SEMANTICS OF NETWORK DATA MANIPULATION LANGUAGES: AN OBJECT-ORIENTED APPROACH

Dipayan Gangopadhyay
Department of Computer Science
The University of Texas at Austin
Austin, TX. 78712

Umeshwar Dayal
Computer Corporation of America
575 Technology Square
Cambridge, MA. 02139

James C. Browne
Department of Computer Science
The University of Texas at Austin
Austin, TX. 78712

ABSTRACT

An axiomatic basis for defining the semantics of navigational data manipulation languages is presented. This basis consists of an abstraction of the network data model achieved by three abstract data types, an assertion language to express properties of database states, and a DML to program the transactions. The proof rules of the DML constructs and the axioms defined on the data types can be used to establish the correctness of transactions. Potential applications of the proposed formalism in language design, semantic definition of existing languages, and integrity management, are outlined via examples.

Keywords: Navigational DML, Formal Semantics, Abstract Data Types, Network Data Model, Verification of Programs

1. INTRODUCTION

Many extant database systems are based on the network data model [CODA 71]. Users' transactions in such a system are formulated as navigational programs [BACH 73]. Verifying that a given transaction preserves consistency is an important problem [BBC 80]. A database state is consistent with respect to the real world if the stored data in that state satisfy the semantic integrity assertions. A transaction preserves consistency if its execution on a consistent database state results in a database state that is also consistent.

Verification requires a precise definition of the data model and the semantics of the data manipulation language (dml) in which the transactions are programmed. Over the past decade, research in programming languages has developed techniques for specifying the semantics of programs and for proving their correctness. In this paper, we present an approach that adapts and applies two of these techniques to provide an axiomatic basis on which formal semantics of network dmls permitting navigation can be built and correctness of dml programs can be proved.

We start by treating the database as a collection of network structured objects, characterized by a few abstract data types [GUTT 78]. We then define an assertion language for expressing properties of a database state in terms of functions and predicates defined on these abstract data types. We also propose a simple dml in which the transactions may be programmed. The dml statements are treated as assignments of network structured values to database objects. Therefore, their semantics can be given in the axiomatic style of Hoare [HOAR 74]. Using the axioms of the dml statements and those of the abstract data types, we can prove that a dml program is correct: if the program is initiated in a database state characterized by a given assertion, we can show that upon completion of the program, the final database state satisfies another given assertion.

Though our treatment in this paper may not include all the details of CODASYL DML, we believe that our axiomatic basis addresses two significant problems in the formal semantics of network dmls. These problems, viz. navigational access and update of shared mutable objects, are explained below.

The existing network dmls allow sequential recod-at-a-time access based upon the concepts of currency pointers and ordering of the records. For example, in CODASYL DML, the effect of the FIND NEXT WITHIN SET operation is dependent upon the CURRENT OF SET and CURRENT OF RUN UNIT pointers as well as upon the order clause for the SET type in question. While implicit currency pointers, manipulable by side-effects, provide flexibility in programming, the presence of side-effects makes the semantics of individual DML statements more complex. For example, the semantics of FIND NEXT, as given in [BILL 76], interact with the semantics of DELETE, because both the statements have implicit effects on currency pointers. Our aim is to identify primitives for navigation that have simple proof rules, (since keeping the proof rules simple can

make verification easier). Hence, we do not use the FIND statements of CODASYL DML with their implicit effects on currency as our primitives. Instead, we make currency explicit by treating the members of an owner-coupled set as an ordered set: the positions of the members are mapped to a sequence of natural numbers. We then provide two functions, POS and its inverse, GET, to perform this mapping. The semantics of more complex statements can then be expressed in terms of these primitives. For example, the FIND NEXT operation of CODASYL DML can be formulated as: first find the positional number; then increment it; and finally use it to get the member at this incremented position. Such explicit manipulation of currency is also advocated in recent language proposals, which introduce cursor variables manipulable by application programs [DATE 80].

The shared mutable object problem [LISK 77] is caused by the participation of a member record in more than one owner-coupled set. Updating the value of this shared record has to be visible to all these owner-coupled sets. In the literature of abstract data types, the specification of shared objects and verification of programs using them have been recognized as a problem [FLON 79]. Traditionally, an object is treated as being indistinguishable from its value; a higher level object is treated as a collection of values of the objects it encompasses. An update operation on an object is then expressed as the assignment of a new value to the object, and is assumed to modify the object itself, where each assignment is assumed to affect only a single object. Thus updating a shared object requires arbitrarily many assignments (in parallel) of new values to all the higher level objects encompassing the shared object. We provide a practical solution to this problem by distinguishing between the identifier designating the object and the value of the object; a separate mechanism (c.f., REFCLASS in Section 2.1) to access the value of an object with a given identifier is then required. In our formulation, owner-coupled sets have as values only the identifiers of the member records. Thus an update on a shared record changes only the value of the record but not its identifier, and can be treated as a single assignment of the new value to this record. Subsequent navigational retrievals through all the owner-coupled sets in which this record is a member, can obtain the new value of the record.

Relatively little work has been done on proving correctness of dml programs [CASA 80, GARD 79]. Though these efforts were similar to ours in spirit, they considered only relational dmls and did not have to deal with the two problems of navigational access and shared mutable objects, which are typical in non-relational dmls. Biller and Neuhold [BILL 76] developed denotational semantics for CODASYL DDL and DML. Our work differs from theirs in two respects: first, we use the technical machinery of abstract data types and Hoare's axiomatics, which are more widely accepted in the programming methodology community than denotational semantics; second, their development aimed at providing a faithful specification of CODASYL DML by including the database i/o area in the language semantics. In contrast, we choose to treat the database i/o area as part of the program's address space, and advocate the explicit use of program variables in defining the semantics of individual DML statements. As is the case with making currency explicit, factoring out the specification of the database i/o area from the semantics of individual DML statements leads to simpler proof-rules, which makes program verification easier.

