A Conceptual Language for Querying Object Oriented Data

Peter J Barclay and Jessie B Kennedy

Computer Studies Dept., Napier University
219 Colinton Road, Edinburgh EH14 1DJ

Abstract. A variety of languages have been proposed for object oriented database systems in order to provide facilities for *ad hoc* querying. However, in order to model at the conceptual level, an object oriented schema definition language must itself provide facilities for describing the behaviour of data. This paper demonstrates that with only modest extensions, such a schema definition language may serve as a query notation. These extensions are concerned solely with supporting the interactive nature of *ad hoc* querying, providing facilities for naming and displaying query operations and their results.

1 Overview

Section 2 reviews the background to this work; its objectives are outlined in section 3. Section 4 describes NOODL constructs which are used to define behaviour within schemata, and section 5 examines how these may be extended for interactive use. The resulting query notation is evaluated in section 6. Section 7 outlines some further work and section 8 concludes.

2 Background

NOM (the Napier Object Model) is a simple data model intended to allow object oriented modelling of data at a conceptual level; it was first presented in [BK91] and is described fully in [Bar93]. NOM has been used to model [BK92a] and to support the implementation [BFK92] of novel database applications, and also for the investigation of specific modelling issues such as declarative integrity constraints and activeness [BK92b] and the incorporation of views [BK93] in object oriented data models.

The data definition and manipulation language NOODL (Napier Object Oriented Data Language) may be used to specify enterprises modelled using NOM. A NOODL schema contains a list of class definitions, which show the name and ancestors of each class.

A class definition also includes the names, sorts, and definitions of the properties of each class. Property definitions may contain simple expressions showing how the value of the property is derived from those of other properties. The ability to specify such derived properties allows the capture of data's behaviour within the enterprise schema. A class definition may also contain operations, constraints, and triggers, which are not discussed in detail here (see [BK91] or [Bar93] for more details).

NOM supports a principle of context substitutability, demanding that where an instance of some class is required, an instance of one of its descendant classes may always be substituted; this requires strict inheritance.

An example NOODL schema is shown in shown in figure 1; the syntax of NOODL is described in [Bar93].

The GNOME system (Generic Napier Object Model Environment) is an implementation of NOM in the persistent programming language Napier88 [DCBM89], [MBCD89]. Although the query language described here has been successfully implemented and may be used to interrogate objects managed by GNOME, this exposition will not address implementation issues.

2.1 NOM's Query Model

The construction of schemata requires an implicit query model for the evaluation of derived properties. Here, objects exchange messages, in response to which the query expressions encoded in the definitions of their properties are evaluated; these evaluations may result in the sending of further messages to further objects. This approach maintains a strong correspondence between real world and model objects.

Specifically, and unlike many other models, NOM does not allow the creation of new objects as the result of a query (although new *collections* of existing objects may be formed). In addition to adherence to the conceptual model, this prevents query results from being detached from the class lattice. This dispenses with the need to classify result-objects in the preexistent hierarchy (*eg*, [DD91], [Kim89]). This principle also provides *closure* in the query model; since queries may return only collections of preexisting objects, these may certainly be the targets of further, cascaded messages. Since no new objects are created, problems of comparing identifiers of objects returned by queries do not arise.

3 Objectives of This Work

A variety of object oriented database query languages exist; some are languages supporting the logical model provided by a particular OODB (*eg*, [MSOP86]); others are implementations of the query languages of semantic data models (*eg*, [KA87]); some are attempts to add object oriented extensions to SQL (*eg*, [ont90], [RS87]). These languages are intended to allow users to perform *ad hoc* queries on a database, to embed these in application programs, or both. However, Kim has argued that many of these languages are inadequate since they are not based on an underlying object oriented model of querying [Kim89], which takes into account the different abstraction mechanisms used in the schema definition against which querying is performed.

However, since an object oriented data model must capture the behaviour of the objects described, a query notation is necessary simply to *describe* an enterprise. The only alternatives are either to use natural language comments to describe the behaviour of data objects (losing precision), or to describe this behaviour in a host programming language (losing the conceptual level of the data description). As Zicari *et al* have pointed out, in object oriented database systems, the query language and the method definition language are seldom the same [Zic91]. Efforts are underway to establish a standard object oriented data description language [Atw93], [Hol93]; such a task would be facilitated by establishing a standard data manipulation language.

Since a data description language (DDL) must necessarily describe how objects interact with each other, it must contain a data manipulation language (DML); hence it is not possible to maintain two separate notations for these functions.

3.1 Integration of the Schema Definition Language and the Query Language

In the course of developing NOODL it became apparent that a schema definition language required considerable behavioural capture in order to specify derived properties, as well as constraints and triggers. When desire to support *ad hoc* querying in GNOME necessitated a query specification language, it was decided to develop this language by the minimal reasonable extension of NOODL. This approach seemed to offer the best integration of modelling and query notations, and would be easiest to implement since a compiler already existed for NOODL as a schema definition language.

