this approach is also the most compatible with the existing language interface concept. The direct language interface approach is therefore our choice as the best approach.

A final point should be made at this time. While the AB(functional)-CODASYL translation turned out to be inadvisable, this may not always be the case. In fact, it may well be the recommended approach for the relational data model, which closely resembles the AB data model. This should continue to be an active area of inquiry.

# IV. TRANSFORMING A FUNCTIONAL SCHEMA TO A CODASYL SCHEMA

One can see that a central part of the chosen approach is to present the CODASYL user with a CODASYL representation of the functional schema. To do this in a time-and-space-efficient manner, MLDS must be able to automatically generate a CODASYL schema representing the current functional schema for the desired database. Such a schema must--to as great an extent as is possible--accurately reflect the characteristics of the functional schema, while preserving its constraints.

The conversion of a functional schema to a CODASYL schema revolves around six main constructs: the entity type, the entity subtype, non-entity types, the uniqueness constraint, the overlap constraint and the set type. The methodology for dealing with each of the six constructs is discussed in this chapter. We conclude the chapter with a detailed example, converting the University database schema of Figure 3 to an equivalent CODASYL schema.

## A. ENTITY TYPES

In this section, we examine the process of transforming functional entity types into an equivalent structure in CODASYL. In doing so, we note that there are two main parts to an entity declaration, namely the entity itself

and the functions associated with the entity. In order to properly illustrate the transformation process, Figure 12

functional

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

UNIQUE title,semester WITHIN course;
```

CODASYL

```
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 12. The Representation of an Entity Type in CODASYL

depicts a functional entity taken from the University database schema of Figure 3, and the CODASYL declarations necessary to represent it. Most of the tansformation process of the entity type to CODASYL are demonstrated using this example.

An entity type is declared to be a CODASYL record type. Additionally, each entity type declared must be the member of a set type which is owned by SYSTEM. This can be seen in the first two CODASYL declaration sections.

Mapping the functions associated with an entity type is a far more complicated process. In the chapter describing the various data models, we stated that functions that are defined over entities can be scalar functions, scalar multi-valued functions, single-valued functions and multi-valued functions. Each of these four types of functions is mapped differently, and is described in turn.

It must also be noted that a very important assumption is made here, namely, that each function name is unique within a functional schema.

Scalar functions are declared as fields in the previous record type which represents the entity type. Referring to Figure 12, the scalar functions in the record type declaration are "title," "semester" and "credits."

Scalar multi-valued functions could be represented by storing an array of values in the record. However, since attribute-based records do not store sets in this manner,

this is not the best representation. Rather, it is necessary to declare the scalar multi-valued function as a field in the appropriate record type. Since only one occurrence of the scalar multi-valued function can be stored in a given record, every occurrence after the first necessitates the creation of a new record. This new record is identical in every way to the one before it, with the single exception of the field in question. Therefore, we then add this field to any others necessary to uniquely define the record. These fields are declared in the record type declaration using

DUPLICATES ARE NOT ALLOWED FOR <field names>

Again, an example of this can be seen in the record type declaration in Figure 12. The issue of uniqueness is taken up in a later section.

Single-valued functions are dealt with in the following manner. A CODASYL set type is declared, whose name is the function name. Its owner is the record type declared for the range entity type, and its member is the record type declared for the domain entity type. This ensures that a single owner for the record type is defined in the set relation. Figure 12 depicts one single-valued function, namely, "deptmt."

Multi-valued functions are defined over entities and return <u>sets</u> of entities. There are two categories of

multi-valued functions to consider: One-to-Many relationships and Many-to-Many relationships.

1. One-to-Many Relationship--if the function is determined not to be many-many, declare a set type with the record type of the domain entity as the owner, and the record type of the range entity as the member.

2. Many-to-Many Relationship--occurs if entity A declares a multi-valued function with entity B as the range entity type, while entity B declares a multi-valued function with entity A as the range entity type. In this case, an extra (link) record type is defined in addition to the record types for entity A and entity B. Two set types are declared--one in which the record type for entity A is the owner and the link record is the member, and another in which the record type for entity B is the owner and the link record is the member.

Referring again to Figure 12, the function "taught_by" is a multi-valued function. We note that there is no way to tell simply from the entity type shown that a multi-valued function exists. This is true for any two entities A and B which have multi-valued functions declared of them. It can be seen that, while entity type A is being converted to the appropriate CODASYL representation, entity type B may or may not have been converted already. However, this is not a problem. Since the functional schema must first be declared, we can traverse this schema at will to check for existing many-to-many relationships. Thus, even if the appropriate record type has not yet been declared, we can declare a set type involving it. When entity type B is converted, the set type declaration has already been in

place, so checks need to be made to ensure that set duplication does not occur.

## B. ENTITY SUBTYPES

In this section, we examine the process for transforming functional entity subtypes into an equivalent structure in CODASYL. Like the entity type mapping performed in the previous section, entity subtypes can be divided into two main parts with respect to the mapping process, namely, the subtype itself and the functions associated with the entity subtype. We describe the mapping process of each in turn.

As before, an entity subtype is declared as a <u>record type</u>. Additionally, a set type is declared in which the member is the record type declared for the entity subtype, and the owner is the record type declared for the "parent" entity type or subtype. So if A is an entity type, B is an entity subtype of A and C is an entity subtype of B, we would have the following partial set declarations:

```
SET NAME IS A_B;
OWNER IS A;
MEMBER IS B;

SET NAME IS B_C;
OWNER IS B;
MEMBER IS C;
```

All functions (scalar, scalar multi-valued, single-valued and multi-valued) defined on the entity subtype are

declared in the same way as have previously been defined for entity types.

## C. NON-ENTITY TYPES

Daplex provides rather extensive facilities for declaring non-entity types. While data types are declared within a record as character, integer or floating point in CODASYL, Daplex has string, integer, floating point and enumeration data types. Using these as building blocks, Daplex declares ranges of values, base types, subtypes of the base types, and derived types which inherit characteristics of a named type or subtype.

In the conversion process described in the previous sections, it is necessary to maintain to the greatest degree possible the integrity constraints set in place by Daplex non-entity type declarations. Comparing Daplex and CODASYL data types, we find the following four transformations:

1. Daplex string maps directly to CODASYL character.

2. Where CODASYL declares an integer and a length, Daplex declares either integers or integer subranges. The straightforward solution is to declare an integer type with the length equivalent to the order of magnitude of the largest integer allowed in the subrange. This, of course, allows integer values to be input which may subsequently violate the Daplex integrity constraints specified. However, we feel it is more important to retain existing CODASYL constructs and operate within them than to make up new constructs to account for Daplex's richer dialect.

3. Daplex <u>floating point</u> maps directly to CODASYL <u>floating point</u>.

4. Daplex allows enumeration types while CODASYL does not. Since we have elected to maintain the strict bounds of CODASYL, we must declare the enumeration type to be CHARACTER, and the length to be that of the largest enumeration. This of course permits the same problems encountered with violating functional integrity constraints as occur with integer declarations.

An important point needs to be made about non-entity types in general. They are provided in Daplex to enforce data integrity constraints, so the task of any mapping is to map the <u>constraint</u> rather than the <u>construct</u>. Failing this goal affords the CODASYL user an opportunity to destroy the integrity of the functional database.

A CODASYL user can compromise the database integrity only when using DML commands which insert or update data. Therefore, appropriate data integrity checks need to be conducted at insert or update time to ensure the legality of the transaction. It should be obvious, therefore, that the generated CODASYL schema cannot maintain data integrity, but merely indicate the structure of the functional database to the CODASYL user. In order to maintain data integrity, some sort of "Integrity Table" could be created. As an example, the valid enumerations of COLOR could be stored in this table. Thereafter, if COLOR is to be inserted or updated, the value for it would have to be resident in the table. Note that the Integrity Table would not play a role in retrieve or delete commands, and

so would not need to be invoked there. Thus, the integrity checks, if desired, must be performed by C code embedded in the appropriate DML insert and update commands.

## D. UNIQUENESS CONSTRAINTS

Daplex represents uniqueness constraints in the following manner:

UNIQUE XXX,YYY WITHIN ZZZ

XXX and YYY represent one or more functions which, when taken together, uniquely describe the entity in question of entity type ZZZ. The uniqueness constraint is easily mapped to the CODASYL schema. Since uniqueness constraints are given in Daplex after entity types and subtypes are declared, their record types have already been declared in the schema conversion. Therefore, one merely locates record type ZZZ and inserts the following key declaration.

DUPLICATES ARE NOT ALLOWED FOR XXX YYY;

It should again be noted that the business of creating a true key is complicated by the attribute-based representation of the functional database. When scalar multi-valued functions are declared as fields in a record type, they must also be declared as part of the key (although an empty set may cause problems). Therefore, at a minimum, when desiring to insert the DUPLICATES NOT ALLOWED clause, the program must check to see if there are

some fields with such a clause. In this case, the new fields are merely added to the existing list.

