

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

**THE ACTIVATION AND TESTING OF THE NETWORK
CODASYL-DML INTERFACE OF THE M²DBMS
USING THE EWIR DATABASE**

by

Timothy J. Werre
and
Barry A. Diehl

June 1996

Thesis Advisor:

C. Thomas Wu

Approved for public release; distribution is unlimited.

19960805 017

DTIC QUALITY ASSURED 1

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE June 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE THE ACTIVATION AND TESTING OF THE NETWORK CODASYL-DML INTERFACE OF THE M ² DBMS USING THE EWIR DATABASE			5. FUNDING NUMBERS	
6. AUTHOR(S) Timothy J. Werre and Barry A. Diehl				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) The Electronic Warfare Integrated Reprogramming Database (EWIRDB) is the primary Department of Defense source for technical parametric performance data on non-communications emitters. A problem of the EWIRDB is that the data are represented in disjoint parametric tree models that are implementation oriented. The parametric tree with its deceptive hierarchical structure, provides a poor modeling construct that obscures the intended semantics and representation of the data, thus making the database difficult to use and understand from a users perspective. The problem addressed by this thesis is to determine if the network model and the network interface of the Multi-Lingual, Multi-Model Database Management System (M ² DBMS) in the Laboratory for Database Systems Research at the Naval Postgraduate School is capable of supporting a representative subset of the EWIRDB. The primary goal of this thesis is to implement a representative portion of the EWIR database on the network interface of the M ² DBMS. In order to accomplish this goal, the following issues must be addressed: First, the network interface must be activated and returned to its original operational state; second, the network interface must be tested to determine its capabilities and limitations; and lastly, the design and specification of a network EWIR data model must be completed prior to its implementation. We successfully reactivated the network interface to its original operational state. However, testing revealed significant limitations of the network interface. Due to these limitations, only the data definition portion of our proposed design was fully implemented.				
14. SUBJECT TERMS Multi-Lingual and Multi-Model Database (M ² DBMS) Network CODASYL-DML Interface; Electronic Warfare Integrated Reprogramming Database (EWIRDB);			15. NUMBER OF PAGES	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

Approved for public release; distribution is unlimited

**THE ACTIVATION AND TESTING OF THE
NETWORK CODASYL-DML INTERFACE
OF THE M²DBMS USING THE EWIR DATABASE**

Timothy J. Werre

Lieutenant Commander, United States Navy
B.S., University of Wisconsin, Madison, 1983

Barry A. Diehl

Captain, United States Army
B.S., The Pennsylvania State University, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

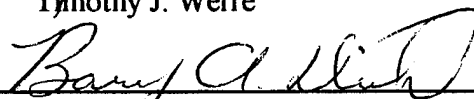
NAVAL POSTGRADUATE SCHOOL

June 1996

Authors:

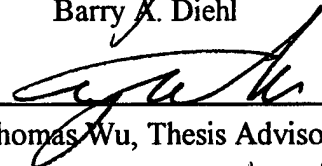


Timothy J. Werre



Barry A. Diehl

Approved by:



C. Thomas Wu, Thesis Advisor



David K. Hsiao, Second Reader



Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

The Electronic Warfare Integrated Reprogramming Database (EWIRDB) is the primary Department of Defense source for technical parametric performance data on non-communications emitters. A problem of the EWIRDB is that the data are represented in disjoint parametric tree models that are implementation oriented. The parametric tree with its deceptive hierarchical structure, provides a poor modeling construct that obscures the intended semantics and representation of the data, thus making the database difficult to use and understand from a users perspective. The problem addressed by this thesis is to determine if the network model and the network interface of the Multi-Lingual, Multi-Model Database Management System (M²DBMS) in the Laboratory for Database Systems Research at the Naval Postgraduate School is capable of supporting a representative subset of the EWIRDB.

The primary goal of this thesis is to implement a representative portion of the EWIR database on the network interface of the M²DBMS. In order to accomplish this goal, the following issues must be addressed: First, the network interface must be activated and returned to its original operational state; second, the network interface must be tested to determine its capabilities and limitations; and lastly, the design and specification of a network EWIR data model must be completed prior to its implementation.

We successfully reactivated the network interface to its original operational state. However, testing revealed significant limitations of the network interface. Due to these limitations, only the data definition portion of our proposed design was fully implemented.

TABLE OF CONTENTS

I. INTRODUCTION	1
A. OVERVIEW.....	1
B. MOTIVATION.....	4
C. THESIS OBJECTIVES	6
D. ORGANIZATION OF THE THESIS.....	7
II. BACKGROUND	9
A. AN OVERVIEW OF THE M ² DBMS	9
1. A Multi-Back-end Database Computer	9
2. A Multi-Lingual, Mult-Model Database System	12
3. The Kernel Data Model and Kernel Data Language	15
B. THE NETWORK DATA MODEL AND ITS LANGUAGE	16
1. Basic Concepts and Structure	17
2. Data Definition in the Network Interface	23
3. Data Manipulation in the Network Interface	27
C. AN OVERVIEW OF THE EWIR DATABASE.....	36
III. ACTIVATION OF THE NETWORK INTERFACE.....	41
A. INITIAL STATE OF THE NETWORK INTERFACE.....	41
B. MODIFICATIONS AND CORRECTIONS	42
1. System Configuration and Directory Structure.....	42
2. Interface and Program Code Modifications	44
C. MASS LOAD FUNCTION.....	48
D. TESTING THE NETWORK INTERFACE	53
1. The Original PARTS Database	55
2. The PARTS2 Database.....	62
3. The PARTS3 Database.....	67
4. The PARTS4 Database.....	71
5. The PARTS5 Database.....	78
6. Summary.....	87

IV. IMPLEMENTATION OF THE EWIR DATABASE.....	89
A. EWIR DATABASE SPECIFICATION.....	90
1. The “Old” EWIR Data Model.....	90
2. The Conceptual EWIR Network Data Model.....	96
3. The Antenna Data	100
4. Transforming the Object-Oriented Model to the Network Model	103
B. DATA DEFINITION OF THE EWIR DATABASE.....	106
1. The Schema Listing	106
2. The Template File.....	109
3. The Descriptor File.....	109
C. LOADING THE EWIR RECORD DATA	112
V. CONCLUSION.....	117
APPENDIX. N_MASS_LOAD() FUNCTION SOURCE CODE	121
LIST OF REFERENCES.....	125
INITIAL DISTRIBUTION LIST	129

LIST OF FIGURES

1. M ² DBMS Cross-Model Accessing Capability	3
2. The Multi-Back-end Database Computer	11
3. The Multi-Lingual and Multi-Model Database System.....	13
4. Multi-Lingual and Multi-Model Interface Design for M ² DBMS	15
5. The Record Type PARTS	18
6. The PARTS Database.....	19
7. A Set Occurrence (S-SP) in the PARTS Database.....	20
8. Conversion of M:N Relationship to 1:N Relationship	22
9. The Schema File: PARTSdmlldb.....	24
10. The LEX and YACC Parsing Process	26
11. The Merging of Data into the EWIRDB.....	38
12. Current Configuration of the M ² DBMS Network.....	43
13. Modification to the add_path() Function Call	45
14. The Original Query File: dmlreq1	47
15. The Parameter "Loading_Data_Flag".....	50
16. Terminating Character Added to the "dbid" Array.....	50
17. Record File (.r) Format	51
18. Original PARTS Database	54
19. PARTS Schema File: PARTSdmlldb.....	56
20. PARTS Record Data File: PARTS.r	56
21. PARTS Template File: PARTS.t.....	57
22. PARTS Descriptor File: PARTS.d	57
23. PARTS Query File: PARTS_QUERIES	58
24. PARTS Query Access and Navigation	61
25. PARTS2 Database.....	62

26. PARTS2 Schema File: PARTS2dml.db	63
27. PARTS2 Record Data File: PARTS2.r	63
28. PARTS2 Template File: PARTS2.t	64
29. PARTS2 Descriptor File: PARTS2.d	64
30. PARTS2 Query File: PARTS2_QUERIES	65
31. PARTS2 Query Access and Navigation	66
32. PARTS3 Database	67
33. PARTS3 Schema File: PARTS3dml.db	68
34. PARTS3 Record Data File: PARTS3.r	68
35. PARTS3 Template File: PARTS3.t	69
36. PARTS3 Descriptor File: PARTS3.d	69
37. PARTS3 Query File: PARTS3_QUERIES	70
38. PARTS3 Query Access and Navigation	70
39. PARTS4 Database	71
40. PARTS4 Schema File: PARTS4dml.db	72
41. PARTS4 Record Data File: PARTS4.r	72
42. PARTS4 Template File: PARTS4.t	73
43. PARTS4 Descriptor File: PARTS4.d	73
44. PARTS4 Query File: PARTS4_QUERIES	74
45. PARTS4 Query Access and Navigation	77
46. PARTS5 Database	78
47. PARTS5 Schema File: PARTS5dml.db	79
48. PARTS5 Record Data File: PARTS5.r	80
49. PARTS5 Template File: PARTS5.t	81
50. PARTS5 Descriptor File: PARTS5.d	81
51. PARTS5 Query File: PARTS5_QUERIES	82
52. PARTS5 Query Access and Navigation	86
53. Parametric Tree Structure [5]	92

54. The Pulsed/Continuous Wave (P/CW) Parametric Tree [3]	93
55. An Exploded View of the P/CW Tree [3].....	94
56. Global View of the Conceptual Schema for the EWIRDB	97
57. Conceptual Schema of the S&TI Emitter [3].....	99
58. Expanded View of the Antenna Data [3].....	101
59. Enhanced View of the Antenna Data.....	102
60. Network Conceptual Schema.....	105
61. The EWIR Schema	107
62. The EWIR Schema (continued)	108
63. The EWIR Template File: TEWIR.t.....	110
64. The EWIR Template File: TEWIR.t (continued).	111
65. The EWIR Descriptor File: TEWIR.d	112
66. The EWIR Record Data File: TEWIR.r	114

LIST OF TABLES

1. Basic CODASYL-DML Commands	29
2. Sample FIND Commands	30
3. Insertion and Retention Constraints	33

ACKNOWLEDGMENTS

We would like to thank Dr. C. Thomas Wu and Dr. David K. Hsiao for their time and guidance in the development of this thesis. Additionally, we would like to thank Tom McKenna and Kevin Coyne for their technical assistance.

We also feel it is important to recognize the support of the staff at the Naval Postgraduate School, particularly Mike Williams and Susan Whalen.

I. INTRODUCTION

This thesis describes our completed efforts to activate and test the network CODASYL-DML interface for the Multi-lingual, Multi-model Database Management System (M²DBMS). Also, documented within this thesis is the implementation of a representative portion of the Electronic Warfare Integrated Reprogramming Database (EWIRDB). This introductory chapter provides an overview of the M²DBMS to include the original motivation for the design and implementation of this system. Also discussed in this chapter are the specific objectives and organization of this thesis.

A. OVERVIEW

Over the past thirty years, the design and implementation of database systems from a software perspective has been virtually unchanged. The typical approach to the design and implementation of a database system has been to define a data model and a corresponding data language. The data model provides for the structure and the form of the data to be stored in the database, as well as a collection of the types of generic operations that are used to access the database. The part of the data model that allows the specification of the database is referred to as the data definition capability of the data model and the data manipulation strategy provides a means to specify database operations that are used to access the stored data.

This approach to database system development supports databases that are created using the single data model and supports applications that are written in the single, corresponding, model-based data language. The result to this approach is a proliferation of homogeneous, mono-lingual database systems that restrict the user to a specific data model and its corresponding model-based data language.

In the Naval Postgraduate School's Laboratory for Database Systems Research, a multi-database system prototype called M²DBMS (Multi-lingual, Multi-model DBMS)

has been researched and developed to overcome this restriction as mentioned above. The M²DBMS system supports heterogeneous databases, each of which is based on a different data model. The system executes transactions of the data language corresponding to each data model supported. This system provides the user with the ability to access and manage a large collection of databases, using several data models with their associated data languages.

The databases currently implemented on the M²DBMS system include: Object-oriented, relational, network, hierarchical, and functional. Accordingly, the system is capable of executing transactions written in OO-DML, SQL, CODASYL-DML, DL/I and DAPLEX. This system supports multiple databases not as a collection of separate systems, but rather with a single kernel data model and language called the attribute-based DBMS. All of the aforementioned heterogeneous databases are organized internally on the basis of the kernel data model. Additionally, all of the heterogeneous transactions are translated or "mapped" into their equivalent transactions in the kernel data language. This non-traditional type of an approach leads to a better data sharing and resource consolidation of heterogeneous databases. Furthermore, the multi-lingual, multi-model approach provides for an effective cross-model accessing capability. This capability enables a user, using a familiar data model and language, to access a database created according to an unfamiliar model and language by further transformations and translations between the models. These transformations and translations are transparent to the user. For example, a relational database user can access a non-relational database (i.e., object-oriented, network, etc.) using the inherit relational data transaction language, SQL. This cross-model accessing capability is unique to the M²DBMS prototype and it is this idea that captures the essence of integrating stand-alone databases into a unified, enterprise database. Figure 1 illustrates the cross-model accessing capability.

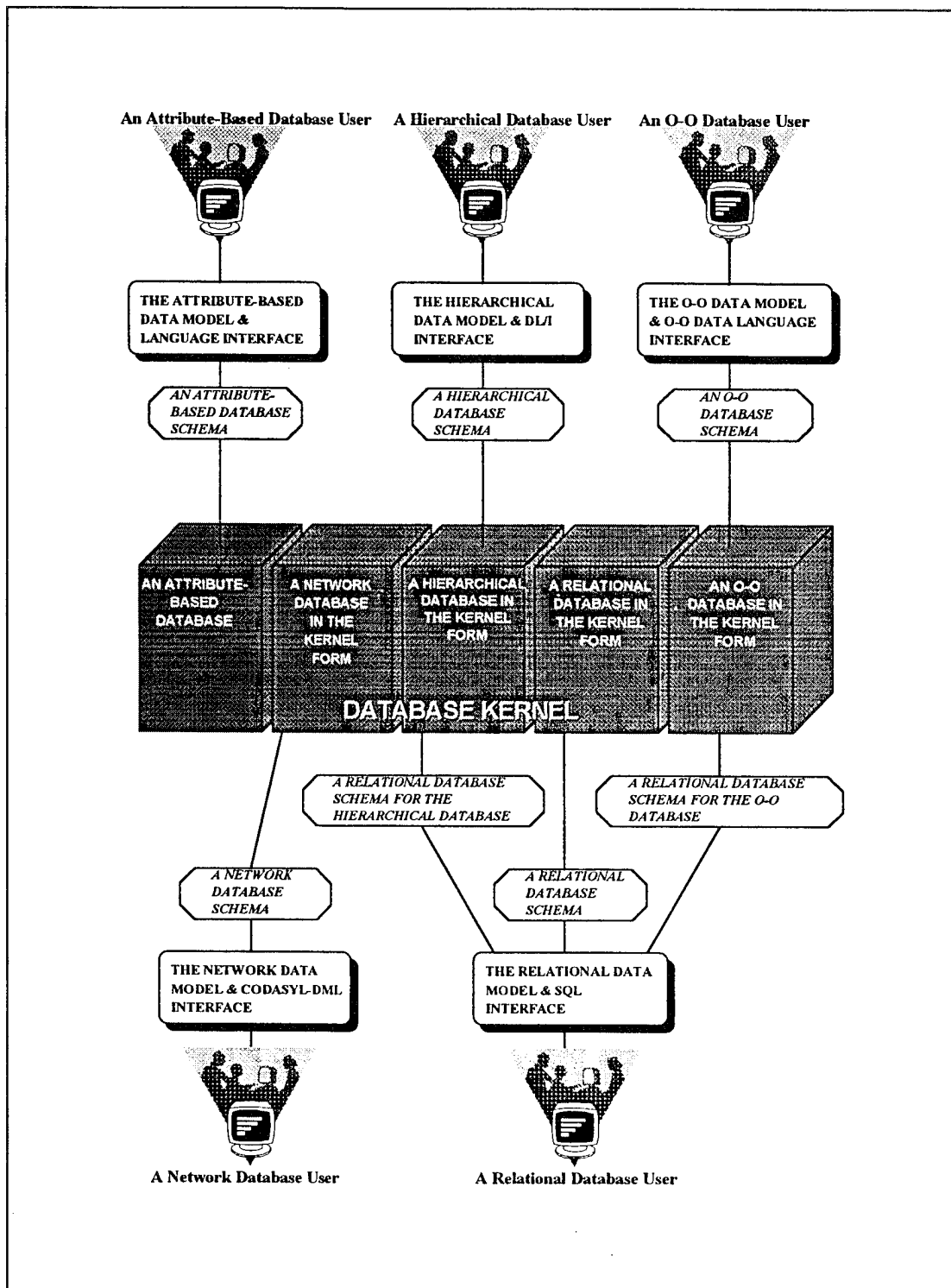


Figure 1. M²DBMS Cross-Model Accessing Capability.

B. MOTIVATION

Before the specific objectives of this thesis are addressed, we consider the significance and motivation of the M²DBMS or the multi-lingual, multi-model database system. Specifically, what are the benefits of a multi-lingual, multi-model database system?

This is especially evident when considering large, corporate and government institutions which often use a myriad of diverse database applications. Often, these types of institutions use a variety of different types of applications using a multitude of database models and languages. For example, four likely databases of one such organization may include a personnel database (supported by a relational database), an inventory database consisting of suppliers, parts and shipments (modeled by a network database), a manufacturing process database (manipulating a functional database) and a product assembly database consisting of assemblies, sub-assemblies and parts (maintained by a hierarchical database). In this example, it is obvious that a single data model and data language is not adequate; one data model and language may be appropriate for one application but not for another one. Thus, a conventional database system supporting a single data model and language is not ubiquitous enough to support the data management requirements of institutions requiring large, diverse database applications.

An alternative to a conventional database system supporting a single data model and language and the M²DBMS ideology is to implement several separate mono-lingual database systems; however, there are some significant problems with maintaining and using several separate mono-lingual databases. These problems include data sharing, data duplication, resource consolidation and training of personnel. Data sharing becomes a problem since the heterogeneous databases cannot communicate between each other, therefore, all information must be maintained by each system (even though the same data may already exist in another database system). Thus, storage capacity is squandered by maintaining the same data in these separate databases. Data duplication is just this

problem described – the same data stored in different model-based databases. Data duplication leads to data inconsistency problems that are created when one database is updated and another database with the same data is not updated. Resource consolidation is violated since hardware and software must exist for each separate system. This also incurs unnecessary maintenance responsibilities and costs. Finally, personnel that are proficient with one data model must be retrained to utilize another model effectively. This makes it more difficult to move personnel between jobs and results in increased training costs when moving personnel is necessary.

Obviously, if it were possible to establish a single data model and language as a universal database standard, then the problems previously addressed would not exist. However, the possibility of this occurring in the near future appears not likely. Although, the relational model has become rather popular over the years, many of the older models continue to persist. This is in a large part due to costs (personnel, training, software, hardware etc.) associated with converting to a new system but also for reasons illustrated by the example presented earlier, demonstrating the need for different models for different applications. Also, with the recent development of the object model, current trends seem to indicate that new data models will continue to emerge to meet the demands of new applications.

In light of this discussion, one can conclude that the M²DBMS system, a multi-lingual, multi-model database system offers a viable benefit over existing and alternative database systems. It provides attainable solutions to the problems previously discussed. First, data sharing and data duplication is essentially eliminated. This is contributed to the fact that all associated data model and data language interfaces (i.e., object-oriented, relational, network etc.) are based on a single kernel data model (using a kernel data language – the Attribute Based Data Language – ABDL) that requires each database language interface to store its data using the kernel data language. Likewise, resource consolidation is optimized since there is only one physical system with a portion of the software common to all interfaces, data models and languages. The cross-model

accessing capability eliminates any personnel retraining problems since it is this capability that enables a user, using a familiar data model and language, to access a database created according to an unfamiliar model and language.

By adding the necessary interface software, the M²DBMS system with cross-model accessing capability accommodates for any number and type of data models and languages while offering a database user access to a heterogeneous database as if it were a homogeneous database system.

C. THESIS OBJECTIVES

There are three objectives to this thesis. The first objective is to activate and test the network CODASYL-DML interface for the M²DBMS system. Although, the network model interface was originally implemented in 1985, over the years, this interface has become non-operational due to changes in hardware configurations, software modifications, interface updates and lack of overall system maintenance. Therefore, it is necessary to activate and test this interface thoroughly. Completion of this objective is necessary prior to executing the next (second) objective of this thesis.

The second objective is to design and implement a data model for a subset of the Electronic Warfare Integrated Reprogramming Database (EWIRDB) for the network interface. This database is the primary Department of Defense (DoD) approved source of electronic warfare (EW) data. Recently, a subset of the EWIRDB was implemented on the object-oriented interface. To continue research in cross-model accessing capability, it is necessary to first implement the EWIRDB in the various other data model interfaces.

The third and final objective of this thesis is to design, write and execute several transactions (queries) in the CODASYL-DML format for the EWIRDB.

Research is ongoing to implement the EWIRDB in the network and relational interfaces of the M²DBMS. Once completed, three operational interfaces (object-oriented, network and relational) will exist that implement a subset of the EWIRDB.

Hence, fulfillment of this requirement will provide a solid foundation to gain further progress in future cross-model accessing capabilities.

D. ORGANIZATION OF THE THESIS

In Chapter II of this thesis, background information is provided on the M²DBMS (structure, design, layout and functionality); the network data model (basic concepts and structure, data definition language and data manipulation language); and the EWIRDB (overview and description). Chapter III describes the initial state of the network CODASYL-DML interface coupled with program code modifications and corrections associated with returning the interface to an operational state. Also discussed therein is the results of testing the network interface. Chapter IV elaborates on the database specification of the EWIRDB to include the theoretical network EWIR data model. Also highlighted in this chapter is the data definition and record data of the EWIRDB. Chapter V concludes the thesis by summarizing the findings and results of this thesis.

II. BACKGROUND

This chapter provides background information on three areas of interest. First, a general overview of the M²DBMS is provided. This discussion briefly explains how the M²DBMS is configured from both a hardware and software perspective. Second, an overview of the network data model and the CODASYL-DML (query) language is described. Third, this chapter introduces the Electronic Warfare Integrated Reprogramming Database (EWIRDB), specifically its role, format and recent implementation of this database in the object-oriented interface of the M²DBMS.

A. AN OVERVIEW OF THE M²DBMS

There are essentially three integral components of the M²DBMS that highlight its design: A multi-back-end based database computer, a multi-lingual, multi-model database system and the kernel data model and kernel data language. The remainder of this section will examine each of these aspects separately but within the context of the overall system.

1. A Multi-Back-end Database Computer

The M²DBMS was originally designed for optimal performance, resource consolidation and a data sharing capability. The essence of the design was to overcome some of the problems and upgrade issues typical of a more traditional database system design. Nonetheless, it was designed to run on standard, off-the-shelf hardware networked on UNIX work stations. From a hardware and architectural point of view, the cornerstone of the system involved the use of multiple back-ends connected in a parallel configuration.

The purpose of the back-ends is to provide the storage and processing functions of the system. Each back-end consists of a single general-purpose workstation that

contains three data drives. One disk drive, a smaller Winchester-type, supports paging; another small disk accommodates for the meta-data (schema); and a larger hard disk supports the base data.

The base data of any particular database is clustered on the system back-ends. Clustering divides the base data across the back-ends in mutually exclusive sets. This kind of a distribution utilizes an efficient clustering algorithm that facilitates parallel access to the data and the execution of database transactions.

The parallel connection between each back-end is via an ethernet LAN using a point-to-point communication protocol for one-to-one communication between separate back-ends and a broadcast communication protocol for one-to-many back-ends [10].

The user interface as well as communication with the back-ends is supported by a controller (front-end). The controller is also a single general-purpose workstation like that used by the back-ends. The controller is connected to the back-ends via a communication bus and to the computer science department's LAN via a "gateway." The controller receives database transactions from the users and broadcasts these transactions to the back-ends (communication is executed through the communication bus via sockets). The back-ends in turn, return the results to the controller for post processing and routing to the users. Figure 2 illustrates a simple depiction of this architecture.

The M²DBMS architecture based upon the use of multiple parallel back-ends provides for increases in performance and capacity. The advantages offered to gain performance by increasing the number of back-ends are known as *response-time reduction* and *response-time invariance* [18]. Response-time reduction conveys that if additional parallel back-ends are added to the system, a reduction in response time will occur that is directly proportional to the number of back-ends added. Likewise, the response-time invariance refers to the idea that if the size of the database is increased and if a proportional number of back-ends is added then the response time is invariant. [19]

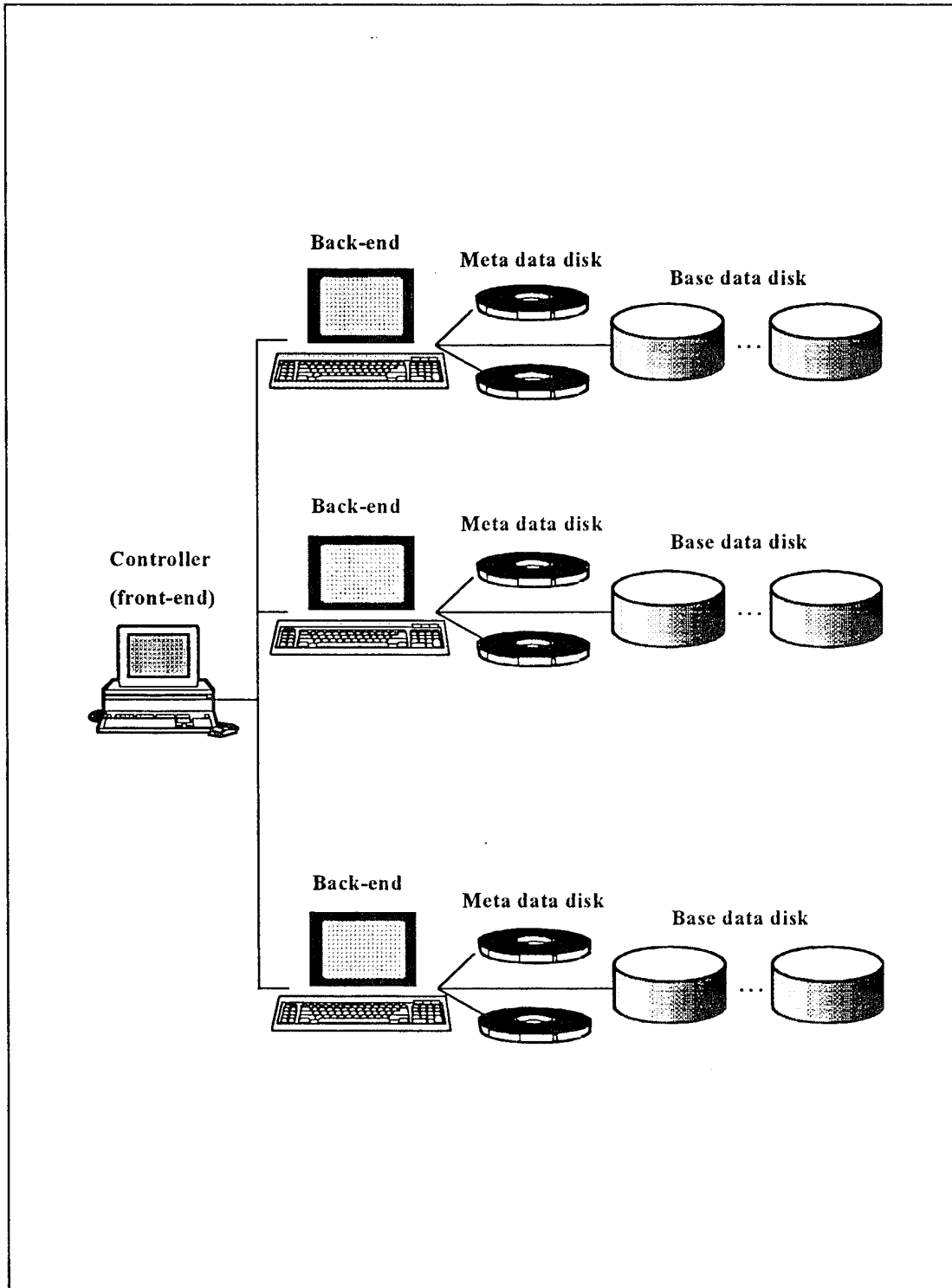


Figure 2. The Multi-Back-end Database Computer.