Further, since NOM was designed as a 'vanilla' model, supporting only those features agreed upon by the majority of object oriented data models, it seemed that this approach might give some insight into the ability of any object oriented data model notation to support *ad hoc* querying, provided only that it allows specification of behavioural aspects of the model without resorting to use of a 3GL.

The facilities of NOODL as a schema definition language, and then its adaptation to *ad hoc* querying, are discussed in subsequent sections.

4 NOODL as a Schema Definition Language

This section shows how, in schema definitions, behavioural expressions may require to be written. The names of properties defined in NOODL schemata can be used as gettors and settors for those properties; this already provides a basis for a navigational query notation. In order to increase behavioural expressivity, it is necessary also to have constructs to handle collections of objects, and to build up more complex query expressions. These are described in the succeeding sections. The examples are based on the schema shown in figure 1.

4.1 Basic Query Expressions

Basic queries are of two types, here called navigational-style and search-style queries.

4.2 Navigational-Style Expressions

A navigational-style query finds some information which is implicit in the enterprise model. For example, consider the schema in figure 1 describing people with spouses, who own coloured cars (assuming one car per person). The property spousesCar-Colour returns the colour of any Person's spouse's car. If `LEIF` is an object of class

```
                  schema Example

domain Colour is ( "crimson", "aquamarine", "puce" )
domain Money is { defined elsewhere }

class Person                             class Employee
properties                               ISA Person
   { stored properties }                 properties
   name     : Text    ;;                    salary  : Money ;;
   city     : Text    ;;                    company : Company
   age      : Number  ;;
   spouse   : Person  ;;
   car      : Car     ;;
   children : #Person ;;
   { derived properties }
   grandchildren       : ##Person is
      self.children.children ;;
   spousesCarColour    : Text is
      self.spouse.car.colour ;;
   likeMindedCarOwners : #Person is
      Person where its.car.make = self.car.make


class Car                                 class Company
properties                                properties
   make   : Text    ;;                       name : Text ;;
   colour : Colour                           city : Text  { etc }

                  end_schema { Example }
```

Fig. 1. Example NOODL Schema

Person, then the expression `LEIF.spouse.car.colour` is a navigational-style query returning the colour of Leif's wife's car.

The tokens `spouse`, `car` and `colour` are messages which elicit the value of the property of the same name; the definition of the property, including whether the value is settable or derived, is found in the enterprise schema (strong encapsulation). The dot operator . sends the message on its right to the object returned by evaluating the expression on its left[1]. The general form of such a navigational style expression is:

```
<receiver>.<message_name>[.<message_name>]*
```

In a property definition appearing in an enterprise schema, `<receiver>` will usually be the reserved word `self`, which denotes the instance receiving the message for the

[1] Although statements involving expressions with one and more than one dot operator are respectively called 'simple' and 'complex' predicates by Kim [Kim89], there is no essential difference between these.

property being defined.

4.3 Search-Style Expressions

A search-style query finds all the objects in some collection which match a given selection criterion. This style of query is more usually associated with *ad hoc* querying of a database, but may reasonably be required also to allow definition of derived properties in an enterprise schema, where the value of that property is a set of instances meeting some condition.

For example, a property likeMindedCarOwners of class Person, which returns all Person instances owning the same kind of car as the instance in question, may be defined as follows:

```
Person where its.car.make = self.car.make
```

The general form of a search-style query expression is:

```
<collection> where <predicate>
```

In an enterprise schema, `<collection>` will usually be the name of a class in the schema, to whose full extent it is taken to refer. In `<predicate>` the reserved word `it` or `its` is taken to stand for an element of the collection, to allow formulation of the selection criterion, and the reserved word `self` refers to the instance of the class defined whose property is being evaluated.

Kim points out that the collection to which a query is posed may be either the direct extent of a class, or its full extent, including the instances of all descendant classes [Kim89]; Chan calls this issue *class qualification* [Cha92]. On account of the principle of context substitutability, the name of a class in a query expression in NOODL is always taken to refer to its full extent; however, at the cost of complicating the selection predicate, it is possible to define precisely which descendant classes are to be included in the search space, rather than just the two possibilities cited by Kim. For example, the following query will retrieve all persons called “Leif”, but omitting any who are instances of the class Manager. Instances of all classes descended from Person, either above or below Manager in the class lattice, are to be included:

```
Person where its.name  = "Leif"
         and its.class <> Manager
```

4.4 Treatment of Set-Valued Properties

The value of a set-valued property is a set of objects or of primitive values. The following operations are available on sets; their names are NOODL reserved words: `union, intersection, difference, add, remove, member, cardinality, element` (and `contains`, described below). The first three perform the appropriate binary set operation and return a new set. The next three add an element, remove an element, or test for its prior inclusion. The operation `element` returns a random element of the set (but always the same element for a given set instance); this allows access to the element of a singleton set, and also permits computational recursion over sets.