## E. OVERLAP CONSTRAINTS

The Daplex User's Manual states that "an overlap constraint determines when an entity may legally belong to more than one terminal subtype within a hierarchy" [Ref. 17]. Overlap constraints are necessary within the functional database because by definition each subtype is assumed to be disjoint. An example of this is found in Figure 13, the PERSON generalization hierarchy for the University database. FACULTY and UNDERGRADUATE are both descendants of PERSON. Unless FACULTY and UNDERGRADUATE are declared to overlap, a person who is an undergraduate cannot at the same time be a faculty member.

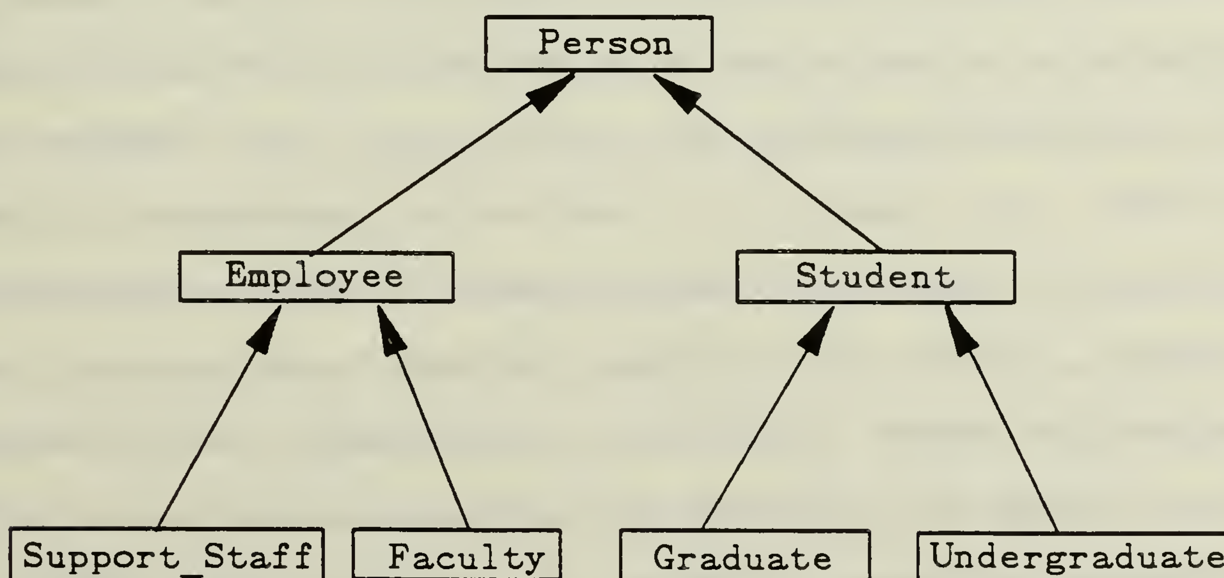



Figure 13. The University Database Person Hierarchy

The difficulty in mapping this to the CODASYL schema rests in the way that entity types and subtypes are related. UNDERGRADUATE is owned in a set type by STUDENT, which in turn is owned in a set type by PERSON. FACULTY is owned by EMPLOYEE, which again is owned by PERSON. One can see that, unless otherwise restricted, a PERSON record which has an UNDERGRADUATE descendant could later have EMPLOYEE and FACULTY records attached to it.

While it is obviously necessary to prohibit unauthorized overlaps, this capability is not available within CODASYL-DML. One cannot in effect say that one set occurrence owned by a record is allowed, but that another is not. Furthermore, one cannot specify that one set type occurrence is prohibitted because another exists. We must therefore find another means of ensuring that the proper overlap constraints are maintained.

One way to accomplish this while preserving the generality of set types is by providing an "Overlap Table". If a given record is the owner of a set (PERSON ---> STUDENT, for example), it may not be permitted to be the owner of other set occurrences (PERSON ---> EMPLOYEE) unless either an overlap is declared between each respective member record type specifically, or an overlap is declared for a descendant of each member record type. Referring to Figure 13 again, if FACULTY is overlapped with GRADUATE, then a PERSON record can own sets of EMPLOYEE and

STUDENT, even if neither FACULTY nor GRADUATE are specified. Again, within EMPLOYEE, the person can be SUPPORT_STAFF or FACULTY, but not both, due to the overlap constraint. All this must be represented in the Overlap Table.

As a summary example, if a record is to be added to the database, the Overlap Table must first be consulted to ensure a record can be added to this set. The Integrity Table may then be checked to ensure that the fields in the record contain allowable values.

## F. SET TYPES

In the following discussion, it is presumed that the reader is familiar with CODASYL set formats in general. In particular, prior theses concerning the design and implementation of a CODASYL interface have defined a restricted grammar for CODASYL set declarations in MLDS. Using angle-brackets to designate optional portions of the declaration, the CODASYL set declaration format used in MLDS is depicted in Figure 14.

In preceeding sections, we have stated that a set type must be declared. However, when doing so, the naming of the set types, insertion and retention rules, and other details differ, depending on the circumstance. When declaring set types, the following guidelines are to be maintained:

```
        SET NAME IS AAA;

        OWNER IS BBB;

        MEMBER IS CCC;

        INSERTION IS AUTOMATIC
                     MANUAL

        RETENTION IS FIXED
                     MANDATORY
                     OPTIONAL

<;

        SET SELECTION IS BY VALUE OF DDD IN EEE
                            STRUCTURAL FFF IN GGG
                            EQ HHH IN III
                            APPLICATION          >

;
```

Figure 14. CODASYL Set Declaration Format

1. Except for sets created from single- or multi-valued functions, the set name is defined as the owner record type name followed by a "_", followed by the member record type name. Single- and multi-valued functions are handled as previously described. If PERSON is the owner and EMPLOYEE is the member, then the set name declaration is:

   SET NAME IS person_employee;

2. Owner and member name declarations correspond to the respective record type names.

3. Because of the particulars of our proposed schema conversion, every record type added or modified which represents an entity type or subtype must belong to a particular set. Therefore, the insertion clause for entity types and subtypes is always:

   INSERTION IS AUTOMATIC

On the other hand, functions for a given entity type or subtype may or may not be used. This affects the declaration and maintenance of set types representing single- and multi-valued functions. Therefore, their insertion clause is always:

INSERTION IS OPTIONAL

4. Retention of a record within a given set is complicated by differing rules, depending on the cause of the set type declaration. To begin with, a set type for which the owner is SYSTEM never allows its member records to switch owners. Also, a member record reflecting an entity subtype always belongs to the same owner record type. Therefore, the retention clause in these cases is

RETENTION IS FIXED

On the other hand, those set types which are set up to describe the single- and multi-valued functions may need to have their members deleted, moved around and reattached at will. In order to do this, the retention clause must be

RETENTION IS OPTIONAL

5. When a record is inserted into a set, the set must be the current of set type. Therefore, set selection is always specified as:

SET SELECTION IS BY APPLICATION;

## G. IMPLICATIONS OF THE METHOD CHOSEN FOR SET-TYPE DECLARATIONS

In the previous section we have discussed the methodology for declaring set types. To review, there are two set type declarations: set types which reflect an ISA

relationship between two entity types or subtypes, and set types which represent a Daplex function.

The first case may be considered to be somewhat of a "standard" set type declaration. By this we mean that a record which falls into this category either owns or is owned by another record. Referring to the University database functional and CODASYL schemata depicted in Figures 3 and 15 respectively, we see that the PERSON_STUDENT set type falls into this category. STUDENT is an entity subtype of PERSON, reflecting an ISA relationship.

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

Figure 15  PERSON_STUDENT Set Type Declaration

On the other hand, the functional schema indicates three functions associated with STUDENT, namely "enrollments", "major" and "advisor". Following the methodology previously described and by viewing the CODASYL schema, we see that three set types are declared, possessing the same names as the Daplex functions. STUDENT is the owner of ENROLLMENTS, but is the member in the MAJOR

and ADVISOR set types. All three set types are of the category which represent Daplex functions.

It is important at this point to recall that the true database, as described by the functional schema, is actually in attribute-based (AB) format. The information for the ENROLLMENTS, MAJOR and ADVISOR set types is all stored in fields within the STUDENT record, in AB format. This is true, even though STUDENT is the owner in one set type but the member in the other two.

When we cause a CODASYL record to be either an owner or a member of a set type, we are merely defining a static relationship among existing occurrences of the two record types. At the AB(functional) level, however, to connect to a set type in CODASYL is to do one of two things. If the set type is one in which the record is a member, we are inputting information into a previously NULL field. On the other hand, if the set type is one in which the record is the owner, we are creating an entirely new record every time a new member is associated with it. Likewise, the DISCONNECT statement is not just the abrogation of a relationship between two records as with CODASYL records. Rather, it either clears a field in the member record, or deletes an owner record entirely, depending on whether the function being represented by the set type comes from the owner or member record. Further discussion on this topic

can be found in sections concerning CODASYL-DML translations for FIND, CONNECT and DISCONNECT.