Through this ability to add any number of back-ends to the M²DBMS, greater performance gains and storage capacity are acquired. This capability can be implemented with minor modifications to the software. Thus, unlike some of the more traditional database systems; M²DBMS supports a scalable architecture without costly modifications or replacements to gain overall system performance and efficiency.

2. A Multi-Lingual, Multi-Model Database System

The architecture of the multi-back-end database computer establishes a solid basis for the M²DBMS software. The base operating system underlying the software layer is UNIX Berkeley Software Distribution (BSD).

There are five language interfaces (not including the attribute-based data model and language) implemented on the system, however, only three are currently operational (object-oriented, network and relational). The language interface software for each language interface transforms its respective data model and data language into the attribute-base data model and language. This capability allows the users to create, maintain and manipulate databases of different data models and languages on one system, thus implies the multi-lingual, multi-model designation.

For each language interface, there are four program modules. Figure 3 (the shaded area) displays the four modules of a given language interface. These modules are the Language Interface Layer (LIL), the Kernel Mapping System (KMS), the Kernel Formatting System (KMS) and the Kernel Controller (KC). Each of the modules on the M²DBMS has been coded using the C programming language. A description of the general interaction among these modules with the subsequent system is provided in the following four paragraphs.

Figure 3 shows a diagrammatic representation of the interaction between the User Data Model (UDM) and the User Data Language (UDL) and the four modules of the language interface as well as the interaction between the Kernel Database System (KDS) and the four modules. The KDS is comprised of the Kernel Data Model (KDM) and the

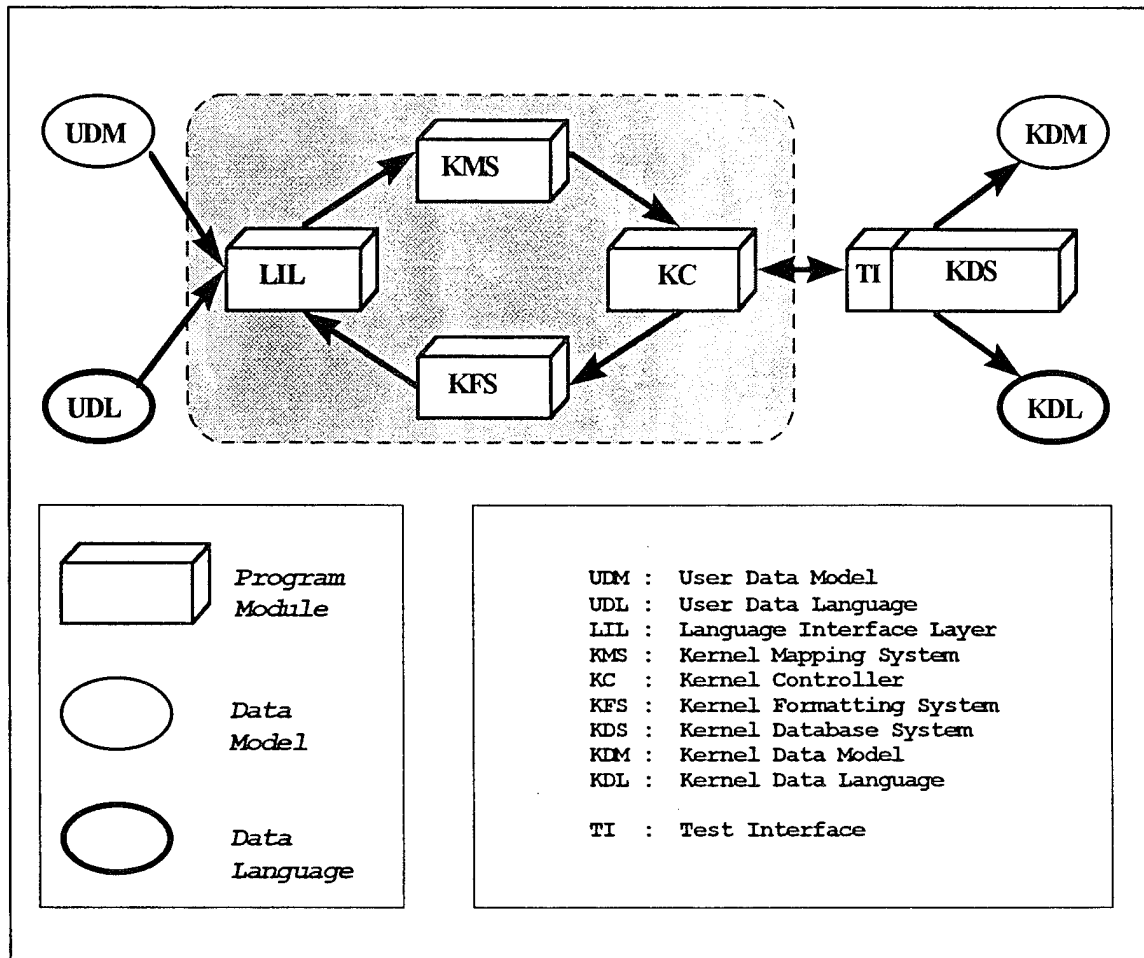


Figure 3. The Multi-Lingual and Multi-Model Database System.

Kernel Data Language (KDL). The Test Interface (TI) within the KDS provides the means in which the four modules of a language interface can interact with the KDS. To begin, the user first interacts with the system through the LIL, using a chosen UDM to issue transactions written in a corresponding model-based UDL. The LIL then forwards these transactions to the KMS. The KMS is a compiler that transforms the UDM and the UDL into a form that can be mapped to the KDS. This implies that the KMS can serve one of two purposes. It either transforms a UDM based database definition (database schema) to an equivalent database definition based on the KDM, or when the user

specifies the execution of a transaction, the KMS translates the UDL to an equivalent transaction in the KDL.

The first purpose is referred to as *data definition transformation*. This occurs when a new database is created. Upon successful transformation of a UDM database definition, the KMS forwards the resulting KDM database definition to the KC. The KC then passes the KDM database definition to the KDS where the new database is defined on the system. Once, the KC finishes processing the KDM database definition, it, in turn notifies the user through the LIL that the database definition has been processed and loaded.

The second purpose of the KMS is referred to as *data-language translation*. This process occurs when the KMS translates transactions written in a specific UDL into equivalent KDL transactions. This is performed in a manner similar to that described with the first purpose of the KMS. The KMS forwards the transactions to the KC which in turn passes the transactions to the KDS for execution, once this execution is complete, KDS sends the results to the KC in KDM format. The KC then forwards the results from the KDS to the KFS. The KFS reformats the KDM results into the appropriate UDM format. The KFS then displays the results through the LIL in the correct UDM form.

Figure 4 shows a depiction of the various model-language interfaces as implemented on the M²DBMS. From this figure one can ascertain the commonality in the software design inherent with each language interfaces. One of the main goals when originally implementing M²DBMS was to achieve a high degree of uniformity and consistency across the various language interfaces. Therefore, as each language interface was designed, uniformity and consistency was maintained throughout the specification of the program structure for each language interface (in terms of the program modules), the communication paths of a language interface (between program modules), the data structures for each language interface and the global variables of a language interface [7]. Hence, to construct and implement a new user data model and language does not require

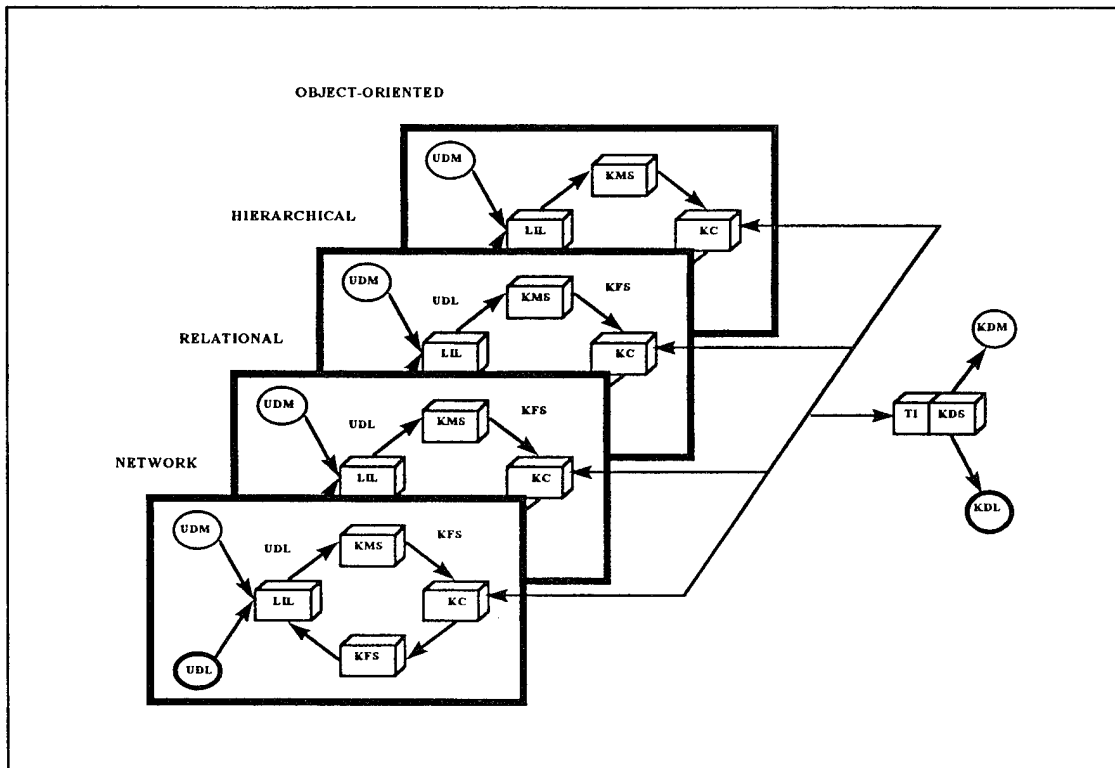


Figure 4. Multi-Lingual and Multi-Model Interface Design for M²DBMS.

a redesign of the entire database system. This concept proved quite effective with the recent implementation of the object-oriented interface.

3. The Kernel Data Model and Kernel Data Language

The attribute-based data model (ABDM) and the attribute-based data language (ABDL) are the kernel data model (KDM) and kernel data language (KDL) of the M²DBMS. The first question one may ask concerns which data model and language should be selected as the KDM and KDL for the M²DBMS. Selecting the proper data model and language for the KDM and KDL was a key decision in the development of the M²DBMS. Before addressing those factors that contributed to the selection of ABDM and ABDL as the KDM and KDL, it is important to review the original requirements of the M²DBMS.

The original requirements for the M²DBMS design included optimal performance as well as resource consolidation and a capability for data sharing. In essence, a system to support many data models and their languages on a single system (i.e., a multi-lingual, multi-model system) without a significant degradation in performance (as compared to the more traditional single model systems). The underlying premise to meet these requirements was to store all data in a single kernel (i.e., the kernel database system). To accomplish this, the M²DBMS uses various mapping functions and algorithms to map the different data models and languages into a single data model and language. Thus schema definitions and transaction requests developed for the traditional data models and their respective languages are either transformed or translated into equivalent schemas of the kernel data model or equivalent transactions in the kernel data language for processing in the kernel database system.

In light of the previous discussion, the following factors established the basis for choosing the ABDM as the kernel data model for the M²DBMS:

- The capability to transform and translate (using mapping) the other traditional languages such as SQL, CODASYL-DML, and DL/I for processing.
- The separate modeling of base data and meta data.
- The clustering of base data into mutually exclusive sets for storage on the back-ends.
- The ability for parallel access to the clustered data.

B. THE NETWORK DATA MODEL AND ITS LANGUAGE

Historically, the network model's structures and language constructs were defined by the CODASYL (Conference on Data Systems Languages) committee, so it is often referred to as the CODASYL network model. Originally, the network model and language were presented in the CODASYL Data Base Task Group's (DBTG) 1971

report; this is sometimes referred to as the DBTG model [6]. This report placed the network model in the public domain, and commercial implementation began shortly thereafter. Development and standardization efforts by ANSI (American National Standards Institute) continued throughout the late 1970's and early 1980's. Any references made to the concepts, structures and characterizations of the CODASYL network model herein are based on the Data Description Language Committee's (DDLC) 1981 *Journal of Development*. The term used in the remainder of this thesis is network model or network data model rather than CODASYL model or DBTG model.

1. Basic Concepts and Structure

Networks can generally be represented by a mathematical structure called a directed graph. Directed graphs are constructed from points or nodes connected by arrows or edges. In the context of the network data model, the nodes can be thought of as data record types, and the edges can be thought of as representing relationships (one-to-one and one-to-many). Thus, the network data model represents data in network structures of record types connected in one-to-one (1:1) or one-to-many (1:N) relationships. This type of structure serves useful in the representation of many hierarchical type relationships.

As previously mentioned, there are two basic structures in the network model, records and sets. Data is stored in records and each record consists of a group of related data values. Records are classified into record types, where each record type describes the structure of a group of records that store the same type of information. Each record type is given a name and format (data type) for each attribute (data item) in the record type. Figure 5 shows the record type PARTS for the PARTS database. Note the PARTS database is used throughout this thesis. This was the original database implemented on the network interface. References [1] and [2] refer to this simple database, thus to provide consistency our research and examples also incorporate the PARTS database.

PARTS				
PNO	PNAME	COLOR	WEIGHT	CITY

<u>Attributes</u>	<u>Format</u>
PNO.....	CHARACTER 6
PNAME.....	CHARACTER 20
COLOR.....	CHARACTER 6
WEIGHT.....	FIXED 4
CITY.....	CHARACTER 15

Figure 5. The Record Type PARTS.

A typical database application has numerous record types, from a few to a few hundred. Representing the various relationships between the records are set types. [6]

A set type expresses a 1:N relationship between two record types (note that the reference to set does not imply the usual mathematical definition of a set). A set type consists of the following:

- A set type name
- An owner record type
- A member record type

These conventions are illustrated in the PARTS database in Figure 6. This figure shows the general form of the data structure. This type of a diagram is called a

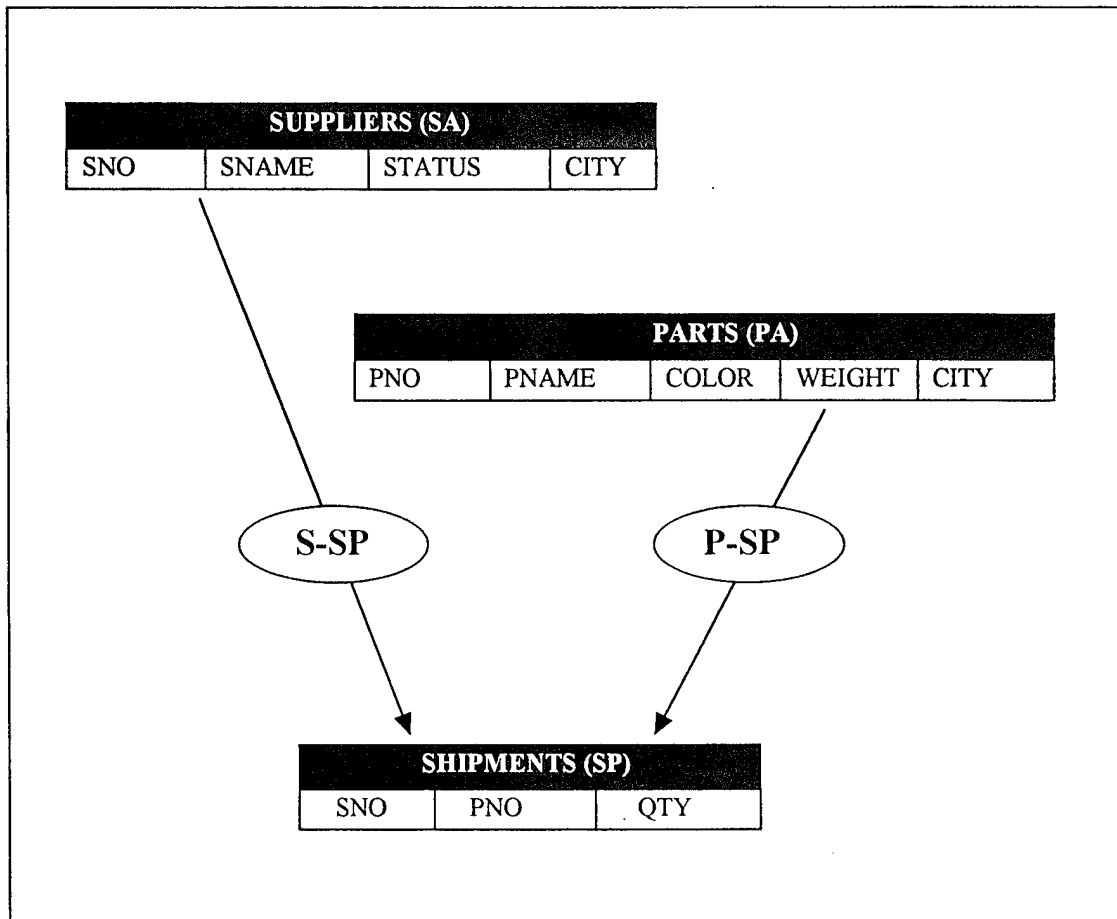


Figure 6. The PARTS Database.

“Bachman diagram” in honor of Charles Bachman, who proposed the idea of data structure diagrams to capture the structure of set types among record types.

The symbols used in this diagram (Figure 6) are as follows: First, sets are denoted by the arrow between record types, with the arrow pointing to the member record type (the “many” or N in the 1:N relationship). Second, each set type name is enclosed in the oval. This convention will be used during the specification of the EWIR database.

The PARTS database has two sets: The suppliers-shipments (S-SP) and the parts-shipments (P-SP). For the set, S-SP, the suppliers record type is the owner record and shipments record type is the member record. Accordingly, the parts record type is

the owner record and the shipments record type is the member record for the P-SP set. In this example, the 1:N relationship incorporates the possibility that zero, one or many (zero or more) shipment records may be related to a given supplier record; or zero or more shipment records may be related to a given parts record. For the S-SP relationship, this implies that at any given time, a supplier may have, say, 15, one or zero shipments in progress. So, in the database itself, there will be many set occurrences (or set instances) corresponding to a set type. At this juncture, it's important to emphasize an important restriction when defining network databases, namely, a set cannot be defined with the same record type (i.e., a record type cannot be both the owner and member record types in a set). Furthermore, no two set occurrences of a set may have records in common. This qualification highlights the pairwise disjointness of set occurrences of a given set type. Figure 7 illustrates a set occurrence for the set type, S-SP of the PARTS database.

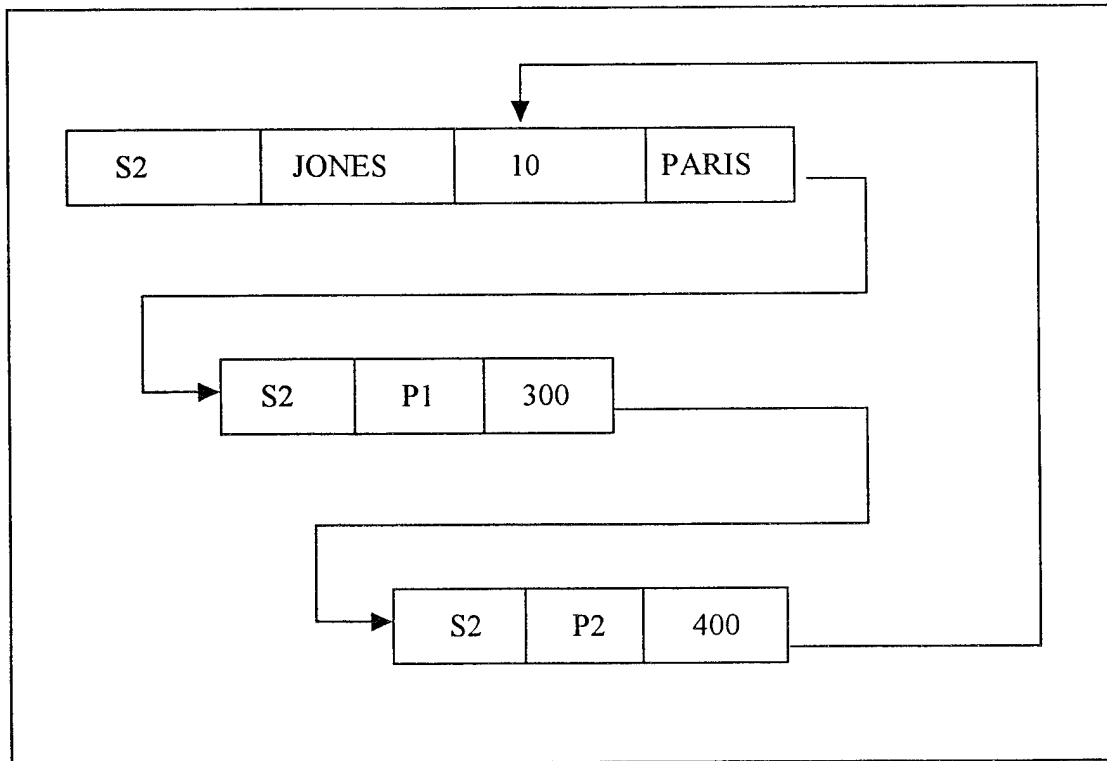


Figure 7. A Set Occurrence (S-SP) in the PARTS Database.

There are several representations to illustrate a set occurrence but the one depicted in Figure 7, where the records of the set occurrence are shown linked together by pointers, is a generally accepted convention that closely resembles the structure in the implementation.

Additionally, there are situations where a relationship is strictly 1:1, such as with a truck and driver (for each truck there is only one driver). This implies that a record of the member record type can appear in only one set occurrence. Thus we must restrict each set occurrence to having a single member record. The network model does not provide for automatically enforcing this constraint, so it should be monitored that this constraint is not violated when a member record is inserted into a set occurrence.

Also note that the PARTS database provides an example that demonstrates the difference between the network data model and the hierarchical data model. Notice that the shipments record type is a member record type of two sets: S-SP and P-SP. In the hierarchical data model, a record type cannot be a member of two different sets. However, this is permitted in the network model, thus providing an additional representational capability. Essentially the hierarchical data model is a subset of the network model.

Thus far, 1:N and 1:1 relationships have been discussed. However, the question still remains how many-to-many (M:N) relationships are represented in the network model. An M:N relationship cannot be directly represented in the network model. Some M:N relationships can be converted to two 1:N relationships. Consider the M:N relationship between STUDENT and CLASS on the left side of Figure 8 (note the M:N relationship is "diagrammatically" denoted as a line with an arrowhead on each end). When two record types, such as STUDENT and CLASS are connected in a M:N relationship, a dummy record (also called a linking record type) can be used. This linking record type uses at least the keys from STUDENT and CLASS records. Other attributes may be added at the discretion of the designer. Figure 8 (right side) also shows the conversion of the M:N relationship to a 1:N relationship in the network model.

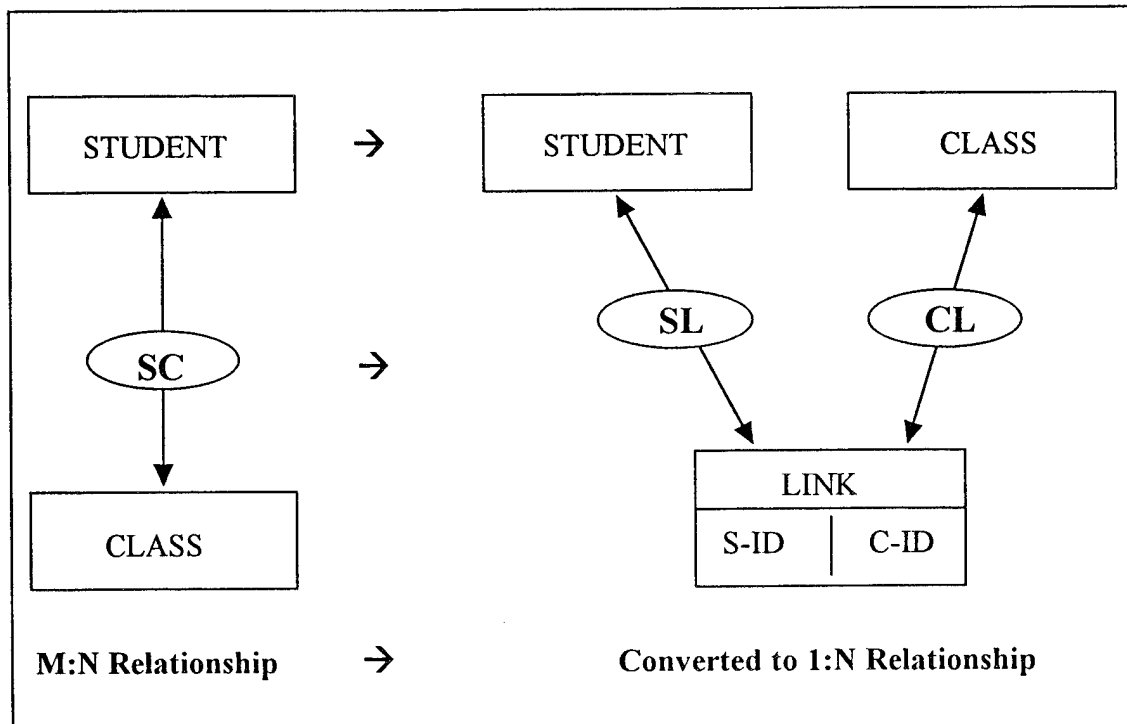


Figure 8. Conversion of M:N Relationship to 1:N Relationship.

Other M:N relationships in an initial design may represent design errors often resulting in the loss of information. Therefore, careful examination is required to ensure the proper relationships exist or that unnecessary relationships are eliminated between record types. Creating artificial records such as linking (dummy) record types inherently requires additional storage and processing requirements, but the network model is now in a more simple form that satisfies the original network model standards [20].

The next two sections of this thesis describe the languages by which the network model is implemented. These languages describe how the data structures are specified, how the data is stored, and how data is manipulated. The Data Definition Language (DDL) is used to define the schema and the Data Manipulation Language (DML) is used to manipulate the data or perform operations on the data.

2. Data Definition in the Network Interface

The network DDL provides the constructs to define a schema for a database. The schema may be defined as the description of a database. The schema declares all the record types, set types, attribute definitions and various constraints (when operations are performed) to the network database. The schema does not produce a database, but describes what one should look like. Figure 9 shows the schema for the PARTS database. The file name for this schema on the M²DBMS is **PARTSdml.db**. The line numbers are not part of the original file but will be used for illustration purposes below. The schema in Figure 9 is described as follows:

- Line 1. assigns the schema a name (supplied by the user).
- Lines 2.-15. are defined as the record section which provides specifications of each record structure, its data items (attributes), and its location. Each record type (Lines 2., 7., and 12.) is identified by name (i.e., SA, PA, SP). For each record type, the component data items or attributes are defined. The record type SA has SNO, SNAME and CITY as its data items. Each data item has a specific format that indicates the data type and length (i.e., CHARACTER 10).
- Lines 16.-27. are defined as the set section. Once all the records have been defined, then the sets may be defined. Definition of a set includes naming the set type and identifying the owner and member record types of the set. For example, the set type SSP (line 16.) has SA as its owner and SP as its member record type.
- Lines 3., 8., 19.-21., and 25.-27. Are related to specifying various constraints on the system. These constraints are applicable when operations are performed on the sets. These operations include insertion, deletion and update operations on sets. Since these operations are performed using the

data manipulation language, they will be discussed in context of the network CODASYL-DML in the next section of this thesis (section 2).

- Line 28., the dollar symbol identifies the end of file.

```
1.  SCHEMA NAME IS PARTS;
2.  RECORD NAME IS SA;
3.    DUPLICATES ARE NOT ALLOWED FOR SNO;
4.    SNO ; CHARACTER 10.
5.    SNAME ; CHARACTER 10.
6.    CITY ; CHARACTER 10.
7.  RECORD NAME IS PA;
8.    DUPLICATES ARE NOT ALLOWED FOR PNO;
9.    PNO ; CHARACTER 10.
10.   PNAME ; CHARACTER 10.
11.   CITY ; CHARACTER 10.
12.  RECORD NAME IS SP;
13.    SNO ; CHARACTER 10.
14.    PNO ; CHARACTER 10.
15.    QTY ; FIXED 4.
16.  SET NAME IS SSP;
17.    OWNER IS SA;
18.    MEMBER IS SP;
19.      INSERTION IS AUTOMATIC
20.      RETENTION IS FIXED;
21.      SET SELECTION IS BY VALUE OF SNO IN SA;
22.  SET NAME IS PSP;
23.    OWNER IS PA;
24.    MEMBER IS SP;
25.      INSERTION IS AUTOMATIC
26.      RETENTION IS FIXED;
27.      SET SELECTION IS BY VALUE OF PNO IN PA;
28.  $
```

Figure 9. The Schema File: PARTSdml.db.

