A Deductive Declarative Object-Oriented Data Model and Query Language based on Narrowing

Zeki O. Bayram
Computer Engineering Department
Bogazici University
Bebek 80815/Istanbul-Turkey
internet: bayram@boun.edu.tr
Fax: 90 212 287 2461
Tel: 90 212 263 1500

Barrett R. Bryant
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294-1170, USA
internet: bryant@cis.uab.edu
Fax: 1 205 934-5473
Tel: 1 205 934-2213

Faik Hakan Bilgen
Computer Engineering Department
Bogazici University
Bebek 80815/Istanbul-Turkey

Abstract
The pure object-oriented data model is not convenient for describing inter-object relationships and can result in very involved and hard to understand queries. Augmenting the object-oriented model with declarative description of the relationships between objects using conditional rewrite rules is a convenient way to address the problem. Under this scheme, inter-object relationships are described declaratively using conditional rewrite rules, and queries are posed also declaratively in the form of equations to be solved. In this paper, we describe an experimental system written in Smalltalk which demonstrates the feasibility and appeal of this approach. Various examples are also given.

Keywords: Declarative, object-oriented, query, data model, database, deductive

1. Introduction

In the object-oriented model of data, real world entities whose representation is needed in the database are directly represented as database objects that are instances of some class. Each object has a certain set of attributes, and a unique object identifier.

Although in this model real world objects are represented very authentically, external relationships among objects are necessarily represented through the use of pointer valued attributes, which necessitates the physical traversal of pointers representing relationships in answering a query. This is less than ideal, since external relationships among objects are declarative, and should be handled that way. In fact, ideally the only pointer valued attributes in an object should be those which relate a part to its component subparts.

We present a solution to this problem which utilizes conditional rewrite rules to represent relationships among objects and a separate object-oriented module for dealing with objects. In [7] we defined a deductive database model (DataFunLog) based on conditional rewrite rules and showed how conditional rewrite rules can be used to declaratively define and query deductive databases. There was no object-oriented functionality in DataFunLog though. The model we develop here, similar to DataFunLog in that both use conditional rewrite rules to define deductive databases, does have object-oriented functionality, permits the posing of queries in the form of equations to be solved, and inter-object relationships can be described through conditional rewrite rules in the model. The operational semantics for answering queries is narrowing and e-unification through transformations.
In terms of implementation, this model is the result of a synthesis between a functional/logic/object-oriented programming language (FLOOP [8]) and a pure object-oriented database system NGO (for Next Generation Opal). Since both NGO and FLOOP are implemented in Smalltalk, their integration has been seamless.

2. Components of the System

In this section we describe the components of the declarative object-oriented database system.

2.1 Next Generation Opal

Next Generation Opal (NGO) is a minimal object-oriented database system. It is modeled after the Gemstone object-oriented database system [9] which uses the Opal query language. NGO provides a type system for the attributes of objects, as well as sets whose contents can only be of a certain type. The system automatically defines methods to access the attributes of objects and set those attributes to certain values. A subset of the attributes of objects belonging to some class can be specified to form a key. If a set contains objects of a given type, and these objects have a certain set of attributes defined as a key for the objects, this information causes the automatic definition of a method for this set of attributes, which, when given a set of values for keys, associatively searches for an object in the set whose key matches the argument given to the method. This is not necessarily part of the object-oriented data model, but it facilitates writing queries. Of course, the user can define any other methods for any class.

<table>
<thead>
<tr>
<th>Object subclass: #Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>instanceVariableNames: 'attributeValuePairs'</td>
</tr>
<tr>
<td>classVariableNames: 'KeyDictionary TypeDictionary'</td>
</tr>
<tr>
<td>poolDictionaries: 'CharacterConstants'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set variableSubclass: #MySet</th>
</tr>
</thead>
<tbody>
<tr>
<td>instanceVariableNames: ''</td>
</tr>
<tr>
<td>classVariableNames: 'SetTypeDictionary'</td>
</tr>
<tr>
<td>poolDictionaries: 'CharacterConstants'</td>
</tr>
</tbody>
</table>

Figure 1: Definition of the classes Tuple and MySet

NGO is implemented through the definition of two new classes and the necessary methods in Smalltalk. These classes are called Tuple and MySet and are depicted in Figure 1. They act as abstract superclasses: they are not meant to have instances of their own, but rather provide the methods their subclasses need. User defined object types (in the relational model, these are called relation schemes) are defined as subclasses of Tuple. Entity sets (alternatively, sets of objects) are defined as subclasses of MySet, whose contents are instances of some subclass of Tuple. Note that in Smalltalk, all references to objects are via pointers, and adding an object to a set means adding its pointer to an object to the set. Hence, an object can be "in" more than one set at any given time.