## H. A COMPLETE MAPPING EXAMPLE

This concludes the discussion of the schema conversion methodology. In order to demonstrate the effect of this methodology, a sample schema conversion is performed. The completed conversion of the University database schema, shown in Figure 3, to an equivalent CODASYL schema is presented in Figure 16.

```
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.
```

Figure 16. CODASYL University Database Schema Conversion

```
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;

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

Figure 16. (Continued)

```
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;

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;
```

Figure 16. (Continued)

```
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;

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 16. (Continued)

# V. MAPPING CODASYL-DML STATEMENTS TO ABDL REQUESTS

Having detailed a methodology for transforming a functional schema to a CODASYL schema, we are now ready to examine the mapping of CODASYL-DML statements into ABDL requests in order to carry out the desired operations on an AB(functional) database. As previously mentioned, we restrict our attention to the subset of CODASYL-DML statements used in the MLDS network interface, namely: FIND, GET, STORE, CONNECT, DISCONNECT, ERASE and MODIFY.

In this chapter, we discuss the above statements in each of their forms, as well as the mapping process required. We present the discussions in the same manner and order as was done with the original MLDS network interface [Ref. 8], so that the similarities and differences between the two may be highlighted.

When describing CODASYL-DML statements, the following notation is used: literals are represented in the upper case, user-supplied variable names are represented in the lower case, and optional clauses are denoted with square brackets.

In order to fully explain each DML statement, the concept of currency is discussed. A description of some of the data structures which are used in the network language

interface and applicable to the CODASYL-functional mapping process follows.

## A. THE NOTION OF CURRENCY

The notion of _currency_ is very fundamental to the network data model. Currency among records and sets may be compared to placemarkers in books, showing what page you were last on in each book. Knowing the current record of a specific record type is essential to the navigation and manipulation of the network database.

For each application program running on the system, a table of _currency indicators_ is maintained. The currency indicator used is a database key, generated by the database management system to uniquely identify each record in the database. The currency indicator table (CIT) identifies the record most recently accessed by the run unit. This is done for each record type, each set type, and any other type. By any other type, we mean that the key for the record of any type most recently accessed is maintained as the _current of the run unit_, the most important of all currencies.

As an example, suppose we have set type A with record type B as the owner and record type C as the member. Suppose further that we are navigating in set type A, and the last thing that we have done has been to access a record of record type C, whose database key is D. Then the current of the run unit, the current of set type A and the

current of record type C are all D values. It can be seen that the current of set type may be either an owner record or a member record, whichever has been accessed last.

## B. DATA STRUCTURES REQUIRED FOR THE MAPPING PROCESS

In order to correctly and efficiently navigate through the database, as well as manipulate information returned as the result of data retrieval operations, two data stuctures are needed, namely, the Currency Indicator Table (CIT), and the Record Buffer (RB). In this section, each is described in turn.

### 1. The Currency Indicator Table (CIT)

As is previously mentioned, a currency indicator table (CIT) is created for each program run using the MLDS network interface. The table and its contents are instantiated upon the first call to the database management system, and are updated continually when navigating in or manipulating the database.

As can be seen from Figure 17, CIT contains entries for the current of the run unit, currents of each record type, and currents of each set type. The information shown for each is the same as that contained in the MLDS network interface CIT, and is sufficient for our purposes.

### 2. The Request Buffer

For many of the CODASYL-DML statements being translated, a series of ABDL requests may be generated

during the mapping process. Some of the requests necessarily wait for the completion of preceeding requests

```
CIT
    RUN_UNIT
         record_type
         database_key

    record_type(i)
         database_key

    set_type(i)
         boolean (is record an owner record)
         record_type
         database_key
         member_record_type
         owner_record_type
         owner_database_key
```

Figure 17. Information Contained in the CIT

in order to execute. Also, the information returned as the result of one statement translation is often used in succeeding statements. This implies the need for some sort of storage capability to hold information needed for later requests or accesses.

The request buffer (RB) is used as the storage medium for information returned by ABDL RETRIEVE requests. There must be one RB for each RETRIEVE request issued. Upon successful execution of a RETRIEVE request, all of the records satisfying the request are maintained in RB. Subsequent requests may then access this information during their execution. RB plays a central role in the mapping

process described in later sections, and will be described in greater detail therein.

## C. MAPPING THE FIND STATEMENTS TO ABDL RETRIEVES

The format of the CODASYL FIND statement is

FIND record_selection_expression [ ]

while the format of the ABDL RETRIEVE request is

RETRIEVE Query Target-list [by Attributes]

There are a number of formats for the FIND statement implemented in MLDS, and each of these is examined in turn.

### 1. The FIND ANY Statement

The FIND ANY statement syntax is as follows:

FIND ANY record_type1 USING item1,...,itemn IN record_type1

The purpose of this statement is to locate any record of type record_type1 whose values for item1 through itemn match the values placed in the record's template in the user work area (UWA).

To map this statement, the word RETRIEVE is substituted for FIND ANY. A query is then formed whose first predicate is (FILE=record_type1). The other predicates in the query are found in UWA, in the form of attribute names with values assigned to them. Since this is a RETRIEVE request, RB is needed to store the information retrieved after request execution.

Having formed the query, the target list is created, which consists of all of the attributes of the requested record. In summary, the translated CODASYL-DML statement is:

```
RETRIEVE ((FILE = record_type1) AND
          (item1 = value1) AND
            :
            :             AND
          (itemn = valuen))
          (all attributes)  [BY record_type1]
```

We note that part of the mapping process from the functional database to the AB(functional) database is that the database key is called by the same name as the record itself. Thus, the optional "BY record_type1" clause is the ordering of the records retrieved by the database key. An example using the University database demonstrates the process of the mapping. The requirement is to find any EMPLOYEE record whose office is 'SP401'. The CODASYL-DML procedure is

```
MOVE 'SP401' TO OFFICE IN EMPLOYEE
FIND ANY EMPLOYEE USING OFFICE IN EMPLOYEE
```

It should be noted that the MOVE statement is an assignment statement found in the host COBOL language. Using the procedure previously described, KMS does the following:

Step 1. 'SP401' is placed in the EMPLOYEE template of the UWA for the attribute OFFICE.

Step 2. A RETRIEVE request is formed which looks like

```
RETRIEVE ((FILE=EMPLOYEE) and
         (OFFICE='SP401'))
         (OFFICE,PHONE,SALARY,DEPENDENTS)
         BY EMPLOYEE
```

Step 3. KMS passes the request to KC for execution.

The result of the above steps is that all EMPLOYEE records satisfying the search criteria are placed in RB, sorted by the database key. Figure 18 shows the contents of RB after the retrieve is executed. When the GET statement is issued, the first record in RB is returned.

2. The FIND CURRENT Statement

The FIND CURRENT statement syntax is as follows:

```
FIND CURRENT record_type1 WITHIN set_type1
```

The FIND CURRENT statement requires no ABDL statements to be generated. The purpose of the statement is to update the current of the run unit with the value resident in the current of set_type1. In other words,

```
+-------------------------+
| <SP401,6460004,44000,2> |
| <SP401,6460049,39000,2> |
| <SP401,6460061,43000,3> |
+-------------------------+
```

Figure 18. Contents of RB After RETRIEVE

```
CIT.RUN_UNIT.type    <----  record_type1
CIT.RUN_UNIT.dbkey   <----  (dbkey of current
                             of set_type1)
```

This statement is useful when we wish to start a search at the current of set_type1, which requires that the current of the run unit be changed to agree with it.

3. <u>The FIND DUPLICATE WITHIN Statment</u>

The syntax of the FIND DUPLICATE WITHIN statement is as follows:

```
FIND DUPLICATE WITHIN set_type1 USING
item1,...,itemn IN record_type1
```

The FIND DUPLICATE WITHIN statement is used for accessing records within a particular set occurrence. It locates the first record_type1 within the current set_type1 occurrence whose values for item1 through itemn match those of the current of set_type1.

An implicit assumption is that the records being requested are already resident in RB. By the nature of FIND DUPLICATE WITHIN, another FIND must have already been issued. Therefore, no RETRIEVE request is issued. Instead, given the record type, set type and data item name(s) specified in the statement, KC locates the RB containing the set. Each record within RB is compared in turn with the values associated with the data items until the first duplicate is found. This record is made

available to the user, and CIT is updated to reflect the new currency status.

4. The FIND FIRST Statement

The FIND FIRST statement syntax is as follows:

FIND FIRST record_type1 WITHIN set_type1

The FIND FIRST statement locates the first member record of a set occurrence. This statement has several other forms: FIND LAST, FIND NEXT, and FIND PRIOR. Since we map them all in the same manner, we only describe the mapping process for the FIND FIRST.