The overall concept for the network data definition process on the M²DBMS is as follows. First, since all physical data resides in the kernel; the network data model must be mapped (mapping implies translating from one data model to another data model) to

its corresponding Attribute-based Data Model (ABDM). This mapping process is somewhat complicated, since the correspondence between network data constructs (i.e., owner records, member records, sets and set occurrences) and attribute-based data constructs becomes intricate when we encode the bi-directional aspects of the network data into the attribute-based data [7]. The key aspect of the mapping process is the retention of the CODASYL notions of records and sets. The CODASYL notion of records and sets is not the same as the attribute-based model's notion of records and sets. As a result of this problem, it was necessary in the original design to introduce an additional facility to individually identify every record in the database. Thus, the attribute "DBKEY" was added to uniquely identify each record. Furthermore, for each network record type, every set for which that record type is a member must be represented. This is accomplished by using a special keyword. The keyword "MEM" which proceeds the set names (i.e., MEMSSP for the SSP set in the PARTS database), catalogs the set name and the set occurrence of which a particular record is a member. This keyword is also used to refer to the owner record of a particular set occurrence. References [2] and [7] describe the various transformations required to map the network data model to the attribute-based data model in detail. The mapping from the network data definition language to the ABDM occurs through the Kernel Mapping System (KMS) via a parser-translator. The KMS parser-translator was programmed using two compiler generation tools provided by UNIX, the LEX and YACC. The LEX (Lexical analyzer generator) is a program generator designed for lexical processing of character input streams. Given a regular-expression description of the input strings, LEX generates a program that partitions the input stream into tokens and provides these tokens on demand to the parser. YACC (yet-another-compiler compiler) is a program generator designed for the *syntactic* processing (using a finite-state automaton) of a tokenized input stream. [7] Figure 10 illustrates the LEX and YACC parsing process.

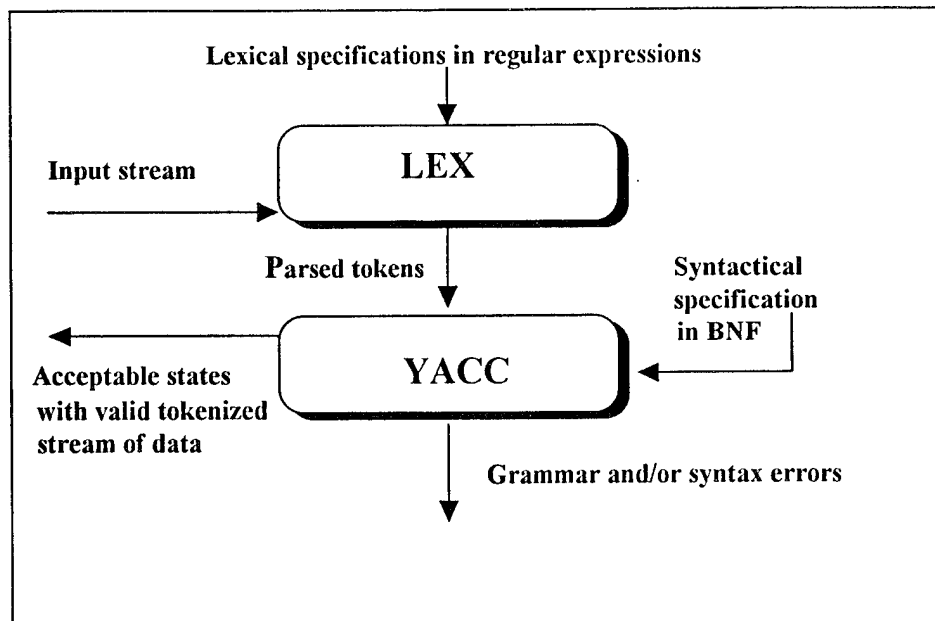


Figure 10. The LEX and YACC Parsing Process.

When the user specifies the filename of a schema (i.e., **PARTSdml**db****) to create a new database, the transaction list of network data definition statements is sent to the KMS via a program loop. This loop in essence traverses the transaction list, calling KMS for each data definition statement in the list. Once the LEX receives a data definition statement, the tokenized version of the definition is passed to the compiler generated by YACC. This compiler takes the tokenized version and verifies it against the set of CODASYL-DML grammar rules which describe the network model's grammar (i.e., a Backus-Naur Form (BNF) representation of CODASYL-DML). As the compiler generated by YACC parses the tokenized database definition, it creates a database schema in the network model form for future conversion into the kernel data model language, ABDL. [7] Other than entering the schema file name, the process just described is transparent to the user. The schema file that the user specifies (i.e., **PARTSdml**db****) is a "source" schema. The source schema is prepared by the user or a programmer. It is this schema that must be converted into a computer readable form

(i.e., an "object" schema) [22]. The object schema is the one created by the compiler generated by YACC.

3. Data Manipulation in the Network Interface

Once the database is designed, created and defined using the DDL, the data manipulation language (DML) enables the user to manipulate the data or perform operations on the data.

The DML used for the M²DBMS network interface is the CODASYL-DML. The CODASYL-DML is a procedural language that is embedded in a host language. The CODASYL-DML is rather extensive, but the M²DBMS incorporates only a small subset of the language commands. Reference [22] provides an in-depth review and description of the CODASYL-DML. The syntax for the CODASYL statements represented on the M²DBMS was derived from reference [22] and the original CODASYL and DBTG literature referenced earlier.

The CODASYL-DML operators process records one at a time, unlike the relational DMLs where operators process entire relations at one time. The operations in a CODASYL database are performed by "navigating" through a tree, network or set occurrences.

Before the actual command statements of the DML are described, it is important to first review the concept of *currency*. The concept of currency is analogous to the notion of "current position." It is a generalization of the familiar notion of current position within a file. Currency determines the access point for the "navigation" through the network. The starting point for the navigation usually begins with the current record of the run unit. The run unit is the current transaction being executed.

The basic idea is that for each application being run on the system (i.e., for each run unit), a table of currency indicators is maintained. A currency indicator is an object whose value at any given time is either null (meaning that it currently identifies no record) or the address of a record in the database, commonly called a database key. A currency

indicator is actually a pointer which points to the record or set most recently accessed by that run unit. The database keys are values generated by the DBMS that uniquely identify each individual record in the database. [23]

Essentially currency indicators function as place markers. When a command is issued and a specific record is found, its “place” is marked by a currency indicator. When a second command is issued, the DBMS refers to the currency indicator to determine which record is to be acted upon. The type and function of the currency pointers are as follows:

- Current of the run unit: As stated previously, run unit refers to the user’s program (transaction) currently being executed. Current of the run unit contains the address of the record, or set instance most recently accessed by the application program.
- Current of the record type: There is a currency indicator for each record type, the address of the record of that type most recently accessed by the program is maintained.
- Current of the set type: A currency pointer that contains the address of the most recently accessed record of a given set type. A separate currency pointer is maintained for each set type. The pointer may point to a record of either the owner or member type depending on which was mostly recently accessed.

These pointers just discussed are updated automatically as accesses in the database take place. A good way to vision currency indicators is as variables in a table.

The basic commands used by the CODASYL-DML support the primary database operations. This thesis will briefly discuss the basic commands implemented on the M²DBMS. These commands are classified as navigational, retrieval and update (for

records and sets) commands. Table 1 illustrates how the commands are grouped by type for purposes of discussion in this thesis.

TYPE	COMMAND
Navigation	FIND
Retrieval	GET
Record Update	STORE
	ERASE
	MODIFY
Set Update	CONNECT
	DISCONNECT

Table 1. Basic CODASYL-DML Commands.

The FIND command is the key command of the CODASYL-DML. This command establishes the currency of the run unit, and may also be used to establish the currency of the record type and set type. FIND commands select and locate a desired record or instance of a set, thus the reason it is referred to as a “navigational” type command. A GET command must then be used to actually retrieve the data (the current occurrence) and return the results to the user. There is a wide variety of FIND commands. Table 2 illustrates the various types of FIND commands. This is only a sample of the more common FIND commands.

The general syntax or format of the FIND command is as follows:

⇒ FIND ANY <record name> [USING <field list>]

The optional parts of the command are denoted by the brackets ([...]) and the names supplied by the user are indicated by the angle brackets (<...>).

TYPE	USE	REMARKS
FIND ANY	Identifies the first record occurrence of any record type in the network database.	Can be used without regard to currency. Often used to set a new currency for the database.
FIND FIRST (LAST)	Identifies the first (last) member record of a set occurrence.	Currency is set to the newly found record.
FIND NEXT (PRIOR)	Used after the FIND FIRST (LAST) command to step through the member records of the set occurrence.	Resets the currency after each instance-at-a time operation.
FIND OWNER	Identifies the owner record occurrence of the current record occurrence of a particular set type.	Sets the current run unit.

Table 2. Sample FIND Commands.

The CODASYL-DML also provides two commands which are not typical database commands. These commands are the GET and MOVE commands. As previously mentioned, the GET command retrieves the current occurrence and returns the result to the user. The GET command compliments the FIND commands and is issued following the FIND command to retrieve the record occurrence as set by the FIND command. The GET command can obtain access to an entire record or it can obtain access to only specific data items (attributes) in the record type (i.e., GET items IN <record type>). The MOVE command is not in the original CODASYL-DML but has been added to the implementation. It is used to initialize the values (i.e., assignment statements) of data item names for record types. The MOVE command precedes various other commands in the CODASYL-DML.

The record update commands (see Table 1) include the STORE, ERASE and MODIFY commands. The STORE command is an insert operation used to place a new record occurrence into the database (the STORE command is not currently operational in

the network interface). This is executed by first constructing a sequence of MOVE commands to create the "record image." Once the record image has been created, then the proper set occurrence for the record must be selected by the DBMS. This is specified by an insertion constraint specified in the schema definition. This constraint is the "INSERTION IS AUTOMATIC" clause (see Figure 9, lines 19. & 25.). This clause tells the DBMS that the new member record must be automatically connected to an appropriate set occurrence when the record is inserted.

The set occurrence in which the new record is stored is determined by the SET SELECTION clause specified in the schema definition (file) for the database (see Figure 9, lines 21. & 27.). There are three options available for specifying SET SELECTION:

- SET SELECTION BY APPLICATION: The application program (transaction) is responsible for selecting the correct occurrence; therefore, the new member record is automatically connected to the current set occurrence.
- SET SELECTION BY VALUE: The system selects the proper occurrence based on data item values specific to the owner of the set occurrence desired. In other words, specify an attribute of the owner record type whose value is used to specify a set occurrence by identifying the owner record of the set.
- SET SELECTION BY STRUCTURAL: 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 (member record) being stored.

The last two options stated above (SET SELECTION BY VALUE and SET SELECTION BY STRUCTURAL) must have the constraint DUPLICATES ARE NOT ALLOWED for the specified field in the owner record (see Figure 9, lines 3. & 8.). This is to ensure a unique owner record is identified and thus a unique set occurrence.

The ERASE command is a deletion operation used to remove records from the network database. There are essentially two forms of this command. The ERASE and

ERASE ALL commands. The ERASE command removes a single record from the current record type of the run unit. The ERASE ALL command removes each member record of the set. It is important to note that for the ERASE command, the current occurrence is removed only if the record is not an owner record occurrence in a non-empty set. However, for the ERASE ALL command, this restriction does not hold, therefore it is possible to destroy all connections between records.

The MODIFY command is used to modify values of data items (attributes) in a record occurrence. This includes modifying all data items or any subset of the data items in the record type. This command 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.

The set update commands (see Table 1) are the CONNECT and DISCONNECT commands. The CONNECT command is used when the user wants to manually insert a record occurrence into the database. When the CONNECT command is used, an insertion constraint (option) must be specified in the schema definition. This constraint is the "INSERTION IS MANUAL" clause. The record to be inserted is the current record of the run unit. The set occurrence in which the record is inserted is determined in the same manner as the STORE command.

The DISCONNECT command is used when the user wants to manually remove a record occurrence from a set. This command disconnects the current record of the run unit from the occurrence of the specified set that contains the record. The record occurrence still remains in the database but it is not a member of any specified set. In order for this to occur, a retention constraint must be specified in the schema definition. This constraint is the "RETENTION IS OPTIONAL" clause which specifies that a member record can exist on its own without being a member in any occurrence of a set. In Figure 9 (lines 20. & 26.), the constraint "RETENTION IS FIXED" is used. This constraint specifies that a member record cannot exist on its own. Table 3 provides a summary of the insertion and retention constraints and their allowable combinations.

<i>Insertion Constraints</i> ↓	<i>Retention Constraints</i>		
	OPTIONAL	MANDATORY	FIXED
MANUAL	Application program handles inserting member record into set occurrence. <u>USE: CONNECT & DISCONNECT</u>	NOT VERY USEFUL	NOT VERY USEFUL
AUTOMATIC	DBMS inserts a new member record into a set occurrence automatically. <u>USE: CONNECT & DISCONNECT</u>	DBMS inserts a new member record into a set occurrence automatically.	DBMS inserts a new member record into a set occurrence automatically.

Table 3. Insertion and Retention Constraints.

The overall concept for the network CODASYL-DML process on the M²DBMS is essentially the same as the DDL process. As with the DDL, the CODASYL-DML statements (commands) must be mapped to ABDL requests (this translation process exceeds the scope of this thesis, but reference [2] provides a complete data language translation from CODASYL-DML to ABDL). This mapping process occurs through the KMS via the same parser-translator used for the processing of DDL statements. However, rather than transforming DDL operations, the KMS, in this case, translates DML operations to equivalent ABDL requests. Furthermore, the input strings to the LEX are for transaction specifications rather than DDL specifications. Once the transactions have been specified, the tokenized transaction is passed to the compiler generated by YACC and verified against the network grammar. As the compiler parses the tokenized transactions, the mapping of the CODASYL-DML transactions to ABDL occurs. During the parsing and verifying of the transaction against the grammar, the KMS also does a great deal of semantic checking of the transaction. This includes

checking to see if all of the different identifiers referenced by the transaction are defined. These identifiers include, but are not limited to, the record type names and the data item (attribute) names in the network language interface. Another semantic check determines if type consistency is maintained. For example, if "QTY = 'Monterey'" is part of a transaction for the PARTS database, obviously there is a type conflict between the identifier, QTY and the string 'Monterey'. To accomplish the semantic checks, the KMS utilizes the network database definition that was stored during the database loading. At the completion of this phase, the compiler generated by YACC outputs a list of one or more ABDL transactions that are equivalent to the input CODASYL-DML transaction. This list of ABDL transactions is then passed back to the LIL, which forwards the list to the KC for execution. [7] Ultimately, the transaction ends in the kernel formatting system (KFS) where the output of the transaction is displayed to the user.

Since the M²DBMS is a research project, it does not contain all the functionality, characteristics, and operations expected in a full-scale commercial product. Only the essential operations for retrieval, deletion, modification and insertion of records and sets are implemented in the network interface. Some of the high-level characteristics that are not supported in the network interface include:

- The ability to define subschemas (views)
- The ability to specify security constraints
- The conditional operational type
- Various operations that support concurrency control

A comprehensive review of the network data model, the DDL and the DML is beyond the scope of this thesis. The intent of this section was to provide a brief introduction to the network data model and illustrate the basic concept and appearance of the DDL and DML. Having familiarity with the network DDL and DML will facilitate

future discussion of the development of the EWIR database subset described in Chapter IV.

To formally evaluate the network model is not an objective of this thesis. However, a few observations that may indicate the strengths and weaknesses of the network data model may prove advantageous, especially when considering the design of the EWIR database. It will be left to the reader to formally evaluate the network data model or to conduct a comparative assessment with the various other data models. As with most design ventures, picking the proper tool (i.e., data model) for the right job is an important consideration. Therefore, depending on the design requirements and analysis, the network model may prove to be the most applicable data model. Summarized below are some general observations of the characteristics of the network model.

- Networks can be very complicated (complex structure).
- 1:N relationships between two record types are established by explicit definition of set type. Records in each set type are connected by physical pointers, thus records are physically connected when they participate in the same set occurrence.
- Essentially two modeling constructs: The record type and the set type.
- Data manipulation tends to be “navigational” in the sense that a user must access a database by explicit traversal through a tree or network, rather than stating the properties of the data of interest.
- Transactions are very tightly constrained by the structure of the record links, and inter-record links tend to be expressed at a very low level, thus an experienced programmer is sometimes required to specify new transactions.
- Data is arranged in a fairly “fixed” structure.
- Inter-connections among data items are not easily molded into a variety of semantic interpretations.

- Provides for a strong capability for protecting the integrity of sets (i.e., insertion and retention constraints).
- Data duplication is sometimes necessary to model multiple networks and to support M:N relationships.
- Concept of currency can be complex and the user (or programmer) should have an understanding of currency indicator tables to avoid errors.
- May yield good performance on a comparative scale.
- Network data model appears to be suited for database systems characterized by large size, well-defined repetitive queries, well-defined transactions and well-defined applications. If this is true, then extensibility may be a problem (i.e., difficult to extend with an unanticipated future application, thus possibly requiring an entire database system reorganization).

Despite whatever conclusions can be drawn from the aforementioned observations, it is interesting to note that because of a large number of DBMSs that were implemented using the network data model, a very high percentage of the databases currently used for day-to-day production in industry and government subscribe to the network model [24].

C. AN OVERVIEW OF THE EWIR DATABASE

As stated in Chapter I, the implementation of a representative portion of the Electronic Warfare Reprogramming Database (EWIRDB) using the network interface is one of the objectives of this thesis. However, before the actual data model and overall implementation of the EWIRDB using the network interface are discussed in Chapter IV of this thesis, it is necessary to first provide a brief introduction and overview of the EWIRDB.

To date, a representative portion of the EWIRDB has been implemented using the object-oriented interface of the M²DBMS [4]. Reference [3] provides an in-depth study of the design (conceptual and logical) and analysis of the EWIRDB for the object-oriented implementation. Likewise, reference [4] details the implementation of the EWIRDB on the M²DBMS. Currently, work is continuing to implement a portion of the EWIRDB in the network (this thesis) and relational (research group Edwards/Scrivener) interfaces of the M²DBMS.

“The EWIRDB is the primary Department of Defense (DoD) approved source for technical parametric and performance data on noncommunications emitters and associated systems [5].” Its primary purpose is to provide the most current and accurate source of information for reprogramming US electronic Warfare (EW) combat systems such as radar warning receivers, combat identification systems, electronic jammers, anti-radiation missiles and various other target sensing systems.

Originally, the EWIRDB was conceived to support software reprogramming of EW systems employed by US combat forces. The EW (combat) systems as mentioned in the preceding paragraph, serve to enhance wartime survivability and effectiveness. Now, the EWIRDB not only supports reprogramming, but serves the additional purposes of supporting EW systems research, development, test, and evaluation; modeling and simulation; acquisition; and training. [5]

The analysis required to populate the EWIRDB and support reprogramming includes specific, in-depth parametric data on radars, jammers, navigational aids, identification friend or foe (IFF) equipment, and a variety of other noncommunications electronic emitters. Also required to support reprogramming includes an emphasis on the assessments of wartime reserve modes and electronic protection capabilities of “foreign” emitters that may degrade EW system performance and thus, the overall survivability and effectiveness of combat forces. In order for EW to be successful, it requires an efficient system and database to collect, store, analyze, maintain and update data. Data may be obtained either directly from measurement or indirectly via electronic intelligence. The

National Air Intelligence Center (NAIC) maintains the latest data, in-depth and specific, on EW systems of the United States, friendly forces, and non-friendly forces. It is this data that is stored in the EWIRDB.

The EWIRDB was developed initially by the US Air Force in the 1970's but has evolved into a joint product. The EWIRDB is now the product of merged data modules from three separate organizational components: The National Security Agency (NSA); the Scientific and Technical Intelligence (S&TI) community, under the jurisdiction of the Defense Intelligence Agency (DIA); and the United States Noncommunications Systems Database (USNCSDB), supported by the Air Force Information Warfare Center (AFIWC).

The NSA provides *observed data* (from their "kilting" database) that results from the direct measurement and analysis of an emitter's electromagnetic signature following the signal intercept. This data assists in describing an emitter's performance. The S&TI community contributes parametric data assessments to the EWIRDB. Systems analysts of this community consider all available sources of information and then estimate or derive the total operational capability of an emitter (derived parametric data in the EWIRDB are referred to as *assessed data*). The USNCSDB maintains data on US owned and operated noncommunications emitters. Analysts of this agency provide inputs based on evaluation of system specifications (also *assessed data*). Figure 11 depicts the merging of data from the three sources into the EWIRDB. [3]

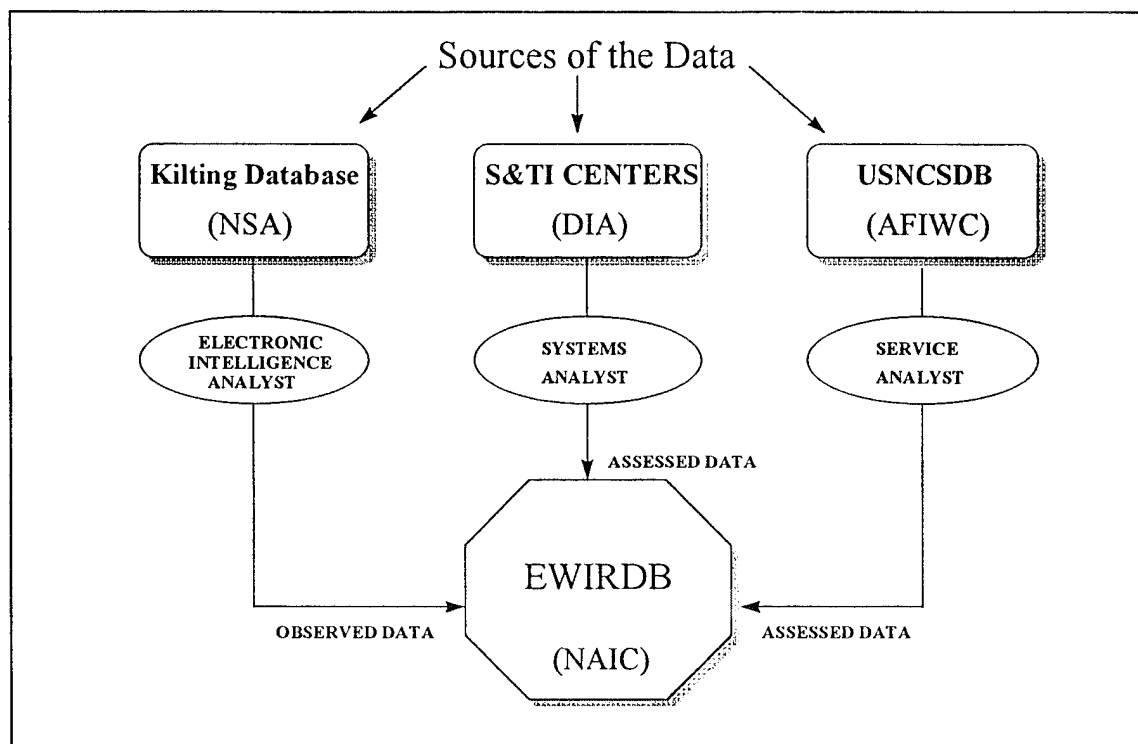


Figure 11. The Merging of Data into the EWIRDB.

Although the EWIRDB can prove to be effective in its implementation, users of the present database system mutually agree that there are difficulties in reading and querying the EWIRDB. These difficulties are related to:

- A problematic data model. Data is represented in a hierarchical tree structure that obscures the users view and meaning of the data.
- A complex format where assessed and observed codes are not standard for all record types.
- A database described in terms of the physical data structure.

This discussion provided a general overview of the EWIRDB. In Chapter IV of this thesis, a more detailed analysis of the EWIRDB will be provided.

III. ACTIVATION OF THE NETWORK INTERFACE

One of the main objectives of this thesis was to determine the current operational state of the network CODASYL-DML interface. If the interface was found to be non-operational, then the goal was to troubleshoot, diagnose, debug and isolate the errors to return the system to its original state. Once the interface was returned to its original state, the next objective was to test the limitations of the interface. This chapter details our findings in accomplishing the above stated goals. The contents of this chapter include: The initial state of the M²DBMS and the network interface; the modifications and corrections implemented to reinstate the network interface; the implementation of a mass load function; and the test findings, resulting in the limitations of the system.

A. INITIAL STATE OF THE NETWORK INTERFACE

When our research began, the network interface was not operational. The only operational interface was the object-oriented interface. The object-oriented interface was recently implemented with research ongoing when the research began for this thesis [4]. The failure of the network interface to operate was apparently discovered when the M²DBMS was modified for operation on the SUN system in June 1993, and was annotated in the "main" function of the `ti.c` file (outer shell that runs the main menu) [16]. No documentation was included as to why the interface did not work, who had discovered that it was not working or how it was tested to determine it was not working. This proved to be an ongoing theme for the network interface. A general lack of documentation (i.e., who, what, when, where, why and how) made for a slow and cumbersome process to return the network interface to an operational status.

The original design and implementation of the Network interface was conducted by Emdi, B. [1], and Wortherly, C. [2]. Their work provided the basis for the design, implementation and a general overview of the "implementational" program code at the

time of their implementation. Reference [1] clearly stated that the Kernel Controller (KC) module was never tested due to an upcoming hardware change to the M²DBMS. In lieu of testing the KC with the Test Interface (TI), software stubs were created to perform the same function as the actual TI procedure. No further documentation could be found that indicated if or how the network interface was tested, and to what extent the interface was operational. Furthermore, upon comparing the original code included in reference [1] with the current code implemented on the system, it became increasingly more obvious that modifications were made to the original code with little or no documentation. The entire M²DBMS has been ported to different configurations several times since the network interface was originally implemented in 1985 [1]. Research conducted by Watkins, S. [16] describes one such porting (Watkins' research proved valuable in understanding the system configuration and communication). However, since the writing of reference [16], additional hardware modifications have been implemented making this reference somewhat obsolete.

B. MODIFICATIONS AND CORRECTIONS

This section begins with a description of the current M²DBMS configuration in conjunction with an overview of the initial directory structure. Also documented within this section are the various interface and program code modifications implemented to return the network interface to an operational state.

1. System Configuration and Directory Structure

The current configuration for the M²DBMS is with the front-end code residing on the front-end (controller), db11, and the back-end code on db13 (back-end). Currently, there is only one back-end that is operational. The terminal db12 is configured as the server for the M²DBMS but also has the potential to be configured as another back-end. Due to a security issue with the Naval Postgraduate School's LAN (December 1995), the

M²DBMS network was isolated from the Computer Science Department's LAN. In February 1996, the terminal db10 was added as a "gateway" to allow remote access (via ftp and rlogin) to the M²DBMS network from the Computer Science Department's LAN. Figure 12 illustrates the current network configuration for the M²DBMS.

