APPLICATION OF ABSTRACT DATA TYPES TO THE IMPLEMENTATION OF DATA BASE MANAGEMENT SYSTEMS

by

A. J. Baroody
and
David J. DeWitt

Computer Sciences Technical Report #307

October 1977

APPLICATION OF ABSTRACT DATA TYPES TO THE IMPLEMENTATION OF DATA BASE MANAGEMENT SYSTEMS

by

A. J. Baroody

and

David J. DeWitt

ABSTRACT

This research describes the use of abstract data types as a design and implementation tool for data base management systems. Abstract data types, represented by generic objects and generic procedures, are used to implement a network data base management system. Generic objects are used to represent the data model, instantiations of the objects to represent the user's schema and subschema, and generic procedures to implement the data manipulation language verbs.

Traditional database management system design approaches are described in which run-time interpretation of the schema and subschema is employed to preserve data independence. Application of abstract data types to the design of a data base management system permits the elimination of the time-consuming run-time interpretation without suffering any loss of data independence. Data abstractions which represent the logical structure of the database are bound at compile-time to the user's program. The data manipulation language verbs included in the user's source program are implemented as parameterized calls to the procedures bound to those abstract data types which are used to represent the logical structure of the database.

Application Of Abstract Data Types To The Implementation of Data Base Management Systems

A.J. Baroody
and
David J. DeWitt

1.Ø INTRODUCTION

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.

Three major abstractions which are supported by high-level languages are procedural abstractions, data abstractions, and control abstractions [Hoar72] [Lisk77]. Procedural abstraction is supported in almost all programming languages, for example, the FORTRAN SUBROUTINE and the ALGOL PROCEDURE [Naur63]. Data abstractions are supported in terms of aggregate objects such as records in PASCAL [Jens75]. Support for control abstractions is relatively new and is provided in terms of constructs such as the CLU iterator [Lisk77].

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 [Dahl7Ø], the cluster in CLU [Lisk77], the form in ALPHARD [Shaw77], and modules in EUCLID [Lamp77], MODULA [Wirt77], and MESA [Gesc77]. All of these language constructs provide the capability to encapsulate an environment consisting of data structures and procedures which access these data structures.

The external view of 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 procedures 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 them.

Hammer [Hamm76] describes the application of data abstraction to enhance the data independence of data bases by exploiting the behavioral view of the semantics of the data. He also proposes the use of abstract data types to map the external schema onto the conceptual schema [ANSI75]. Smith and Smith [Smit77], based on Hoare's discriminated union structure [Hoar72], have formalized the concept of a generic object as a primitive for describing models of the real world. Associated with each instance of a generic object is a set of invariant properties which must be satisfied by the data in the data base. Gries [Grie77] has defined the concept of a generic procedure to be a "procedure operating on a parameter of any data type for which cetain basic operations have been defined."

Our research uses generic objects to represent the data model, instantiations of the objects to represent the user's schema and subschema, and generic procedures to implement the data manipulaton language (DML) verbs. We are investigating the application of abstract data types, represented by generic objects and generic procedures, to the implementation of network data base management systems. We foresee two major advantages of this approach. First, by using abstract data types to structure the design of a data base management system (DBMS), the resulting software should be more reliable [ICRS75] [Kost76] [Lind76] [LDRS77]. However, the significant advantage of our approach is a potentially dramatic improvement in the performance of a DBMS which is implemented using abstract data types. This is possible because application of abstract data types to a DBMS permits the elimination of the time-consuming run-time interpretation of the schema and subschema. Instead, data abstractions which represent the logical structure of the data base are bound at compile time to the user's program. In addition, the data manipulation language (DML) verbs 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 data base. In this way we can eliminate run-time interpretation of the schema and subschema without suffering any loss of data independence.

In Section 2.Ø, we will describe the traditional techniques which have been used to implement network data base management systems. Then, in Section 3.Ø, we present a description of the use of abstract data types as a design and implementation tool for data base management systems. Section 4.Ø contains a discussion of our future research plans.

