Constructs for a Non Procedural Object-oriented Database Manipulation Language OSQL/N

I. Bradley

Department of Computer Science

University of Calgary

Calgary, Alberta, Canada

Abstract Constructs for an object-oriented non procedural data base manipulation language called OSQL/N are presented. The major feature is that the language allows for conditional manipulation of an object together with a quantified set of its related objects. The constructs required for this were borrowed from a language called not N, which is implicitly object oriented, but which was originally developed as an attempt at a more natural relational data base language. The constructs behind NQL/N are much more suited to object oriented data bases, and the OSQL/N language structure as a result, allows for incorporation of the whole array of quantifiers that are commonly used in natural language, that is the natural quantifiers.

Keywords data base, object oriented, natural quantifier, non procedural language, quantifier, OSQL/N

Introduction

The object-oriented (OO) approach to database management evolved from the OO approach to programming [1, 6, 7], and commercial systems [2, 5, 10, 12, 13, 14, 15] are finding use in such application areas as software engineering, document preparation and management, and in the design and production of engineering parts.

A consistent feature of the OO approach is that associated with every object representation, which crudely corresponds to a tuple in the relational approach [8, 9], is a system generated identifier together with both lists of references, and individual references, to the object identifiers of related objects.

In an OO database, the relationships can be one-to-many, binary many-to-many, ternary many-to-many, recursive many-to-many, and ISA one-to-one relationships, as with relational databases. However, unlike relational databases, these relationships are defined by the supporting individual and lists of references, and are all included in the conceptual database definition. Furthermore, the reference attributes supporting a relationship in the OO database definition are two-way.

Typically OO database systems involve a built-in OO programming language that is procedural [12, 13, 14], unlike non procedural relational database manipulation languages like SQL [9]. Because OO databases do not consist of relations, it is not pos sible to consistently apply SQL to them, although there has been a notable attempt for the O_2 OO system [11].

The lack of a generally accepted powerful non procedural language for OO databases is thus a serious shortcoming, and con-

siderable effort is currently being devoted to remedying this situation. Most attempts hithertoo have been based on modifications to SQL [11]. In this paper a quite radical alternative is presented, namely a language based on a modification of SQL/N.

SQL/N is a language that deals implicitly with objects [3]. It allows the use of natural quantifiers, and can be used, somewhat unconventionally, with relational databases. However, with relational databases SQL/N is deployed in a manner not in the spirit of the relational approach, because of its intrinsic object orientation; in reality an OO viewpoint, which is quite different from the relational viewpoint, is needed to use SQL/N effectively with relational databases. However, because of the intrinsic object orientation of SQL/N, it is a relatively easy matter to adapt it for use with OO databases.

The object orientation of SQL/N can be understood from the following example. Suppose a relational database about oil companies and wholely owned wells, as follows:

COMPANY (COMPNAME, INCORP)

WELL(COMPNAME , WELLID, TYPE)

The company attributes are company name and place incorporated. The well attributes are company that owns the well, well identification number and type of well (dry, oil, or gas). There is a one-to-many (also referred to as 1:n or parent-to-child) relationship between company and well, based on the relationship attribute COMPNAME.

This is clearly relational database, but because it is quite simple, we can view it in two distinct ways, either as a con-

ventional relational database , or as a crude OO database. Thus it can first be viewed simply as two distinct relations, with a common relationship field giving rise to a 1:n relationship. This is the conventional relational viewpoint. It can also be viewed, however, as a collection of company tuples, each of which is associated with a number of wells; in other words, it can be viewed as a collection of objects, where an object is a company and its associated wells.

Now consider the retrieval: Find the place of incorporation and name of each company whose wells are all dry.

The relational SQL expression deals with the two relations, and must deal with all WELL tuples, as follows:

```
SELECT COMPNAME, INCORP
FROM COMPANY
WHERE COMPNAME NOT IN (SELECT COMPNAME
                       FROM WELL
                       WHERE TYPE NOT = 'DRY');
```

This expression, although impeccable from a point of view of con ventional set theoretic logic, is hardly intuitively compelling, and has to be thought about. And unfortunately, there is no simpler way in SQL of expressing it.