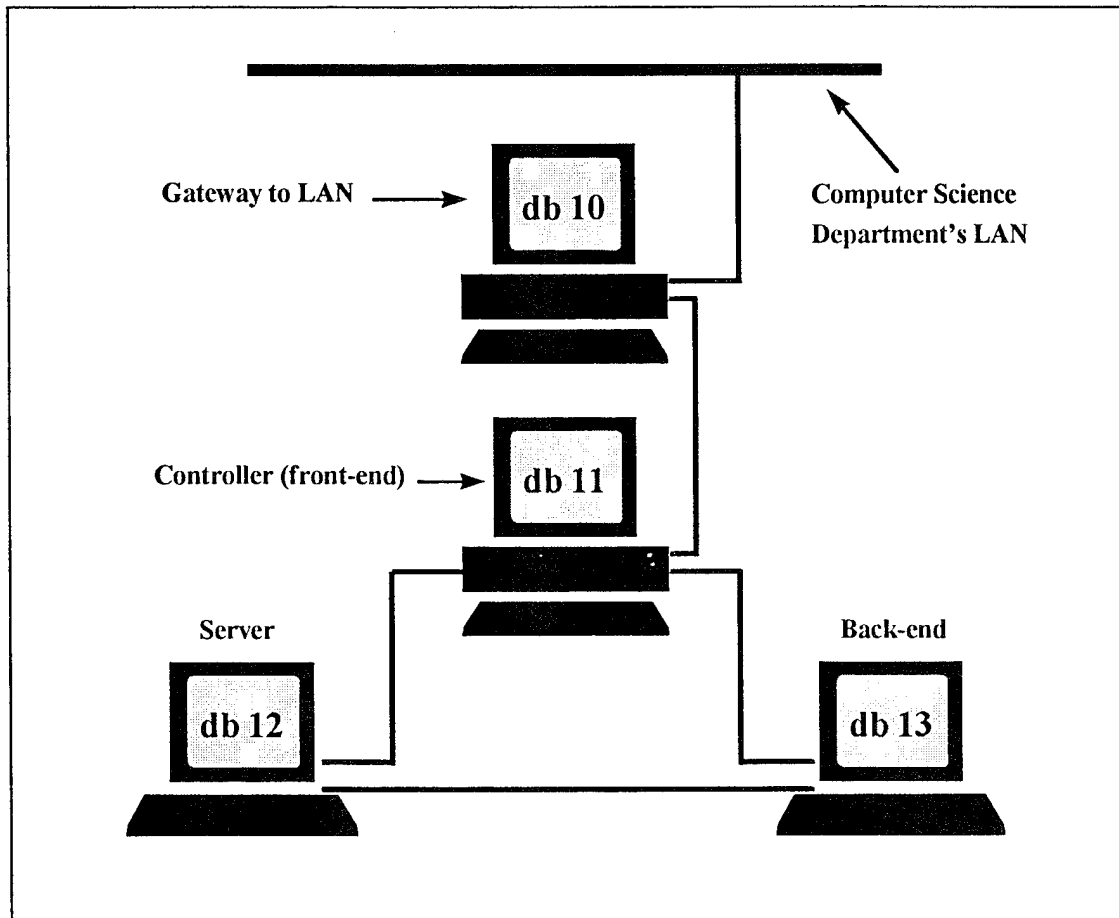


Figure 12. Current Configuration of the M²DBMS Network.

The code in the "greg" directory (the top directory in tree structure containing all the files and directories for the system) on the front-end (db11) is considered to be the master copy for all the system files and directories of the M²DBMS. To prevent "contamination" of the "original" front-end code on db11 and to support our research

efforts coupled with the research conducted by Edwards & Scrivener, two new complete directories were created ("*werre*" & "*edwards*") containing all the "original" files and directories supported in the "*greg*" directory. Additionally, the back-end (db13) required the creation of two new directories to support the back-end executable files and trace files. These directories are named "*be.werre*" and "*be.edwards*." This ensured interference would not occur between research groups during troubleshooting and testing. The *UserFiles* directory is the only common directory used by each language interface. A complete system tape back-up was also completed prior to any modifications being made on the system. Upon completion of our research and that of Edwards & Scrivener, the modified files and directories will be ported back to the master copy and tested. This master copy will remain in the "*greg*" directory.

2. Interface and Program Code Modifications

The initial attempt to run the M²DBMS network interface was unsuccessful. Reference [8] proved invaluable in providing information to assist in debugging, compiling, reconfiguring front and back-ends, changing versions and file organization. It should be noted, however, that hardware and configuration changes have occurred since the writing of reference [8] making it somewhat inaccurate. Figure 12 illustrates the current configuration vice those of reference [8].

The first problem encountered in operating the network interface was a modification made to the *add_path()* function described in [16]. The *add_path()* function is used to add the correct directory path to the file the user wishes to access (i.e., all schema, descriptor, template and data files are located in the *UserFiles* directory vice the current run directory). This function was changed in 1993 but not modified in every data model and language interface. This change required the passing of an additional parameter in the *add_path()* function call and is illustrated in Figure 13.

Add_path(FileName)	char location[40]
	add_path(location, FileName)
<i>(Original version)</i>	<i>(Modified version)</i>

Figure 13. Modification to the add_path() Function Call.

This change was made to every instance of the *add_path()* function in the *Dml* directory. Additionally, the DATA_AREA variable in the **licommon.h** file was modified to reflect the correct path to the *UserFiles* directory.

The function call *create_net_db_list()* was found inactive (i.e., commented out) in the main function of the **ti.c** file. This comment was removed to activate this function call. This enabled the network interface to initialize and allowed troubleshooting of the **lil.c** file.

When tested, numerous run time errors were detected in the **lil.c** file which is the main language interface shell for the network interface. A run time error in the function call *n_process_old()* contained within the **lil.c** file was created by the following lines of code:

```
⇒ DBL_S$Use(dml_info_ptr→dmi_curr_db.cdi_dbname);
⇒ TI_S$AssignDB(cuser_net_ptr→ui_uid.dml_info_ptrdmi_curr_db.dci_dbname);
```

These lines of code were made inactive to enable the network interface to run. No apparent negative effects were detected.

Also, a run time error in the function call *dml_show_schema()* was corrected by making the second part of the while loop inactive (delimited by the comment symbols */*, */*), as shown below:

```
⇒ while (rec_ptr) /*&&(strcmp (rec_ptr→nrn_name,BLANKSPACE)!=0)) */
```

Several other minor modifications in the *lil.c* file corrected the remaining run time errors. The menu instructions for the M²DBMS network interface could now be followed without run time errors to the extent of executing transactions (queries). This included the loading of the **PARTS** database schema which is further discussed in the final section (“Testing the Network Interface”) of this chapter. Note, however, that data has not yet been loaded. The files for the **PARTS** database were found in the *UserFiles* directory. The schema was defined in the **PARTSdml.db** file. This schema loaded correctly and created the corresponding descriptor and template files (**PARTS.t** & **PARTS.d**) for the **PARTS** database. Because of certain system limitations all descriptor and template files must be “manually” copied (or moved) to the back-end, and placed in the *UserFiles* directory on db13. Further discussion of the schema, template, descriptor and record files will be provided in the final section of this chapter.

In all prior research work conducted on the network interface, the issue of loading the data was never discussed. The object-oriented and relational interfaces employ a mass load utility (i.e., *mass_load()* function) that allows for the direct loading of data onto the back-end into the Kernel Database System (KDS) in ABDL format. However, the network interface lacked this utility. From our research, it appears that the data was originally intended to be loaded by executing various STORE (one of the commands of the network CODASYL data manipulation language) transactions. The first five network CODASYL-DML queries of the original query file (see Figure 14) for the **PARTS** database appear to indicate that the STORE command was an attempt to store the necessary data on the back-end prior to executing further transactions.

```

1. MOVE BUY1 TO SNO IN SA
   MOVE DEC TO SNAME IN SA
   MOVE MONTEREY TO CITY IN SA
   STORE SA
   @

2. MOVE BUY2 TO SNO IN SA
   MOVE IBM TO SNAME IN SA
   MOVE SANJOSE TO CITY IN SA
   STORE SA
   @

3. MOVE BUY3 TO SNO IN SA
   MOVE NCR TO SNAME IN SA
   MOVE ATLANTA TO CITY IN SA
   STORE SA
   @

4. MOVE PP1 TO PNO IN PA
   MOVE MMU TO PNAME IN PA
   MOVE MONTEREY TO CITY IN PA
   STORE PA
   @

5. MOVE PP2 TO PNO IN PA
   MOVE DATABUS TO PNAME IN PA
   MOVE SANJOSE TO CITY IN PA
   STORE PA
   @

.
.
.
.

11. MOVE BUY1 TO SNO IN SP
    FIND ANY SP USING SNO IN SP
    GET SP
    @

12. MOVE BUY1 TO SNO IN SP
    MOVE PP1 TO PNO IN SP
    MOVE 300 TO QTY IN SP
    FIND ANY SP USING SNO IN SP
    MODIFY SP
    $

```

Figure 14. The Original Query File: dmlreq1.

Numerous attempts and modifications were implemented to execute these transactions but these attempts proved unsuccessful, often resulting in run time errors. Thus the need for a “data load” utility still remained. Although implementing a mass load function or some form of a “data-load utility” was not originally considered an objective of this thesis, it was obvious that this functionality was needed, especially to complete further testing of this interface and to implement the EWIRDB.

C. MASS LOAD FUNCTION

Since loading data could not be accomplished by using transactions that attempted to STORE data, a mass load function was implemented to accomplish this task. The mass load functionality already existed for the object-oriented and relational interfaces. Upon reviewing the program code for the mass load function underlying these interfaces, it was decided to model the *mass_load()* function for the network interface after the object-oriented interface.

The new mass load function for the network interface was designated as *n_mass_load()*. It is found in the file **mass_ld.c** in the **Lil** directory. Detailed code for this function can be obtained in Appendix A of this thesis. In addition to adding the **mass_ld.c** file, the following significant modifications were made to the code:

- Relocated the **commdata.def** file to the **Lil** directory. This file contained the *struct* definitions used in the *n_mass_load()* function.
- Added the mass load option to the user interface menu in the *n_process_old()* function contained within the **lil.c** file.
- Added an additional parameter to the *dml_check_requests_left()* function. Since the *n_mass_load()* function calls the *dml_check_requests_left()* function located in the **chreq_left.c** file in the **KC** directory, it was necessary to add another parameter to the *dml_check_requests_left()* function call.

This modification was necessary to differentiate between calling this function for loading data versus retrieving data. Figure 15 depicts the parameter *Loading_Data_Flag* which is passed as TRUE when calling the *dml_check_requests_left()* function from the *n_mass_load()* function. This prevents the *dml_check_requests_left()* function from calling the *n_file_results* function, thus alleviating any run time errors when loading the data. The *Loading_Data_Flag* is set to FALSE when called from the *dml_execute()* function to allow filing the results of the FIND request.

Until now, all of the modifications in implementing the *n_mass_load()* function were local to the **Dml** directory. Therefore, none of the other interfaces were affected by these modifications. However, when trying to run the mass load function a problem was encountered in the *open_db_file()* function contained in the **utilities.c** file found in the **version/COMMON** directory (this file is used by all the interfaces). This function takes a database file name (stored in a character array called *dbid*); determines the type of file extension (i.e., *-.r*, *-.t*, or *-.d*); removes the extension and passes the database filename minus the extension back to the calling function. The problem existed in the *dbid* array after the file extension was removed. The array would be "contaminated" with an unwanted carriage return at the end of the array. This was remedied by forcing a terminating character to the end of the array once the file extension was removed (see Figure 16). This correction must be made to the master "version" in the "greg" directory. This change has been tested with the other interfaces with no adverse affects.

```

dml_check_requests_left (file_ptr, owner_key_flag, Loading_Data_Flag)
struct net_file_info *file_ptr;
int *owner_key_flag;
int Loading_Data_Flag;
{ ...

msg_type = TI_R$Type ();
switch (msg_type) {
    case CH_ReqRes:
        TI_R$ReqRes (&rid, response);
        done = last_response_chk ();
        if (response[1] == CSignal)                /* no response from backend */
            results_are_not_returned = TRUE;

        /* if request is a Find and no response received, the record doesn't exist */

        if((dml_info_ptr->dmi_operation==FindReq)&&
            (results_are_not_returned==TRUE))

            printf("\n Error-THIS RECORD DOES NOT EXIST IN THE DBASE
\n\n");
        else if (Loading_Data_Flag == TRUE) /* if just loading data */
            printf("Loading Data \n");
        else{ /* if retrieving data, need to file results */
            t= n_file_results(file_ptr, owner_key_flag);
            if(t> file_ptr->nfi_max_chars)
                file_ptr->nfi_max_chars = t;
        }
        break ;
}

```

Figure 15. The Parameter "Loading_Data_Flag".

```

Strncpy(dbid, fptr->base, len-2) /* removes last 2 characters from file name */

dbid[len-2]='\0'; /* adds terminating character to end of filename */

```

Figure 16. Terminating Character Added to the "dbid" Array.

To use the mass load function the user must first generate a template file and descriptor file (.d & .t) using the DDL (data definition language) compiler. These files are necessary because they provide the environment that will maintain the relationships between the attribute value pairs. The user must then manually create a record file (.r extension) that contains the record data in the proper sequence as determined by the template file created by the DDL compiler when loading the schema file. The format for the record file can be seen in Figure 17.

```
PARTS3
@
Pa
1 Pp1 Mmu Monterey
2 Pp2 Databus Sanjose
3 Pp3 Harddrive Salinas
@
Sp
4 Buy1 Pp1 100 1
5 Buy2 Pp2 50 2
6 Buy3 Pp3 75 3
7 Buy3 Pp1 50 1
8 Buy2 Pp1 100 1
$
```

Figure 17. Record File (.r) Format.

The *mass_load()* function is a process consisting of four steps. First, the function will open the “record” file (.r file) and check for a match between the database name in the file and the name of the current executing database. The function will then check to see if the database name in the file is in agreement with the database name currently executing. These must agree or the function will abort. If the names match, then the data read is recognized as the template name. The function will then open the

template already on the back-ends using the “other pointer process” embedded within the function.

Next, the *mass_load()* function will read the data from the record file one by one. With each read, the function will read an attribute name from the template file. The matching of a data element and an attribute name will create the attribute value pair. As pairs are created, the function creates an INSERT statement in the attribute data language for each individual item read. This processing continues until the ampersand (@) is encountered. The ampersand symbol acts as the demarcation between templates. When encountered, the function will stop processing and read the next template. Processing continues until the dollar (\$) symbol is encountered. The dollar symbol marks the end of the file.

Once the end of file is encountered, the *mass_load()* function passes the INSERT request statements to REQP in the Kernel System. The REQP receives these INSERT statements throughout the TI and checks each statement for proper format and syntax. If all of the statements pass the error checking, the INSERTs are executed and completed. [9]

During our testing, it was discovered that all data must be entered with the first letter capitalized and all additional letters in lower case. The reason for this is because of the function *fix_up_ABDL_req()* found in the Kc directory. For example, the following transaction is used to find any SA record that has a SNO of BUY1:

```
MOVE BUY1 TO SNO IN SA
FIND ANY SA USING SNO IN SA
GET SA.
```

This transaction produces the following request message:

```
[RETRIEVE ((TEMP=SA) and (SNO = BUY1)) (SNO, SNAME, CITY, DBKEY) BY  
DBKEY]
```

Before being sent to the back-end to retrieve the requested data, the message is modified by the function *fix_up_ABDL_req()* to produce the following message:

```
[RETRIEVE ((TEMP = Sa) and (SNO = Buy1)) (SNO, SNAME, CITY, DBKEY) BY  
DBKEY]
```

The reason for this modification is not really known. We can only assume that this was done to ensure consistent format of the data in the transactions. Additionally, if the data were able to be loaded via STORE transactions vice the *mass_load()* function, the data stored in the back-end would be in this format and therefore consistent throughout.

It was also found that only alphabetical characters are permitted as both data values and attribute names (no dashes or underscores). This made readability of attributes and data somewhat confusing since you could not separate words.

D. TESTING THE NETWORK INTERFACE

Now that the interface is operational and data can be loaded, the capabilities of the interface were investigated. It should be noted that our goal in the testing phase was to determine operational limitations, not necessarily to follow on and identify and correct the code that caused these limitations. Therefore our discussion is from the users viewpoint by way of determining what "types" of schemas can be realistically used and

the limitations of these schemas. Correcting, modifying or expanding the capabilities is left for follow-on research.

We discovered that the only database developed for the interface was the PARTS database referenced in [1] and [2] and illustrated in Figure 18. How or if this interface was tested remains unknown. Our testing strategy was to determine if several basic transactions (queries) could be performed on several variations of the PARTS database. The commands used for the queries were the “Find Any” command for an individual record, the “Find Member Record” command to find the member record within the current set, and the “Find Owner” command to find the owner record in the current set. These basic queries allow us to: Verify accessing a record at a given “level” (level refers to the nesting level of a specific record type within a diagrammatic network schema); to navigate “downward” in the record structure hierarchy to access a record in a set; and to navigate “upward” to find the owner of a record in a set.

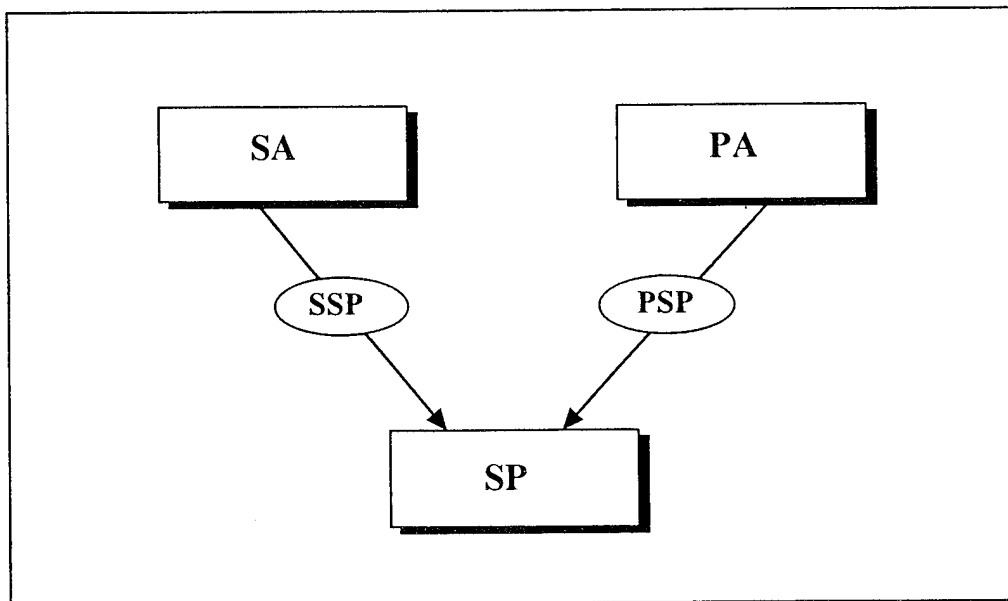


Figure 18. Original PARTS Database.

The next five subsections describe our test findings as a result of testing the aforementioned queries on different variations of the PARTS database. First, the original PARTS database was tested and subsequently four different versions of this database were evaluated. Each subsection illustrates a diagrammatic representation of the database (schema) in conjunction with its associated schema file, record data file, template file, descriptor file, query file, and a query access and navigation diagram.

1. The Original PARTS Database

We began our testing with the original PARTS database as depicted in Figure 18. The schema file, record data file, template file, descriptor file, and query file associated with the PARTS database are shown in Figures 19, 20, 21, 22, and 23 respectively. The schema file, **PARTSdmlldb** already existed in the *UserFiles* directory. The template file and the descriptor file were also available in the *UserFiles* directory and are recreated every time the “**PARTSdmlldb**” schema is loaded. Unless the schema file is modified, the template and descriptor files are only moved to the back-end *UserFiles* directory once. The template file is necessary in assisting the user in developing the properly formatted record file that contains the data for the database. The record file for the original PARTS database is the **PARTS.r** file. The queries used for testing the PARTS database are located in the query file named **PARTS_QUERIES**. Note that the numbers identifying each query in the query file are for illustration purposes and are not part of the actual query file (**PARTS_QUERIES**). This remains true for all subsequent query files.

Queries 1, 2 and 3 were executed properly indicating that each record could be accessed directly. Queries 4 and 5 were used to test navigating from owner records in SA and PA to set member records in SP (navigating downward). Both of these queries were successful. Queries 6 and 7 were used to determine if the owner could be found for a record in SP that belonged to the set(s) SSP and/or PSP. Query 6 ran successfully but query 7 failed.

```

SCHEMA NAME IS PARTS;
RECORD NAME IS SA;
  DUPLICATES ARE NOT ALLOWED FOR SNO;
  SNO ; CHARACTER 10.
  SNAME ; CHARACTER 10.
  CITY ; CHARACTER 10.
RECORD NAME IS PA;
  DUPLICATES ARE NOT ALLOWED FOR PNO;
  PNO ; CHARACTER 10.
  PNAME ; CHARACTER 10.
  CITY ; CHARACTER 10.
RECORD NAME IS SP;
  SNO ; CHARACTER 10.
  PNO ; CHARACTER 10.
  QTY ; FIXED 4.
SET NAME IS SSP;
  OWNER IS SA;
  MEMBER IS SP;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF SNO IN SA;
SET NAME IS PSP;
  OWNER IS PA;
  MEMBER IS SP;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF PNO IN PA;
$

```

Figure 19. PARTS Schema File: PARTSdmlldb.

```

PARTS
@
Sa
1 Buy1 Dec Monterey
2 Buy5 Jan Sanjose
3 Buy2 Oct Sanjose
4 Buy4 MAR Sanjose
5 Buy3 Mar Salinas
@
Pa
6 Pp1 Mmu Monterey
7 Pp2 Databus Sanjose
8 Pp3 Harddrive Salinas
@
Sp
9 Buy1 Pp1 100 1 6
10 Buy2 Pp2 50 3 7
11 Buy3 Pp3 75 5 8
12 Buy2 Pp1 50 3 6
13 Buy2 Pp1 100 3 6
$

```

Figure 20. PARTS Record Data File: PARTS.r.

```

PARTS
3
5
Sa
TEMP s
DBKEY i
SNO s
SNAME s
CITY s
5
Pa
TEMP s
DBKEY i
PNO s
PNAME s
CITY s
7
Sp
TEMP s
DBKEY i
SNO s
PNO s
QTY i
MEMSSP i
MEMPSP i

```

Figure 21. PARTS Template File: PARTS.t.

```

PARTS
TEMP b s
! Sa
! Pa
! Sp
@
$

```

Figure 22. PARTS Descriptor File: PARTS.d.

1. MOVE BUY2 TO SNO IN SA
 FIND ANY SA USING SNO IN SA
 MOVE 50 TO QTY IN SP
 FIND SP WITHIN SSP CURRENT USING QTY IN SP
 GET SNO IN SA
 @
2. MOVE PP1 TO PNO IN PA
 FIND ANY PA USING PNO IN PA
 MOVE 100 TO QTY IN SP
 FIND SP WITHIN PSP CURRENT USING QTY IN SP
 @
3. MOVE BUY3 TO SNO IN SP
 FIND ANY SP USING SNO IN SP
 FIND OWNER WITHIN SSP
 @
4. MOVE BUY3 TO SNO IN SP
 MOVE PP3 TO PNO IN SP
 MOVE 150 TO QTY IN SP
 MOVE BUY3 TO SNO IN SA
 MOVE PP3 TO PNO IN PA
 STORE SP
 @
5. MOVE BUY6 TO SNO IN SA
 MOVE JUL TO SNAME IN SA
 MOVE MADISON TO CITY IN SA
 STORE SA
 @
6. MOVE BUY1 TO SNO IN SA
 MOVE MADISON TO CITY IN SA
 FIND ANY SA USING SNO IN SA
 MODIFY CITY IN SA
 @
7. MOVE BUY1 TO SNO IN SA
 FIND ANY SA USING SNO IN SA
 GET CITY IN SA
 @
8. MOVE 100 TO QTY IN SP
 FIND ANY SP USING QTY IN SP
 GET SNO, QTY IN SP
 @
9. MOVE PP1 TO PNO IN SP
 FIND ANY SP USING PNO IN SP
 FIND OWNER WITHIN PSP
 \$

Figure 23. PARTS Query File: PARTS_QUERIES.

To determine the cause of failure a script file of the debugging output was examined. To explain the failure, it is necessary to review the format of the data stored in the SP record. The SP record consists of the following attributes as determined by the schema and template files:

DBKEY : its own unique dbkey
SNO : character attribute
PNO : character attribute
QTY : fixed attribute (integer)
MEMSSP : dbkey for the owner record in the SSP set
MEMPSP : dbkey for the owner record in the PSP set

The MEM attributes are the key attributes when conducting a query searching for the owner record of the SP record being evaluated. They tell you the unique DBKEY of the owner record of the SP record being evaluated.

Query 6 searches for the record in SP which has a SNO of Buy3. As a result the following record is found:

DBKEY : 11
SNO : Buy3
PNO : Pp3
QTY : 75
MEMSSP : 5
MEMPSP : 8

Once this record is retrieved the system determines the DBKEY for the owner record in the SSP set. The system correctly generates the following two retrieve messages:

[RETRIEVE ((TEMP = Sp) and (SNO = Buy3)) (SNO, PNO, QTY, MEMSSP, MEMPSP, DBKEY) BY DBKEY]

[RETRIEVE ((TEMP = SA) and (DBKEY = 5)) (SNO, SNAME, CITY)]

The first message finds the SP record with SNO of Buy3 and the second message uses the MEMSSP value found from that retrieve message to generate the second retrieve message which finds the SA record with a DBKEY = 5. This correctly retrieves the following results:

[SNO Buy3 SNAME Mar CITY Salinas ?]

Query 7 finds the first record in SP which has a PNO of Pp1. As a result, the following record is found:

DBKEY : 9
SNO : Buy1
PNO : Pp1
QTY : 100
MEMSSP : 1
MEMPSP : 6

Once this record is retrieved the system should find the DBKEY for the MEMPSP set. However, the system generates the following two retrieve requests:

[RETRIEVE ((TEMP = Sp) and (PNO = Pp1)) (SNO, PNO, QTY, MEMSSP, MEMPSP, DBKEY) BY DBKEY]

[RETRIEVE ((TEMP = PA) and (DBKEY = 1)) (PNO, PNAME, CITY)]

As you can observe, the first retrieve message is correct and it finds the record as shown above, however, the second retrieve message is incorrect. Instead of using the MEMPSP DBKEY value of 6, it incorrectly uses the MEMSSP DBKEY value of 1. Figure 24 summarizes where successful data access could be made in the PARTS database and where navigation was possible.

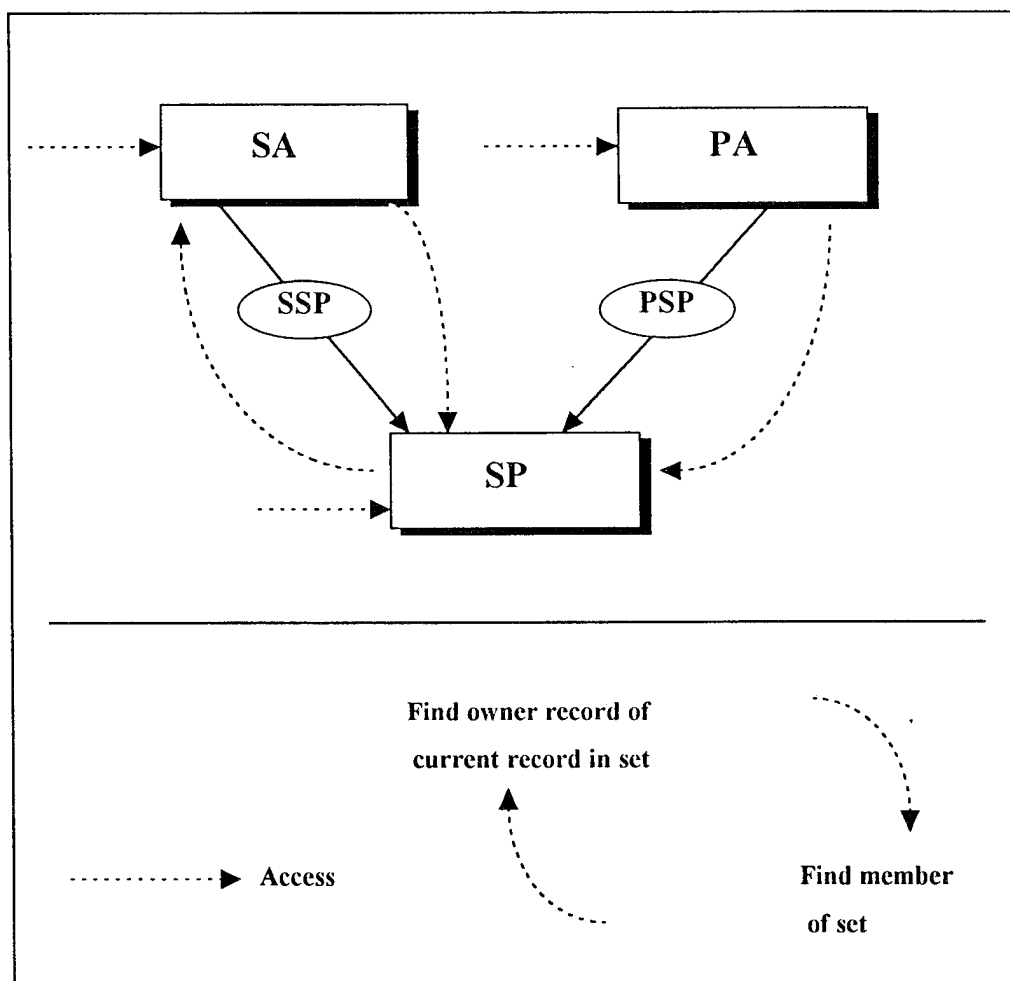


Figure 24. PARTS Query Access and Navigation.