2.Ø TRADITIONAL IMPLEMENTATION TECHNIQUES

Two alternative implementation strategies of a network or CODASYL [CODA71,73] DBMS are possible [Wied77]. The first is to encode the schema and subschema into an internal form, referred to as the object schema and the object subschema. Using the record types and set types referenced in a DML query, 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 DML command. The advantage of this approach is the high degree of flexibility provided by interpretation. The disadvantage is the execution time required to access the object schema and the object subschema and to interpret them.

The alternative approach is to compile the user DML commands into a directly executable form. The claimed disadvantages of this approach are its lack of data independence and the necessity to recompile all user programs for any change to the schema.

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 DBMS 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.

The major disadvantages of compiling the schema are in the area of programming language support for database management. The first disadvantage is the necessity to recompile all programs which access the database structures whenever the schema is modified. It is also more difficult to isolate the schema from user programs and thus it is more difficult to guarantee that user programs are not dependent upon the physical structure of the database. Coupled with these two problems is the fact that few programming languages have the capability to define structures as complex as those that exist within a DBMS.

By contrast, translating the schema and subschema into an object schema and an object subschema offers a number of advantages. The first of these is that 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, which reflect changes in the logical structure of the database, will, in general, necessitate restructuring the database. Control of concurrent access to the database is also easier since information about each process accessing the database is available in an interpretive environment.

The CODASYL verbs represent an example of generic procedures since the type of each parameter is a generic "record" or "set". User calls to these procedures specify actual parameters which are the names of user-defined record types or set types. Since in general programming languages do not support generic procedures, these procedures are implemented by accessing the object schema and the object subschema to supplement the information provided by the actual paramenters.

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 DBMS 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.

3.Ø DBMS IMPLEMENTATION USING ABSTRACT DATA TYPES

3.1 Introduction

As shown in Figure 3.1, there are four components to a data base management system which is implemented using abstract data types. The collection of ABSTRACT OBJECTS forms the base of this approach and is comprised of three abstract data types: Mass Storage Record, User Work Area (UWA) Record, and UWA Set. Each of these is composed of a collection of attributes which can be divided into data attributes and procedure attributes. The Data Manipulation Language (DML) verbs (e.g. FIND, GET, STORE, etc.)

are a collection generic procedures [Grie77] which are defined completely in terms of the attribute procedures defined in the three abstract data types. The abstract object instantiations supplement the generic information provided with the actual parameters in calls to the DML verbs (see Section 3.5). The Schema is the first component of Figure 3.1 which is data base dependent. The compiled schema consists of instantiations of the ABSTRACT OBJECTS corresponding to the record and set types defined by the Data Base Administrator as the logical structure of the data base. The SUBSCHEMA provides a means of restricting and reformatting the user's view of the structure of the data base he is accessing. Each of these four components will be described in detail in Sections 3.3 to 3.6.