The "clean" data model and simple primitives for navigational access that we define here have several potential uses. First, they can be embedded directly in a programming language to give a precisely-defined navigational DML. Second, they can serve to define precisely the semantics of existing navigational languages such as CODASYL DML [CODA 71] and UDL [DATE 80]; this would facilitate uniform understanding of the language semantics by users and implementors alike, and also would make it possible to verify that programs written in them preserve consistency. In addition to program verification, precise semantics also lay the groundwork for program synthesis. This is important in multi-model database systems, in which a high-level query language is used as an intermediate language for mapping between models [ZANI 79, HD 81, SMIT 75]. Queries in the mapping language have to be compiled automatically into equivalent DML programs. Doing this correctly requires a precise semantic definition of both the query language and the navigational DML. Finally, our data model and primitives can be used as the basis for designing new navigational DMLs; defining the semantics of a new DML in terms of these primitives would expose any semantic errors or inconsistencies in the design. In this paper we illustrate some of these points via examples. A complete treatment of these applications is a subject of future research. We see the semantics developed here as the starting point for that research.

In Section 2, we develop the specification of the abstract data types in an object-oriented view of database. In Section 3, we formulate the assertion language and illustrate its use. In Section 4, we present the DML statements and their axiomatic semantics. We also illustrate the use of these axioms in verifying dml programs. In section 5, we briefly summarize the applications of our formalism.

2. OBJECT ORIENTED MODEL OF A NETWORK DATABASE

A network schema describes two sets of types, viz., a set of record types, and a set of owner-coupled set (OCS) types. A network database stores occurrences of record types and OCS types. In this section, we develop an object-oriented model of a network database. We first describe what an object-oriented model is, in general, and then show how network schemas and navigational access can be described in terms of this model.

2.1. Object-oriented model

An object-oriented system is viewed as a set of typed objects. Each object is uniquely identified by an identifier. An object holds a value, and its type[1] designates a set of possible values that the object can hold. Given an object type S and a countable set I of object identifiers, there exists, at any instant, an evaluation function VAL#S: I --> S, which retrieves the value of the object identified by a given identifier. This evaluation function is, however, time dependent, because an update operation on an object will change its value. Also, the function as defined above is partial because objects can be created or deleted from the system and only the existing objects can have a value. If we assume undefined to be a special value in each type S and designate the value of a non-existing object to be this special value, the evaluation function can be redefined as a total function VAL#S: I --> S U {undefined}. In this formulation, the creation or deletion operation on an object i in I also changes VAL#S.

An object i1 of type S1 is said to refer to (or reference) another object i2 of type S2 if VAL#S1(i1)=i2. A navigational retrieval of the value of the referenced object i2 by following the reference from i1 is expressed as VAL#S2(VAL#S1(i1)). As the update on a referenced object does not affect its identifier, a navigational retrieval following the update will result in the updated value of the referenced object. A special case of reference, called null reference, occurs when the referenced object has undefined as its value.

We define a reference class to be a countable set of objects of a given type S, as in [LUCK 79]. At any instant of time, the value D#S of a reference class is the set of pairs of object identifiers and object values, i.e., is exactly the evaluation function VAL#S. As the update operations on the objects result in a new evaluation function, these operations result in a new value of the corresponding reference class.

We characterize the behaviour of the objects under the update operations by treating the values of reference classes as instances of a parameterized abstract data type REFCLASS[S], where S is the set of all object types defined for the system. The update operations on individual objects are defined as the constructor[2] functions on data type REFCLASS[S], because they construct new instances (values) of the associated reference class. The specifications of this abstract data type are given in figure 2-1.

It may be noted that we ignore error conditions in our specification of abstract data types. For example, in axiom d4 of 2-1, we state that delete operation has no effect on an empty reference class, rather than specify an error condition. In general, we adhere to Guttag's scheme of specifying error conditions in which a special error value is returned by a function when the function is "illegally" used [GUTT 78]. The meanings of the operations on REFCLASS are given below:

EXTEND(D#S,i)::
adds an object with identifier i to D#S.

ASSIGN(D#S,i,v)::
assigns a value v to object i in D#S. It will be written as D#S[i<-v].

VAL(D#S,i):: extracts the value of object i from reference class D#S. It will be written as D#S[i].

DEFINED(D#S,i)::
This is true if object identifier i denotes an existing object in reference class D#S.

---

[1] Strictly speaking, a type designates a set of operations and axioms on these operations in addition to the set of possible values.