Upon encountering the FIND FIRST, KMS must ensure that record_type1 is a member record type of set_type1. Assuming this to be true, KMS forms a RETRIEVE request that retrieves every member of the current set_type1 occurrence into its RB. Once this is accomplished, the first record in the RB may be returned in the case of the FIND FIRST, or the last record in the case of FIND LAST.

For FIND NEXT and FIND PRIOR, it must be assumed that the set occurrence has previously been retrieved into RB. Therefore, the interface simply locates the current of set_type1 and returns the record after it for FIND NEXT, or the record before it for FIND PRIOR. Since all records for a set occurrence are already in RB, there is no need for additional RETRIEVEs.

There are several ways in which all members of a given set occurrence are obtained. The choice of which to use is dictated by the particular situation. In a previous section, discussing the implications of the method of set type declaration chosen, we have stated that there are two kinds of set types: those which reflect an ISA relationship, and those which represent a Daplex function. The methods for finding the set occurrence members are dependent on the kind of set type declared, and are described as follows.

```
(<file,PERSON>,<person,7>,<name,'Allan Jones'>,
    <ssn,000000007>)

(<file,STUDENT>,<student,7>,<advisor,5>,
    <major,1>,<enrollments,3>)
(<file,STUDENT>,<student,7>,<advisor,5>,
    <major,1>,<enrollments,4>)
(<file,STUDENT>,<student,7>,<advisor,5>,
    <major,1>,<enrollments,5>)
```

Figure 19. PERSON and STUDENT Records for 'Allan Jones'

As previously mentioned, when defining the formats for set type declarations, set types reflecting an ISA relationship have names which consist of the name of the owner record type, followed by "_", followed by the name of the member record type. If the set type reflects an ISA relationship, then the primary keys for both owner and

member record types are identical at the AB(functional) level. This primary key similarity is depicted in Figure 19, which shows PERSON and STUDENT AB records for 'Allan Jones'. As seen in the PERSON hierarchy of Figure 13, STUDENT has an ISA relationship with person, as reflected in the CODASYL set type declaration, PERSON_STUDENT. Here, we see that in file PERSON, the database key value is 7 (person = 7), while in file STUDENT, the database key value is again 7 (student = 7). Thus, the ABDL request being formed in this case takes advantage of this and the name overloading previously mentioned, and consists of the following RETRIEVE request:

```
RETRIEVE ((FILE=record_type1) AND
(record_type1=CIT.set_type1.owner_database_key))
(all attributes)   [BY record_type1]
```

Due to the intentional name-overloading approach taken, set types which represent a Daplex function have the same name as the Daplex function. If the set type represents a Daplex function, then there are again two possibilities; the function belongs to the owner record type, or the function belongs to the member record type. Since the functional schema remains available for use, we can traverse it to determine which record type the function belongs to.

If the set type represents a Daplex function which belongs to the owner record type, then we make use of the

database key for the owner record. This is again reflected in the STUDENT records of Figure 19. STUDENT is the owner of the "enrollments" function. We therefore perform a retrieve for all STUDENT records with (student=7). This gives us the database keys of the ENROLLMENT records we need. We then generate one or more RETRIEVE requests--one for each database key returned. If we know that we are in this kind of set type, the ABDL requests formed are:

```
RETRIEVE ((FILE=CIT.owner_record_type) AND
          (CIT.owner_record_type=
              CIT.set_type1.owner_database_key))
          (set_type1)  [BY CIT.owner_record_type]

RETRIEVE ((FILE=record_type1) AND
          (record_type1= (each dbkey returned
                          in previous RETRIEVE)))
          (all attributes)  [BY record_type1]
```

Finally, if the set type represents a Daplex function belonging to the member record type, then we are dealing with a set type which by definition has only one member--the member record occurrence that we are seeking. In this case, we need to find a member record such that its attribute (whose name corresponds to the set type name) has a value equal to the owner record's database key. Therefore, the ABDL request corresponding to this case is as follows:

```
RETRIEVE ((FILE=record_type1) AND
         (set_type1=CIT.set_type1.owner_database_key))
         (all attributes)  [BY record_type1]
```

It can be seen that we are overloading the set-type and record-type names using these methods. For Daplex functions, the set-type name is the same as the appropriate AB(functional) attribute name. In both cases, the database key-field name is the same as the record-type name. It is this name overloading which allows us to systematically and efficiently translate and process the preceeding CODASYL-DML statements.

As an example, let us consider the following request: Find all the grades associated with enrollments by 'Allan Jones'. A possible CODASYL procedure to accomplish this would be:

```
MOVE 'Allan Jones' TO NAME IN PERSON
FIND ANY PERSON USING NAME IN PERSON
FIND FIRST STUDENT WITHIN PERSON_STUDENT
MOVE 'NO' TO EOF
FIND FIRST ENROLLMENT WITHIN ENROLLMENTS
PERFORM UNTIL EOF = 'YES'
    GET ENROLLMENT
    (add grade in ENROLLMENT to result list)
    FIND NEXT ENROLLMENT WITHIN ENROLLMENTS
END_PERFORM
```

The series of FINDs are necessary to navigate through the CODASYL database representation from a known point (i.e., name='Allan Jones') to the appropriate ENROLLMENT record(s). Once the correct PERSON record is found with the FIND ANY statement, we encounter the first

FIND FIRST statement. By the process previously described, we determine that PERSON_STUDENT is a set type reflecting an ISA relationship. The ABDL request generated for it is therefore:

```
RETRIEVE ((FILE=STUDENT) AND
          (STUDENT=CIT.PERSON_STUDENT.owner_database
                _key))
          (STUDENT,ADVISOR,MAJOR,ENROLLMENTS)
          BY STUDENT
```

The RETRIEVE request returns all records satisfying the above criteria to RB, from which the first record is selected and returned.

The second FIND FIRST statement refers to a set type representing a Daplex function. Checking the functional schema, we find that the function belongs to the owner record type, or STUDENT. Therefore, the series of ABDL requests issued is as follows:

```
RETRIEVE ((FILE=STUDENT) AND
         (STUDENT=CIT.ENROLLMENTS.owner_database_key))
         (ENROLLMENTS)
         BY STUDENT
```

(If one or more records are retrieved and residentin RB, the second ABDL request is generated).

```
RETRIEVE ((FILE=ENROLLMENT) AND
          (ENROLLMENT= {each "enrollments" database
                             key from the previous
                             RETRIEVE}))
          (ENROLLMENT,CLASS,GRADE)
          BY ENROLLMENT
```

If one or more records are returned from the first RETRIEVE request, then an equivalent number of the second RETRIEVE requests is generated, one for each record returned in the first RETRIEVE. All records returned are placed in the appropriate RB, from which the first record is selected.

The PERFORM loop is required because in CODASYL only one record at a time is made available to the user. Within the PERFORM loop, the information needed is taken from the current of ENROLLMENTS. A FIND NEXT statement is then issued. In the RB containing the records satisfying the RETRIEVE requests, the record following the current of ENROLLMENTS is selected and is processed in the next iteration of the PERFORM loop. When the last record has been processed, the EOF flag is changed to 'YES,' and the procedure is concluded.

5. <u>The FIND OWNER Statement</u>

The syntax of the FIND OWNER statement is as follows:

FIND OWNER WITHIN set_type1

The mapping of this statement is straightforward, since all information necessary is already resident within CIT. KMS examines the CIT entry for set_type1 and extracts the database key and record type of the owner from it. With this information, the following RETRIEVE request is formed:

```
RETRIEVE ((FILE=CIT.set_type1.owner_record_type) AND
         (CIT.set_type1.owner_record_type =
              CIT.set_type1.owner_database_key))
         (all attributes)
```

6. The FIND WITHIN CURRENT Statement

The syntax of the FIND WITHIN CURRENT statement is as follows:

```
FIND record_type1 WITHIN set_type1 CURRENT
    USING item1, ... ,itemn IN record_type1
```

The FIND WITHIN CURRENT statement is very similar to the FIND DUPLICATE statement. The difference is that where FIND DUPLICATE uses values resident in the current of set type, FIND WITHIN CURRENT uses values resident in UWA. KMS ensures that the record type specified is a member record type of the set type, and that each data item specified is defined for the record type. The request generated is as follows:

```
RETRIEVE ((FILE=record_type1) AND
         (record_type1=CIT.set_type1.owner
            _database key)
         AND (item1=user_value1)
         AND ...
         AND (itemn=user_valuen)
         (all attributes)  [BY record_type1]
```

## D. MAPPING THE CODASYL GET STATEMENTS

The GET statements used in CODASYL-DML are data retrieval statements, just as FIND statements are, except that only records previously retrieved by FIND statements can be accessed by GET statements. While FIND statements

bring records into the appropriate RB and update applicable CIT entries, they cannot actually access the records for the purpose of displays or printouts. This is the purpose of the GET statements.

As was done in the network interface, we issue instructions to KC for handling GET statements instead of mapping them to ABDL RETRIEVEs. There are three options in connection with the GET statement, and each is discussed in turn.