Where any other message is sent to a set, this message is mapped over all the set's elements (and, where appropriate, the set of responses returned). This allows a very transparent treatment of sets, and also allows queries to return nested collections which preserve the association structure of the enterprise model. So for example, the expression `LEIF.children` returns the set of Leif's children, and the expression `LEIF.children.name` returns the set of names of Leif's children. Given the family tree shown in figure 2, various expressions and the objects they return are shown in table 1.

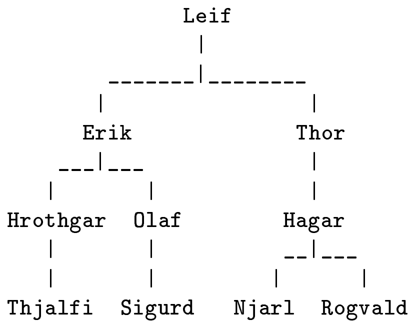

Fig. 2. Family Tree

expression	value
LEIF.children	{ERIK, THOR}
LEIF.children.children	{{HROTHGAR, OLAF}, {HAGAR}}
LEIF.children.children.children	{{{THJALFI}, {SIGURD}},{{NJARL, ROGVALD}}}

Table 1. Set-Valued Expressions from Family Tree

The `contains` operator determines whether a value is present anywhere within a nested collection. This allows the structure of the nested collections to be retained, unlike the 'flattening' operators provided in some object oriented query languages (*eg*, the `reunion` operator of LIFOO [BM81], or the `flatten` operator in ENCORE [SZ90]). If it *is* required to flatten a collection, this can still be done by using `contains` to select from the top level collection everything present in any nested subcollection. For example, {{HROTHGAR, OLAF}, {HAGAR}} can be flattened to {HROTHGAR, OLAF, HAGAR} by evaluating

```
Person where LEIF.children.children contains it
```

This expression picks out and returns all instances of class Person which are present (at any level) in the nested collection.

4.5 More Complex Expressions

More complex behavioural expressions may be built from sequences or compositions of simple query expressions; these may also contain conditional or update statements[2].

The target for a query is an object or collection of objects. Since the result of a query is also an object or collection of objects, the query model is *closed*; hence it is possible to compose, or nest, queries.

It has been proclaimed by Atkinson *et al* that the data model of an object oriented database system should be computationally complete [ABD+89]; if computational completeness is not to be relegated to a host programming language, it is necessary that is should be provided by the data language. For this purpose, NOODL has a conditional expression in the form of an `if` statement. Provision of a conditional expression, together with implicit iteration over sets and recursive function calls, allows the language to be considered computationally complete. Properties (and operations and triggers) may thus be defined which in principle perform arbitrary computations.

5 Adaptation to *Ad Hoc* Querying

Preceding sections have focused on the subset of NOODL dealing with what are traditionally considered data manipulation tasks. NOODL allows the construction of enterprise schemata where the behaviour of objects may be represented. In particular, it is possible to specify rules for the derivation of the values of properties.

NOODL allows the design of schemata for object oriented databases such as the GNOME system. However, one feature normally provided by a database management system is some form of *ad hoc* querying, allowing casual users to explore the data stored without resorting to writing programs. Persistent applications have included form-based [CMA87] or graphical [BFK92] interfaces, and also browsers for the persistent store [KD90]. Here, some simple extensions to NOODL are presented which allow it to be used as an *ad hoc* query language. Unlike forms and some graphical interfaces, NOODL is not application specific when used as a querying mechanism; it may be used for any enterprise describable within the object oriented modelling approach adopted here. Further, the data store is interrogated at the conceptual level of the enterprise model, rather than at the level of programming language constructs.

The following sections investigate what extensions to NOODL would be necessary in order for it to function as an *ad hoc* querying language.

5.1 Requirements for *Ad Hoc* Querying

Used as a tool for data description, NOODL provides facilities for retrieving and updating values or sets of values, and for selecting objects satisfying some criterion. Only modest extensions to these facilities are required for *ad hoc* querying.

[2] The semantics of *mixed* query and update statements may be problematical as discussed by Ghelli *et al* in [GOPT92].

Firstly, since there is no notion of 'display' in data definition, some means of showing what the user wants to know must be provided.