[2] The set of functions defined over an abstract data type is partitioned into two classes: constructors and observers. A constructor function results in a new instance of the defined data type. An observer function results in a value of type other than the defined data type. For example, for data type stack, Push may be a constructor, and Isempty may be an observer.

```
type REFCLASS[S]  (* instances written as D#S *)
requires
     S: set of object types
     ID: set of object identifiers
     OBJ: set of object values
     IDLIST: set of list-of-identfiers
operations
     INIT: S              --> D#S
     EXTEND: D#S X ID     --> D#S
     ASSIGN: D#S X ID X OBJ    --> D#S
     DELETE: D#S X ID     --> D#S
     VAL:    D#S X ID     --> OBJ
     FIND:   D#S X QUAL   --> IDLIST
     DEFINED: D#S X ID    --> BOOLEAN
axioms
d1. VAL(INIT(S),i) = undefined
d2. VAL(EXTENDS(D#S,i1),i2)
    = if i1=i2
         then undefined
         else VAL(D#S,i2)
d3. VAL(ASSIGN(D#S,i1,o),i2)
    = if i1=i2
         then o
         else VAL(D#S,i2)

d4. DELETE(INIT(S),i) = INIT(S)
d5. DELETE(EXTEND(D#S,i1),i2)
    =if i1=i2
        then D#S
        else EXTEND(DELETE(D#S,i2),i1)
d6. DELETE(ASSIGN(D#S,i1,o),i2)
    = if i1=i2
         then D#S
         else ASSIGN(DELETE(D#S,i2),i1,o)

d7. FIND(INIT(S),q) = [] (* [] denotes an
                            empty list *)
d8. FIND(EXTEND(D#S,i),q) = FIND(D#S,q)
d9. FIND(ASSIGN(D#S,i,o),q)
    = if q(i)
         then [i] || FIND(D#S,q)
                        (* || denotes list
                           concatenation  *)
         else FIND(D#S,q)
d10. DEFINED(INIT(S),i) = false
d11. DEFINED(EXTEND(D#S,i1),i2)
     = if i1=i2
          then true
          else DEFINED(D#S,i2)
d12. DEFINED(ASSIGN(D#S,i1,o1),i2)
     = DEFINED(D#S,i2)

d13. (* axiom of equality *)
     D#S = D#S'
     iff (∀i)((DEFINED(D#S,i)=DEFINED(D#S',i)
               and (VAL(D#S,i) = VAL(D#S',i))
```

Figure 2-1: Definition of data type REFCLASS

DELETE(D#S,i)::
: removes object i from reference class D#S. All references to this object become null references.

FIND(D#S,q)::
: retrieves a set of object identifiers for the object values that satisfy qualification expression q.

The FIND operation is the basis of value based search in our model. The qualification expression parameter q is a single variable boolean expression where the variable denotes objects of type S (the target type of the search) and q involves functions and predicates defined over the values of type S. For example, in a reference class of objects of type INTEGER, the qualification expression to find all objects with values less than 5 can be

q <--> (VAR x: VAL#INTEGER(x) < 5)

2.2. Network Structured Values

A traditional network schema N describes two sets of types, namely, the set of record types T, and the set of owner-coupled set (OCS) types L. A network database, which is an extension of N, stores occurrences of these record types and OCS types. In terms of our object-oriented model, a database is a set of objects, where the set of object types S is the union of L and T. The set of objects in the database is partitioned by type into several reference classes, one for each object type. A database state is the set of values of the constituent reference classes.

A record occurrence of type T in T is a record valued object (record) whose value is a k-tuple <a1,a2,..,ak>, where each ai in DOM(Fi), Fi is a field of record type T, and DOM(Fi) is the domain of values associated with field Fi. An OCS occurrence of type L in L is an OCS-valued (OCS) object whose value is a 2-tuple <t,m>, where t is the identifier of the owner record and m is an ordered set of identifiers of the member records. Thus, an OCS references both its owner record as well as the constituent member records. As a result, the effect of an in-place update of any referenced member record is visible on subsequent navigational retrievals via the referencing OCS.

The record-values and OCS-values are treated as instances of two parameterized abstract data types, RECORD[T] and OCS[L], respectively. Such an object-value can be assigned to an object in the reference class for the object type in question. The definitions of these two abstract data types appear in figures 2-2 and 2-3.

```
type  RECORD[T]
requires
     T: set of record types
     F: set of field types
     D: set of domains
operations
     EMPTY: T              --> RECORD
     WRITE: RECORD X F X D --> RECORD
     READ:  RECORD X F     --> D
axioms
r1. READ(CREATE(T),F,d) = NULL
                        (* each domain has
                           the special value
                           NULL *)
r2. READ(WRITE(r,F1,d),F2)
    = if F1=F2
         then d
         else READ(r,F2)
```

Figure 2-2: Definition of data type RECORD

The meanings of the operations defined on data type RECORD[T] are given below:

EMPTY(T):: creates an empty record-value with the value NULL for each of its fields.

WRITE(r,F,v):: stores value v in field F in record-value r.

READ(r,F):: extracts the value of field F from record-value r.

The meanings of the operations defined on data type OCS[L] are given below:

CREATE(L,r,T):: creates an empty OCS-value of type L, with record r of type T as its owner.

ADD(s,r,T):: add record r as a member to OCS-value s.

HEAD(s):: retrieves the identifier of owner record from an OCS-value s.

OWNS(s,i):: is true if record i is a member of OCS-value s.

POS(s,r,Q):: retrieves the position number of record r within the set of members in OCS-value s; the binary predicate Q(x,y) determines the ordering of the set of members.

GET(s,n,Q):: retrieves the identifier of the member record which is in the nth position within the set of members in OCS-value s. The ordering is determined by binary predicate Q(x,y).

REMOVE(s,r,T):: removes a member record with identifier r from OCS-value s. The argument T denotes the type of member record.