The SQL/N expression deals only with the objects involved, that is, to understand it, think of a company object and its associated wells:

```
SELECT COMPNAME, INCORP
```

```
FROM [EACH] COMPANY [TUPLE]
WHERE FOR ALL RELATED WELL [TUPLES] (TYPE = 'DRY')
```

Thus the expression, although formal, is close to what we might construct in English, where normal thought processes are in terms of each individual object and objects related to that object.

As is well known, a major problem with SQL is incorporating the universal quantifier, and the reason for this is largely the non-object-orientation of the approach. However, because SQL/N is intrinsically OO, it is an easy matter to include not only the standard existential and universal quantifiers, but also the large number of natural quantifiers that occur in natural language, such as: for all but one, for a majority of, for at least six, for no, and so on. Thus the following retrieval involves the natural quantifier for most, or for the majority of:

Get the place of incorporation and name name of each company, the majority of whose wells are dry.

In SQL/N, the above SQL/N expression can be reused, if we merely change the quantifer FOR ALL to FOR MOST:

```
SELECT COMPNAME , INCORP
FROM [EACH] COMPANY [TUPLE]
WHERE FOR MOST RELATED WELL [TUPLES] (TYPE = 'DRY');
```

This is possible because of an expression of this type intrinsically deals with an object - a company, and any quantity (specified by the quantifier) of associated objects. Thus any of a large number of quantifiers fit in naturally.

In contrast, with SQL, the relations COMPANY and WELL must be considered in their entirety to handle this retrieval. We need to count the wells that are dry, and those that are not dry, for each company, and specify that the count of those that are dry exceeds the count of those that are not:

```
SELECT COMPNAME , INCORP

FROM COMPANY
WHERE (SELECT COUNT(*) FROM WELL

       WHERE TYPE = 'DRY'

         AND COMPANY.COMPNAME = WELL.COMPNAME)

   >

      (SELECT COUNT(*) FROM WELL

       WHERE TYPE NOT = 'DRY'

         AND COMPANY.COMPNAME = WELL.COMPNAME);
```

This somewhat oblique method of specification in SQL is the direct result of the set theoretic orientation of the language.

SQL/N was originally designed to manipulate relational databases. A severe disadvantage to the SQL/N approach with relational databases is the lack of any specification of a relationship in the relational database definition, for a method of referencing a specific relationship is needed in order to apply natural quantifiers to it. With SQL/N and relational databases a subterfuge or a patch [3], the details of which are not relevant here, was needed here to allow reference to relationships. With an OO database such a subterfuge is unnecessary since relationships are ex-

plicitly detailed in the OO database definition. This availability of relationship definitions allows easy adaptation of SQL/N constructs to form a powerful non procedural language for manipulating OO databases, as will be demonstrated. SQL is soundly based on conventional set theory, which requires only the existential and universal quantifiers. SQL/N is also soundly based, but on a set theory that allows the natural quantifiers [4]. This adaptation of SQL/N to OO data bases might be usefully called OSQL/N.

The project database

Before introducing the detailed OO database definition used for illustrating language constructs, a brief summary of OO concepts is in order. In the OO approach, an object has a unique identity that is independent of any values it contains [6,7]. An object normally has associated attributes (sometimes called "instance variables"), and, as in the relational approach, one of these attributes, or a group of them, may be regarded as a primary key. However, because every object has a unique identity, an object need not have a primary key. Instead, the database system will generate a unique object identifier, which may or may not be accessible by the user, depending on the database system employed.

As well as a possible primary key attribute, an object may have either simple attributes, such as a quantity or a name, whose type allows either literal numeric or alphnumeric values, or other defined values, such as the type DATE , indicating a date value. An object may also have collection attributes, such as sets or lists,

for example, the set of keywords in a document, or a list of object identifiers to support a relationship. The relational approach does not allow collection attributes.