An enterprise schema is a complete conceptual description of an enterprise, serving as a 'definitive text' for applications serving the enterprise. *Ad hoc* querying, on the other hand, is an iterative, explorative procedure; facilities are required to incrementalise the construction of queries, and allow a feedback loop between formulating a query and seeing the result (query-building). The syntax of a query language should support these activities as naturally as possible.

Such support is provided by local names, which may be assigned to the (collections of) objects returned by a query, or to a query itself. Together with the display operation, these provide a sufficient basis for *ad hoc* querying. Displaying data and managing local names and definitions are facilities available during a query session; the NOODL constructs supporting these activities are not used within an enterprise schema.

Such locally defined query expressions may be viewed as 'behaviour constructors' ([YO91]); they may also be used to express join-like queries, retrieving together information not related through the relationships encoded in the enterprise schema ('coincidences in the data').

The following sections describe extensions to NOODL which support the necessary functionality; some examples of *ad hoc* queries executed using this system may be found in the appendix.

5.2 Conceptual *Ad Hoc* Query Model

A query model is implicit in the specification of derived properties. The concept of a *querier* extends this model to *ad hoc* use.[3] The querier is an object which straddles the interface between the real world and the data space. Its user interface allows a user to construct NOODL queries, which are syntax- and sort-checked, and could in principle be optimised; the corresponding sequence of messages are then sent into the data space, and a new collection of objects constructed in response to these. The querier then represents these objects to its human user in some intelligible format.

5.3 Local Definitions — Tags

The two approaches to querying supported are navigational-style and search-style queries. In constructing complex queries (especially nested queries), it is often useful to break the query down into distinct components. The user should be allowed to construct queries incrementally, rather than being forced to resubmit an entire query when only a component subquery requires modification.

To support this approach, the querier allows local definitions. Here, a local name, called a *tag*, is introduced to refer to an intermediate query result. This tag is known only within the query session, not within the enterprise schema. As a convention, such tags are written in upper case.

Earlier examples used the token `LEIF` to refer to the object representing the person Leif; this name is defined to the querier as a tag as shown, and prevents the need to embed the expression locating the object LEIF in queries which refer to it.

[3] GNOME's query facility is an implementation of this querier.

```
Person where its.name = "Leif" ;;
tag LEIF result.element
```

Here, `element` removes the object LEIF from the singleton set which is returned by the query; `result` is a NOODL reserved word providing a tag always bound to the result of the most recently executed query.

Tags are more useful where entire collections are retrieved, examined interactively to ensure that they do indeed contain the correct objects, and then used as building blocks for the construction of other more complex queries. The general form of a tag definition is:

```
tag <tag_name> <query_expression>
```

The concept of a local definition is further developed in a succeeding section on query methods, which are locally defined query expressions.

5.4 Seeing the Result

So far, queries which return (collections of) objects have been discussed, with no indication of how the result may be displayed. Here, the relational model has a clear advantage, since relations correspond closely to the concept of a table; tuples retrieved from relations may be displayed as tables, the relational `project` operator (*ie*, the SQL command `select`) being used to customise the contents of these tables.

The objects returned by NOODL queries are collections of objects, possibly with properties which are other objects, and possibly with properties which are collections. It is harder to display these as tables, since neither complex objects nor collections conveniently fit into a single slot. A further problem is that although a certain collection of objects is returned as a query result, we may wish to display information relating to several classes of object (*eg*, 'show the name and salary of all IBM employees, and the make and colour of the car they drive').

These difficulties constitute an under-estimated difficulty in arriving at a clear conceptual query model for object oriented data. This section demonstrates how the NOODL `show` command, (available in interactive mode only), can be used for the tabular visualisation of query results (including nested tabulation); object-valued properties and set-valued properties will be discussed, together with display of information from different but navigationally-linked classes; display of information from classes unrelated in the application schema will be treated in the following section.

The `show` Command The `show` command displays the requested information for each of a collection of objects, with the option of attaching textual headings to the resultant table. Tabularisation is automatically provided. The information requested may be the value of any property of the object, or of any other object to which it is navigationally linked. The example below shows the name and salary of all IBM employees, together with the make and colour of their car.

```
Employee where its.company.name = "IBM" ;;
show "name" result.name, "earns" result.salary,
     "car" result.car.make, "car colour" result.car.colour
```

```
name            earns   car     car colour
=============   =====   =====   ==========
Leif Svensson | 20000 | Volvo | crimson
Erik Leifsson | 15000 | Volvo | aquamarine
```