MEMBERS(s):: retrieves the set of identifiers of the member records from OCS-value s.

```
type OCS[L]   (* written as s *)
requires
     RID: set of record identifiers
     T: set of record types
operations
     CREATE: L X RID X T  --> OCS
     ADD:    OCS X RID X T    --> OCS
     REMOVE: OCS X RID X T    --> OCS
     HEAD:   OCS      --> RID
     OWNS:   OCS X RID --> RID
     MEMBERS: OCS  --> {RID}
     POS:    OCS X RID X ORDER -->  INTEGER
     GET:    OCS X INTEGER X ORDER --> RID
axioms
o1. HEAD(CREATE(L,r,T)) = r
o2. HEAD(ADD(s,r,T)) = HEAD(s)
o3. OWNS(CREATE(L,r1,T),r2) = false
o4. OWNS(ADD(s,r1,T),r2)
    = if r1=r2  then true
                else OWNS(s,r2)
o5. POS(CREATE(L,r1,T),r2,Q) = 0
o6. POS(ADD(s,r1,T),r2,Q)
    = if Q(r2,r1)  then POS(s,r2,Q)
                   else 1 + POS(s,r2,Q)
o7. GET(CREATE(L,r,T),n,Q) = nil
o8. GET(ADD(s,r,T),n,Q)
    = if n=POS(ADD(s,r,T),r,Q)
         then r
         else if n > POS(ADD(s,r,T),r,Q)
                then GET(s,n-1,Q)
                else GET(s,n,Q)

o9. MEMBERS(CREATE(L,r,T)) = {}
o10. MEMBERS(ADD(s,r,T)) = {r} U MEMBERS(s)

o11. REMOVE(CREATE(L,r1,T1),r2,T2)
     = CREATE(L,r1,T1)
o12. REMOVE(ADD(s,r1,T1),r2,T2)
     = if r1=r2
          then s
          else ADD(REMOVE(s,r2,T2),r1,T1)
```

Figure 2-3: Definition of data type OCS

Navigational access within an OCS-value is supported by the two operations POS and GET. Based on the ordering predicate Q, function POS maps the identifiers of the member records to their ordinal numbers 1,..,N. Function GET performs the inverse mapping. The ordering predicate Q is a binary predicate defining the ordering of the member records within an OCS. For example, if employee records (EMP) under an OCS DE are ordered in descending order of the salary field (SAL), the ordering predicate Q that relates two employee records e1 and e2, is

Q(e1,e2) <--> D#EMP[e1].SAL > D#EMP[e2].SAL

The different varieties of FIND statements in CODASYL DML can be expressed in terms of our functions FIND, POS, and GET. For example, FIND FIRST T WITHIN SET S, where the SET SELECTION CLAUSE for OCS type S is VIA OWNER, can be modelled by first using FIND(D#S,P) to retrieve identifier s of the selected OCS of type S; and then using GET(VAL(s),1,Q) to retrieve the identifier of the first member record. Note that the SET SELECTION CLAUSE is encoded in the qualification expression P, which is an argument of the FIND operation; the ordering clause defined over the OCS type S is expressed by the ordering predicate Q, which is an argument of the GET operation. Similarly, FIND NEXT WITHIN can be modelled as GET(VAL(s),n+1,Q) where s is the identifier of the selected OCS and n is the ordinal number of the current member record. Thus, the implicit side-effects on currency pointers in CODASYL DML are translated into explicit arithmetic operations on ordinal numbers.

We believe that the definitions of these three data types REFCLASS, RECORD and OCS provide a simple abstraction of the structures and primitive operations of network databases. The features of the CODASYL DDL specifications [CODA 71] not handled in our formulation are:

- repeating groups
- multiple member types for an OCS type
- storage structure information.

The various specifications of ordering, membership class, and primary keys (fields for which duplicates are not allowed) will be treated in our formulation as integrity assertions. We give some examples of how these assertions can be expressed in section 3.

3. DATABASE ASSERTION LANGUAGE

Any database state can be constructed by repeated application of the constructor operations of types REFCLASS, RECORD and OCS. Similarly properties of a database state can be observed by applying a sequence of observer operations of these data types. Integrity assertions are examples of properties of database states. In this section, we define an assertion language to express such properties of database states by extending the many-sorted first order predicate calculus [ENDE 72] with terms involving database objects. These new terms, called database terms, are:

1. all symbols denoting object types, fields, object identifiers, object values and reference classes;
2. all symbols denoting list-of-object-identifiers;
3. all functions and predicates defined on data types REFCLASS, RECORD and OCS;
4. all terms obtained from 1,2 and 3 by function composition.

Apart from the database terms, we assume the existence of the following operators on type LIST : NULL, FIRST, REST and COUNT. We also assume the existence of comparison predicates for equality, etc., defined over the domains associated with the field types.

An Example Database

We now introduce a simple example that illustrates the ability of our assertion language to express properties of database states. Consider the database schema shown in figure 3-1. Order clauses and membership class clauses are missing from the schema. We express these, instead, as integrity assertions.

```
SCHEMA PERSONNEL
TYPE STRING = PACKED ARRAY[1..12] OF CHAR;
TYPE   DEPT = RECORD  (* DEPARTMENT *)
                DNAME: STRING;
                BUDGET: REAL
              END;
       EMP  = RECORD  (* EMPLOYEE *)
                SSNO: STRING;
                DNAME: STRING;
                SALARY: REAL
              END;
       DE   = OCS  (* DEPARTMENT'S EMPLOYEES *)
                OWNER: DEPT;
                MEMBER: EMP
              END;
       MGR  = OCS   (* MANAGER'S SUBORDINATES *)
                OWNER: EMP;
                MEMBER:EMP
              END;
END PERSONNEL
```

Figure 3-1: Example Database Schema

Example 1. Automatic Membership
In OCS type DE the members are automatic. This is expressed as:

(C1):: (∀e∈EMP)(∃de∈DE): OWNS(D#DE[de],e)

Here e and de are variables of type object identifier bound to the object types EMP and DE. Note that D#DE[de] is our notation for VAL(D#DE,de) and is required for the type conversion from object identifier to its value.

