THE EVALUATION OF ABSTRACT DATA TYPES AS AN IMPLEMENTATION TOOL FOR DATABASE MANAGEMENT SYSTEMS

by

Anthony James Baroody, Jr.

Computer Sciences Technical Report #330

August 1978

THE EVALUATION OF ABSTRACT DATA TYPES AS AN IMPLEMENTATION TOOL FOR DATABASE MANAGEMENT SYSTEMS

Anthony James Baroody, Jr.

Under the supervision of
Assistant Professor David Johns DeWitt

ABSTRACT

This dissertation examines the application of current research on abstract data types and on generic procedures to the implementation of a network model database management system. The data manipulation routines are examples of generic procedures. The data manipulation routines utilize the descriptors of the record and set types in the schema and subschema to determine the functions which are to be performed for a given actual parameter.

A generic procedure model of a database management system is presented and used to describe the techniques used in current systems to provide these descriptors to

the data manipulation routines. To preserve data independence, an approach employing run-time interpretation is commonly used.

The generic procedure model is extended to represent the schema and subschema as a shared collection of abstract data types. The data manipulation routines are implemented in terms of the data attributes and procedures associated with the abstract data types. When coupled with environment control mechanisms based on SIMULA 67, the abstract data type model is shown to support data independence, sharing of the schema and subschema, and access control to the schema through the subschema, while eliminating run-time interpretation.

The abstract data types representing the schema and subschema are generated by extending the concept of generalization developed by Smith and Smith. Generalization is used to define generic objects which represent the network data model. These objects are used as templates to translate the schema and subschema into the equivalent declarations of abstract data types which implement the generic objects.

A simulation model is developed to compare the abstract data type model with the interpretive approach. The simulation model is used in combinatation with analytic modeling to study the locality of references to

THE EVALUATION OF ABSTRACT DATA TYPES AS AN IMPLEMENTATION TOOL FOR DATABASE MANAGEMENT SYSTEMS

BY

ANTHONY JAMES BAROODY, JR.

A thesis submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1978

ABSTRACT

This dissertation examines the application of current research on abstract data types and on generic procedures to the implementation of a network model database management system. The data manipulation routines are examples of generic procedures. The data manipulation routines utilize the descriptors of the record and set types in the schema and subschema to determine the functions which are to be performed for a given actual parameter.

A generic procedure model of a database management system is presented and used to describe the techniques used in current systems to provide these descriptors to the data manipulation routines. To preserve data independence, an approach employing run-time interpretation is commonly used.

The generic procedure model is extended to represent the schema and subschema as a shared collection of abstract data types. The data manipulation routines are implemented in terms of the data attributes and procedures associated with the abstract data types. When coupled with environment control mechanisms based on SIMULA 67, the abstract data type model is shown to support data independence, sharing of the schema and subschema, and access control to the schema through the subschema, while eliminating run-time interpretation.

The abstract data types representing the schema and subschema are generated by extending the concept of generalization developed by Smith and Smith. Generalization is used to define generic objects which represent the network data model. These objects are used as templates to translate the schema and subschema into the equivalent declarations of abstract data types which implement the generic objects.

A simulation model is developed to compare the abstract data type model with the interpretive approach. The simulation model is used in combinatation with analytic modeling to study the locality of references to schema descriptors, the effect of multiprogramming on system overhead, and the effect of mass storage I/O time on system performance.

ACKNOWLEDGMENTS

I would like to express my appreciation to my advisor, David DeWitt. His encouragement and assistance during the development of this thesis made this research a pleasure for me. I would also like to thank Ed Desautels and Larry Travis for their critical reading and constructive suggestions during the preparation of this thesis.

Thanks go also to Dick Venezky for all of our conversations about database management, which created my interest in this area, and to Jim Gish and Rich LeBlanc for the many discussions of SIMULA.

A very special thanks goes to the Madison Academic Computing Center. Without their financial assistance this research could never have been completed. The opportunity to work with both DMS 1100 and SIMULA at MACC had a very strong effect upon the direction of this research.

But most of all I would like to thank my wife, Linda. Her encouragement, support and willingness to assume extra burdens at home made this work possible. Her patience, help and understanding during the past 4 years were crucial to completing this work. And her proofreading was invaluable.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGMENTS iv

TABLE OF CONTENTS v

TABLE OF FIGURES viii

CHAPTER 1
INTRODUCTION 1

1.1 The Data Model 3
1.1.1 The Schema 6
1.1.2 The Subschema 6
1.1.3 The Data Manipulation Routines 7
1.1.3.1 The Access Methods 8
1.1.4 The Data Manipulation Language 9
1.2 The Database Management Environment 10
1.3 Current Research on Abstraction Techniques and Programming Methodology 11
1.4 This Dissertation 13
1.4.1 The Approach 14
1.4.2 The Results 15
1.5 An Overview 16

CHAPTER 2
A GENERIC PROCEDURE MODEL OF DATABASE MANAGEMENT SYSTEM ARCHITECTURES 18

2.1 The Network Data Model 20
2.1.1 The Schema 21
2.1.2 The Subschema 29
2.2 Generic Procedures 30
2.3 Analysis of Current Systems 33
2.4 A Generic Procedure Model of Database Management Systems 46

CHAPTER 3
REPRESENTING THE SCHEMA AND SUBSCHEMA AS A SHARED COLLECTION OF ABSTRACT DATA TYPES 50

3.1 Data Independence 52
3.2 Abstraction 56
3.3 The Schema and Subschema as Shared Abstract Data Types 62
3.3.1 An Example Database: The PRESIDENTIAL Schema 63
3.3.2 The Generic Objects 65
3.4 Programming Language Support Required 75

CHAPTER 4
AN IMPLEMENTATION 92

4.1 The User Environment 93
4.2 The Abstract Objects 95
4.2.1 The Mass Storage Record Generalization 97
4.2.2 The UWA_Record Generalization 104
4.2.3 The UWA_SET Generalization 110
4.3 The Data Manipulation Routines 116
4.4 Redefinition of the User Environment in the Subschema 121
4.5 An Implementation 127

CHAPTER 5
A PERFORMANCE ANALYSIS 132

5.1 A Performance Model 139
5.2 The Workload 153
5.2.1 The Many-to-Many Relationship 154
5.2.2 The Cyclic Relationship 163
5.2.3 Initial Loading of the PRESIDENTIAL Database 168
5.3 Conclusions 173
5.3.1 An Evaluation of the Cost of Interpretation 173
5.3.2 The Effect of Multiprogramming 179
5.3.3 Other Performance Costs 190

CHAPTER 6
CONCLUSIONS AND FUTURE RESEARCH 193

6.1 Conclusions 193
6.2 Future Research 197

REFERENCES 200

APPENDIX A 212

APPENDIX B 217

TABLE OF FIGURES

Figure 2.1	The Network Data Model Set	21
Figure 2.2	A Grammar for the Data Definition Language	23
Figure 2.3	A Grammar for the Data Definition Language (continued)	26
Figure 2.4	A Model of Generic Procedures	31
Figure 2.5	Mappings of the View of Data	36
Figure 2.6	The Macro Approach: Complete Compile-time Binding of the Schema and Subschema	38
Figure 2.7	The Library Approach: Compilation of the Access Methods in DMSII	40
Figure 2.8	The Library Approach: Compilation of the User Program in DMSII	41
Figure 2.9	The Interpretive Approach-Phase 1: Compilation of the Schema and Subschema in DMS 1100	42
Figure 2.10	The Interpretive Approach-Phase 2: Compilation of the User Program in DMS 1100	43
Figure 2.11	The Interpretive Approach-Phase 3: Interpretation of the Encoded Schema in DMS 1100	45
Figure 2.12	A Generic Procedure Model of a Database Management System	48
Figure 3.1	The PRESIDENTIAL Schema	64
Figure 3.2	An Example of the MASS_STORAGE_RECORD Generalization	67
Figure 3.3	An Example of the UWA_RECORD Generalization	68

Figure 3.4	An Example of the UWA_SET Generalization	70
Figure 3.5	An Abstract Data Type Model of the Data Manipulation Routines	72
Figure 3.6	Coexisting Implementations of an Abstraction	77
Figure 3.7	Coexisting Implementations of the UWA_RECORD Instances in the PRESIDENTIAL schema	79
Figure 3.8	Binding the VIRTUAL Procedure LOCATE in the Schema	88
Figure 3.9	Binding the VIRTUAL Procedure LOCATE in the Subschema	88
Figure 4.1	The Definition of the User Environment	94
Figure 4.2	The MASS_STORAGE_RECORD Hierarchy	98
Figure 4.3	An Example of the MASS_STORAGE_RECORD Hierarchy	99
Figure 4.4	The PRESIDENT DDL Record Type Declaration	102
Figure 4.5	The M_S_PRESIDENT Declaration	103
Figure 4.6	The UWA_RECORD Generalization	105
Figure 4.7	The UWA_PRESIDENT Declaration	109
Figure 4.8	The UWA_SET Generalization	111
Figure 4.9	The NATIVE_SONS DDL Set Type Declaration	114
Figure 4.10	The UWA_NATIVE_SONS Declaration	115
Figure 4.11	Procedure Calls to the Schema	119
Figure 4.12	Data Manipulation Routine FETCH	120
Figure 4.12	Data Manipulation Routine STORE	121

Figure 4.14	A DDL Declaration of the PRES_ADMIN Subschema	122
Figure 4.15	The PRES_ADMIN Subschema	124
Figure 4.16	The Definiton of the User Environment	125
Figure 4.17	Procedure Calls to the Subschema	127
Figure 4.18	Compilation of the Abstract Objects and the Data Manipulation Routines	128
Figure 4.19	Compilation of the DDL Schema into a Collection of Abstract Data Types	130
Figure 4.20	Translation and Linking of the User Program Containing Data Manipulation Routine Calls	131
Figure 5.1	The Set Structure for the Many-to-Many Relationships	155
Figure 5.2	A Program to Traverse the Many-to-Many Structure	157
Figure 5.3	Traversing the Many-to-Many Structure. Exponential Distribution: System A	161
Figure 5.4	Traversing the Many-to-Many Structure. Exponential Distribution: System B	162
Figure 5.5	An Example of the Cyclic Set Structure in the PRESIDENTIAL Database	163
Figure 5.6	A Program to Traverse the Cyclic Structure	165
Figure 5.7	Traversing the Cyclic Structure. Exponential Distribution: System A	166
Figure 5.8	Traversing the Cyclic Structure. Exponential Distribution: System B	167
Figure 5.9	A Program to Load the PRESIDENTIAL Database	169

Figure 5.10	Loading the PRESIDENTIAL Database. Exponential Distribution: System A	171
Figure 5.11	Loading the PRESIDENTIAL Database. Exponential Distribution: System B	172
Figure 5.12	Schema Descriptor I/O Versus the Size of the Available Buffer Space	181
Figure 5.13	The Relative Utilization as a Function of the Multiprogramming Level (System A: No CPU-I/O Overlap)	184
Figure 5.14	The Relative Utilization as a Function of the Multiprogramming Level (System B: No CPU-I/O Overlap)	185
Figure 5.15	The Relative Utilization as a Function of the Multiprogramming Level (System A: Complete CPU-I/O Overlap)	188
Figure 5.16	The Relative Utilization as a Function of the Multiprogramming Level (System B: Complete CPU-I/O Overlap)	189

CHAPTER 1

INTRODUCTION

A database management system is a system for the storage and retrieval of information from a stored collection of data, the database. The database management system attempts to achieve the following sometimes conflicting goals:

- data sharing
- data integrity
- data protection
- data independence.

The goal of data sharing implies that the database management system supports concurrent access to the database by multiple users. The goal of data integrity means that the database management system attempts to guarantee the validity of data stored in the database by employing such mechanisms as range checks on input data. By contrast with data integrity, data protection means that the

database management system protects the database from inadvertent modification caused by either programming or machine errors and from attempts to access or modify the database by unauthorized users.

The goal of data independence is to an extent related to data sharing but encompasses much more than data sharing. Data independence is an attempt to protect programs accessing the database from requiring modification if the structure of the database is modified. This means that the database management system presents an interface to the user which the database management system attempts to guarantee will not be affected by modification of the physical structure of the database. Thus the user can assume that the database has certain logical and physical structures and that the database management system presents at the user interface a database conforming to that logical and physical structure. This independence is achieved by presenting to the user an abstraction of the database in which the details of actual machine implementation are hidden from the user. In this way the user is isolated from modifications to the actual implementation of the database and hence independent of physical and logical structures actually used to support the database. A complete discussion of data independence is presented in Section 3.1.

1.1 The Data Model

Each user of a database regards the database as the computer representation of the information relevant to his application. The database attempts not only to represent entities in the user's view by their attributes such as values, but also to represent them in terms of their relationships with other entities. The data structures employed to represent these relationships are called the data model. Coupled with a description of the data items within each record, they form the data model definition. Thus the data model definition is a complete description of the information content of the database.

Data models are primarily characterized by the relationships which can be represented between entities. The three major data models in use are:

1. Relational data model
2. Hierarchical data model
3. Network data model.

The relational model represents each entity by its values and its membership in sets, called relations. The relation may be viewed as tabular with rows of the table termed tuples, and data items within the relation are referred to as attributes. There are no explicit links connecting the tuples. Instead, relationships are implicitly

stated by storing within a relation identifiers which link it to other relations or by creating relations whose domains are the key values for other relations.

The hierarchical data model and the network data model are both graph-oriented data models. The hierarchical model is restricted to tree-structured relationships. Information is represented in the database by storing the values associated with entities in the nodes of the tree and by representing relationships between entities by the arcs between nodes. The hierarchical model can thus represent 1-to-many relationships between a single entity, termed the owner, and n other entities, termed the members. The tree structure constrains a given member entity to have only a single owner entity, and no member entity at a given level of the tree may own an entity at a higher level of the tree.

In the network model, an entity is represented by storing the values associated with it as the nodes, which are termed records, of a graph and by representing the relationships between entities by the arcs of the graph. The basic structure implemented is the set, with a single entity serving as the set owner. A set occurrence contains zero or more entities as the members. The use of a general graph rather than a tree removes the constraint that no member entity may be shared by multiple owner en-

tities and the constraint that a lower member may not be the owner of some entity higher in the graph. Thus the network model can represent many-to-many relationships between entities.

A committee of the American National Standards Institute, X3/SPARC, has been studying database management systems in an attempt to develop standards [ANS75]. They have defined three levels of interfaces to database management systems: the conceptual level, the internal level, and the external level.

The conceptual level represents the community view of the database. This view is defined in a conceptual schema which represents the entire database model. The conceptual schema corresponds to the network model schema which is described in Section 1.1.1.

The internal level is the view of the physical storage structures of the database. An internal schema defines the physical formats of database attributes and defines where and how they are stored and accessed. The internal level corresponds to the network model Device Media Control Language. The Device Media Control Language is not examined in this dissertation.

A user of the database management system views data from the database as represented in a format close to the needs of his application. This representation, or exter-

nal view, is of a subset of the database and is defined by an external schema. The external schema corresponds to the network model subschema which is described in Section 1.1.2.

1.1.1 The Schema

The data model defines a collection of logical structures which are available to the users of a database management system. In order to utilize this structure for applications, the structure of the actual database must be defined in terms of the structures in the data model. The definition of the database is termed the schema, or data model definition. The schema defines the types of entities which may exist within the database and also defines the relationships which may exist between these entities. The schema thus is a definition of the logical structure of a database and is shared by all users of the database.

1.1.2 The Subschema

A database is the integrated collection of data from an organization. Users of a database management system are typically interested in only the subset of the database which is relevant to their application. For example,

a user in the registrar's office of a university is not interested in and perhaps should be denied access to information about faculty members' salaries, but rather is interested in the courses taught by faculty members. The subschema, or data submodel definition, is the definition of the subset of the database to be used by an application. The subschema is thus a restriction of the total schema, which defines the structure of the entire database, and represents a definition of an application-dependent window into the database.

The subschema defines the correspondence between the schema and the application by defining the names of entities to be accessed by the user and the relationships between these entities which are relevant to the user's application.

1.1.3 The Data Manipulation Routines

The data manipulation routines handle all accesses to the database. A request from a user to access the database is in terms of a procedure call to a data manipulation routine. This procedure uses the schema and subschema to determine the necessary operations to be performed on the database.

1.1.3.1 The Access Methods

The access methods communicate with the database in units of physical mass storage blocks or pages and communicate with the data manipulation routines in units of physical records. The access methods isolate the database management system from storage-dependent functions such as disc scheduling policies, etc. The access methods also provide physical data independence for the database management system by isolating the database management system from physical data structures employed in storing the database and from the indexing paths which are used to efficiently access physical records in the database.

As previously discussed, the data manipulation routines communicate with the access methods in terms of physical records. The database management system transforms these physical records into logical records which are the communication level with the user. Transforming a physical record into a logical record implies operations such as removing pointer fields which should not be visible to the user. It also includes reformatting data items which are stored in a compressed form and performing necessary decoding and type conversions. More importantly, a logical record may actually be composed of pieces of sev-

eral physical records or may be a computed value derived from several physical records.

The user requests retrieval of logical records by the database management system and identifies the desired record either in terms of a unique identifier or a Boolean qualification. It is the function of the database management system to transform this identifier or Boolean qualification into requests to the access methods for retrieval of the necessary physical records needed to satisfy the user query.

1.1.4 The Data Manipulation Language

The user interacts with the database management system either through a procedural programming language or through a non-procedural query language. The user language is termed a data manipulation language and in general supports the following operations:

1. Retrieval of data stored in the database
2. Addition of new data to the database
3. Modification of data currently stored in the database
4. Deletion of data from the database.

1.2 The Database Management Environment

A database management system provides facilities for the user to create complex data structures, to establish relationships between these structures, and to perform operations on these structures. Thus a database management system is a large complex software system. In contrast to the environment of sequential programs, three features distinguish the environment of a database management system.

Probably the most important of these features is that concurrent access to the database by multiple users must be provided. A database is a shared resource and it is the responsibility of the database management system to control access to the database in order to guarantee the correctness of modifications to the database.

The database can be regarded as global to all programs which access it. Not only is the data stored in the database global to all user programs, but the declaration of the database structure and relationships which may exist between these structures are also global to all user programs. The data types of individual items within the database are shared. The definition of data structures, procedures, and relationships which may exist between the data are also shared. The database and its accompanying

declaration can thus be regarded as an ALGOL-like block which encompasses all instances of user programs which access the database.

The individual program shares the data in the database and its description. However, instead of regarding the database as local to the address space of the user program, the user program views the database as residing in a separate address space. The user program must execute explicit primitives "store" and "retrieve" to move data between the database and the user's local address space.

1.3 Current Research on Abstraction Techniques and Programming Methodology

As discussed earlier, a database management system represents a large complex software system. Research in programming methodology and programming languages is an active area of computer science. A number of developments in this area are relevant to the design of database management systems. One of these is the development of tools to support abstraction. An abstraction is a description of a system component which does not completely describe all the details of its implementation.

Three major abstractions supported by high-level languages are procedural abstractions, data abstractions, and control abstractions [Hoa72] [SWL77]. Procedural abstraction is supported in almost all programming languages, for example, the FORTRAN SUBROUTINE [ANS66] and the ALGOL PROCEDURE [Nau63]. Data abstractions are supported in terms of aggregate objects such as records in PASCAL [JW75]. Support for control abstractions is relatively new and is provided in terms of constructs such as the ALPHARD iterator.

An important extension of data abstractions is the capability to bind procedures and data abstractions together in a shared environment to create abstract data types. Languages that support user definition of abstract data types include: the class of SIMULA 67 [DMN70], the cluster in CLU [LSAS77], the form in ALPHARD [SWL77], and modules in EUCLID [LHLMP77], MODULA [Wir77], and MESA [GMS77], etc. All of these language constructs provide the capability to encapsulate an environment consisting of data structures and procedures which access these data structures.

An abstract data type is a description of a system component which does not specify all the details of its implementation, but gives a global view of its properties through the operations which are available as the proce-

dures of the abstract data type. Binding together data structures with procedures thus allows extension of languages with the definition of new types which are characterized not by the implementation of the data structures, but by the operations that are performed on it.

1.4 This Dissertation

Recent investigations in the area of programming methodology have examined the role of programming languages in writing complex software systems (see Section 1.3 and Section 3). The objective of this dissertation is to study the use of high-level programming languages which support abstract data types as a tool for the implementation of database management systems. In particular, the design and performance of a network model database management system will be examined.

The major features which must be supported by this network database management system are summarized as follows:

- Representation of the complex objects within the database
- Sharing the schema and the subschema among user programs
- Controlling access to the schema through the subschema
- Controlling concurrent access to the database
- Providing facilities for protection, security, recovery.

1.4.1 The Approach

A model of database management systems is developed in terms of four components of the database management system. The network data model is represented by a collection of abstract data types (Section 4.1). The data manipulation routines are defined as a collection of generic procedures which are defined completely in terms of the attributes of the abstract objects (Section 4.2). The schema (Section 4.3) and the subschema (Section 4.4) are then compiled to form instantiations of the abstract ob-

jects in the network data model. In this approach the schema and subschema represent abstract data types which are shared by all user programs referencing the database. The user program thus executes in an environment defined by the abstract objects, the data manipulation routines, the schema and the subschema.

Data independence is examined in terms of the ability to control access rights to the shared schema and subschema. The data manipulation routines are shown to have different access rights to the schema and subschema than those provided to the user program. Data independence is described in terms of the access rights control supported by programming languages.

1.4.2 The Results

There are several major advantages to representing the schema and subschema as shared abstract objects. First, by using abstract data types to structure the design of a database management system, the resulting software should be more reliable [ICRS75] [Kos76] [Lin76a] [Wor77]. By using a programming language which supports specification and verification of abstract data types, we can verify data independence at compile-time. In addition, it is shown in this dissertation that abstract data

types and programming language support for environment control allow the programming language to support data independence and subschema functions directly.

Another significant advantage of our approach will be an improvement in the performance of a database management system which is implemented using abstract data types. This is possible because application of abstract data types to a database management system permits the elimination of the time-consuming run-time interpretation of the schema and subschema used in systems such as IBM's IMS [McG77] and UNIVAC's DMS 1100 [SPE75a] [SPE75b]. Instead, data abstractions which represent the logical structure of the database are bound at load-time to the user's program. In addition, the data manipulation routine calls included in the user's source program are implemented by parameterized calls to the procedures bound to the abstract data types which were used to represent the logical structure of the database. In this way we can avoid run-time interpretation of the schema and subschema without suffering any loss of data independence.

1.5 An Overview

The schema and subschema perform two different functions within a database management system. The first is

to describe the actual parameters to the data manipulation routines. The second is to define the environment for data manipulation language programs. Chapter 2 introduces a generic procedure model of the data manipulation routines to describe binding the actual parameter descriptors to the data manipulation routines. In Chapter 3 this model is modified to represent the schema and subschema as shared abstract data types. An implementation of a network data model system using this abstract data type model is presented in Chapter 4. Chapter 5 analyzes the performance enhancement provided by utilizing compile-time binding rather than run-time binding through interpretation.

CHAPTER 2

A GENERIC PROCEDURE MODEL OF DATABASE MANAGEMENT SYSTEM ARCHITECTURES

The aim of this dissertation is to examine the architecture of database management systems in the context of current research on programming methodologies. In particular, programming language support for extended types, generic procedures and protection mechanisms will be examined.

Three data models are currently in widespread use as follows: the relational [Cod70] [SWKH76] [Ast76], the hierarchical [McG77], and the network model [COD71] [COD73]. This dissertation examines in detail the design of a network database management system.

As a consequence of the many existing implementations of the network model and the theoretical advantages of the relational model in terms of data independence, many papers have been published comparing the network model with

the relational model [MMC76] [Oll75]. This discussion of the relative merits of the two data models has clarified that the relational model and the network model present two very distinct levels of abstraction to the user. The user of a network-based system must be aware of details of the access methods which the relational user lets the system manage.

To avoid the problems inherent in contrasting the different levels of abstraction of the two models, several authors have compared the two models in terms of their similarities and the features that they both provide to the user [Oll75] [SH75] [MMC76]. In parallel with these research efforts the ANSI committee, ANSI/SPARC described earlier, was formed to study the standardization of database management systems. Based on the proposals of ANSI/SPARC, Date [Dat76] and Tsichritzis [Tsi75] [Tsi76] have independently developed programming languages to implement database management systems which support the coexistence of all three data models. The languages are based on a system data model very similar to the network model. Since the network model can be used to implement the relational model and the hierarchical model, solving the problems of implementation for the network data model will result in a solution which will be generalizable to include all three data models.

2.1 The Network Data Model

To analyze the architecture of database management systems it is essential to understand the function of each system component. In Section 1.1-1.5 the major components of a database management system were introduced as follows: the data model, the schema, the subschema, the data manipulation routines, and the data manipulation language. The function of each of these components will now be examined in detail.

The data model defines the class of entities and relationships which may exist within a database. The basic entity in the network data model is the <u>record</u>. A record is the basic unit of transfer between the user and the database. A record is similar to the records of PASCAL and COBOL and the structures of PL/1 and is composed of a number of related data items. A data item is the smallest nameable unit within the databse.

Relationships between record types may be named and are defined in terms of <u>sets</u>. A set is a named relationship and represents a logical relationship between one record type, the owner type, and another record type, the member type. The user of a database management system can view a set as being a multiply-connected list structure as shown in Figure 2.1. Other structures such as pointer ar-

rays and B-trees may be used to implement the set. However, the techniques described in this dissertation are also applicable to these structures and we will only discuss the structure shown below.

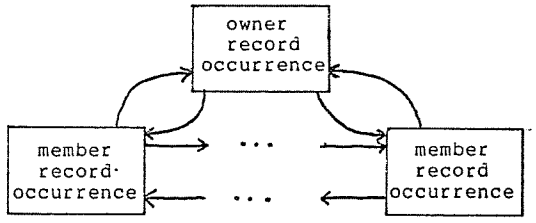


Figure 2.1 The Network Data Model Set

User queries to the database are stated in terms of commands to traverse this logical structure by moving from one member to the next or previous member, from an owner to a member, and from a member to its owner.

2.1.1 The Schema

The network data model schema is the definition of a database and is expressed in terms of the structures of the network data model. The schema contains the declaration of record types and set types which may exist in the

database. The schema describes records and sets in terms of data characteristics such as data items with records. It also includes the declaration of an access and storage strategy to be associated with each record type along with the definition of logical subdivisions of the database, or areas, in which each record type is to be stored. The CODASYL DBTG Report describes in detail a language, the Data Definition Language (DDL), to be used to describe the logical structures and storage structure of the database. The DDL provides the capability to describe the database in terms of logical subdivisions, or areas, records, and sets. A grammar (BNF) for the DDL is presented in Figure 2.2 and Figure 2.3.

Two aspects of the CODASYL DBTG Report are not considered in this dissertation. The first is the DDL descriptions of the privacy control and integrity constraints which the data manipulation routines are to enforce on the database. The second aspect is the Device Media Control Language (DMCL) which is described in the report and which is a tool to specify the mapping of the logical database structure onto the physical storage media. While these features of the report are not considered here, we believe that our approach could incorporate them without difficulty.

```
<SCHEMA>              ::=<SCHEMA SECTION><AREA SECTION>
                         <RECORD SECTION><SET SECTION>
<SCHEMA SECTION>      ::=SCHEMA NAME IS ID
<AREA SECTION>        ::="Not Implemented"
<RECORD SECTION>      ::=RECORD SECTION <RECORD DECL LIST>
<SET SECTION>         ::=SET SECTION <SET DECL LIST>
<RECORD DECL LIST>    ::=<RECORD DECL>
<RECORD DECL LIST>    ::=<RECORD DECL LIST><RECORD DECL>
<RECORD DECL>         ::= <RECORD NAME><RECORD CODE>
                          <LOCATION MODE>
                          <DUP> <WITHIN> <ITEM LIST>
<RECORD NAME>         ::= RECORD NAME IS ID
<RECORD CODE>         ::= RECORD CODE IS INTNUM
<LOCATION MODE>       ::= DIRECT
<LOCATION MODE>       ::= CALC ID USING ID
<LOCATION MODE>       ::= VIA ID SET
<DUP>                 ::= DUPLICATES ARE NOT ALLOWED
<DUP>                 ::= DUPLICATES ARE ALLOWED
<WITHIN>              ::= WITHIN ID
<ITEM LIST>           ::= <TYPE> ID EOL
<TYPE>                ::= INTEGER
<TYPE>                ::= REAL
<TYPE>                ::= TEXT
<TYPE>                ::= BOOLEAN
```

Figure 2.2 A Grammar for the Data Definition Language

Rather than being a data structure similar to PASCAL records, a DDL record declaration has several facets as shown in the BNF in Figure 2.2. A declaration of each item comprising the record and its type is provided by the <ITEM LIST>. Bound to this data structure is the <LOCATION MODE> clause which describes both a storage allocation structure and an access mechanism for the record.