2.2 The FLOOP language

The language of implementation of FLOOP, as well as the underlying object-oriented component, is Smalltalk.

FLOOP programs consist of conditional rewrite rules of the form \( f(a_1,a_2,\ldots,a_n) : (b_1=d_1, b_2=d_2, \ldots, b_m=d_m) \rightarrow \text{rhs} \) where \( a_1, a_2, \ldots, a_n, b_1, b_2, d_1, d_2, \ldots, b_m, d_m \) and \( \text{rhs} \) are all terms, and \( f \) is a function name. \( a_1, a_2, \ldots, a_n \) are not allowed to contain any function names. The operational semantics for FLOOP is narrowing and e-unification through the method of
transformations [8,13,17]. The transformations employed implement an innermost narrowing strategy and can handle arbitrary object-expressions by making calls to an underlying object-expression evaluator. The interpreter for FLOOP is written in Smalltalk/V [1] and the Smalltalk interpreter is the underlying object-expression evaluator.

2.3 Marriage of NGO and FLOOP
Since a rewrite rule in FLOOP can contain any kind of Smalltalk object, these objects can naturally be instances of subclasses of Tuple, and the integration of two tools comes naturally.

3. Problem Definition
In this section, we specify a simple but realistic database scheme which will be the basis for comparison of the pure object oriented model and the declarative model developed in this paper. The problem is to represent three sets of objects, namely courses, students and teachers, the relationships among them and answer queries about the objects. There are only two explicit relationships among objects: teachers teach courses and students take courses. The relationship between teachers and courses is one-to-many and the relationship between students and courses is many-to-many.

The attributes of a teacher are name, age, sex and salary. The attributes of a student are name, age, sex and GPA. The attributes of a course are name and timeAt (the time the course is taught at). Given this data definition, we would like to pose the following queries:

- What courses does student s1 take that are taught by teacher t2?
- Who are all the students of teacher t1?
- What other teachers teach a course at the same time that teacher t1 teaches a course?

4. Solution Using NGO (Purely Object-Oriented)
In this section, we show a purely object-oriented solution to the problem described in the previous section. We explain each section of the code as we proceed.

```
Tuple subclass: #Person
    attributeName: 'name age sex'
    attributeType: 'Symbol Integer Character'
    key: 'name'.
```

Create a subclass of Tuple, called Person, with attributes name, age and sex. The attributes have type (class) Symbol, Integer and Character respectively (note that the hash mark in front of a sequence of characters makes it a symbol in Smalltalk). Objects of type Person have a key field, name. Person is an abstract superclass, since we shall not use it for creating objects, but rather for defining the attributes that are common to both Teacher and Student classes. Similarly, we have the definition of the Course entity type.

```
Tuple subclass: #Course
    attributeName: 'name timeAt'
    attributeType: 'Symbol Integer'
    key: 'name'.
```

Next we have the definition of Teacher and Student, both descendants of Person.

3
We see that the key attributes for Teacher and Student are empty. This is because both Teacher objects and Student objects inherit the key of Person objects, i.e. name. In general, the key attributes inherited from a parent are appended to the list of the key attributes defined in a class. In this case, no attribute is defined as a key for Teacher, so the set of attributes comprising the key for a teacher is consists only of "name" which is inherited from Person. Also note that an attribute called teaches was necessary in order that the relationship between teachers and the courses could be established (the chosen scheme of representing the relationships among different objects is by no means the only one possible, but we believe it is a reasonable one). For Student objects, an attribute takes of type CourseSet has been defined which will contain the Course objects that a student takes.

These definitions define an instance of CourseSet to be a set containing Course objects, an instance of StudentSet to be set containing Student objects and an instance of TeacherSet to be a set of Teacher objects. Instances of these classes will be depositories for the objects we shall create in the database, as well as being used in setting up the relationships among objects.