Example 2. Ordering of members
The member records in OCSs of type DE are in descending order by SALARY. This is expressed as:

(C2):: (∀de∈DE)(∀e1,e2∈EMP): Q(e1,e2)
--> (POS(D#DE[de],e1,Q) < POS(D#DE[de],e2,Q))

where the ordering predicate Q is
Q(e1,e2) <--> (D#EMP[e1].SALARY
> D#EMP[e2].SALARY).
Note that D#EMP[e1].SALARY is shorthand for READ(D#EMP[e1],SALARY).

Example 3. Structural Constraint
Subordinate employees must be in the same department as their manager.

(C3):: (∀m∈MGR)(∀e∈EMP):
(OWNS(D#MGR[m],e) -->
(D#EMP[HEAD(D#MGR[m])].DNAME
= D#EMP[e].DNAME))

It may be noted that both "recursive sets' (e.g., OCS type MGR) and structural constraints are proposed in CODASYL 78 [MANC 78].

Example 4. Duplicates Not Allowed constraint
The SSNO field in EMP records is a primary key.

(C4):: (∀e∈EMP):
(COUNT(FIND(D#EMP,(VAR x: D#EMP[x].SSNO =
D#EMP[e].SSNO)))
= 1)

Note that in Example 4 we have used the cardinality function COUNT defined over the data type LIST.

Example 5. General Integrity Constraint
No employee can earn more than his manager.

(C5):: (∀m∈MGR)(∀e∈EMP):
(OWNS(D#MGR[m],e) -->
(D#EMP[e].SALARY
<= D#EMP[HEAD(D#MGR[m])].SALARY))

Note that in CODASYL DDL, one has to use a trigger procedure to check this constraint.

From a pragmatic viewpoint, this assertion language has the advantage that we can express a large class of integrity assertions non-procedurally. In contrast, CODASYL DDL provides special constructs for some of these constraints; all other constraints must be coded as trigger procedures. A more important point is that this assertion language is backed by a proof theory of first order calculus augmented by the axioms of the data types REFCLASS, RECORD and OCS. The proof theory allows us

- to prove that some properties of a database are valid over all database states
- to detect inconsistencies in integrity specifications during the database design phase
- to detect redundant specifications of integrity assertions, thereby minimizing the number of integrity assertions to be maintained.

4. DATA MANIPULATION LANGUAGE AND PROOF RULES

In this section, we design a simple data manipulation language over the primitive operations of the three abstract data types by integrating the data types with the control structures of PASCAL.

The integration is achieved by including the database objects in the execution enviroment of the programming language, so that a dml program can directly access the database objects. One consequence of this integration is that the programmer can use the control structures for manipulating the database in the same way as he uses them to manipulate the usual program variables. This advantage has been recognized in a number of recent database language proposals [DATE 80], [WASS 79], [SCHM 77]. The other consequence is that the user work area and currency pointers are eliminated from the description of the semantics of the dml statements. This makes the semantics of the dml statements "cleaner", thereby improving the understanding of programs, and more importantly, simplifying the proof-rules necessary for program verification. (The reader may contrast our formulation with that by Biller and Neuhold [BILL 76], which included the specification of the database I/O area in the language semantics).

4.1. Data Manipulation Lanuage

We extend the type declaration facilities of PASCAL by including the schema definitions of RECORD and OCS. The variables introduced are object variables and list variables. A list

variable can have as its value a list of object identifiers. An object variable (analogous to a tuple variable in relational languages) can have as its value an object identifier.

The dml statements for updating the database are as follows:

M1. CREATE R(r)
This statement creates a record object of type R. The variable r denotes this stored record.

M2. CREATE L(1) WITH R(r)
This creates an OCS object of type L, with the record denoted by r (of type R) as its owner. The variable 1 denotes this OCS.

M3. CONNECT R(r) TO L(1)
This connects the record denoted by r to the OCS denoted by s.

M4. STORE v IN R(r).F
This stores the value v in the field F of the record denoted by r.

M5. DISCONNECT R(r) FROM L(1)
This disconnects the record denoted by r from the OCS denoted by 1.

M6. DELETE S(s)
This deletes the object denoted by s. The object type is denoted by S.

The retrieval statement in our dml has the form:

M7. ASSIGN x WITH e
The target variable x is either an object variable, a list variable or a usual PASCAL variable. The source expression e is a retrieval expression as defined below.

A retrieval expression is a term formed by function compostion of only the observer functions defined on the data types REFCLASS, RECORD and OCS. A retrieval expression can be used to form boolean expressions used for controlling WHILE statements or IF-THEN-ELSE statements.

The dml statements M1-M7 can be used in conjunction with all PASCAL statements. However, we introduce a specific form of FOR-statement [HOAR1 72], which is useful for sequential processing of database objects. This statement is shown below:

M8. FOREACH x IN X DO M
Here X is a list variable (or a type name) and x is an element of the list (or object variable of given type). The statement M is repeatedly executed for each element in the list X. The statement M is not allowed to change either x or X.