Among the <LOCATION MODE> options available are DIRECT, CALC, and VIA a set type. The DIRECT declaration specifies that storage for record occurrences is to be allocated by the DBMS using a system procedure. Subsequently the record occurrences may only be accessed by using a pointer variable whose value is an address in the database. The CALC option specifies that a hashing function or an index is created to be used in allocating storage for record occurrences and that subsequently the record may be retrieved by providing a value of its key. The VIA set option means that a record is stored "close to" its owner, and that subsequently it can be located only through a linkage established through its membership in the specified set. The <RECORD DECL> also contains integrity information in the form of the <DUPLICATES> clause which specifies whether duplicate key values are permitted. This checking is to be performed automatically by the database management system. The <WITHIN> clause indicates in which logical subdivisions of the database this record type may be stored. This information is procedural; that is, it informs the data manipulation routines how to store and retrieve instances of the record types.

The <SET DECL> in the DDL (Figure 2.3) is used to declare the relationships which exist in a database. These structures consist of directed graphs linking records. A

set type is a named relationship between record types with one record type serving as the set owner type and one or more record types serving as the set members.


```
<SET DECL LIST>     ::=<SET DECL>
<SET DECL LIST>     ::=<SET DECL LIST><SET DECL>
<SET DEC>           ::= <SET NAME> <SET CODE> <MODE>
                        <ORDER> <OWNER> <MEMBER>
<SET NAME>          ::= SET NAME IS ID
<SET CODE>          ::= SET CODE IS INTNUM
<MODE>              ::= MODE IS CHAIN
<MODE>              ::= MODE IS POINTER ARRAY
<ORDER>             ::= ORDER IS <OC> <SWITHIN>
<OC>                ::= FIRST
<OC>                ::= LAST
<OC>                ::= NEXT
<OC>                ::= PRIOR
<OC>                ::= SORTED
<SWITHIN>           ::= WITHIN ID BY ID
<OWNER>             ::= OWNER IS ID
<MEMBER>            ::= MEMBER IS <CLASS CODE> <LINKED>
                        <MOD> <SOS>
<MOD>               ::= EPSILON
<MOD>               ::= <MORDER> <MDUP>
<MORDER>            ::= <UPDOWN> KEY IS ID
<MDUP>              ::= DUPLICATES ARE FIRST
<MDUP>              ::= DUPLICATES ARE LAST
<MDUP>              ::= DUPLICATES ARE NOT ALLOWED
<MDUP>              ::= DUPLICATES ARE SYSTEM ORDERED
<SOS>                   ::=SET OCCURRENCE SELECTION
                        IS THRU <SOSMODE>
<SOSMODE>           ::= CURRENT OF SET
<SOSMODE>           ::= LOCATION MODE OF OWNER
<SOSMODE>           ::= ID USING CURRENT OF SET
<SOSMODE>           ::= ID USING LOCATION MODE OF OWNER
<UPDOWN>            ::= ASCENDING
<UPDOWN>            ::= DESCENDING
```

Figure 2.3 A Grammar for the Data Definition Language (continued)

The set is the fundamental structure of a CODASYL database and serves a number of different functions. The first function is to serve as a building block for data

structures in the database. In this role the set performs like a mathematical function. The declaration of set owner and member record type establishes a domain and range of the set in terms of record types. Given an owner record occurrence, it is possible to map to a member record occurrence. This mapping is performed using set occurrences to provide an access path to record occurrences in the database. The user views a set owner record occurrence and the set member record occurrences as linked by logical pointers. (Since system-owned sets are easily modeled, they are not considered in this dissertation.)

A CODASYL DBTG system supports the concept of a current position within a set and a current occurrence of each record type. The currency values are represented as pointers into the database. Using these currency values, the user can traverse the pointers linking record occurrences along three different paths. Given a pointer to a set owner record occurrence, it is possible to navigate through the set occurrence to locate specific member record occurrences. It is also possible to move from any member record occurrences to either another specified member record occurrence or to the set owner record occurrence. Any choice of an underlying set implementation must support all of these access paths. The <SET DECL> also specifies a form of integrity information in the

<CLASS CODE> clause. Like the <LOCATION MODE> clause in the <RECORD DECL>, the <CLASS CODE> specifies procedural information. The function of this clause is to specify whether the member record occurrences may exist in the database without being a member of the set. Record occurrences which are AUTOMATIC will automatically be inserted into an occurrence of the set type when they are stored in the database by calling a data manipulation routine. MANUAL members, however, must be explicitly inserted into the proper set occurrence by the user.

The set owner record type and member record type declared in the schema serve to establish a general relationship between record types. Specific instances of relationships are dependent upon the values of the data items in the record occurrences. The data dependence of these relationships is specified in the <SOS> and <SORDER> clauses of the set declaration. The <SOS> specifies the procedure to be used by the data manipulation routines to locate the proper owner record occurrence in order to insert a new record occurrence into the set occurrence. This selection criterion is a function of data values such as the calc key for the set owner or is expressed in terms of the currency information. A relationship which is also data dependent exists among the member record occurrences. This dependence is expressed by the <SORDER> clause in the

set declaration. The <SORDER> specifies that member record occurrences may be unordered, may be ordered using one of the items of the record, or may be ordered in terms of the time of insertion into the set occurrence. Thus the schema declares the procedure to be used by the data manipulation routines to insert a new member into a set occurrence.

2.1.2 The Subschema

The schema is used to describe the entire logical content of the database. The schema is shared by all users who access the database and is language independent. Due to the support of various constructs in programming languages or to restrict access to various parts of the database, the user does not directly access the database through the schema.

Instead the user utilizes a subschema, or mapping function, which describes the portion of the database structure described in the schema which is to be accessed in the user's application [Dat75]. The subschema provides for the removal of areas, set types, record types, or items within a record type from the user's view of the database. Any area, set type, record type or item may be renamed in the subschema, and a different type may be associated with an item in the record types visible to the user. In addition, a different set selection criterion may be associated with the set types present in the user's view.

2.2 Generic Procedures

The actual parameters to the data manipulation routines described in Chapter 1 are of the generic types "record" and "set". The data manipulation routines utilize the information in the schema and subschema to describe the actual parameters in order to determine the function to be performed. Thus the data manipulation routines are an example of generic procedures.

Gries and Gehani [GG77] defined generic procedure to be a "procedure operating on a parameter of any data type for which certain basic operations have been defined." A generic procedure is a procedure which accepts arguments of different types and which can perform different computations based on the type of the actual parameter. The plus (+) operation in ALGOL 60 [Nau63] represents such a procedure since it accepts operands of types integer, real, complex,etc. The plus operation performs different computations depending on the type of the operand. The

types of the operands can be regarded as implicit parameters to the operator.

A model of generic procedures is shown in Figure 2.6. The data descriptor contains a description of the characteristics of all valid actual parameter types. The descriptor of the actual parameter characteristics may be bound to the generic procedure either at compile-time or at run-time. If the binding occurs at compile-time, then the descriptor is the type information from the compiler symbol table. If the binding is performed at run-time, some form of interpretation is used.

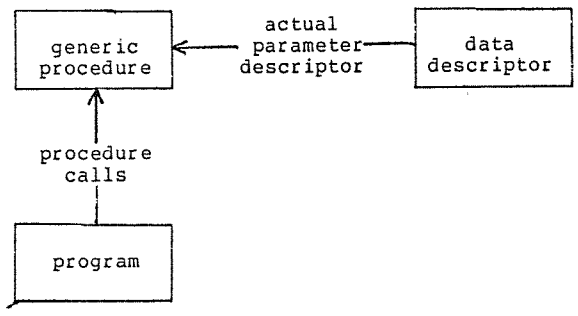


Figure 2.4 A Model of Generic Procedures

Gries and Gehani studied the concept of generic procedures and their relationship to abstraction. Abstraction allows the classification of several different types into a single "generic" class for which a set of common characteristics and parameters is identifiable. Thus procedures may be written which specify formal parameters which are members of the generic class. Due to the differences in implementation which are possible for the different types associated with the generic class, the computation performed by the procedure may vary, depending upon the type of the actual parameter. A mechanism is suggested which supports the declaration of the procedure in terms of its generic specifications, i.e., its interface to all objects in the generic class. This interface specification is used for verifying the correct usage of the procedure. In addition to the definition of the generic interface, an instance of the procedure implementation must be supplied for each type of actual parameter. The procedure body instances are expanded at compile-time as macros which depend upon the type of the actual parameters to the procedure. A major limitation of the work by Gries and Gehani is that they only consider language-defined types, such as PASCAL scalar types, subrange types, array types, and record types with variants.

Generic procedures were also examined by Wegbreit in EL1 [Weg74]. EL1 represents type information as a MODE which is associated with each variable and which is testable at run-time. A procedure may perform operations on its arguments depending upon the run-time value of the argument's MODE.

2.3 Analysis of Current Systems

One of the major design decisions in developing a database management system is the determination of when the data manipulation routines bind the data descriptor from the schema and subschema to their actual parameters. In general, the longer binding can be delayed, the easier data independence is to support.

Wiederhold discusses the alternatives of binding at run-time and of binding at compile-time [Wie77]. In the first approach the schema and subschema are encoded into an internal form, referred to as the object schema and the object subschema. Using the record types and set types referenced as actual parameters in a data manipulation routine call, the object schema and object subschema are accessed to retrieve the record type and set type descriptors. These descriptors are then interpreted to perform the data manipulation language command.

The advantage of this approach is the high degree of flexibility provided by interpretation. Data-oriented modifications to the schema do not require programs utilizing the database to be recompiled. However, this advantage is not as significant as it first seems. Modifications to the schema will in general necessitate restructuring of the database. The effort required to recompile programs utilizing the database is insignificant in comparision to the effort required to restructure the entire database.

The interpretive approach has some significant disadvantages, however. Interpretation will require increased processing time to interpret the object schema and the object subschema. If the object schema and subschema are stored on secondary storage to facilitate their being shared by all programs accessing the database, then increased I/O cost will be incurred in addition to the increased processing cost. Coupled with the use of generic procedures is the database management system environment which emphasizes concurrency and real-time response. Few programming methodologies support compile-time analysis and debugging of such programs. Execution-time debugging in such an environment is very difficult.

In the alternative approach, each of the user's data manipulation routine calls is compiled into a directly ex-

ecutable form by binding the schema and subschema to the user program at compile-time. The traditional advantages of compilation are increased execution efficiency and the facility to utilize multiple programming languages and library routines. An apparent advantage in the database management system application is that no explicit space is required for the object schema and the object subschema. This advantage is perhaps illusory, since in a compiled approach the information from the schema is still present but is now distributed in the code, rather than being isolated into an encoded representation. However, the generated code is optimized for space since it only contains procedures explicitly referenced in the schema.

These advantages are offset by several significant disadvantages. The first is that multiple users imply multiple object modules. Each of these object modules contains code to manage concurrent access to the database. This decentralization of concurrency control makes control of concurrent access more difficult than if a single resource manager were controlling access to the database.

Another disadvantage of this approach is its potential for loss of data independence. Any modification of the schema or subschema will require all object modules using the database to be regenerated. More importantly, after combining the schema and subschema with the user

program it is difficult to guarantee that the user program is isolated from knowledge of the access methods declared in the schema and subschema. Coupled with these problems is the fact that few programming languages have the capability to define structures as complex as those that exist within a database management system.

Larson [Lar78] extends this analysis of when binding occurs by considering both <u>when</u> and <u>how</u> the schema information is bound to the data manipulation routines. The mappings, or transformations of the representations of information, which occur in the development and execution of a user program are shown in Figure 2.5. The solid lines represent mappings and the dashed lines represent algorithm-data interactions.

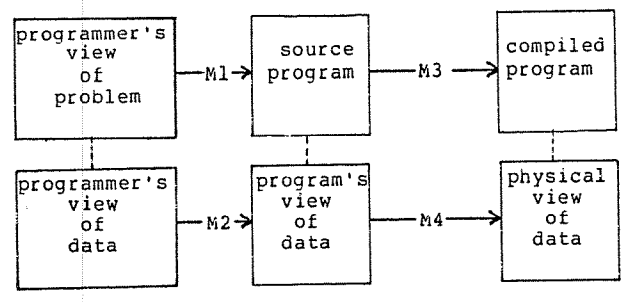


Figure 2.5 Mappings of the View of Data

The mapping M1 occurs during the development of an algorithm to solve the user's problem. M2 is the process of compiling a source program into executable form.

M2 and M4 are very important in the database environment. M2 represents the process of mapping the real-world structures and relationships of the user's application onto the data structures and relationships supported by the data model. M2 is formalized in the schema and subschema which define the mapping of the user's view of the database, which is in a form relevant to his application, onto the data model.

M4 represents the binding of the description of the database structure described in the schema and subschema to the data manipulation routines. Larson describes four techniques for <u>how</u> and <u>when</u> this mapping occurs.

The first approach is the <u>macro approach</u> and is essentially the compiled approach described above. The compiler translates the user's data manipulation routine calls directly into executable code by using information in the source schema and subschema. Using techniques similar to macro expansion, the compiler replaces the user's data manipulation routine call with code to perform the desired operation. The approach is shown in Figure 2.7.

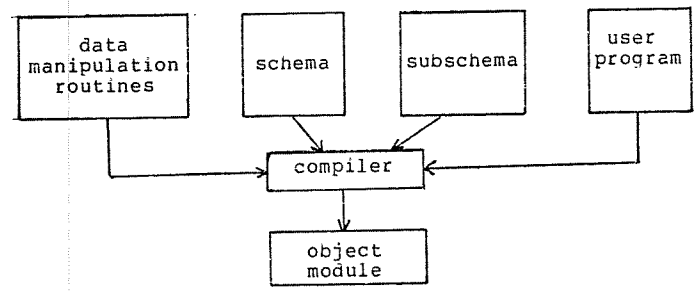


Figure 2.6 The Macro Approach: Complete Compile-Time Binding of the Schema and Subschema

The resulting object module binds all information about access strategies declared in the schema and subschema to the data manipulation routine calls. The object module can thus directly access the database at run-time. As described above, the advantage of this approach is run-time efficiency. This disadvantage is offset by the loss of data independence and the decentralization of concurrency control.

The second approach is the <u>library approach</u>. Rather than binding all information from the schema and subschema to the user program at compile-time, data independence suggests that this binding be delayed as long as possible. In library approach binding of the schema and subschema to

the data manipulation routines occurs partially at compile-time and partially at run-time. The user's data manipulation routine calls are translated into calls to a library of procedures. This library is then bound to the user program at load-time. This approach is similar to an I/O library in which language operations such as READ, GET, PUT, and WRITE are translated into calls to routines in an I/O library. Parameters required by the library procedures, such as file name, record length, block length, device on which the file is located, etc., are supplied from the user program, from the job control language, or from file control blocks.

We believe that a mechanism which is similar to the library approach is utilized in Burroughs' DMSII [BUR75]. DMSII uses two phases of compilation as shown in Figure 2.7 and Figure 2.8. During the first phase the data manipulation routines, the schema, and the subschema are compiled to produce an object module which is a collection of procedures, the access methods. In the second phase, the user program is compiled. The schema and subschema are accessed during compilation to perform type checking for the entities declared in the schema and subschema. Binding of the user environment is completed at load-time by binding the access methods to the user program. By separating the access methods from the user program, it is

possible to modify the access methods without affecting the user programm.

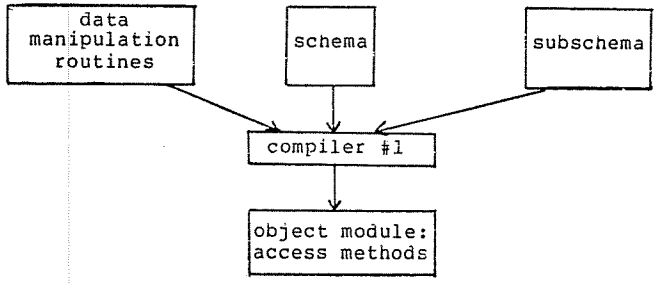


Figure 2.7 The Library Approach: Compilation of the Access Methods in DMSII

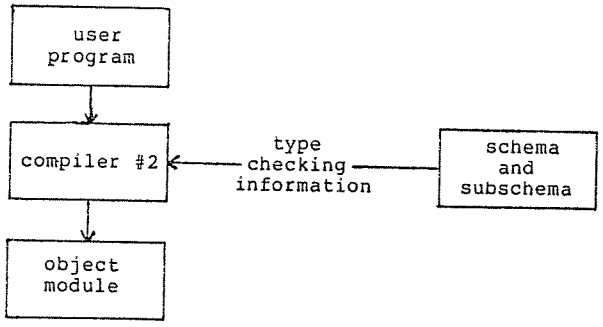


Figure 2.8 The Library Approach: Compilation of the User Program in DMSII

The most frequently used implementation technique is to postpone binding the schema and subschema to the data manipulation routines until run-time. This is the interpretive approach. In DMS 1100 [SPE75a] [SPE75b], which we believe is a typical example of this approach, three phases of binding are used. The first phase translates the source schema into a form suitable for interpretation at run-time.

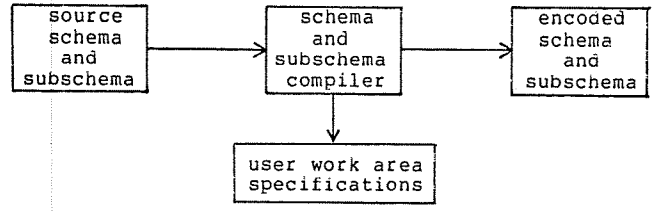


Figure 2.9 The Interpretive Approach-Phase 1: Compilation of the Schema and Subschema in DMS 1100

A secondary output of the first phase is a set of specifications for a user work area which will serve as a communications buffer between the user program and the database.

The user program is then compiled as follows:

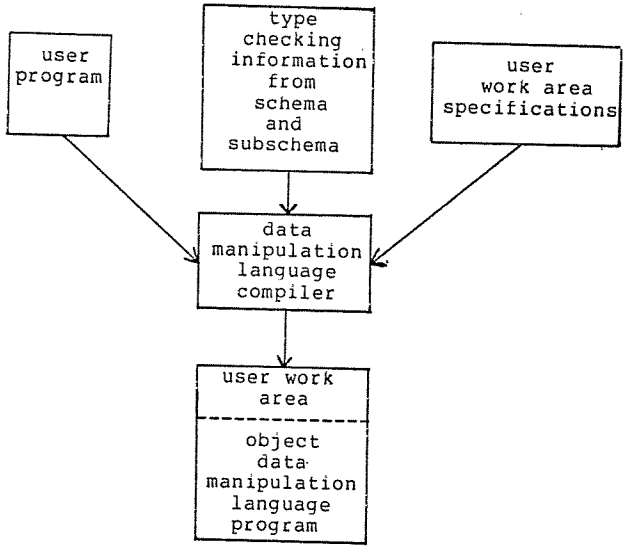


Figure 2.10 The Interpretive Approach-Phase 2: Compilation of the User Program in DMS 1100

Binding of the access methods specified in the schema and subschema occurs at run-time. Each call from the user program to a data manipulation routine specifies one or more actual parameters. For each actual parameter the corresponding descriptor is located in the encoded schema.

This descriptor is interpreted to determine the function to be performed by the data manipulation routine.

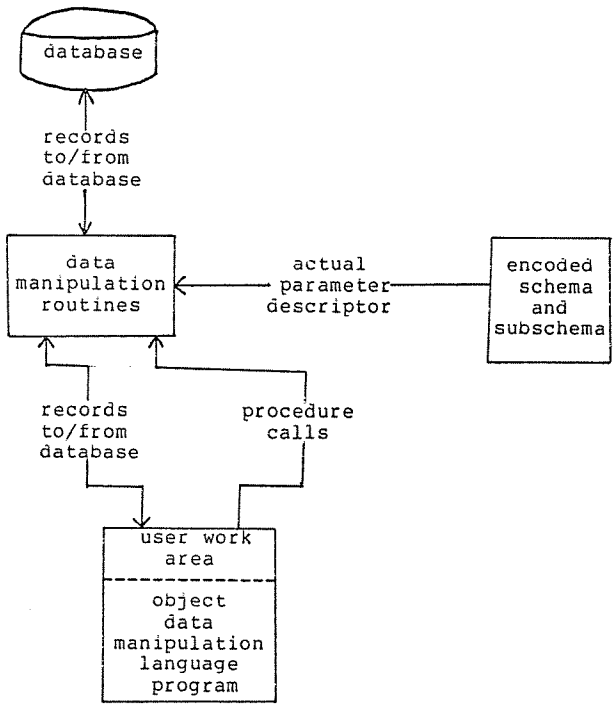


Figure 2.11 The Interpretive Approach-Phase 3: Interpretation of the Encoded Schema in DMS 1100

The fourth approach is the automatically generated procedure approach. The data manipulation routine calls

are treated as calls to procedures which are generated at the time that the database is opened. This approach is similar to a host computer generating code to be executed by a slave computer, such as a channel when performing the I/O operation. In the general database case, the host and slave computer are the same computer. This approach is in the development stage. An example of a system similar to this approach is described by Stemple [Ste76].

2.4 A Generic Procedure Model of Database Management Systems

Hoare [Hoa71] discussed the concept of a discriminated union to describe a set whose members may be of different types. Associated with each member is a tag which can be used to determine the type of the member. A database can be regarded as a collection of record types with variants, or a discriminated union of records. In this context the data manipulation routines can be regarded as generic procedures. The record tag, or variant indicator, is used to determine the computation to be performed by the procedure. The variant concept of PASCAL must be extended in the database environment, since the schema and subschema associate with a record the specifications of the data items plus the specification of an ac-

cess strategy. The access strategy specifies the procedures to be used to store and to retrieve record instances from the database. The information defining the characteristics of each variant within the database is contained in the schema and the subschema. The data manipulation routines thus utilize the schema and subschema to determine what functions are to be performed for a given actual parameter to the procedure.

By representing the data manipulation routines as generic procedures, we can model a database management system in the following way:

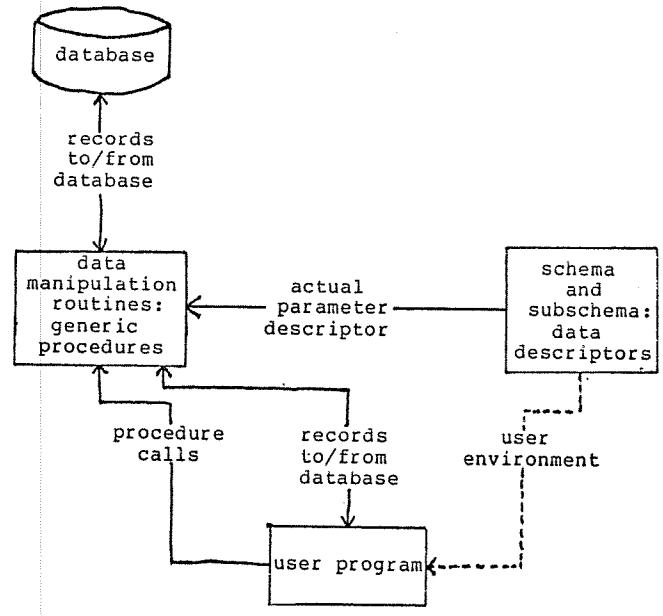


Figure 2.12 A Generic Procedure Model of a Database Management System

This model of a database management system is very powerful. It allows us to analyze current implementation of database managment systems in terms of when binding of the data descriptor occurs. It also provides a convenient framework for unifying research on the application of programming methodologies and of abstraction techniques to the design of database management systems. It also pro-

vides a vehicle for examining data independence in terms of several areas of research in programming languages and operating systems design: abstraction, protection, access control, modularity, and environment binding. Using this model, the study of existing implementations of database management systems becomes a case study of the following features:

1. Time of binding the data descriptor
2. Mechanisms employed to share the data descriptors among all user processes accessing the database through a particular schema and subschema
3. Controlling access to attributes within the data descriptor to enforce or guarantee data independence.

CHAPTER 3

REPRESENTING THE SCHEMA AND SUBSCHEMA AS A SHARED COLLECTION OF ABSTRACT DATA TYPES

As discussed in Chapter 1, two functions of a database management system are to provide data independence and to support sharing the schema and subschema. Since data independence isolates user programs from the details of the database's structural description contained in the schema, we believe that these functions are in conflict. This chapter examines the use of abstraction techniques and their related facilities for access control to resolve this conflict.

As shown earlier, the schema and subschema perform two functions. The first is to provide descriptions of the actual parameters to the data manipulation routines. These descriptors contain the details of the physical and logical implementation of the database. As a result of data independence user programs are isolated from these

details. We feel that data independence is related to the information hiding function of abstraction. Data independence and the uses of abstraction in database management systems are discussed in Section 3.1 and Section 3.2, respectively.

The second function of the schema and subschema is to define the environment for user programs. This environment is the set of associations between identifiers and data objects which may be referenced by a user program. The user environment is composed of the names of the data manipulation routines, the set and record of the network data model, and the names and characteristics of the data items, record types, and set types that are declared in the schema and that are visible through the user's subschema. This environment is shared by all users of a subschema.

As stated earlier, we feel that mechanisms for sharing the schema and subschema are in conflict with the mechanisms required to restrict access to the schema and subschema in order to insure data independence. This conflict can be resolved by using protection mechanisms to provide multiple levels of access rights to the schema and subschema. One level of access rights is the access rights to all details of implementation required by the data manipulation routines. Another level is the environ-

ment definition needed by the user in which the details of implementation are hidden. Section 3.3 examines the use of abstract data types as a mechanism for providing these different levels of access to the schema and subschema and for sharing the schema and subschema. Section 3.4 describes the features which a programming language must provide to support the abstract data type model.

3.1 Data Independence

The goal of data independence has had a very significant impact upon the architecture of current database management systems. Data independence is an attempt to isolate user programs from knowledge of the database structure. Thus the user of a database is able to interact with it through data manipulation routines without knowledge of the database's structure.

The major economic incentive for an organization to use a database management system is data independence since data independence means that no maintenance of user programs is required as the database structure changes. This implies that programs which are developed to utilize a database are isolated from the evolution of the database structure. A database represents the integrated information of an organization and the structure of the database

represents a compromise among the performance needs of all users. As new applications evolve and older applications are terminated, the structure of the database will change to match the new needs of the organization. Without data independence, every change in the database structure could possibly require modifications in all user programs which access the database.

Date [Dat75] defines data independence as the immunity of the application to changes in the storage structure and access strategy of the database. That is, the applications accessing a database do not depend on any one particular storage structure or access strategy and that knowledge of the physical structure and access path is not built into the applications. Date describes three reasons why data independence is necessary. The first is to provide different users of the database different views of the same data. In the simplest sense, it would be impossible to integrate two applications which did not share the same view of the data without data independence.

Suppose the first application wishes to manipulate a variable value as a real while another application wishes to view the same variable value as a character string. Unless the database management system performs type conversions between the stored form of the data and the form required by an application, it would be impossible to in-

tegrate the data of these two applications into a single database without forcing at least one of the applications to be modified.

The second requirement for data independence is that the database represents the integrated data of the organization. The structure of the database represents a performance compromise between the needs of all the applications accessing the database. A change in the performance needs of the applications may necessitate a modification in the database structure. Thus, the structure employed to represent the database must be allowed to evolve independently from applications.

The third major requirement for data independence is the dynamic nature of the database. New applications may be created at any time following the initial development of the database. These applications may require modification of the database to include new information or to restructure the existing information. Data independence isolates existing applications from these modifications to the database structure.

The two levels of data independence which exist within a database management system correspond to the two levels of storage structures within the database system. The first level is physical data independence. A physical structure describes the representation of data as it is

stored within the system. Thus a physical structure is defined in terms of pointers, character sets, numeric representations, record blocking factors, and access methods. Physical data independence is the isolation of user programs from the underlying storage structure. User programs remain unaffected except for performance by changes to the physical structure of the database.

Stonebraker [Sto74] analyzes data independence in an attempt to characterize the transformation to the database which are permitted and which preserve data independence. Data independence isolates the user from the following seven transformations to the database:

1. Physical relocation of stored files
2. Conversion of stored files from one data type to another data type
3. Replacing of one hashing algorithm by another
4. Addition of indexes
5. Duplication of stored data
6. Splitting a record into two separate parts
7. Combining two separate records into a single record.

The second level of data independence is logical data independence. The logical storage structure is the collection of the names of entities, the relationships be-

tween entities, and the operations on these entities that together form the user's view of the database and are defined in the schema. Logical data independence is the isolation of user programs from the logical structure defined in the schema.