Where the value of a property is a collection, each value of the collection is shown nested in the table; nesting may in principle be to any depth. The query below shows all people together with their children and grandchildren.

```
show "person" Person.name,
     "children" Person.children.name,
     "grandchildren" Person.children.children.name

person          children            grandchildren
=============   =================   ====================
Leif Svensson | Erik Leifsson     | Hrothgar Eriksson
                                  | Olaf Eriksson
              | Thor Leifsson     | Hagar Thorsson
Erik Leifsson | Hrothgar Eriksson | Thjalfi Hrothgarsson
              | Olaf Eriksson     | Sigurd Olafsson
Thor Leifsson | Hagar Thorsson    | Njarl Hagarsson
                                  | Rogvald Hagarsson

- - - - - - - - - - - - etc - - - - - - - - - - - - -
```

An attempt to `show` the value of an object-valued (*ie*, non-lexical) property will show its object identifier as shown below.

```
Person where its.name = "Leif" ;;
tag LEIF result.element        ;;
show "Leif's wife" LEIF.spouse

Leif's wife
===========
o# 26473

Object where its.oid = 26473 ;;
show "Leif's wife's name" result.name

Leif's wife's name
========================
Freyja  Thorgrimssdottir
```

5.5 Local Definitions — Query Methods

Query methods are behaviour constructors, which locally name some query in the same way that tags locally name some object or collection of objects. Query methods are introduced by the NOODL reserved word `defun`, and after definition may be used within a query session as if they were properties defined for the relevant class in the enterprise schema.

Query methods may be used to incrementalise the construction of complex queries, and also to represent 'join-like' queries.

The join operator is highly used in the relational model. Often, the need for a join arises directly from the limitations of this model; since all attributes must be primitive, a query such as the example above must be expressed as something like:

```
SELECT PERSON.NAME, PERSON.WAGE, CAR.MAKE, CAR.COLOUR
FROM PERSON, CAR, COMPANY
WHERE PERSON.CAR_REG = CAR.REG_NO
  AND PERSON.COMPANY_NO = COMPANY.COMPANY_NO
  AND COMPANY.NAME = 'IBM'
```

Here, the joins really encode the navigational link between person, car, and company, inherent in the conceptual schema; this kind of join is never necessary in NOM since object-valued properties are permitted.

Sometimes, however, a join will be used in the relational model to search for a 'coincidence' in the data — some relationship not directly expressed in the enterprise schema. The need to retain the ability to express such queries in object oriented models has been pointed out by Shaw and Zdonik [SZ90], and by Yu and Osborn [YO91].

Consider a query to show the name of all companies located in the same city as each person, based on the schema in figure 1; this requires a join-like search. The query can be achieved by defining a query method[4] `same_city` which returns the colocated companies of its person argument as follows:

```
defun same_city ( Person ) : # Company is
  Company where its.city = self.city
```

This defines a query method which traverses the extent of class Company and returns all those instances whose city property matches that of its argument, an instance of class Person; this method may be applied to an instance of class Person as if it were a property defined in its schema. The class on which the query method is defined is shown in parentheses after its name; it is to an instance of this class (or its descendants) that the reserved variable `self` refers when it appears in the following definition.

In order to show the colocated cities of each person, `same_city` is mapped over the extent of class Person:

```
show "the person" Person.name,
     "colocated companies" Person.same_city.name
```

Further, the example shows that this approach may also be used to express queries which return properties of more than one object, without the need to create new objects or classes at query time.

[4] The syntax for the definition of query methods is similar to that for the definition of *operations* in NOODL schemata.

5.6 Treatment of Null Values

Zicari *et al* have pointed out that few object oriented query models have attempted a treatment of null values [Zic91]. Although problematical in the relational model, it is possible that approaches to nulls based on the criterion of identity (*eg*, that of Larner [Lar91]) may be well suited to an object oriented model where identity is a central concept ([KC86]). This is a topic for further research.

The current version of NOM, however, takes a simple, pragmatic approach to the treatment of null values. In the same way that a class Object is provided as the most general class (used to hold those definitions common to all classes) the class Bottom is provided as the most specific class. From the principle of context substitutability, this means that an instance `bottom` of class Bottom can appear in the context of an instance of any other class. `bottom` is a NOODL reserved word referring to such a (newly-generated) instance of class Bottom.

All the properties inherited by class Bottom are overridden by definitions which always return that same instance of class Bottom in response to any object-valued message, or suitable fail-values in response to any primitive-valued message. In this way, an instance `bottom` can cascade through a query expression of arbitrary depth.

For example, if evaluation of the expression `LEIF.spouse.car.colour` should fail because Leif has no wife, the following evaluation sequence will occur:

```
LEIF.spouse.car.colour
  -> bottom.car.colour
         -> bottom.colour
                -> ""
```

This pragmatic approach has desirable properties, described in [Bar93]. Particularly, the use of `bottom` as a null value fits into the conceptual framework of NOM, and the presence of a `bottom` cannot cause a message expression to fail to be evaluated; `bottom` may also be used to represent object deletion. Expressions which may involve `bottom` may be statically sort-checked ensuring semantic consistency. The approach has been implemented, and could also be extended to gather debugging information automatically for cases where unexpected null values are encountered.