An example program, based on the schema in figure 3-1, is shown in figure 4-1. This program is intended to give a 10% raise to all employees of the department that is the owner of an OCS object with identifier de.

```
VAR e: RECORD EMP;
    M: LIST OF MGR; de: OCS DE;
    s: REAL;
BEGIN
(* assume the object variable de has the
   identifier of the selected OCS object
   of type DE *)

   ASSIGN E WITH (MEMBERS(D#DE[de]));
   FOREACH e IN E DO
   BEGIN (* process this employee e *)
     ASSIGN s WITH (D#EMP[e].SALARY);
     IF   s <= 80K
     THEN
        STORE (s * 1.1) IN EMP(e).SALARY
   END
END.
```

Figure 4-1: Example program to update member records selectively

4.2. Proof Rules

The formal semantics of the dml statements M1-M8 can now be given in Hoare's axiomatic style. In this approach, the semantics of a programming language statement M are given by two assertions, called the precondition and the postcondition. Precondition P describes the initial state of the system before M is executed and postcondition Q describes the final state after M's execution. For example, the semantics of the PASCAL assignment statement x:=e are given as:

$$Q[e/x] \quad \{x:=e\} \quad Q$$

The symbol Q[e/x] stands for the assertion obtained by replacing all the free occurrences of symbol x in assertion Q by expression e.

As all the dml statements, M1-M5, are assignments to the respective reference classes, we give their axiomatic semantics as follows:

A1.
(not defined(D#R,r) --> Q)[D#R'/D#R]{M1}Q
where
D#R' = extend(D#R,r)[r <- EMPTY(R)].

The axiom states that if the object r did not exist before, then statement M1 will extend the reference class with object r and assign to it the record value EMPTY(R).

A2.
(not defined(D#L,1) --> Q)[D#L'/D#L]{M2} Q
where
D#L' = extend(D#L,1)[1<-Create(L,r,R)]

A3. Q[D#L'/D#L] {M3} Q
where D#L' = D#L[1<-add(D#L[1],r,R)]

A4. Q[D#R'/D#R] {M4} Q
where
D#R' = D#R[r<-write(D#R[r],F,v)].

A5. Q[D#L'/D#L] {M5} Q
where
D#L' = D#L[s <-- remove(D#L[s],r,R)]

A6. Q[D#S'/D#S] {M6} Q
where D#S' = delete(D#S,s)

A7. Q[e/X] {M7} Q

A8. (X = X1||[x]||X2),I(X1) {M} I(X1||[x])
--> I([]) {M8} I(X)
where [] denotes the empty list, and || denotes the list concatenation operator.

The proof-rules for the usual PASCAL statements are those given by Hoare and are shown in figure 4-2.

A9. Q[e/x] {x:=e}Q
A10. P{S}R, (R-->Q) --> P{S}Q
A11. (P-->R), R{S}Q --> P{S}Q
A12. P{S1}R, R{S2}Q --> P{S1;S2}Q
A13. (P and B{M}Q), (P and not B{M'}Q)
--> P{IF B THEN M ELSE M'}Q
A14. (P and B{M}P)
--> P{WHILE B DO M}(P and not B)

Figure 4-2: Proof-rules for PASCAL control structures

4.3. Proof Technique

We now discuss the use of these axioms in proving the correctness of dml programs. A program S is proved correct by showing that if an initial assertion P holds before the program, then a final assertion Q holds on completion of the program. That is, we have to prove a formula P{S}Q, written in Hoare's pseudological notation. Each of the axioms A1-A14 is of the form H1 and H2 and ... and Hn --> P{S}Q. Hence, to show P{S}Q one has to establish the truth of each antecedent Hi. For example, consider axiom A13: in order to show P{if B then M else M'}Q, we have to show the antecedents (P and B){M}Q and (P and not B){M'}Q. Also, note that axioms A10 and A11 require us to prove sufficiency conditions, e.g., P-->R in axiom A11. Since these sufficiency conditions are formulas in our assertion language, they can be proved using the proof theory of our assertion language (i.e., the axioms of the data types together with the proof rules of first order calculus).

5. APPLICATIONS

In sections 2,3, and 4, we presented axiomatic semantics of network database operations. Based on the definitions of three abstract data types, we developed a framework for formulating integrity assertions, characterizing database structures, and verifying dml programs. Our formulation of the semantics is a starting point for research on a number of problems in network database technology. First, the formulation can serve as a basis for providing interpretive semantics of other navigational languages for the network data model; such interpretive semantics provide a precise definition of the language. Second, the proof system developed in this paper makes it possible to verify dml programs. A third application of our formalism is to language design. Inconsistencies in language design usually arise out of interactions between different components of a language, even when each component by itself is seemingly well understood. We shall illustrate the application of our formalism to each of these problems.

5.1. Interpretive Semantics of Existing Languages

Throughout the paper, we have presented examples of how CODASYL DML constructs can be defined in our model. Here we present some examples of interpreting constructs from a recent language proposal, UDL [DATE 80].

Example 1. Setting cursor varible
An employee record under a given department may be retrieved in UDL by the statement:

```
FIND FIRST EMP
   UNDER UNIQUE DEPT VIA DE
   WHERE DEPT.DNAME = 'research'  SET (e)
```

Here the UNDER clause specifies a fanset (OCS) from which the first member record is to be selected. In our formulation, we shall encode this specification in a qualification expression as:

```
P:: (VAR x: D#DEPT[HEAD(D#DE[x])].DNAME
                              = 'research')
```

The semantics of the UDL statement is then given as:

```
ASSIGN de WITH (FIRST(FIND(D#DE,P)));
ASSIGN e WITH (GET(D#DE[de],1,Q))
```

where Q is the ordering predicate for the OCS DE. Note that the cursor variable e of the UDL statement is represented by a corresponding record variable in our DML. However, a cursor variable is implemented in UDL as having pointers to the selected record as well as to the selected ordered set. In our formulation, the cursor variable e must thus be represented by the pair <e,de>.