Logical data independence is the isolation of user programs from the structure defined in the schema. There are two major aspects to logical data independence [Ber76]. The first is the ability of a database management system to support different views (subschemas) of the the same database (schema). The second is the ability to allow modification to the schema without impairing existing applications. The applications should be isolated from the following modifications to the schema which represent evolutionary changes to the database structure:

1. Change in the type of a data item
2. Addition of new fields to a record
3. Addition of new relationships, or sets, between record types
4. Addition of new record types.

3.2 Abstraction

Data independence is closely related to current work on abstraction techniques in programming methodology. One

of the aims of programming methodology research has been the development of techniques for decomposing a large software system in order to make the design and implementation of such a system intellectually manageable. Abstraction is a major tool in these efforts.

An abstraction is a model of a system component which intentionally omits some details of the implementation. The objective of abstraction is to present an external view of the system component which contains only those details which are relevant to the users of that component.

The major design feature of the THE system [Dij65] was that it was implemented as a set of levels of abstractions, or virtual machines. Each abstraction hides from levels above it the details of implementation which are irrelevant to users of that level. By employing several levels of abstraction, Dijkstra was able to develop an operating system which was both intellectually manageable and which possessed properties about the interactions of components which were provable.

These concepts of abstraction have had a significant impact upon research on database management systems. Data independence is a requirement that user programs be isolated from details concerning the underlying physical and logical structure of the database. Senko, et al. [SAAF73] developed the Data Independent Access Method to

model a database management system utilizing a multi-level model as follows:

1. The Entity Set Model: the user view
2. The String Model: access paths
3. The Encoding Model: mapping access paths into a linear address space
4. The Physical Model: binding the data to storage devices.

Baker and Yeh [BY77] [YB77] extended the concept of a multi-level system by describing a methodology for the design of a database management system using stepwise refinement. The result of applying this methodology is a hierarchy of levels, or virtual machines, which represent functional abstractions. Each level consists of a number of modules, or data abstractions. Associated with each level of the system are performance parameters. The specification of a module includes specifications of its performance in terms of these parameters. As an aid in the system design process these specifications are verified level by level.

In the design of data structures, abstraction arises from the need to recognize the similarity between objects and to concentrate on those properties that are shared by many objects while ignoring the differences between them. The use of abstraction in the development of data struc-

tures was examined by Hoare [Hoa72]. Hoare applied abstraction to sets of objects to create discriminated unions. A discriminated union is a set defined to be the union of two or more previously known sets. Since two sets may have members of different component types, a discriminated union provides a method to distinguish the type of the member by means of a tag associated with each member.

Another form of abstraction is supported in almost all programming languages -- the function or procedure. Procedures allow a problem to be decomposed into a number of units and provide a mechanism for hiding the details of implementation. Thus the user of a procedure is concerned only about the results of the computation performed by the procedure and not about how the computation is implemented.

An abstract data type combines the concepts of data abstraction and procedural abstraction. An abstract data type allows a data structure to be encapsulated with the procedures which access the data structure. Thus the data structure may be defined in terms of the operators on the structure rather than by its implementation technique. For example, a binary tree may be defined in terms of operations to search for an existing element and to insert a

new element. As an other example, a stack may be defined only in terms of the operations push and pop.

A major contribution of using abstract data types is that the responsibility for verifying the security, integrity, and correctnesss is placed on the designer and not on the user. Abstract data types are supported in a number of programming languages: the class in SIMULA 67 [DMN70], the form in ALPHARD [WSL77], the cluster in CLU [LSAS77], the module in MESA [GMS77], the module in MODULA [Wir77], the module in EUCLID [LHLMP77], etc.

Minsky [Min76] utilizes abstract data types to extend and model the file concept used in traditional data processing applications. A file is viewed as an abstract data type. This abstract data types possesses as one of its attributes the record space of the file in secondary storage. It also contains an attribute, the global space, which contains information describing the characteristics and status of the file as a whole and which is small enough to be stored in main memory. Bound to these data attributes are procedures which manipulate the record space and global space. The procedures are comprised of file operators such as OPEN, FIND, etc., and internal procedures which are not visible to users and which perform various functions such as validity checking, access control, and type conversion.

One of the most significant works on abstraction in database management systems is the work of Smith and Smith on relational databases [SS77a] [SS77b]. This work formalizes the use of foreign keys described by Codd [Cod70] as an aggregation. An aggregation is an abstraction of a relationship between objects. Aggregation allows details concerning the objects themselves to be ignored when the relationship is being analyzed. Generalization abstracts the properties of objects within the database. A generalization is an abstraction in which a set of similar objects is regarded as instances of a generic object. That is, the differences between individual entities are ignored and their common properties are identified and used to classify the individual objects as a single, named, generic object. By explicitly naming generic objects, it is possible to identify generic operators for the generic objects, to specify the attributes of generic objects, and to specify the relationships between generic objects. The properties of each generic type may be formalized by a set of invariant properties. These properties should be satisfied by all relations in the database and should remain invariant following operations on the database. The concepts of generalization and abstraction are also used by the Smiths to support different user views of the database.

3.3 The Schema and Subschema as Shared Abstract Data Types

The generic procedure model was used in Chapter 2 to describe the binding of the actual parameter descriptors to the data manipulation routines. In this section abstract data types are analyzed in terms of the generic procedure model as a technique to support sharing the schema and subschema. Abstract data types are used to represent the generic objects of the network data model. The data manipulation routines can now be implemented in terms of the procedures, or operators, associated with these generic objects. The user's schema and subschema are represented as a collection of instantiations of these abstract data types and are bound to the data manipulation routines at load-time. This is an implementation of the automatically generated procedure approach described by Larson. Rather than generating the procedures for each user when the database is OPENed, our approach generates the schema and subschema once when the database schema and subschema are defined. By using the access control features of abstract data types, an environment for user programs is defined by the schema and subschema in which the details of the database structure are hidden. By hiding the details of implementation, this representation of the schema and subschema resolves the conflict between sup-

porting data independence and sharing the schema and subschema.

3.3.1 An Example Database: The PRESIDENTIAL Schema

The examples which follow are based on the PRESIDENTIAL database defined by Fry and Sibley [FS76]. The schema for the PRESIDENTIAL database is shown in Figure 3.1. The name of a record type is shown within each box and the arrows are labeled with the name of a set type. The direction of the arrows is from the owner record type to the member record type. A Data Definition Language declaration of the PRESIDENTIAL schema is presented in Appendix A.

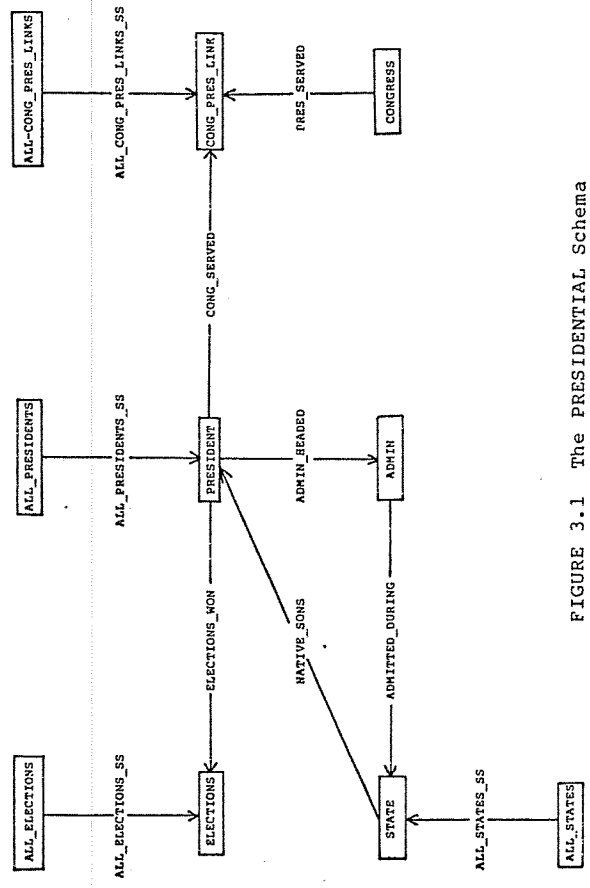


FIGURE 3.1 The PRESIDENTIAL Schema

3.3.2 The Generic Objects

The database is a resource which is shared by all users. Instead of the database being local to a user's address space, the database resides in a separate address space. The user program must execute explicit primitives to move record instances between the database and the user's local address space. A user work area (UWA) serves as a communications buffer for all record instances transferred between the user and the database. As a consequence of this separation of address spaces, there are two different representations, or viewpoints, of the database which the database management system must support.

The first representation is the physical database which is shared by all user programs. Associated with a record instance within the database are the logical pointers which the data manipulation routines use to traverse the database and which are hidden from users. Also associated with each record instance in the database are the encoded forms of the user-visible data items which are defined in the <RECORD DECL> in the schema.

The database can be regarded as a discriminated union of records. The record type and hence the associated properties of each record instance can be determined by testing at run-time the tag associated with the record in-

stance. By analyzing the common properties of records as they appear in the database we can abstract their properties to define a generalization, or abstraction, the MASS_STORAGE_RECORD. This generalization represents an abstraction of record instances in the physical database residing in mass storage. It provides an abstraction of the participation of a record instance in set occurrences in the database. Using this generalization, or generic object, the schema is translated to define abstract data types corresponding to each record instance type. For example, abstract data types M_S_PRESIDENT, M_S STATE, etc. from the PRESIDENTIAL database are defined using the MASS_STORAGE_RECORD as a template. Details of this generaliztion and examples of the definition of the abstract data types representing the PRESIDENTIAL schema are described in Section 4.2. An example of the MASS_STORAGE_RECORD generalization, or abstraction, in the PRESIDENTIAL schema is shown diagramatically in Figure 3.2. The details of the MASS_STORAGE_RECORD generalization are given in Section 4.2.

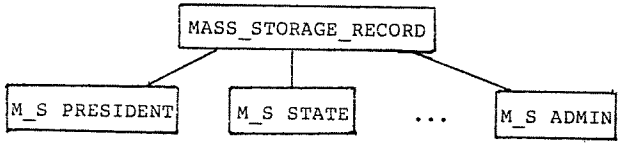


Figure 3.2 An Example of the MASS_STORAGE RECORD Generalization

A second view of the database is that view provided to the user through the User Work Area (UWA). The UWA provides the user a view of the record and set types defined in the schema which are visible through the user's subschema. The concept of a record and set in the network data model can be abstracted to define generalizations. These generic objects can be used to define the user's view of the database.

The <RECORD DECL> discussed in Section 2.1.2 is used to generalize the common set of properties possessed by all record types. These properties are used to define a generic object, the UWA_RECORD, which is an abstraction of the network data model record type. Associated with the UWA_RECORD are data attributes and procedures, or operators, which represent the procedural information contained

in the <LOCATION MODE> clause of the <RECORD DECL>. Additional operators are associated with the UWA_RECORD which map the user's view of a record type to/from its corresponding representation as a MASS_STORAGE_RECORD in the database. The UWA_RECORD generalization is used to define abstract data types corresponding to each record type in the schema. An example from the PRESIDENTIAL schema is shown in Figure 3.3. The UWA_RECORD generalization is examined in detail in Section 4.1.

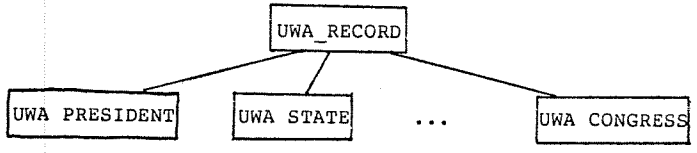


Figure 3.3 An Example of the UWA_RECORD Generalization

The user view of the database consists of both the record and the set, which represents relationships between record types. A generalization, or abstraction, of the network data model set type can be defined by using the <SET DECL> described in Section 2.1.2. This generalization is the UWA_SET and represents the properties common

to all set types. Associated with the UWA_SET are attributes which describe the record types, that is, MASS_STORAGE_RECORD's and UWA RECORD's, which participate in the set as owner and member. The attributes of the UWA_SET also include procedures, or operators, which implement the procedural aspects of the <SET DECL>, such as the set occurrence selection criterion specified by the <SOS> clause.

The UWA_SET generalization is related to the concept of aggregation examined by Smith and Smith. (An aggregation is a generic object which represents a relationship between two relations). The UWA_SET actually defines two relationships. The first is between MASS_STORAGE_RECORD instances which participate in set occurrences and is implicitly represented by the procedures associated with the UWA_SET. The second is between UWA_RECORD's which represent the set owner and set member. This relationship is explicitly represented by the owner UWA_RECORD and member UWA_RECORD pointers in the UWA_SET instance. The procedures associated with the UWA_SET map between the relationship between UWA_RECORDS which is visible to users and the relationship between MASS_STORAGE_RECORDS which is not visible to users.

The UWA_SET generalization is used to generate abstract data type definitions corresponding to the set

types defined in the schema and subschema. An example of the UWA_SET generalization in the PRESIDENTIAL schema is shown in Figure 3.4. The UWA_SET is described in Section 4.2.

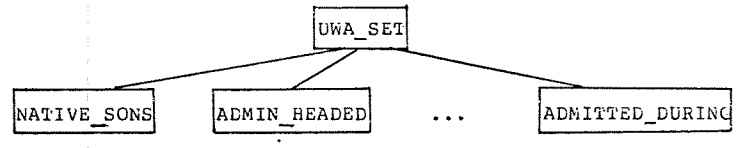


Figure 3.4 An Example of the UWA SET Generalization

The generic procedure model was used in Chapter 2 to describe the context dependence of the data manipulation routines. Since they are shared procedures and the schema and subschema are also shared, the data manipulation routines do not execute in the local environment at their point of declaration as does an ALGOL procedure. They execute instead in a non-local environment which is associated with the database. As described earlier in the generic procedure model, the data manipulation routines utilize a descriptor of their actual parameters to execute in this non-local environment. Depending upon implementation decisions, binding this descriptor occurs at compile-time, load-time, or run-time.

Abstract data types encapsulate an environment consisting of data structures and procedures which operate on these structures. By representing the record type (set type) of the network data model as an abstract data type, we are defining the semantics of the data structures associated with each record (set) since these structures can only be modified by the procedures, or operators, bound to the record (set).

The user schema and subschema are now represented as a collection of instances of these generic abstract data types. The operators associated with these abstract data types can be regarded as primitives to be used in the implementation of the data manipulation language. Each data manipulation routine is implemented as calls to the primitives associated with its actual parameters, which are of type UWA_RECORD and type UWA_SET. Rather than evaluating a descriptor associated with the actual parameter, the data manipulation routine uses the actual parameters to indirectly call the correct procedure instance within the schema and subschema. The data manipulation routines are described in detail in Section 4.2.

By defining the schema and subschema as abstract data types, the schema and subschema have been converted from a passive role in the generic procedure model to an active role. This is shown diagramatically in Figure 3.5.

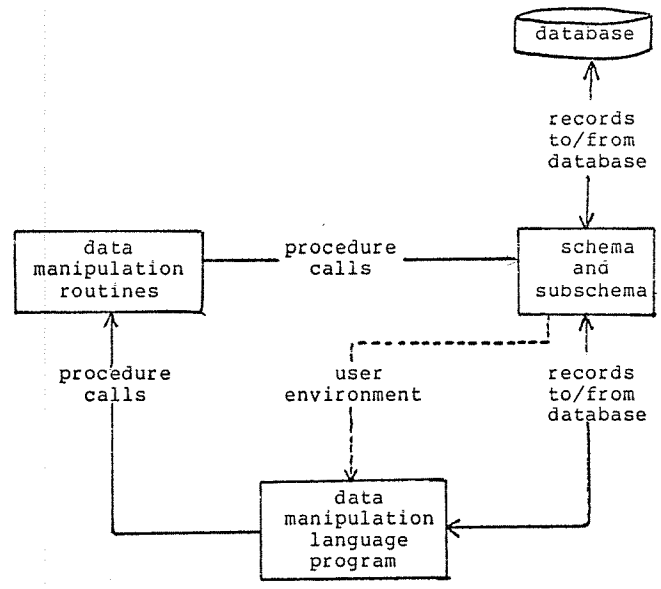


Figure 3.5 An Abstract Data Type Model of the Data Manipulation Routines

The abstract data type model shown in Figure 3.5 is similar to the design of a secure operating system developed by Robinson, et al [RLNS72]. The operating system is designed as a hierarchy of abstract machines implemented using abstract data types. Each level of the system is a manager of all objects of a given type, such as directories or segments. Each manager makes available a set of operations on these objects and provides protection for

all the objects it manages by enforcing protection rules for the management of its type of abstract object.

By defining the schema and subschema as a collection of abstract data types, the data manipulation routines are converted into a similar type manager. The environment of the user and the environment of the data manipulation routines are two different protection domains. Each domain has a different set of access rights to the schema and subschema. The user is only able to see an environment consisting of the data manipulation routines and the data attributes of the UWA_RECORD and UWA_SET instances corresponding to the record and set types defined in the schema and subschema. In the environment of the data manipulation routines, however, the operators associated with the UWA_RECORD and UWA_SET are visible. The MASS_STORAGE_RECORDS corresponding to each UWA_RECORD are also visible. In this way the conflict between data independence and sharing of the schema and subschema which was described earlier can be resolved.

There are additional advantages of using abstract data types to represent the schema and subschema. The first is that they allow sharing the schema and subschema to be directly supported within the data manipulation language. They also provide a means of binding at compile-time the procedural information specified in the schema and

subschema to achieve increased performance. This impact on performance is analyzed in detail in Chapter 5. As described by Brodie and Tsichritzis [BT77], if specification techniques are employed for defining the schema and subschema, then the user environment presented by the subschema can be guaranteed to remain constant even though the schema and subschema are modified as the database changes.

3.4 Programming Language Support Required

The objective of this dissertation is to study the use of high-level languages as an implementation tool for database management systems by studying the implementation of the abstract data type model. To implement this model the features the programming language must support are the following database management functions:

- Representation of the complex objects within the database
- Sharing the schema and the subschema among user programs
- Controlling access to the schema through the subschema
- Controlling concurrent access to the database
- Providing facilities for protection, security, and recovery.

A number of the programming language features used in the implementation described here are based on the concepts of SIMULA 67 [DMN70]. The algorithms which follow are presented in the syntax of SIMULA 67. A discussion of SIMULA 67 is presented in Appendix B.

The network model database is composed of complex list structures. To represent these structures the language should support pointer variables, user-defined types, and abstract data types. In addition, in order to support generic objects, it is necessary to separate the declaration of an abstract data type from its implementation. The external view of an abstract data type provides to its user a view of the generic, or abstract, properties of the object. The declaration of the abstract object is the definition of these properties. The external view can be used in the development of any modules which access this particular object.

The implementation of these properties is hidden from view. The function of separating the declaration from the implementation is to isolate the use of the abstract data type from its implementation. Frequently this isolation is even stronger than simply hiding the details of implementations. To strengthen this isolation, the declaration of the properties and behavior of the abstract object are created independently from its implementation. For example, CLU completely separates the process of describing the external, abstract properties of an abstract data type from the process of defining their implementation.

VIRTUAL attributes are a mechanism defined in SIMULA 67 for separating the declaration of an abstraction from

its implementation. Using VIRTUAL attributes multiple implementations of a common abstraction may coexist, as shown in Figure 3.6. VIRTUAL procedures are very similar to forward procedures in PASCAL, except that multiple implementations are allowed, one provided by the schema and another implementation possibly provided by each subschema.

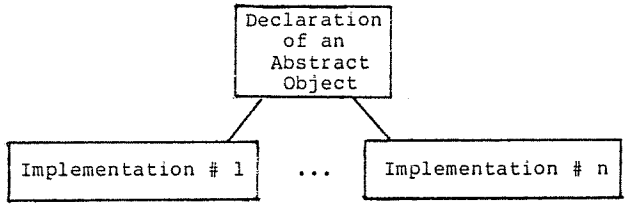


Figure 3.6 Coexisting Implementations of an Abstraction

The procedure attributes of the UWA_RECORD and UWA SET abstract data types were declared to be VIRTUAL. The data attributes of the OWNER_MSREC and MEMBER_MSREC abstract data types were also declared as VIRTUAL. When a UWA_RECORD (or UWA SET) abstract data type is instantiated to form, for example, a UWA_PRESIDENT corresponding to the PRESIDENT record type, a customized procedure for each VIRTUAL procedure will be generated based on the schema

definition of that record (or set) type. Furthermore, by using environment concatenation, each VIRTUAL procedure can be redefined based on the subschema through which a user program accesses the database. The schema is defined as a collection of instances of the ABSTRACT_OBJECTS in which each instance may have an independent implementation of the VIRTUAL attributes.

As an example, the UWA_RECORD contains a VIRTUAL procedure LOCATE which implements retrieval of MASS_STORAGE_RECORD instances using the strategy specified in the <LOCATION MODE> clause of the schema. Using the UWA_RECORD generalization as a template, abstract data type definitions are generated which correspond to the record types declared in the schema and which implement the particular forms of <LOCATION MODE> specified within the record type declarations in the schema. In the PRESIDENTIAL schema, UWA_PRESIDENT is an implementation of this record abstraction and possesses as an attribute a procedure LOCATE which implements hashing on LASTNAME. UWA_ADMIN is also an implementation of this abstraction, but possesses an implementation of LOCATE which performs retrieval via the logical pointers of the ADMIN_HEADED set type. Each instance of a UWA_RECORD in the schema possesses an independent implementation of LOCATE which is

based on the <LOCATION MODE> clause of the schema record type declarations (see Figure 3.7).

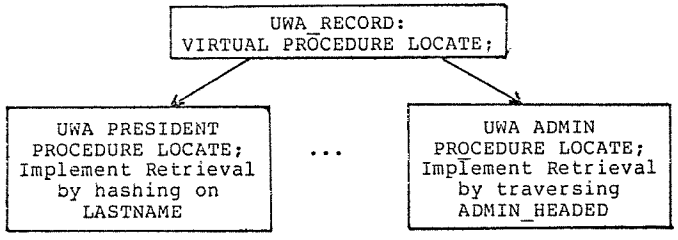


Figure 3.7 Coexisting Implementations of the UWA_RECORD Instances in the PRESIDENTIAL schema

Support for sharing the schema and subschema and for controlling access to the schema through the subschema appear to be conflicting goals. However, they are both related through the use of environment control facilities. Two very different environment control capabilities are necessary to support the schema and the subschema. The first of these concepts is environment concatenation. The CODASYL DBTG Report describes compilation of the schema, the subschema, and the data manipulation language program as independent separate steps. In an interpretive approach the encoded tables produced by previous steps are available at each subsequent step and are utilized by the

database management system to interpret the user's data manipulation language program. In a compiled approach, each step can be regarded as compilation of a separate module. Each step in this compilation represents an enhancement of the user environment. Techniques of separate compilation allow a compiler to encode the environment of a module and pass it to a module compiled later. To make both the schema and the subschema environments available to the user data manipulation language program, the programming language used to implement the database management system must support the ability to concatenate the environment defined by the schema with the environment defined by the subschema. This concatenated environment is then available as a "global" environment to the data manipulation language program.

The data attributes of the schema represent the global status of the database. In contrast to SIMULA, only one instance of the schema exists and all of its attributes are assumed to be shared. To represent the data which is local to a user, such as the data items associated with the UWA_RECORD instances, private variables are used. A private variable is a variable which is assumed to be part of a shared abstract data type. But a private copy of the variable is generated for each user of the abstract data type. The concept of private variables was

introduced by Owicki [Owi77] as a method of simplifying concurrent program verification. Owicki presents proof rules for verifying the correct implementation of shared objects containg private variables.

Support for the schema and subschema requires another level of environment control. The subschema must support the following abilities:

- o Removal of one or more areas declared in the schema
- o Removal of one of more set types declared in the schema
- o Removal of one or more record types declared in the schema
- o Removal of one or more record items declared in the schema
- o Renaming of an area, set type, record type or data item declared in the schema
- o Association of a new data type with a data item declared in the schema
- o Association of a new set occurrence selection clause with a set type declared in the schema.

Two major programming language concepts needed to support the relationship between the schema and subschema are binding and environment or name scope control. Binding is the establishment of a value to be associated with an identifier. In the case of the schema or the subschema, binding a record type means associating the entire declaration of a record type with the identifier for the record type supplied in the schema or subschema.

ALGOL employs block structured binding. It allows an identifier to be declared in more than one name scope. However, usage of an identifier is at all times unambiguous, as the innermost accessable definition is always chosen. Block structure with the concept of parallel inner blocks is very close to the hierarchical structure of the schema and multiple subschemas. The ability to concatenate name scopes can be examined very readily in the context of ALGOL block structure conventions. The schema represents the outermost block and the subschemas represent a collection of parallel nested blocks. Name scope concatenation allows the environment of the outermost block, the schema, to be concatenated with the environment defined by each of the subschemas. This combined environment, with compiler support for separate compilation, can be made available as the enviroment for the data manipulation language program. Thus the data manipulation language programs sharing the subschema represent another level of blocks which are nested within the subschemas so

that several user programs can share the same schema and subschema.

What programming language features are required to implement the subschema functions? The ability to define a new identifier to represent an area, set type, or record type requires that the language support the ability to define aliases. Given this capability, the subschema can declare a new identifier of the desired area, record, or set type. The definition of an alias is a command to the compiler to create a second name for an object or data item. All references in the subschema are required to reference the object using the alias.

This alias definition may also supply a new data type to be associated with a data item. A reference to the data item thus requires a coercion to be invoked. If such coercions are not automatically generated by the compiler, an alternative approach is available. The most general form of this approach is related to languages which allow encapsulation of data structures and procedures. In such a language an equivalent type declaration can provide an identifier with a new data type value and a coercion procedure. Such redefined attributes are actually virtual since they are computed from the data item value occurrence using a coercion from the type declared in the schema. The problem with this general solution is the problem

of assignment to the virtual identifier. The reference to the virtual attribute is actually a function call, and most programming languages do not provide syntax which supports assignment to such a function. This problem can be solved by encapsulating two coercion procedures, one for references and the other for assignments.

This coercion can be associated with the alias in a similar manner to BLISS structure definitions [WRH71]. In BLISS the declaration of a structure such as an array consists of two components: a declaration of the structure and a declaration of the expression for computing the address of an individual element of the structure. This address generating expression is bound as a macro to the identifier and is expanded for each identifier reference.

The remaining subschema functions are more difficult to support. Eliminating visiblility of identifiers in the subschema can be implemented using several approaches. The first approach is an extension of the concept of environment concatenation. The basic function of environment concatenation is in strong conflict with the desire to restrict access to an identifier. The ALGOL convention makes all identifiers visible in the outer block visible within an inner block. The environment which a block may pass to its descendants is the environment it inherited from outer blocks plus the locally defined environment.

The problem in controlling access rights is to provide a means to restrict at the subschema level those identifiers which are declared in the schema. The HIDDEN specification [Pal76] allows a block to restrict the environment which it passes to inner blocks. A HIDDEN specification may be applied to either an identifier inherited from an outer block or may be applied to a local identifer. An identifier which is HIDDEN is invisible in inner blocks. To accomplish this it must be possible to declare at the subschema level that an identifier declared in the schema is HIDDEN and is not accessible to user data manipulation language programs using the subschema.

An alternative approach is based on the concept of EXPORTED and IMPORTED declarations which are used in EUCLID [LHLMP77], MESA [GMS77], and TELOS [THLZ77]. The subschema and schema can be implemented in the following manner. All identifiers in the schema are declared as EXPORTED, e. g., available externally. The subschema then declares that only the identifiers explicitly referenced in the subschema are IMPORTED, or available to the data manipulation language environment. Thus all identifiers from the schema are available and may be referenced by a subschema. Only those identifiers explicitly referenced within a subschema are available via the subschema. The major difficulty with this approach is the problem of con-

trolling the data manipulation language program so that it does not attempt to declare any schema identifiers as IMPORTED and thus bypass the subschema.

The use of HIDDEN specifications (or the alternate syntax NOT HIDDEN) provides a means to restrict the environment passes to its descendants. HIDDEN may specify that either a name in the inherited environment is hidden or that a locally defined name is hidden. We feel that this feature is significantly different from the use of EXPORT, which names locally defined identifiers but does not allow the IMPORTEDed identifiers to be in the blocks EXPORTed declaration. By using HIDDEN specifications, the abstract data type model shown in Figure 3.4 can be incorporated into ALGOL 60 block structure and yet provide the environment restriction capabilities required to implement the model.

The redefinition of the set occurrence selection clause is a more complex problem. It allows the subschema to rebind part of the schema set type declaration. A technique to provide this ability is based on VIRTUAL procedures. Associated with the declaration of the VIRTUAL name is a declaration of the set occurrence selection clause which is associated with the set type declaration in the schema. Subsequently this declaration of the set occurrence selection clause may be replaced by a declara-

tion in the subschema which is bound to the VIRTUAL name. If no redefinition occurs in the subschema, the schema version is used.