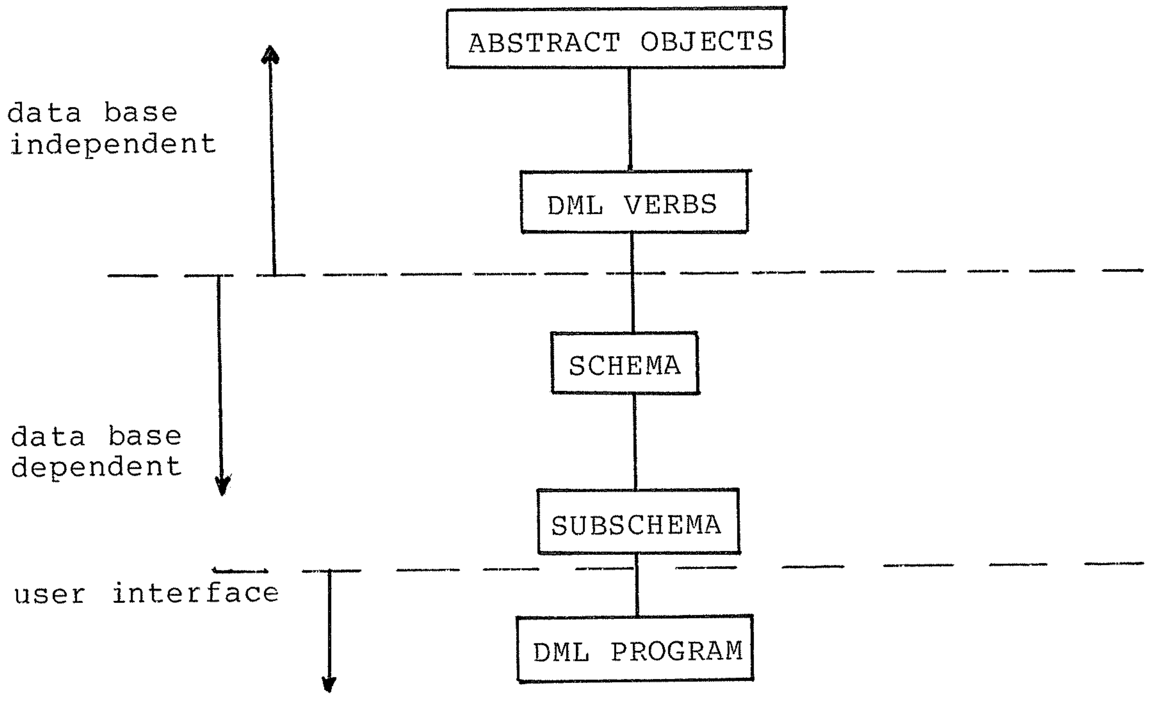

Figure 3.1

The major features which must be supported by such a compiled schema include:

- Representation of the complex objects within the data-base
- 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.

3.2 An Example Data Base

Throughout this section we will use the SUPPLIER-PARTS data base as an example to illustrate its representation when the DBMS is constructed using abstract data types. The logical structure of this data base is shown in Figure 3.2. There are three record types: SUPPLIER, PART, and SP. There are two set types: SUPPLIES and SUPPLIED-BY.

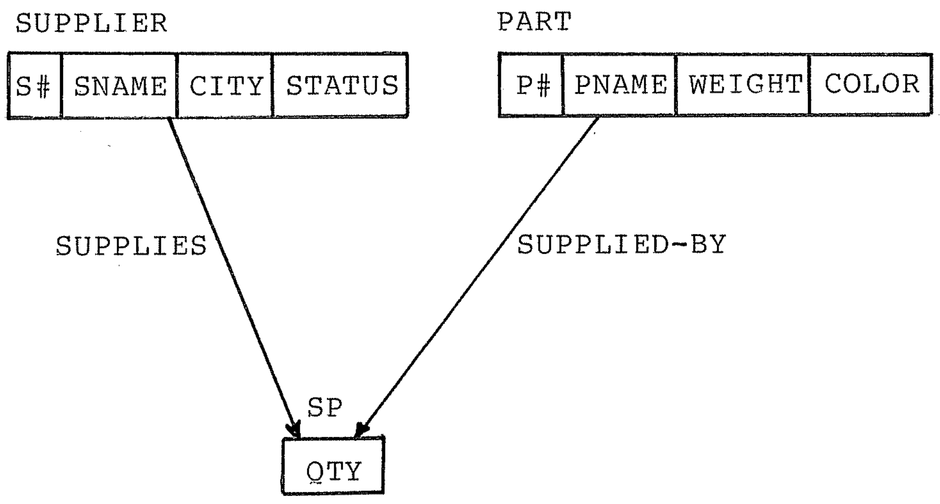

Figure 3.2

3.3 The ABSTRACT OBJECTS

The collection of ABSTRACT OBJECTS forms the foundation of our approach to the implementation of a data base management system. This collection consists of three abstract data types: Mass Storage Record, UWA Record, and UWA Set.