6 An Evaluation of Query Models

Yu and Osborn [YO91] have evaluated the query models proposed by Manola and Dayal [MD86], Osborn [Osb88], Straube and Öszu [SO91], and Shaw and Zdonik [Sha87]. Table 2 is a summary of their results, together with an evaluation of NOODL (as a query notation) in the same framework. A 'Y' means that the criterion is met, a 'N' that it is not met, a dot that it is partially met, and a question mark that it is not clear from any available documentation whether it is met.

The criteria of the evaluation are explained in detail in [YO91]. Yu and Osborn state that the features checked in their evaluation, although at times somewhat mutually incompatible, are generally desirable. It is interesting to notice, that although not originally designed for *ad hoc* querying, NOODL compares favourably within this framework.

	Manola + Dayal	Osborn	Straube + Ozsu	Shaw + Zdonik	NOODL
OBJECT ORIENTEDNESS					
object identities	.	.	.	Y	.
encapsulation	Y	N	Y	Y	Y
inheritance	N	N	Y	N	Y
polymorphism	N	.	N	N	Y
classes + collections	N	Y	N	Y	Y
heterogeneous sets	?	Y	Y	Y	Y
EXPRESSIVENESS					
extends rel. alg.	Y	Y	N	Y	.
object constructors	.	Y	N	N	N
invocable behaviours	Y	N	Y	Y	Y
behaviour constructors	N	N	.	N	Y
dynamic type creation	Y	Y	N	Y	N
querying closures	Y	N	N	N	Y
behaviours as objects	N	N	N	N	N
FORMALNESS					
formal semantics	N	N	Y	.	N
closed	N	N	Y	Y	Y
PERFORMANCE					
strong typing	Y	N	Y	Y	Y
optimisable	Y	Y	Y	Y	Y
DATABASE ISSUES					
object lifetimes	N	.	N	?	.
schema evolution	N	.	N	N	Y
calculus	N	N	.	N	N

Table 2. Summary of Yu and Osborn's Evaluation (NOODL added)

7 Further Work

The version of NOODL described here has been used for investigation of various issues in object oriented data modelling. However, for practical use it requires some extensions, whether used for data definition (incorporation of query expressions in schemata), for embedding in programs, or for *ad hoc* querying. These extensions include a wider range of primitive sorts (including graphics sorts), a wider range of collection types, aggregate functions and collection literals. It is planned to pursue some case studies in the use of NOODL, and to experiment with, and perhaps automate, the mapping of NOODL onto the query languages provided by some proprietary object oriented database systems.

It is planned to map the querying constructs of NOODL onto a formal model such as list [Tri90] or object [CT94] comprehensions; this will enable the application of appropriate logical optimisation techniques (*eg*, [TW90], [JG91]). Addition of indexing structures also remains to be undertaken.

The tabular visualisation mechanism provided by the current version of NOODL is intended as a basic, minimum facility for the presentation of data. Work is ongoing to develop graphical tools which will integrate querying with schema management, and provide more comprehensive forms of data visualisation.

8 Conclusion

The use of NOODL to express queries over object oriented data arose, not from the intention to design a language for *ad hoc* querying, but from the recognition that if a data description language is to capture the behavioural aspects of the data, it must be capable of expressing data manipulation.

The notation presented has been based on a conceptual query model, which the authors believe is simple and natural. Provision of such a 'vanilla' query model, based on the features essential to the object oriented paradigm, makes it possible to construct enterprise schemata without losing the conceptual level by embedding in a host programming language.

Addition of some modest features have extended the notation sufficiently to allow construction of a wide class of *ad hoc* queries, including some that the query languages of many prototype OODBMSs are unable to express (see [BK93] and [CHT93]). The `show` command allows a tabular visualisation of object oriented data. Local definitions support the incremental construction of complex queries, and query functions support the incremental construction of join-like subqueries without the need to create new object identities. These added features do not extend the semantics of the notation; rather, they simply make it more convenient for interactive use.

NOODL provides simple but powerful constructs for querying. Although it has distinctive aspects, such as the transparent treatment of set-valued properties, it is proposed not so much as a novel query language, but rather as a demonstration of the integration of schema definition and querying notations.

When an *ad hoc* query notation is supported, these queries should be expressed in the same notation as the conceptual model. Failing to do this has two undesirable consequences: firstly, the number of notations the user must master may increase to as many as three; and secondly, cognitive dissonance may arise if the conceptual model, which is likely to be held up as a reference during the construction of queries, is expressed differently.