Consider as an example the set type ADMIN_HEADED. The ADMIN_HEADED instance in the schema is the concatenation of the properties of the UWA_SET and of the ADMIN_HEADED set type declared in the schema. The procedure LOCATE associated with the UWA_SET implements the set selection criterion specified in the schema. The procedure LOCATE specified in the UWA SET is VIRTUAL and consequently is supplied an implementation in the declaration of the UWA_ADMIN_HEADED. Since LOCATE is a VIRTUAL procedure a transfer vector is incorportated in each instance of the UWA_SET. This transfer vector is used to invoke the implementation of the procedure supplied in the schema (see Figure 3.8). If a new set occurrence selection criterion is specified in the subschema, this transfer vector is used to bind the new LOCATE procedure supplied in the subschema as shown in Figure 3.9. A transfer table is used to bind all of the new implementations of the VIRTUAL procedures supplied by the subschema to the corresponding attributes of the UWA_RECORD or UWA_SET in the schema.

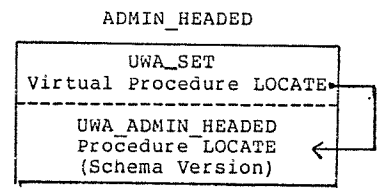


Figure 3.8 Binding the VIRTUAL procedure LOCATE in the Schema

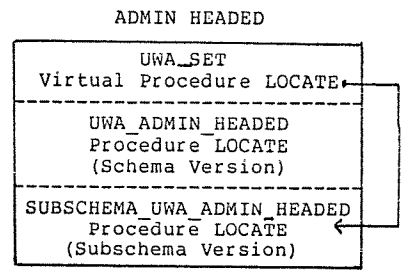


Figure 3.9 Binding the VIRTUAL Procedure LOCATE in the Subschema

A major source of complexity in a database management system is the concurrent execution of user programs which modify the database. Substantial research has been done in this area utilizing an abstract data type known as a

monitor [Bri75] [Hoa74] to control resource allocation. Monitors bind information about the state of the shared resource to procedures which implement semaphores in order to schedule processes which require mutually exclusive use of the resource.

Monitors can be employed in a compiled schema to manage concurrency. Associated with each UWA_RECORD instance in the schema is a monitor which implements procedures to lock and to unlock record instances within the database. This technique requires that each record occurrence has a unique identifier and that each user program has a unique identifier which is known to tne database management system. The structures local to the monitor are a list of all record instances which are currently locked and a queue of user programs which are waiting on a locked record. All data manipulation language routines must incorporate calls to the monitor procedures associated with the UWA_RECORD instance.

As described by Brinch Hansen, monitors do not completely solve the problem of concurrency control. Associated with the schema must be some form of deadlock detection or avoidance algorithm. Deadlock handling is actually more difficult using a monitor for each resource type. A monitor encapsulates the data describing the current user programs which are blocked waiting for exclusive use of

a record instance. A deadlock detection algorithm must have access to the data describing the status of all record types, which contradicts the concept of encapsulating the data. The access control rights must provide the deadlock detection routine access to the data of all monitors, but must enforce restricted access rights for each monitor such that it only has access to its local data.

The remaining functions to consider are features for protection, security, and recovery. The abstract data type model enhances all of these functions. Protection means that the database is protected from inadvertant modification due either to programming or to machine errors. A major task of protection is error confinement [Den76] [Lin76b]. The use of abstract data types forces the components of the database management system to interact along a fixed and known set of paths. No unexpected interactions are possible. By restricting the environment which a component can access and by requiring the component to interact with other components in fixed way, error confinement, and hence protection, will be enhanced.

The same environment restriction mechanisms used to enhance protection will also enhance security and recovery. Security means that a user is only allowed to access and to modify information which is relevant to his task and to which he possesses an access privilege. By re-

stricting component interactions to a fixed set of ways, the ability to verify a user's access privileges is facilitated.

Error recovery is also facilitated by the environment encapsulation of the abstract data type model. By restricting the environment accessable to a module, the potential damage due to an error is also restricted. Thus the detection of errors may be simplified, and the scope of the recovery operation is reduced.

CHAPTER 4

AN IMPLEMENTATION

Chapter 3 introduced an abstract data type model of the data manipulation routines. In this model the schema and the subschema are represented as a collection of shared abstract data types. Three abstract data types, or generic objects, were introduced in Chapter 3: the MASS_STORAGE_RECORD, the UWA_RECORD, and the UWA_SET. These three objects form a collection referred to in this chapter as the ABSTRACT_OBJECTS.

The user environment consists of the ABSTRACT_OBJECTS, the data manipulation routines, and the UWA_RECORD and UWA_SET instances representing tne schema and subschema. The components of this environment are discussed in this chapter.

The user schema is represented as a collection of instances of the UWA_RECORD and UWA_SET which correspond to the record and set types defined in the schema. The gen-

eration of the user schema using the UWA_RECORD and UWA_SET generalizations is discussed in Section 4.1.

Each of these generic objects is composed of a collection of attributes which can be divided into data attributes and procedure attributes. The data manipulation routines are defined completely in terms of the attributes associated with these objects. The implementation of the data manipulation routines is described in Section 4.3. The subschema provides a means of restricting and reformatting the user's view of the structure of the database he is accessing. The use of environment concatenation and of VIRTUAL procedures to define the subschema is discussed in Section 4.4.

4.1 The User Environment

The user environment consists of the names of the data manipulation routines, the set and record of the network data model, and the names and characteristics of the data items, record types, and set types that are described in the schema and that are visible through the user's subschema. This environment is shared by all users of a subschema and is associated with the user program by using environment concatenation. To simplify the discussion which follows, we will initially consider only the role of

the schema. The subschema will be treated later in Section 4.4.

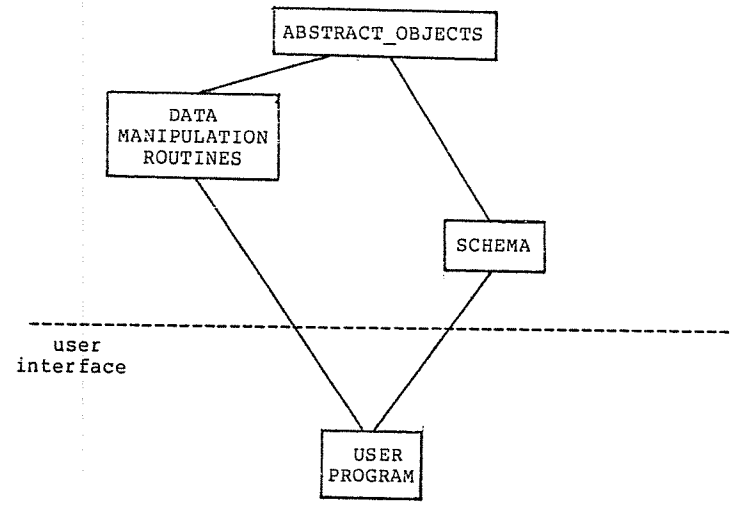


Figure 4.1 The Definition of the User Environment

This logical structure represents the abstract data type model shown earlier in Figure 3.4 modified to conform to name scope conventions similar to those of ALGOL 60. The user environment grows in the downward direction, similar to nested blocks in ALGOL 60. But to support data independence HIDDEN specifications are used. Each compo-

nent of the system inherits an environment from the component above it. It then may either enhance this environment by declaring local identifiers and may pass its local environment to its descendants in its complete form or may restrict a part of the environment it passes to its descendants by using a HIDDEN specification.

4.2 The Abstract Objects

A generalization is an abstraction in which a set of similar objects are regarded as instances of a single named generic object. The differences between objects are ignored and their common properties are identified. In the network data model a database is a shared resource for which several generalizations exist. These generalizations were introduced in Section 3.3 and were used to describe the two different views of the database which the database management system must support. The first representation of the database is the physical database shared by all users. Records as they exist in physical mass storage are generalized to define the MASS_STORAGE_RECORD which is described in Section 4.2.1.

The second view of the database is the view which is provided to the user through the User Work Area (UWA). The UWA provides a view of the record and set types de-

fined in the schema and visible through the user's subschema. The concepts of records and sets are abstracted to define two generalizations: the UWA_RECORD described in Section 4.2.2 and the UWA_SET described in Section 4.2.3.

The schema is the definition of the structure of a database in terms of the structures available in the data model. In the network data model, the schema is written in the Data Definition Language (DDL) described in Section 2.1. To represent the schema as a collection of abstract data types, this DDL schema declaration is translated into the equivalent declaration of a collection of abstract data types. These declarations are generated by using the declarations of the ABSTRACT_OBJECTS described in this section as templates. For each record type declared in the schema, a declaration of a corresponding MASS_STORAGE_RECORD and of a UWA_RECORD is generated. For each set type declared in the schema a corresponding UWA_SET declaration is also generated. These abstract data type declarations are then compiled and are bound to the user program at <u>load-time</u> by using environment concatenation.

Each of these abstract data types uses environment concatenation to bind the generic properties of the generalizations in the ABSTRACT_OBJECTS to the specific proper-

ties of the record type or set type declared in the schema. Each compiled abstract data type declaration also supplies an implementation of the VIRTUAL attributes of the generalization on which it is based.

4.2.1 The Mass Storage Record Generalization

As described in Section 3.3.1, the database is a shared resource and the sharing is managed by the database management system. The database can be regarded as a discriminated union of records, as described in Section 3.2. The record type and hence the associated properties of each record instance within the database can be determined by testing, at run-time, the tag associated with the record instance. By analyzing the common properties of record instances within the database, we can define a generalization, or abstraction, of records as they exist in mass storage. This generalization, the MASS_STORAGE_RECORD, is shown in Figure 4.2. Associated with a record instance in the database are the logical pointers which the data manipulation routines use to traverse the database and which are hidden from users.

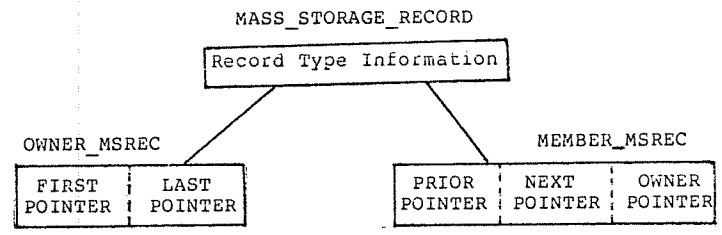


Figure 4.2 The MASS_STORAGE_RECORD Hierarchy

The MASS_STORAGE_RECORD generalization is used to define a generic hierarchy consisting of two subabstractions. The subabstractions OWNER_MSREC and MEMBER_MSREC represent the participation of a record instance in set occurrences within the database. The OWNER_MSREC generalization abstracts the information describing the ownership of set occurrences by a record instance, while the MEMBER_MSREC generalization abstracts the membership of a record instance in one or more set occurrences.

As shown in Figure 4.3, M_S_CONGRESS, which corresponds to the CONGRESS record type in the PRESIDENTIAL schema, is declared to be a set owner. Figure 4.3 also shows that M_S_CONG_PRES_LINK, which corresponds to the CONGRESS_PRESIDENT_LINK record in the schema, is defined

to be a set member. The OWNER_MSREC and MEMBER_MSREC abstractions are not mutually exclusive, as shown by M_S_PRESIDENT which is both a set owner and a set member.

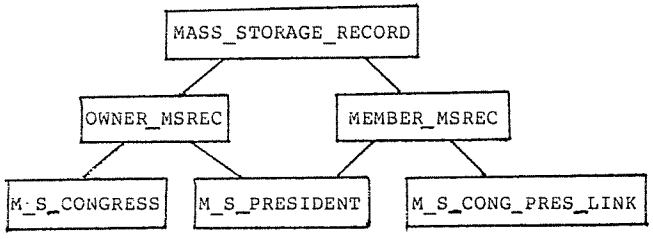


Figure 4.3 An Example of the MASS_STORAGE_RECORD Hierarchy

This MASS_STORAGE_RECORD generalization, or generic object, represents a template for creating record instances in mass storage. The MASS_STORAGE_RECORD, which has no procedure attributes, has the following data attributes:

1. Record Type Information - This field is used to tag each record instance in the database with a type descriptor so that run-time checking can be performed.
2. OWNER_MSREC - This component of the MASS_STORAGE RECORD is used to describe the set types which are owned by this record type. The data attributes of the OWNER_MSREC are a FIRST pointer field and a LAST pointer field. These data attributes are VIRTUAL.
3. MEMBER_MSREC - This component of the MASS_STORAGE_RECORD abstract data type is used to describe the set types in which the record participates as a member. The data attributes of the Member Subabstraction are PRIOR, NEXT, and OWNER pointer fields. These data attributes are VIRTUAL.

This definition of the MASS_STORAGE_RECORD is used to translate each of the record type declarations in the schema into equivalent abstract data type declarations. Using the MASS_STORAGE_RECORD as a template, environment

concatenation, which was described in Section 3.5, is used to combine the properties of the record type declared in the schema with those properties defined in the MASS_STORAGE_RECORD generalization. Concatenated to each MASS_STORAGE_RECORD instance are the possibly encoded forms of the user-visible data items which are defined in the <RECORD DECL> in the schema.

The properties of the OWNER_MSREC and the MEMBER MSREC were declared to be VIRTUAL. This means that no implementation of these properties was provided in the original declaration of the MASS_STORAGE_RECORD. Instead, implementations are supplied with the declaration of each object which is an instance of the generalization. For example, when an OWNER_MSREC (or MEMBER_MSREC) is instantiated to create a M_S_PRESIDENT abstract data type, a customized implementation of the FIRST and LAST pointer fields (or PRIOR, NEXT and OWNER pointer fields) is generated to represent the participation as owner (member) of M_S_PRESIDENT for set occurrences in the database. The PRESIDENT record type declaration and the corresponding M_S_PRESIDENT declaration are shown in Figure 4.3 and 4.5, respectively.

The abstract data type M_S_PRESIDENT uses environment concatenation to combine this customized implementation of tne MASS_STORAGE_RECORD with the attributes which are

unique to the PRESIDENT record type and which are declared in the schema. Thus a M_S_PRESIDENT abstract data type concatenates to the MASS_STORAGE_RECORD the data items LASTNAME, FIRSTNAME, etc, which are declared in the schema.

```
RECORD NAME IS PRESIDENT
     RECORD CODE IS 5
     LOCATION MODE IS CALC USING LASTNAME
     DUPLICATES ARE ALLOWED
     TEXT LASTNAME, FIRSTNAME,INITIAL
     TEXT MONTH_BORN
     INTEGER DAY_BORN,YEAR_BORN
     TEXT HEIGHT, COLLEGE, ANCESTRY, RELIGION
     TEXT MONTH_DIED
     INTEGER DAY_DIED, YEAR_DIED
     TEXT CAUSE_OF_DEATH, FATHER, MOTHER
```

Figure 4.4 The PRESIDENT DDL Record Type Declaration

```
MASS_STORAGE_RECORD CLASS M_S_PRESIDENT;
BEGIN HIDDEN ALL;

     COMMENT Implement set member pointers ;

     REF(MASS STORAGE RECORD) ARRAY NEXT, PRIOR,
          OWNER (1:2);

     COMMENT Index 1 = member of ALL_PRESIDENTS_SS
             Index 2 = member of NATIVE_SONS;

     COMMENT Implement set owner pointers ;

     REF (MASS_STORAGE_RECORD) ARRAY FIRST, LAST (1:3);

     COMMENT Index 1 = Owner of ADMIN_HEADED
             Index 2 = Owner of ELECTIONS_WON
             Index 3 = Owner of CONGRESS_SERVED;


     TEXT LASTNAME, FIRSTNAME,INITIAL;
     TEXT MONTH BORN;
     INTEGER DAY BORN,YEAR_BORN;
     TEXT HEIGHT, COLLEGE, ANCESTRY, RELIGION;
     TEXT MONTH_DIED;
     INTEGER DAY DIED, YEAR DIED;
     TEXT CAUSE OF DEATH, FATHER, MOTHER;

     COMMENT Initialize Type Information ;

     RCODE := 5; ID := "PRESIDENT";
END;
```

Figure 4.5 The M_S PRESIDENT Declaration

4.2.2 The UWA Record Generalization

The other view of the database is the view provided to the user through the UWA. The UWA provides the user a view of the record and set types defined in the schema which are visible through the user's subschema. The concept of a record in the network data model can be abstracted to define a generalization, the UWA_RECORD, which represents the user's view of records in the database.

The <RECORD DECL> discussed in Section 2.1.2 is used to generalize the common set of properties possessed by all record types. These properties can be used to define a generic object, the UWA_RECORD, which is an abstraction of the network data model record type as shown in Figure 4.5. Associated with the UWA_RECORD are data attributes. Also associated with the UWA_RECORD are procedures, or operators, which represent the procedural information contained in the <LOCATION MODE> clause of the <RECORD DECL>. Additional operators are associated with the UWA_RECORD which map the user's view of a record type to/from its corresponding representation as a MASS_STORAGE_RECORD in the database.

RECORD TYPE INFORMATION
CURRENT
NUMSETSOWNED
NUMSETSMEMBER
UWA SET INSTANCE POINTERS
LOCATE PROCEDURE
ALLOCATE PROCEDURE
STORE PROCEDURE
LOAD PROCEDURE

Figure 4.6 The UWA_RECORD Generalization

The UWA_RECORD abstract data type represents a template for a record type in the User Work Area. Its data attributes are:

1. Record type information - For run-time checking of record occurrences.
2. CURRENT - Current instance of the MASS_STORAGE_RECORD of this record type.
3. UWA_SET Instance Pointers - A pointer to all UWA_SET instantiations to which the record type instantiation is an owner or member. A UWA_PRESIDENT Record instantiation of the UWA_RECORD would contain five instances of the UWA_SET Pointer attribute - one each for the ALL_PRESIDENTS_SS, NATIVE_SONS, ADMIN HEADED, ELECTIONS_WON, and CONGRESS_SERVED set types.
4. NUMSETSMEMBER - The number of sets in which the record type participates as a member.
5. NUMSETSOWNED - The number of sets in which the record type participates as an owner.

The procedure attributes of the UWA_RECORD abstract data type are:

1. LOCATE Procedure - This procedure is responsible for locating a record instance in mass storage and making it current of record and current of all sets in which it participates. It is a VIRTUAL procedure attribute.
2. ALLOCATE Procedure - This VIRTUAL procedure allocates a new MASS_STORAGE_RECORD occurrence using the MASS_STORAGE_RECORD as a template and the <LOCATION MODE> clause from the schema as a storage allocation procedure.
3. LOAD (STORE) Procedure - This VIRTUAL procedure is used to load (store) the items in a UWA_RECORD occurrence from (to) mass storage after the correct MASS_STORAGE_RECORD occurrence is located (allocated).

The definition of the UWA_RECORD described in Section 4.2.2 is used to translate the record type declarations in the schema into equivalent UWA_RECORD declarations. As an example, the UWA_PRESIDENT declaration shown in Figure 4.7 is generated from the record type declaration shown in Figure 4.4. Each of the VIRTUAL procedures in the UWA_RECORD declaration is implemented as part of UWA_PRESIDENT. For example, procedure LOCATE is implemented to perform retrieval of a M_S_PRESIDENT instance by hashing with LASTNAME as a key.

```
UWA_RECORD CLASS UWA PRESIDENT;
BEGIN
     HIDDEN LOCATE, ALLOCATE, STORE,LOAD;
     PRIVATE LASTNAME,FIRSTNAME,INITIAL,MONTH_BORN,
          DAY_BORN,YEAR_BORN,HEIGHT,COLLEGE,ANCESTRY,
          MONTH_DIED,DAY_DIED,YEAR_DIED,
          CAUSE_OF_DEATH,FATHER,MOTHER,CURRENT;
     TEXT MONTH_BORN,
     INTEGER DAY_BORN, YEAR_BORN;
     TEXT HEIGHT, COLLEGE, ANCESTRY, RELIGION ;
     TEXT MONTH_DIED;
     INTEGER DAY_DIED, YEAR DIED;
     TEXT CAUSE_OF_DEATH, FATHER, MOTHER;

     PROCEDURE ALLOCATE;
     COMMENT Allocate a new M_S_PRESIDENT using a
          hashing algorithm with LASTNAME as the key;

     PROCEDURE LOCATE;
     COMMENT Retrieve an M_S_PRESIDENT by hashing
          with LASTNAME as a key ;

     PROCEDURE STORE;
     COMMENT Perform encoding as required and copy
          local data items into data items of
          current M_S_PRESIDENT ;

     PROCEDURE LOAD;
     COMMENT Perform decoding as required and copy
          data items from current M_S_PRESIDENT
          into local data items ;

      COMMENT Initialize type information ;

      RECORDID := "PRESIDENT"; RECORDCODE :=5;
      NUMSETSOWNED := 3; NUMSETSMEMBER := 2;

END  of UWA_PRESIDENT;
```

Figure 4.7 The UWA_PRESIDENT Declaration

The local data items of UWA_PRESIDENT are declared as PRIVATE variables (see Section 3.5). By declaring the local data items of a UWA_RECORD instance as private, they are considered to be part of the shared abstract data type, but an individual copy of these variables is created for each user program using the schema. In this way each user program can be guaranteed not to interfere with other user programs.

4.2.3 The UWA_SET Generalization

The second component of the user's view of the database is the set which represents relationships between record types. A generalization, or abstraction, of the network data model set type can be defined by using the <SET DECL> described in Section 2.1.2. This generalization is the UWA_SET and represents the properties common to all set types. Associated with the UWA_SET are attributes which describe the record types, that is, MASS_STORAGE_RECORDs and UWA_RECORDs, which participate in the set as owner and member. The attributes of the UWA_SET also include procedures, or operators, which implement the procedural aspects of the <SET DECL>, such as the <SOS> clause (see Figure 4.8).

SET TYPE INFORMATION
CURRENT
OWNER UWA_RECORD POINTER
MEMBER UWA_RECORD POINTER
AUTOMATIC
OWNER_OFFSET
MEMBER OFFSET
INSERT PROCEDURE
LOCATE PROCEDURE
REMOVE PROCEDURE
REORDER PROCEDURE

Figure 4.8 The UWA_SET Generalization

The UWA_SET generalization is related to the concept of aggregation. (An aggregation is a generic object which represents a relationship between two relations.) However, the UWA_SET actually defines two relationships. The first is between MASS_STORAGE_RECORDS instances which participate in set occurrences and is implicitly represented by the procedures associated with the UWA_SET. The second is between the UWA_RECORDs which represent the set owner and set member. This relationship is explicitly represented by the OWNER UWA_RECORD pointer and the MEMBER UWA_RECORD pointer. The UWA_SET generalization is used as a template for the set types in a database. Its data attributes are:

1. Set Type Information - For run-time checking of set types.
2. CURRENT - Current instance of MASS_STORAGE_RECORD types participating in the set type.
3. OWNER UWA_RECORD Pointer.
4. MEMBER UWA_RECORD Pointer.
5. AUTOMATIC - A Boolean which indicates either manual or automatic set membership.
6. OWNER_OFFSET - The index of the correct FIRST and LAST pointers in tne owner MASS_STORAGE_RECORD type.
7. MEMBER_OFFSET - The index of the correct NEXT, PRIOR, and OWNER pointers in the member MASS_STORAGE_RECORD type.

The procedure attributes of the UWA Set abstract data type are:

1. INSERT - A VIRTUAL procedure to insert a new member record occurrence into the set using the SET ORDER clause from the schema.
2. LOCATE - A VIRTUAL procedure to locate an occurrence of a set based on the SET OCCURRENCE SELECTION clause of the schema.
3. REMOVE - A VIRTUAL procedure to remove a specific member record occurrence from the set occurrence.
4. SCAN - A VIRTUAL procedure to traverse a set occurrence by following the pointers which link the owner record occurrence and the member record occurrences and the pointers which link the member record occurrences.

The definition of the UWA_SET described in Section 4.2.3 is used to translate the set type declarations in the schema into equivalent UWA_SET declarations. As an example, the NATIVE_SONS set type declaration is shown in Figure 4.9 and the corresponding UWA_NATIVE_SONS is shown in Figure 4.10. Each of the VIRTUAL procedures of the UWA_SET declaration is implemented as part of the UWA_NATIVE_SONS declaration.


```
SET NAME IS NATIVE_SONS
    SET CODE IS 5
    OWNER IS STATE
    MEMBER IS PRESIDENT
    ORDER IS SORTED
    ASCENDING KEY IS LASTNAME
    DUPLICATES ARE LAST
    SET OCCURRENCE SELECTION IS THRU
    LOCATION MODE OF OWNER
```

Figure 4.9 The NATIVE_SONS DDL Set Type Declaration

```
UWA_SET CLASS UWA NATIVE SONS;
BEGIN
     HIDDEN UWA_OWNER,UWA_MEMBER,INSERT,LOCATE,
          REMOVE,REORDER,MEMBER_OFFSET,OWNER_OFFSET;

     PROCEDURE LOCATE;
     COMMENT Implement Set Occurrence Selection through
          the location Mode of the Owner, e.g.,
          call UWA_STATE.LOCATE;

     PROCEDURE REMOVE;
     COMMENT Remove the indicated M_S_PRESIDENT
          from a set occurrence.  Use OWNER_OFFSET
          and MEMBER OFFSET to access the correct
          pointer locations ;

     PROCEDURE SCAN;
     COMMENT Based on the actual parameter switch value
          move from the current record along the
          correct pointer.  Use OWNER_OFFSET and
          MEMBER_OFFSET to access the pointer locations.

     PROCEDURE INSERT;
     COMMENT Insert a new M_S_PRESIDENT instance into a
          set occurrence.  Maintain the set order as
          sorted in ascending order using  LASTNAME.;

     COMMENT Initialize type information ;

     SETCODE := 5; MEMBER_OFFSET := 2;
     OWNER_OFFSET := 1; ID := "NATIVE_SONS";

END OF UWA_NATIVE SONS;
```

Figure 4.10 The UWA_NATIVE SONS Declaration

4.3 The Data Manipulation Routines

The data manipulation routines are generic procedures. For example, the FETCH data manipulation routine is a procedure with only one actual parameter, a UWA_RECORD. Each record (set) declared in the schema provides a detailed description of a record's (set's) characteristics. All of the characteristics of the actual parameter must be known to execute the data manipulation routine.

Chapter 2 described current techniques used to bind this information to the data manipulation routines. Chapter 3 introduced an abstract data type model in which the schema is represented as a collection of abstract data types. By representing the schema as a collection of abstract data types, this information from the schema is bound to instantiations of the UWA_RECORD and UWA_SET. The use of VIRTUAL attributes allows the definition of the specifications of the ABSTRACT_OBJECTS without an implementation being supplied. Using the declarations of the ABSTRACT_OBJECTS as templates, the schema is then compiled into a collection of implementations of the ABSTRACT_OBJECTS. Rather than perform interpretation, the data manipulation routine uses each actual parameter to indirectly reference the correct data or procedure attrib-

ute in the schema. Thus the data manipulation routines are completely implemented in terms of the data attributes and procedures of the abstract data types.

VIRTUAL attributes allow us to separate the declaration of an abstract data type's external properties from their implementation. The data manipulation routines are defined completely in terms of the VIRTUAL properties of the ABSTRACT_OBJECTS, since at the time that the data manipulation routines are compiled implementations of these attributes will not exist. Consequently, the ABSTRACT_OBJECTS and the data manipulation routines are database independent.

As shown earlier in Figure 4.1, the user environment grows following rules similar to ALGOL 60 name scope conventions. However, the user program specifies the compiled collection of abstract data types which are bound at load-time to the user program. Thus the environment for the data manipulation routines is actually specified in the upward direction in Figure 4.1.

Abstract data types allow us to encapsulate the information required by the data manipulation routines within instances of the ABSTRACT_OBJECTS. These ABSTRACT_OBJECT instances are environments which are passed as parameters to the data manipulation routines. Through the use of HIDDEN specifications, the implementa-

tion details present in this environment are not visible to users, but are visible to the data manipulation routines since the VIRTUAL attributes of the ABSTRACT_OBJECTS are visible in the name scope of the data manipulation routines.

Each actual parameter is used to indirectly reference the correct data attribute or procedure attribute associated with an abstract data type instance in the schema. (See Figure 4.11. The dashed lines represent the environment definition from Figure 4.1). Thus the details of the schema are bound at run-time, without interpretation, through the actual parameters to the data manipulation routines.