Declare the local variables to be used in the ensuing program segment and create the containers for objects.

Create a student object, and add it to the students set. Note that attribute names are used by NGO to automatically define methods by the same names for instances of a class. As an example, the methods age and age: are defined for instances of class Person. The first method returns the age attribute of the receiver, whereas the second one sets the age attribute to some value (of type Integer, or an object that is an instance of a subclass of Integer). These methods are inherited by subclasses of Person, that is why a Student object understands the messages given to it above. Other Student objects are defined similarly.

| teachers students courses answer1 answer2 answer3 |
| teachers := TeacherSet new. |
| students := StudentSet new. |
| courses := CourseSet new. |

| students add: ( (Student new) name: #s1; age:20; sex:$M; takes:(CourseSet new); gpa: (3.5) ). |

| teachers add: ( (Teacher new) name: #t1; age:37; sex:$F; |
| teaches: (CourseSet new); salary: 10000 ). |
Add the teacher objects to the depository for such objects (i.e. the set \textit{teachers}). Other teachers are added similarly.

\begin{verbatim}
courses add: ( (Course new) name: #c1; timeAt:10 ).
\end{verbatim}

Add the course objects to the depository for such objects (i.e. the set \textit{courses}).

\begin{verbatim}
((students name: #s1) takes) add: (courses name:#c1).
((students name: #s1) takes) add: (courses name:#c4).
\end{verbatim}

Object creation has now been completed. Above, we define the relationships among objects. The student whose name is \texttt{s1} takes the course with the name \texttt{c1}. Note that the method \texttt{name:} has been automatically defined for instances of class \texttt{StudentSet} since \texttt{name} is the key of a \texttt{Student} object. This method \texttt{name:anObject}, which belongs to the \texttt{students} set, returns an object that \texttt{students} contains whose \texttt{name} attribute matches \texttt{anObject}. Similarly with the set \texttt{courses}.

More generally, if a set S\textsubscript{1} (an instance of a subclass of \texttt{MySet}) contains objects that are instances of some class C\textsubscript{1} that is a descendant of \texttt{Tuple} and C\textsubscript{1} has a key consisting of attributes a\textsubscript{1}, \ldots, a\textsubscript{n}, then the method \texttt{a\textsubscript{1}: \ldots, a\textsubscript{n}:} is defined for S\textsubscript{1} automatically by the system. It returns the object in S\textsubscript{1} whose key matches the argument of a\textsubscript{1}: \ldots, a\textsubscript{n} (if one exists).

\begin{verbatim}
((teachers name: #t1) teaches) add: (courses name:#c1).
((teachers name: #t1) teaches) add: (courses name:#c3).
\end{verbatim}

Teacher \texttt{t1} teaches courses \texttt{c1} and \texttt{c2}. The way it works is as follows: the object for \texttt{t1} is found in the set \texttt{teachers}, the attribute \texttt{teaches} of that object is obtained (which is a set), and into that set the object for \texttt{c1} (which comes from the set \texttt{courses}) is added. Next, we have the queries implemented as Smalltalk programs.

\textbf{4.1 What courses does s1 take that are taught by t2?}

\begin{verbatim}
answer1 := CourseSet new.
((teachers name: #t2) teaches)
do: [:course1 |
   ((students name: #s1) takes)
do: [:course2 | (course1=course2)
   ifTrue:
      [ answer1 add: course1 ] ]].
\end{verbatim}

\textbf{4.2 Who are the students of t1?}

\begin{verbatim}
answer2 := StudentSet new.
students
do: [:aStudent |
   ((teachers name: #t1) teaches)
do: [:course1 |
   (aStudent takes)
do: [:course2 |
   (course1 = course2)
   ifTrue: [ answer2 add: aStudent ]]]].
\end{verbatim}
4.3 What other teachers teach a course at the same time that t1 teaches a course?

answer3 := TeacherSet new.
teachers do:
  [ :aTeacher]
    (aTeacher teaches) do:
      [ :course1]
        ((teachers name:#t1 teaches) do:
          [ :course2]
            ((course1 timeAt)=(course2 timeAt)
              and: ![((aTeacher name)=#t1 not) ])
            ifTrue:[answer3 add: aTeacher]].