2. The PARTS2 Database

Further testing was conducted by reversing the position of the records PA and SA as shown in Figure 25. This was accomplished by reversing their positions in the schema file illustrated in Figure 26. This also reverses the order in which the attributes MEMSSP and MEMPSP are located in the SP record. This new database was named PARTS2. The schema file, record data file, template file, descriptor file, and query file associated with the PARTS database are shown in Figures 26, 27, 28, 29, and 30 respectively. Queries equivalent to those used in the original PARTS database yielded identical results. The MEM DBKEY attribute value that is located first in the SP record is the only one used when an "owner" query is executed. This verifies that when more than one MEM attribute belongs to a record, the system always reads the first value even if it is not the correct attribute value (DBKEY) for the MEM set that the query is searching for. Figure 31 illustrates where successful data access could be made in the PARTS2 database and where navigation was possible. This is consistent with Figure 24.

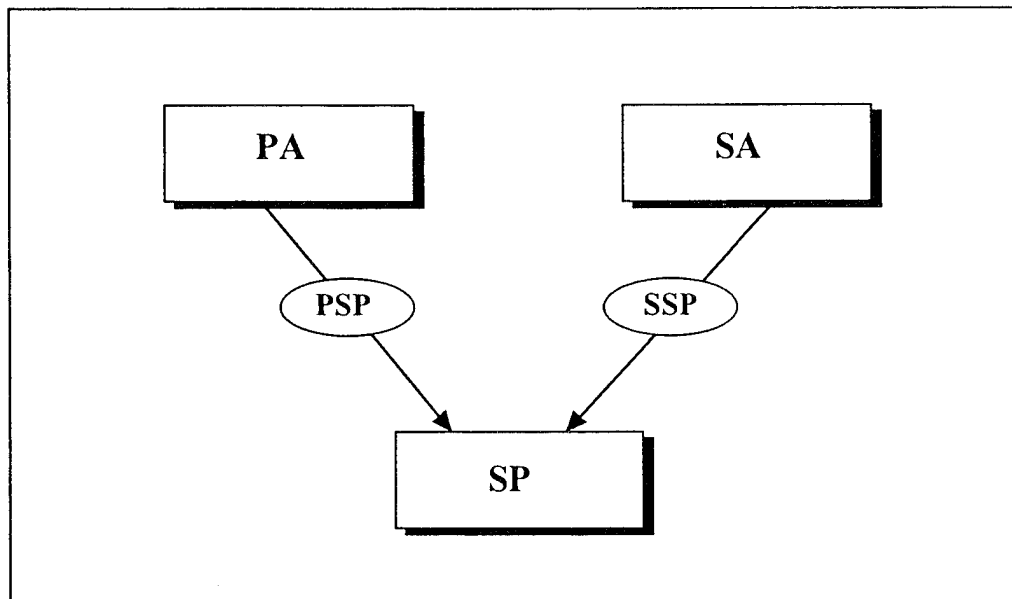


Figure 25. PARTS2 Database.

```

SCHEMA NAME IS PARTS2;
RECORD NAME IS PA;
  DUPLICATES ARE NOT ALLOWED FOR PNO;
  PNO ; CHARACTER 10.
  PNAME ; CHARACTER 10.
  CITY ; CHARACTER 10.
RECORD NAME IS SA;
  DUPLICATES ARE NOT ALLOWED FOR SNO;
  SNO ; CHARACTER 10.
  SNAME ; CHARACTER 10.
  CITY ; CHARACTER 10.
RECORD NAME IS SP;
  SNO ; CHARACTER 10.
  PNO ; CHARACTER 10.
  QTY ; FIXED 4.
SET NAME IS PSP;
  OWNER IS PA;
  MEMBER IS SP;
  INSERTION IS AUTOMATIC
  RETENTION IS FIXED;
  SET SELECTION IS BY VALUE OF PNO IN PA;
SET NAME IS SSP;
  OWNER IS SA;
  MEMBER IS SP;
  INSERTION IS AUTOMATIC
  RETENTION IS FIXED;
  SET SELECTION IS BY VALUE OF SNO IN SA;
$

```

Figure 26. PARTS2 Schema File: PARTS2dml.db.

```

PARTS2
@
Pa
1 Pp1 Mmu Monterey
2 Pp2 Databus Sanjose
3 Pp3 Harddrive Salinas
@
Sa
4 Buy1 Dec Monterey
5 Buy2 Jan Sanjose
6 BUY3 Oct Sanjose
7 Buy4 MAR Sanjose
8 Buy5 Mar Salinas
@
Sp
9 Buy1 Pp1 100 1 4
10 Buy2 Pp2 50 2 5
11 Buy3 Pp3 75 3 6
12 Buy3 Pp1 50 1 6
13 Buy2 Pp1 100 1 5
$

```

Figure 27. PARTS2 Record Data File: PARTS2.r.

```

PARTS2
3
5
Pa
TEMP s
DBKEY i
PNO s
PNAME s
CITY s
5
Sa
TEMP s
DBKEY i
SNO s
SNAME s
CITY s
7
Sp
TEMP s
DBKEY i
SNO s
PNO s
QTY i
MEMPSP i
MEMSSP i

```

Figure 28. PARTS2 Template File: PARTS2.t.

```

PARTS2
TEMP b s
! Pa
! Sa
! Sp
@
$

```

Figure 29. PARTS2 Descriptor File: PARTS2.d.

```

1. MOVE BUY2 TO SNO IN SA
   FIND ANY SA USING SNO IN SA
   MOVE 50 TO QTY IN SP
   FIND SP WITHIN SSP CURRENT USING QTY IN SP
   GET SNO IN SA
   @
2. MOVE PP1 TO PNO IN PA
   FIND ANY PA USING PNO IN PA
   MOVE 100 TO QTY IN SP
   FIND SP WITHIN PSP CURRENT USING QTY IN SP
   @
3. MOVE BUY3 TO SNO IN SP
   FIND ANY SP USING SNO IN SP
   FIND OWNER WITHIN SSP
   @
4. MOVE BUY3 TO SNO IN SP
   MOVE PP3 TO PNO IN SP
   MOVE 150 TO QTY IN SP
   MOVE BUY3 TO SNO IN SA
   MOVE PP3 TO PNO IN PA
   STORE SP
   @
5. MOVE BUY6 TO SNO IN SA
   MOVE JUL TO SNAME IN SA
   MOVE MADISON TO CITY IN SA
   STORE SA
   @
6. MOVE BUY1 TO SNO IN SA
   MOVE MADISON TO CITY IN SA
   FIND ANY SA USING SNO IN SA
   MODIFY CITY IN SA
   @
7. MOVE BUY1 TO SNO IN SA
   FIND ANY SA USING SNO IN SA
   GET CITY IN SA
   @
8. MOVE 100 TO QTY IN SP
   FIND ANY SP USING QTY IN SP
   GET SNO, QTY IN SP
   @
9. MOVE PP1 TO PNO IN SP
   FIND ANY SP USING PNO IN SP
   FIND OWNER WITHIN PSP
   $

```

Figure 30. PARTS2 Query File: PARTS2_QUERIES.

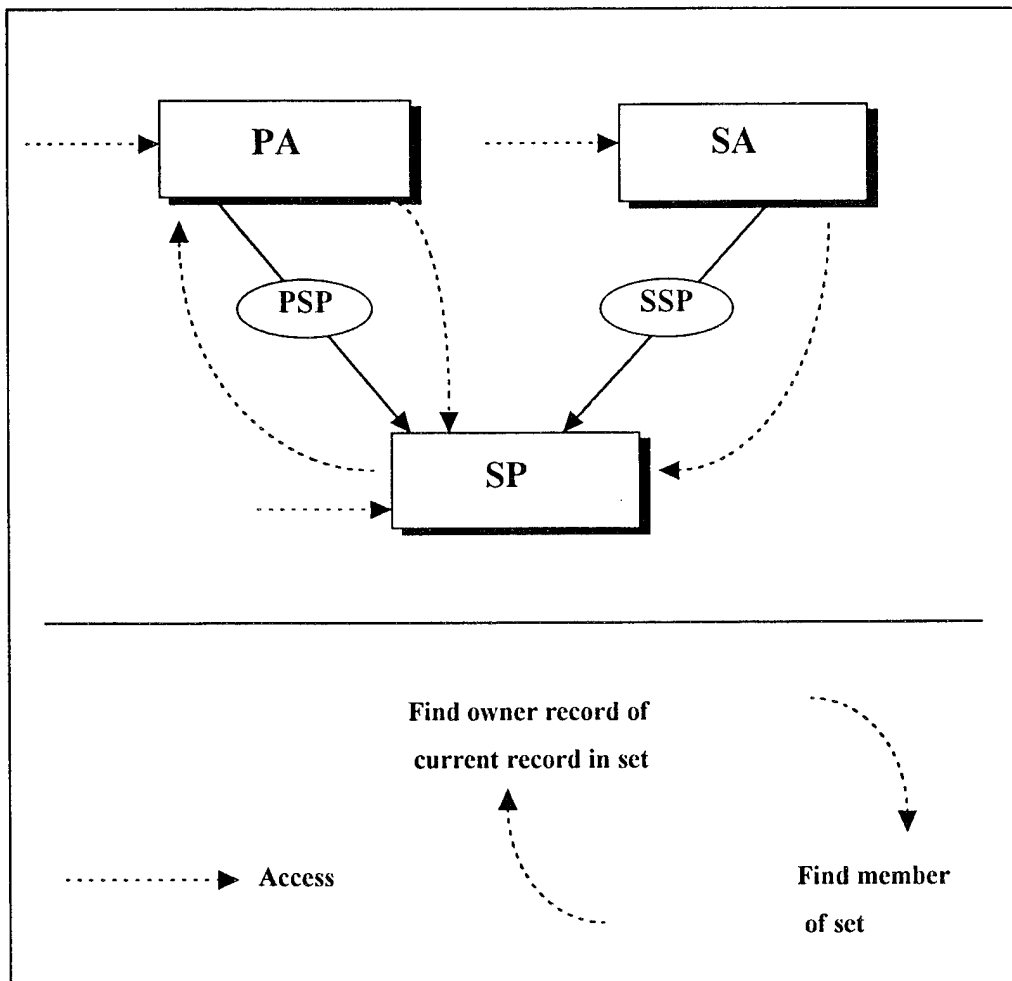


Figure 31. PARTS2 Query Access and Navigation.

3. The PARTS3 Database

A reduced version of the PARTS database named PARTS3 (see Figure 32) which removed the second owner record was tested to verify correct operation with only one set and one owner record type. The schema file, record data file, template file, descriptor file, and query file associated with the PARTS database are shown in Figures 33, 34, 35, 36, and 37 respectively. All The queries in Figure 37 executed successfully using the data from the PARTS3.r record file. Queries 1 and 2 demonstrate direct record access, query 3 shows downward navigation by finding a record belonging to the PSP set, and query 4 demonstrates upward navigation by finding the owner of a SP record. Figure 38 shows the navigation and access paths for the PARTS3 database.

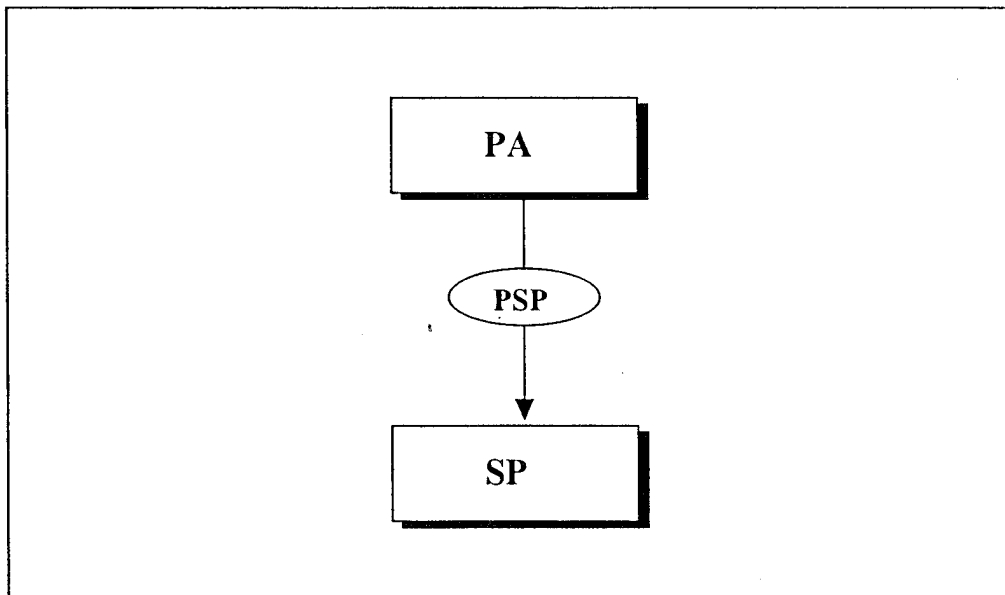


Figure 32. PARTS3 Database.

```

SCHEMA NAME IS PARTS3;
RECORD NAME IS PA;
  DUPLICATES ARE NOT ALLOWED FOR PNO;
    PNO ; CHARACTER 10.
    PNAME ; CHARACTER 10.
    CITY ; CHARACTER 10.
RECORD NAME IS SP;
  SNO ; CHARACTER 10.
  PNO ; CHARACTER 10.
  QTY ; FIXED 4.
SET NAME IS PSP;
  OWNER IS PA;
  MEMBER IS SP;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF PNO IN PA;
$

```

Figure 33. PARTS3 Schema File: PARTS3dml.db.

```

PARTS3
@
Pa
1 Pp1 Mmu Monterey
2 Pp2 Databus Sanjose
3 Pp3 Harddrive Salinas
@
Sp
4 Buy1 Pp1 100 1
5 Buy2 Pp2 50 2
6 Buy3 Pp3 75 3
7 Buy3 Pp1 50 1
8 Buy2 Pp1 100 1
$

```

Figure 34. PARTS3 Record Data File: PARTS3.r.

```

PARTS3
2
5
Pa
TEMP s
DBKEY i
PNO s
PNAME s
CITY s
6
Sp
TEMP s
DBKEY i
SNO s
PNO s
QTY i
MEMPSP i

```

Figure 35. PARTS3 Template File: PARTS3.t.

```

PARTS3
TEMP b s
! Pa
! Sp
@
$

```

Figure 36. PARTS3 Descriptor File: PARTS3.d.

```

1. MOVE PP1 TO PNO IN PA
   FIND ANY PA USING PNO IN PA
   MOVE 100 TO QTY IN SP
   FIND SP WITHIN PSP CURRENT USING QTY IN SP
   @
2. MOVE BUY2 TO SNO IN SP
   FIND ANY SP USING SNO IN SP
   FIND OWNER WITHIN PSP
   $

```

Figure 37. PARTS3 Query File: PARTS3_QUERIES.

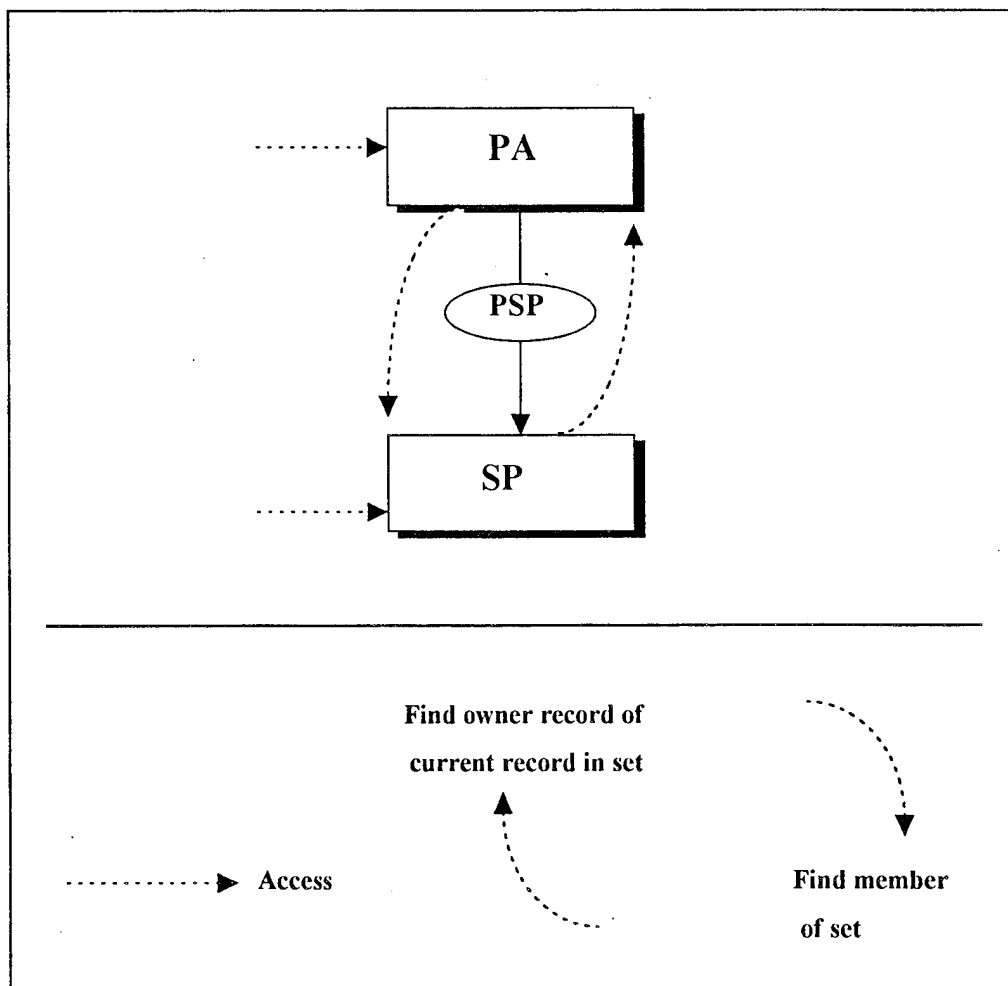


Figure 38. PARTS3 Query Access and Navigation.

4. The PARTS4 Database

The PARTS4 database added a third level to the hierarchy (network) of the PARTS database as shown in Figure 39. Testing this configuration assisted in determining: What level or depth could be accessed by a query starting at the top level; if an interior record can act as both a record belonging to a set and as an owner of its own set; and if a bottom level record can navigate upward to find its owner(s). The schema file, record data file, template file, descriptor file, and query file associated with the PARTS4 database are shown in Figures 40, 41, 42, 43, and 44 respectively.

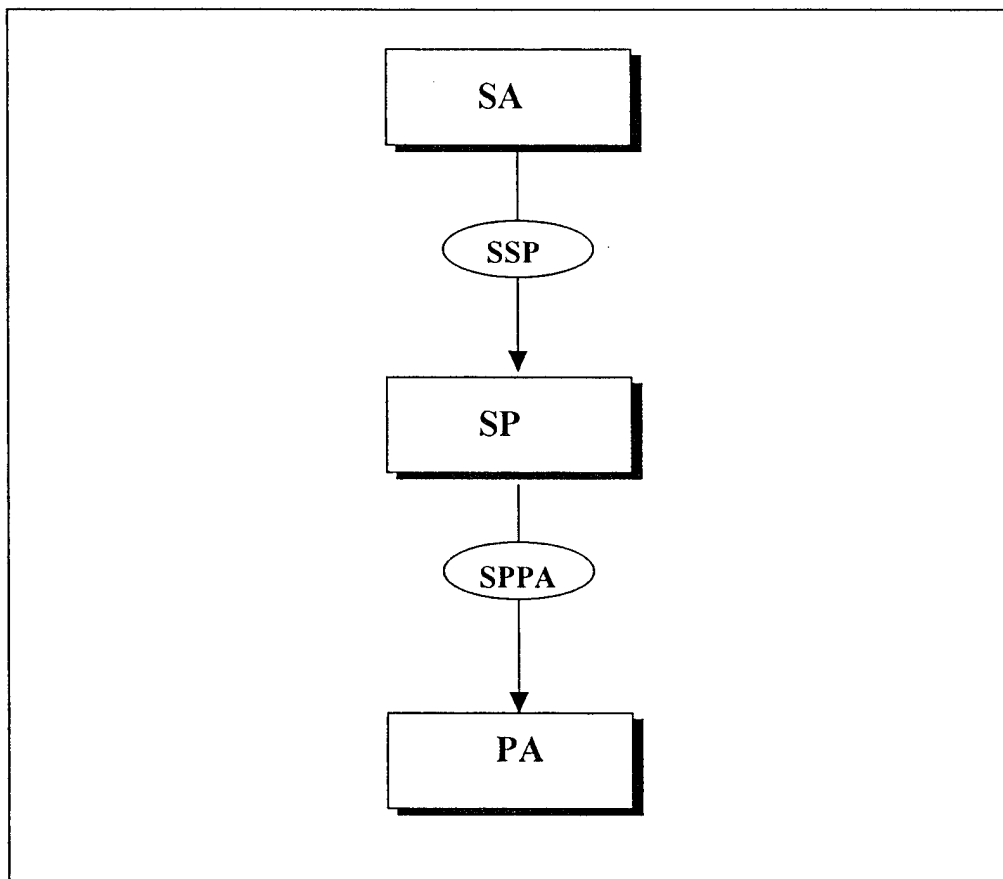


Figure 39. PARTS4 Database.

```

SCHEMA NAME IS PARTS4;
RECORD NAME IS SA;
  DUPLICATES ARE NOT ALLOWED FOR SNO;
    SNO ; CHARACTER 10.
    SNAME ; CHARACTER 10.
    CITY ; CHARACTER 10.
RECORD NAME IS PA;
  DUPLICATES ARE NOT ALLOWED FOR PNO;
    PNO ; CHARACTER 10.
    PNAME ; CHARACTER 10.
    CITY ; CHARACTER 10.
RECORD NAME IS SP;
  DUPLICATES ARE NOT ALLOWED FOR SPNO;
    SNO ; CHARACTER 10.
    SPNO ; CHARACTER 10.
    QTY ; FIXED 4.
SET NAME IS SSP;
  OWNER IS SA;
  MEMBER IS SP;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF SNO IN SA;
SET NAME IS SPPA;
  OWNER IS SP;
  MEMBER IS PA;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF SPNO IN
SP;
$

```

Figure 40. PARTS4 Schema File: PARTS4dmldb.

```

PARTS4
@
Sa
1 Buy1 Dec Monterey
2 Buy2 Jan Sanjose
3 Buy3 Oct Sanjose
4 Buy4 Mar Sanjose
5 Buy5 Mar Salinas
@
Pa
6 Pp1 Mmu Monterey 9
7 Pp2 Databus Sanjose 10
8 Pp3 Harddrive Salinas 11
@
Sp
9 Buy1 Sp1 100 1
10 Buy2 Sp2 50 2
11 Buy3 Sp3 75 3
$

```

Figure 41. PARTS4 Record Data File: PARTS4.r.

```

PARTS4
3
5
Sa
TEMP s
DBKEY i
SNO s
SNAME s
CITY s
6
Pa
TEMP s
DBKEY i
PNO s
PNAME s
CITY s
MEMSPPA i
6
Sp
TEMP s
DBKEY i
SNO s
SPNO s
QTY i
MEMSSP i

```

Figure 42. PARTS4 Template File: PARTS4.t.

```

PARTS4
TEMP b s
! Sa
! Pa
! Sp
@
$

```

Figure 43. PARTS4 Descriptor File: PARTS4.d.

```

1. MOVE PP1 TO PNO IN PA
   FIND ANY PA USING PNO IN PA
   GET PNO IN PA
   @
2. MOVE SP1 TO SPNO IN SP
   FIND ANY SP USING SPNO IN SP
   GET QTY IN SP
   @
3. MOVE BUY3 TO SNO IN SA
   FIND ANY SA USING SNO IN SA
   GET SA
   @
4. MOVE BUY1 TO SNO IN SA
   FIND ANY SA USING SNO IN SA
   MOVE 100 TO QTY IN SP
   FIND SP WITHIN SSP CURRENT USING QTY IN SP
   GET SPNO IN SP
   @
5. MOVE SP3 TO SPNO IN SP
   FIND ANY SP USING SPNO IN SP
   FIND OWNER WITHIN SSP
   @
6. MOVE SP3 TO SPNO IN SP
   FIND ANY SP USING SPNO IN SP
   MOVE HARDDRIVE TO PNAME IN PA
   FIND PA WITHIN SPPA CURRENT USING PNAME IN PA
   GET PA
   @
7. MOVE PP1 TO PNO IN PA
   FIND ANY PA USING PNO IN PA
   FIND OWNER WITHIN SPPA

$

```

Figure 44. PARTS4 Query File: PARTS4_QUERIES.

Queries 1 through 3 executed successfully and demonstrated that records could be directly accessed on every level of the hierarchy. Query 4 executed successfully proving that navigation was possible from the top level record to a member record, one level below. Query 5 also executed successfully showing that navigation was possible from a mid-level member record to an upper owner record. The goal of query 6 was to navigate from a mid-level owner record to a bottom level member record. This query

failed and further investigation demonstrates why. Query 6 initially accessed SP to find any record SP with a SPNO of Sp3. This query should return the following SP record:

DBKEY : 11
SNO : Buy1
SPNO : Sp3
QTY : 75
MEMSSP : 3

Following the retrieval of this record reveals that a system generated retrieve request should be constructed to find a record PA that has a PNAME of Harddrive and has a MEMSPPA value of 11 which corresponds to the DBKEY value of the SP owner record. The following describes what actually occurs. Request 1 is formed to retrieve the proper SP record.

[RETRIEVE ((TEMP = Sp) and (SPNO = Sp3)) (SNO, SPNO, QTY, MEMSSP, DBKEY) BY DBKEY]

The above retrieve statement is correct and it returns the following record:

[SNO Buy3 SPNO Sp3 QTY 75 MEMSSP 3 DBKEY 11]

This is the correct SP record. Now a retrieve request should be made to retrieve the PA record with a MEMSPPA of 11. However, the following retrieve statement is generated:

[RETRIEVE ((TEMP = Pa) and (MEMSPPA = 3) and (PNAME = Harddrive)) (PNO, PNAME, CITY, DBKEY) BY DBKEY]

Note that the MEMSPPA value requested is 3 rather than 11. Instead of using the DBKEY of the owning SP record to find a PA record belonging to the set SPPA, it acquires the MEMSSP value instead and attempts to link the PA record to the DBKEY of the SA record to which the owning SP record belongs. The query failed to return any record because no PA record existed that had a MEMSPPA of 3. This query illustrates that a mid-level owner record is not able to navigate downward to access a member record belonging to the set which it owns.

Query 7 also failed. This query was used to determine if a record more than two levels deep in the schema could find its set owner record existing at the mid-level of the schema. The query causes the system to generate two retrieve statements. The first retrieve is as follows:

```
[RETRIEVE ((TEMP = Pa) and (PNO = Pp1)) (PNO, PNAME, CITY, MEMSPPA, DBKEY)]
```

This retrieve statement is correct and it returns the following PA record:

```
[PNO Pp1 PNAME Mmu CITY Monterey MEMSPPA 9 DBKEY 6]
```

Then the system generated the following retrieve in response to the find owner request of query 7:

```
[RETRIEVE ((TEMP = SP) and (DBKEY = 0)) (SNO, SPNO, QTY)]
```

Note that the actual owning record of the PA record returned from the first retrieve shown above has a DBKEY of 9 (the MEMSPPA value). The system is unable to locate the correct DBKEY and arbitrarily assigns DBKEY = 0. This causes the query to fail because it's looking for a SP record with a DBKEY equal to zero.

The access and navigation paths for the PARTS4 database are annotated in Figure 45. For a three level database, direct access to records at any level is possible. However, navigation is only possible from the top level to the mid-level (i.e., SA owner for SP member) or from the mid-level to the top level (i.e., SP member to SA owner).