3.3.1 Mass Storage Record Abstract Data Type

The Mass Storage Record (Figure 3.3) represents a template for record instances in mass storage. The data attributes of this abstract data type are:

1. Record Type Information - this field is used to tag each record instance in the data base with a type descriptor so that run-time checking can be performed.
2. Data Items - fields which contain the data in a record occurrence. An instantiation of the Mass Storage Record for the SUPPLIER record type in the SUPPLIER-PARTS data base would contain four data items: S#, SNAME, CITY, and STATUS.
3. Owner Subabstraction - 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 Subabstraction are a FIRST pointer field and a LAST pointer field.

   A SUPPLIER record type instantiation of a Mass Storage Record would contain one instance of the Owner Subabstraction since the SUPPLIER record is the owner of only one set type - the SUPPLIES set. The FIRST pointer field of an occurrence of the SUPPLIER record type in the data base, would contain the secondary storage address (a data base key) of the first SP member record occurrence in its SUPPLIER set occurrence.

A SP record type instantiation of the Mass Storage Record abstract data type would not contain any instances of the Owner Substraction.

4. Member Subabstraction - this component of the Mass Storage Record abstract data type is used to describe in which set types the record participates as a member. The data attributes of the Member Subabstraction are PRIOR, NEXT, and OWNER pointer fields.

A SP record type instantiation of a Mass Storage Record would contain two instances of the Member Subabstraction since the SP record type is a member of two set types. The PRIOR, NEXT, and OWNER fields of a SP record occurrence in the data base are mass storage addresses of the prior, next, and owner record occurrences for each set occurrence in which the record occurrence participates.

The Mass Storage Record does not have any procedure attributes.

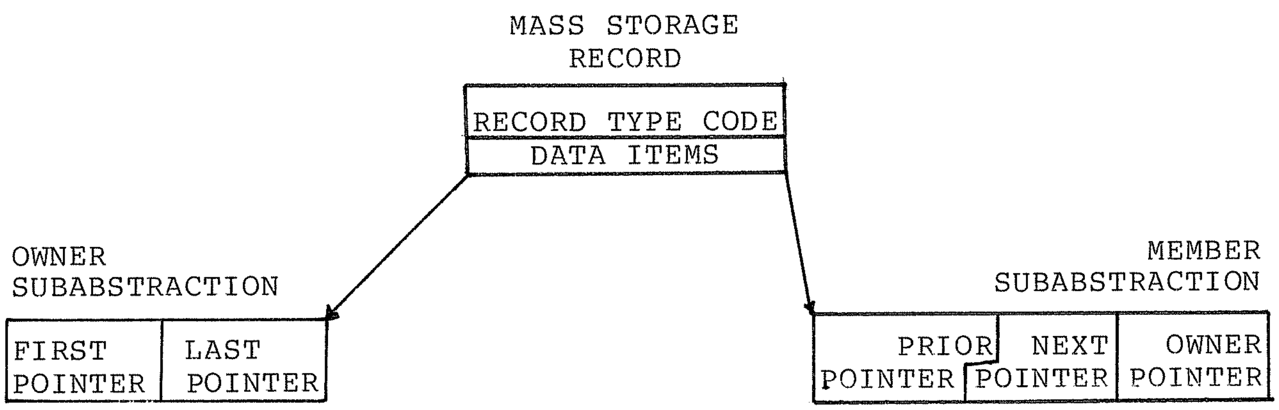

Figure 3.3

UWA RECORD

```
DATA ITEMS
CURRENT
NUMSETSOWNED
NUMSETSMEMBER
UWA SET INSTANCE POINTERS
LOCATE PROCEDURE
ALLOCATE PROCEDURE
STORE PROCEDURE
LOAD PROCEDURE
```