These points can be illustrated by the database definition in Figure 1 for the project database, which concerns document management.

```
Document:  <
    doc#:        Document;
    title:       STRING;
    revised:     DATE;
    topic:       STRING;
    keyword:     SET[STRING];
    authlist:    LIST[Person];
    chaplist:    LIST[Chapter];>

Chapter:   <
    chap#:       Chapter;
    doc#:        Document;
    title:       STRING;
    npages:      INTEGER; >

Person: <
    pers#:       Person;
    doclist:     LIST[Document];
    pname:       STRING;
    position:    STRING;        >
```

```
Program: <
    prog#:      Document
    title:      STRING;
    lang:       STRING;
    runlist:    LIST[Run];  >

Run: <
    run#:       Run;
    prog#       Program;
    machine:    STRING;
    rundate:    DATE; >
```

Figure 1

With the exception of cyclic or recursive relationships, the database in Figure 1 contains the common types of relationships that are encountered in OO databases. It is a modified version of a database used by Cattell in a discussion of OO databases [7]. It has 1:n relationships as well as a binary many-to-many and a subtype (ISA) relationship, this last relationship permitting utilization of the inheritance concept in OO databases.

The main object type is **Document**, where each object represents a document. A document can have many chapters, with each chapter represented by a **Chapter** object.

A person can be an author of a document. A person can author many documents and a document can be authored by many persons.

A person is represented by a **Person** object and the object **Authact** enables the resulting many-to-many relationship between **Document** and **Person** objects.

The object **Program** represents a computer program. Since a program is a kind of document, the set of unique (system generated) object identifiers (**prog#** values) in **Program** objects are to be found among the set of unique object identifiers (**doc#** values) in **Document**. Thus the object types **Document** and **Program** form a subtype hierarchy, the relationship between the two being an ISA relationship. A **Run** object represents an execution of a program. Since a program can be executed many times there is a 1:n relationship between **Program** and **Run**.

Note the significance of the ISA relationship in the database. Because of this relationship, a **Program** object inherits not only the attributes of the corresponding **Document** but also each relationship in which **Document** participates. Thus we can have both legitimate requests involving attribute inheritance, such as:

"What are the titles of programs executed more than 100 times?"

and legitimate requests involving relationship inheritance:

"Who are the authors of C programs that have never been executed?"

The names of the attributes in Figure 1 were chosen to make the semantics self-apparent. Where this is not so, more detailed dis-

cussion later in the paper should clarify matters. Note that the system generated object identifier for each object type is specified in Figure 1 using the object type. Thus the object identifier **doc#** must have the type **Document**, and **chap#** the type **Chapter.**

Language constructs for manipulation of simple objects

To retrieve information from a single object type, the syntax of SQL can be used, except where a collection SET attribute is involved in a condition. Thus the following could specify the retrieval:

Get the titles of chapters with more than 10 pages:

```
select title [attribute] from [each] Chapter [object]
where pages > 10;
```

Where a diferent syntax is preferred, the essential syntactic terms have to be:

```
<<<item-retrieved><object type>> ... ><condition>
```

Syntax for the case where a collective attribute is involved is discussed in a later section.

Language constructs for 1:n relationships

Consider the portion database definition in Figure 1 that includes the 1:n relationship between **Document** and **Chapter** objects. In the object **Chapter**, the attribute **chap#**, although system generated, is taken as naming the object identifier for a chapter of a document. Accordingly, the collective attribute **chaplist** in **Document**, which is a list of **chap#** values, gives a list of the object identifiers of the chapters of that document, so that the type of **chaplist** must be LIST[**Chapter**]. Furthermore, in a **Chapter** object, there is an attribute **doc#** with the type **Document**, that is, its value must be a **Document** object identifier. The attributes **chaplist** and **doc#** are reference or relationship attributes. They are used instead of the primary and foreign keys of the relational approach, and precisely define the 1:n relationship between the objects **Document** and **Chapter**.

Now suppose we are dealing with a specific **Document** object. To specify a quantity of its chapters, that is, a quantity of its related **Chapter** objects the construct needed must specify a where condition or expression as follows:

<where-expression>:= <quantifier><related objects><condition>

Thus if we wanted to specify a document where all chapers had exactly 10 pages, this specification in principle has to be

<all-quantifier><related objects>(pages = 10)

In the syntax of a computer language, the quantifier symbol could be any common quantifier notation, such as FOR ALL ITS/FOR EACH OF

ITS. The condition specification would involve the attribute name, a relational operator, and a literal value, such as: (page = 10).