4.4 Analysis of the Purely Object Oriented Solution

We note that in all these queries, we had to traverse sets of objects, perform tests on objects, and place the resulting objects in an answer set. It is obvious that in all the above cases, the intended meaning of the query is not readily observable from the code implementing the query, and the level of nesting of iterations can be deep, even for simple queries.

5. Solving the Problem Declaratively

In this section we give a declarative solution to the specified problem. We first proceed with the definition of Teacher, Student and Course classes, as in the NGO solution. Note that the set valued attributes teaches and takes are no longer needed, neither do we need to define keys.

| Tuple subclass: #Person
  attributeNames: 'name age sex'
  attributeTypes: 'Symbol Integer Character'. |

| Tuple subclass: #Course
  attributeNames: 'name timeAt'
  attributeTypes: 'Symbol Integer'. |

| Person subclass: #Teacher
  attributeNames: 'salary'
  attributeTypes: 'Integer'. |

| Person subclass: #Student
  attributeNames: 'gpa'
  attributeTypes: 'Float'. |

We then declare the local variables for the Smalltalk code to follow. Then, objects are created and relationships among them specified. rb is the set of rewrite rules.

```
| rb answer1 answer2 answer3 |
rb := RuleBase new2.
rb addRules: ' 
  students([#s1]):() -> { (Student new) name: #s1; age:20; sex:$M; gpa: (3.5) }.

.........
```

Create the Student objects. Each student object is given a tag or an object identifier that identifies the object uniquely. For example, the student whose name is s1 is given the tag s1.
The tags shall be used in relating objects to one another. Note that we did not have to use student names as object identifiers; we could have used any other symbol instead. But under the circumstances, using them is more convenient and suggestive of the object the tags represent.

Create the Teacher objects.

```
teachers(#t1);( ) ->
   { (Teacher new) name: #t1; age:37; sex:$F; salary: 10000 }.
```

Create the Course objects. We are now ready to describe the relationships among objects.

```
courses(#c1);( ) ->
   { (Course new) name: #c1; timeAt:10 }.
```

Define the takes relationship. For example, the first two rules above describe the fact that Student object s1 takes Course objects c1 and c4.

```
takes(#s1);( ) -> { #c1 }.
takes(#s1);( ) -> { #c4 }.
```

Here we define the teaches relationship between Teacher objects and Course objects. The first two rules, for example, describe the fact that Teacher object t1 teaches Course objects c1 and c3.

This is all that is required to create objects and relationships between objects. Next, we come to the queries. This is where we shall see the real difference between stating the query declaratively and imperatively (as was the case in the last section).

5.1 What courses does s1 take that are taught by t2?

```
answer1 := rb solveEquations: ' ( takes(#s1) = C, teaches(#t2) = C, courses(C)=C2 ) '
   instantiating: '(C2)'.
```

The rule base rb that contains the rules making up the database is sent the message solveEquations:instantiating: which causes the invocation of a method that solves the equations given in the first argument and instantiates the variables specified in the second argument with the computed substitution(s). The answer returned is a set of ordered lists of terms. Terms in this context are object terms.

Note in the query the use of logical variables C and C2. Logical variables always start with a capital letter in FLOOP. In the above query, we are interested only in the values that C2 will be bound to, so that is the only variable that is specified in the argument to instantiating.
In plain English, we could read the query as: Find all \( C_2 \) such that \( C \) is (the tag of) a course taken by \( s_1 \), \( C \) is (the tag of) a course taught by \( t_2 \) and \( C_2 \) is the actual course object represented by \( C \).

5.2 Who are the students of \( t_1 \)?

\[
\text{answer2} := \text{rb solveEquations: } '( \text{ teaches(#t1)}=C_1, \text{ takes(S)}=C_1, \text{ students(S)}=S_2 )' \\
\text{instantiating: } '(S_2)'.
\]

In plain English, the query says: Find all \( S_2 \) such that \( t_1 \) teaches \( C_1 \), \( S \) takes \( C_1 \) and \( S_2 \) is the actual object represented by \( S \).