Figure 3.4

3.3.2 UWA Record Abstract Data Type

The UWA Record abstract data type (Figure 3.4) represents the template of a record type in the User Work Area. Its data attributes are:

1. Data Items - the fields which contain the data in a record occurrence.
2. Current - current of 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 SP Record instantiation of the UWA record would contain two instances of the UWA Set Pointer attribute - one for the SUPPLIES set type and one for the SUPPLIED-BY set type. NUMSETSMEMBER is the number of sets in which the record type participates as a member. NUMSETSOWNED is 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. For example, assume that the LOCATION MODE of the SUPPLIER record is CALC (hashed) on SNAME. Then, the LOCATE procedure of the SUPPLIER record type instantiation of the UWA Record will be a procedure which locates the desired SUPPLIER record occurrence by hashing on the SNAME value in the UWA supplier record instance.
2. ALLOCATE Procedure - This VIRTUAL procedure allocates a new mass storage record for the record type according to the record type instantiation of the corresponding MASS Storage Record utilizing the procedure specified in the schema.
3. LOAD (STORE) Procedure - This VIRTUAL procedure is used to load (store) the items in a user work area record occurrence from (to) mass storage after the mass storage record occurrence is located (allocated).

3.3.3 UWA Set Abstract Data Type

The UWA Set abstract data type (Figure 3.5) is used as a template for the set types in a data base. Its data attributes are:

1. Set Type Information - for run time checking of Set types.
2. Current - current of set type
3. OWNER UWA Record Pointer
4. MEMBER UWA Record Pointer

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. REORDER - a VIRTUAL procedure to reorder member record occurrences in a set occurrence based on a key specified in the SUBSCHEMA.

UWA SET

```
         SET TYPE INFORMATION
               CURRENT
      OWNER UWA RECORD POINTER
      MEMBER UWA RECORD POINTER
          INSERT PROCEDURE
          LOCATE PROCEDURE
          REMOVE PROCEDURE
          REORDER PROCEDURE
```

Figure 3.5

3.3.4 Discussion of VIRTUAL Procedure Attributes

Every procedure attribute of the UWA Record and UWA Set abstract data types was declared to be VIRTUAL. When a UWA Record (or Set) abstract data type is instantiated to form, for example, a SUPPLIER 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 data base. (See Section 3.6)

However, as we will demonstrate in Section 3.5, all DML verbs (e.g. FIND, GET, STORE, etc.) are generic procedures written solely in terms of the VIRTUAL procedures defined in the UWA Record and SET abstract data types thus insuring no loss of data independence.

3.4 An Example

Figure 3.6 represents the SUPPLIER-PARTS database in terms of UWA Records and Sets. For clarity not all of the currency pointers have been drawn in.

3.5 DML Verbs

The DML verbs are generic procedures. For example, the FIND verb is a procedure with only one actual parameter, a mass storage record type. Each record (set) declared in the schema provides a detailed description of a record's (set's) characteristics. All of these characteristics must be known to execute the verb. The traditional implementation techniques uses the generic information supplied by the actual parameters and the contents of the encoded schema and subschema (obtained through interpretation) to perform the requested action. When a DBMS is implemented using abstract data types this information is bound to instantions of the UWA Record and UWA Set by environmental concatenation of the schema and the subschema to the user's program. Thus all the additional information required by the DML verbs is bound at compile time and no run time interpretation is