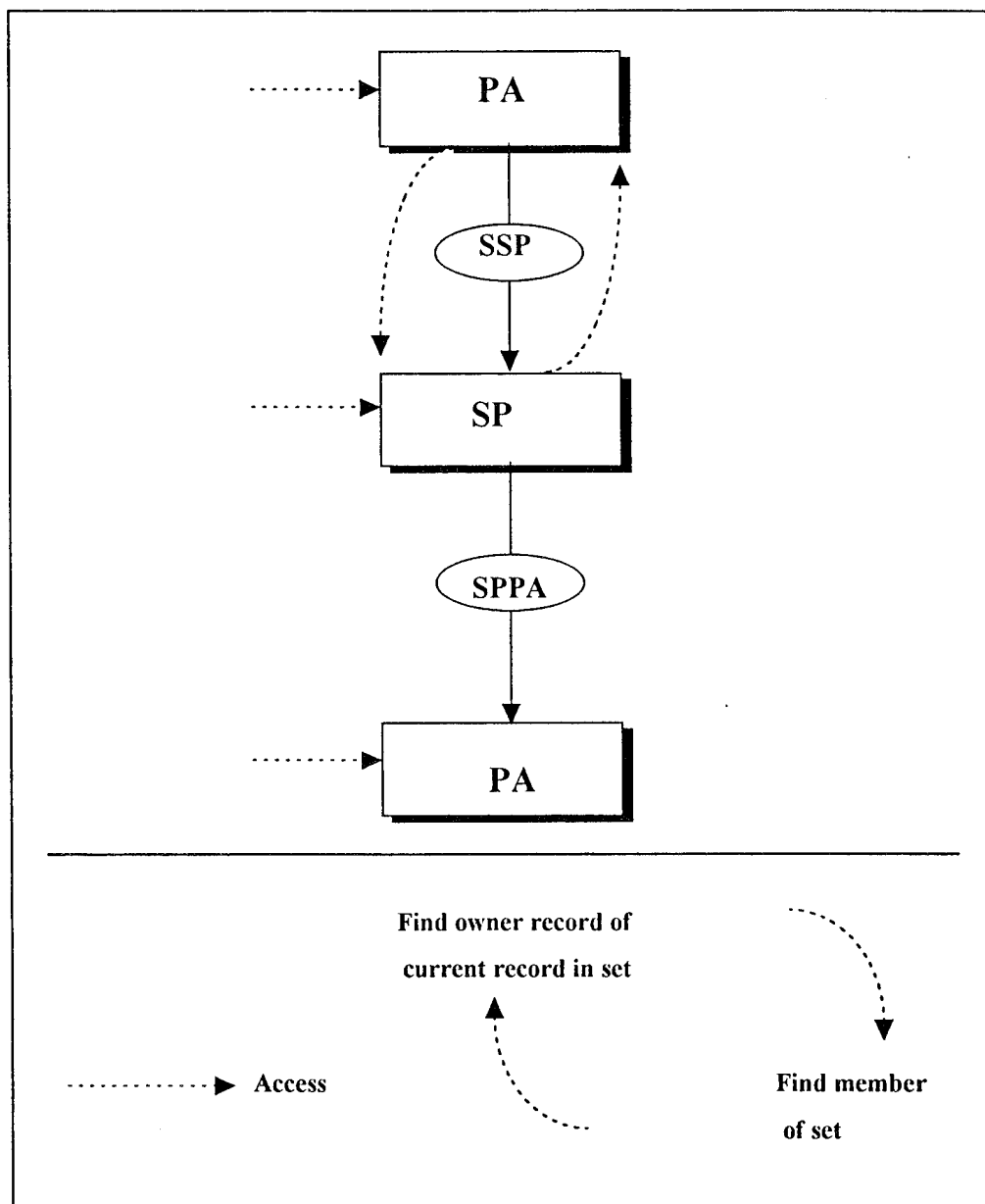


Figure 45. PARTS4 Query Access and Navigation.

5. The PARTS5 Database

The PARTS5 database (see Figure 46) utilized the same schema as the PARTS4 database with the addition of a new top level owner record which has a set relationship with the bottom level record in the PARTS4 database (see Figure 39). This schema design was tested to determine if adding an extra top level record would change the access and navigation characteristics of the PARTS4 schema. The schema file, record data file, template file, descriptor file, and query file associated with the PARTS5 database are shown in Figures 47, 48, 49, 50, and 51 respectively.

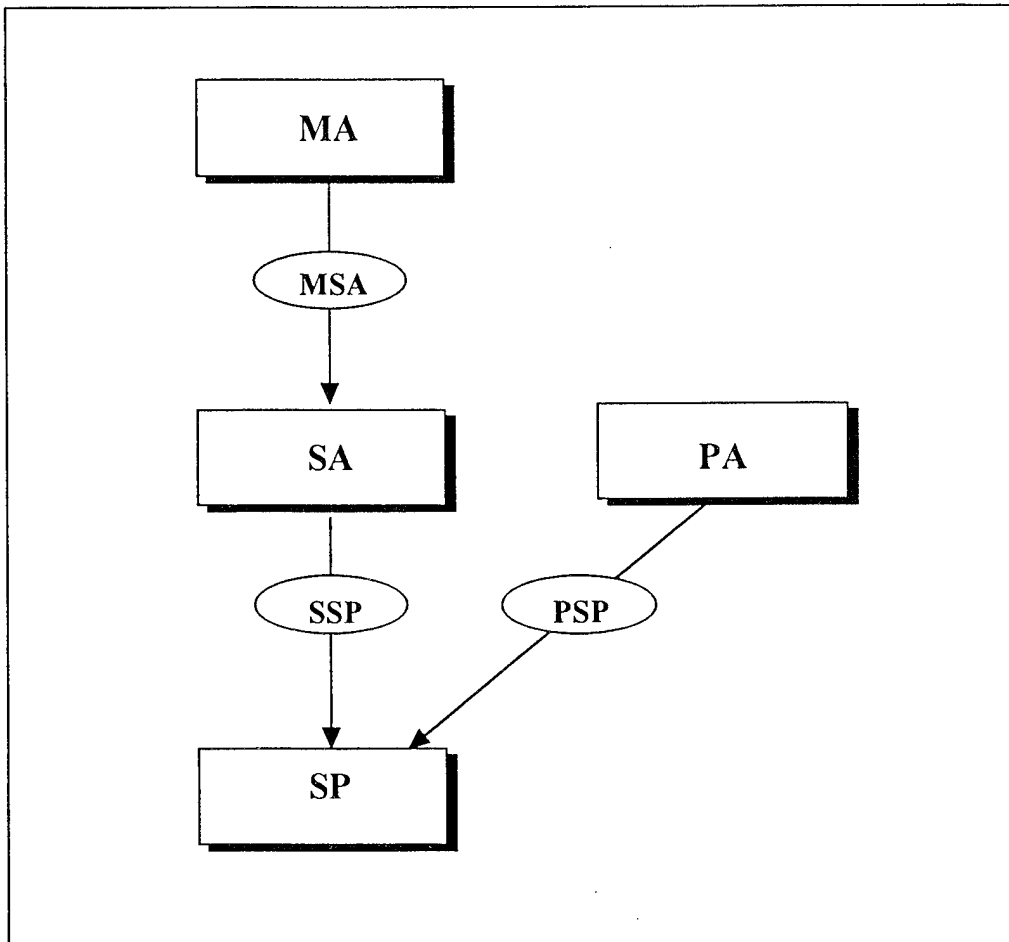


Figure 46. PARTS5 Database.

```

SCHEMA NAME IS PARTS5;
RECORD NAME IS MA;
    DUPLICATES ARE NOT ALLOWED FOR MNO;
    MNO ; CHARACTER 10.
    MNAME ; CHARACTER 10.
RECORD NAME IS SA;
    DUPLICATES ARE NOT ALLOWED FOR SNO;
    SNO ; CHARACTER 10.
    SNAME ; CHARACTER 10.
    CITY ; CHARACTER 10.
RECORD NAME IS PA;
    DUPLICATES ARE NOT ALLOWED FOR PNO;
    PNO ; CHARACTER 10.
    PNAME ; CHARACTER 10.
    CITY ; CHARACTER 10.
RECORD NAME IS SP;
    SNO ; CHARACTER 10.
    PNO ; CHARACTER 10.
    QTY ; FIXED 4.
SET NAME IS MSA;
    OWNER IS MA;
    MEMBER IS SA;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        SET SELECTION IS BY VALUE OF MNO IN MA;
SET NAME IS SSP;
    OWNER IS SA;
    MEMBER IS SP;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        SET SELECTION IS BY VALUE OF SNO IN SA;
SET NAME IS PSP;
    OWNER IS PA;
    MEMBER IS SP;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        SET SELECTION IS BY VALUE OF PNO IN PA;
$

```

Figure 47. PARTS5 Schema File: PARTS5dml.db.

```
PARTS5
@
Ma
14 Ma1 Packers
15 Ma2 Vikings
16 Ma3 Bears
@
Sa
1 Buy1 Dec Monterey 14
2 Buy5 Jan Sanjose 15
3 Buy2 Oct Sanjose 16
4 Buy4 MAR Sanjose 14
5 Buy3 Mar Salinas 15
@
Pa
6 Pp1 Mmu Monterey
7 Pp2 Databus Sanjose
8 Pp3 Harddrive Salinas
@
Sp
9 Buy1 Pp1 100 1 6
10 Buy2 Pp2 50 3 7
11 Buy3 Pp3 75 5 8
12 Buy2 Pp1 50 3 6
13 Buy2 Pp1 100 3 6
$
```

Figure 48. PARTS5 Record Data File: PARTS5.r.

```

PARTS5
4
4
Ma
TEMP s
DBKEY i
MNO s
MNAME s
6
Sa
TEMP s
DBKEY i
SNO s
SNAME s
CITY s
MEMMSA i
5
Pa
TEMP s
DBKEY i
PNO s
PNAME s
CITY s
7
Sp
TEMP s
DBKEY i
SNO s
PNO s
QTY i
MEMSSP i
MEMPSP i

```

Figure 49. PARTS5 Template File: PARTS5.t.

```

PARTS5
TEMP b s
! Ma
! Sa
! Pa
! Sp
@
$

```

Figure 50. PARTS5 Descriptor File: PARTS5.d.

```

1.  MOVE MA1 TO MNO IN MA
    FIND ANY MA USING MNO IN MA
    GET MA
    @
2.  MOVE BUY4 TO SNO IN SAA
    FIND ANY SA USING SNO IN SA
    GET SNO IN SA
    @
3.  MOVE PP2 TO PNO IN PA
    FIND ANY PA USING PNO IN PA
    GET PNO IN PA
    @
4.  MOVE PP1 TO PNO IN SP
    MOVE 50 TO QTY IN SP
    FIND ANY SP USING PNO, QTY IN SP
    GET SNO, PNO, QTY IN SP
    @
5.  MOVE MA2 TO MNO IN MA
    FIND ANY MA USING MNO IN MA
    MOVE JAN TO SNAME IN SA
    FIND SA WITHIN MSA CURRENT USING SNAME IN SA
    GET SA
    @
6.  MOVE BUY2 TO SNO IN SA
    FIND ANY SA USING SNO IN SA
    MOVE PP2 TO PNO IN SP
    FIND SP WITHIN SSP CURRENT USING PNO IN SP
    GET SNO, PNO IN SP
    @
7.  MOVE BUY1 TO SNO IN SP
    FIND ANY SP USING SNO IN SP
    FIND OWNER WITHIN SSP
    GET SNO IN SA
    @
8.  MOVE BUY3 TO SNO IN SA
    FIND ANY SA USING SNO IN SA
    FIND OWNER WITHIN MSA
    GET MNO IN MA
    @
9.  MOVE PP1 TO PNO IN PA
    FIND ANY PA USING PNO IN PA
    MOVE 100 TO QTY IN SP
    FIND SP WITHIN PSP CURRENT USING QTY IN SP
    GET PNO, QTY IN SP
    @
10. MOVE BUY1 TO SNO IN SP
    FIND ANY SP USING SNO IN SP
    FIND OWNER WITHIN PSP
    GET PNO IN PA
    $

```

Figure 51. PARTS5 Query File: PARTS5_QUERIES.

Once again, we verified that each record could be accessed directly. Queries 1 through 4 accomplished this and executed correctly. Query 5 tested the ability to navigate from the top level record, MA, down one level to find a set member record in SA. This executed properly and thus remains consistent with the results found from query 4 for the PARTS4 database.

Query 6 was identical to query 6 in the PARTS4 database. The goal was to access a mid-level owner record, SA, and navigate downward to access a member record in SP belonging to the set SSP. As expected, the query failed for the same reason. The first "find" generates the following retrieve statement:

```
[RETRIEVE ((TEMP = Sa) and (SNO = Buy2)) (SNO, SNAME, CITY, MEMMSA, DBKEY) BY DBKEY]
```

This request returns the following SA record data:

```
[SNO Buy2 SNAME Oct CITY Sanjose MEMMSA 16 DBKEY 3]
```

The next retrieve statement is generated from the find SP within SSP of the current line of the query illustrated below:

```
[RETRIEVE ((TEMP = SP) and (MEMSSP = 16) and (PNO = PP2)) (SNO, PNO, QTY, DBKEY) BY DBKEY]
```

Note that once again instead of assigning MEMSSP equal to the DBKEY of the owner record (in this case 3), it takes the value of the set it belongs to (MEMMSA) which is 16. Figure 33 shows that this record does not exist.

Query 7 was identical to query 7 in the PARTS4 database. This query was used to determine if a record more than two levels deep in the schema could find its set owner

record existing at the mid-level of the schema. This query also failed for the same reason as it did in the PARTS4 database. The first FIND from line 2 of query 7 generates the following retrieve statement for a SP record:

```
[RETRIEVE ((TEMP = Sp) and (SNO = Buy1)) (SNO, PNO, QTY, MEMSSP,
MEMPSP, DBKEY)]
```

This retrieve request returns the following record:

```
[SNO Buy1 PNO Pp1 QTY 100 MEMSSP 1 MEMPSP 6 DBKEY 9 ?]
```

The FIND OWNER request from line 3 generates the following retrieve request:

```
[RETRIEVE ((TEMP = SA) and (DBKEY = 0)) (SNO, SNAME, CITY)]
```

Note that once again instead of using the MEMSSP value of "6" which corresponds to the owning record in SA it assigns DBKEY = 0. From Figure 33 you can see that no record with a DBKEY = 0 exists.

Query 8 is identical to query 5 from the PARTS4 database. Query 8 ran successfully, again showing that navigation from a mid-level member record upward to its top level owner record is possible.

Query 9 is similar to query 5 from the PARTS database. This again showed the ability to navigate from a top level record (PA) to a member record (SP). The only difference was that the member record also existed as a third level record (see Figure 32, the schema). This query ran correctly so the level of the member record makes no difference as long as the owning record is itself a top level record.

Query 10 attempted to navigate from a bottom level record (SP) to the second owner record (PA) and is identical to query 7 in the PARTS database. This query also

failed but for a different (although similar) reason. The FIND from line 2 of query 7 (see Figure 35) generates the following retrieve statement:

```
[RETRIEVE ((TEMP = Sp) and (SNO = Buy1)) (SNO, PNO, QTY, MEMSSP,
MEMPSP, DBKEY) BY DBKEY]
```

This returns the following SP record:

```
[SNO Buy1 PNO Pp1 QTY 100 MEMSSP 1 MEMPSP 6 DBKEY 9 ?]
```

The FIND OWNER request from line 3 of query 7 generates the following retrieve statement:

```
[RETRIEVE ((TEMP = PA) and (DBKEY = 0)) (PNO, PNAME, CITY)]
```

This time the system cannot identify the proper DBKEY and uses a default value of zero. Once again, (see Figure 33) you can see that a record with a DBKEY = 0 does not exist as a PA record.

The access and navigation paths are annotated in Figure 52 for the PARTS5 database. For a three level database, direct access to records at any level is possible. Navigation is only possible from the top level to the mid-level (i.e., MA owner to SA member), from the top level to the bottom level record (PA owner to SP member) or mid-level to top level (SA member to MA owner).

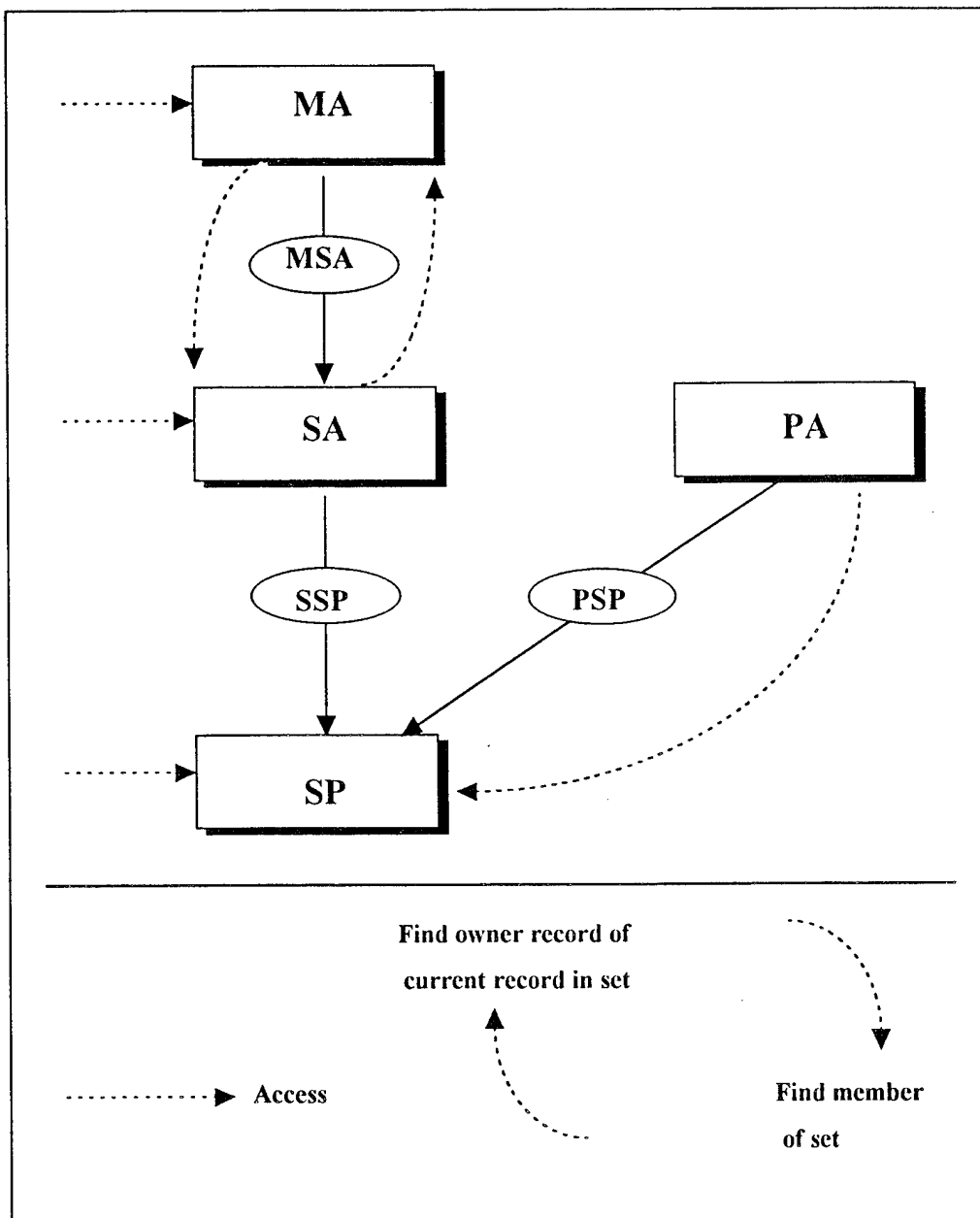


Figure 52. PARTS5 Query Access and Navigation.

6. Summary

As evident from the testing conducted, any record can be directly accessed. When navigation is required for finding ownership or finding member records, you must only navigate one level down from a top level owner record. To navigate upward, you must start at a second level record and only access the first owner record that the member record belongs to. If the record belongs to more than one owner, the first owner is the only one that may be accessed.

These limitations restrict the usefulness of the system tremendously. Therefore, the theoretical EWIR database schema developed for this thesis is not "implementable." With the exception of a very basic schema (i.e., the PARTS database), the network interface is very limited. Further evaluation of the program code to investigate and correct these limitations is not an objective of this thesis but is recommended for future research.

IV. IMPLEMENTATION OF THE EWIR DATABASE

This chapter addresses the second objective of this thesis: To design and implement a network data model for a representative portion of the EWIR database. It is important to note at this time, our global strategy for the design and implementation of the network EWIR database. The strategy involved three steps:

1. Develop the conceptual design of the network EWIR model.
2. Test the network interface to determine the system's capabilities and limitations (i.e., Chapter III, section D of this thesis).
3. Based on the findings from step 2, either implement the design from step 1 or modify the design so that it can be implemented on the M²DBMS; therefore, actually demonstrating the current system's capabilities and limitations.

In actuality, since the research work for this thesis was conducted by a two member research team, it was favorable to execute steps 1 and 2 concurrently thus saving time. This strategy provided for an efficient, systematic research process resulting in conclusive evidence that describes the capabilities and limitations of the network interface.

Section A of this chapter provides the EWIR database specification, to include the format, structure and limitations of the "old" EWIR database, and the design of the "new" network EWIR database. Using the network EWIR database specification of Section A, Section B discusses the translation of this specification into a database schema using the network Data Definition Language (DDL). The schema is illustrated in this section as well as the resulting DDL compiler files, the template and descriptor files.

Finally, Section C of this chapter addresses the loading of the EWIR database record data into the network interface.

Once the EWIR specification is translated into the network schema, and the network interface processes the schema (and associated template and descriptor files), the network interface is ready to accept the record data. After the record data has been loaded successfully, the process of creating a “new” network EWIR database is complete. Thus the database is ready to accept the commands for manipulating the database and executing transactions.

A. EWIR DATABASE SPECIFICATION

The specification described herein is for a small sampling of the EWIRDB; specifically for the antenna data of a given emitter. As stated earlier in the overview of the EWIRDB, reference [3] provides a detailed study of the design and analysis of the EWIRDB for the object-oriented implementation. Therefore, to gain more of a global perspective of the EWIRDB, it may be first advantageous to review reference [3] prior to reviewing this section and chapter. This section is written assuming the reader already has a basic understanding of the EWIRDB’s structure and format. The object-oriented conceptual design as described in [3] will also serve as the basic conceptual design for our network model implementation. Furthermore, this thesis does not advocate which data model is “optimal” for implementing the EWIRDB, but rather it proposes how one might implement the EWIRDB in a network model representation.

1. The “Old” EWIR Data Model

As mentioned previously (Chapter II, section C), the existing format of the EWIRDB is relatively complex. Data is represented in a hierarchical tree where various reference codes and alike are used to link related data items together throughout the hierarchy. This type of data model is difficult to understand and to interpret the

semantics of the data, thus hindering the users ability to gain a meaningful view of the Electronic Warfare (EW) data. Essentially, the complex format of the database obscures the meaning and semantics of the EW data.

The EWIRDB is a noncommunications electronic emitter database. It is comprised of data on individual emitters. An emitter is described by parametric information as well as administrative, commentary and reference data. Each emitter has different types of information, stored as different types of data records. The parametric tree is the structure used to maintain the detailed parametric characteristics of an emitter.

There are essentially three hierarchical structures or trees that are used to maintain the EWIR data. First, the P/CW (Pulsed/Continuous Wave) tree is used for evaluating and identifying electromagnetic energy radiated by emitters; second, the RPA (Receiver Parametric Performance) tree contains receiver design and performance information on the receiver portion of emitter systems; and finally the ECM (Electronic Countermeasures) tree describes jamming systems and are referenced in the development of ECCM (Electronic Counter-Countermeasures) to overcome the jammer threat. The P/CW tree coupled with the RPA tree provide a comprehensive report on an emitter's performance.

The EWIRDB uses the combined KILTING/EWIRDB parametric tree as a formatted structure to describe and catalog the parameters associated with noncommunications emitters. The tree is a management tool that orders a long list logically and hierarchically in a way that proceeds from broad characteristics through levels of successively finer ones. Figure 53 illustrates a sample layout of a parametric tree. The numbering system shows how the tree forms branches at each level. Each parametric tree is subordinate to only one other level. [5]

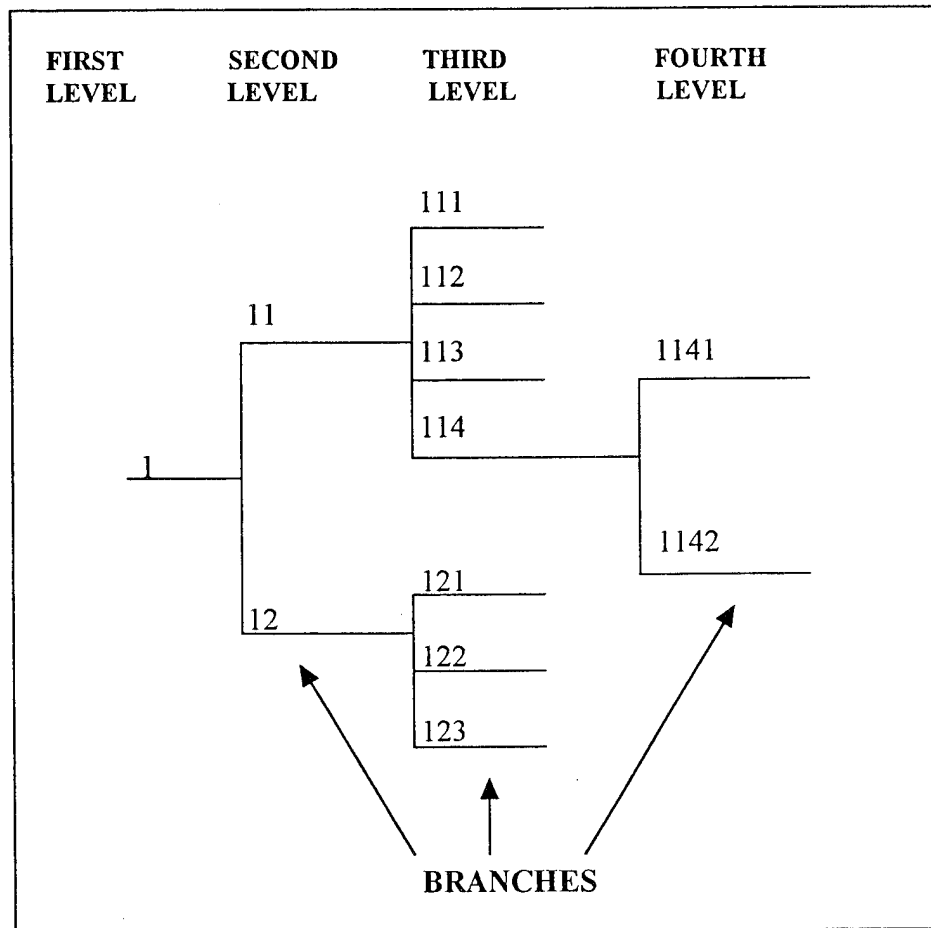


Figure 53. Parametric Tree Structure [5].

The P/CW tree is illustrated in Figure 54. For illustration purposes, we will focus on the second and third level of this tree (i.e., the antenna related data). Note that each branch contains a heading or label. For example, in the second level, the heading “ANTENNA” of the **12 B ANTENNA** branch is a branch name. Furthermore, the branch number (i.e., 12) provides a means to navigate through the tree. If we start with the **12 B ANTENNA** branch and follow on to one of the next branches in the next level, we observe one of two numbers, 121 or 122. If we continue with 121 through to the next level and branch, the next numbers are 1211, 1212, and 1213. This numbering system is used throughout the depth of the parametric tree and provides a means of identifying a unique path from the root of the tree to the desired branch.

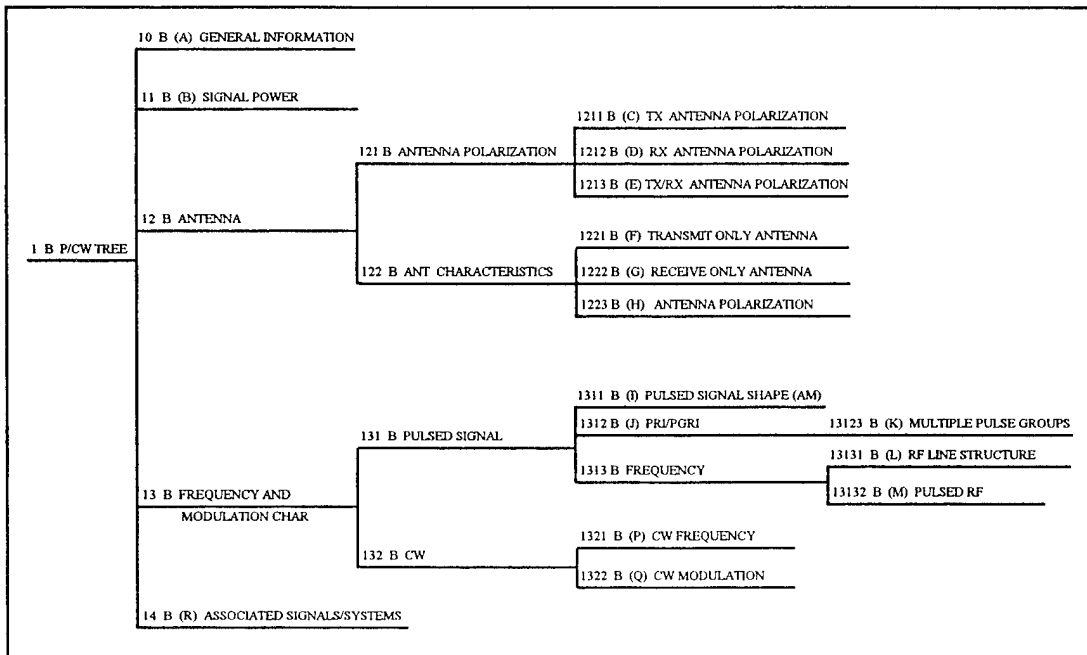


Figure 54. The Pulsed/Continuous Wave (P/CW) Parametric Tree [3].