To specify the <related objects>, where in English the genitive expression <chapters of document> are used, a precise relationship specification is needed, since there could be more than one relationship between two objects. To explicitly specify the relevant object of the relationship, only the reference list **chaplist** is needed, but for ease of reading, the object name can be included. Some syntactic possibilities for <related objects> are:

<where-expression>:=

<quantifier><quantified xreference><condition>

<quantified xreference>:=

<child reference list><object name>[objects]

<quantified xreference>:=

<object name> [objects] [[listed] in] <child reference list>

These possibilities can be illustrated by the retrieval:

Get the document title for each document with at least 4 chapters with more than 10 pages.

In this case the required natural quantifier is FOR AT LEAST ITS 4:

select title from Document

where for at least its 4 chaplist Chapter [objects] (pages > 10)

```
select title from Document
where for at least its 4 Chapter [objects] [listed] in chaplist
                                                    (pages > 10)
```

There are clearly many other syntactic possibilities, although only one semantic possibility. For the purposes of this paper we settle on the first, as it appears to be the most easily understood by humans.

If the quantifier in the retrieval above is changed, to FOR MOST OF ITS, for example, only the quantifier in the OO predicate need be changed, as in:

```
select title from [each] Document [object]
where for most of its chaplist Chapter objects (pages > 10);
```

A common case involves retrieval of the parent object conditional upon both parent object attributes and related child objects. The same <quantified xreference> syntax as above is used, except for the need to insert a simple condition. A simple example shows how this is easily done: Retrieve the names of documents about databases, where all chapters have at least 10 pages.

```
select title from Document
where topic = 'databases'
  and for all of its chaplist objects (pages > 10)
```

In this case the inserted condition is: topic = 'databases'.

A formal syntax for where-expressions involving quantification of related objects could therefore be as shown in Figure 2:

```
<where-expression>:=
        <condition> <op> <quantifed xreference>
<quantified xreference>:=
              <reference><object name> [objects]<condition>
<op>:= AND/OR
<reference>:= <parent identifier>/
               <child reference list>
```

Figure 2

All of the above quantifier retrieval examples involved retrieving data from a parent object, given conditions in an associated child object, with a 1:n relation. In such expressions we used the syntax variable <child reference list>. The converse case involves retrieval of a child, given conditions for the parent. Since for a given object, there can be only one parent, or, if the database integrity constraints allow it, no parent, only two natural quantifiers are needed to cover almost all contingencies, namely, FOR ITS [ONE/ONLY] and FOR NO.

Exactly the same syntax as shown in Figure 2 above can be used for child retrieval with a conditional parent, except that instead of the syntax variable <child reference list>, which specifies the list (such as **chaplist**) of related child objects, we

use the syntax variable <parent identifier>, which specifies a reference (such as **doc#**) to the parent entity. Although the syntax for <quantified xreference> is shown in Figure 2 in <where-expression> as:

<quantified xreference>:=

<reference><object name>[object]<condition>

the following is another equivalent possibility:

<quantified xreference>:=

<object name> [object] [in] <parent identifier><condition>.

These possibilities can be illustrated by the retrieval:

Get the number of pages for each chapter for the document entitled 'database structures'.

We use the almost trivial quantifier FOR ITS [ONE]:

```
select chapter#, pages from Chapter
where for its one doc# Document [object]
                        (title = 'database structures');
```

```
select chapter#, pages from Chapter
where for its one Document [object] in doc#
                    (title = 'database structures);
```

Once more, there are clearly many other syntactic possibilities.. For the purposes of this paper we settle on the first, as it ap-

pears to be the most easily understood by humans. It was shown in the formal syntax above for <where-expression> in Figure 2.

Another common type of retrieval expression involves retrieval of a child object where there is a condition in the child object attributes, as well as a condition in the parent. We have sufficient syntax in <where-expression> above to cover this, as illustrated by the retrieval:

> Get the names of chapters with more than 10 pages in documents on databases.

```
select title from Chapter
where (pages > 10)
  and for its one doc# Document [object] (topic = 'databases');
```

A further retrieval possibility involving a 1:n relationship concerns retrieval from both the parent and child objects, for example, in the retrieval:

> Get the document name, chapter name, and number of pages for each chapter with more than 100 pages in a document about data bases.