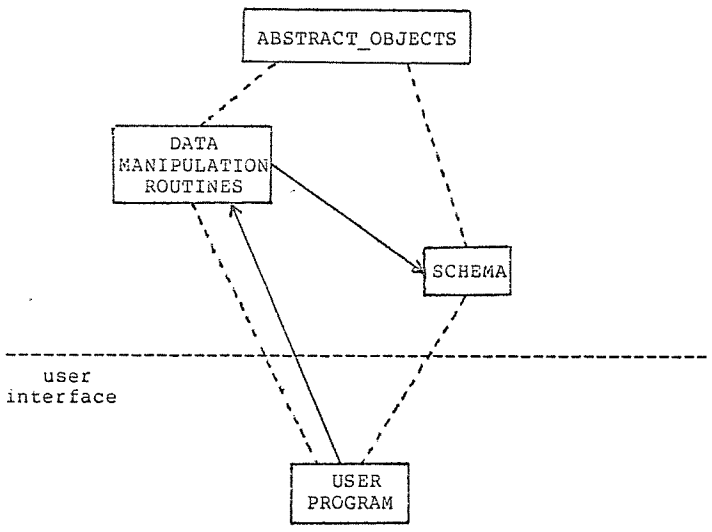


Figure 4.11 Procedure Calls to the Schema

Figures 4.12 and 4.13 contain implementations of the data manipulation routines FETCH and STORE, respectively. In both of these examples RECPTR references an instantiation of the UWA_RECORD abstract data type (i.e., a record type).


```
PROCEDURE FETCH(RECPTR); REF (UWA RECORD) RECPTR;
BEGIN INTEGER I;

     COMMENT use the schema location mode to find
         the MASS_STORAGE_RECORD and then copy
         MASS_STORAGE_RECORD fields into UWA RECORD
         fields;

     COMMENT Make the MASS_STORAGE RECORD the current
         of record type;

     RECPTR.CURRENT :- RECPTR.LOCATE;

     COMMENT Load all data items into the UWA ;

     RECPTR.LOAD (RECPTR.CURRENT);

     COMMENT Make the record occurrence the current of
          set in all sets in which it participates ;

     FOR I:=1 STEP 1 UNTIL RECPTR.NUMSETSOWNED DO
          RECPTR.OWNER(I).CURRENT :- RECPTR.CURRENT;
     FOR I:=1 STEP 1 UNTIL RECPTR.NUMSETSMEMBER DO
          RECPTR.MEMBER(I).CURRENT :- RECPTR.CURRENT;
END;
```

Figure 4.12 Data Manipulation Routine FETCH

```
PROCEDURE STORE(RECPTR); REF (UWA_RECORD) RECPTR;
BEGIN INTEGER J; REF (MASS_STORAGE_RECORD) MSREC,OREC;

     COMMENT generate a new mass storage record
          using the schema information ;

     MSREC :- RECPTR.ALLOCATE;
     RECPTR.STORE(MSREC);
     RECPTR.CURRENT :- MSREC;

     COMMENT make the mass storage record the current of
          set for each set type it owns;

     FOR J := 1 STEP 1 UNTIL RECPTR.NUMSETSOWNED DO
          RECPTR.OWNER(J).CURRENT :- MSREC;

     COMMENT Insert the record into all sets
         in which it is a member.;

     FOR J := 1 STEP 1 UNTIL RECPTR.NUMSETSMEMBER DO
     BEGIN
          OREC :- RECPTR.MEMBER(J).LOCATE;
          RECPTR.MEMBER(J).INSERT(MSREC,OREC);
          RECPTR.MEMBER(J).CURRENT :- MSREC;
     END;
END;
```

Figure 4.12 Data Manipulation Routine STORE

4.4 Redefinition of the User Environment in the Subschema

The generalizations defined in Section 4.2 represent the components of the network data model. This section investigates the use of these generalizations to represent the schema and subschema. The subschema provides a means

of restricting and reformatting the user's view of the structure of the database he is accessing.

A subschema definition, PRES_ADMIN, is shown in Figure 4.14. This subschema restricts the user's view of the PRESIDENTIAL database to the ALL_PRESIDENTS, PRESIDENT, and ADMIN record types and to the ALL-PRESIDENTS_SS and ADMIN_HEADED set types. In addition, the <SOS> clauses of the ADMIN_HEADED set is changed from LOCATION MODE OF OWNER in the schema to CURRENT OF SET in the subschema.

```
SUBSCHEMA PRES_ADMIN OF PRESIDENTIAL SCHEMA
BEGIN
     RECORD=ALL_PRESIDENTS,
     RECORD=PRESIDENT,
            ALIAS FOR LASTNAME IS PNAME
     RECORD=ADMIN,
     SET=ALL_PRESIDENTS,
     SET=ADMIN_HEAD
         REDEFINING
         SET OCCURRENCE SELECTION IS CURRENT OF SET,
END
```

Figure 4.14 A DDL Declaration of the PRES_ADMIN Subschema

The PRES_ADMIN subschema in Figure 4.14 is translated into a collection of declarations defining the abstract data types from the schema which are visible through the subschema, as shown in Figure 4.15. The specification NOT HIDDEN is an alternate syntax for the HIDDEN specification

described earlier and indicates that only those objects explicitly listed are visible to users of the subschema. All UWA_RECORD's and UWA_SET'S not explicitly listed are invisible to users.

The subschema is also used to define an alias for the identifier LASTNAME declared in the schema. The definition of an alias is a command to the compiler to create a second name for an object or data item. This alias definition may also supply a new data type to be associated with a data item. All references in the subschema are required to reference the object using the alias. A reference to the alias, PNAME, also requires a coercion to be performed.

The set ADMIN_HEADED in the schema is the concatenation of the properties of the UWA_SET and of the ADMIN_HEADED set type in the schema. The procedure attribute LOCATE of the UWA_SET implements the set occurrence selection criteria specified in the schema set type declaration. LOCATE is a VIRTUAL procedure and consequently is supplied an implementation in the declaration of the UWA_ADMIN_HEADED in the schema. Specification of a new set occurrence selection criteria for the ADMIN_HEADED set type in the subschema causes a new implementation of LOCATE to be generated. This new implementation of the VIRTUAL procedure is accessed at run-time by the data ma-

nipulation routines, since it is bound to the ADMIN_HEADED UWA_SET and replaces the implementation supplied in the schema. (See Figure 4.17. The dashed lines represent the environment definition from Figure 4.16). Details on the binding of VIRTUAL procedures were described in Section 3.4.

```
PRESIDENTIAL CLASS PRES_ADMIN;
BEGIN

     COMMENT Prefixing with PRESIDENTIAL concatenates
             the schema environment to the subschema.
             But HIDDEN specifications are used to
             restrict the environment made available
             through the subschema ;

     COMMENT New declarations are provided for
             PRESIDENT and for ADMIN HEADED.;

     NOT HIDDEN ALL_PRESIDENTS, PRESIDENT, ADMIN;
     UWA_PRESIDENT CLASS SUBSCHEMA_UWA_PRESIDENT;
     COMMENT Equivalence LASTNAME to PNAME;

     UWA_ADMIN_HEADED CLASS SUBSCHEMA_UWA_ADMIN HEADED;
     COMMENT Implement LOCATE to return the
             current of set, e.g. CURRENT;

END;
```

Figure 4.15 The PRES_ADMIN Subschema

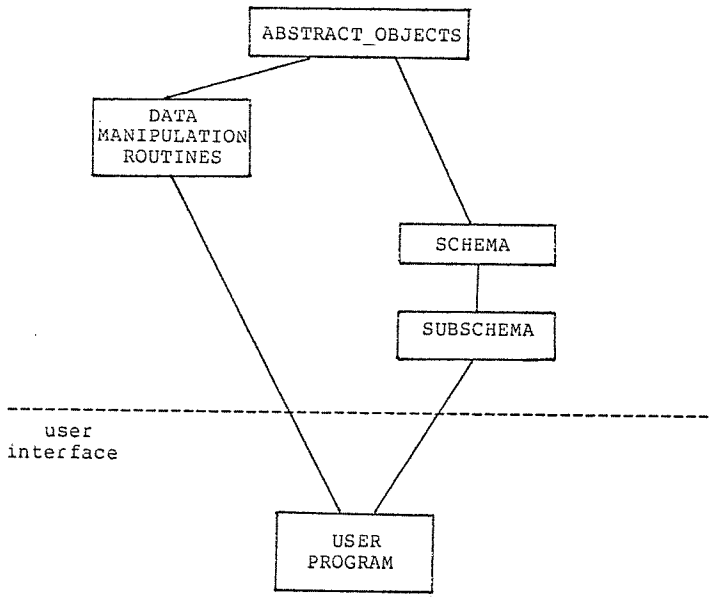


Figure 4.16 The Definiton of the User Environment

This structure represents the abstract data type model shown earlier in Figure 3.4 modified to conform to the name scope conventions of ALGOL 60. The user environment grows in the downward direction, similar to block nesting in ALGOL 60. But to support data independence HIDDEN specifications are used. Each component of the system

inherits an environment from the component above it. It then may either enhance this environment by declaring local identifiers and may pass its local environment to its descendants in its complete form or may restrict a part of the environment it passes by using a HIDDEN specification. The ability to declare the details of implementation as HIDDEN attributes at the subschema interface means that this structure directly supports data independence by isolating user programs from these details.

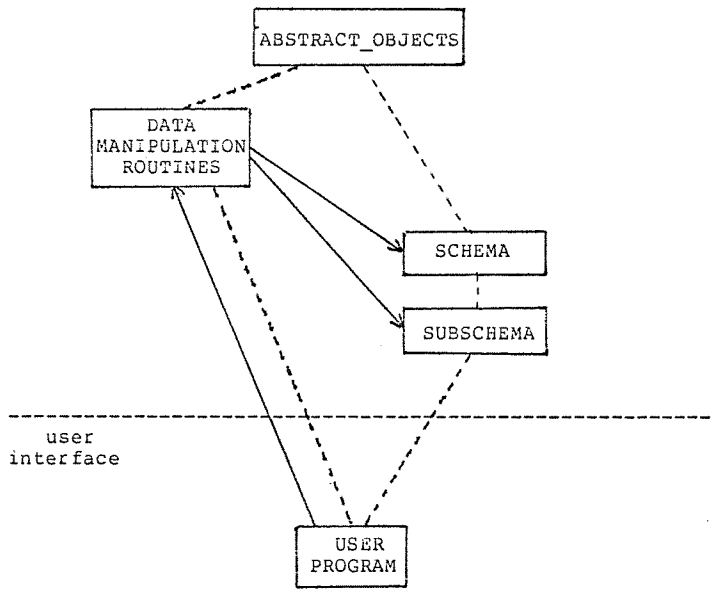


Figure 4.17 Procedure Calls to the Schema and Subschema

4.5 An Implementation

A database management system using the abstract data type model described in this Chapter has been implemented in SIMULA 67 on the Madison Academic Computing Center's UNIVAC 1110. As shown in Figure 4.18, the declarations of the MASS_STORAGE_RECORD, the UWA_RECORD, and the UWA_SET

described in Section 4.2 along with declarations of the data manipulation routines described in Section 4.3 are compiled to produce an object module. This object module, the compiled data manipulation routines, is database independent and is shared by all users of the system.

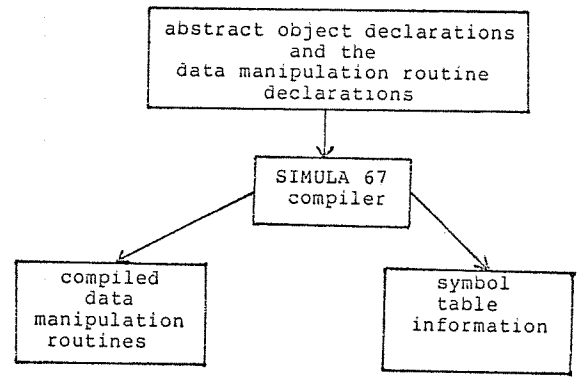


Figure 4.18 Compilation of the Abstract Objects and the Data Manipulation Routines

The grammar for the data definition language (DDL) presented in Appendix A was used to implement a syntax-directed compiler using a table-driven SLR(1) parser. This compiler accepts as input a schema definition written in DDL and generates as output the declaration of a collection of MASS_STORAGE_RECORDS, UWA_RECORDs and

UWA_SETs which correspond to the record types and set types in the schema. Each of the declarations in this collection contains a customized implementation of the VIRTUAL attributes of the corresponding abstract object which it implements. As an intermediate step these declarations are generated in SIMULA 67 and are then compiled as shown in Figure 4.18. Switches are implemented within the DDL compiler to enable it to automatically generate the simulation model used to study system performance in Chapter 5.

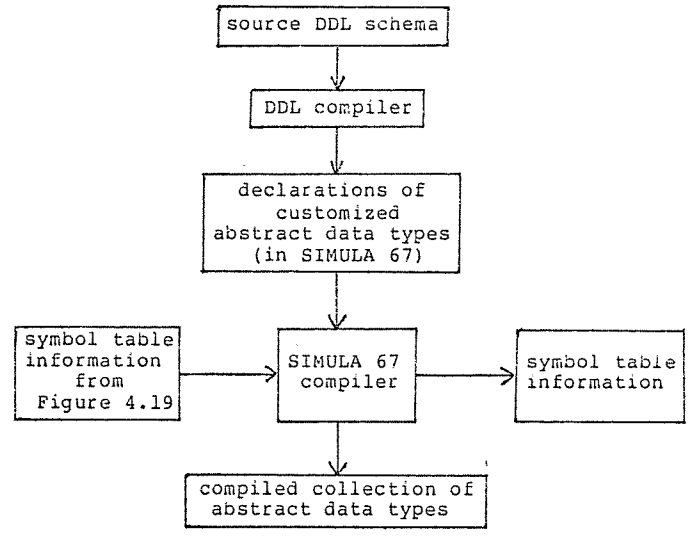


Figure 4.19 Compilation of the DDL Schema into a Collection of Abstract Data Types

In this way, a customized, database-dependent system is automatically generated. User programs are compiled and linked to this system for execution as shown in Figure 4.20.

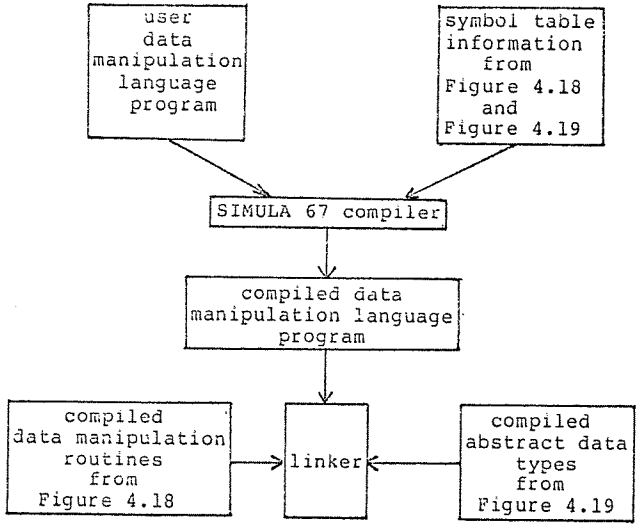


Figure 4.20 Translation and Linking of the User Program Containing Data Manipulation Routine Calls

CHAPTER 5

A PERFORMANCE ANALYSIS

The generic procedure model was introduced in Chapter 2. This model was extended in Chapter 3 to represent the schema and subschema as a shared collection of abstract data types. These abstract data types were generated using the generalizations of the network data model record and set which were defined in Chapter 4.

The advantages of representing the schema and subschema as abstract data types include increased reliability, improved support for data independence and for sharing the schema and the subschema, and increased run-time efficiency in contrast to the interpretive approach. The increased run-time efficiency is achieved by implementing the data manipulation routines as calls to the procedures associated with the abstract data types representing the schema and subschema. This eliminates the need for run-time interpretation and its associated

overhead. In many other programming environments substitution of compilation for interpretation significantly improves performance. In this chapter we will attempt to partially answer the question of whether the elimination of run-time interpretation has a significant impact on performance in the database environment.

This chapter characterizes the effect of interpretation overhead on system performance. Section 5.1 presents a performance model which is used in this analysis. Section 5.2 presents the workloads which are used in the analysis and examines the results of their execution. Section 5.3 analyzes these results to characterize the execution efficiency gained by representing the schema and subschema as a collection of abstract data types.

Performance analysis of database management systems has traditionally emphasized the analysis of input/output activity. The size of databases and the need for flexible sharing described earlier have required that the database be stored in secondary memory. The database is in a separate address space from the local address space of the user program. Users must execute explicit commands to transfer record instances between their local address space and the database. Since the database is stored in secondary memory, these commands are actually input/output instructions.

By contrast, the interpretation of the actual parameter descriptor is performed by the central processor. Since there is a large performance disparity between the access time of conventional storage devices and the cycle time of the central processor, interpretation may not be a significant overhead in the database environment.

The objective of this analysis is to characterize the run-time overhead required for interpretation. As a means of providing some qualitative measure of this overhead, the run-time interpretation cost will be contrasted with the corresponding input/output costs. For simplification we ignore the effects of virtual memory management by the operating system. Cost is defined to mean the overhead of system resource usage measured in seconds of time. Initially we examine the monoprogrammed case. The multiprogrammed case is discussed in Section 5.3.2.

Let C_{USER} be the cost to execute a user program. This cost has traditionally been assumed to be the cost of performing I/O to the database. In this analysis we show that the cost to perform interpretation, C_{INT}, is also a significant cost. Assuming other costs to be negligible, then

$$C_{USER} = C_{I/O} + C_{INT}$$

where $C_{I/O}$ is the cost to perform I/O to the database. The user response time is directly proportional to the system overhead which is measured by C_{USER}. Since in the database environment subsequent operations are dependent upon the results of the current operation, these costs are additive. In the monoprogrammed case the performance gain which may be achieved by overlapping I/O with interpretation is negligible.

The relative utilization factor, P, is a measure of the utilization of system resources required to perform interpretation relative to the resources used to perform I/O to the database. Since the user response time is directly related to the I/O and interpretation cost, measurement of P allows some qualitative predictions about user response time. For example, if P equals 1, then one-half the cost of executing a user program is the cost of performing I/O to the database and the remaining cost is interpretation, assuming all other costs are negligible. If P equals 1, then elimination of C_{INT} will reduce user response time by 50% and double system throughput, unless there are bottlenecks which are hidden by the interpretation overhead.

The objective is to measure

P = Relative Utilization

where

$$P = \frac{\text{cost to interpret the schema and subschema}}{\text{cost to perform I/O to the database}}$$

As described above, C_{INT} is the cost to interpret the schema and and $C_{I/O}$ is the cost to perform I/O to the database. Then

$$P = \frac{C_{INT}}{C_{I/O}}$$

The cost to perform I/O to the database is the product of the cost to transfer a single page, C_{PAGE}, and the number of pages which are transferred. If N_{PAGE} equals the number of pages transferred, then

$$C_{I/O} = C_{PAGE} * N_{PAGE}$$

The cost to interpret the schema and subschema consists of two components. The first component is the cost to locate and retrieve the required descriptors, C_{RET}. The second component is the cost to evaluate the descriptors once tney have been retrieved. Let C_{EVAL} be the cost to evaluate the descriptors. Then

$$C_{INT} = C_{RET} + C_{EVAL}$$

Assume that there is some average cost, I_{EVAL}, to interpret or evaluate a descriptor and that the number of descriptors which are evaluated is given by N_{EVAL}. Then

$$C_{EVAL} = N_{EVAL} * I_{EVAL}$$

The cost to locate the required descriptor if it is resident is negligible. However, if tne descriptor is not resident, a descriptor page fault is generated and an I/O operation must be performed to retrieve it. Assuming that M_{PAGE} is equal to the number of descriptor page faults and that the cost of transferring a descriptor page is the same as the cost of transferring a database page, C_{PAGE}, then

$$C_{RET} = M_{PAGE} * C_{PAGE}$$

and, thus

$$P = \frac{M_{PAGE} * C_{PAGE} + N_{EVAL} * I_{EVAL}}{N_{PAGE} * C_{PAGE}}$$

Assuming that there is a constant, k, which relates the cost of evaluating the descriptor to the cost of transferring a single page to mass storage, then

$$I_{EVAL} = K * C_{PAGE}$$

and

$$P = \frac{M_{PAGE} + K * N_{EVAL}}{N_{PAGE}}$$

5.1 A Performance Model

The relative utilization factor, P, can be measured by studying the memory reference patterns produced during the execution of user programs. Rodriguez-Rosell [Rod76] utilized memory reference behavior to empirically study the locality of database transactions in IMS. A database management system performs a number of the same functions as an operating system. One of these functions is managing a local buffer pool, which is effectively a private virtual memory, using techniques similar to those used in an operating system to manage virtual memory. This buffer pool is used for I/O to and from the database. As de-

scribed in Section 2.1, the basic unit of transfer between the database management system and mass storage is the physical page. A record reference is actually a reference to a group of locations within a page. A record request causes the buffer pool to be examined to determine whether the desired record is resident. If the record is missing, this generates a <u>database fault</u> and the page containing the requested record must be loaded from mass storage into the buffer pool. Rodriguez-Rosell used empirical measurement of database faults to study the locality of user programs.

In a system in which schema interpretation is employed, there are actually two address spaces, or virtual memories, which are managed by the database management system. The first is the buffer pool corresponding to the database address space in mass storage. The second is the buffer pool corresponding to the address space in which the encoded schema resides. By generating reference strings to these address spaces and analyzing the page faults which these reference strings produce, the overhead of I/O activity can be derived. By studying the reference strings generated by user programs, it is possible to determine the amount of interpretation performed by the data manipulation routines.

Since actual determination of these reference strings would involve a large overhead, a combination of simulation and modeling is used instead. A user program can be regarded as sequences of data manipulation routine calls separated by sequences of non-database operations. In order to measure P using simulation the following are required:

1. A method to generate semantically correct strings of data manipulation routine calls
2. A method to convert these strings of data manipulation routine calls into page references
3. A method to analyze these strings and to determine the number of I/O instructions and schema/subschema descriptor interpretations which they represent.

The basic approach to be employed is to replace the implementations of the procedures associated with the collection of UWA_RECORD and UWA_SET instances corresponding to the user schema. These new implementations simulate input/output operations to the database and are used to execute the user workload. For example, the procedure LOCATE associated with each UWA_RECORD instance is replaced with a new implementation which simulates retrieval of a record instance from the database. To simplify the analysis, the subschema is not considered. The simulation

schema monitors each point at which input/output is needed while simulating the results of the operation. It also monitors each point at which run-time interpretation would be required but is being avoided by our approach. The results of processing the user workload are counts of all record requests required to execute the workload and a count of the interpretation of record descriptors and set descriptors which would have been required in other systems but was avoided..

Based on these resource utilization values, the performance of an actual database management system will depend upon the configuration on which it executes. The relevant characteristics of the configuration are the processor cycle time and the I/O time to secondary storage. In order to simplify the analysis, all instructions are assumed to execute in one CPU cycle.

In order to study the effect of secondary storage I/O time on performance, two configurations will be used in this analysis. The first configuration, System A, is representative of current technology. It consists of a processor with a 1-microsecond cycle time and a conventional secondary storage system based on the IBM 3330 disc technology [Bre78]. C_{PAGE} is equal to the time required to transfer a single block, or page, from secondary storage to main memory. This time is the sum of the average ac-

cess time, or latency time, plus the time required to actually transfer the block. The average access time is independent of the block size and is assumed to be 35 milliseconds for System A. The time required to transfer a block is dependent upon both the transfer rate of the device and upon the block size. The transfer rate of the disc is assumed to be 800K bytes per second. Assuming a block size of 4K bytes, this results in a transfer time of 5 milliseconds per block. Thus for System A C_{PAGE} is equal to 40 milliseconds.

The second configuration, System B, is based on new memory technologies which reduce the gap between processor speed and mass storage speed. The processor is assumed to have a cycle time of 1 microsecond. To study the effect of mass storage performance on the relative utilization factor, P, System B is assumed to have a charge-coupled device (CCD) mass storage system with an access time of 100 microseconds and a transfer rate of 10M bytes per second [The78]. Based on these performance parameters and assuming a block size of 4K bytes, C_{PAGE} for System B is 500 microseconds.

It is realistic to anticipate that the development of new memory architectures, such as charge-coupled devices, bubble memories, and electron-beam-accessed memories, will have a significant impact upon system architecture [Mye76]

[Reg76] [Par76]. One expected impact is the development of memory hierarchies employing random-access memories, CCD's or bubble memories, and conventional disc memories. The effect of such a hierarchy will be a reduction in the I/O time required to access secondary storage. Given the cost/size ratio of CCD's and other new memory technologies, it is reasonable to expect this hierarchical memory system will be used for the file system. This will result in a merger of the techniques used to manage programs in main memory and to manage the file system. This merger will result in simpler operating systems and will also significantly reduce the time, or cost, to access the file system [Wen75]. This reduction in the I/O cost to the file system will have an impact upon the performance of database management systems. We analyze a portion of this impact in these simulations.

As described earlier, the cost of interpretation, C_{INT}, consists of two components. The first component is the cost to locate a descriptor. If the descriptor is resident in the system buffer pool, this cost is zero. If the descriptor is not resident, this cost is C_{PAGE}.

The second component of the interpretation cost is the cost to evaluate the descriptor. I_{EVAL} is the average processor interpretation time required to evaluate a descriptor after the descriptor has been retrieved. This

time is the time required to decode each component of the descriptor and to determine the operation which it specifies. For example, the encoded descriptor for the PRESIDENT record would state that the <LOCATION MODE> of the PRESIDENT record is CALC and that LASTNAME is to be used as a key. The descriptor also contains the location of the LASTNAME field within the UWA RECORD. Interpretation involves examining the descriptor, determining that the <LOCATION MODE> is CALC, and then determining the location of the key. The value of LASTNAME is then retrieved and passed to a procedure which computes a secondary storage address based on the key value. An estimate of the average interpretation time, I_{EVAL}, for both System A and System B is 20 microseconds.

M_{PAGE} is determined by simulation. In actual systems using interpretation, management of the schema and subschema descriptors varies. In one approach, all descriptors may be loaded when the database is opened. In this approach, M_{PAGE} is equal to the number of pages required to hold all the descriptors. Other systems store the descriptors in secondary storage. The data manipulation routines must retrieve descriptors as they are needed and manage the descriptor buffer space.

Both approaches are simulated by providing a descriptor buffer pool whose size is parameterized. Simulations

were performed with a varying percentage of the required descriptor buffer space available in this buffer. A demand fetch policy is simulated for loading descriptors into the buffer. If 100% of the required buffer space is available, then all descriptors are loaded only once. If less than 100% of th required space is available, then fetching and replacing descriptors are simulated. A least recently used replacement policy is used to replace descriptors in the descriptor buffer. This descriptor loading model is used to measure M_{PAGE}.

Since the number and the content of the schema descriptors are fixed, M_{PAGE} can be directly measured by the simulation. However, due to the dynamic nature of the size and the content of the database, N_{PAGE} can not be directly measured by the simulation. Instead N_{PAGE} is derived through a combination of simulation and modeling.

The simulation monitors the number of records requested while processing the workload. If the requested record is present in the database buffer pool, a record request will not generate a database fault. On the other hand, if the requested record is not present then a database fault occurs and the requested record (that is, the page which contains it) is transferred from secondary storage.

Berelian [Ber77] analyzed three different types of record requests and the associated probabilities that they will produce a page fault as follows:

Prob(1) = Probability of a database page fault when a record occurrence is accessed using a data item value as a key

Prob(2) = Probability of a database page fault when the next (or prior) member record occurrence of a set is accessed from the current member record occurrence

Prob(3) = Probability of a database page fault when the owner (member) record occurrence of a set is accessed from a member (the owner) record occurrence.

Berelian used analytic methods to derive expressions for each of these probabilites in terms of the buffer pool size, the average record length, the page size, etc.

The randomizing effect of hashing techniques used to implement retrieval by key value means that Prob(1) is not affected by the ability of the database management system to cluster related records. However, Prob(2) and Prob(3) are both reduced by using techniques which cluster records which are frequently accessed together.

Empirical data on the locality of database accesses is available from several sources. In the measurements

performed on IMS by Rodriguez-Rosell the page fault probability in the monoprogrammed case was in the range 0.80 < page fault probability < 0.95. Rodriguez-Rosell showed that in the multiprogrammed case there is high locality between concurrent user transations. In the concurrent case the page fault probability was less than 0.20.

Schkolnick [Sch77] measured similar values in a study of the effect of clustering records within hierarchical systems. Using measurements from IMS, Schkolnick measured page fault probabilities in the range of 0.10-0.20.

This empirical data does demonstrate dramatic reductions in page fault probabilities due to clustering of related records during traversal of chains within the database. These results are used to estimate the probability that a record request generates a page fault. We assume that clustering is used in set implementations and that the miss ratio is 20%.