Example 2. Relative retrieval using cursor

Suppose the FIND statement in the example above has been used to set the cursor variable e. One can then use the following statement in UDL to select the next member record within the same OCS :

```
FIND UNIQUE (EMP AFTER e) SET (e)
```

This statement is interpreted in our model as:

```
ASSIGN e WITH (GET(D#DE[de],
                  (POS(D#DE[de],e,Q)+1),
                  Q)
```

where Q is the ordering predicate defined over the members of the OCS DE. Note again that we use ordinal numbers of the member records for navigating within one OCS.

5.2. Verification of DML Programs

The verifiability of programs can play an important role in integrity management. In particular, when a fixed set of preanalyzable transactions is to be designed for a closed DBMS (such as an airline reservation system), one can verify once (at design time) that each transaction preserves the integrity assertions as invariants. Alternatively, the proof system may be used to derive run-time tests for integrity maintainence that are less expensive than the original integrity assertions. This technique has been proposed for the relational model in [BBC 80], [BB 82]. (Our proof system is limited to a serial execution of programs. We have not addressed the problems of inconsistency arising out of concurrent execution.)

To illustrate the application of the proof system, we consider the dml program shown in figure 4-1 and integrity assertion C6 shown below:

```
(C6):: (∀e∈EMP): (e in MEMBERS(D#DE[de])
            -->  D#EMP[e].SALARY <= 88K)
```

This assertion states that the employees within the selected department can earn at most 88K as their salaries. In order to show that the program preserves integrity assertion C6, we have to prove the formula C6{Program}C6.

We postulate an invariant I(S)

```
I(S):: (E ⊆ MEMBERS(D#DE[de]))
   and ((∀i) (i in S --> D#EMP[i].SALARY<=88K))
   and ((∀j) (j in (E - S)
                    --> D#EMP[i].SALARY<=88K))
```

Let M denote the body of the FOREACH statement. Define

```
 P <--> if D#EMP[e].SALARY <= 80K
     then (∀i) ((i in (E1 || [e]))
                    --> D#EMP[i].SALARY<=88K)
       and (∀j) (j in (E - (E1 || [e]))
                    -->D#EMP[j].SALARY<=88K)
          else I(E1 || [e])
```

It can be mechanically verified that

```
          P{M}I(E1 || [e]).
```

This can be done by using axioms A13 and A4 to produce a sufficiency condition. By using axiom r2 of data type RECORD this sufficiency condition can be reduced to

```
(y <= 80K) --> ((y * 1.1) <= 88K)
```

which is true.

We must next prove the tedious but trivial lemma:

```
(E = E1 || [e] || E2) and I(E1) --> P
```

and this gives us by axiom A11 :

```
((E = E1 || [e] || E2) and I(E1)){M}I(E1 || [e])
```

The proof rule A8 for the FOREACH statement enables us to conclude

```
          I([]){FOREACH e IN E DO M}I(E)
```

Thus we have shown that

```
I([]) :: (E c MEMBERS(D#DE[de]))
         and ((∀ j) (j in E
            --> D#EMP[j].SALARY <= 88K))
```

Now by axioms A7 and A11, we generate another sufficiency condition

```
C6 --> (∀ j) (j in MEMBERS(D#DE[de])
                  --> D#EMP[j].SALARY <= 88K),
```

which is also true.

This concludes the proof of the formula C6{program}C6.

5.3. Language Design

When plain English text is used to describe the semantics of a language, potential inconsistencies may be overlooked. We motivate this point by an example.

Consider the semantics of the CODASYL statements FIND RECORD WITHIN SET USING IDS, and FIND NEXT DUPLICATE WITHIN SET USING IDS. In these statements, two different access methods viz., navigational retrieval and value based search interact. The normal use of these statements is to use the former once and then to repeatedly use the latter. The semantics of the second statement (we quote from [OLLE 78], p.156) are:

> The DBCS begins its search for the identified record at the current record of the set type and proceeds to search in the order defined by the set ordering criteria of that type.

Obviously, if the items of the USING clause are order keys, the above semantics are appropriate. But when the items are search keys, the above semantics would not make sense because the search key order may be different from the set order (i.e., the order imposed by the set ordering criterion).

In our model, we would give an interpretive semantics as follows:

```
FIND NEXT DUPLICATE EMP RECORD
        WITHIN DE SET USING SALARY = 20K
<-->
FIND (D#EMP, (VAR x: (D#EMP[x].SALARY = 20K)
             and (POS(D#DE[de],x,Q)
                  = 1 + POS(D#DE[de],e,Q))))
```

where e and de refer to the current EMP record and current DE OCS respectively and Q is the ordering predicate. Since the ordering predicate Q is specified explicitly as argument to POS in the retrieval expression, there is no ambiguity. Using the approapriate ordering predicate determines whether the records are retrieved in the search key order or set order.

It is difficult to detect inconsistencies, such as the above, if the semantics of the language are informally described. We argue, therefore, that formal semantic specification should go hand in hand with language design.