The syntax already presented covers this:

```
select title, pages from Chapter, and title from Document
where pages > 100
and for its one doc# Document [object] (topic = 'databases')
```

With a 1:n relationship, a retrieval condition can require the non existence of related child objects. For example, suppose we have:

Get the titles of documents about databases that have no chapters.

```
Select title from Document
where topic = 'databases' and
 for none of its chaplist Chapter objects (chap# > NULL)
```

We need a simple way of specifying non existence. One simple way would be a NULL value such that all object identifier values for existing objects exceed it. Then a simple test for existence would be a condition that the object identifier exceeded NULL. This is illustrated by the above retrieval expression.

Language constructs for many-to-many relationships

In the OO approach, a many-to-many relationship can involve either two objects, in the case where there is no intersection data, or three objects, for the case where the third object type concerns intersection data. Consider now the many-to-many relationship between a document and a person, where one or more persons can author one or more documents. The two-object version of the relationship is illustrated by the database definition in Figure 3a, as abstracted from Figure 1, with the three object version in Figure 3b.

```
Document:  <
    doc#:        Document;
```

```
    title:      STRING;
    revised:    DATE;
    keyword:    SET[STRING];
    chaplist:   LIST[Chapter];
    authlist:   LIST[Person];  >

Person: <
    pers#:      Person;
    doclist:    LIST[Document];
    pname:      STRING;
    position:   STRING;        >
```

Figure 3a

```
Document:  <
    doc#:       Document;
    title:      STRING;
    revised:    DATE;
    keyword:    SET[STRING];
    chaplist:   LIST[Chapter];
    authlist:   LIST[Person];
    actlist:    LIST[Activity];  >

Authact:  <
    act#:       Authact;
    doc#:       Document;
    pers#:      Person;
```

```
    payment:    INTEGER; >

Person: <
    pers#:      Person;
    doclist:    LIST[Document];
    actlist:    LIST[Authact];
    pname:      STRING;
    position:   STRING;         >
```

Figure 3b

In the case of no intersection data, the OO approach simply treats the many-to-many relationships as two symmetric one parent for many children relationships. Thus the constructs for retrieving a parent given conditions involving the children, as developed in the previous section, may be used with this kind of many-to many relationship - specifically we may use the <where-expression> syntax given in the previous section in Figure 2. As an example, suppose the retrieval:

Get the documents about databases with more than two authors who are systems analysts.

This can be expressed:

```
select title from Document
where (topic = 'databases')
  and for at least its 2 personlist Person objects
                          (position = 'systems analyst')
```

A converse retrieval would be:

Get the name each engineer who has never authored any documents about computers.

```
select pname from Person
where (position = 'engineer')
and for none of its doclist Document objects (topic = 'computers');
```

In the case of intersection data, there are simply two symmetric 1:n relationships, referring to Figure 3a, between **Document** and **Authact**, and between **Person** and **Authact**. These can be handled like normal 1:n relationships. However, some retrievals will require the use of all three objects, and a nesting of quantified expressions. The where-expression formalism given in the previous-section does not cover unlimited nesting of quantified cross references, or even no quantified cross reference. Two minor changes, as shown in Figure 4, allows it to do so, and thus give it quite general retrieval power with both 1:n and many-to-many relationships.

```
<where-expression>:=
     <condition>/<condition> <op> <quantifed xreference>
quantified xreference:=
             <reference><object name> [objects]<where-expression>
op:= AND/OR
<reference>:= <parent identifier>/
              <child reference list>
```

Figure 4

For example, consider the retrieval:

> Retrieve the name of each document about computers written by one or more systems analysts, all of whom were paid more than $100.

A further level of nesting is needed. By nesting the constructs already developed, in compliance with the <where-condition> syntax above, we get the expression:

```
select title from [each] Document [object]
where topic = 'databases'
  and for each of its authlist Authact objects
       ((payment > 100) and for its [one] pers# Person object
                                  (position = 'systems analyst'))
```

It is useful to compare this expression with the equivalent relational SQL expression, if the database were relational (i.e. all collection attributes are omitted):

```
SELECT TITLE FROM DOCUMENT
WHERE TOPIC = 'DATABASES' AND DOC# NOT IN
     (SELECT DOC# FROM AUTHLIST WHERE
          PAYMENT NOT > 100) OR PERS# NOT IN
                        (SELECT PERS# FROM PERSON
                         WHERE POSITION = 'SYSTEMS ANALYST'));
```

The nesting does not take place in a natural manner, and what is more, the required negation of the implicit existential quantifier means that we have to negate a complex predicate, and use De Morgan's rules for negation of compound expresssions to do it correctly (examine carefully the use of the logical operator OR), none of which complication is necessary with the OO version above.