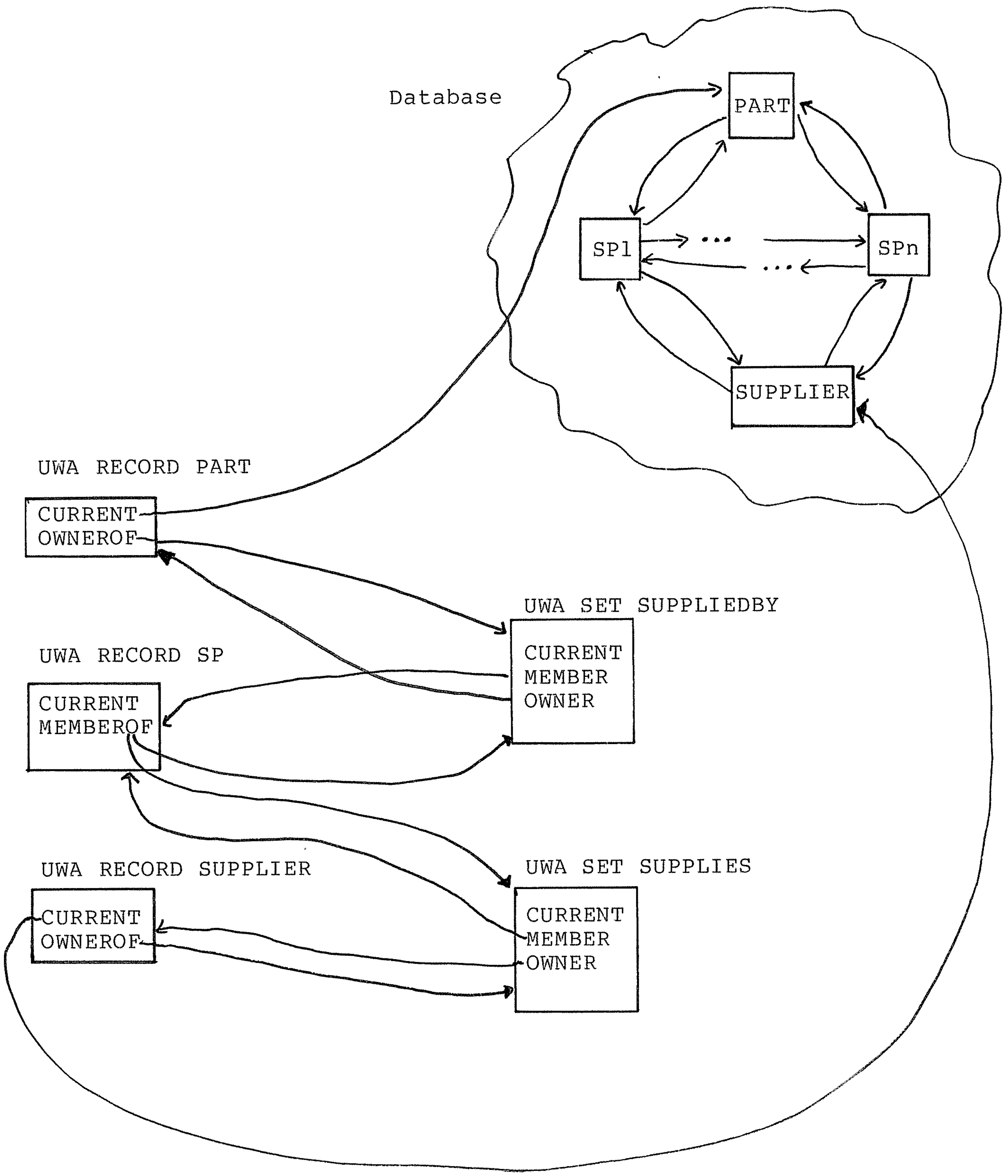
Database
PART
SP1
SPn
SUPPLIER
UWA RECORD PART
CURRENT
OWNEROF
UWA SET SUPPLIEDBY
CURRENT
MEMBER
OWNER
UWA RECORD SP
CURRENT
MEMBEROF
UWA RECORD SUPPLIER
CURRENT
OWNEROF
UWA SET SUPPLIES
CURRENT
MEMBER
OWNER

FIGURE 3.6

required.

Figures 3.7 and 3.8 contain, respectively, implementations of the DML verbs FETCH and STORE. In both of these examples RECPTR points to an instantiation of the UWA RECORD abstract data type (ie. a record type).