Also represented in the P/CW tree are subfile codes. The subfile codes are captured by parentheses (i.e., the **(B)** in “**11 B (B) SIGNAL POWER**” in Figure 54). Subfile codes were intended to convey the semantics of high-level emitter and signal characteristics [3]. The subfiles actually contain the parametric data. They are major groupings (i.e., subtrees) within the parametric tree which contains the logically related data.

Usage codes are also represented in the P/CW tree. Usage codes distinguish which branches and parameters are applicable to which users. These codes are necessary since all the branches and parameters in the EWIRDB are not applicable to all users. [3] For example, one branch may provide useful data for a kilting analyst but not for an S&TI analyst. Therefore, the B in 121 B ANTENNA in Figure 54 indicates that the ANTENNA branch is applicable to all users of the database; whereas other codes such as K, E, and N are used for the various other agencies.

Figure 55 introduces yet another notation used in the parametric tree. Here, a parameter is shown with a two digit decimal number that differentiates between two parameters in a given branch. The branch number combined with the two-digit decimal number is referred to as the parametric number. Thus, locating a parameter within the tree is a matter of indexing into the data via the parametric number. For example, parametric number 121X3.10 indexes to the parameter .10 B 3 PATTERN PEAK OFFSET under the 121X3 B 2 CROSS POLARIZATION CHAR branch in Figure 55 (The X in the branch number is a variable that specifies the type of antenna being considered, i.e., transmit, receive, or transmit and receive. The variable takes on the value 1, 2, or 3, accordingly). [3]

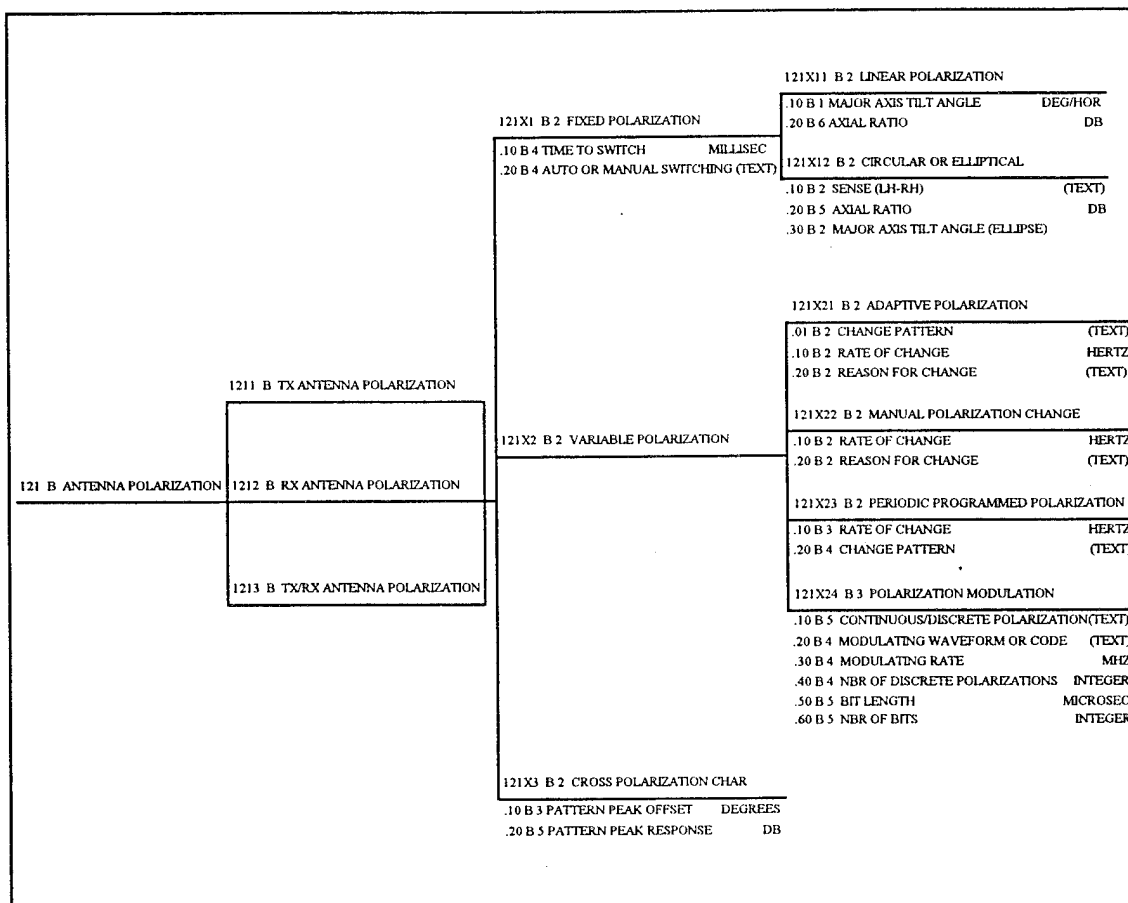


Figure 55. An Exploded View of the P/CW Tree [3].

Thus far we have examined one small portion of the hierarchical structure used within the EWIRDB parametric trees. This hierarchical structure uses branch headings, subfile codes, usage codes and a decimal notation to identify various EW parameters within branches of the parametric tree. This method of modeling the EWIR data clearly demonstrates some of the complexity of the structure.

In addition to the complex structure of the "hierarchical type" model, this format lacks the more expressive concepts with which to associate more meaning or semantics to the data. Therefore, this modeling construct is also misleading in its representation of parametric data. It is this inability to accurately represent the data semantics that reduces the EWIRDB's effectiveness as a database. Furthermore, the user must work much harder to process, decipher and interpret the data.

An example of a poor data relationship is provided in Figure 55. Specifically, the branches labeled **121X1 B 2 FIXED POLARIZATION**, **121X2 B 2 VARIABLE POLARIZATION**, and **121X3 B 2 CROSS POLARIZATION CHAR**. This type of hierarchical structure appears to indicate that antenna polarization is either fixed, variable, or cross. However, this is not true. Actually, antenna polarization can be either fixed or variable, and cross polarization is a characteristic that may be exhibited by all antennas. This is one example in which the model does not adequately portray the true data semantics.

Another example of poor data representation is the relationship between **FIXED POLARIZATION** and **LINEAR POLARIZATION**. According to the model (Figure 55), it appears that linear polarization is part of fixed polarization. However, linear polarization is actually a potential characteristic of all antenna polarization types [4]. Again, the EWIR tree structure lacks to capture the actual meaning of the data.

A further problem of the EWIRDB involves the "global" representation or layout of the data. To gain a complete assessment of an emitter's performance, one must consider both the P/CW tree and the RPA tree. Therefore, parametric data is dispersed over two large disjoint structures. This implies that the user must search both trees for

their associated parameters to ensure all performance criteria is considered for any given emitter. [3]

To further compound matters, the P/CW and RPA trees are designed to characterize only the parametric data. However, various other important information is contained within the EWIRDB. This information includes administrative, commentary, and reference data. This data is associated with the parametric data and provides additional information in describing any given emitter's performance.

The organization of the administrative, commentary, and reference data is contained within a file structure format. This file structure is not a data model but rather an output format file structure. This format known as TERF (Technical Electronic intelligence Reference File Format) is also a complex and rather cryptic format and structure. Reference [3] provides a more detailed description of this format.

In summary, the EWIRDB presents several problems associated with data representation and data modeling which affect the users ability to access, interpret, and understand the data. It is these problems that lead us to pursue the possibility of using some of the more classical data models (i.e., network model) to implement a small representative portion of the EWIRDB on the M²DBMS.

2. The Conceptual EWIR Network Data Model

To reiterate, the EWIRDB is a very large and intricate database, therefore, reference [3] is based on a portion of the EWIRDB. Likewise, reference [4] implements a subset of the specification proposed by reference [3]. To examine either of the specifications of reference [3] and [4] would exceed the scope of this thesis, thus our goal was to extract a small "sample" of the specification of the above references and model this "sample" in terms of a network representation.

The models and specifications represented by references [3] and [4] were designed for an object-oriented implementation on the M²DBMS. However, the conceptual design suggested by [3] will also serve as the conceptual design for our

network representation. The purpose of the conceptual design is to illustrate the specifications and requirements of the database in some conceptual schema of the database. This schema is a high-level description of the structure of the database. This phase (i.e., conceptual design) is conducted totally independent of a particular DBMS that will be used to implement the database. Therefore, since the conceptual design proposed by [3] has been accepted, our network model representation will be based on the object-oriented conceptual design.

Figure 56 depicts a global view of the conceptual schema proposed by [3]. As described earlier (Chapter II), the EWIRDB is the result of the merging of data from three contributory sources. Figure 56 shows this merging using the concept of aggregation. The symbols used in all of the conceptual diagrams is consistent with reference [3].

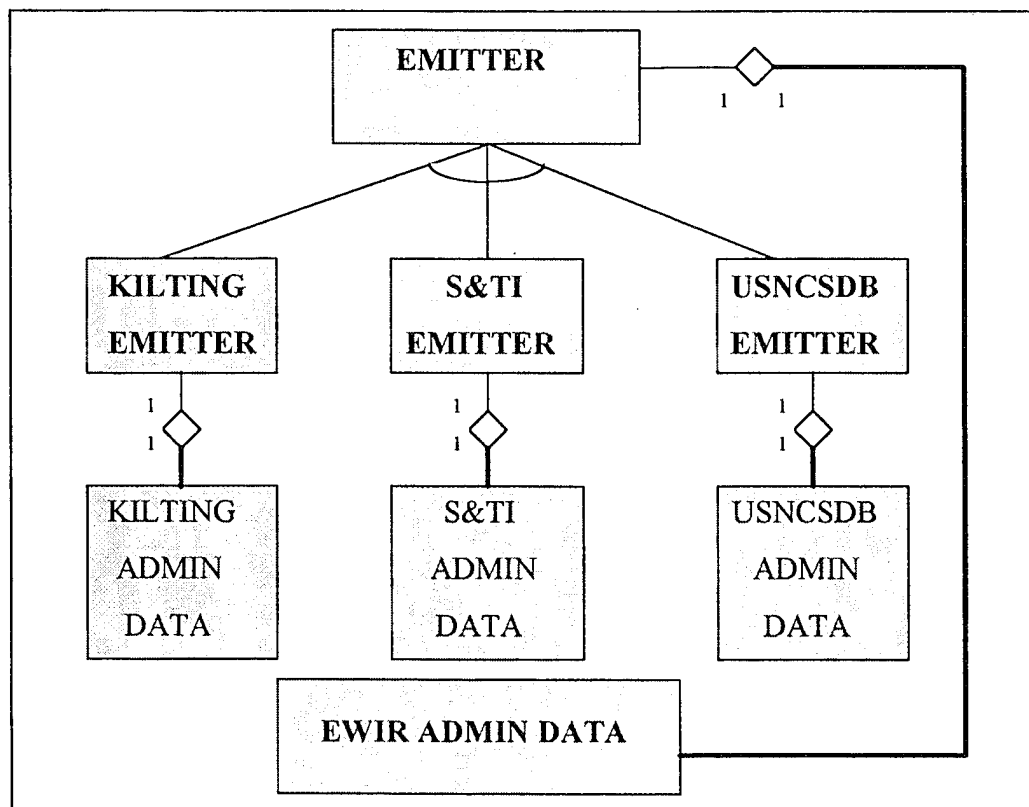


Figure 56. Global View of the Conceptual Schema for the EWIRDB.

Our specification represents a portion of the parametric data (i.e., antenna data) contained within one of the emitter type objects. Figure 57 shows a conceptual schema of the S&TI emitter data. The S&TI agency searches all available information and generates performance assessments. These assessments are then used to assist in the development of receiver capabilities. Likewise, USNCSDB data, which is derived from equipment specifications, also includes receiver performance data. Essentially the S&TI and the USNCSDB conceptual designs are the same.

The kilting emitter, however, is different than the S&TI and USNCSDB emitters. The kilting emitter conceptual schema does not contain receiver data. Kilting data are obtained from the direct analysis and measurement of emitter signals following signal intercept. Therefore, kilting data reveals nothing about receiver performance. [3]

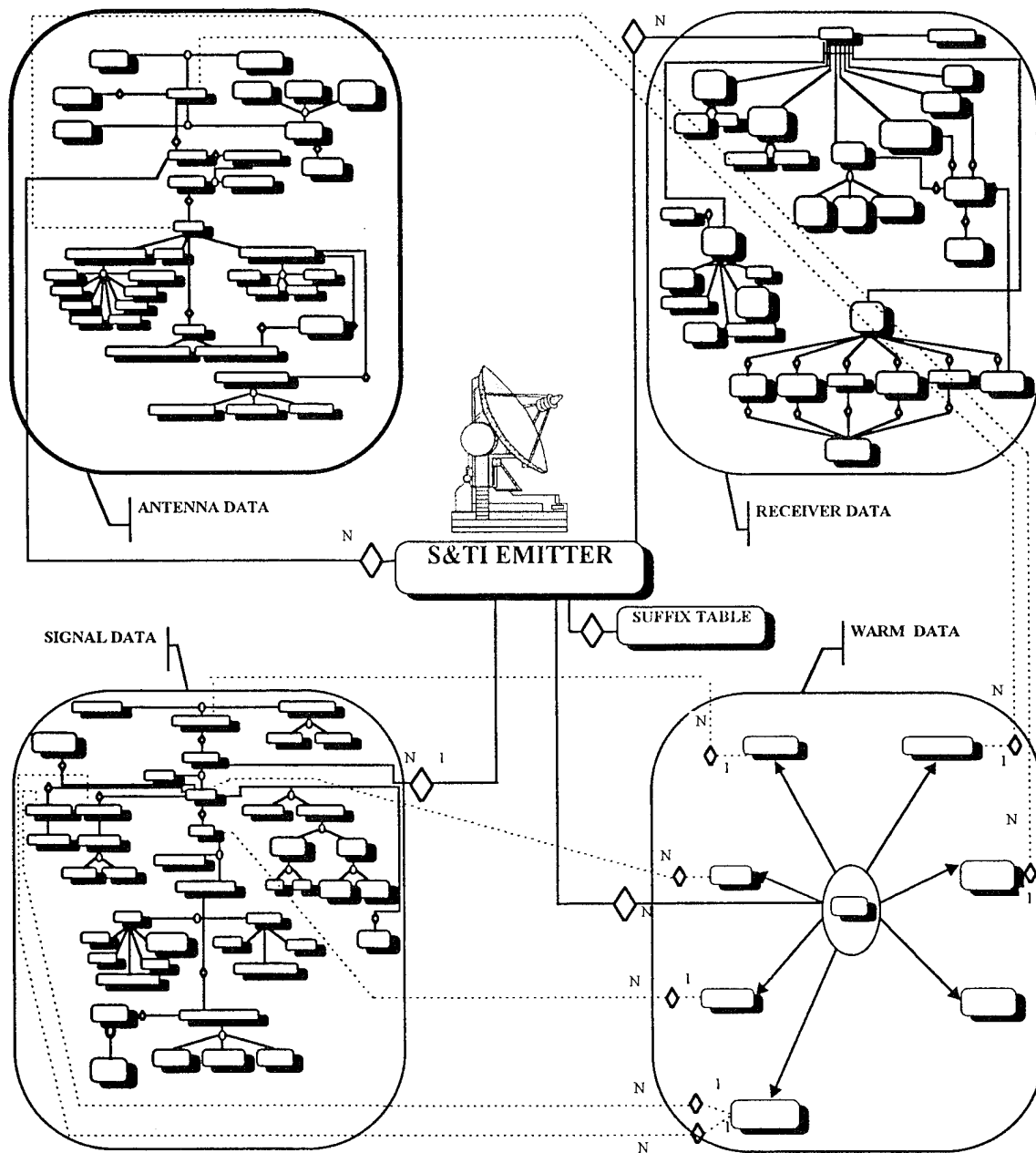


Figure 57. Conceptual Schema of the S&TI Emitter [3].

3. The Antenna Data

Common to all of the emitter types (S&TI, USNCSDB, and Kiltling) is the antenna data. Our portion of the EWIR implementation is based on this data. Figure 57 shows the antenna data grouping within the composite of the S&TI emitter and the other related data groupings (i.e., Receiver data, Signal data, and WARM data). Figure 58 depicts an expanded view of the antenna data grouping. The objects that are shaded gray in Figure 58 represent the objects that will be transformed to our network model representation of the antenna data (individual attributes are not show here but will be illustrated in the network conceptual schema). Figure 59 shows yet another (clearer) view with some of simbling branches omitted (from Figure 58). Again, the symbols used in these figures are consistent with reference [3].

The antenna data grouping is one of the integral components of the emitter. The 1:N relationship between emitter and antenna signifies that any one emitter may have many different antenna components. The 1:1 relationship between antenna and radiation pattern and the 1:1 relationship between antenna and polarization indicates that any specific antenna has a radiation pattern and a polarization. Polarization contains a specialization with two disjoint constraints. This signifies that polarization has four possible combinations: Linear-fixed, linear-variable, circular_or_elliptical-fixed, and circular_or_elliptical-variable. In addition, cross polarization is modeled correctly by the 1:1 relationship with polarization.

An antenna may radiate either directionally or omni-directionally as indicated by the specialization and the disjoint hierarchy. If the antenna radiates directionally then it is associated with one or more scanning techniques and/or one or more tracking functions. The specializations of scan and track are associated with an overlapping constraint that indicates a directional antenna having the possibility of simultaneous scanning techniques (i.e., mechanical, manual and electrical) or tracking functions (i.e., mechanical and electrical).

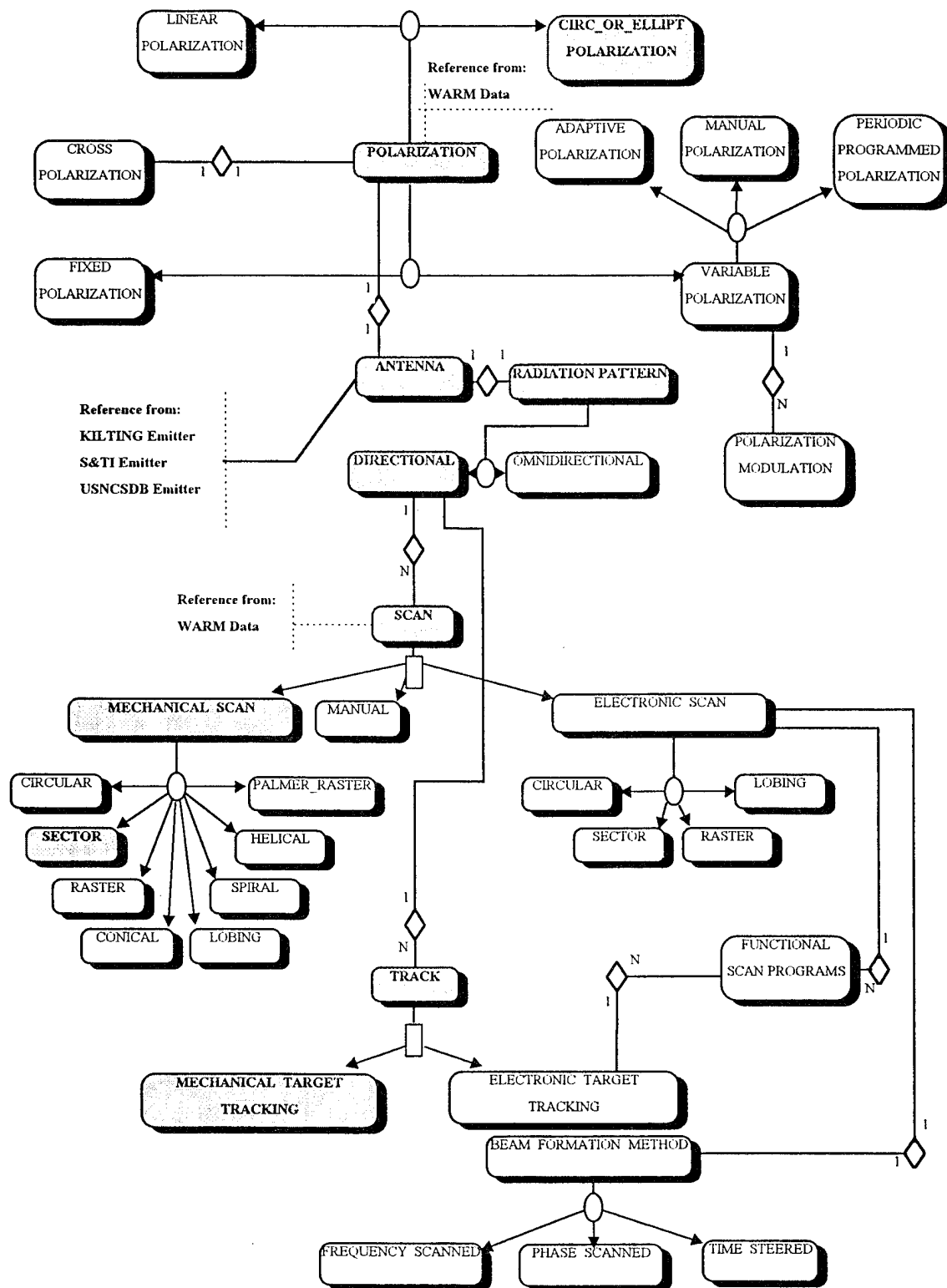


Figure 58. Expanded View of the Antenna Data [3].

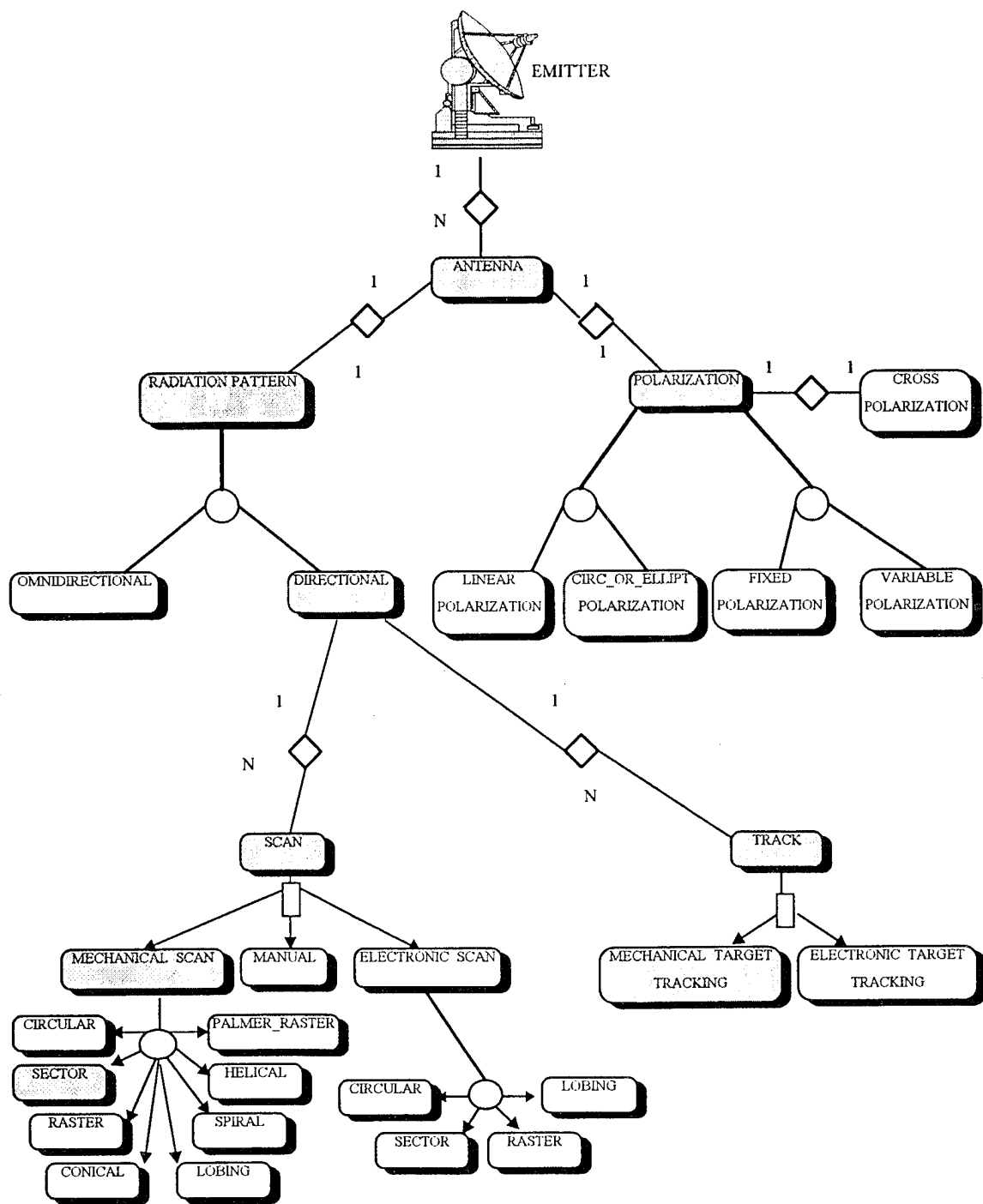


Figure 59. Enhanced View of the Antenna Data.

In comparing Figure 59 (or Figure 58) with the antenna data parametric tree representation discussed earlier, it is clearly evident that the new conceptual design proposed by [3] is a significant improvement over the parametric tree hierarchy.

4. Transforming the Object-Oriented Model to the Network Model

Since our conceptual view of the antenna data is an object-oriented "semantic" representation, it is necessary to examine the relationship between the network data model and the object-oriented model (i.e., how to transform the object-oriented model to the network model).

In a limited context, the object-oriented model (Figure 59) can be mapped to the network model. However, all the relationships of the object-oriented model are limited to binary (two-object) sets and 1:1 and 1:N relationships in the network model. Recollect from Chapter II, section B., of this thesis, that in a network schema we can explicitly represent a relationship type if it is 1:N. If the relationship type is 1:1, a set type may be used, however, a constraint must be enforced to ensure that each set instance has at most one member record. For M:N relationship types, the standard representation is to use the two set types and linking record. It is also worth mentioning that the network model allows for vector fields and repeating groups. This allows us to directly represent composite and multivalued attributes. However, this modeling capability is not necessary with our representation.

For the diagrammatic network representation of data structures, we have logical records, which are connected with other logical records through physical links consisting of the records' addresses on disk. Each link represents a relationship between exactly two records. The relationship between two record types connected by the binary link is referred to as a set. [20]

To map a 1:1 or a 1:N relationship to the network model, first, it is necessary to create a set type that relates the two record types. For a 1:1 relationship type, the owner and member record type are arbitrarily chosen; however, it is preferable to choose the

record that represents total participation in the relationship type as the member record. Another possibility for mapping 1:1 relationships is to merge the objects, their attributes, and the relationship into one record. This is useful if both objects in the relationship participate totally (i.e., total participation).

For 1:N relationships, the record type that represents the object on the “one” side of the relationship becomes the owner record type, and the record type that represents the other object on the “many” side becomes the member record type. [6]

The result of mapping Figure 59 (only the shaded objects) to the network conceptual model is illustrated in Figure 60. This represents our conceptual schema for the antenna data in the network model representation.