Appendix — Query Examples

This section presents two more substantial queries expressed in NOODL. (A more complete set of example queries, adapted from those used by Gray *et al* in [GKP92, chapter 2], may be found in [Bar93]). These examples are adapted from queries used in [CHT93], and refer to the following NOODL schema, adapted from Chan's paper:

```
                  schema Chans_Examples

class Company                     class Financier
properties
   name   : Text       ;;         class LoanGivingDealer
   models : # CarModel            ISA Dealer, Financier

class CarModel                    class Address
properties                        properties
   facilities : # Text   ;;          street   : Text ;;
   dealers    : # Dealer             postcode : Text ;;
                                     city     : Text

class Dealer                      class Garage
properties                        properties
   name    : Text    ;;              name    : Text ;;
   address : Address                 address : Address

                  end_schema { Chans_Examples }
```

(Query 1) Find names and prices of all non-Ford models that have an insurance group lower than 5; the answer should include only hatchbacks and saloons with radio and cassette player:

```
defun non_ford ( CarModel ) :  Boolean is   { determine whether this }
      Company where its.name <> 'Ford'  ;   { model is available     }
      result.models contains self       ;;  { from non Ford dealer   }

Carmodel where its.carType = 'Saloon'
            or its.carType = 'Hatchback'
           and its.facilities contains 'tapeplayer'
           and its.facilities contains 'radio'
           and its.non_ford             ;;

show result.name, result.price          ;;
```

(Query 2) Show names of Ford models and Ford dealers that provide loan as a financial option for car purchase; the answer should include only dealers located in a city where there are at least two garages:

```
{ show number of garages in city where a dealer is located }

defun num_garages ( Dealer ) : Number is
      Garage where its.address.city = self.address.city     ;
      result.cardinality                                   ;;

defun ideal_dealer ( CarModel ) is
      LoanGivingDealer where self.dealers contains it
                    and its.num_garages >= 2              ;
      show result.name, result.address                   ;;


tag ford_companies Company where its.name = "Ford"       ;;
CarModel where ford_companies.models contains it         ;;
show result.name, result.ideal_dealer
```

References

[ABD+89] M Atkinson, F Bancilhon, D DeWitt, K Dittrich, D Maier, and S Zdonik. The Object Oriented Database System Manifesto: (a Political Pamphlet). In *Proceedings of DOOD*, Kyoto, Dec 1989.

[Atw93] Thomas Atwood. The Object DBMS Standard. *Object Magazine*, pages 37 – 44, September-October 1993.

[Bar93] Peter J Barclay. *Object Oriented Modelling of Complex Data with Automatic Generation of a Persistent Representation.* PhD thesis, Napier University, Edinburgh, 1993.

[BFK92] Peter J Barclay, Colin M Fraser, and Jessie B Kennedy. Using a Persistent System to Construct a Customised Interface to an Ecological Database. In Richard Cooper, editor, *Proceedings of the 1st International Workshop on Interfaces to Database Systems*, pages 225 – 243, Glasgow, 1992. Springer Verlag.

[BK91] Peter J Barclay and Jessie B Kennedy. Regaining the Conceptual Level in Object Oriented Data Modelling. In *Aspects of Databases (Proceedings of BNCOD-9)*, pages 269 – 305, Wolverhampton, Jun 1991. Butterworths.

[BK92a] Peter J Barclay and Jessie B Kennedy. Modelling Ecological Data. In *Proceedings of the 6th International Working Conference on Scientific and Statistical Database Management*, pages 77 – 93, Ascona, Switzerland, Jun 1992. Eidgenössische Technische Hochschule, Zurich.

[BK92b] Peter J Barclay and Jessie B Kennedy. Semantic Integrity for Persistent Objects. *Information and Software Technology*, 34(8):533 – 541, August 1992.

[BK93] Peter J Barclay and Jessie B Kennedy. Viewing Objects. In *Advances in Databases (Proceedings of BNCOD-11)*, pages 93 – 110. Springer Verlag (Lecture Notes in Computer Science Series), 1993.

[BM81] O Boucelma and JL Maitre. Querying Complex-Object Databases. Internal report, University of Marseilles, 1981.

[Cha92] Daniel Chan. A Survey of Object Oriented Database Query Languages. Internal report, University of Glasgow, Feb 1992.

[CHT93] Daniel KC Chan, David J Harper, and Philip W Trinder. A Case Study of Object Oriented Query Languages. In *Proceedings of the International Conference on Information Systems and the Management of Data*, pages 63 – 86. Indian National Scientific Documentation Centre (INSDOC), 1993.

[CMA87] RL Cooper, DK MacFarlane, and S Ahmed. User Interface Tools in PS-algol. Technical report PPRR-56-87, Universities of Glasgow and St Andrews, Mar 1987.