Language constructs for collection SET attributes

There is a (non-referential) collection SET attribute in the **Document** object, namely **keyword**, as in Figure 1. Clearly, the relationship between documents and keywords is implicitly many-to-many. Since **keyword** is a set, it can participate in set-theoretic conditions. Thus, to handle sets, a condition in the syntax expression in Figure 4 must allow for both a relational condition and a set-theoretic expression. Thus the retrieval:

What are the names of documents with keywords 'tax' and 'file'?

could be expressed:

```
Select title from Document
where ('tax', 'file') in keyword;
```

Here the connective 'in' denotes set inclusion, as in SQL, so that we have the set theoretic condition that the set ('tax', file') is contained within the set **keyword**. The converse retrieval:

What are the document titles and keywords for documents about

databases?

could be expressed:

```
Select title [attribute], keyword [set] from Document
where topic = 'databases'.
```

Set-theoretic constructs with relationships

Even when not dealing with collection (set) attributes, there is still a general need for set theoretic constructs with relationships in OO manipulation languages, which we now address. For example, the retrieval:

Retrieve the title of each document about databases where the texts: 'objects', 'relations', 'keys', 'lists' are used for chapter titles.

applied to the database in Figure 1 is set oriented. The retrieval cannot be carried out using the <where-condition> syntax in Figure 4. In passing it might be noted that SQL is ideally suited to such retrievals, since it has a set-theoretic basis. Thus there is a need to develop set-theoretic constructs for both 1:n and many-to-many relationships with OO data manipulation languages.

Considering the retrieval above, a database document is retrieved if the titles of its set of **Chapter** objects contains a specified list of values. The problem is a syntax for the expression 'the (set of) titles of its set of **Chapter** objects'. A

suitable syntax for this that, in addition, allows for further nesting of expressions, is:

set [of]<child-attribute> [from] <child reference list>
<objectname> [objects]/
set [of]<child-attribute> [from] <child reference list>
<objectname> [objects] where <where-expression>

The second version allows for nesting.

The retrieval can now be constructed:

```
Select title from Document
where topic = 'databases' and
('objects', 'relations', 'keys', 'lists')  in
 set [of] title [from] chaplist Chapter [objects]
```

The second nested syntax expression above allows for much more sophisticated, if unlikely, expressions:

Retrieve the title of each document about databases where the texts: 'objects', 'relations', 'keys', 'lists' are used for the titles of chapters, provided the chapters have less than 50 pages.

```
Select title from Document
where topic = 'databases' and
('objects', 'relations', 'keys', 'lists')  in
```

```
set [of] title [from] chaplist Chapter [objects] where
                                        npages < 50
```

This is, of course, very similar to the structure of the equivalent expression in SQL, which, since it is fundamentally a set-theoretic language, allows for the most general set theoretic conditions in a simple and consistentmanner:

```
SELECT TITLE FROM DOCUMENT
WHERE TOPIC = 'DATABASES'
  AND ('OBJECTS, 'RELATIONS', 'KEYS', 'LISTS') IN
        (SELECT TITLE FROM CHAPTER
         WHERE CHAPTER.DOC# = DOCUMENT.DOC#
           AND NPAGES < 50)
```

In retrievals involving three objects, it is conceivable that the condition (in the above example: npages < 50) could expand to include quantified related objects. The syntax presented allows for this.