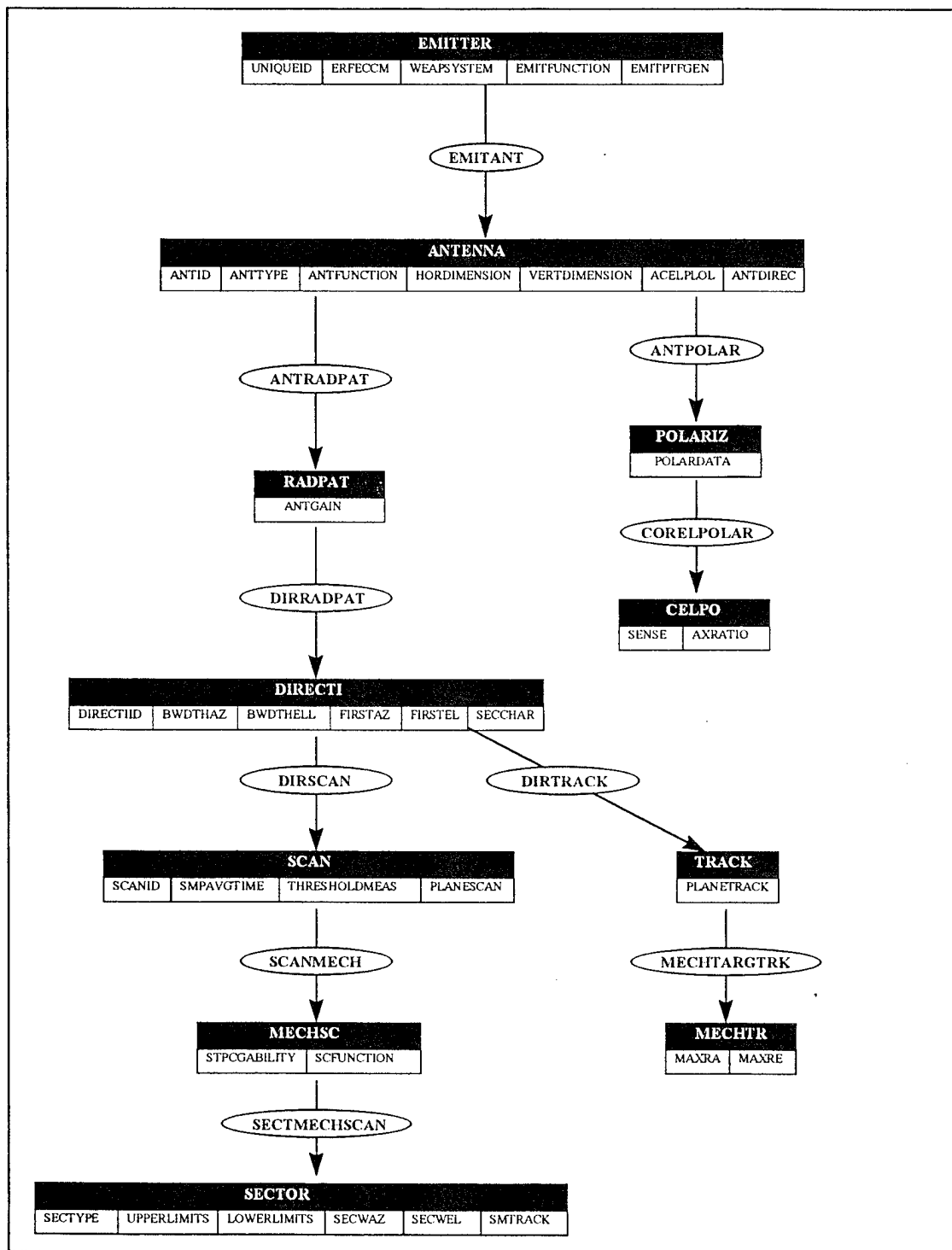


Figure 60. Network Conceptual Schema.

B. DATA DEFINITION OF THE EWIR DATABASE

Now that the specification and the conceptual schema for our network model has been completed, the next issue is to translate the conceptual schema (Figure 60) to a database schema that the network interface can process. Once the schema is processed, the network interface generates the corresponding template and descriptor files described later in this section.

1. The Schema Listing

Chapter II, section B., of this thesis addresses how the DDL is used to implement the schema. This is a relatively simple process once the specification and conceptual schema (Figure 60) has been developed. The actual database schema for our specification is depicted in Figure 61 and continued in Figure 62. The filename is **TEWIRdml.db**.

One of the problems, however, in translating from the specification to the actual database schema involves some formatting restrictions inherent to the network interface. Specifically, naming conventions and syntax within the schema file itself. Record names, attribute names and set names are limited to fifteen characters. Furthermore, dashes and underscores are not permitted for any of the names. Our specification (Figure 60) reflects names that are accepted by the network interface. Therefore, these exact names are used in the actual database schema reflected in Figure 61. However, it is not common practice to make these names difficult to read or understand. Usually it is desirable to have semantically meaningful record, attribute and set type names that are the same for both the specification and the actual database schema. We chose to use names in the specification that we knew were implementable in the network interface database schema to provide for a direct mapping from the conceptual schema to the actual database schema. Also, for continuity, we attempted to use similar names as those used in the object-oriented specification and implementation (i.e., reference [4]).

```

SCHEMA NAME IS TEWIR;
RECORD NAME IS EMITTER;
DUPLICATES ARE NOT ALLOWED FOR UNIQUEID;
    UNIQUEID ;          CHARACTER 15.
    ERFECCM ;          CHARACTER 15.
    WEAPSYSTEM ;        CHARACTER 15.
    EMITFUNCTION ;      CHARACTER 15.
    EMITPTFGEN ;        CHARACTER 15.
RECORD NAME IS ANTENNA;
    Duplicates ARE NOT ALLOWED FOR ANTID;
    ANTID ;            CHARACTER 15.
    ANTTYPE ;          CHARACTER 15.
    ANTFUNCTION ;      CHARACTER 15.
    HORDIMENSION ;     CHARACTER 15.
    VERTDIMENSION ;    CHARACTER 15.
    ACELPLOL ;         CHARACTER 15.
    ANTIDIREC ;        CHARACTER 15.
RECORD NAME IS POLARIZ;
    POLARDATA ;        CHARACTER 15.
RECORD NAME IS CELPO;
    SENSE ;            CHARACTER 15.
    AXRATIO ;          CHARACTER 15.
RECORD NAME IS RADPAT;
    ANTGAIN ;          CHARACTER 15.
RECORD NAME IS DIRECTI;
    DIRECTIID ;        CHARACTER 15.
    BWDTHAZ ;          CHARACTER 15.
    BWDTHEL ;          CHARACTER 15.
    FIRSTAZ ;          CHARACTER 15.
    FIRSTEL ;          CHARACTER 15.
    SECCHAR ;          CHARACTER 15.
RECORD NAME IS SCAN;
    SCANID ;           CHARACTER 15.
    SMPAVGTIME ;       CHARACTER 15.
    THRESHOLDMEAS ;    CHARACTER 15.
    PLANESCAN ;        CHARACTER 15.
RECORD NAME IS MECHSC;
    STPCGABILITY ;     CHARACTER 15.
    SCFUNCTION ;        CHARACTER 15.
RECORD NAME IS SECTOR;
    SECTYPE ;          CHARACTER 15.
    UPPERLIMITS ;      CHARACTER 15.
    LOWERLIMITS ;      CHARACTER 15.
    SECWAZ ;           CHARACTER 15.
    SECWEL ;           CHARACTER 15.
    SMTRACK ;          CHARACTER 15.
RECORD NAME IS TRACK;
    PLANETRACK ;       CHARACTER 15.
RECORD NAME IS MECHTR;
    MAXRA ;            CHARACTER 15.
    MAXRE ;            CHARACTER 15.
SET NAME IS EMITANT;
    OWNER IS EMITTER;
    MEMBER IS ANTENNA;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        SET SELECTION IS BY VALUE OF UNIQUEID IN EMITTER;

```

Figure 61. The EWIR Schema.

```

SET NAME IS ANTPOLAR;
OWNER IS ANTENNA;
MEMBER IS POLARIZ;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF ANTID IN ANTENNA;
SET NAME IS CORELPOLAR;
OWNER IS POLARIZ;
MEMBER IS CELPO;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF POLARDATA IN POLARIZ;
SET NAME IS ANTRADPAT;
OWNER IS ANTENNA;
MEMBER IS RADPAT;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF ANTID IN ANTENNA;
SET NAME IS DIRRADPAT;
OWNER IS RADPAT;
MEMBER IS DIRECTI;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF ANTGAIN IN RADPAT;
SET NAME IS DIRSCAN;
OWNER IS DIRECTI;
MEMBER IS SCAN;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF DIRECTIID IN DIRECTI;
SET NAME IS SCANMECH;
OWNER IS SCAN;
MEMBER IS MECHSC;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF SCANID IN SCAN;
SET NAME IS SECTMECHSCAN;
OWNER IS MECHSC;
MEMBER IS SECTOR;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF SCFUNCTION IN MECHSC;
SET NAME IS DIRTRACK;
OWNER IS DIRECTI;
MEMBER IS TRACK;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF DIRECTIID IN DIRECTI;
SET NAME IS MECHTARGTRK;
OWNER IS TRACK;
MEMBER IS MECHTR;
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED;
    SET SELECTION IS BY VALUE OF PLANETRACK IN TRACK;
$

```

Figure 62. The EWIR Schema (continued).

2. The Template File

As a result of successfully loading the EWIR schema on the network interface of the M²DBMS, a template file is generated. The filename for the template file for our network EWIR database is **TEWIR.t**. This template file is shown in Figure 63 and continued in Figure 64. The purpose of the template file is to describe the structure or specification of the network database in an equivalent attribute-based (kernel) database.

The first line of the template file corresponds to the template file name (or name of the database). The next line indicates the number of templates within the database. The next line shows the number of attributes in that particular template. The following line depicts the template name (corresponds to the record type). After the template name the attributes for that template are listed with their respective type (i.e., string, integer, etc.). After all the attributes for that template have been listed, then the next number listed represents the number of attributes for the next template. This process continues until all of the templates and attributes in the database have been identified.

3. The Descriptor File

Also derived from the successful loading of the schema file, is the descriptor file. The filename for the descriptor file for our network EWIR database is **TEWIR.d**. The descriptor file also contributes to specifying the transformation of the network database to the attribute-based database. The descriptor file provides a listing of all the record types in the network database to the kernel system. Figure 65 illustrates the descriptor file for our implementation.

Furthermore, the descriptor file contains information concerning the "constraints" placed upon the attributes within the template. There are three descriptor types that may be imposed on an attribute. For example, the type "a" descriptor type specifies that an attribute exhibits a disjointed range of values (i.e., $0 \leq \text{attribute value} \leq 50$). The type "b" descriptor type indicates an attribute with a specific value (i.e., Emitter Function =

phased array). The type "c" descriptor type indicates an attribute that possesses a dynamic range that is initialized at run time. [9]

```
TEWIR          /* template file name */
11             /* number of templates */
7             /* number of attributes in the Emitter template */
Emitter        /* template name: Emitter */
TEMP s        /* "Emitter" is an attribute template of type string */
DBKEY I        /* DBKEY is an attribute of type integer */
UNIQUEID s
ERFECCM s
WEAPSYSTEM s
EMITFUNCTION s
EMITPTFGEN s
10
Antenna
TEMP s
DBKEY i
ANTID s
ANTTYPE s
ANTFUNCTION s
HORDIMENSION s
VERTDIMENSION s
ACELPLOL s
ANTDIREC s
MEMEMITANT i
4
Polariz
TEMP s
DBKEY i
POLARDATA s
MEMANTPOLAR i
5
Celpo
TEMP s
DBKEY i
SENSE s
AXRATIO s
MEMCORELPOLAR i
4
Radpat
TEMP s
DBKEY i
ANTGAIN s
MEMANTRADPAT i
```

Figure 63. The EWIR Template File: TEWIR.t.

```

9
Directi
TEMP s
DBKEY i
DIRECTIID s
BWDTHAZ s
BWDTHEL s
FIRSTAZ s
FIRSTEL s
SECCHAR s
MEMDIRRADPAT i
7
Scan
TEMP s
DBKEY i
SCANID s
SMPAVGTIME s
THRESHOLDMEAS s
PLANESCAN s
MEMDIRSCAN i
5
Mechsc
TEMP s
DBKEY i
STPCGABILITY s
SCFUNCTION s
MEMSCANMECH i
9
Sector
TEMP s
DBKEY i
SECTYPE s
UPPERLIMITS s
LOWERLIMITS s
SECWAZ s
SECWEL s
SMTRACK s
MEMSECTMECHSCAN i
4
Track
TEMP s
DBKEY i
PLANETRACK s
MEMDIRTRACK i
5
Mechtr
TEMP s
DBKEY i
MAXRA s
MAXRE s
MEMMECHTARGTRK i

```

Figure 64. The EWIR Template File: TEWIR.t (continued).

```
TEWIR
TEMP b s
! Emitter
! Antenna
! Polariz
! Celpo
! Radpat
! Directi
! Scan
! Mechsc
! Sector
! Track
! Mechtr
@
$
```

Figure 65. The EWIR Descriptor File: TEWIR.d.

C. LOADING THE EWIR RECORD DATA

As we mentioned earlier in this thesis (Chapter III, section B.), the issue of loading the record data using the network interface was never addressed in prior research. Evident from our research, it appears that the data was intended to be loaded via the CODASYL-DML “STORE” command. This command allows for a record-by-record insertion of data, but this command is not functional as we described in Chapter III.

The issue of loading data in the network interface was overcome by implementing a *mass load* function that was structured after the mass load function for the object-oriented interface (relational interface also employs the mass load function). The mass load function “reads” the record data file (“r” file) to load and populate the database. It is important to note that prior to loading the database, the template, descriptor, and record files must be created. Loading the database depends on information contained within all of these files. Also, the template and descriptor files must be present on the back-end system, for it these files that help manage the data between the various back-

ends. These files are not automatically placed on the back-end system, therefore, the user must manually copy the template and descriptor files to the back-end. Chapter III, section C., of this thesis describes the mass load function and the loading of data in greater detail.

The record file contains all the record data the user intends to store in the database. When the user wishes to add more data to the database, this data must be added to the record data file, and then processed through the mass load function again. Since the STORE command is not functional, it is not possible for the user to load data one record at a time.

In commercial database systems, loading the data is usually not an issue. The user typically manipulates an easy-to-use interface that allows for the quick insertion of data. The user is not concerned with the physical storage structures of the data or how the data is stored. However, to load data in a network database on the M²DBMS, the user must create the record data file to successfully load the data. Figure 66 shows the record data file (**TEWIR.r**) for the EWIR database.

The record data names and values indicated in this file are not the actual real-world EWIR database names and values. Since this thesis is an unclassified research project, real-world names and data values were not used. Our intent was to employ the same names and values used by the object-oriented implementation to preserve continuity. However, the formatting restrictions discussed earlier precluded us from doing so in most cases.

To construct the record data file (**TEWIR.r**), the user must use the template file (**TEWIR.t**) as a template or guide to place the appropriate data with its corresponding record types and attributes. The record types in the record data file are placed in the same order as they are found in the template file (or schema since the template is derived from the schema). The attribute values are placed in the record data file according to their order in the template file. The "@" symbol in the record data file is used to separate the various record types (see Figure 66).

```

TEWIR                                /* name of the database */
@
Emitter                             /* record type name */
1 Ee1 Ww1 Aa10 Parabolic Modpulsewave /* a record in the record type */
2 Ee2 Ww2 Aa6  Phasedarray Modpulsewave
3 Ee3 Ww3 Sa21 Phasedarray Parabolic
@                                  /* delimiter symbol */
Antenna                             /* next record type */
4 Aa1 Phasedarray Longrngaa 3ft 4ft Pdata1 Rad1 1
5 Aa2 Square_sail Longrngaa 3ft 4ft Pdata2 Rad2 2
6 Aa3 Parabolic Lngrngaa 325ms 300kw Pdata3 Rad2 3
@
Polariz
7 Pdata1 4
8 Pdata2 5
9 Pdata3 6
@
Celpo
10 Left I20db 7
11 Parabolic I300kw 8
12 Parabolic I20db 9
@
Radpat
13 10db 4
14 10db 5
15 4ft 6
@
Directi
16 Dir1 325ms 300kw 4ft 325ms Sca1 13
17 Dir2 300kw 4ft 325ms 300kw Sca2 14
18 Dir2 300kw 4ft 325ms 300kw Sca2 15
@
Scan
19 SCA1 325ms 4ft Phasedarray 16
20 SCA2 300ms 325kw Parabolic 17
21 SCA2 300ms 325kw Parabolic 18
@
Mechsc
22 Phasedarray Parabolic 19
23 Parabolic Modpulsew 20
24 Parabolic Modpulsew 21
@
Sector
25 Unidirectional 128ms 100ms 325ms 300kw Tr1 22
26 Parabolic Upperlevel2 Lowerlevel2 300kw 4ft Tr1 23
27 Parabolic Upperlevel2 Lowerlevel2 300kw 4ft Tr2 24
@
Track
28 Horiz45 16
29 Parabolic 17
30 Parabolic 18
@
Mechtr
31 128ms Upperlevel12 28
32 Upperlevel12 128hz 29
33 Upperlevel12 128hz 30
$

```

Figure 66. The EWIR Record Data File: TEWIR.r.

With the successful loading of the EWIR record data file by way of the mass load function, this completes the data definition and creation of our network EWIR database. The next chapter summarizes our findings and results of this thesis and provides recommendations for future research involving the network interface of the M²DBMS.

V. CONCLUSION

In this thesis, we have demonstrated the use of the M²DBMS as a test-bed for implementing a portion of a real-world database, specifically the Electronic Warfare Integrated Reprogramming Database (EWIRDB). The EWIRDB is an important tool used in electronic warfare (EW) research, development and battlefield analysis. It contains mission-critical data on the EW systems of friendly and hostile forces.

However, in its current format, structure and "model", the EWIRDB is very complex and difficult to understand from a users point of view. The parametric tree with its deceptive hierarchical structure, provides a poor modeling construct that obscures the intended semantics and representation of the EW data.

This thesis is the result of continuing research to examine the possibility of representing the EWIRDB using various other data models (i.e., network, relational, and object-oriented) on the M²DBMS to enhance the data representation, semantics and query processing of the EWIRDB. With the accomplishment of producing various other data model representations of the EWIRDB, the objective of providing cross-model access capabilities among these different models can then be pursued.

Currently research has been completed in modeling and implementing a portion of the EWIRDB on the object-oriented interface. With the completion of this thesis and that of research group, Edwards/Scrivener, a portion of the EWIRDB will be modeled and implemented on the network and relational interfaces of the M²DBMS.

To reiterate, our primary goal was to implement a representative portion of the EWIRDB on the network interface. However, in order to attain this goal, it was necessary to first address and accomplish the following:

- Activate the network interface
- Test the interface to determine its capabilities and limitations

- Design a network EWIR model

The remainder of this chapter describes the contributions of our research and potential future research areas.

We have successfully returned the network interface to its original operational state. We have documented the changes made to the code and have implemented these changes to reinstate the network interface. Correction and modifications to the program code were made throughout the program modules of the language interface layer. Upon discovering problems with the “STORE” command of the CODASYL-DML, we implemented a mass load function that processes a record data file to quickly load and populate the database.

Once the network interface was operational and data was loaded, we investigated the capabilities and limitations of the interface by thoroughly testing a variety of different schemas and queries. Our testing strategy comprised of several basic queries performed on a sample “PARTS” database. The basic queries used in the testing assisted in determining record access points and possible navigation traversals within the network hierarchy and structure.

The results of testing revealed significant limitations. Records could be accessed directly; however, navigation was essentially constrained to a two level hierarchy. These limitations imposed significant restrictions on the query processing of the network interface, thus making the network interface virtually useless in its utility. Based on these findings, it was not feasible to design, write and execute queries for our proposed EWIRDB design. Therefore, only the data definition portion of our proposed network EWIR model was successfully implemented.

As we just alluded to, a subset (i.e., the antenna data group) of the EWIRDB was represented in the network model form. The object-oriented conceptual design served as the basis for our network model. This thesis presented the network schema listing, the template file, the descriptor file and the record data file for our EWIR network model.

The schema was processed on the network interface and the template and descriptor files were properly generated. In addition, the record data file was loaded successfully. Therefore, the entire data definition process of our proposed network EWIR model design was fully implemented.

There are many potential areas for future research concerning the M²DBMS. Many of these possibilities have already been expressed in prior work conducted on the M²DBMS. Therefore, we will only mention the following.

We have proven that the network interface is extremely limited in its ability to execute transactions and queries. Therefore, if it is desired to conduct a full and detailed implementation using the network interface, serious inquiry to redesign the network interface should be considered.

APPENDIX. N_MASS_LOAD() FUNCTION SOURCE CODE

This appendix contains the source code for the mass load function for the network interface. The function is named *n_mass_load()* which is located in the **mass_ld.c** file in the **Lil** directory. The following source code is from the **mass_ld.c** file.

```
/* n_mss_load.c */

#include <stdio.h>
#include <ctype.h>
#include <strings.h>
#include <licommdata.h>
#include <dml_lildcl.h>
#include <dml.h>
#include "commdata.def"
#include "flags.def"

n_mass_load(fptrl,db_name, cnt)
FILE *fptrl;
char db_name[];
int cnt;
{
    /* Load the database records */
    int c, i, z, owner_flag, Record_cnt = 0;
    struct net_file_info *file_ptr;
    struct rtemp_definition *tmpl_ptr, *head_tmpl_list, *read_tmpl();
    char hold[2],
          record[REQLength+1],
          tmpl_name[AVLength+1],
          value_string[ANLength+1];

    hold[1] = '\0';
#ifdef EnExFlag
    printf("Enter n_mass_load\n");
    fflush(stdout);
#endif

    /* load a records file */

    /* Inform user about what is happening : added 930915 */
    printf("\n\n          <Loading Records, Please Stand By>
\n\n");

    /* read past the database id */
    while (getc(fptrl) != '\n');

    /* read template definition into memory */
    if (!(head_tmpl_list = read_tmpl(db_name)))
        return (FALSE);
}
```

```

#ifndef prrec_flag
    printf("    ");
#endif

#ifdef TimeFlag
    system("date > Time_data");
#endif

    while (TRUE) {

        while ((c = getc(fptrl)) == '\n');
#ifdef pr_flag
        printf("c = >%c<\n", c);
#endif

        /* new template */
        if (c == '@')
        {
            read_string_file(fptrl, tmpl_name, AVLength);
#ifdef pr_flag
            printf("tmpl_name = >%s<\n", tmpl_name);
#endif

            /* find template structure */
            tmpl_ptr = head_tmpl_list;
            while (strcmp(tmpl_ptr->rt_name, tmpl_name))
                if (tmpl_ptr->rt_next_tmpl)
                    tmpl_ptr = tmpl_ptr->rt_next_tmpl;

            else
            {
                UserError(19);
                free_tmpl_list(head_tmpl_list);
                return (FALSE);
            } /* end if(tmpl_ptr->rt_next_tmpl) */

        } else if (c == '$' || c == EOF) /* end of file */
            break;

        /* construct and transmit a record */
        else
        {
            strcpy(record, "[INSERT(");
            for (i = 0; i < tmpl_ptr->no_entries; i++) {
                strcat(record, "<");
                strcat(record, tmpl_ptr->rt_entry[i].attr_name);
#ifdef pr_flag
                printf("%s,", tmpl_ptr->rt_entry[i].attr_name);
#endif
                strcat(record, ",");
                if (!i)
                    strcat(record, tmpl_name);
            } else
            {
                if (i == 1) {
                    ungetc(c, fptrl);

```

```

    }
    read_string_file(fptrl, value_string, ANLength);
#ifdef pr_flag
    printf("%s\n", value_string);
#endif
    strcat(record, value_string);
    } /* end else (!i) */
    strcat(record, ">");
    if (i < tmpl_ptr->no_entries-1)
        strcat(record, ",");
    } /* end for loop */
    strcat(record, "]\n");
#ifdef prrec_flag
    printf("%d ", ++Record_cnt);
    printf("-> %s", record);
    printf("\nlength of record is %d\n\n", strlen(record));
#endif

    /* send the request to Request Preparation */
    for(z = 0; z < cnt; z++)
        TI_SSTrafUnit(db_name, record);

#ifdef prrec_flag
    Record_cnt += cnt;
    if(!(Record_cnt%5))
    {
        printf("%6d", Record_cnt);
        if (!(Record_cnt%100))
            printf("\n      ");
    }
    fflush(stdout);
#endif

    owner_flag = FALSE;
    dml_check_requests_left(file_ptr, owner_flag, TRUE);

    } /* end else (c== '@') */

} /* end while(TRUE) */

#ifdef TimeFlag
    system("date >> Time_data");
#endif

    /* free memory used by rtemp_definition's */
    free_tmpl_list(head_tmpl_list);
    printf("\n\n");

#ifdef EnExFlag
    printf("Exit n_mass_load \n");
    fflush(stdout);
#endif
} /* end n_mass_load */

```


LIST OF REFERENCES

- [1] Emdi, B., *The Implementation of a Network CODASYL-DML Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.
- [2] Worthery, C., *The Design and Analysis of a Network Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.
- [3] Coyne, K., *The Design and Analysis of an Object-Oriented Database of Electronic Warfare Data*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1996.
- [4] McKenna, T., Lee, J., *The Object-Oriented Database and Processing of Electronic Warfare Data*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1996.
- [5] National Air Intelligence Center, *Electronic Warfare Integrated Reprogramming Database (EWIRDB) Guide*, Volume 1, April 1994.
- [6] Elmasri, R., and Navathe, S., *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., 1994.
- [7] Demurjian, S., *The Multi-Lingual Database System - A Paradigm and Test-Bed for the Investigation of Data-Model Transformation, Data Language Translations and Data-Model Semantics*, Doctoral Thesis, Ohio State University, January 1987.
- [8] Meeks, A., *The Instrumentation of the MultiBackend Database System, Master's Thesis*, Naval Postgraduate School, Monterey, California, June 1993.
- [9] Kellet, D., Tae Wook, K., *Supporting the Object-Oriented Database on the Kernel Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.
- [10] Bourgeois, P., *The Instrumentation of the Multimodel and Multilingual User Interface*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1993.

- [11] Coker Jr., H., *Accessing a Functional Database via CODASYL-DML Transactions*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1987.
- [12] Sheehan, W., *The Design of a DL/I-to-Network Interface for the Multi-Model, Multi-Lingual, Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1989.
- [13] Walpole, D., Woods, A., *Accessing Network Databases via SQL Transactions in a Multi-Model Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1989.
- [14] Rodeck, B., *Accessing and Updating Functional Databases Using CODASYL-DML*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1996.
- [15] Hsiao, D. K., "Interoperable and Multidatabase Solutions for Heterogeneous Databases and Transactions," a speech delivered at ACM CSC 1995, Nashville, Tennessee, March 1995.
- [16] Watkins, S., *A Porting Methodology for Parallel Database Systems*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.
- [17] Barnes, G., *A Conceptual Approach to Object-Oriented Data Modeling*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1994.
- [18] Hsiao, D. K., "A Parallel, Scalable, Microprocessor-Based Database Computer for Performance Gains and Capacity Growth," IEEE Micro, December 1991.
- [19] Hall, J. E., *Performance Evaluations of a Parallel and Expandable Database Computer - The Multi-Backend Database Computer*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1989.
- [20] Hansen, G., and Hansen, J., *Database Management and Design*, Prentice-Hall, Inc., 1992.
- [21] Yao, S. Bing, *Principles of Database Design - Volume I Logical Organizations*, Prentice-Hall, Inc., 1985.
- [22] Olle, William T., *The Codasyl Approach to Data Base Management*, John Wiley & Sons, Ltd., 1978.

- [23] Date, C. J., *An Introduction to Database Systems, Fourth Edition*, Addison-Wesley Publishing Company, Inc., 1986.
- [24] Navathe, S. B., *Evolution of Data Modeling for Databases*, Communications of the ACM, Vol. 35, No. 9, September 1992.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
 8725 John J. Kingman Rd., STE 0944
 Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
 Naval Postgraduate School
 411 Dyer Rd.
 Monterey, California 93943-5101

3. Chairman, Code CS 1
 Computer Science Department
 Naval Postgraduate School
 Monterey, California 93943

4. Dr. C. Thomas Wu, Code CS/KA 1
 Computer Science Department
 Naval Postgraduate School
 Monterey, California 93943

5. Dr. David K. Hsiao, Code CS/HS 1
 Computer Science Department
 Naval Postgraduate School
 Monterey, California 93943

6. LCDR Timothy J. Werre 1
 Computer Science Department
 Naval Postgraduate School
 Monterey, California 93943

7. CPT Barry A. Diehl 1
 Computer Science Department
 Naval Postgraduate School
 Monterey, California 93943

8. Sharon Cain 1
NAIC/SCDD
4115 Hebble Creek Rd
Wright-Patterson AFB, Ohio 45433-5622
9. Doris Mleczko, Code p22305 1
Weapons Division
Naval Air Warfare Center
Pt. Mugu, CA 93042