<u>DML FETCH VERB</u>

```
FETCH(RECPTR); REF(UWA_RECORD);
BEGIN INTEGER I;
/* use the schema location mode to find the mass storage record*/
/* copy mass storage fields into UWA record fields */

     /* Make record occurrence current of record type */
     RECPTR.CURRENT :- RECPTR.LOCATE;
     /* Load all data items into the UWA */
     RECPTR.LOAD (RECPTR.CURRENT)
     /* 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 3.7

DML STORE VERB

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

/* generate a new mass storage record using the schema information */
     MSREC :- RECPTR.ALLOCATE;
     RECPTR.STORE(MSREC);
     RECPTR.CURRENT :- MSREC;

/* make the mass storage record the current of set for */
/* each schema set it owns                             */

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

/*  Insert the mass storage 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 3.8

3.6 The SCHEMA and SUBSCHEMA

One of the most crucial concepts required to support the implementation of the schema and the subschema is environment control. 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 DML program as independent separate steps. In an interpretative approach the encoded tables produced by previous steps are available at each subsequent step and are utilized by the DBMS to interpret the user's DML 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 DML program, the programming language used to implement the DBMS 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 DML program. However 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 Associate a new data type with a data item declared in the schema
- o Associate a new set occurrence selection clause with a set type declared in the schema
- o Allow reordering of the data items within a record type declared in the schema.

The 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-stuctured 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 DML program. Thus the DML 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.

Examining the function of the subschema, what programming language features are required? 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 new data types, such as records. Given this capability, the subschema can declare a new identifier of the desired area, record, or set type. Code is then associated with the subschema to initialize this new identifier to have the same value as the area, set type, or record type declared in the schema.

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 problem in controlling access rights is to provide a means to restrict at the subschema level those identifiers which are declared in the schema. 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 DML programs using the subschema.

An alternative approach is based on the concept of EXPORTED and IMPORTED declarations which are used in EUCLID [Lamp77], MODULA [Wirt77], and TELOS [Trav77]. 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 DML 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 controlling the DML program so that it does not attempt to declare any schema identifiers as IMPORTED and thus bypass the subschema.

Redefiniton of data types for items within a record type can be handled automatically if equivalences can be defined which require coercions, or type conversions, to be performed automatically when the equivalenced identifier is referenced. 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.

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. The programming language used must allow certain tokens to be declared as VIRTUAL names. 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 declaration in the subschema which is bound to the VIRTUAL name. If no redefinition occurs in the subschema, the schema version is used.

CLU and ALPHARD provide a more elegant solution to this problem. Both languages allow the definition of abstract data types to be separated from implementations of such an object. Thus, a set type could be declared as an abstract data type with one implementation of the set occurrence selection clause provided by the schema and another possibly provided by the subschema.

3.7 Concurrency