5.3 What other teachers teach a course at the same time that \( t_1 \) teaches a course?

\[
\text{answer} := \text{rb solveEquations: } ' \\
\text{ teaches(#t1)}=C_1, \\
\text{ courses(C1)}=C_2, \\
\text{ teaches(T)}=C_3, \\
\text{ NOT}(T=#t1)=\text{true}, \\
\text{ courses(C3)}=C_4, \\
\text{ MSG}(C_2,#\text{timeAt}) = \text{MSG}(C_4,#\text{timeAt}), \\
\text{ teachers(T)=T_2 }' \\
\text{instantiating: } '(T_2)'.
\]

This query can be stated in English as: Find all \( T_2 \) such that \( C_1 \) is a course taught by \( t_1 \), \( C_2 \) is the actual Course object represented by \( C_1 \), another teacher \( T \) teaches \( C_3 \), \( T \) is different from \( t_1 \). \( C_4 \) is the Course object represented by \( C_3 \), \( C_2 \) and \( C_4 \) are taught at the same time, and \( T_2 \) is the Teacher object represented by \( T \). MSG(C_2,#timeAt) sends the message timeAt the object denoted by \( C_2 \). Remember that timeAt is an attribute of courses objects and all attributes are automatically defined as methods in NGO.

6. Discussion

Looking back at the previous sections of this paper, we have the following observations.

- We see that in the transition from the pure object-oriented data model to a combined functional/logic/object-oriented model, the following changes have happened.
  - Sets as depositories of objects are no longer needed. Function names act as classifiers of objects. For example, the set teachers has been replaced by the (multi-valued) function teachers.
  - Explicit object identifiers permit the declarative definition of the relationship among objects. The object identifier is in the form of a constant in the argument of the function whose body creates the object.
- The clarity of the queries expressed in the combined functional/logic/object-oriented data model is far superior to the same queries expressed in the pure object-oriented data model.
- In the combined scheme, we have lost nothing of the advantages that the pure object-oriented model offers (classes, class hierarchies, inheritance, encapsulation, methods, etc.). We have taken these features, and put them in a context where declarative description of inter-object relationships and declarative posing of queries are possible.
- And finally, the examples presented do not involve recursively defined rules, but recursion is fully permitted in FLOOP and thus is available in this combined functional/logic/object-oriented data model.
7. Related Work

Recently, the area of deductive databases and logic programming [12, 16] has been extended to incorporate object-orientation and various deductive object-oriented database languages have been proposed.

COL (Complex Object Language) [2,3,4,5] is a rule-based database language for complex objects. It is an extension of Datalog [18]. Complex objects in COL are typed trees constructed recursively using tuple and set constructors. A distinguishing characteristic of COL is the availability of data functions. Lou and Ozsoyoglu [14] propose extending Horn clause logic languages with object-oriented features in the language LLO. LOGRES [10] is a database system which supports classes of objects, with “generalization hierarchies” and object sharing. It is rule-based, extends Datalog [18] to support sets, multiset, sequences and “controlled forms of negation.” It permits queries and queries and updates through the rule-based paradigm. Object id unification is possible.

A different approach to incorporating object oriented functionality into deductive databases is taken in [11, 15] where higher order logic, which can be mapped into first order logic, is proposed. These schemes are based upon extending predicate calculus with higher order syntax which increases the modeling and programming capabilities of deductive databases.

The MOOD project, described in [6] attempts to integrate concepts of object-oriented and deductive databases by “providing powerful knowledge modeling techniques via class hierarchies of complex objects, dealing with object-bases primarily in a non-procedural, declarative way, handling large object bases efficiently.”

8. Conclusion and Future Research Directions

We described a combined functional/logic/object-oriented database model that provides for the declarative description of inter-object relationships through conditional rewrite rules and permits queries to be posed declaratively in the form of a set of equations to be solved.

We demonstrated the utility of our approach in an example. Our example made clear that in the pure object-oriented model, the necessity to represent inter-object relationships with pointer valued attributes makes the representing of inter-object relationships and posing of queries difficult to write and understand. On the other hand, representing inter-object relationships using conditional rewrite rules and posing queries declaratively in the form of equations to be solved was seen to be very natural and convenient.

Our system at this point is a research tool, designed to demonstrate the workability of our idea. As such, it lacks many of the features needed in a full featured database system (persistent objects, concurrency control, integrity constraints, schema evolution, etc.). Any future work would include the incorporation of these features in our system without changing the declarative nature of the representation of inter-object relationships and declarative posing of queries.

References