1. <u>The GET and GET record-type Statements</u>

In the absence of further specifications, the GET statement places the entire current of the run unit record into UWA for further access by the user. To do this, KMS informs KC that the record in RB containing the current of the run unit is to be passed to the user. In this case, it doesn't matter what the type of the current of the run unit is.

The GET record_type option is identical to the GET option except that it specifies a record type. The record type being accessed must first be in the current of the run unit RB before this option can be executed. KMS therefore checks the record type in the current of the run unit RB before proceeding further. In this case, again, all data items are returned to the user.

2. <u>The GET item1,...,itemn Statement</u>

The difference between this option and the other GET options is that the user specifies the data items of the record which are to be returned. The syntax of this GET statement is:

GET item1,...,itemn IN record_type1

As before with the GET record_type option, the desired record type must be resident in RB containing the current of the run unit. KMS therefore checks to ensure that the record type is correct in RB, and also that the data items listed match the data items in the record type specified. Once this is done, KMS issues instructions to KC. Specific data items are returned from the records accessed.

E. MAPPING THE DATA UPDATING STATEMENTS

In this section, we consider the statements which perform data-updating operations, namely, CONNECT, DISCONNECT, MODIFY, STORE and ERASE. In several cases, mapping these statements in such a way as to achieve the desired effect on an AB(functional) database is a complex and involved process. When we consider what it is we are actually doing to the AB(functional) database upon execution of several of these statements, we find that we must go to great lengths to preserve the integrity of the AB(functional) database. The effects of such statements

upon the AB(functional) database are discussed, in turn, in the section describing that statement's mapping process.

### 1. The CONNECT Statement

The syntax of the CONNECT statement is as follows:

```
CONNECT record_type1 TO set_type1,...,set_typen
```

The CONNECT statement is used for manual insertion of the current of run unit into the current occurrences of the set types specified. As such, the current of the run unit must be the member record type of each set type specified. Furthermore, its insertion clause must be:

INSERTION IS MANUAL

In order to provide a clear illustration of the various ways by which the CONNECT statement can affect the AB(functional) database, Figure 20 depicts a functional entity subtype declaration, occurrences of its AB(functional) records, and the CODASYL schema declarations associated with it. Although similar to the University database schema, we have contrived this example to demonstrate the steps which could be taken when mapping the CONNECT statement, as well as others encountered later in this section.

The CONNECT statement has a profound effect on the AB(functional) database and its integrity. Because of the method by which we have declared set types when we

functional

```
        SUBTYPE student IS person
            ENTITY
                major    : department;
                enrolled : SET OF enrollment;
                phone#   : SET OF string;
            END ENTITY;
```

AB(functional)

```
(<FILE,student>,<student,7>,<major,1>,<enrolled,3>,
                 <phone#,2174>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,3>,
                 <phone#,2175>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,3>,
                 <phone#,2469>)

(<FILE,student>,<student,7>,<major,1>,<enrolled,4>,
                 <phone#,2174>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,4>,
                 <phone#,2175>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,4>,
                 <phone#,2469>)

(<FILE,student>,<student,7>,<major,1>,<enrolled,5>,
                 <phone#,2174>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,5>,
                 <phone#,2175>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,5>,
                 <phone#,2469>)
```

Figure 20. A Functional/AB(functional)/ CODASYL Mapping Example

CODASYL

```
RECORD NAME IS student
    phone# : CHARACTER 4;

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

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

Figure 20. (Continued)

establish a connection between a member record and an owner record in CODASYL, we are, in actuality, inserting information into an existing AB(functional) record, creating a new AB(functional) record, or even a new set of AB(functional) records, depending on the circumstances. In each of these cases, new information is placed into the AB(functional) database as a result of the CONNECT statement.

As before, set types can be divided into two general categories: those representing ISA relationships, and those representing Daplex functions. The set types representing Daplex functions can be further divided into two categories: those where the information concerning the function is stored in the owner record, and those where the information concerning the function is stored in the member record.

Set types representing ISA relationships and declared with AUTOMATIC insertion clauses cannot be used in a CONNECT statement. On the other hand, set types representing Daplex functions declared with MANUAL insertion clauses are available for use in a CONNECT statement.

Before we discuss the mapping of the CONNECT statement into appropriate ABDL requests, we must first consider what the current of run unit and the owner of a set occurrence actually represent. To do this, let us

review a portion of the functional-AB(functional) database mapping process. In a functional entity type or subtype we may encounter a scalar multi-valued function. Referring to Figure 20 again, the "phone#" function is a scalar multi-valued function. By definition, each of the values associated with an occurrence of the scalar multi-valued function belong to the same entity type or subtype occurrence. When we represent this entity type or subtype occurrence in the AB(functional) database, we actually have one or more record occurrences, one for each of the scalar multi-valued function values. In every one of these records, each attribute-value pair is identical to the corresponding attribute-value pairs in the other records, with the exception of the attribute representing the scalar multi-valued function. In Figure 20, we see this reflected in the AB(functional) record occurrences. So when we wish to deal with an entity type or subtype which contains a scalar multi-valued function, we may, in actuality, be dealing with a <u>set</u> of AB(functional) records.

When we map the Daplex schema into an equivalent CODASYL schema, the scalar multi-valued function becomes a field in the record type representing the entity type or subtype. As a result, when we have a current of the run unit or an owner of the set occurrence which contains a field representing a scalar multi-valued function, we are again dealing with a <u>set</u> of AB(functional) records, and we

update them as a whole when executing the CONNECT statement.

When we wish to connect a member record in the current of the run unit to a set type, precisely where the information concerning the set type is stored in the functional schema determines what actions are taken with respect to the AB(functional) record(s). Basically, the information concerning the set type is stored either in the owner or member record type of the set type. We discuss each possibility in turn.

a. Information Resides in Owner Record

When the information concerning the Daplex function, represented by the CODASYL set type, resides in the owner record of the set type, the functional schema indicates that the function points to a set of entity type or subtype occurrences. This functional set can be null, or it can contain one or more members in it. If the functional set is null, then the CODASYL set type occurrence for it has no member records associated with it yet. Referring to Figure 20, this might correspond to the case where "enrolled" is a null set. In this case, the AB(functional) record occurrence shown in Figure 20 would be reduced to that depicted by Figure 21. Note that there are still three AB(functional) records remaining, due to the scalar multi-valued function "phone#." If there is no

(<FILE,student>,<student,7>,<major,1>,<enrolled,NULL>,
<phone#,2174>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,NULL>,
<phone#,2175>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,NULL>,
<phone#,2469>)

Figure 21. AB(functional) Occurrence for NULL Enrolled Function

scalar multi-valued function, there is only one AB(functional) record.

With the discussion in the previous paragraph in mind, we see that there are actually four possible courses of action to be taken when the information resides in the owner record of the set type, depending on whether the functional set is null or not, and again on whether there is one or more scalar multi-valued functions associated with the entity type or subtype. We describe each possible course of action in turn.

(1) Null Functional Set--No Scalar Multi-Valued Function. If the functional set is null and there are no scalar multi-valued functions associated with the entity type or subtype, then the AB(functional) record corresponding to the owner of the set-type occurrence is indeed the only record to be updated. In this case, we replace the null value for the attribute whose name corresponds to the CODASYL set type name with the database

key of the current of the run unit. Thus, the ABDL request formed in this case is:

```
UPDATE ((FILE=CIT.set_type1.owner_record_type) AND
        (CIT.set_type1.owner_record_type=
                 CIT.set_type1.owner_database_key))
        (set_type1=CIT.RUN_UNIT.database_key)
```

(2) Null Functional Set--Scalar Multi-Valued Function. If the functional set is null and there is a scalar multi-valued function associated with the entity type or subtype, we must update the null value in each AB(functional) record created for the scalar multi-valued function. Therefore, we must retrieve all applicable records using information stored in CIT. Unfortunately, the database key of the owner of the set-type occurrence is not enough information to perform the RETRIEVE request. We need to know the names of all the attributes which do not represent scalar multi-valued functions, as well as the values associated with them for the owner of the set-type occurrence. Once we have this information, we use it to form an UPDATE request which replaces the null values in the attribute whose name corresponds to the set-type name with the database key of the current of the run unit. To obtain this information means the creation of a procedure written in the host programming language, and as such is not discussed further herein. Assuming we can obtain this information, the ABDL requests required are as follows:

```
RETRIEVE ((FILE=CIT.set_type1.owner_record_type) AND
          (CIT.set_type1.owner_record_type=
                   CIT.set_type1.owner_database_key))
          (all attributes)
```

(We now utilize the aforementioned procedure to determine which values are of interest to us in the RB holding the results of the RETRIEVE request, and extracting the values we need)