The randomizing effect of CALCed access using a key value is in contradiction to the aims of clustering. Thus CALCed access should produce a higher number of page faults than traversing a chain within the database which exploits the effects of clustering. In the workloads presented in Section 5.2 less than 10% of the record requests are via a key value. To incorporate this effect, it will be assumed that:

N_{PAGE} = 1/3 * Number of record requests

As discussed earlier, efforts to study the performance of database management systems have concentrated on studying the input/output activity. Efforts to improve the performance have taken two directions. The first direction of research has attempted to optimize the number of actual input/output instructions needed to access a record requested by the user by optimizing the access methods. Typical of this reasearch is work by Cardenas [Car73][Car75] and Yao [Yao77] [YM77].

A very different approach has been taken by Berelian [Ber77] [Bi77] and Mitoma [Mit75] [MI75]. They postulate that the input/output activity due to a user program is based on the number of different record types which must be accessed to satisfy the user's request. Techniques such as integer programming are utilized to design an optimal schema. The schema is optimal in that the number of different record types required to satisfy the user's typical request is minimized.

These performance evaluation efforts have all been limited due to the complexity of the database environment. The performance of a database management system is dependent upon three factors:

1. The complexity of the database structure
2. The physical size of the database
3. The complexity of the system workload.

Since these three factors are interrelated in ways which are very difficult to characterize, the simulation approach we are using has several major advantages. By simulating the computations associated with instantiations of the UWA_RECORD and UWA_SET, the effect of the database structure is incorporated. We feel that earlier research, such as that by Mitoma, did not include these relationships, even though these relationships may have an effect on performance. For example, the LOCATION MODE of the STATE record in the PRESIDENTIAL database is CALC using STATENAME as a key. The SET OCCURRENCE SELECTION for the NATIVE_SONS set is VIA LOCATION MODE OF OWNER. When a new M_S_PRESIDENT record is to be inserted into an instance of the NATIVE-SONS set, the INSERT or STORE data manipulation routine invokes the LOCATE procedure of the NATIVE_SONS UWA_SET. NATIVE_SONS.LOCATE implements retrieving the correct owner occurrence by calling the LOCATE procedure associated with the STATE UWA record. STATE.LOCATE lo-

cates the correct STATE record instance by hashing on STATENAME. This process could have continued several levels if the the LOCATION MODE of the STATE record was VIA SET. Therefore, by simulating the execution of the procedures associated with the UWA_RECORD and UWA_SET the effects on performance of the relationships within the schema are explicitly measured.

The relationship between the database size and the workload complexity is not as completely specified but is introduced through Monte Carlo techniques. A user program can be regarded as consisting of sequences of data manipulation routines separated by sequences of non-database related operations. Measurement of the costs to execute the data manipulation routine calls is our objective. The number of data manipulation routine calls executed by a user program can be classified into one of three types. The first type of user program contains a constant number of data manipulation routine calls, that is, number of data manipulation routines called is not data dependent.

The number of data manipulation routine calls in the other two cases is data dependent. In the first of these cases, a data value is input or generated and is used to determine the number of data manipulation routine calls to be executed. For example, a user program may input the values of data items and then may STORE a new record in-

stance. This combined operation may be repeated until an end of file condition occurs on the input device.

The second data-dependent type of user program uses either an explicit value retrieved from the database or implicit information from the database, such as the size of a set occurrence, to determine the number of data manipulation routine calls to execute. For example, a user may FETCH all the PRESIDENT record instances which are members of an occurrence of NATIVE_SONS set. Thus this program is dependent upon the size of the set instance.

User programs whose resource usage is dependent upon values stored in the database or upon the size of the database are impossible to simulate using the techniques described above. Instead, to use this type of program as a workload for the simulations actual programs were traced during their execution. Monte Carlo techniques, parameterized in terms of the database size, were then used within synthetic programs derived from these traces to represent the resource usage of the actual programs. In this way the resource demands of programs which can not be simulated are used to produce synthetic workloads which can be used in the simulations.

5.2 The Workload

The workload for the simulations is a mix of synthetic programs based on applications which access the PRESIDENTIAL database. The purpose of a workload in performance evaluation is to provide a representative measure of the resource demands which are placed upon the system by the user workload. Our objective is to analyze the architecture of database management systems. Since an existing workload was not available, a workload which we believe is representative of the usage of database systems was generated. This workload consists of three synthetic programs. Each program is described in this section, along with the simulation results from its execution. The simulation results are then analyzed in Section 5.3 to evaluate the performance gains of the abstract data type model.

Two important synthetic programs can be derived from analyzing the network data model. The first is traversal of a many-to-many relationship and is examined in Section 5.2.1. The second is traversal of a cycle between several set occurrences and is presented in Section 5.2.2. A third syntnetic program is based on intial loading of the database. A program based on loading tne PRESIDENTIAL database is described in Section 5.2.3.

5.2.1 The Many-To-Many Relationship

The set is the basic component of database structure in the network data model. The complex structures of a database are tne result of interconnecting set structures. Using the set structure, two basic interconnections, or subcomponents, are possible. These subcomponents, which are structures built from sets, are the basic building blocks of the network data model.

The first structure is created by defining in the schema a single record type to be a member of two set types. Each set type has a different record type as owner (see Figure 5.1). This is the technique used in the network data model to define many-to-many relationships.

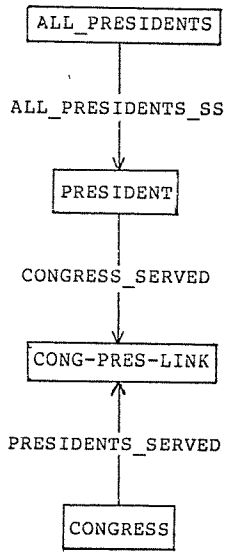


Figure 5.1 The Set Structure for the Many-to-Many Relationships

A program to determine the congresses which were headed by each president will traverse all occurrences of this structure in the database. Traversing this structure is performed by a sequence of data manipulation routine calls. The first call locates the instance of the

M_S_ALL_PRESIDENTS. The logical pointers of the ALL_PRESIDENTS_SS are traversed to move from M_S_ALL_PRESIDENTS to the first instance of M_S_PRESIDENT. The logical pointers of the CONGRESS SERVED set are then traversed to move from the M_S_PRESIDENT instance to a M_S_CONG_PRES_LINK instance, which is a member of both a CONGRESS_SERVED set occurrence and a PRESIDENT_SERVED set occurrence. The logical pointers of the PRESIDENTS_SERVED set occurrence are then traversed to locate the instance of the M_S_CONGRESS which is owner of the PRESIDENT_SERVED set occurrence. These operations are repeated for each instance of M_S_PRESIDENT (see Figure 5.2).

```
PRESIDENTIAL BEGIN
    INTEGER I,J,K,LAMBDA,MAXSETSIZE,L1,L2;
    REAL V1,V2,V3;
    OUTTEXT("SIMULATION OF RETRIEVAL OF ALL CONGRESSES");
    OUTTEXT(" SERVING ALL PRESIDENTS");
    OUTIMAGE;
    LAMBDA := ININT;
    MAXSETSIZE := 5.6*I;
    ALLPRESIDENTS.PRESIDENTS :-COPY("PRESIDENTS");
    FINDLOCNMODE(ALLPRESIDENTS);
    COMMENT while more presidents fetch the next
          president;
        L1 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
            COMMENT Random_value is one of three pseudo-
                random number generators:exponential,
                Gaussian or uniform  ;
    FOR J := 1,2 STEP 1 UNTIL L1 DO
    BEGIN
        FINDNEXT(PRESIDENT,ALLPRESIDENTSSS);
        GET(PRESIDENT);
        COMMENT while more cong_pres_link records
              get the owner;

        L2 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
        FOR K := 1 STEP 1 UNTIL L2 DO
        BEGIN
            FINDNEXT(CONG_PRES_LINK,CONGRESS_SERVED);
            GET(CONG PRES LINK);
            CONGRESSSERVED.SUPPRESS := TRUE;
            FINDOWNER(CONG_PRES_LINK,PRESIDENT_SERVED);
            GET(CONGRESS);
        END;
    END;
EXIT: END OF DML;
```

Figure 5.2 A Program to Traverse the Many-to-Many Structure

This program simulates locating all members of a set occurrence. Monte Carlo techniques were used to determine the number of times each subsequence of data manipulation routines is executed. This number represents the set size, or the number of members participating in a set occurrence. The simulated values of set size were computed using three different pseudo-random generators.

The first pseudo-random number generating function assumes that the sizes of set instances are exponentially distributed with a rate, or intensity, equal to LAMBDA. The distribution is terminated so that the probability of a set size greater than MAXSETSIZE is zero. The relationship between LAMBDA and MAXSETSIZE was derived by modeling the relationship between the average set size and the maximum set size of set occurrences within the PRESIDENTIAL database.

The second pseudo-random number generating funtion is Gaussian, with a mean of LAMBDA and a standard deviation of twice the MAXSETSIZE. This distribution was chosen so that 1% of the values returned by the density function were less than zero and 1% were greater than MAXSETSIZE. All values less than zero and greater than MAXSETSIZE were rejected.

The third and final pseudo-random number generating function is a uniform distribution. The minimum set size is zero and the maximum is MAXSETSIZE.

The program shown in Figure 5.3 was executed with each of these pseudo-random number generating functions. The three pseudo-random number generating functions showed only a weak dependence of the relative utilization upon average set size. However, the three distributions converge to the same value for large values of LAMBDA. For this reason only the exponential probability function was used to generate pseudo-random numbers for the remaining simulations. The results of the simulations using the exponential distribution are shown in Figure 5.3 and Figure 5.4 for System A and System B, respectively. The ordinate, the relative utilization, represents the ratio of the cost to interpret the schema descriptors to the cost of performing I/O to the database. The abscissa, the average set size, is the average number of member record instances in a set occurrence.

Each figure shows three plots of the relative utilization, P, versus the average set size, LAMBDA. Each curve represents a simulation in which a different size schema descriptor buffer pool was available. The upper curve was produced by providing space for only a single schema descriptor in the buffer pool. Each transition to

interpret a different schema descriptor results in an I/O operation to retrieve the required descriptor.

The middle curve in all the figures was produced by providing a schema descriptor buffer pool whose size approximates the working set size for schema descriptors. At any point during execution, the data manipulation routines require some subset of the schema descriptors to be resident in order to avoid excessive I/O activity to retrieve descriptors. The size of this subset is the working set size. Measurement of this working set size is described in Section 5.3. As changes in the working set of the schema descriptors occur, I/O operations are executed to load the new working set using a demand fetch policy. The I/O activity to retrieve descriptors reflected in this curve is a measure of these transitions in locality.

The bottom curve in each figure was produced by providing a descriptor buffer pool which is large enough to contain all of the schema descriptors required to execute a user program. Using a demand fetch policy, each descriptor is retrieved the first time it is referenced and then it remains resident. In this case, essentially all interpretation overhead is CPU activity. Thus, P is a measure of the CPU overhead to evaluate descriptors relative to the overhead to perform I/O to access records in the database.

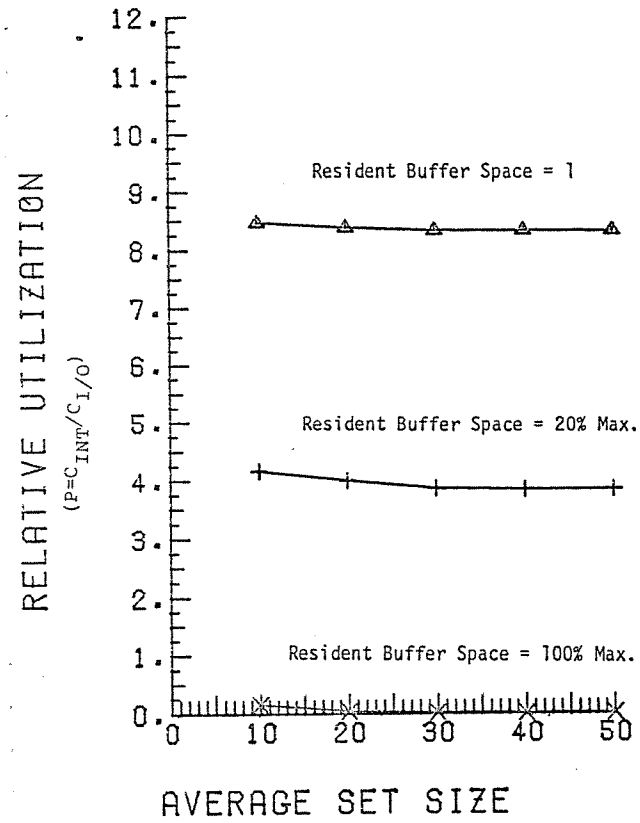


Figure 5.3 Traversing the Many-to-Many Structure. Exponential Distribution: System A

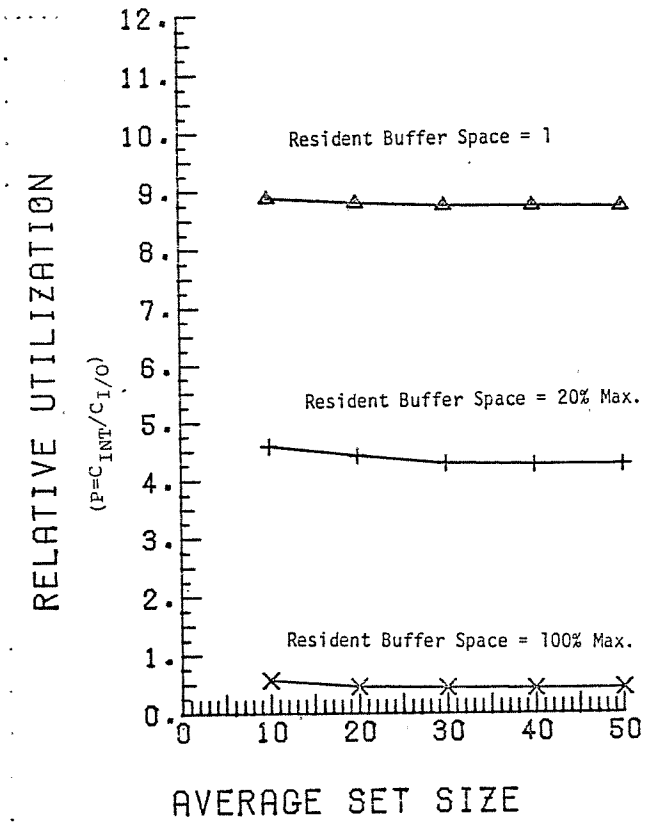


Figure 5.4 Traversing the Many-to-Many Structure. Exponential Distribution: System B

5.2.2 The Cyclic Relationship

A second major substructure in the network data model is created by defining set types in such a way that the set instances form a cycle. An example from the PRESIDENTIAL database is shown in Figure 5.5.

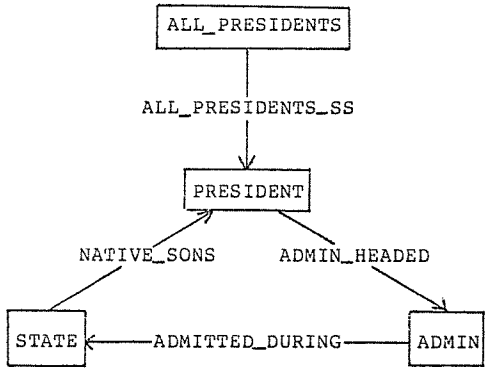


Figure 5.5 An Example of the Cyclic Set Structure in the PRESIDENTIAL Database

Traversing this structure is performed by first locating M_S_ALL_PRESIDENTS. FINDNEXT is then called to locate M_S PRESIDENT. FINDNEXT is then invoked to move to the first M_S_ADMIN by traversing the logical pointers of the ADMIN HEADED set type. FINDNEXT is again invoked to

move to an M_S_STATE instance by traversing the pointers of the ADMITTED_DURING set type and to move subsequently to an M_S PRESIDENT instance along the pointers of the NATIVE SONS set type (see Figure 5.6). These operations were repeated for all instances of M_S_PRESIDENT. The results of executing this program are shown in Figure 5.7 and Figure 5.8.

```
PRESIDENTIAL BEGIN
    INTEGER I1,I2,J,K,LAMBDA,MAXSETSIZE,L1,L2,L3,L4;
    OUTTEXT("SIMULATION OF TRAVERSING CYCLE ");
    OUTIMAGE;
    LAMBDA := ININT;
    MAXSETSIZE := 5.6*LAMBDA;
    ALL PRESIDENTS.PRESIDENTS :-COPY("PRESIDENTS");
    FINDLOCNMODE(ALL PRESIDENTS);
    COMMENT while more presidents fetch the
        next president ;
    L1 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
    FOR J := 1,2 STEP 1 UNTIL L1 DO
    BEGIN
        FINDNEXT(PRESIDENT,ALL_PRESIDENTS SS);
        GET(PRESIDENT);
        COMMENT while more admin records get
            the next president ; .
        L2 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
        FOR K := 1,2 STEP 1 UNTIL L2 DO
        BEGIN
            ALL_PRESIDENTS SS.SUPPRESS:=TRUE;
            FINDNEXT(ADMIN,ADMIN HEADED);
            COMMENT while more states find the
                next state ;
            L3 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
            FOR I1 := 1,2 STEP 1 UNTIL L3 DO
            BEGIN
                ADMIN_HEADED.SUPPRESS := TRUE;
                FINDNEXT(STATE,ADMITTED_DURING);
                COMMENT while more presidents,
                    find the next president;
                L4 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
                FOR I2 := 1,2 STEP 1 UNTIL L4 DO
                BEGIN
                    ADMITTED DURING.SUPPRESS:= TRUE;
                    ALL PRESIDENTS SS.SUPPRESS := TRUE;
                    FINDNEXT(PRESIDENT,NATIVE_SONS);
                    GET(PRESIDENT);
                END;
            END;
        END;
    END;
EXIT:END;
```

Figure 5.6 A Program to Traverse the Cyclic Structure

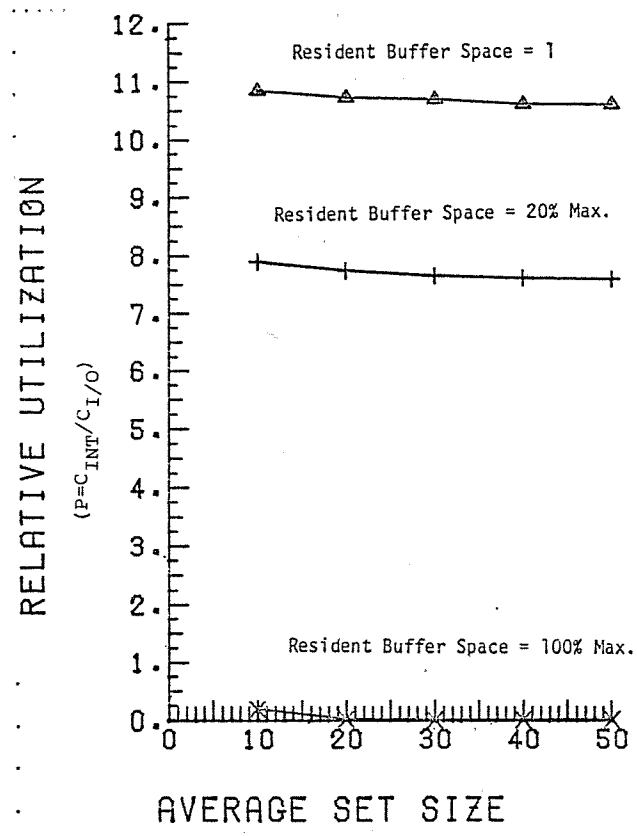


Figure 5.7 Traversing the Cyclic Structure. Exponential Distribution: System A

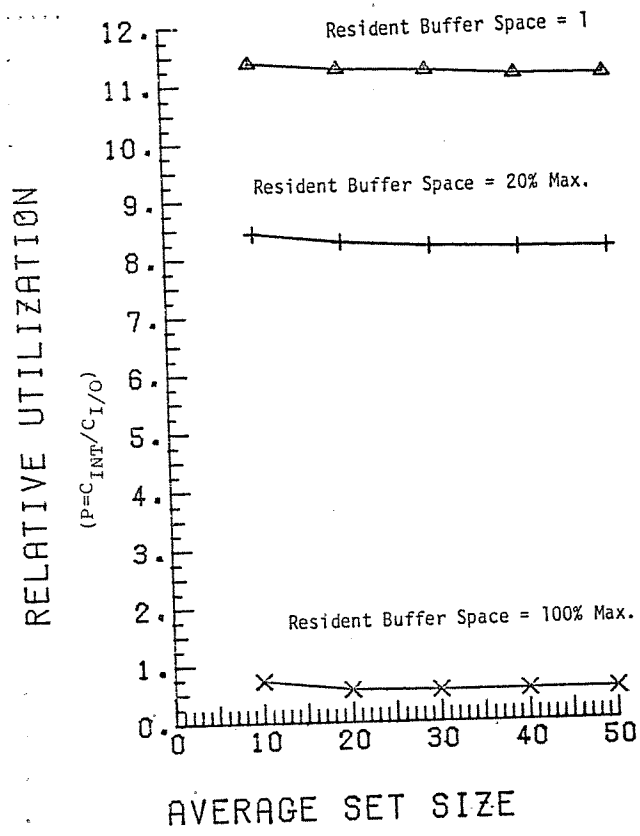


Figure 5.8 Traversing the Cyclic Structure. Exponential Distribution: System B

The relative utilizations shown in these figures are almost identical to those shown earlier in Section 5.2.1 for traversing the many-to-many structure. The explanation for this is that the two user programs actually differ only by the selection of the pointers which are used to traverse the database. In one case a pointer to the set owner record instance is followed; in the other a pointer to the next member is followed. The interpretation required in both these cases is very similar and hence the results are very similar.

5.2.3 Initial Loading of the Presidential Database

The third synthetic program was derived by tracing a program which loaded the PRESIDENTIAL database. This trace was used to determine the semantically valid subsequences of data manipulation routine calls which were executed during the initial creation of the database. For example, a call to GET is only semantically correct following a call to FINDNEXT, FINDLOCNMODE, etc. The number of times these subsequences are executed during the simulations is determined by generating pseudo-random numbers which represent the set size as discussed earlier. The synthetic program is shown in Figure 5.9 and the results are shown in Figure 5.10 and Figure 5.11.

```
PRESIDENTIAL BEGIN
    INTEGER I,J,K,LAMBDA,MAXSETSIZE,L1,L2;
    OUTTEXT("SIMULATION OF DATABASE LOADING");
    OUTIMAGE;
    LAMBDA := ININT;
    MAXSETSIZE := 5.6 * LAMBDA;
    STORE(ALL_PRESIDENTS);
    STORE(ALL_STATES);
    STORE(ALL_CONG_PRES_LINKS);
    STORE(ALL_ELECTIONS);
    L1 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
    FOR I := 1 STEP 1 UNTIL L1 DO
        STORE(STATE);
    L1 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
    FOR I := 1 STEP 1 UNTIL L1 DO
    BEGIN
        STORE(PRESIDENT);
        L2 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
        FOR J := 1 STEP 1 UNTIL L2 DO
            STORE(CONG_PRES_LINK);
    END;
    L1 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
    FOR I := 1 STEP 1 UNTIL L1 DO
    BEGIN
        STORE(ADMIN);
        L2 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
        FOR J := 1 STEP 1 UNTIL L2 DO
        BEGIN
            ADMITTED_DURING.SUPPRESS := TRUE;
            FINDLOCNMODE(STATE);
            INSERT(STATE,ADMITTED_DURING);
        END;
    END;
```

Figure 5.9 A Program to Load the PRESIDENTIAL Database


```
    L1 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
    FOR I := 1 STEP 1 UNTIL L1 DO
        STORE(ELECTION);
    L1 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
    FOR I := 1 STEP 1 UNTIL L1 DO
        STORE(CONGRESS);
    L1 := RANDOM_VALUE(LAMBDA,MAXSETSIZE);
    FOR I := 1 STEP 1 UNTIL L1 DO
    BEGIN
        FINDNEXT(CONG_PRES_LINK,ALL_CONG_PRES_LINKS_SS);
        GET(CONG_PRES_LINK);
        FINDLOCNMODE(CONGRESS);
        PRESIDENT_SERVED.SUPPRESS := TRUE;
        FINDCURRENT(CONGRESS_PRES_LINK);
        INSERT(CONG_PRES_LINK,PRESIDENT_SERVED);
    END;
EXIT:END;
```

Figure 5.9 A Program to Load the PRESIDENTIAL Database (continued)

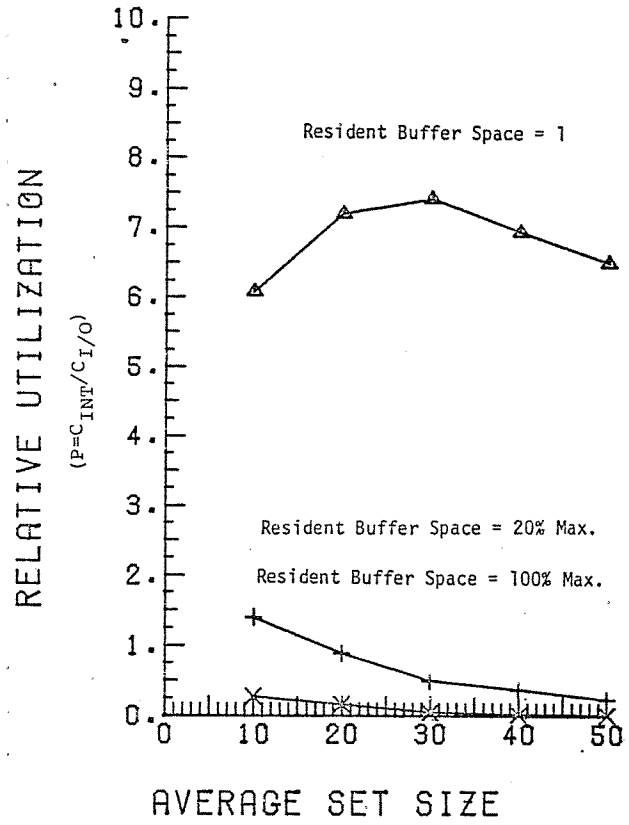


Figure 5.10 Loading the PRESIDENTIAL Database.
Exponential Distribution: System A

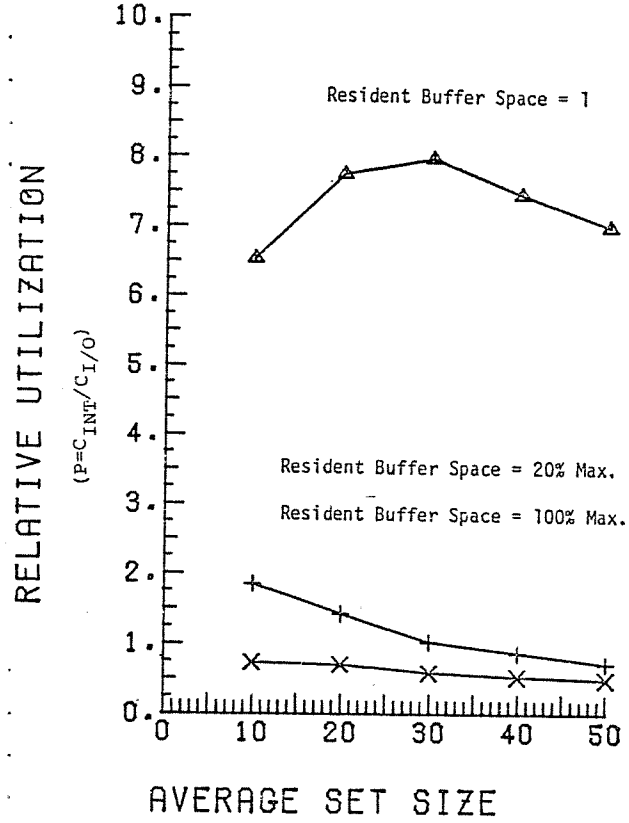


Figure 5.11 Loading the PRESIDENTIAL Database.
Exponential Distribution: System B

5.3 Conclusions

The simulations discussed in Section 5.2 provide a large amount of information about the performance of a database management system. One of the major results is a measurement of the frequency of interpretation which is described in Section 5.3.1. The simulation results also provide information about the effect of multiprogramming upon system performance. These results are discussed in Section 5.3.2. Additional costs related to system performance are discussed in Section 5.3.3.

5.3.1 An Evaluation of the Cost of Interpretation

One of the major results of the simulations described in Section 5.2 is a measurement of the cost of interpretation in database management systems. Averaged over all of the simulations, interpretation occurs approximately 3.5 times per record access. Given the assumptions described earlier, this implies 10 interpretations of schema descriptors per I/O operation to the database. The simulation model introduced significant simplifications and additional requirements for interpretation of the schema were ignored, such as interpretation of authorization and privacy information. Interpretation of the subschema was

not considered. We believe that it is reasonable to expect the ratio of interpretation to I/O is even higher in actual systems than was shown by these simulations.