[CT94] Daniel KC Chan and Philip W Trinder. Object Comprehensions: A Query Notation for Object-Oriented Databases. In *Proceedings of BNCOD-12*, Guildford, Surrey, 1994. Springer Verlag.

[DCBM89] Alan Dearle, Richard Connor, Fred Brown, and Ron Morrison. Napier88 - A Database Programming Language? In *Proceedings of DBPL-2*, pages 213 – 230, 1989.

[DD91] KC Davis and LML Delcambre. Foundations for Object Oriented Query Processing. *Computer Standards and Interfaces*, 13:207 – 212, 1991.

[GKP92] Peter MD Gray, Krishnarao G Kulkarni, and Norman W Paton. *Object Oriented Databases: A Semantic Data Model Approach*. Prentice Hall, 1992.

[GOPT92] Giorgio Ghelli, Renzo Orsini, Alvaro Pereira Paz, and Phil Trinder. Design of an Integrated Query and Manipulation Notation for Database Languages. Technical report FIDE/92/41, Universities of Pisa, Salerno, Glasgow and Sviluppo Research Laboratory, 1992.

[Hol93] Glenn Hollowell. *Handbook of Object Oriented Standards: the Object Model*. Addison Wesley, 1993.

[JG91] Zhuoan Jiao and Peter MD Gray. Optimisation of Methods in a Navigational Query Language. In *Proceedings of DOOD-2*, pages 22 – 41, 1991.

[KA87] KG Kulkarni and MP Atkinson. Implementing an Extended Functional Data Model in PS-algol. *Software Practice and Experience*, 17(3):171 – 185, 1987.

[KC86] S Khoshafian and GC Copeland. Object Identity. In Norman Meyrowitz, editor, *Proceedings of OOPSLA*, pages 406 – 416, Portland, Oregon, September 1986.

[KD90] Graham Kirby and Alan Dearle. An Adaptive Graphical Browser for Napier88. Technical report, University of St Andrews, 1990.

[Kim89] Won Kim. A Model of Queries for Object Oriented Databases. In Peter MG Aspers and Gio Wiederhold, editors, *Proceedings of VLDB*, pages 423 – 431, Amsterdam, 1989. Morgan Kaufmann.

[Lar91] Adrian Larner. On Nulls. Internal report, IBM, Warwick, 1991.

[MBCD89] R Morrison, F Brown, R Connor, and A Dearle. The Napier88 Reference Manual. Technical report PPRR-77-89, Universities of Glasgow and St Andrews, Jul 1989.

[MD86] F Manola and U Dayal. PDM: an Object Oriented Data Model. In *Proceedings of the International Workshop on Object Oriented Database Systems*, pages 18 – 25, Sep 1986.

[MSOP86] D Maier, DJ Stein, A Otis, and A Purdy. Development of an Object Oriented DBMS. In *Proceedings of OOPLSA*, pages 472 – 482, 1986.

[ont90] ONTOS SQL User's Guide. *(ONTOS documentation)*, Dec 1990.

[Osb88] SL Osborn. Identity, Equality, and Query Optimisation. In KR Dittrich, editor, *Advances in Object Oriented Database Systems (Proceedings of the 2nd International Workshop on Object Oriented Database Systems)*, pages 346 – 354. Springer Verlag, Sep 1988.

[RS87] LA Rowe and MR Stonebraker. The POSTGRES Data Model. In *Proceedings of VLDB-13*, pages 83 – 96, Brighton, Sep 1987.

[Sha87] GM Shaw. An Object Oriented Query Algebra. *Bulletin of the IEEE Technical Committee on Data Engineering*, 12(3):29 – 36, Sep 1987.

[SO91] DD Straube and MT Özsu. Queries and Query Processing in Object Oriented Database Systems. *ACM Transactions on Information Systems*, pages 387 – 430, 1991.

[SZ90] GM Shaw and SB Zdonik. A Query Algebra for Object Oriented Databases. In *Proceedings of the 6th International Conference on Data Engineering*, pages 154 – 162. IEEE Computer Society Press, 1990.

[Tri90] Phil Trinder. Comprehensions, a Query Notation for DBPLs. Technical report CSC90/R16, University of Glasgow, 1990.

[TW90] Phil Trinder and Philip Wadler. Improving List Comprehension Database Queries. Technical report CSC90/R4, University of Glasgow, 1990.

[YO91] L Yu and SL Osborn. An Evaluation Framework for Algebraic Object Oriented Query Models. In *Proceedings of the 7th International Conference on Data Engineering*, pages 670 – 677. IEEE Computer Society Press, 1991.

[Zic91] Roberto Zicari. A Framework for Schema Updates in an Object Oriented Database System. In *Proceedings of the 7th International Conference on Data Engineering*, pages 2 – 13. IEEE Computer Society Press, 1991.