If we incorporate this additional syntax for general set-theoretic conditions into that given in Figure 4, a complete where-expression results, as shown in Figure 5:

```
<where-expression>:=
     <condition>/<condition> <op> <quantifed xreference>
quantified xreference:=
             <reference><object name> [objects]<where-expression>
```

op:= AND/OR

<reference>:= <parent identifier>/<child reference list>

condition:= <relational condition>/<set-theoretic condition>

set-theoretic condition:=

<collection attribute> in <set of values>/

<set of values> in <collection attribute>/

<set of values > in <set of attribute values>

set of attribute values:=

set [of] <child-attribute> [from] <child reference list>

<objectname> [objects]/

set [of] <child-attribute> [from] <child reference list>

<objectname> [objects] where <where-expression>

The syntax is clearly recursive and can be expanded indefinitely, to allow for any level of nesting of expressions. It could serve as the basic definition of an OSQL/N predicate.

Language constructs for handling inheritance

No new language constructs are needed for manipulation of inheritance. For example, consider the retrieval:

Get the names of authors of C programs that have never executed on machine m42.

Between **Person** and **Document** there is a many-to-many relationship, and therefore, via inheritance, also between **Person** and **Program**, which is the relationship required for the retrieval. Furthermore, the LIST attribute **personlist** in **Person** holds object identifiers of

Program objects, since these are also object identifiers of **Document** objects. The retrieval can therefore be easily expressed as follows:

```
Select pname from Person
where for at least 1 of its doclist Program objects
(lang = 'C' and for none of its runlist Program objects
                                        (machine = 'm42'))
```

All expressions involving inheritance can thus be handled in this way.

Summary

Language constructs for an OO non procedural data base manipulation language, that might be called OSQL/N, has been presented. Languages based on these constructs can manipulate quantified relationships between objects in a manner close to natural language manipulation of objects, and in a manner that allows for the use of natural quantifiers. This is in contrast to SQL, which manipulates relationships between objects in a set-theoretic manner that re quires specification of which sets of objects are involved and which are not involved. Languages based on this template can also handle expressions that are primarily set-theoretic in nature - an area where SQL is naturally suitable - but do so in an open-ended manner that allows set-theoretic conditions to be further qualified, if required, by conditions involving naturally

quantified relationships between objects. In consequence, an OSQL/N language can be expected to have at least the retrieval power of SQL, and be easier to read, since it would be closer to natural language in construct.

REFERENCES

1. Abiteboul, S., Hall, R. IFO, a formal semantic data base model, ACM Trans. on Database Systems, 12 (4), 1987.

2. Bancilhon, F., et al. The design and implementation of O_2, an object-oriented DBMS, in "Advances in Object Oriented Database Systems," K. R. Dittrich, ed., Computer Science Lecture Notes 334, Springer Verlag, New York, 1988.

3. Bradley, J. SQL/N and attribute/relation associations implicit in functional dependencies, Int. J. Computer & Information Science 12(20), 1983.

4. Bradley, J. Recursive relationships and natural-quantifier set theoretic expression techniques, Computer Journal, to appear, 1992.

5. Bret, R., et al. The Gemstone Data Management System, in "Object-Oriented Concepts, Databases and Applications, W. Kim, F.H. Lochovsky, Eds., Addison-Wesley, Reading, Mass, 1988.

6. Cardenas, A. F., McLeod, D. "Research Foundations in Object-Oriented and Semantic Databases," Prentice Hall, Englewood Cliffs, New Jersey, 1990.

7. Cattel, R. G. G. "Object Data Management" . Addison Wesley, 1991.

8. Codd, E. F. Relational databases, a practical design for productivity, CACM, 25(2), 1982, 109-117.

9. Date, C. J. "Introduction to Database Systems, 5th ed., Addison Wesley, Reading, Mass., 1990.

10. Kim, W. et al. Features of the ORION object-oriented DBMS, in "Object-Oriented Concepts, Databases and Applications, W. Kim, F.H. Lochovsky, Eds., Addison-Wesley, Reading, Mass, 1988.

11. Lecluse, C., Richard, P., and F. Velez. O_2, an object-oriented data model, Proc. ACM SIGMOD Conference, 1989.

12. Object Design. ObjectStore Reference Manual, Object Design, Inc., Burlington, Mass., 1990

13. Objectivity. Objectivity Database Reference Manual, Objectivity Inc., Menlo Park, California, 1990.

14. Ontologic. ONTOS Reference Manual, Ontologic Inc., Billerica, Mass., 1989.

15. Versant Object Technology. Versant Reference Manual, Versant Object Technology Inc., Menlo Park, California, 1990.