```
UPDATE ((FILE=CIT.set_type1.owner_record_type) AND
        (CIT.set_type1.owner_record_type=
                  CIT.set_type1.owner_database_key)
        AND (attribute1=value1)
        AND ...
        AND (attributen-valuen))
        (set_type1=CIT.RUN_UNIT.database_key)
```

As previously mentioned, the attributes and values in the UPDATE request are found using the host programming language procedure.

(3) Non-Null Functional Set--No Scalar Multi-Valued Function. If the functional set is not null and there is no scalar multi-valued function associated with the entity type or subtype, then we must create another AB(functional) record whose attributes and values are identical to the owner of the set-type occurrence, with the exception of the attribute whose name corresponds to the set-type name. This attribute gets the value of the database key of the current of the run unit. Again, we retrieve the record which is the owner of the set-type occurrence, and extract all attributes and values for use in an INSERT request. We can use the aforementioned host language procedure to accomplish this. Assuming the

existence of this procedure, the ABDL requests required are as follows:

```
RETRIEVE ((FILE=CIT.set_type1.owner_record_type) AND
         (CIT.set_type1.owner_record_type=
                CIT.set_type1.owner_database_key))
         (all attributes)
```

(We utilize the host language procedure, obtaining each attribute and value from the record returned above)

```
INSERT (<FILE,CIT.set_type1.owner_record_type>,
        <CIT.set_type1.owner_record_type,
              CIT.set_type1.owner_database_key>,
        <data itemi,valuei>,
        <set_type1,CIT.RUN_UNIT.database_key>)
```

(4) Non-Null Functional Set--Scalar Multi-Valued Function. Finally, if the functional set is not null but there is a scalar multi-valued function associated with the entity type or set type, we must make a copy of every record representing this scalar multi-valued function and possessing the database key of the owner of the set type occurrence. However, the attribute whose name corresponds to the set type name gets the value of the database key of the current of run unit.

To do this, we must retrieve the AB(functional) record which is the owner of the set-type occurrence. Having done this, we use our host language procedure to obtain all non-scalar multi-valued function attributes and their values. These in turn are used to retrieve every AB(functional) record with these values,

thereby capturing all records representing the scalar multi-valued function. Finally, for each record in RB from the last RETRIEVE, we insert a new record whose values are the same as the one in RB, with the exception of the attribute whose name corresponds to the set-type name. This value becomes the database key of the current of the run unit. The ABDL requests required are as follows:

```
RETRIEVE ((FILE=CIT.set_type1.owner_record_type) AND
                 (CIT.set_type1.owner_record_type=
                      CIT.set_type1.owner_database_key))
                 (all attributes)
```

(We utilize the host language procedure, obtaining each non-scalar multi-valued function attribute and value from the record returned above)

```
RETRIEVE ((FILE=CIT.set_type1.owner_record_type) AND
          (CIT.set_type1.owner_record_type=
                  CIT.set_type1.owner_database_key)
          AND (attribute1=value1)
          AND  ...
          AND (attributen=valuen))
          (all attributes)
```

(For each record in the RB required for the above RETRIEVE, the following INSERT request is generated)

```
INSERT (<FILE,CIT.set_type1.owner_record_type>,
        <CIT.set_type1.owner_record_type,
                  CIT.set_type1.owner_database_key>,
        <data itemi,valuei>,
        <set_type1,CIT.RUN_UNIT.database_key>)
```

This concludes the discussion on the possible actions to be taken when the information concerning the Daplex function, represented by the CODASYL set type, resides in the owner record of the set type.

b. Information Resides in Member Record

When the current of the run unit is the record whose AB(functional) record needs updating, we are dealing with the set-type which has only one CODASYL member record. In terms of AB(functional) records, this translates to all the records with the same database key. Therefore, the issue of whether or not a scalar multi-valued function exists in the record is unimportant. Instead, we simply update all records whose database key is equivalent to the database key of the current of the run unit. The update is to the attribute whose name corresponds to the name of the set type. This attribute value becomes the database key of the owner record of the set type. The ABDL request necessary is as follows:

```
UPDATE ((FILE=record_type1) AND
        (record_type1=CIT.RUN_UNIT.database_key))
        (set_type1=CIT.set_type1.owner_database_key)
```

2. The DISCONNECT Statement

The syntax for the DISCONNECT statement is as follows:

```
DISCONNECT record_type1 FROM set_type1,...,set_typen
```

The DISCONNECT statement requires the current of the run unit to be a member type of the set type(s) listed, and that the record be removed from the set occurrences that are current.

As is discussed in the section on the CONNECT statement, when we wish to add certain types of information to existing records, we connect its CODASYL representation to a set type occurrence. The effect of the CONNECT statement is to add a value to an attribute, or to add an entire AB(functional) record, or even to add a set of AB(functional) records.

Similarly, when we wish to remove information from the AB(functional) database, one way to do it is with the DISCONNECT statement. The DISCONNECT statement is basically the opposite of the CONNECT statement, and causes the current of the run unit to be disconnected from the set(s) listed.

The result of the DISCONNECT statement may be that a certain attribute value is nulled out, or that an AB(functional) record is deleted from the database, or that an entire set of AB(functional) records are deleted from the database. The circumstances under which each possibility occurs depends again on whether the function information is contained in the owner or member record. Each of these cases is discussed in turn.

When the information concerning the Daplex function, as represented by the CODASYL set type, resides in the owner record, the functional schema again indicates that the function points to a set of entity type or subtype occurrences. This function set is either a singleton or

contains two or more members. Figure 22 depicts a singleton with a scalar multi-valued function, since there is only one value for "enrolled". Figure 20 shows three members in the "enrolled" function set, even though there are actually nine AB(functional) records due to the scalar multi-valued function "phone#."

```
(<FILE,student>,<student,7>,<major,1>,<enrolled,3>,
                                <phone#,2174>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,3>,
                                <phone#,2175>)
(<FILE,student>,<student,7>,<major,1>,<enrolled,3>,
                                <phone#,2469>)
```

Figure 22. Singleton AB(functional) "Enrolled" Function Set

When mapping the DISCONNECT statement to appropriate ABDL requests, it turns out that the most important factor is whether the function set is a singleton, or whether it has two or more members. If the function set is a singleton as shown in Figure 22, we merely null out the value of the attribute whose name corresponds to the set type name. The ABDL request required in this case is as follows:

```
UPDATE ((FILE=CIT.set_type1.owner_record_type) AND
        (CIT.set_type1.owner_record_type=
                  CIT.set_type1.owner_database_key)
        (set_type1=NULL)
```

One can see that, in the case of a scalar multi-valued function as shown in Figure 22, all applicable AB(functional) records are updated by the above ABDL request. The combination of the database key and the function value in the request is sufficient to specify the correct AB(functional) records.

If the function set has two or more members as in Figure 20, we delete all the records with the correct combination of the database key and the function value. The ABDL request for this is as follows:

```
DELETE ((FILE=CIT.set_type1.owner_record_type) AND
        (CIT.set_type1.owner_record_type=
                  CIT.set_type1.owner_database_key)
        AND (set_type1=CIT.RUN_UNIT.database_key))
```

Again, if a scalar multi-valued function is part of the owner record type, all the appropriate AB(functional) records are deleted in the above DELETE request.

We have seen that it is necessary to determine if the function set contains two or more records or not, in order to know whether to issue an UPDATE or DELETE ABDL request. This must be accomplished by a host programming language procedure, and once again this is assumed to be available.

When the current of the run unit is the record whose AB(functional) record needs updating, we are, by definition of the schema translation process, dealing with a singleton function set. Even though there may be many

records in the AB(functional) database with the same database key, all these contain the same value for the attribute whose name corresponds to the set-type name. Therefore, we merely need to null out the value of this attribute. The ABDL request necessary to do this is as follows:

```
UPDATE ((FILE=record_type1) AND
        (record_type1=CIT.RUN_UNIT.database_key) AND
        (set_type1=CIT.set_type1.owner_database_key))
        (set_type1=NULL)
```

A final note should be made at this time concerning the CONNECT and DISCONNECT statements. These two statements constitute the means by which certain values in the AB(functional) database are modified. To modify attributes representing functions, we must first disconnect them, thus replacing the previous value with NULL. We may then reconnect them to another set type occurrence owner, which is the equivalent of replacing the AB(functional) NULL value with a new database key. Unfortunately, no other method is available in CODASYL to accomplish this.

3. <u>The MODIFY Statement</u>

The syntax of the MODIFY statement is as follows:

```
MODIFY record_type1    , or
MODIFY item1,...,itemn IN record_type1
```

The MODIFY statement causes the entire current of the run unit to be modified, or only certain specified data items in it. This is reflected in the two optional syntaxes shown above. The information used to modify the current of the run unit is specified in the UWA of this record.

In essence, we have already used the mapping of this statement extensively in the CONNECT and DISCONNECT statements. That is, we have found it necessary to perform UPDATEs on the AB(functional) database. This is exactly what is done to the current of the run unit in this case. KMS retrieves the values for the specified data items from the UWA, and uses them to form the following UPDATE request(s), one for each data item specified in UWA.