Given the CPU cycle time of 1 microsecond assumed for both System A and System B, these measurements showed a lower bound of approximately 150 microseconds of CPU processing per I/O operation. For System A this overhead is small in terms of the cost of the average I/O time of 40 milliseconds per block. However, for System B, in which an average I/O time of 500 microseconds is assumed, the interpretation overhead is significant in comparison to the cost of performing I/O to the database.

In the simulations where no retrieval of schema descriptors was required, all the interpretation overhead was CPU overhead. In this case the interpretation cost was approximately one-half the I/O cost to the database. As described earlier, this value indicates that user response time will decrease by 30% if interpretation is eliminated and if all other sources of system overhead are negligible, as we have assumed. The gain in response time which is achieved by eliminating interpretation will be even larger if retrieval of schema descriptors is required. In the cases where only a working set of schema descriptors could be resident, a value of 3-4 was measured for P. This indicates that only 25% of the system over-

head is due to I/O to retrieve records requested by the user. The remaining overhead is due to the retrieval and evaluation of schema descriptors.

We believe that the abstract data type model presented in this dissertation would provide significant advantages in the future by eliminating the run-time overhead of interpretation. The implication of these simulation results is that replacement of slower mass storage devices with higher speed devices may alleviate a current bottleneck in terms of system performance. It should be expected that higher speed storage devices would increase system throughput. However, the gains in system performance will be limited by the overhead of performing interpretation. This overhead will seriously impact the performance of future systems unless the architecture of database management systems is modified.

This discusssion has not considered the effect of a reduction in CPU cycle time. Processors are currently available with cycle times significantly lower than the value of 1 microsecond we have assumed, and it is plausible to expect the development of general-purpose processors with a 10 nanosecond processor cycle time.

While this gain in speed will improve system performance, the performance gains it will provide are not as significant as those which can be achieved by improving

memory performance. System B achieved significant performance improvements over System A simply by replacing the mass storage system with a newer, faster technology device. Hierarchical memories are being developed whose performance characteristics approximate those of random-access memory by employing caches, random-access memory, CCD's or bubble memories, and conventional discs [Reg76] [Pan76] [Pan77]. If this memory performance is achieved, it will significantly reduce the performance gap between processor times and mass storage access times.

With a memory hierarchy elimination of interpretation is even more significant. Multiprogramming is used to overcome the current disparity between processor speed and I/O speed. If this disparity is significantly reduced, it may no longer be necessary to use multiprogramming unless timesharing must be supported. Instead, it will be feasible to wait for an I/O operation to complete rather than switching the CPU to another process. With the elimination of multiprogramming a CPU scheduling discipline such as FIRST-COME-FIRST-SERVED is a natural choice. Using this scheduling discipline, the average user waiting time, the variance of response time, and the system throughput are proportional to the processing requirements of the user jobs. Elimination of

run-time interpretation is then an obvious technique to increase the system performance.

The interpretation overhead discussed above is completely avoided when the schema is represented by a collection of abstract data types. In contrast, two different sources of overhead are present. The first is the use of indirect addressing to reference the correct data attribute or procedure attribute within the schema. This overhead is insignificant compared to the overhead of interpretation.

The second overhead which is present at run-time is much more significant. By implementing the data manipulation routines in terms of the procedure attributes of the abstract data types representing the schema, each reference to the schema involves a context switch. Given the overhead of procedure entry and exit, the number of procedure calls which are executed by each data manipulation routine could involve an overhead which is comparable to the CPU time required to perform interpretation.

However, several techniques are available to reduce this overhead. The first approach to this problem employs techniques similar to the WITH statement in PASCAL and the INSPECT statement in SIMULA 67. This approach would allow the context of the abstract data types referenced within the data manipulation routine to be bound once when the

data manipulation routine is entered, rather than being bound each time the abstract data type was referenced within the body of the data manipulation routine.

The second approach replaces the data manipulation routines described in Section 4.4 with a set of routines which only manage concurrent access to the database. Rather than compiling the data manipulation routines as closed subroutines as was done in Section 4.4, each data manipulation routine is used as the definition of an open subroutine. Each user data manipulation routine call in the user program is replaced by the body of the open subroutine. In this way, the context switching between the user program and the data manipulation routines is avoided.

However, this approach involves some risk. The data manipulation routines represent a different protection domain than the user program. By treating the data manipulation routines as open subroutines the access rights of the data manipulation routines are introduced into the user program. Protection facilities must be used to prevent the user program from maliciously or erroneously gaining use of these access rights.

5.3.2 The Effect of Multiprogramming

Another major result which can be derived from the simulations is that the ability or the inability to maintain the required schema descriptors resident in memory has a significant impact upon the performance of systems using interpretation. Whereas the studies by Rodriguez-Rosell demonstrated weak locality in database references within a single user transaction, our simulations show a strong locality in references to schema descriptors within a single user program, as shown in Figure 5.12. This data was derived from analyzing the execution of synthetic programs described in Section 5.2.1. The fraction of schema descriptors available refers to the number of schema descriptors which can be resident in the system buffers relative to the total number of descriptors in the schema. The I/O activity is normalized to the I/O rate when only one descriptor may be resident.

The cost of interpretation is composed of two components. The first is the cost to locate the desired descriptor. The second is the cost to evaluate the descriptor. If the desired descriptor is not resident in the system buffer, locating the descriptor involves an I/O operation to mass storage.

The data shown in Figure 5.12 relates the need to retrieve descriptors to the size of the system descriptor buffer pool relative to the total number of descriptors in the schema. This data demonstrates that as the relative size of the schema buffer pool increases, the I/O cost to retrieve descriptors drops significantly. The data shown in Figure 5.12 shows a strong locality in references to schema descriptors in the monoprogrammed case. This locality is very similar to the locality seen in virtual memory systems. If some minimum fraction of descriptor buffer space is available, the I/O activity drops to a relatively constant level for all higher fractions of buffer space. This implies that if a working set of schema descriptors can be maintained resident in memory, performance will be improved by reducing the overhead of interpretation to mainly CPU overhead.

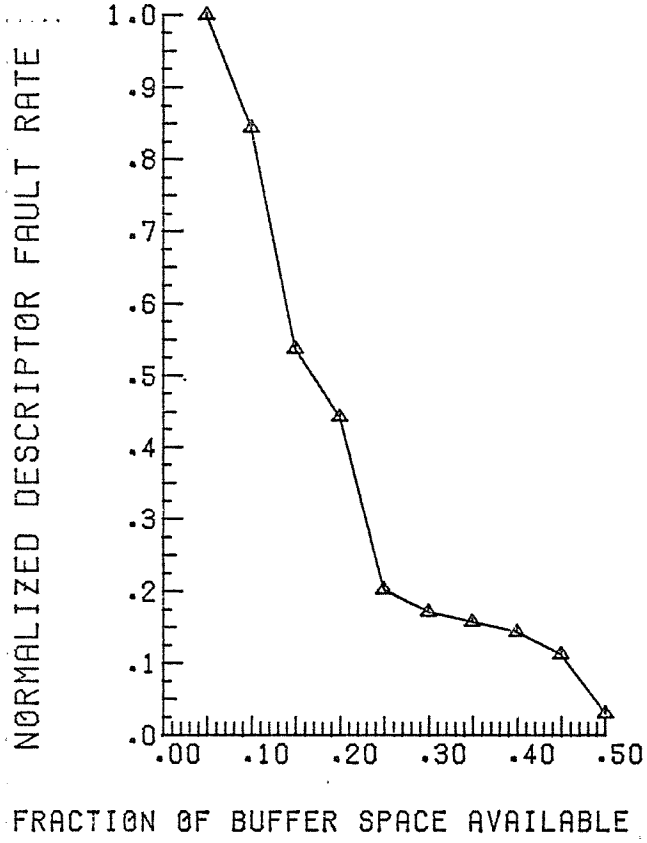


Figure 5.12 Schema Descriptor I/O Versus the Size of the Available Buffer Space

Just as in the case of virtual memory systems within operating systems, this locality implies that management of schema descriptors in the multiprogrammed case will have a significant impact upon performance. For a small schema, that is, a schema with a small number of record types and set types, it may be reasonable to load the entire schema into memory when the database is opened. But for a large schema, that is, a schema with many record types and set types, this may not be possible. If concurrent users are accessing disjoint subsets of the database, then contention for schema buffers and the resulting increased rate of schema descriptor faults will significantly impact the system performance.

The simulations described in Section 5.2.1 and Section 5.2.3 were modified to study the effect of multiprogramming upon system performance. A mix of the synthetic programs described earlier was executed concurrently, with each synthetic program generating data manipulation routine calls with a Poisson interarrival time. This mix of synthetic programs was then simulated in concurrent execution using the techniques described earlier. Based on our assumption that the cost of interpretation and the cost of I/O are additive, these simulations show a very strong impact of multiprogramming upon the relative utilization. This increase is due to the increased rate

of schema descriptor faults as contention for schema buffers increases. Figure 5.13 and Figure 5.14 show the relative utilization as a function of the multiprogramming level.

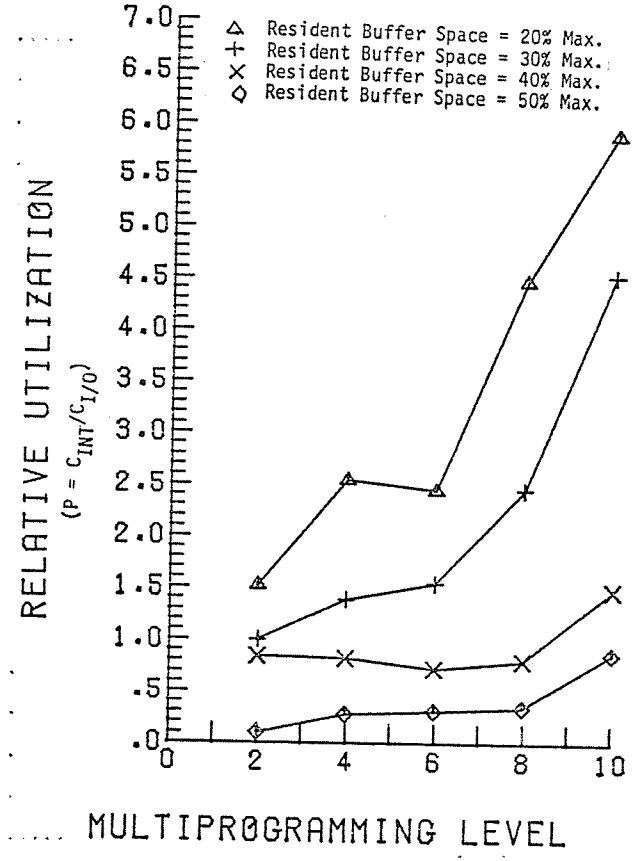


Figure 5.13 The Relative Utilization as a Function of the Multiprogramming Level (System A: No CPU-I/O Overlap)

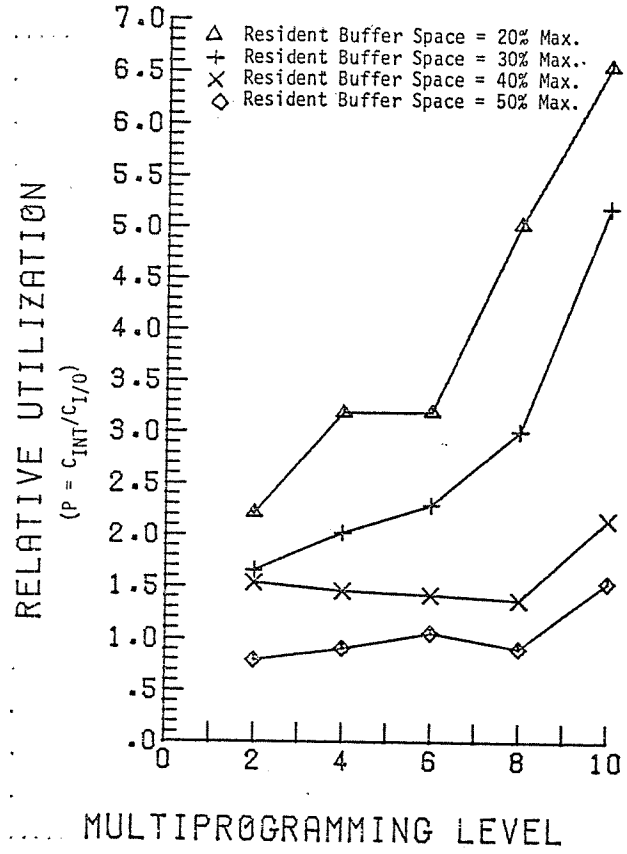


Figure 5.14 The Relative Utilization as a Function of the Multiprogramming Level (System B: No CPU-I/O Overlap)

While overlapping CPU usage with I/O does not improve performance in the monoprogrammed case, it should improve performance in the multiprogrammed case. To establish the maximum degree of this improvement, complete CPU-I/O overlap was assumed. Based on this assumption, C_{INT} is now defined to be the greater of C_{RET} and C_{EVAL}. Figure 5.15 and Figure 5.16 show the relative utilization when complete overlapping of the CPU and of I/O is assumed for System A and System B, respectively. Each figure shows four curves which were produced by varying the size of the schema descriptor buffer pool relative to the total number of descriptrs in the schema. Figure 5.15 snows that for System A, no significant change in the relative utilization, P, is gained by overlapping evaluation of descriptors with retrieval of descriptors. This shows that the system is I/O bound. In constrast, the values of the relative utilization shown in Figure 5.16 do show a change for System B when overlapping of evaluation and retrieval is assumed. This is due to the closer balance between the CPU cycle time and the I/O time and indicates that the assumption that the cost of evluation and the cost of retrieval are additive is not valid for System B. In tne lowest curve shown in this figure, C_{INT} equals C_{RET} for several points while for other points C_{INT} equals C_{EVAL}. Furtner research should be conducted to analyze the effect

of multiprogramming on database management system performance.

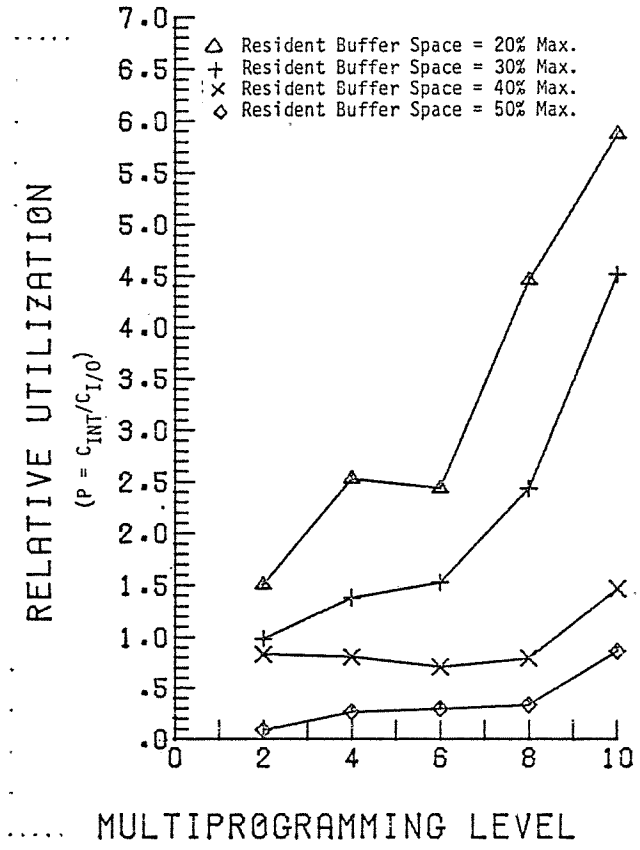


Figure 5.15 The Relative Utilization as a Function of the Multiprogramming Level (System A: Complete CPU-I/O Overlap)

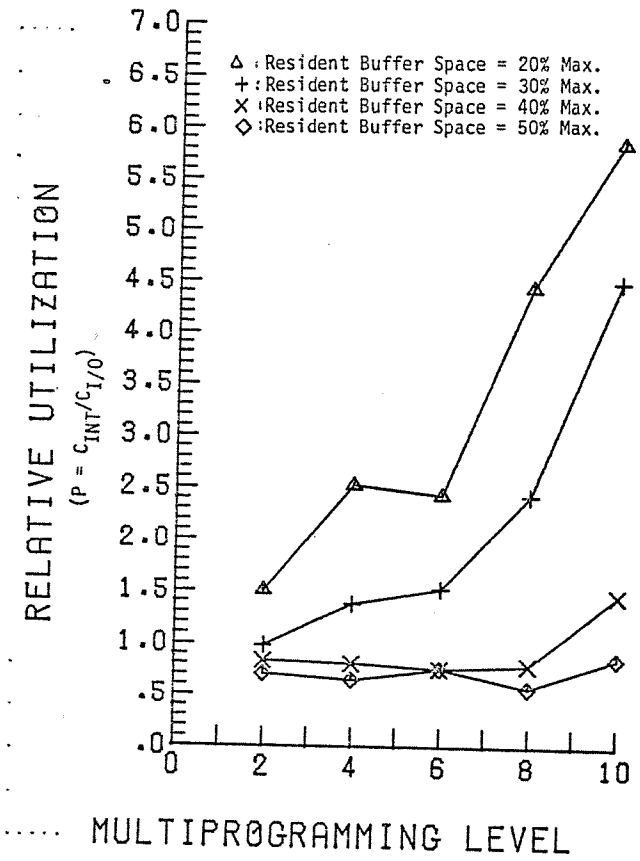


Figure 5.16 The Relative Utilization as a Function of the Multiprogramming Level (System B: Complete CPU-I/O Overlap)

These figures demonstrate that the schema descriptor buffers become a limiting resource when multiprogramming is introduced. As contention for the buffers increases, the system overhead increases dramatically. This phenomenon is similar to thrashing in virtual memory systems. In systems in which it is not possible to maintain all schema descriptors resident in memory, it demonstrates the dramatic performance gains which elimination of interpretation in the abstract data type model will provide.

5.3.3 Other Performance Costs

The interpretation and I/O costs studied above are the two major run-time costs in a database management system. A number of other costs are examined in this section, including factors such as compilation cost, program size, and program locality.

The run-time overhead of the interpretive approach is avoided by binding more information at compile-time and load-time. Two equivalent programs were written to execute with the PRESIDENTIAL database. One program was compiled and executed using DMS 1100. The other program was compiled using SIMULA 67 and executed using the simulation techniques described in Section 5.1. The cost to translate the PRESIDENTIAL schema using DMS 1100 was approxi-

mately one-half the cost to translate the schema into a collection of abstract data types and to compile these declarations. The run-time size of the linked data manipulation routines and the user program was equivalent in both cases. The compiled schema represented by abstract data types contains only those features of the network data model which are actually used in the source schema. By contrast, the data manipulation routines in the interpretive approach must implement code to handle all options possible within the network data model. While the compiled schema described above does not contain a number of features implemented by DMS 1100, we believe that use of code optimization and of library procedures to reduce duplicated code would enable the compiled approach to remain comparable in size to the size of a system using interpretation.

An important measure of program behavior is locality, particularly in a virtual memory system. It is unclear what effect representing the schema as a collection of abstract data types has on locality. Further measurements beyond the scope of this dissertation would be required to answer this question.

Representation of the schema by a collection of abstract data types eliminates the need for the data manipulation routines to manage a buffer pool of schema descrip-

tors. This eliminates a significant source of run-time overhead and should result in a simplification of the data manipulation routines.

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH

6.1 Conclusions

This dissertation has examined the application of current research on abstract data types and on generic procedures to the implementation of a network model database management system. The data manipulation routines are examples of generic procedures. The information defining the characteristics of each record type and set type is contained in the schema and subschema. The data manipulation routines utilize this information to determine the functions which are to be performed for a given actual parameter. A generic procedure model of a database management system was presented and used to describe the techniques used in current systems to provide this information to the data manipulation routines.

The generic procedure model was extended to represent the schema and subschema as a shared collection of abstract data types. In this abstract data type model the schema and subschema have been converted from their passive role in the generic procedure role to an active role. In the abstract data type model the data manipulation routines are now defined completely in terms of the data attributes and procedures associated with the abstract data types.

The abstract data types representing the schema and subschema are generated by extending the concept of generalization developed by Smith and Smith. Generalization is used to define three generic objects which represent the network data model: the MASS_STORAGE_RECORD, the UWA_RECORD, and the UWA_SET. Using these generic objects as templates, the user schema and subschema are translated into the equivalent declarations of a collection of abstract data types which are implementations of the generic objects. Using environment concatenation, this collection of abstract data types is bound to the user program and to the data manipulation routines at <u>load-time</u>.

The actual parameters to the data manipulation routines are used to indirectly reference at <u>run-time</u> the correct data attribute or procedure attribute associated with an abstract data type instance in the schema or

subschema. When coupled with environment control mechanisms based on SIMULA 67, the abstract data type model was shown to directly support data independence, sharing of the schema and subschema, and access control to the schema through the subschema.

A commonly used implementation technique in current systems is interpretation of the actual parameter descriptors at run-time by the data manipulaion routines. A simulation model was developed to compare the abstract data type model with the interpretive approach. A combination of simulation and analytic modeling was used to determine the impact of interpretation upon system performance. The cost of interpretation was shown to consist of two components: The first is the cost to locate the correct descriptor; the second is the cost to evaluate the descriptor.

The simulations measured both the frequency of interpretation to evaluate descriptors relative to record requests to the database and the frequency of retrieving schema descriptors from mass storage due to descriptor faults. Given the assumptions of the simulation model, it was shown that schema descriptor evaluation occurred approximately 3.5 times per record request to the database. Due to the simplifications introduced by our simulation

model, we believe that the ratio of interpretation to record requests is even higher in actual systems.

The simulations also demonstrated that within a single user program a strong locality exists in references to schema descriptors. The ability or inability to maintain a working set of schema descriptors resident in memory has a significant impact upon the performance of systems using interpretation, particularly as the level of multiprogramming increases. A working set of descriptors is required to maintain adequate system performance. Since concurrent users may be accessing disjoint subsets of the schema, the working set size in the multiprogrammed case may be larger than in the monoprogrammed case. For a large schema with many record types and set types it may not be possible to maintain a working set of descriptors resident in memory, and hence a degradation in performance will occur.

The simulation model was also used to study the effect of mass storage I/O time on system performance. Using current memory technology, the major cost of interpretation is the cost to retrieve descriptors. While replacement of current mass storage devices may alleviate a bottleneck in current systems, the gains in system performance will be limited by the overhead of descriptor evaluation.

6.2 Future Research

Several areas of research related to the abstract data type model are of interest. The first is a study of program locality in the abstract data type model. A memory reference model was introduced in Chapter 5 and was used to evaluate the performance of a system implemented using abstract data types versus a system implemented using an interpretive approach. The user program, the schema, and the data manipulation routines were treated as three separate address spaces, or segments. The simulations measured the total number of references to each of these segments. This model does not provide enough information about the frequency of transitions between these segments to adequately study the program locality in a system implemented using abstract data types.

A second area for future research is the application of the abstract data type model to distributed environments. In a distributed environment where user programs are executed concurrently on multiple processors and where the schema and subschema are not stored on all processors, the interpretive approach will be very expensive. In a distributed system, if a desired descriptor is not stored on the processor executing the user program, the cost of retrieving the correct descriptor will involve communica-

tion over a network. In the abstract data type model this communcation is not required, since all information required to access the database is bound to the user program. The only communication over the network required in such a system would be transfer of the user program to the processors where it is to be executed and the subsequent return of the results of execution to the user.

A third area of further research is the extension of the abstract data type model to support multiple data models. As discussed in Chapter 1, ANSI/SPARC has proposed an architecture for database management systems which incorporates three data models: the external model, the conceptual model, and the internal model. The three currently used data models, the hierarchical model, the network model, and the relational model, can be supported as three external models in this architecture.

We believe that the abstract data type model is a feasible implementation technique for this architecture. Using techniques similar to those described in this dissertation, each view of the database is represented by a collection of abstract data types. These abstract data types implement a view of the database in terms of their data attributes and procedures. The procedures associated with the abstract data types in a given level of the system are implemented in terms of the data attributes and

procedures of levels which are closer to the actual details of implementation. For example, the abstract data types representing the relational view of the database may be implemented in terms of the data attributes and procedures which represent the conceptual level. The abstract data types at the conceptual level are themselves implemented in terms of the data attributes and procedures of the internal level.

REFERENCES

[ANS66] American National Standards Institute. American National Standard FORTRAN (ANS X3.9-1966). American National Standards Institute, New York, N.Y. 1966.

[ANS75] ANSI/X3/SPARC Interim report 75-02-08, FDT BULLETIN OF ACM SIGMOD, Vol. 7, No. 2, February 1975.

[Ast76] Astrahan M. M., et al. System R: relational approach to database management. ACM TRANS. ON DATABASE SYSTEMS, Vol 1., No. 2, June 1976, pp 97-137.

[BY77] Baker, J., and Yeh, R. T. A hierarchical design methodology for data base systems. Department of Computer Science TR-70, The University of Texax at Austin, April 1977.

[Ber77] Berelian, E. A methodology for database design in a paging environment. Ph.D. Dissertation, The University of Michigan, 1977.

[BI77] Berelian, E., and Irani, K. B. Evaluation and optimization. PROC. INTERNAT. CONF. ON VERY

LARGE DATABASES. Tokyo, Japan, October 6-8, 1977, pp. 545-555.

[Ber76] Berg, J. (Ed.) Database directions: the next step. SIGMOD RECORD, Vol. 8, No. 4, November 1976.

[Bre78] Brechtlein, R. Comparing disc technologies. DATAMATION, Vol. 24, No. 1, January 1978, pp. 139-150.

[Bri75] Brinch Hansen, P. The purpose of concurrent PASCAL. PROC. OF THE IEEE INTER. CONF. ON RELIABLE SOFTWARE. April 21-23, 1975., pp. 305-309.

[BT77] Brodie, M. L. and Tsichritzis, D. Data base constraints. In A PANACHE OF DBMS IDEAS, Tsichritzis, D. (Ed.), University of Toronto Technical Report CSRG-78, University of Toronto, February 1977, pp. 19-36.

[BUR75] Burroughs Corporation. BURROUGHS B6700/B7700 DMSII DATA AND STRUCTURE DEFINITION LANGUAGE (DASDL) REFERENCE MANUAL. Form No. 5001084, Burroughs Corporation, October 1975.

[Car73] Cardenas, A. F. Evaluation and selection of file organizations -- a model and system. COMM. ACM, Vol. 16, No. 9 . September 1973, pp. 540-548.

[Car75] Cardenas, A. F. Analysis and performance of inverted database structures. COMM. ACM. Vol. 18, No. 5, May 1975. pp. 253-263.

[COD71] CODASYL DATA BASE TASK GROUP REPORT. ACM, New York. 1971.

[COD73] DATA DESCRIPTION LANGUAGE JOURNAL OF DEVELOPMENT, Document C13.6/2:13, U.S. Government Printing Office, Washington, D.C., 1973.

[Cod70] Codd, E. F. A relational model of data for large shared data banks. COMM. ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[DMN70] Dahl, O. J. , Myhrharg, B. , and Nygaard, K. THE SIMULA 67 COMMON BASE LANGUAGE. Pub. S-22, Norwegian Computing Center, Oslo, 1970.

[Dat75] Date, C. J. AN INTRODUCTION TO DATABASE SYSTEMS. Addison-Wesley. Reading, Mass., 1975.

[Dat76] Date, C. J. An architecture for high-level language extensions. ACM-SIGMOD INTERNAT. CONF. ON MANAGEMENT OF DATA. June 1976, Washington, D. C., pp. 101-122.

[Den76] Denning, P. J. Fault-tolerant operating systems. ACM COMPUTING SURVEYS, Vol. 8, No. 4, December 1976. pp. 359-389.

[Dij68] Dijkstra, E. W. The structure of the THE multiprogramming system. COMM. ACM, Vol 11, No. 5, May 1968, pp. 341-346.

[Fra77] Franta, W. R. THE PROCESS VIEW OF SIMULATION. North-Holland Publisning Company. New York, New York, 1977.

[FS76] Fry, J. P. and Sibley, E. H. Evolution of data-base management systems. ACM COMPUTING SURVEYS, Vol. 8, No. 1, March 1976, pp. 7-42.

[GMS77] Geschke, C.M., Morris, J. H. , and Satterthwaite, E. H. Early experience with MESA. COMM. ACM., Vol. 20, No. 8, August 1977, pp. 540-553.

[GG77] Gries, D. , and Gehani, N. Some ideas on data types in high-level languages. COMM. ACM, Vol. 20, No. 6, June 1977, pp. 414-420.

[Gut77] Guttag, J. Abstract data types and the development of data structures. COMM. ACM, Vol. 20, No. 6, June 1977. pp. 396-404.