6. CONCLUSION

The primary objective of this work was to provide a clean semantics of navigational dmls for network databases. Our goal was to identify language primitives for which we could develop simple proof rules. We found that to meet this goal, we had to make explicit all the effects of dml statements; e.g., their effects on currency and their interactions with the program workspace.

The technical machinery used in this work consisted of two well-known techniques from programming methodology, viz. abstract data types and Hoare's axiomatics. We adapted and applied these techniques to develop a coherent framework for network database languages and integrity management.

To completely characterize data semantics, we must specify the effects of operations on the data. The applicability of abstract data types is, therefore, being investigated by a number of researchers in the area of database design [BROD 80], [PING 80]. The object-oriented approach, presented in this paper, should contribute towards this investigation. In particular, the problem of shared mutable objects can arise also in conceptual schema design, because an object type may be encompassed by more than one higher level object type along the generalization hierarchy [SMIT2 77]. The object-oriented approach developed in this paper can be used to deal with this problem.

Acknowledgment

We wish to thank Professor Jay Misra and Dr. Don Good for helpful discussions on the ideas presented in this paper. We gratefully acknowledge also the comments of Dr. Frank Manola, which helped to clarify many opaque passages in an earlier version of this paper.

REFERENCES

[BACH 73] Bachman, C. W. The Programmer as Navigator. CACM 16(11):653-658, Nov., 1973.

[BB 82] Bernstein, P.A. and Blaustein, B.T.
Fast Methods for Testing Qualified Relational Calculus Assertions.
In Proc. ACM SIGMOD. ACM, June, 1982.

[BBC 80] Bernstein, P.A., Blaustein, B.T., and Clarke, E.M.
Fast Maintainence of Semantic Integrity Assertions using Redundant Aggregate Data.
In Proc. 6th. VLDB, pages 126-136. 1980.

[BILL 76] Biller, H., Glatthar, W. and Neuhold, E. J.
On the Semantics of Databases: The Semantics of Data Manipulation Languages.
In G. M. Nijssen (editor), Modelling in Database Management Systems, . North Holland, 1976.

[BROD 80] Brodie, M.L.
The Application of Data Types to Database Semantic Integrity.
Information Systems 5(4), 1980.

[CASA 80] Casanova, M. and Bernstein, P. A.
A Formal System for Reasoning about Programs Accessing a Relational Database.
ACM TOPLAS 2(3), 1980.

[CODA 71] Database Task Group of the CODASYL Programming Language Committee
ACM, New York, 1971.

[DATE 80] Date, C.J.
An Introduction to the Unified Data Language (UDL).
In Proc. 6th. VLDB. ACM, 1980.

[ENDE 72] Enderton, H.
A Mathematical Introduction to Logic.
Academic Press, 1972.

[FLON 79] Flon, L. and Misra, J.
A Unified Approach to the Specification and Verification of Abstract Data Types.
In Proc. of Symposium on Reliable Software. IEEE, 1979.

[GARD 79] Gardarin, G. and Melkanoff, M.
Proving Consistency of Database Transactions.
In Proceedings of 5th. VLDB, Brazil. ACM, 1979.

[GUTT 78] Guttag, J. V. and Horning, J. J.
The Algebraic Specification of Abstract Data Types.
Acta Informatica 10(1), 1978.

[HD 81] Hwang, H-Y. and Dayal, U.
Using the Entity-Relationship Model to Implement Multi-Model Database Systems.
In P. P. Chen (editor), Entity-Relationship Approach to Information Modeling and Analysis, pages 237-258. ER Institute, Saugus, Calif., 1981.

[HOAR 74] Hoare, C. A. R. and Lauer, P.
Consistent and Complementary Formal Theories of the Semantics of Progamming Languages.
Acta Informatica 2:135-155, 1974.

[HOAR1 72] Hoare, C.A.R.
A Note on the For Statement.
BIT 12:334-341, 1972.

[LISK 77] Liskov, B. et. al.
Abstraction Mechanisms in CLU.
CACM 20(8), 1977.

[LUCK 79] Luckham, D. C. and Suzuki, N.
Verification of Array, Record, and Pointer operations in Pascal.
ACM TOPLAS 1(2), 1979.

[MANO 78] Manola, F.
A Review of the 1978 CODASYL Database Specifications.
In Proc. of 4th. VLDB, Berlin, pages 232-242. ACM, 1978.

[OLLE 78] Olle, T.W.
The CODASYL Approach to Database Management.
John Wiley and Sons, 1978, .

(PING 80) Brodie, M.L. and Zilles, S.N. (editors).
Proc. Workshop on Data Abstraction, Databases and Conceptual Modelling.
ACM, Pingree Park, Colorado, 1980.

[SCHM 77] Schmidt, J.W.
Some High Level Language Constructs for Data Type Relation.
ACM TODS 2:247-267, 1977.

[SMIT 75] Smith, J. M., Bernstein, P. A., Dayal, U., Goodman, N., Landers, T., Lin, K.W.T., and Wong, E.
Multibase - Integrating Heterogeneous Distributed Database Systems.
In Proc. AFIPS NCC, pages 487-499. 1975.

[SMIT2 77] Smith, J.M. and Smith, D.C.P.
Database Abstraction - Aggregation and Generalization.
ACM TODS 2(2):105-133, 1977.

[WASS 79] Wasserman, A.I.
The Data Management Facilities of PLAIN.
In Proc. ACM SIGMOD, pages 60-70. 1979.

[ZANI 79] Zaniolo, C.
Multi-Model External Schemas for CODASYL Database Management Systems.
In Proc. IFIP TC-2, pages 157-176. North Holland, June, 1979.