```
UPDATE ((FILE=record_type1) AND
        (record_type1=CIT.RUN_UNIT.database_key))
        (data itemi = user valuei)
```

So, if two fields in the record type were to be modified, two UPDATE requests would be generated. Of course, for the MODIFY option which changes the entire record, KMS would have to generate an UPDATE request for each field within the record type.

4. The STORE Statement

The syntax of the STORE statement is as follows:

```
STORE record_type1
```

The STORE statement is used to insert a new record into the database. The field values of the record are constructed in the UWA before the STORE statement is called. The STORE statement is also used to place the new record into certain set-type occurrences of which it is a member.

In a previous thesis concerning the MLDS network interface, three set selection options were specified and dealt with, namely, BY APPLICATION, BY STRUCTURAL and BY VALUE [Ref. 8]. Because of the method by which we performed the schema mapping, only the BY APPLICATION method of set selection is used. Also, the STORE statement places records only into the set types whose insertion clause is AUTOMATIC.

In addition to the set selection and insertion requirements above, the interface must determine if any of the fields being inserted has a DUPLICATES NOT ALLOWED clause associated with it. If one or more fields cannot have duplicates, a RETRIEVE request must be formed to see if the specific <u>combination</u> of these fields already exists in the AB(functional) database. As a result, we see that an INSERT request and possibly a RETRIEVE request must be generated for each STORE statement. The RETRIEVE request is generated to determine the status of duplicates, and the INSERT request is to store the record if no duplicate exists.

For each attribute in the record to be inserted, whose name corresponds to a set-type name, we must check to see if the set-type insertion clause is AUTOMATIC. If it is, then the value of this attribute becomes the database key of the owner of the set-type occurrence.

Of great importance to the integrity of the database is the proper handling of overlap constraints. As previously discussed, an Overlap Table must be provided that keeps track of the set types which may coexist with others. This Overlap Table prohibits set occurrences whose owner is the owner of another set type, which is considered to be disjoint from the first. On the other hand, set types may be defined where the information is stored in the member type, as when mapping single-valued functions. These set types may be allowed to coexist with the other sets owned by the record we are trying to add to the database. The Overlap Table must include the necessary information to maintain the integrity of the AB(functional) database.

Since the information contained in the Overlap Table cannot be accessed by ABDL requests, a procedure written in the host programming language must be written for this purpose. In this thesis, we assume the existence of such a procedure.

In summary, the process of mapping the STORE statement to appropriate ABDL requests is as follows:

(If any fields are designated as DUPLICATES NOT ALLOWED, KMS forms the following RETRIEVE request. Values for the appropriate fields are found in the UWA record template)

```
RETRIEVE ((FILE=record_type1) AND
          (data itemsi = user valuesi))
          (record_type1)
```

(If anydatabase keys are returned, an errormessageis issued. If not, using the host language procedure previously described, we next check the Overlap Table to see if the insertion of this record into the AB(functional)database will violate its integrity. If so, an error message is issued. Otherwise, the following ABDL request may be executed)

```
INSERT (<FILE,record_type1>,<record_type1,***>,
        <data itemsi,user valuesi>,
        <set_typesi,CIT.set_typesi.owner
                   _database_key>)
```

Here, we see that the data item values come from the UWA record template, and the set type values come from the appropriate set types whose insertion clauses are AUTOMATIC.

5. <u>The ERASE Statement</u>

The final statement to be mapped in our direct language interface is the ERASE statement. The mapping of this statement is made difficult because restraints must be applied to it, not only due to CODASYL rules, but functional rules as well. Let us look at the ERASE statement and its CODASYL restraint. The syntax for the ERASE statement is as follows:

```
ERASE record_type1
```

The ERASE statement deletes the current of the run unit from the database. Obviously, the record in the current of the run unit must be of type "record_type1". The CODASYL restraint placed on this statement is that the record cannot be an owner of a non-null set type occurrence. Briefly, this means that we must perform a RETRIEVE request to see if there are any set-type occurrences for which the owner database key is the database key of the current of the run unit.

In addition to the CODASYL restraint mentioned above, a further limitation is applied as a result of the target database being functional. In order to describe the functional restraint on the CODASYL ERASE statement, let us briefly discuss the functional counterpart to it--the Daplex DESTROY statement.

The Daplex DESTROY statement deletes an entity from the functional database. If an entity subtype exists wherein an occurrence of it derives from the entity we wish to delete, the subtype occurrence is deleted as well. This follows on down to the leaves of the hierarchy to which the original entity belongs. As an example, and referring to the PERSON generalization hierarchy depicted in Figure 13, if we wish to delete a particular EMPLOYEE entity, then any SUPPORT_STAFF or FACULTY entities associated with this EMPLOYEE entity would be deleted as well.

There is an important limitation placed on the DESTROY statement in Daplex. If the entity we wish to delete is referenced by some other database function, the DESTROY operation is aborted. An example of this can be described using the fragment of the University database schema shown in Figure 23. Herein, we see the entity subtypes FACULTY, STUDENT and GRADUATE.

```
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;
```

Figure 23. University Database Schema Fragment

Let us assume that we wish to delete a FACULTY record from the database. We would issue the appropriate DESTROY statement, specifying a particular FACULTY entity to be deleted. However, one can see that FACULTY is referenced by both STUDENT and GRADUATE. Daplex requires that neither STUDENT nor GRADUATE have any entities which

reference the particular FACULTY entity we wish to delete. In effect, if a STUDENT entity has our FACULTY entity for an advisor, the advisor needs to be changed before the FACULTY entity can be deleted. In a like manner, our FACULTY entity cannot be part of an "advisory_committee" function in GRADUATE, if it is to be deleted.

When we consider how to map the CODASYL ERASE statement, we need to keep in mind the limitations imposed by the Daplex DESTROY statement. Because we are ultimately dealing with a functional database, we must only perform operations on it that are consistent with its integrity constraints. Therefore, we must apply the functional restraint to our usage of the CODASYL ERASE statement. This is done in the following manner.

Recalling how the functional schema is mapped to CODASYL, we note that the "advisor" function in STUDENT is mapped to a set type whose owner is FACULTY, and whose member is STUDENT. Similarly, "advisory_committee" in GRADUATE is mapped to a set type whose owner is GRADUATE, and whose member is FACULTY. So if there are any records in the ADVISORY_COMMITTEE set type, or any member records in the ADVISOR set type which correspond to the FACULTY record we wish to erase, the process must abort. Where CODASYL requires that the record not be the owner of any non-null set occurrences, Daplex that requires the record not be a member of any set occurrences other than the one

connecting it to its parent in the generalization hierarchy. In this case, the FACULTY member record forms an ISA relationship with its owner record type, STUDENT.

This is not a fatal combination of restraints by any means. Not being a member of a set occurrence simply requires that the record be disconnected from the set type. Not being an owner of a set occurrence requires that its members disconnect from it and reconnect to another owner record before the ERASE operation is carried out.

Therefore, two separate types of RETRIEVE requests need to be issued in conjunction with the ERASE statement: one type that finds all of the set occurrences for which the current of run unit is the owner, and the other type which finds all of the set occurrences for which the current of the run unit is a member. If both of these RETRIEVEs return null RBs, a DELETE request is issued which removes the current of the run unit from the database. It is not necessary to take any actions with respect to the possible set occurrence between the record and its parent in the generalization hierarchy, because all information pertaining to the ISA relationship is carried in the member record.

The ABDL requests necessary to map the ERASE statement are as follows:

(For each set type which lists the current of run unit record type as owner, the following request is generated)

```
RETRIEVE ((FILE=CIT.set_typei.member_record_type) AND
          (set_typei=CIT.RUN_UNIT.database_key))
          (set_typei)
```

(If the RB for the above request is non-null, abort the operation. If not, then for each set type which lists the current of run unit record type as member, with the exception of one which represents an ISA relationship, the following request is generated)

```
RETRIEVE ((FILE=CIT.set_typei.owner_record_type) AND
          (set_typei=CIT.RUN_UNIT.database_key))
          (set_typei)
```

(If the RB for the above request is non-null,abort the operation.If not,then the following request is generated)