[Hoa72] Hoare, C. A. R. Notes on data structuring. In STRUCTURE PROGRAMMING by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Academic Press, New York, 1972.

[Hoa74] Hoare, C. A. R. Monitors: an operating system structuring concept. COMM. ACM, Vol. 17, No. 10, October 1974, pp 549-557.

[ICRS75] INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE. SIGPLAN NOTICES, Vol. 10, No. 6, June 1975.

[JW75] Jensen, J., and Wirth, N. PASCAL USER MANUAL AND REPORT. Springer-Verlag, New York. 1975.

[Kos76] Koster, C. H. A. Visibility and types. PROC. OF THE SIGPLAN/SIGMOD CONF. ON DATA: ABSTRACTION, DEFINITION, AND STRUCTURE, March 22-24, 1976, pp. 179-190.

[LHLMP77] Lampson, B. W. , Horning, J. J. , London, R. L. , Mitchell, J. G.. , and Popek, G. L. Report on the programming language EUCLID. SIGPLAN NOTICES, Vol. 12, No. 2, February 1977.

[Lar78] Larsen, J. A. Personal communication, May 1978.

[Lin76a] Linden, T. A. The use of abstract data types to simplify program modifications. PROC. OF SIGPLAN/SIGMOD CONF. ON DATA: ABSTRACTION, DEFINITION, AND STRUCTURE, March 22-24, 1976, pp. 12-23.

[Lin76b] Linden, T. A. Operating system structures to support security and reliable software. ACM COMPUTING SURVEYS, Vol. 8, No. 4, December 1976, pp. 410-445.

[LSAS77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. COMM. ACM., Vol. 20, No. 8, August 1977, pp 564-576.

[McG77] McGee, W. C. The information management system IMS/VS. IBM SYSTEMS JOURNAL, Vol. 16, No. 2. 1977, pp. 84-168.

[MMC76] Michaels, A. S., Mittman, B., and Carlson, C. R. A comparison of relational and CODASYL approaches to data-base management, ACM COMPUTING SURVEYS, Vol. 8, No. 1, March 1976, pp. 125-151.

[Min76] Minsky, N. Files with semantics. PROC. OF THE ACM-SIGMOD INTERNAT. CONF. ON MANAGEMENT OF DATA, June 2-4, 1976, pp. 65-74.

[Mit75] Mitoma, M. F. Optimal database schema design. Ph.D. Dissertation, The University of MIchigan, 1975.

[MI75] Mitoma, M. F., and Irani, K. B. Automatic data base schema design and optimization. PROC. INTERNAT. CONF. ON VERY LARGE DATA BASES, Framingham, Mass. September 1975, pp. 286-321.

[Mye76] Myers, W. Key developments in computer technology: a survey. COMPUTER, Vol. 9, No. 11, November 1976, pp. 48-79.

[Myh77] Myhre, O. Protecting attributes of a local class. SIMULA NEWSLETTER, Vol. 5, No. 4, November 1977, Norwegian Computing Center, Oslo, Norway, pp. 14-15.

[Nau63] Naur, P. ed.. Revised report on the algorithmic language ALGOL 60, COMM. ACM, Vol. 6, No. 1, January 1963, pp. 1-17.

[NCC71] Norwegian Computing Center. SIMULA USERS GUIDE UNIVAC EXEC 8. Pub. No. S-36-0, Norwegian Computing Center, Oslo, Norway, 1971.

[NCC72] Norwegian Computing Center. SIMULA PROGRAMMER'S REFERENCE MANUAL. Pub. No. S-37-0, Norwegian Computing Center, Oslo, Norway, 1972.

[Olle75] Olle, T. William. A practitioner's view of relational data base theory. FDT-QUARTERLY BULLETIN OF ACM-SIGMOD, Vol 7, No. 3-4, 1975, pp. 29-43.

[Pal76a] Palme, J. New feature for module protection in SIMULA. SIGPLAN NOTICES, Vol. 11, No. 5, May 1976, pp. 59-62.

[Pal76b] Palme, J. The class concept in the SIMULA programming language. FOA REPORT C 10052-M3(E5), National Defense Research Institue, Stockholm, Sweden, August 1976.

[Pan76] Panigrahi, G. Charge-coupled memories for computer systems. COMPUTER, Vol. 9, No. 4, April 1976, pp. 33-41.

[Pan77] Panigrahi, G. The implications of electronic serial memories. COMPUTER, Vol. 10, No. 7, July 1977, pp. 18-25.

[Reg76] Rege, S. L. Cost, performance, and size tradeoffs for different levels in a memory hierarchy. COMPUTER, Vol. 9, No. 4, April 1976, pp. 43-51.

[RLNS77] Robinson, R., Levitt, K. N., Neumann, P. G., and Saxena, A. R. A formal methodology for the design of operating system software. In CURRENT TRENDS IN PROGRAMMING METHODOLOGY, VOL. 1, SOFTWARE SPECIFICATION AND DESIGN. R. T. Yeh (Ed.), Prentice-Hall, Inc. Englewood Cliffs, N. J., 1977.

[Rod78] Rodriguez-Rosell, J. Empirical data reference behavior in data base systems. COMPUTER Vol. 9, No. 11, November 1976, pp. 9-13.

[Sch77] Schkolnick, M. A clustering algorithm for hierarchical structures. ACM TRANS. ON DATABASE SYSTEMS, Vol. 2., No. 7, May 1977, pp. 27-44.

[SAAF73] Senko, M. E., et al. Data structures and accessing in data-base systems. IBM SYSTEMS JOURNAL, Vol. 12, No. 1, 1973, pp. 30-93.

[SWL77] Shaw, M., Wulf, W. A. , and London, R. L. Abstraction and verification in ALPHARD: defining and specifying iterators and generators. COMM. ACM., Vol. 20, No. 8, August 1977, pp 553-564.

[SS77a] Smith, J. M., and Smith, D. C. P. Database abstractions: aggregation and generalization. ACM TRANS. ON DATABASE SYSTEMS, Vol. 2, No. 2, June 1977, pp 105-133.

[SS77b] Smith, J. M. and Smith, D. C. P. Database abstractions: aggregation. COMM. ACM, Vol. 20, No. 6, June 1977, pp. 405-413.

[SPE75a] Sperry Univac. 1100 SERIES DATA MANAGEMENT SYSTEM (DMS 1100) SCHEMA DEFINITION DATA ADMINISTRATOR REFERENCE MANAUAL, UP-7907, Rev. 2, Sperry Univac, 1975.

[SPE75b] Sperry Univac. 1100 SERIES DATA MANAGEMENT SERIES (DMS 1100) SYSTEM SUPPORT FUNCTIONS DATA ADMINISTRATOR REFERENCE MANUAL, UP-7909, Rev. 3, Sperry Univac, 1975.

[Ste76] Stemple, D. W. A database management facility for automatic generation of database managers.

ACM TRANS. ON DATABASE SYSTEMS, Vol. 1, No. 1, March 1976, pp. 79-94.

[Sto74] Stonebraker, M. A functional view of data independence. ACM-SIGFIDET WORKSHOP ON DATA DESCRIPTION, ACCESS, AND CONTROL, Ann Arbor, Mich., May 1-3, 1974, pp. 63-81.

[SH75] Stonebraker, M. and Held, G. Networks, hierarchies, and relations in data base management systems. Memorandum No. ERL M504 (1975, Electronicc Research Laboratory, Univ. of California, Berkeley, Calif., 1975.

[SWKH76] Stonebraker, M., Wong, E., Kreps, P., and Held, G. The design and implementation of INGRES. ACM TRANS. ON DATABASE SYSTEMS, Vol. 1, No. 3, September 1976, pp. 189-222.

[TF76] Taylor, R. W., Frank, R. L. CODASYL Database Management Systems. ACM COMPUTING SURVEYS, Vol. 8, No. 1, March 1976, pp. 67-76.

[The78] Theis, D. J. An overview of memory technology. DATAMATION, Vol. 24, No. 1, January 1978, pp. 113-131.

[THLZ77] Travis, L., Honda, M., LeBlanc, R., and Zeigler, S. Design rationale for TELOS, a PASCAL-based AI language. PROC. SYMPOSIUM ON ARTIFICIAL IN-

TELLIGENCE AND PROGRAMMING LANGUAGES, August 15-17, 1977, pp. 67-76.

[Tsi75] Tsichritzis, D. A network framework for relational implementation., in DATA BASE DESCRIPTION, B. C. Douque and G. M. Nijssen (Eds.), North Holand, Amersterdam, The Netherlands, 1975, pp. 269-282.

[Tsi76] Tsichritzis, D. LSL; a link and selector language. ACM-SIGMOD INTERNAT. CONF. ON MANAGEMENT OF DATA. June 1976, Washington, D. C., pp 123-134.

[TK76] Tsichritzis, D., and Klug, A. An example of ANSI/SPARC architecture. In A PANACHE OF DBMS IDEAS, D. Tsichritzis (Ed.), University of Toronto Tech. Report CSRG-78, University of Toronto, February 1977, pp. 38-55.

[TL76] Tsichritzis, D., and Lochovsky, F. H. Hierarchical database management. ACM COMPUTING SURVEYS, Vol. 8, No. 1, March 1976, pp. 105-123.

[Weg74] Wegbreit, B. The treatment of data types in EL1. COMM. ACM, Vol. 17, No. 5, May 1974, pp. 251-265.

[Wen75] Wensley, J. H. The impact of electronic disks on system architecture. COMPUTER, Vol. 8, No. 2, February 1975, pp. 44-48.

[Wie77] Wiederhold, J. DATABASE DESIGN. McGraw-Hill, New York, 1977.

[Wir77] Toward a discipline of real-time programming. COMM. ACM, Vol. 20, No. 8, August 1977, pp. 577-583.

[Wor77] Wortman, D. B. (Ed.). PROC. OF THE ACM CONFERENCE ON LANGUAGE DESIGN FOR RELIABLE SOFTWARE. SIGPLAN NOTICES, Vol. 12, No. 3, March 1977.

[WRH71] Wulf, W. A., Russell, D. B., and Haberman,A. N. BLISS: a language for systems programming, COMM. ACM. Vol. 14, No. 12, December 1971, pp. 780-790.

[Yao77] Yao, S. B. An attribute based model for database access cost analysis. ACM TRANS. ON DATABASE SYSTEMS. Vol. 2, No. 1, March 1977, pp. 45-67.

[YM77] Yao, S. B.,. and Merten, A. Selection of file organizations through analytic modeling. PROC. INTERNAT. CONF. ON VERY LARGE DATA BASES, Framingham, Mass., September 1975, pp. 255-257.

[YB77] Yeh, R. T., and Baker, J. W. Toward a design methodology for DBMS: a software engineering approach. PROC. INTERNAT. CONF. ON VERY LARGE DATABASES, Tokyo, Japan, October 6-8, 1977, pp 16-27.

APPENDIX A

THE PRESIDENTIAL SCHEMA

```
IDENTIFICATION DIVISION
SCHEMA NAME IS PRESIDENTIAL
DATA DIVISION

RECORD SECTION

RECORD NAME IS ALL_PRESIDENTS
    RECORD CODE IS 1
    LOCATION MODE IS CALC
    USING PRESIDENTS DUPLICATES ARE NOT ALLOWED
    TEXT PRESIDENTS

 RECORD NAME IS ALL_STATES
     RECORD CODE IS 2
     LOCATION MODE IS CALC
     USING STATES DUPLICATES ARE NOT ALLOWED
     TEXT STATES

 RECORD NAME IS ALL_CONG_PRES_LINKS
     RECORD CODE IS 3
     LOCATION MODE IS CALC
     USING CONG_PRES_LINKS DUPLICATES ARE NOT ALLOWED
     TEXT CONG_PRES_LINKS

 RECORD NAME IS ALL_ELECTIONS
     RECORD CODE IS 4
     LOCATION MODE IS CALC
     USING ELECTIONS DUPLICATES ARE NOT ALLOWED
     TEXT ELECTIONS
```

```
RECORD NAME IS PRESIDENT
    RECORD CODE IS 5
    LOCATION MODE IS CALC
    USING LASTNAME DUPLICATES ARE LAST
    TEXT LASTNAME
    TEXT FIRSTNAME
    TEXT MIDDLEINITIAL
    TEXT MONTH_BORN
    INTEGER DAY_BORN
    INTEGER YEAR_BORN
    TEXT HEIGHT
    TEXT PARTY
    TEXT COLLEGE
    TEXT ANCESTRY
    TEXT RELIGION
    TEXT MONTH_DIED
    INTEGER DAY_DIED
    INTEGER YEAR_DIED
    TEXT CAUSE_OF_DEATH
    TEXT FATHER
    TEXT MOTHER
    INTEGER NUMOCCUPATIONS
    INTEGER NUMMARRIAGES

RECORD NAME IS ADMIN
    RECORD CODE IS 6
    LOCATION MODE IS VIA ADMIN_HEADED SET
    TEXT ADMIN_KEY
    TEXT INAUGURATION_MONTH
    INTEGER INAUGURATION_DAY
    INTEGER INAUGURATION_YEAR
    INTEGER NUM_VICE_PRES

RECORD NAME IS STATE
    RECORD CODE IS 7
    LOCATION MODE IS CALC
    USING STATENAME DUPLICATES ARE NOT ALLOWED
    TEXT STATENAME
    INTEGER YEAR_ADMITTED
    TEXT CAPITAL
    INTEGER AREA_SIZE
    INTEGER AREA_RANK
    INTEGER  POPULATION_SIZE
    INTEGER  POPULATION_RANK
    INTEGER  ELECTORAL_VOTES
    INTEGER  NUM_MAJOR_CITIES
```



```
RECORD NAME IS ELECTION
    RECORD CODE IS 8
    LOCATION MODE IS VIA ALL_ELECTIONS_SS SET
    INTEGER  ELECTION_YEAR
    INTEGER  WINNER_ELECTORAL_VOTES
    INTEGER  NUM_LOSERS

RECORD NAME IS CONGRESS
    RECORD CODE IS 9
    LOCATION MODE IS CALC
    USING CONGRESS_KEY DUPLICATES ARE NOT ALLOWED
    TEXT  CONGRESS_KEY
    INTEGER  NUM_PARTY_SENATE
    INTEGER  NUM_PARTY_HOUSE

RECORD NAME IS CONG_PRES_LINK
    RECORD CODE IS 10
    LOCATION MODE IS VIA CONGRESS_SERVED SET
    TEXT  CONGRESS_KEY


SET SECTION

SET NAME IS ALL_PRESIDENTS_SS
    SET CODE IS 1
    OWNER IS ALL PRESIDENTS
    MEMBER IS PRESIDENT AUTOMATIC
    ORDER IS SORTED
    ASCENDING KEY IS LASTNAME
    DUPLICATES ARE LAST
    SET OCCURRENCE SELECTION IS

SET NAME IS ALL_STATES_SS
    SET CODE IS 2
    OWNER IS ALL_STATES
    ORDER IS SORTED
    MEMBER IS STATE AUTOMATIC
    ASCENDING KEY IS STATENAME
    DUPLICATES ARE NOT ALLOWED
    SET OCCURRENCE SELECTION IS
    THRU LOCATION MODE OF OWNER
```

```
SET NAME IS ALL_CONG_PRES_LINKS SS
    SET CODE IS 3
    OWNER IS ALL_CONG_PRES_LINKS
    ORDER IS SORTED
    MEMBER IS CONG_PRES_LINK AUTOMATIC
    ASCENDING KEY IS CONGRESS_KEY
    DUPLICATES ARE LAST
    SET OCCURRENCE SELECTION IS
    THRU LOCATION MODE OF OWNER

SET NAME IS ALL_ELECTIONS_SS
    SET CODE IS 4
    OWNER IS ALL_ELECTIONS
    ORDER IS SORTED
    MEMBER IS ELECTION AUTOMATIC
    ASCENDING KEY IS ELECTION_YEAR
    DUPLICATES ARE NOT ALLOWED
    SET OCCURRENCE SELECTION IS
    THRU LOCATION MODE OF OWNER

SET NAME IS NATIVE_SONS
    SET CODE IS 5
    OWNER IS STATE
    ORDER IS SORTED
    MEMBER IS PRESIDENT AUTOMATIC
    ASCENDING KEY IS LASTNAME
    DUPLICATES ARE LAST
    SET OCCURRENCE SELECTION IS
    THRU LOCATION MODE OF OWNER

SET NAME IS ADMIN_HEADED
    SET CODE IS 6
    ORDER IS SORTED
    MODE IS CHAIN
    OWNER IS PRESIDENT
    MEMBER IS ADMIN AUTOMATIC
    ASCENDING KEY IS ADMIN_KEY
    DUPLICATES ARE NOT ALLOWED
    SET OCCURRENCE SELECTION IS
    THRU LOCATION MODE OF OWNER
```



```
SET NAME IS ADMITTED_DURING
    SET CODE IS 7
    OWNER IS ADMIN
    ORDER IS SORTED
    MEMBER IS STATE MANUAL
    ASCENDING KEY IS STATENAME
    DUPLICATES ARE NOT ALLOWED
    SET OCCURRENCE SELECTION IS
    THRU CURRENT OF SET

SET NAME IS ELECTIONS_WON
    SET CODE IS 8
    OWNER IS PRESIDENT
    ORDER IS SORTED
    MEMBER IS ELECTION AUTOMATIC
    ASCENDING KEY IS ELECTION_YEAR
    DUPLICATES ARE NOT ALLOWED
    SET OCCURRENCE SELECTION IS
    THRU LOCATION MODE OF OWNER

SET NAME IS CONGRESS_SERVED
    SET CODE IS 9
    OWNER IS PRESIDENT
    ORDER IS SORTED
    MEMBER IS CONG_PRES_LINK AUTOMATIC
    ASCENDING KEY IS CONGRESS_KEY
    DUPLICATES ARE NOT ALLOWED
    SET OCCURRENCE SELECTION IS
    THRU LOCATION MODE OF OWNER

SET NAME IS PRESIDENTS_SERVED
    SET CODE IS 10
    OWNER IS CONGRESS
    ORDER IS LAST
    MEMBER IS CONG_PRES_LINK MANUAL
    SET OCCURRENCE SELECTION IS
    THRU CURRENT OF SET
```

APPENDIX B

A SUMMARY OF SIMULA 67

We have presented all algorithms and examples in the syntax of SIMULA 67. SIMULA is a general-purpose programming language containing ALGOL 60 as a subset. The lnaguage is completely described in the REVISED REPORT ON ALGOL 60 [Nau63] and in the SIMULA 67 COMMON BASE LANGUAGE [DMN70]. In this appendix we briefly describe the SIMULA concepts most relevant to tnis dissertation: the class and reference variables. For a detailed discussion of the language tne reader is referred to Franta [Fra77], Dahl, et al. [DMN70], Palme [Pal76b], and the Norwegian Computing Center descriptions of the UNIVAC 1108/1110 implementation of SIMULA [NCC71] [NCC72].

The basic structuring concept in ALGOL 60 is the block, which we define as follows:

```
<block>        ::= BEGIN <block head><block body> END
<block head>   ::=<declarations>
<block body>   ::=<statements>
```


The declarations within a block define its local environment. A block is also considered as a statement, so that in the context of the statements which surround the block, the local declarations of the block are invisible. Each time a block is entered, the declarations are put into effect and when the block is exited, the activation record associated with these declarations is deallocated.

The main disavantage of the ALGOL 60 block is that a program which creates a new block instance can never interact with it as an object which exists and has attributes, since the block has disappeared by the time control returns to the statements which surround the block. The effects of executing the block are only visible through its modification of global variables.

The SIMULA class is an extension of the ALGOL block and may be regarded as an ALGOL block whose activation record is not deallocated after control leaves the block. Thus the class is a generalization of a block instance where several instance of a block may coexist and may interact with each other. An instance of a class is called an <u>object</u> of that class, where each object is created using the class declaration as a template.

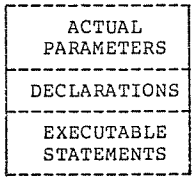
The execution of an ALGOL program consists of a sequence of dynamically nested block instances. In SIMULA a class instance may survive its initiating statement and

remains in existence as long as it is needed. Its attributes remain accessible while it exists.

The declaration of a class has the following form:

```
<class declaration> ::=<prefix> CLASS <class id>;
                        <formal parameter part>;
                        <specification part>;
                        <class body>
<prefix>            ::=<empty>
                    ::=<class id>
<class body>        ::=<statement>
```

The <formal parameter part> is a list of identifiers which occur within the class body and which are assigned values based on the actual parameters when a new class instance is created. A specification for the type and transmission mode for each formal parameter is provided by the <specification part>. The formal parameters and the variables and procedures declared in the <class head> constitute the attributes of the class. An object created using a class declaration as a template has the following form:



The class provides a mechanism to define abstract data types in which data and procedures operating on that data are encapsulated together. An example is shown below. In this example, the class box has six attributes consisting of four data attributes and two procedure attributes.

```
CLASS BOX(LENGTH,WIDTH,HEIGHT,DENSITY;
REAL LENGTH,WIDTH,HEIGHT,DENSITY;
BEGIN
REAL PROCEDURE VOLUME;
   VOLUME := LENGTH*WIDTH*DENSITY;
REAL PROCEDURE WEIGHT;
   WEIGHT := VOLUME * DENSITY;
END OF BOX;
```

By using PROTECTED and HIDDEN specifications [Pal76a] [Myh77] the class may be protected from the outside so that programs using a class may not access all of the data and procedures of the class directly. Protection checks

may be performed at compile-time to verify the correct use of a class by other modules of the program.

Objects are generated using the class declaration as a template by calling the object generator NEW. NEW behaves as a function and returns a reference or pointer to the newly generated object. A new variable type, REF or reference variables, is defined in SIMULA and is used to reference objects. The declaration of a REF variable must specify the class of objects which it may reference. This specification is known as its qualification. The syntax of a reference variable declaration is as follows:

```
<reference variable decl>::= REF (<qualification>)
                             <class id>
<qualification>          ::= <class id>
```

As shown in the example below, the value returned by a call to the object generator NEW may be assigned to a reference variable provided it has the correct qualification.


```
REF (BOX) FIRST_BOX, SECOND_BOX;
FIRST BOX :- NEW BOX(5.0,5.0,5.0,1.0);
SECOND_BOX :- NEW BOX(3.0,4.0,5.0,2.0);
```

Three special operators are used for reference variables

```
:-    denotes reference assignment
==    denotes a test for reference equality
=/=   denotes a test for reference inequality.
```

The attributes of an object may be accessed using a dot notation similar to PL/1 and PASCAL. This capability is called remote accessing. Thus we can access the length value associated with FIRST_BOX by writing FIRST_BOX.LENGTH. A mechanism similar to the WITH in PASCAL is used to form a global connection to an object. This mechanism has the form:

```
<connection statement>::=INSPECT <reference expr>
                         DO <connection block1>
                         <otherwise clause>
<connection block1>   ::=<statement>
<connection block2>   ::=<statement>
<connection clause>   ::=WHEN <class id> DO
                         <connection block1>
<otherwise clause>    ::=<empty>
                      ::=OTHERWISE <statement>
<connection part>     ::=<connection clause>
                      ::=<connection part>
                         <connection clause>
```

Example:

```
INSPECT FIRST BOX DO <statement1> OTHERWISE <statement2>
```

The <reference expr>, FIRST_BOX in the example, is evaluated. If it has a non-null value then <statement1> is connected to the oject resulting from evaluating <reference expr> and is executed. The attributes of the object accessed as a result of evaluating the expression may be referenced as local variables without using remote accessing within <statement1>. Repeated references to the attributes of FIRST_BOX in the example above do not add additional overhead for accessing the object.

As described above, the class is an example of an abstract data type. An abstraction is a description of a system component in which certain details of implementation are intentionally hidden. It is frequently useful to construct complex programs from simpler components. SIMULA provides the capability to construct hierarchies of classes, or abstract data types, by using class concatenation. In the earlier description of the class declaration it was noted that a class declaration could be prefixed by the identifier of another class. This operation can be regarded as a binary operation between two class declarations. Suppose that class C1 is declared as follows:

```
CLASS C1 ( <formal parameter1>);
   <specification part1>;
   BEGIN
       <declarations1>
       <executable statements1>;
   END of C1;
```

```
C1 CLASS C2( <formal parameters2>);
   <specification part2>;
   BEGIN
       <declarations2>;
       <executable statements2>;
   END of C2;
```

Using class concatenation, each object of the C2 class is a compound object and is the concatenation of the properties of both C1 and C2. An object of class C2 is considered to be a subclass of C1. That is, a prefixed class is considered to be a subclass to the prefix class. Thus the declaration of C2 shown above is equivalent to the following declaration:

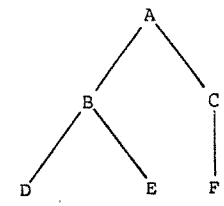
```
CLASS C2 (<formal parameter1>, <formal parameter2>);
   <specification part1>;
   <specification part2>;
   BEGIN
       <declarations1>;
       <declarations2>;
       <executable statements1>;
       <executable statements2>;
   END of C2;
```

Using the example of a BOX shown earlier, we can develop more complex concepts by extending the properties of a BOX as follows:

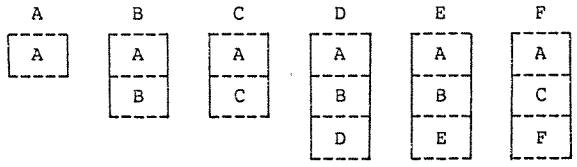

```
BOX CLASS CONTAINER(DESTINATION,ORIGIN
                    ,TARIFF,SHIPPER);
REF (PORT) DESTINATION, ORIGIN;
TEXT SHIPPER; REAL TARIFF;
BEGIN

   COMMENT CONTAINER has all the attributes of a BOX
           previously defined plus the new attributes
           of a container ;

   REAL PROCEDURE SHIPPING_COST;
       SHIPPING_COST :=
         WEIGHT*TARIFF*DESTINATION.DISTANCE(ORIGIN);
END OF CONTAINER;
```

The concept of a subclass was introduced earlier. A class may have multiple subclasses so that a hierarchy of subclasses may be created as follows:

Declarations	Hierarchy
CLASS A;	
A CLASS B;	
A CLASS C;	
B CLASS D;	
B CLASS E;	
C CLASS F;	



Each subclass is actually a compound object possessing all of the attributes of its prefix chain plus its locally defined attributes:



The attributes of a class declaration may be declared as VIRTUAL. The UNIVAC 1108/1110 implementation allows all data types and procedures to be declared as VIRTUAL. Declaration of an attribute as VIRTUAL provides two functions. The first function is to provide access at one prefix level to an attribute which is more conveniently declared at an inner prefix level. In this way SIMULA allows the declaration of the properties of an abstract data type to be separated from its implementation.

As an example, the procedural attributes of the UWA_RECORD described in Chapter 4 and Chapter 5 were declared as VIRTUAL. This means that these attributes could be accessed in the name scope of the data manipulation

routines. By declaring these attributes as VIRTUAL their implementation could be supplied in the user schema which is at an inner prefix level. This allows the usage of these attributes in the data manipulation routines to be checked at compile-time, even though an implemenation for them may not exist.

The second function of VIRTUAL attributes is to permit redeclaration at one prefix level of an attribute which was declared and used at an outer prefix level. Thus several implementations may coexist, but only one may be accessible. If matching implementations are given at more than one prefix level, then the innermost prefix level is used to match the VIRTUAL attribute.

As an example, the procedure attribute LOCATE of the UWA_SET implements set occurrence selection and is a VIRTUAL procedure. By defining this procedure as VIRTUAL both the schema and subschema may supply a definition of the set occurrence selection criteria to be associated with a set type. Since the subschema is at an inner prefix level to the schema, the implementation specified in the subschema is matched to the VIRTUAL procedure LOCATE.

A block in SIMULA may be prefixed with a <class id>. When the block is entered, an instance of the class named by the prefix is generated. The block executes as though it were connected to this object. That is, all of the at-

tributes of the prefixed object are accessible as though they are local to the block without the use of remote accessing. If the class named as the prefix is a subclass, then all of the attributes of the compound object representing the subclass are also accessible as local attributes. In this way hierarchies of environments can be created.

The environment defined by the abstract objects, the data manipulation routines, the schema, and the subschema is made available to the user program using this technique. The declarations which define the user environment using the PRESIDENTIAL schema and the PRES_ADMIN subschema are shown below.


```
CLASS ABSTRACT_OBJECTS;
        .
        .
        .
ABSTRACT_OBJECTS CLASS DATA_MANIPULATION_ROUTINES;
        .
        .
        .
DATA_MANIPULATION_ROUTINES CLASS PRESIDENTIAL;
        .
        .
        .
PRESIDENTIAL CLASS PRES_ADMIN;
        .
        .
        .
 PRES_ADMIN BEGIN
            COMMENT the user program ;
                .
                .
                .
            END;
```