A major source of complexity in a DBMS 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 [Brin75] [Hoar74] 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 record type in the schema is a monitor which implements procedures to lock and to unlock record occurrences within the database. This technique requires that each record occurrence have a unique identifier and that each user program have a unique identifier know to the DBMS. The structures local to the monitor are a list of all record occurrences which are currently locked and a queue of user programs which are waiting on a locked record. All DML verbs which modify a record occurrence incorporate calls to the monitor procedures to lock and unlock the record type.

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.

4.Ø FUTURE RESEARCH PLANS

There are several areas which we plan to investigate in the future. They include:

- Complete an implementation of our proposed DBMS architecture using the programming language SIMULA
- Either through analytic modelling or simulation attempt to measure the performance of a DBMS implemented using abstract data types versus an interpreted schema
- Expand the analysis of programming languages and their suitability to be used to compile the schema (e.g. examine CLU and ALPHARD)
- Use the concepts of abstract data types to analyze their suitability to implement a system designed according to the ANSI/SPARC architecture [ANSI75].
- Develop a more detailed solution to the concurrency problem

REFERENCES

[ANSI75] ANSI/X3/SPARC Interim report 75-02-08, FDT BULLETIN OF ACM SIGMOD, Volume 7, Number 2, (February 1975).

[Brin75] Brinch Hansen, P. The purpose of concurrent PASCAL. PROC. OF THE IEEE INTER. CONF. ON RELIABLE SOFTWARE. April 21-23, 1975., pp 305-309.

[CODA71] CODASYL DATA BASE TASK GROUP REPORT. ACM, New York. 1971.

[CODA73] CODASYL Data Description Language Committee. DATA DESCRIPTION LANGUAGE JOURANL OF DEVELOPMENT, Document C13.6/2:13, U.S. Government Printing Office, Washington, D.C. 1973.

[Dahl70] Dahl, O. J. , Myhrharg, B., and Nygaard, K. THE SIMULA 67 COMMON BASE LANGUAGE. Pub. S-22, Norwegian Computing Center., Oslo, 1970.

[Gesc77] Geschke, C.M., Morris, J. H. , and Satterthwaite, E. H. Early experience with MESA. COMM. ACM. 20,8 (August 1977). pp. 540-553.

[Grie77] Gries, D. , and Gehani, N. Some ideas on data types in high-level languages. COMM. ACM 20,6 (June 1977), pp. 414-420.

[Hamm76] Hammer, M. Data abstractions for data bases. PROC. OF THE SIGPLAN/SIGMOD CONF. ON DATA: ABSTRACTION, DEFINITION, AND STRUCTURE, March 22-24, 1976. pp. 58-59.

[Hoar72] 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.

[Hoar74] Hoare, C. A. R. Monitors: an operating system structuring concept. COMM. ACM 17,10 (October 1974), pp 549-557.

[ICRS75] PROC. OF THE IEEE INTERNATIONAL CONFERENCE ON RElIABlE SOFTWARE. April 21-23, 1975.

[Jens75] Jensen, J. and Wirth, N. PASCAL USER MANUAL AND REPORT. Springer-Verlag, New York. 1975.

[Kost76] 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.

[Lamp77] Lampson, B. W. , Horning, J. J. , London, R. L. , Mitchell, J. G.. , and Popek, G. L. Report on the programming language EUCLID. SIGPLAN NOTICES, Volume 12, Number 2 (February 1977).

[LDRS77] PROC. OF THE ACM CONFERENCE ON LANGUAGE DESIGN FOR RELIABLE SOFTWARE, March 28-30, 1977.

[Lind76] 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.

[Lisk77] Liskov, B., Snyder,A. Atkinson, R. and Schaffert, C. Abstraction mechanisms in CLU. COMM. ACM. 20,8 (August 1977). pp 564-576.

[Naur63] Naur, P. (ed.). Revised report on the algorithmic language ALGOL 60. COMM. ACM, 6,1 (January 1963) pp. 1-17.

[Shaw77] Shaw, M., Wulf, W. A. , and London, R. L. Abstraction and verification in ALPHARD: defining and specifying iterators and generators. COMM. ACM. 20,8 (August 1977). pp 553-564.

[Smit77] Smith, J. M., and Smith, D. C. P. Database abstractions: aggregation and generalization. ACM Transactions on Database Systems Vol. 2, No. 2, (June 1977) pp 105-133.

[Trav77] Travis, L., Honda, M., LeBlanc, R., and Zeigler, S. Design rationale for TELOS, a PASCAL-based AI language. PROC. SYMPOSIUM ON ARTIFICIAL INTELLIGENCE AND PROGRAMMING LANGUAGES. August 15-17, 1977. pp. 67-76.

[Wied77] Wiederhold, J. DATABASE DESIGN. McGraw-Hill Book Company. New York. 1977.

[Wirt77] Wirth, N. Toward a disipline of real-time programming. COMM. ACM. 20,8 (August 1977). pp. 577-583.