```
DELETE ((FILE=record_type1) AND
        (record_type1=CIT.RUN_UNIT.database_key))
```

We note that in the above ABDL requests, we do not generate a RETRIEVE request to search for owner information in the record we wish to delete. Since we already have the record in the RB for the current of the run unit, KMS checks the record itself to ensure that no information is present indicating that the record owns other records in a set occurrence.

As is previously mentioned in Chapter 2, there is another ERASE statement available to the CODASYL user--the ERASE ALL statement. This statement deletes the current of the run unit from the database. Additionally, if the

record is an owner of a set occurrence, each record in the set occurrence is deleted as well. Also, any set occurrences associated with these records are themselves the object of this recursive ERASE operation.

Due to restraints imposed by the functional database, we do not map this statement to an equivalent ABDL request,as it would violate the integrity of the AB(functional) database. The reason for this is as follows.

Referring once more to Figure 23, let us assume we wish to use the ERASE ALL option on a FACULTY record. As is previously discussed, the "advisor" function in the STUDENT entity subtype maps into a CODASYL set type whose owner is FACULTY, and whose member is STUDENT. If the FACULTY record we wish to delete is the owner of a STUDENT record by means of this set type, we are restrained from executing an ERASE statement. However, under the rules of the ERASE ALL statement, not only are we free to delete the FACULTY record, but we are required to delete the STUDENT record as well. Bsides being contrary to the intent and operation of the Daplex DESTROY statement, this side effect may have an adverse effect on the integrity of the AB(functional) database. Therefore, we do not map the ERASE ALL statement to an equivalent ABDL request.

Not mapping the ERASE ALL statement does not limit our ability to manipulate the database. In reality, the

ERASE ALL statement is nothing more than a convenience for the CODASYL user to use, in the infrequent case that he wishes to delete large portions of the database recklessly. The same effect can be caused by using repeated ERASE statements instead.

# VI. CONCLUSIONS

As is discussed in the introduction, the standardized approach to the design and implementation of a database system resulted in single-model, single-language database systems with their inherent lack of flexibility and extensibility. The multi-lingual database system has been designed and implemented specifically to address these and other problems. MLDS currently provides facilities to store and manipulate information using any of the five sets of data models and data languages, namely, the hierarchical/DL/I, relational/SQL, network/CODASYL-DML, functional/Daplex and attribute-based/ABDL.

## A. THE CONTRIBUTION OF THE RESEARCH

In this thesis, we have addressed the topic of accessing and manipulating information stored in one data model with the data manipulation facilities of a different data model. Specifically, we have presented a methodology for allowing the network/CODASYL-DML user to access and/or manipulate a functional database as supported by MLDS. This is the first step in the process of extending our multi-lingual database system to a true multi-model database system (MMDS).

In the thesis, we recognized that several approaches may be taken with respect to the mapping process. Each has

its place in possible data model combinations, so that the methodology which works for mapping from one model to another may not be the one which is used for a different data model combination. We discussed three different approaches, and gave our reasons for selecting one as the best approach. The other two approaches should be considered when other data model combinations are studied.

The chosen approach has entailed translating the functional database schema to an equivalent network schema, and mapping CODASYL-DML statements to ABDL requests which accomplish the intent of the CODASYL-DML statements, while preserving the integrity of the AB(functional) database. The constructs used for the network schema, as well as the CODASYL-DML statements mapped to ABDL requests were taken from those used in the MLDS network interface [Ref. 8]. Due to the methodology used to translate the functional database schema to an equivalent network schema, some network schema constructs have not been needed. As a result, certain CODASYL-DML statements which utilized these schema constructs have not been mapped to ABDL requests.

When mapping CODASYL-DML statements to ABDL requests, we have found it necessary to recall that the target database is an attribute-based representation of a functional database. Therefore, rather than blindly mapping CODASYL-DML statements, the integrity of the database as well as the intent of the equivalent statements

in the Daplex data language have been taken into account. One result of this is that the ERASE ALL statement option has not been mapped to equivalent ABDL requests, because it would compromise the integrity of the AB(functional) database, had it been mapped.

We feel that the methodology presented in this thesis is sufficient for an implementation, the next step to providing the first operational portion of MMDS. We also believe that other mappings from one data model to another are possible, using one of the three approaches presented herein. The result will be a database management capability that is unique in the world today, allowing for greatly increased productivity and flexibility in the workplace.

## B. SOME OBSERVATIONS AND INSIGHTS

Although it is difficult to say with certainty, it seems that using CODASYL-DML to manipulate a functional database is much easier to do when the database is stored in an attribute-based form, rather than in the functional form. The functional model supports recursive relationships, while the network model does not. We have found CODASYL-DML to be insufficient to do all the things we wish to do to the functional database, as demonstrated by the need for an Overlap Table to maintain the disjointness of entities.

What seems to make the mapping possible is the transformation of the functional model to the attribute-based model, which successfully captures its characteristics and constraints. Because of the method by which this is done--especially the naming conventions of the database key which allows for name-overloading--we can successfully deal with the AB(functional) database using CODASYL-DML.

This process of transformation to an intermediate (or kernel) data model is not unlike a mathematical process, which greatly simplifies certain second-order partial differential equation problems, the Laplace Transform. Using the Laplace transformation, we map a second-order partial differential equation from the Euclidian space to Laplace space, obtain a linear equation, operate on it in ways that we cannot with the second-order equation, obtain a solution and perform an inverse Laplace transformation to map the solution back to the original space. The new solution is one that we could not easily reach from the original second-order partial differential equation. The Laplace transformation process allows us to sidestep many difficult problems we may otherwise have had to face.

In a like manner, we have transformed the functional database to an AB(functional) database--one which we can operate on using CODASYL-DML. We may also be able to operate on it with relational and/or hierarchical data

manipulation languages, although this has not been studied. It seems more likely that the more complex models, such as the network and functional data models, will be able to operate with a greater degree of success on the transformed databases of the less complex models. This is an area which deserves more study.

## LIST OF REFERENCES


1. Demurjain, S.A. and Hsiao, D.K., "New Directions in Datasbase-Systems Research and Development," in the Proceedings of the New Directions in Computing Conference, Trondheim, Norway, August, 1985; also in Technical Report, NPS-52-85-001, Naval Postgraduate School, Monterey, California, February 1985.

2. Banerjee, J. and Hsiao, D.K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," Proceedings of National ACM Conference, 1978.

3. Banerjee, J. and Hsiao, D.K., "The Use of a Database Machine for Supporting Relational Databases," Proceedings 5th Workshop on Computer Architecture for Nonnumeric Processing, August 1978.

4. Banerjee, J., Hsiao, D.K., and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, January 1980.

5. Macy, G., Design and Analysis of an SQL Interface for a Multi-Backend Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1984.

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

7. Weishar, D., Design and Analysis of a Complete Hierarchical Interface for a Multi-Backend Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.

8. Wortherly, C.R., The Design and Analysis of a Network Interface for a Multi-Lingual Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

9. 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.

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

11. Kloepping, G.R. and Mack, J.F., The Design and Implementation of a Relational Interface for the Multi-Lingual Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

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

13. Anthony, J.A. and Billings, A.J., The Implementation of an Entity-Relationship Interface for the Multi-Lingual Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

14. The Ohio State University, Columbus, Ohio, Technical Report OSU-CISRC-TR-81-7, Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion, and Capacity Growth (Part 1), by D.K. Hsiao and M.J. Menon, August 1981.

15. The Ohio State University, Columbus, Ohio, Technical Report, OSU-CISRC-TR-81-8, Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion, and Capacity Growth (Part 2), by D.K. Hsiao and M.J. Menon, August 1981.

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

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

18. Tsichritzis, D.C. and Lochovsky, F.H., Data Models, Prentis-Hall, 1982.

19. Banerjee, J. and Hsiao, D.K., "A Methodology for Supporting Existing CODASYL databases with New Database Machines," Proceedings of National ACM Conference, 1978.

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

21. The Ohio State University, Columbus, Ohio, Technical Report OSU-CISR-TR-77-4, DBC Software Requirements for Supporting Network Databases, by J. Banerjee, D.K. Hsiao, and D.S. Kerr, November 1977.

22. Smith, J.M., et al, "Multibase - Integrating Heterogeneous Distributed Database Systems," National Computer Conference, 1981, Vol. 50, 1981.

23. Spaccapietra, S., et al, "An Approach to Effective Heterogeneous Databases Cooperation", Distributed Data Sharing Systems, edited by R.P. van de Riet and W. Litwin, North-Holland Publishing Company, 1982.

INITIAL DISTRIBUTION LIST

| | | No. Copies |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 0142<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |
| 3. | Department Chairman, Code 52<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |
| 4. | Curriculum Officer, Code 37<br>Computer Technology<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 1 |
| 5. | Professor David K. Hsiao, Code 52HQ<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |
| 6. | Steven A. Demurjian, Code 52<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |
| 7. | Mr. Frank Manola<br>Computer Corporation of America<br>Four Cambridge Center<br>Cambridge, Massachusetts 02142-1489 | 1 |
| 8. | Rev. Alvin W. Rodeck<br>201 N. Pine St.<br>Nokomis. Illinois 62075 | 1 |
| 9. | Capt. Brian D. Rodeck<br>6319 Ember Ct.<br>Manassas, Virginia 22111 | 3 |



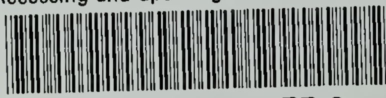
18070 2

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5002

NOV 27
JUN - 9

213614

Thesis
R66554 Rodeck
c.1 Accessing and updating functional databases using CODASYL-DML.


Accessing and updating functional databa
3 2768 000 67955 9
DUDLEY KNOX LIBRARY